實戰

Ext JS: The Definitive Guide

Ext JS 权威指南

黄灯桥 著

机械工业出版社
China Machine Press

本书有两大特点：第一，授人以鱼，根据 Ext JS 的构成体系全面系统地讲解了其开发方法与技巧，每个知识点都辅之以翔实的案例，同时包含大量最佳实践，适合系统学习和开发参考；第二，授人以渔，宏观上对 Ext JS 的整体架构进行了分析，微观上则通过源代码深刻揭示了 Ext JS 的工作机制与原理，对于想了解 Ext JS 工作原理和在开发中碰到疑难问题的读者尤为有帮助。

全书一共 22 章：第 1 章简要介绍了学习 Ext JS 必备的基础知识、JSON、Ext JS 4 的新特性，以及其开发工具的获取、安装与配置；第 2 章介绍了 Ext JS 4 的获取、Ext JS 库的配置与使用、语法、本地化，以及一个经典的入门示例；第 3 章详细讲解了调试的工具及技巧，这是本书的重要内容，希望所有 Web 开发者都能掌握；第 4 章全面介绍了 Ext JS 的基础架构；第 5~9 章分别讲解了 Ext JS 的事件及其应用、选择器与 DOM 操作、数据交互、模板与组件、容器、面板、布局与视图；第 10 章和第 11 章分别详细介绍了重构后的 Gird 和与 Gird 同源的树；第 12~16 章分别讲解了表单、窗口、按钮、菜单、工具条、图形、图表，以及其他组件和实用功能；第 17~19 章分别介绍了 Ext.Direct、动画功能和拖放功能；第 20~22 章则分别讲解了扩展与插件、主题开发、MVC 应用的架构。

图书在版编目（CIP）数据

Ext JS 权威指南 / 黄灯桥著 . —北京：机械工业出版社，2012.6

ISBN 978-7-111-38063-4

I. E…　II. 黄…　III. Java 语言 – 程序设计　IV. TP312

中国版本图书馆 CIP 数据核字（2012）第 071222 号

机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码　100037）
责任编辑：佘　洁
北京市荣盛彩色印刷有限公司印刷
2012 年 6 月第 1 版第 1 次印刷
186mm×240mm • 62.75 印张
标准书号：ISBN 978-7-111-38063-4
定价：119.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991；88361066
购书热线：（010）68326294；88379649；68995259
投稿热线：（010）88379604
读者信箱：hzjsj@hzbook.com

前　言

近一段时间，随着 HTML 5 和 CSS 3 的逐步升温，浏览器大战进行得如火如荼，而开发人员最关心的是 JavaScript 框架在 HTML 5 和 CSS 3 的大潮下会走向何方。

随着 iPhone 4S 的出现，手机大战也进行得如火如荼。目前的情况是，不但手机系统进行着混战，手机应用的框架也在进行混战。

应该说 Ext 公司在这方面触觉是很敏锐的，它选择合并，将公司更改为 Sencha，并通过 Sencha 加入到手机框架大战中。本以为 Ext JS 会止步于 Ext JS 3.3，想不到 Sencha 在手机框架中站稳脚跟后，2011 年又推出了 Ext JS 4.0，对这个颇受欢迎的框架进行了更多的革新，加入了很多 HTML 5 和 CSS 3 的元素。

在过渡到 HTML 5 和 CSS 3 的时期，Ext JS 框架能继续发展，对开发人员来说是天大的喜事。因为在 HTML 5 和 CSS 3 还没完全一统江湖的时候，开发人员要继续使用目前的开发工具进行开发。而 Ext JS 3 暴露出来的问题也需要不断修正和更新，这正是 Ext JS 4 需要解决的。

很不错，Ext JS 4 终于来了。

为什么写这本书

从 Ext JS 2.0 开始，尤其是 Ext JS 3.0 推出后，使用 Ext JS 的开发人员越来越多。而随着 Ext JS 4 的推出，估计会掀起一个新的技术学习热潮。Ext JS 4 进行了架构的调整，做了一系列的规范化，例如类名的规范化、UI 组件的渲染的规范化等，还重构了不少 UI 组件，这些对熟悉 Ext JS 3 的开发人员来说都要重新学习。对于新入门的读者，更是急需 Ext JS 4 方面的书籍来指导。因此，笔者决定写一本介绍 Ext JS 4 的书。

目前市面上介绍 Ext JS 的图书基本都是以应用为主，深入介绍 Ext JS 的很少，这也是笔者写本书的一个缘由。在本书中，不但介绍了如何使用 Ext JS 4，还深入 Ext JS 4 的源代码，通过抽丝剥茧的分析，让大家对 Ext JS 4 有更深入的理解，从而使开发人员在使用 Ext JS 4 时少走弯路。

本书除了深入介绍 Ext JS 4 外，还有一个主要目的，就是希望大家能掌握 Firebug 这个工具，学会如何使用 Firebug 去分析 JavaScript 代码的运行过程，提高分析能力及调试水平，

减少 bug 的发生和调试时间。

本书的特色

笔者在 Ext JS 的学习和交流中发现，很多使用者对 Ext JS 的整体架构不太熟悉，造成对应用中的一些问题混淆不清，从而导致应用中出现或多或少的错误。例如，对 Ext JS 的三层架构不熟悉，造成在 UI 中找数据的问题。而目前各类 Ext JS 书籍中很少提及这些方面，所以，针对使用者普遍存在的问题和容易混淆的地方，本书通过原理讲解和示例讲述“为什么是这样的”以及“为什么要这样做”。

本书的主要特色就是通过源代码的讲解，让读者明白 Ext JS 的类是如何运作的，然后通过运行原理及实战来学习如何使用这些类，并循序渐进地学习 Ext JS。

本书面向的读者

本书介绍了 Ext JS 4 中的几乎所有功能，并对其源代码进行了分析和讲解。因此，读者不但可以学习如何使用 Ext JS 4，还可以通过阅读源代码加深对 Ext JS 的理解，并从源代码中吸取别人的经验，提高自己的脚本编写能力。只要读者具备 Web 编程的基础，阅读本书的内容就不会有什么问题。

如果读者是新手，就应根据第 1 章关于学习 Ext JS 必需的基础知识一节的建议，去阅读相应的书籍，掌握 Web 编程的基础知识，尤其是 JavaScript、HTML 和 CSS 的相关知识。当然，能了解面向对象、三层架构等知识对阅读本书是相当有帮助的。

如果读者对 Ext JS 有一定基础，则可通过本书深入了解 Ext JS 的运行原理，加深对 Ext JS 的认识，尤其是对 Ext JS 新的开发框架的认识。

本书示例中的服务器端代码，使用 C# 和 Java 两种语言编写，所以只要熟悉这两种语言中的一种，对阅读本书都会有很大帮助。不过，Ext JS 是一个基于客户端的 JavaScript 框架，对于使用何种服务器端开发语言没有要求，因此即使是不懂 C# 和 Java 语言的读者，阅读本书也不会有太多困难。

使用本书的要求

在使用本书的示例时，最好安装 Firefox 4.0 以上版本，并且安装 Firebug 扩展。

要运行本书中带有服务器端代码的示例，需要：

- IIS 或 Tomcat 7.0
- .Net：Visual Studio 2010
- Java：Eclipse Helios Service Release 2 或 Spket 1.6.18

- SQL Server 2005

本书资源包请登录华章网站（www.hzbook.com）下载。

如何阅读本书

本书是采用循序渐进的方式来介绍 Ext JS 4 的，各章之间会有一定的关联，因此建议读者按章节的编排顺序逐章阅读。本书中有些示例是在 Firebug 中示范的，所以最好在自己的机器上安装好 Firefox 和 Firebug，以备使用。这样做的目的是让大家在阅读过程中熟悉 Firebug 的一些调试技巧，这些技巧在调试 Ext JS 时是相当有用的。例如，有初学者曾咨询为什么在 Grid 中显示不了数据，查了很久都查不到原因，让笔者帮忙看一下代码，笔者让他先用 Firebug 查看 Store 是否已提交数据请求，在 Firebug 网络面板中，他看到请求已发送，但是返回的是服务器端代码错误，最后查到仅仅是“将公有方法设置成私有方法”这样简单的修正。在很多时候，就是这样的错误造成大麻烦，所以笔者建议多使用 Firebug，好的工具可以事半功倍。

本书秉承实践是最好的老师的精神，立足于“自己动手，丰衣足食”的原则，因而希望读者在阅读本书的时候，能亲自动手去实践一下。Let's do it!

本书约定

- 类名：本书中大量使用了简写的类名，在没有特殊说明的地方，简写类名就是指 Ext JS 源代码中的某个类。在附录中可查到各简写对应的类名。
- 粗体文字：粗体文字表示在学习和使用中需要重点记忆或注意的内容。
- 粗体代码：粗体代码表示代码中的关键代码。例如：

```
var r = [], ri = -1,
    m = nthRe.exec(a == "even" && "2n" || a == "odd" && "2n+1" || !nthRe2.
        test(a) && "n+" + a || a),
    f = (m[1] || 1) - 0, l = m[2] - 0;
```

联系作者

希望本书能给每位读者带来收益。如果对本书有任何意见和建议，或者有任何技术上的问题，请与笔者联系。笔者非常希望收到大家的建议以便提高创作水平，也非常乐意与大家一起探讨和分享有关 Ext JS 开发的问题，甚至是更广泛的 Web 开发问题。如果想联系笔者，请发邮件到 huangdengqiao@yahoo.cn，或者通过微博 http://weibo.com/gerneal 互动。

致谢

在本书的出版过程中，得到了机械工业出版社华章公司的编辑的大力支持，尤其是杨福川编辑。正是由于他们的辛勤劳动，才使笔者能够顺利完成本书的创作，笔者向他们表示衷心的感谢！

在此，笔者也对网络上有名或无名的技术英雄表示衷心的感谢！若没有他们，笔者在工作和学习中碰到的困难会很难解决。有了他们，很多问题都迎刃而解了。

目　录

第 1 章　Ext JS 4 开发入门

在这一章，我们将介绍一些与 Ext JS 有关的基础知识以及 Ext JS 4 的一些主要变化。从 2008 年接触 Ext JS 到现在，不少初学者咨询过我，应该怎样才能学好 Ext JS？这是一个很好的问题，每个人的学习方法不同，在学习 Ext JS 之前的基础都不同，因而如何学好 Ext JS 因人而异。不过 Ext JS 始终是一个 JavaScript 的框架，有其局限性，掌握了 Javascript、HTML 和 CSS 等知识，再加上掌握 Ext JS 的框架结构，就足够应付 Ext JS 的学习了。如果再有点面向对象编程的知识，那就如虎添翼了。下面让我们开始学习 Ext JS 之旅。

1.1　学习 Ext JS 必需的基础知识

1. JavaScript

嗯，这个还用说吗？ Ext JS 本来就是一个 JavaScript 的框架，而且使用 Ext JS 就需要使用 JavaScript 语法来开发，需要 JavaScript 的知识是必然的了。问题的关键是，开发人员对 JavaScript 知识的掌握也有深浅之分。譬如，我碰到一些开发人员，对 JavaScript 算是很熟悉了，但是不会 JSON，不会直接使用 JSON 对象，在使用 Ext JS 的过程中，需要使用 JSON 对象的时候，居然是通过组装字符串的方式，然后使用 eval 方法将其转换为对象来使用的。这就是因为 JavaScript 学习深度不足造成的。

那么，到底 JavaScript 要学到多深才能学好 Ext JS 呢？这个问题也不太好回答。不过以笔者的经验来说，建议仔细认真地看一次《JavaScript 权威指南（第 5 版）》[⊖]和《JavaScript 高级程序设计（第 2 版）》这两本书，重点关注和理解以下内容：

- ❑ 函数。
- ❑ 变量。
- ❑ 作用域（scope）。
- ❑ 原型模式（prototype）。
- ❑ 闭包。
- ❑ 文档对象模型 DOM。
- ❑ Document 对象。
- ❑ 动态 HTML 事件以及事件处理。
- ❑ Ajax。
- ❑ JSON。

以上这些知识对于了解和使用 Ext JS 非常有帮助。因 JSON 涉及服务器端的处理，所以

⊖ 《JavaScript 权威指南（第 5 版）》已由机械工业出版社引进出版。书号：9787111216322。——编辑注

本书会在本章 1.2 节讲解。

2. HTML

HTML 知识，这个对于 Web 开发人员来说应该是很熟悉的东西。这里主要就是需要熟悉 HTML 的各种标记。

3. CSS

因为 CSS 知识的缺乏，很多开发人员在使用 Ext JS 出现显示问题时不知道如何调试，不知道如何去找错误。而这对学习理解 Ext JS 也造成了障碍。事实上，全部的 JavaScript 框架，就是通过脚本的方式生成页面元素与 CSS，通过控制这些元素和 CSS，实现需要的功能的，因而，掌握好 CSS 知识，对学习 Ext JS 非常重要，这要引起重视。譬如，对于布局，了解布局对象生成的页面元素和 CSS，对加深布局的理解和使用是很有帮助的。因此，我建议对 CSS 还不是很熟悉的开发人员，好好补上这一课。其实，这个也不难学。因为 Ext JS 4 中已经嵌入了很多 HTML 5 和 CSS 3 的内容，笔者建议阅读一下《HTML 5 与 CSS 3 权威指南》这本书，在补课的前提下顺便学习 HTML5 与 CSS3。

4. 面向对象的知识

Ext JS 框架是完全基于面向对象思想创建的，能掌握这方面的知识，对理解整个框架的运作和使用是非常有帮助。所以，我建议大家好好看看《设计模式》这本书。这不但对学习 Ext JS 有帮助，对学习服务器端的开发语言也是很有帮助的。

5. 三层架构的知识

Ext JS 本身在客户端就使用了三层架构，所有 UI 组件都是表现层，Store 是数据访问层。明白了这个，就可避免在 UI 组件里找数据了。而通过 Ajax 技术，把浏览器当做表现层，服务器端当做数据访问层。这样，服务器只通过 XML 格式或 JSON 格式提供必要数据就行了，全部的表现可在客户端通过 Ext JS 实现。这样就避免了服务器端代码和客户端代码混搭在一起的问题，从而实现了客户端与服务器端的脱钩，简单来说，就是定义好数据的通信格式，写客户端的可以不管服务器端，写服务器端的可以不管客户端，客户端根据通信格式接收数据，服务器根据通信格式提供数据就行了。这样的优点就是，无论服务器端的架构怎么变（例如我今天用 .NET 的，明天或许用 Java 的）都没关系，按通信格式提供数据客户端都能正确显示。客户说，这个界面不好，要改，没关系，修改客户端就行了，因为数据还是那些数据，与服务器端无关。

总的来说，只有明白了三层架构才能深入了解 Ext JS 的开发思想，才能使用它开发出好的 Web 应用程序。

介绍三层架构的书不多，原因可能是这个不算太复杂，理解上也不难。而且这已经是属于架构师的工作范围了。如果你的目标是架构师的话，可以阅读一下《架构之美》和《企业应用架构模式》。

6. 其他的建议

实践是最好的老师，如果碰到问题，或者有什么的想法：Just do it！错误是在所难免的，

只有做过，才懂得为什么不能这样，为什么要这样。笔者就是在错误中成长起来的，所谓吃一堑长一智！写 Web 应用，最大的问题就是宕机而已，有啥可怕的！问人，或许能解决一时的问题，但是不可能一个项目从头到尾都问别人。“自己动手，丰衣足食”是真理！

如果有时间，建议多看看 Ext JS 的 API 文档和多研究一下 Ext JS 压缩包中的例子，这对学习也是大有裨益的。如果能更进一步研究研究源代码，那就更好了。

如果看英文没问题，建议多上上 Ext JS 官网，看看里面的博文，去论坛走走，多交流交流，也是不错的。

1.2　JSON 概述

1.2.1　认识 JSON

XML 虽好，可作为数据交换格式，有时会喧宾夺主，标记比数据还多，徒增流量。更重要的是，在 JavaScript 中处理 XML 实在太不便利了。而 JSON，没有附加的标记，在 JavaScript 中可作为对象处理，因而渐渐成了目前 Web 开发的标准数据交互格式。

JSON 的英文全称是“JavaScript Object Notation”，意思就是 JavaScript 对象表示法。它是一种基于文本的、独立于语言的轻量级数据交换格式。它来源于 ECMA-262 第三版定义的 JavaScript 对象直接量（literal）。它不但易于阅读和编写，还易于机器解析和生成，而且完全独立于语言的文本格式，因而，JSON 是一种理想的数据交换语言。

1.2.2　JSON 的结构

JSON 有对象和数组两种结构。

对象结构以“{”（大括号）开始，“}”（大括号）结束。中间部分由 0 个或多个以“,”（逗号）分隔的“关键字（key）/ 值（value）”列表构成，而关键字与值之间必须以“：”（冒号）分隔。其结构语法如下：

```
{
    key1:value1,
    key2:value3,
    …
}
```

从上面的结构可以看到，这种结构的 JSON 有点类似其他语言中的字典或散列表。结构中的关键字是字符串，而值可以是字符串、数值、true、false、null、对象或数组。

注意　true、false 和 null 必须全部为小写字母。当值为对象或数组时，变量记录的是对象的指针。

数组结构以“[”（中括号）开始，“]”（中括号）结束。中间部分由 0 个或多个以“,”分隔的值（value）列表构成，其结构语法如下：

```
[value1,value2,…]
```

值可以是字符串、数值、true、false、null、对象或数组。

注意 如果在最后一个“关键字 / 值”后，“}”之前有 1 个“,”，如“{a:1,b:2,}”在 IE 8 及其之前版本的浏览器会报错，而在 IE 9 或其他浏览器则不会报错。这也是有些程序在 Firefox 中运行正常，在 IE 8 及其之前的浏览器中不能运行的主要原因。数组结构的 JSON 也存在这个问题，需要特别注意。

1.2.3 JSON 的例子

下面是一个 JSON 例子：

```
{
    1          : "这是允许的",
    "2"        : "这是允许的",
    "."        : "这是允许的",
    "中文"     : "这是允许的",
    count      : 3,
    person     : [
                   {id:1,name:"张三"},
                   {id:2,name:"李四"}
                 ],
    object     : {
                   id : 1,
                   msg : "对象里的对象"
                 }
}
```

从上面的例子可以看到，数字“1”也可以作为关键字。为什么呢？这是因为在 JavaScript 中，会自动根据需要将数字转换为字符串。从“2”、“.”和“中文”中可以看到，基本上，只要是字符串都可以作为关键字，不过笔者不建议使用这样的名称，因为这会造成对象访问上的问题，除非你永远保持使用“对象 [关键字]”的方式访问对象，这会在 1.2.4 节中详细描述。

在“person”中，其值是一个数组结构的 JSON，而该 JSON 又是由 JSON 结构的值构成的，这说明，这两种结构的 JSON 数据是可以嵌套使用的。而“object”值则表明 JSON 结构也可以作为值嵌套在 JSON 结构中。

1.2.4 在 JavaScript 中使用 JSON

因为 JSON 是 JavaScript 的一个子集，所以可以在 JavaScript 中轻松地读取、修改 JSON 中的数据并向 JSON 中添加数据。

在开始学习下面的内容之前，请先准备一个空白页面，然后在装有 Firebug 的 Firefox 中打开该页面，最后在 Firefox 中打开 Firebug 窗口并在控制台的命令行中输入以下代码：

```
var obj={
```

```
1          : "这是允许的",
"2"        : "这是允许的",
"."        : "这是允许的",
"中文"     : "这是允许的",
count      : 3,
person     : [
               {id:1,name:"张三"},
               {id:2,name:"李四"}
             ],
object     : {
                 id : 1,
                 msg : "对象里的对象"
             }
}
```

如果对 Firebug 还不熟悉，可先阅读第 3 章 3.1 节的内容。以上代码定义了一个 JSON 对象并赋值给变量 obj。从代码中可以看到，定义一个 JSON 对象非常简单，只要按照数据格式把数据写在“{ }”中就可以了。当然，你也可以定义一个空的 JSON 对象，代码如下：

```
var obj={};
```

以上代码就定义了一个空的 JSON 对象，在很多时候会使用到。

1. 读取单个数据

在 JSON 中读取数据有两种方法。第一种方法是在“.”（小数点）后加上关键字。第二种方法是在中括号中包含关键字。下面我们来测试一下这两种方法。

首先在 Firebug 中输入的代码后面加入以下代码：

```
console.log(obj.1)
```

代码中“obj”是指向 JSON 对象的变量，“1”是 JSON 对象关键字。“console.log”是 Firebug 用来在控制台输出信息的命令。

单击“运行”，会看到如图 1-1 所示的错误信息。

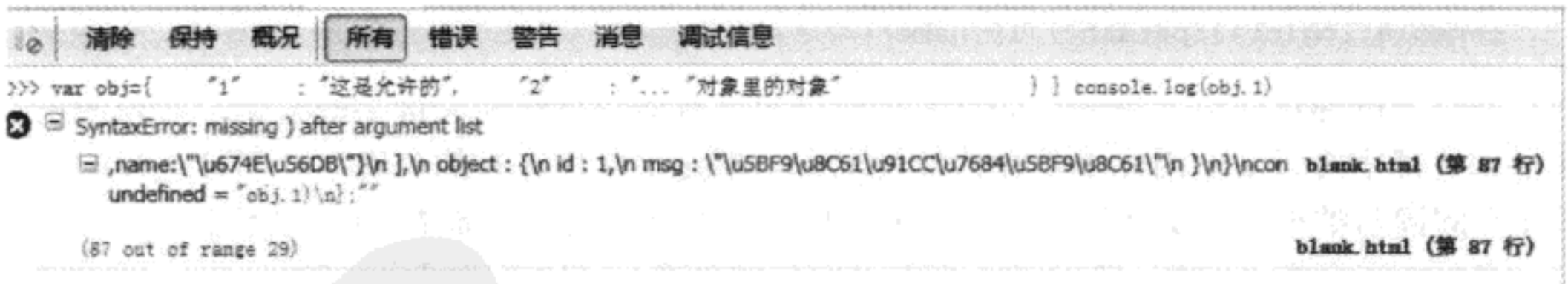


图 1-1　单击“运行”后的错误信息

这说明不能通过该方法读取关键字为数字的数据。将代码中的“1”替换为“2”、“.”，也会出现错误信息。

单击“清除”按钮清除控制台的信息后，将名称修改为“中文”，最后单击“运行”可看到如图 1-2 所示的信息。

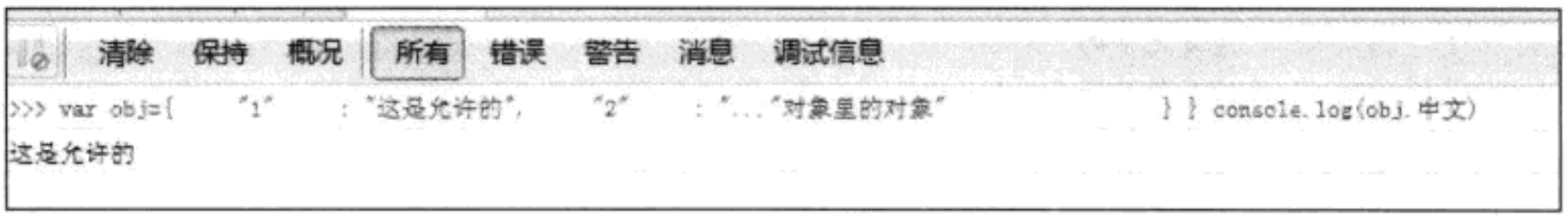


图 1-2 名称修改为“中文”后的运行结果

这说明中文是可以直接在“.”后使用的。

如果要读取“person”中第 2 个对象的“name”，可将“console.log”代码修改为下面的代码：

```
console.log(obj.person[1].name)
```

代码中，因为“person”是数组对象，所以可通过索引访问数据。代码运行后会在控制台显示“李四”。

同样，要访问“object”中的“msg”，可执行以下代码：

```
console.log(obj.object.msg)
```

下面介绍另外一种访问方法，将输出代码修改为以下代码：

```
console.log(obj[1])
```

代码中，中括号里面的“1”是关键字。运行后可在控制台看到“这是允许的”。如果要读取名称为“.”的数据，必须将小数点用双引号包裹起来，其代码如下：

```
console.log(obj["."])
```

小数点等运算符需要使用双引号包裹是因为这些符号在语句中被当成运算符了，而不是像数字一样自动转换为字符串。

要读取“person”第 2 个数据中的“name”，可使用以下代码：

```
console.log(obj["person"][1]["name"])
```

当然，你也可以结合两种方法来访问数据，譬如以下代码与上面代码的作用一样：

```
console.log(obj["person"][1].name)
```

从以上的测试可以了解到，如果要读取以数字或运算符作为关键字的数据，必须使用中括号加关键字的方法，因此不是在特殊情况下，不建议使用这样的关键字，而且这样的名称不便于阅读与维护。中文名称嘛，笔者也不推荐，不过根据自己需要灵活掌握吧。

最后一点要指出的是，在代码中要保持一致的风格，以便于代码的维护。

2. 遍历数据

要遍历 JSON 对象中的数据，可使用 for…in 循环。譬如要遍历上面示例中“obj”的数据，可编码如下：

```
for(var c in obj){
     console.log(c+":",obj[c])
}
```

代码中，通过循环可一个个读取 JSON 中的关键字，并将名称保存在变量“c”中，然后可使用中括号加关键字的方法读取数据。

在 Firebug 中运行该循环将看到如图 1-3 所示的显示结果。

```
>>> var obj={    "1"    : "这是允许的",    "2"    : "...r(var c in obj){    console.log(c+":",obj[c]) }
1: 这是允许的
2: 这是允许的
.: 这是允许的
中文: 这是允许的
count: 3
person: [ Object { id=1,  name="张三" }, Object { id=2,  name="李四" } ]
object: Object { id=1,  msg="对象里的对象" }
```

图 1-3　运行循环后的显示结果

3. 修改数据

要修改 JSON 对象的数据很简单，和普通变量赋值没什么区别。譬如要将示例中“count”的值修改为 10，可以执行以下的代码：

```
obj.count=10
```

或

```
Obj["count"] = 10
```

4. 添加数据

向 JSON 对象添加数据，使用以下格式的代码就行了：

```
JSON_Object.key= value
```

或

```
JSON_Object[key] = value
```

代码中，“JSON_Object”是指向 JSON 的变量，“key”是关键字，“value”是值。譬如在示例中增加关键字“sex”，值为“男”的数据，代码如下：

```
obj.sex = "男"
```

或

```
obj["sex"] = "男"
```

5. 删除数据

要删除 JSON 对象内的属性，使用 delete 语句就可以了。例如：

```
var json={name:"张三",sex:"男"}
delete json.sex
console.log(json)
```

执行后的结果为：

```
Object { name="张三"}
```

属性 sex 已被删除。

1.2.5 在 .NET 中使用 JSON

1. JSON.NET 概述

当 JSON 逐渐成为 Ajax 的标准数据交互格式时，在 .NET 中处理 JSON 数据只能使用字符串拼接的方法，十分麻烦，因而催生了 JSON.NET 这个项目。

JSON.NET 是一个免费的开源项目，大家可以登录 http://json.codeplex.com/ 下载最新版本，本书使用的版本是 4.0 release 1，本节的示例将使用该版本进行演示。

JSON.NET 的功能有很多，本书主要讲述以下两个 Ext JS 项目常用的功能：

- 通过序列化方法将 .NET 对象转换为 JSON 对象。
- 使用 LINQ to JSON 读写 JSON 对象。

2. 配置 JSON.NET

在 JSON.NET 压缩包的 bin 目录下有 Net、Net20、Net35、Silverlight 和 WindowsPhone5 个目录，目录里有对应不同 .Net Framework 版本的库文件，根据使用的 .Net Framework 版本使用对应的库文件就可以了。譬如本书的例子使用的是 .Net Framework 4.0 版本，因而将 Net35 目录下的 Newtonsoft.Json.Net35.dll 文件添加到项目的 bin 目录就可以了。

要使用序列化功能，需在代码中加入以下引用代码：

```
using Newtonsoft.Json;
```

如果要使用 LINQ to JSON，需在代码中加入以下引用代码：

```
using Newtonsoft.Json.Linq;
```

3. 序列化

在开发 Web 应用时，一般都需要将数据库查询出的数据转换为 JSON 格式文本传送回客户端，这就需要进行序列化。在 JSON.NET 中，要进行序列化，常用的是 JsonConvert 对象的 SerializeObject 方法。其基本的语法格式如下：

```
JsonConvert.SerializeObject(object)
```

代码中“object”就是要序列化的 .NET 对象。序列化后的返回值是字符串。

下面我们通过一个例子来加深一下认识。例子主要实现的功能是将微软示例数据库“Northwnd”中客户表（Customers）的所有客户数据以 JSON 格式返回客户端，其代码如下：

```
public string Message { get; set; }

    protected void Page_Load(object sender, EventArgs e)
     {
         NorthwindEntities ne = new NorthwindEntities();
         var q = ne.Customers.OrderBy(m=>m.CompanyName)
            .Select(m=>new {
```

```
        m.CustomerID,
        m.CompanyName,
        m.Country,
        m.City,
        m.Address,
        m.PostalCode,
        m.Phone,
        m.Region
    });
    Message = JsonConvert.SerializeObject(q);
}
```

代码中，首先从实体模型中查询出数据集合“q”，然后将数据集合“q”序列化成 JSON 格式字符串并赋值给变量“Message”，最后在页面中输出。在浏览器中打开页面将看到以下的结果：

```
[{"CustomerID":"ALFKI","CompanyName":"Alfreds Futterkiste","Country":"Germany",
    "City":"Berlin","Address":"Obere Str. 57","PostalCode":"12209","Phone":"030-
    0074321","Region":null},
...,
{"CustomerID":"WOLZA","CompanyName":"Wolski Zajazd","Country":"Poland","City":"Wa
    rszawa","Address":"ul. Filtrowa 68","PostalCode":"01-012","Phone":"(26) 642-
    7012","Region":null}]
```

从上面的例子可以看到，将查询数据序列化成 JSON 文本是一件非常简单的事。其实，对 .NET 对象的序列化还有很多方式，囿于篇幅，本书就不一一介绍了，有兴趣可以详细阅读 JSON.NET 的文档。

4. LINQ to JSON

事实上，Ext JS 对数据返回的格式是有一定要求的，并不是简单地返回序列化后的数据就行，这时就需要用到 LINQ to JSON。LINQ to JSON 的作用就是根据需要的 JSON 格式组织文本数据。

LINQ to JSON 需要使用到 JObject、JArray、JPropery 和 JValue 4 个对象，这 4 个对象的详细说明如表 1-1 所示。

表 1-1　LINQ to JSON 对象说明

对　象	说　明
JObject	生成一个 JSON 对象，形象点来说就是生成“{}”
JArray	生成一个 JSON 数组，形象点来说就是生成“[]”
JProperty	生成一个 JSON 数据，格式如下： JProperty(name,value) name: 类型为字符串，关键字 value：类型为对象，值
JValue	直接生成一个 JSON 值。在生成仅有值的数组时，就需要使用 JValue 进行转换。其格式如下： JValue(value) value：类型为对象，值

下面我们通过一个例子说明如何使用 LINQ to JSON。Ext JS 的所需 JSON 格式数据一般如下：

```
{
  "total":5, // 记录总数
  "rows":[
     //JSON 对象格式的数据列表
  ]
}
```

示例将演示如何根据以上格式返回客户表数据，代码如下：

```
public string Message { get; set; }

protected void Page_Load(object sender, EventArgs e)
{
    NorthwindEntities ne = new NorthwindEntities();
    var q = ne.Customers.OrderBy(m => m.CompanyName)
        .Select(m => new
        {
            m.CustomerID,
            m.CompanyName,
            m.ContactName
        }
        ).ToList();
    Message = new JObject(
        new JProperty("total", q.Count()), //创建记录总数
        new JProperty("rows",
            new JArray(     //创建数据数组
                from p in q
                select  new JObject(
                    new JProperty("id", p.CustomerID),
                    new JProperty("cpname", p.CompanyName),
                    new JProperty("contactName", p.ContactName)
                    )
                )
            )
        ).ToString();
}
```

从上面的代码可以看到，构建固定格式的 JSON 数据是相当直观的。将粗体代码与格式数据对比，可以看到，最外层的 JObject 创建了格式中最外层的“{}”，然后依次使用 JProperty 生成记录总数数据和数据列表。而代码中的 JArray 的作用就是生成“[]”，将使用 LINQ to JSON 方式生成的一个个数据对象组合成数组。本来是希望直接通过 LINQ to JSON 将实体模型转换成 JSON 的，但这样会产生“LINQ to Entities 仅支持无参数构造函数和初始值设定项”的错误，因而本示例先将查询的数据转换为列表（ToList()），再进行转换。使用 LINQ to JSON 可直接在 select 语句生成 JSON 数据对象，无须其他转换过程，相当方便。在使用 select 语句生成数据对象时，首先使用 JObject 生成“{}”，然后使用 JProperty 生成对象的数据。

在浏览器中打开页面，将看到以下的结果：

```
{ "total": 91, "rows": [ { "id": "ALFKI", "cpname": "Alfreds Futterkiste",
```

```
        "contactName": "Maria Anders" }, { "id": "ANATR", "cpname": "Ana Trujillo
        Emparedados y helados", "contactName": "Ana Trujillo" },
...,
{ "id": "WOLZA", "cpname": "Wolski Zajazd", "contactName": "Zbyszek
    Piestrzeniewicz" } ] }
```

注意　千万不要使用序列化的方式生成“rows”的值，如下面的代码：

```
new JProperty("rows",JsonConvert.SerializeObject(q)) ;
```

因为这样生成的“rows”值是字符串，而不是数组。

5. 处理客户端提交的 JSON 数据

有时候，在客户端以 JSON 格式将数据提交到服务器比较方便。譬如，直接在 Grid 修改了不同行和列的数据，最后一次性将修改的数据提交到服务器端处理，这时候，使用 JSON 格式提交数据会很方便，例如以下提交的数据：

```
[
   {id:"12345",title:"文章一",author:"李四"},
   {id:"12367",author:"张三"},
   {id:"17777",isShow:true}
]
```

数据表示在 Grid 中修改了 3 行数据，第 1 行修改了标题（title）和作者（author），第 2 行修改了作者，第 3 行修改了是否显示（isShow）。

在服务器端使用 JObject 或 JArray 的 Parse 方法就可轻松地将字符串转换为 JSON 对象，然后通过对象的方法提取数据，下面是服务器端的处理代码：

```
public string Message { get; set; }
protected void Page_Load(object sender, EventArgs e)
{
    string jsonString = @"
        [
           {id:'12345',title:'文章一',author:'李四'},
           {id:'12367',author:'张三'},
           {id:'17777',isShow:true}
        ]
        ";
    JArray json = JArray.Parse(jsonString);
    Message = @"<table border='1'>
            <tr><td width='80'>ID</td><td width='100'>字段</td><td width='100'>
                值</td></tr>
        ";
    string tpl = "<tr><td>{0}</td><td>{1}</td><td>{2}</td></tr>";
    foreach (JObject jobject in json)
    {

        foreach (var a in jobject)
        {
            if(a.Value.ToString()!="id")
                Message += String.Format(tpl, (string)jobject["id"], a.Key,
                    a.Value);
```

```
            }
        }
        Message += "</table>";
```

在代码中，因为已知数据是使用数组形式提交的，所以采用 JArray 的 Parse 方法。

什么？你不知道数据是以数组还是对象形式提交？

这……

数据的提交方式应该是一种双方的约定，不然要处理未知的数据会很麻烦，所以不用担心这个问题。

第 1 层 foreach 循环用来获取 JObject 对象。第 2 层 foreach 用来获取修改过的字段名称。在这里要注意，数据默认已约定存在 id 这个字段。

代码运行后将看到如图 1-4 所示结果。

从结果可以看到，数据已经被分拆出来，这样你就可以根据 id 和字段去更新数据库了。

ID	字段	值
12345	id	"12345"
12345	title	"文章一"
12345	author	"李四"
12367	id	"12367"
12367	author	"张三"
17777	id	"17777"
17777	isShow	true

图 1-4　处理客户端提交的 JSON 数据页面的显示结果

1.2.6　在 Java 中使用 JSON

1. Gson 概述

Gson 是谷歌的一个开源项目，其作用是在 Java 对象和 JSON 之间实现相互转换。大家可登录 http://code.google.com/p/google-gson/ 下载最新版本，本书使用的版本是 1.6 版，本节的示例都将使用该版本。

Gson 的功能很多，这里囿于篇幅就不一一介绍了。本节的重点是讲述如何使用 Gson 生成 Ext JS 格式的返回数据。

2. 配置 Gson

要使用 Gson，将 gson-1.6.jar 文件复制到项目的“lib”目录就行了。譬如在动态 Web 项目中使用 Gson，将文件复制到“\WebContent\WEB-INF\”下的 lib 目录即可。

要引用 Gson，需在引用文件中加入以下代码：

```
import com.google.gson.*;
```

你也可以根据需要细化引用。

3. 使用 Gson

要生成 1.2.5 节中介绍的 JSON 格式数据，需要使用 JsonObject 和 JsonArray 两个对象，这两个对象的详细说明如表 1-2 所示。

表 1-2　JsonObject 与 JsonArray 的详细说明

对　象	说　明
JObject	生成一个 JSON 对象，形象点来说就是生成“{}”
JArray	生成一个 JSON 数组，形象点来说就是生成“[]”

这两个对象是如何使用的，请看下面的代码：

```
String connectionUrl = "jdbc:sqlserver://192.168.0.254:1433;" +
   "databaseName=Northwind;;user=sa;password=abcd-1234";

Connection con = null;
Statement stmt = null;
ResultSet rs = null;

try {
// 使用 JDBC 从数据库获取数据 Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");

   con = DriverManager.getConnection(connectionUrl);
   String SQL = "SELECT CustomerID,CompanyName,ContactName " +
         "FROM Customers order by CompanyName";
   stmt = con.createStatement();

   rs = stmt.executeQuery(SQL);
   int count = 0;  // 计算记录总数
   // 构建数据列表
   JsonArray array=new JsonArray();
   while (rs.next()) {
   // 构建每行数据对象
      JsonObject obj= new JsonObject();
      obj.addProperty("id", rs.getString("CustomerID"));
      obj.addProperty("cpname", rs.getString("CompanyName"));
      obj.addProperty("contactName", rs.getString("ContactName"));
      array.add(obj);
      count++;
   }
   // 构建返回格式数据
   JsonObject json=new JsonObject();
   json.addProperty("totals", count);
   json.add("rows", array);
   response.getWriter().write(json.toString());
   rs.close();

}
catch (Exception e) {
   response.getWriter().write(e.getMessage());
}
finally {
   if (rs != null) try { rs.close(); } catch(Exception e) {}
   if (stmt != null) try { stmt.close(); } catch(Exception e) {}
   if (con != null) try { con.close(); } catch(Exception e) {}
}
```

代码中，注释“构建数据列表”之前的代码是实现数据库查询数据的，而我们的重点是JSON，所以我们的关注点是注释下面的代码。因为 Gson 不支持 LINQ to JSON，所以我们必须一步步地构建 JOSN 数据。

首先创建一个 JsonArray 对象，准备在循环中插入数据。在 while 循环中，每行数据就是一个 JsonObject 对象，因而要创建新的 JsonObject 对象，然后使用 addProperty 方法将每列数据添加到 JsonObject 对象中，最后是将这个 JsonObject 对象使用 JsonArray 的 add 方法添加到数组中。这样，要返回的数据列表就构建完成了。

最后一步就是构建最外层的 JsonObject 对象，这个步骤比较简单。首先是使用 addProperty 方法添加记录总数，然后使用 add 方法将 JsonArray 对象作为“rows”的值添加到 JsonObject 对象中，最后使用 toString 方法转换成字符串返回客户端。在这里要注意 JsonObject 对象的 addProperty 方法和 add 方法的区别，addProperty 方法是用来添加原生数据类型的，而 add 方法是用来添加 JsonElement（包括 JsonObject、JsonArray、JsonPrimitive 和 JsonNull）对象的，详细的说明可阅读 Gson 的 API。

4. 处理客户端提交的 JSON 数据

在 Java 中要处理 1.2.5 节中介绍的 JSON 数据，可使用 JsonParser 对象的 Parse 方法，具体代码如下：

```
response.setContentType("text/html; charset=utf-8");
    String josnStr = "[" +
     "{id:'12345',title:'文章一',author:'李四'},"+
     "{id:'12367',author:'张三'},"+
     "{id:'17777',isShow:true}"+
"]";
String tplString ="<tr><td>%1$s</td><td>%2$s</td><td>%3$s</td></tr>";
 response.getWriter().write("<table border='1'>"+
                "<tr><td width='80'>ID</td><td width='100'>字段</td><td width=
                    '100'>值</td></tr>"
                );
 JsonParser jparser = new JsonParser();
 JsonArray ja = jparser.parse(josnStr).getAsJsonArray();
 for (JsonElement je : ja) {
   JsonObject jo = je.getAsJsonObject();
 Set<Map.Entry<String, JsonElement>> jset= je.getAsJsonObject().entrySet();
 String id = jo.get("id").getAsString();
 for (Map.Entry<String, JsonElement> map : jset) {
      String key =map.getKey();
      if(key !="id"){
                response.getWriter().write(
                      String.format(tplString, id,key,map.getValue())
                );
          }
    }
}
response.getWriter().write("</table>");
```

代码中，首先使用 JsonParser 对象的 Parse 方法将字符串转换为 JsonElement 对象，然后使用 getAsJsonArray 方法将其转换为 JsonArray。通过循环从 JsonArray 中获取 JsonObject

对象。因为 JsonObject 没有提供获取关键字的方法，所以要将 JsonObject 对象转换为 Set，再将 Set 中的数据转换为 Map 集合，最后使用 getKey 方法提取关键字。

代码运行后的结果参见图 1-4。

1.2.7　更多有关 JSON 的信息

在本书只是简单地介绍 C# 和 Java 这两种开发语言处理 JSON 数据的方法，如果读者需要其他语言有关的 JSON 的信息，或不喜欢笔者介绍的两个 JSON 库，可登录 http://www.json.org/json-zh.html 获取更多信息。

1.3　Ext JS 4 概述

1. 令人兴奋的 Ext JS 4

曾经笔者以为 Ext JS 会止步于 Ext JS 3，而全力投身于移动开放领域。而且 HTML 5 的强势出现，在笔者看来，会对 JavaScript 框架带来一定的冲击，Ext JS 前景不太乐观。想不到在 2010 年年底，在 Sencha 博客上出现一篇名为《Ext JS 4 Preview: Refactoring & Standardizing the Rendering Process》的文章，才知道，Ext JS 即将发布新版本，而且新版本不是简单地在 Ext JS 3 基础上增加新功能，而是完全重构，以获得更好性能。在持续的等待中，Sencha 博客不断有介绍 Ext JS 4 的文章发表，在这些文章中逐渐明晰了 Ext JS 4 的改变，而这些改变，每一项都是那么令人兴奋。

2. Ext JS 3 与 Ext JS 4 的兼容问题

Ext JS 4 因为框架重构，使用 Ext JS 3 开发的应用程序如果要平滑转移到 Ext JS 4 平台上，工作量是巨大的，如何才能实现两者间的兼容，让已有的 Ext JS 3 项目可以使用 Ext JS 4 呢？Ext JS 4 团队提供了以下两种方法：

- JS 兼容层：为 Ext JS 3 提供一个可选的 JavaScript 文件，其作用是在 Ext JS 4 加载后，为 Ext JS 3 代码提供别名和重写功能，从而让许多 Ext JS 3 代码能在 Ext JS 4 中正确运行。
- 沙盒模式：实际就是将 Ext JS 全局对象名称修改为 Ext 4，这样就不会与 Ext JS 3 产生冲突了。这样，两个版本的 Ext JS 就可在同一个页面使用。详细情况可浏览 Ext JS 4 包中的“Ext JS 3 & 4 on one page”示例。

3. 新的类系统

在 2011 年 1 月 19 日的博客文章《Count-down to Ext JS 4: Dynamic Loading and New Class System》中，出现了如图 1-5 所示的新的类图。

从图中可以看到类系统在原有的基础上添加了 mixin 和 require 这两个新特性。实际新特性还有不少，请看下面的新特性列表：

- 使用 Ext.define 替代 Ext JS 3 中的 Ext.extend 来定义新类。
- 实现类的自动依赖管理，以便实现动态加载。

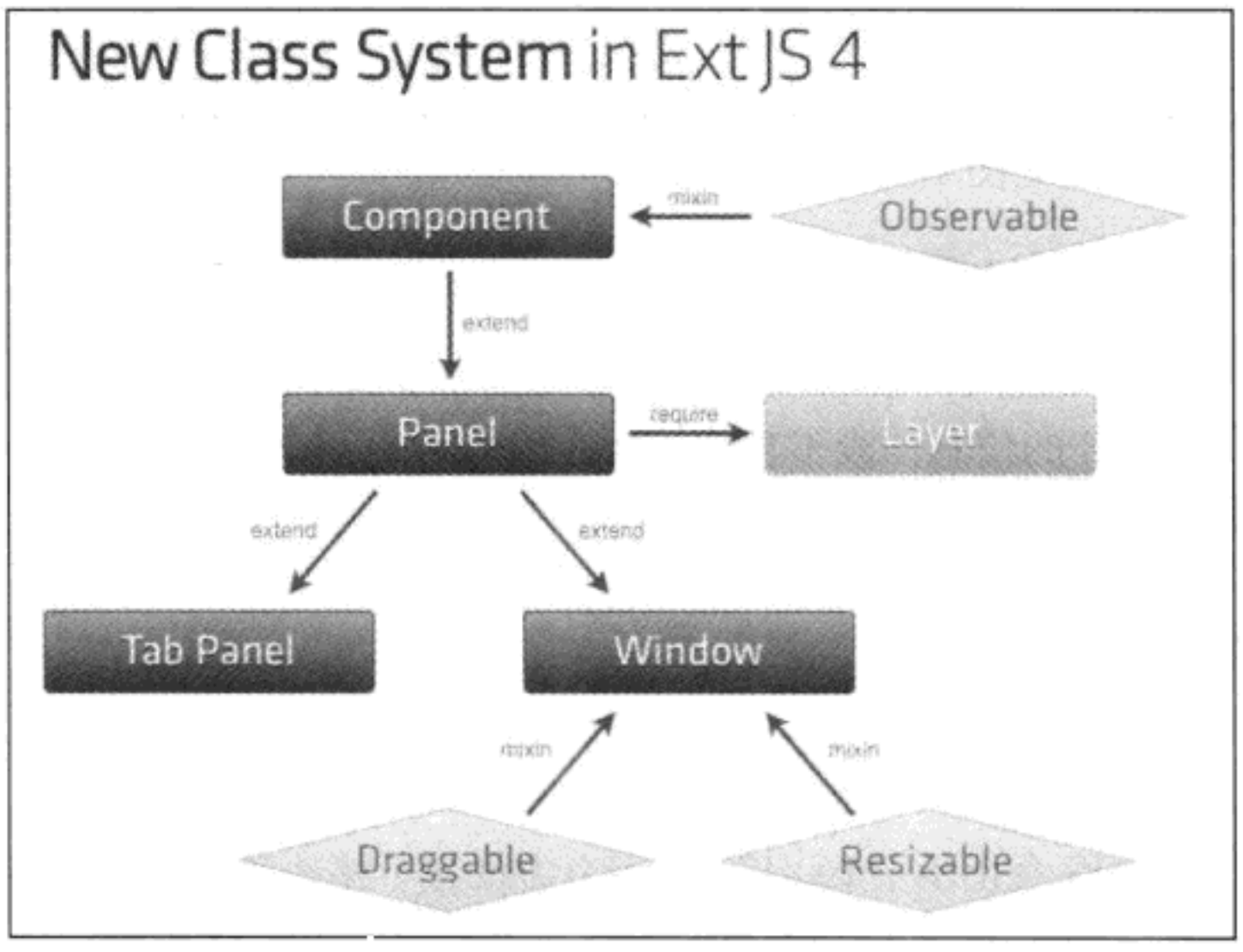


图 1-5　新的类图

- 通过 Mixins 将一些特殊功能添加到类中，如图 1-5 所示的通过 Mixins 功能为 Window 添加了拖放和缩放功能。
- 可创建类似 Java 或 C# 中的静态类（Statics）。
- 可为配置选项自动生成 set 和 get 方法。
- 定义类的时候，如果需要，可自动生成命名管道，而不是像 Ext JS 3 一样，需要先定义命名管道，之后才可以定义类。
- 动态加载。

4. 动态加载

动态加载的作用就是根据应用程序需要加载相应的脚本文件，而不是一次性加载所有脚本或加载一个保护全部功能的框架文件。其主要目的就是提高页面加载速度。

Ext JS 4 有一个完整的类依赖图，在加载某个类的时候，会依据依赖图递归下载所需的类文件，从而使应用程序正确运行。

实现此功能的要点就是在定义类的时候，设置类的 requires 或 uses 属性。两个属性的区别在于 uses 属性只是使用到该类，而不是必需的，这些类可以异步加载，而且不需要实例化。

动态加载用于加载 Ext JS 的类文件时，对于使用组件不多的应用程序来说相当不错，但是对于一个大的应用程序，建议还是使用完整的框架包。主要原因是拆分的类文件虽然减少了页面的下载流量，但是会增加服务器请求数量，增加服务器的负担，因而未必能加快页面加载速度，这个要权衡好。不过，在单页面应用中，使用动态加载模型文件、用户自定义组件等是不错的选择。

5. mixins

这是 Ext JS 4 中一个很实用的新特性。使用 mixins 配置属性，可为类添加特殊的功能。

它有点类似插件，不过它在类初始化的时候会混合在类的原型中。与如图 1-5 所示的 Window 类一样，可以轻松地将拖放或缩放混合到类中。

6. 自动的配置功能

在定义新类的时候，在 config 属性中定义的任何属性，类系统都会自动为其添加 set、get、reset 和 apply 方法，从而能够在调用代码中配置这些属性。

7. 新增的数据模型

在 Ext JS 4 中新增了数据模型特性，它与 3 版的 Record 类类似，但是功能更强大。在模型内就可以实现验证、关联和数据处理等功能。

8. 全新的绘图与图表功能

在 Ext JS 3 中使用的是基于 Flash 的图表，在使用上会受到一定的限制。在 Ext JS 4 中，使用 Canvas、SVG 和 VML 等基本图形功能，实现了全新的绘图和图表功能。

这是让企业应用开发人员最兴奋的一个功能。因为终于可以在客户端轻松实现强大的图表功能，不再需要关心那些烦人的 Flash 了。

9. 重新架构的 Grid 组件

Grid 组件在 Ext JS 4 中进行了重新架构。在新的架构下，EditorGrid 消失了，在 Grid 下就可轻松实现编辑功能。下面是重新架构后的 Grid 拥有的新特性：

- ❑ 智能渲染：在 Ext JS 3 中进行 Gird 渲染时，无论你是否需要这个特性，渲染都会生成这个特性的标记，从而降低渲染速度和性能。在第 4 版，渲染就聪明得多了，它按需渲染，只生成所需的标记，因而大大提高了渲染速度和性能。
- ❑ 标准的布局：因为采用了智能渲染，Grid 的许多部件都可以做成标准的组件并集成到标准的布局管理系统中，再不依赖自定义的内部的标记和 CSS。这方面突出的例子是 HeaderContainer 类，在 Ext JS 4，列标题是真正的容器，这样就可以使用 HBox 布局，让你使用 Flex 值定义列的宽度。
- ❑ 特性支持：在 Ext JS 4 中，有一个名称为 Ext.grid.Feature 的类，通过它可以非常灵活地为 Grid 创建新功能，而不是像 Ext JS 3 那样，通过插件实现新的功能。目前已经实现的特性功能包括 RowWrap、RowBody、分组（Grouping）以及分块（Chunking）或缓冲（Buffering）。
- ❑ 虚拟滚动：该功能主要为 Grid 提供虚拟的、按需加载的数据视图，从而实现无分页浏览。
- ❑ 编辑改进：在 Ext JS 4 中，EditorGrid 消失了，代替它的是 Editing 插件，从而可在任何 Grid 中实现编辑功能。而且，RowEditor 扩展已经成为了 Ext JS 4 的一个组件。
- ❑ 数据视图（DataView）：GridView 现在扩展自 DataView，从而使 Grid 可以轻松定制，而且也使 Grid 可以和 DataView 使用相同的选择模型，如实现键盘导航选择行等。

10. 新的主题特性

在 Ext JS 3 中，要改变 Ext JS 的主题是相当不容易的事情，需要做大量的工作。在 Ext JS 4 中，采用了 Compass 和 SASS，从而使更换主题成为一件非常轻松的工作。SASS 是标准 CSS

的一个超集，支持以下高级功能：

- 内部选择器。
- 变量。
- Mixins。
- 选择器继承。
- 编译和压缩。
- 为特殊应用程序中不需要的组件完全删除 CSS。

1.4 Ext JS 的开发工具的获取、安装与配置介绍

1.4.1 Ext Designer

Ext Designer 是一个所见即所得的创建 Ext JS 界面的工具软件，目前版本是 1.2 版，支持 Ext JS 3.x 和 4.x 版本，可在 http://www.sencha.com/products/designer/download/ 下载试用版本。

图 1-6 语言选择窗口

1. 安装

双击下载文件将看到如图 1-6 所示的语言选择窗口。

选择简体中文后，单击“OK”按钮后将看到如图 1-7 所示的设定窗口。

图 1-7 设定窗口

单击“前进”按钮，将看到如图 1-8 所示的软件授权协议窗口。

图 1-8　软件授权协议窗口

选择“我接受此协议”，单击“前进”按钮，将看到如图 1-9 所示的安装目录窗口。

图 1-9　安装目录窗口

选择目录后，连续单击两次“前进”按钮，进入安装过程，安装结束后，将看到如图 1-10 所示的安装完成窗口。

图 1-10　完成安装窗口

2. 使用

单击桌面上的“Ext Designer”图标，将看到如图 1-11 所示的注册窗口，使用 Sencha 论坛的账号登录后，将看到如图 1-12 所示的购买信息窗口。

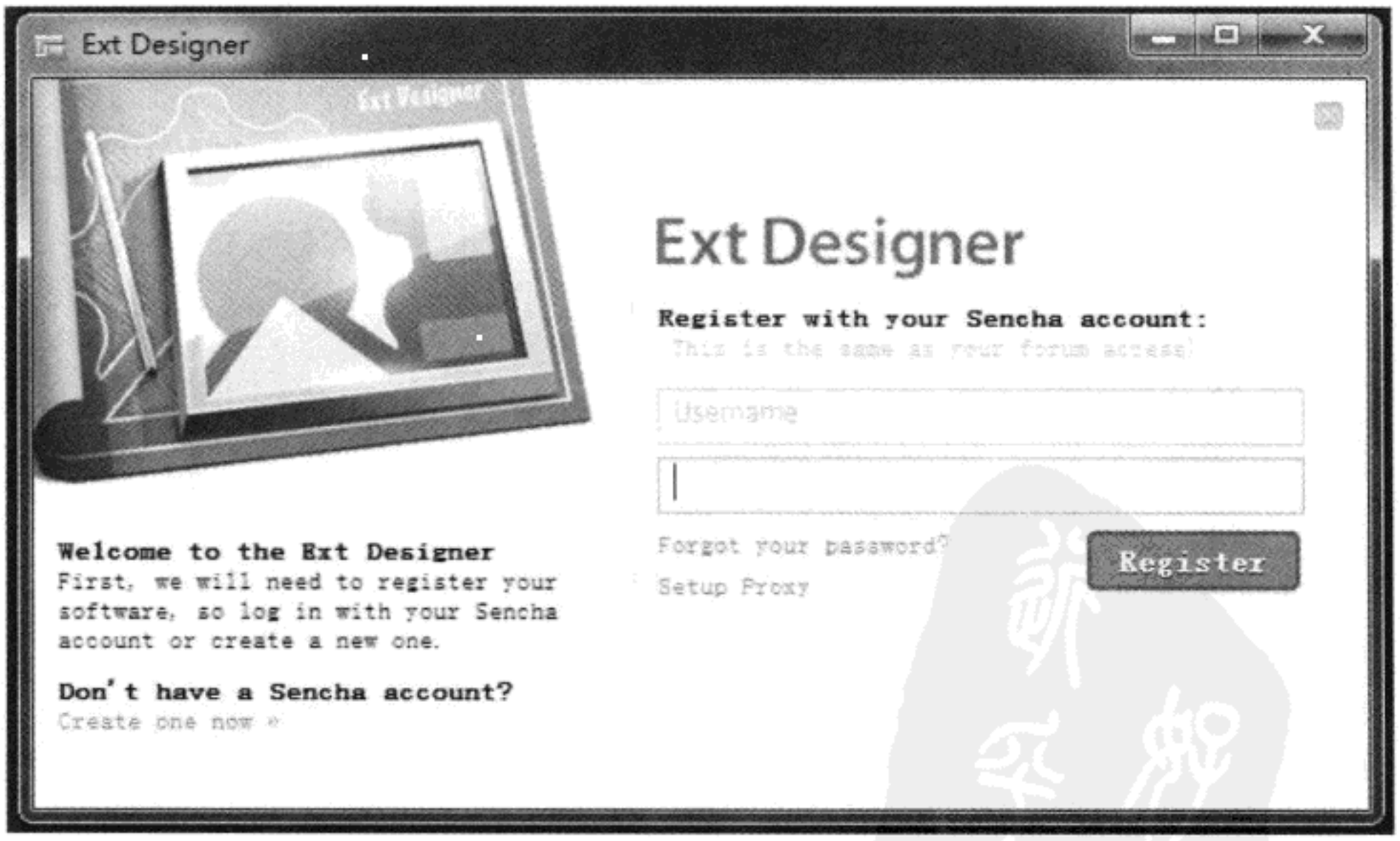


图 1-11　注册窗口

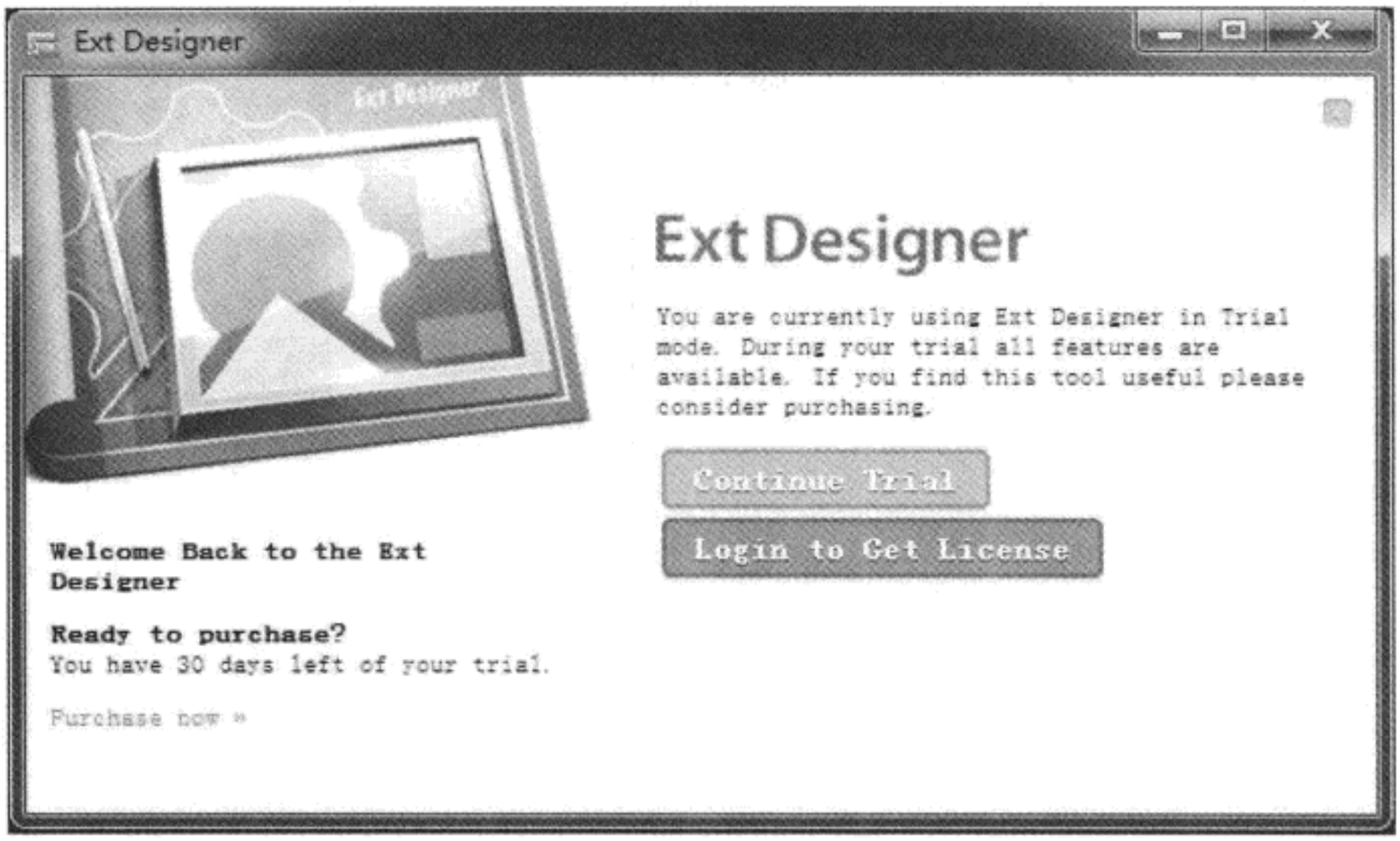


图 1-12　购买信息窗口

单击“Continue Trial”按钮，将看到如图 1-13 所示的项目选择窗口。

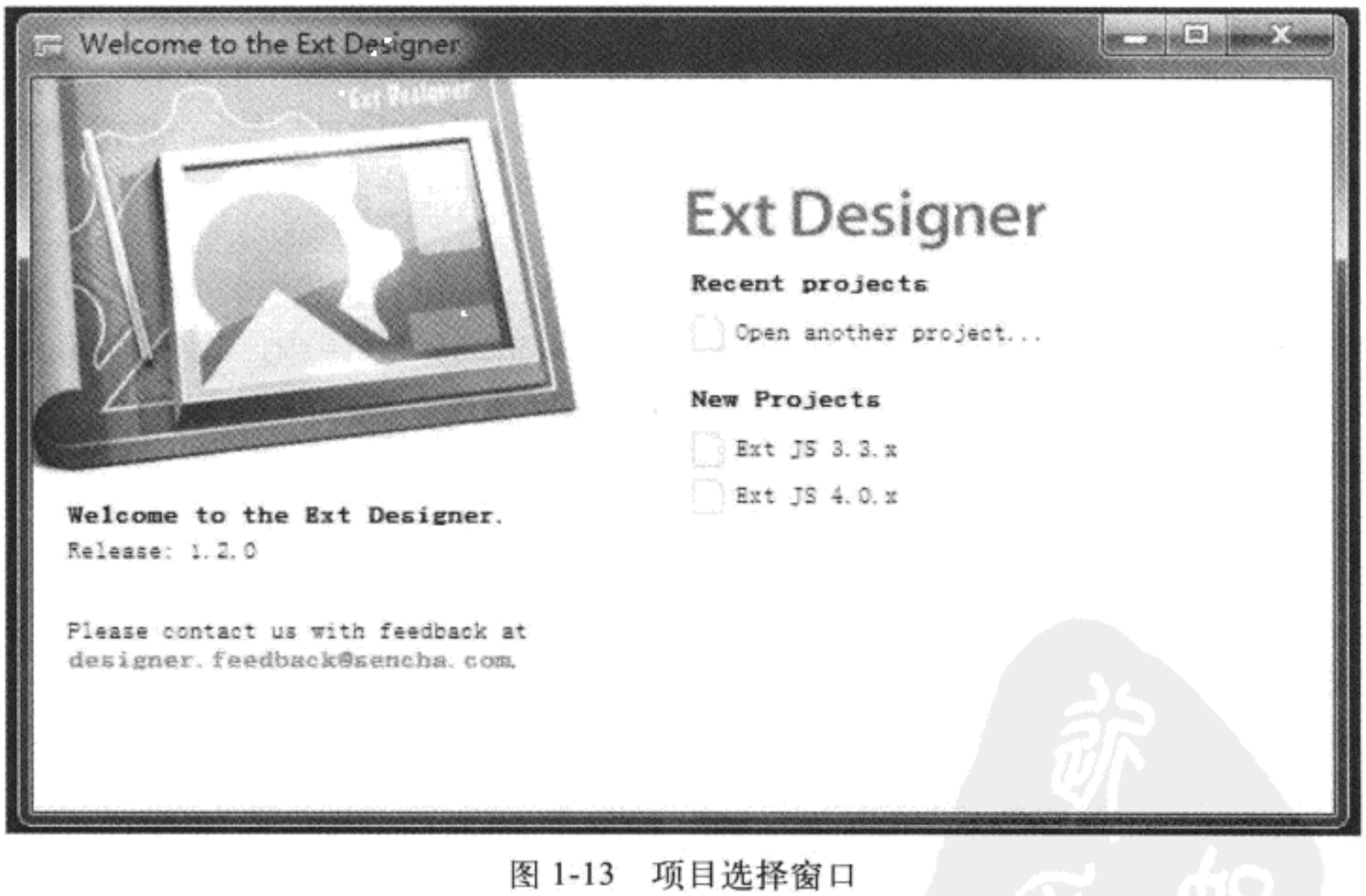


图 1-13　项目选择窗口

单击“Ext JS 4.0.x”开始一个新项目，将看到如图 1-14 所示的主窗口。

从主窗口可以看到，主要分为 3 个区域，左边是组件选择区域，中间是编辑区域，右边是组件树、Store 和属性窗口，与我们习惯的开发工具没有区别。

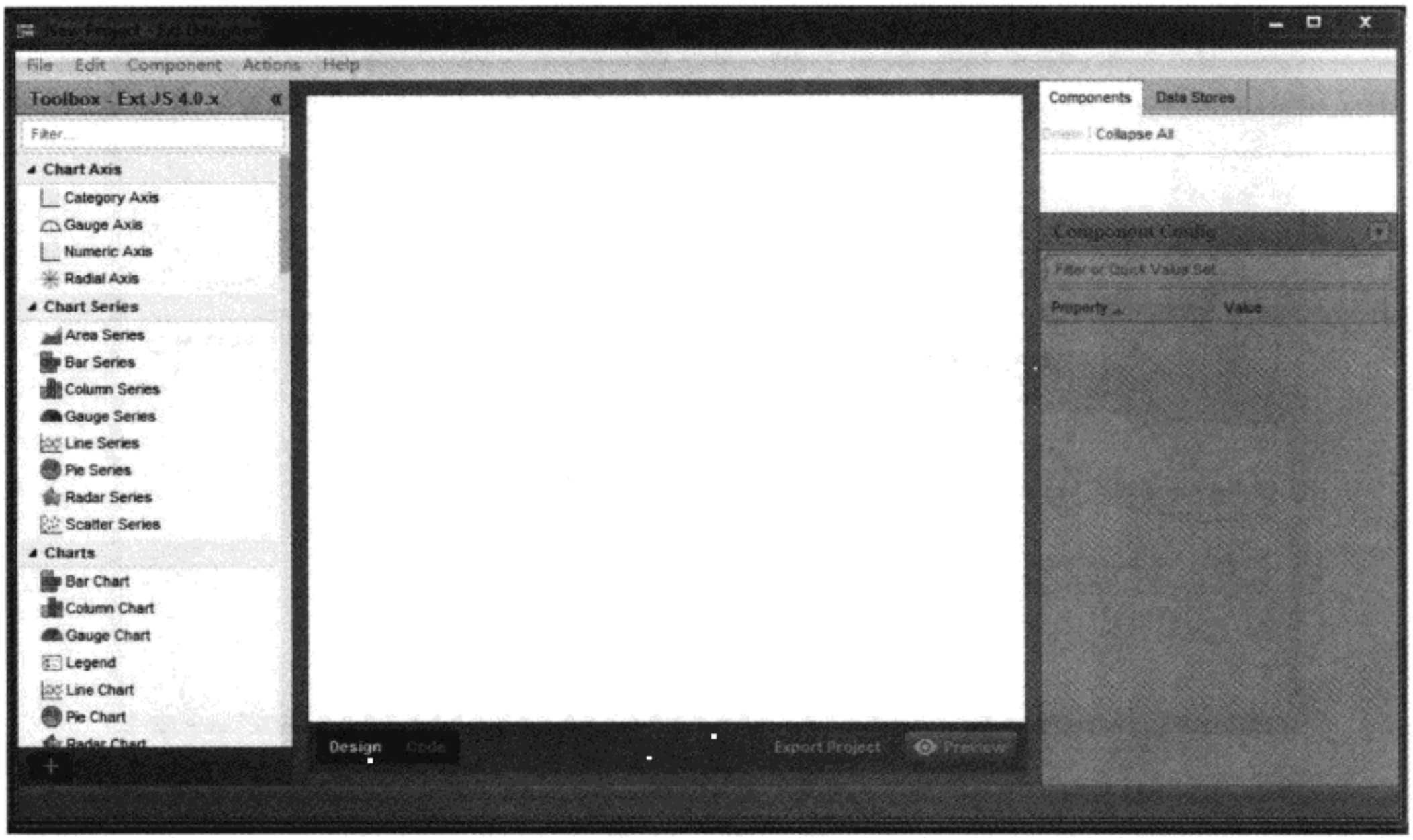


图 1-14　主窗口

我们尝试制作一个通用的邮件界面。在左边找到 Containers 面板，把 Viewprot 拖动到编辑区域，然后设置其 layout 属性为 border。接着找到“Tree/Tree Grid”面板，把“Tree Panel”拖动到 Viewport 内，然后设置其 width 属性为 200，maxWidth 属性为 300，region 属性为 west，勾选 split 属性。

接着在 Containers 面板拖动一个 Container 组件到 Viewport 内，设置其 layout 为 border。

再在 Containers 面板内拖动一个 Panel 组件到 Container 组件内，设置 region 属性为 south，属性 height 为 100，maxHeight 为 300，勾选 split 属性。

最后在 Grid 面板，拖动一个 Grid Panel 到 Container 组件内。

我们将看到如图 1-15 所示的最终效果。

单击“Preview”按钮，将看到如图 1-16 所示的预览窗口。

单击“Code”按钮将切换到如图 1-17 所示的代码显示窗口。

代码分为 Class 和 JSON 两种格式。Class 格式会把刚才的设计定义为一个扩展，保存到一个脚本文件内，然后通过动态加载或直接使用 Script 标记引用，再实例化。而 JSON 格式则可直接使用 create 方法进行实例化。

总体来说，Ext Designer 是一款不错的 Ext JS 设计工具，尤其是在使用 Ext JS MVC 应用框架的时候。单个包 219 美元也不算太贵，值得购买一套，然后找专人负责界面的设计，笔者觉得是物有所值的。

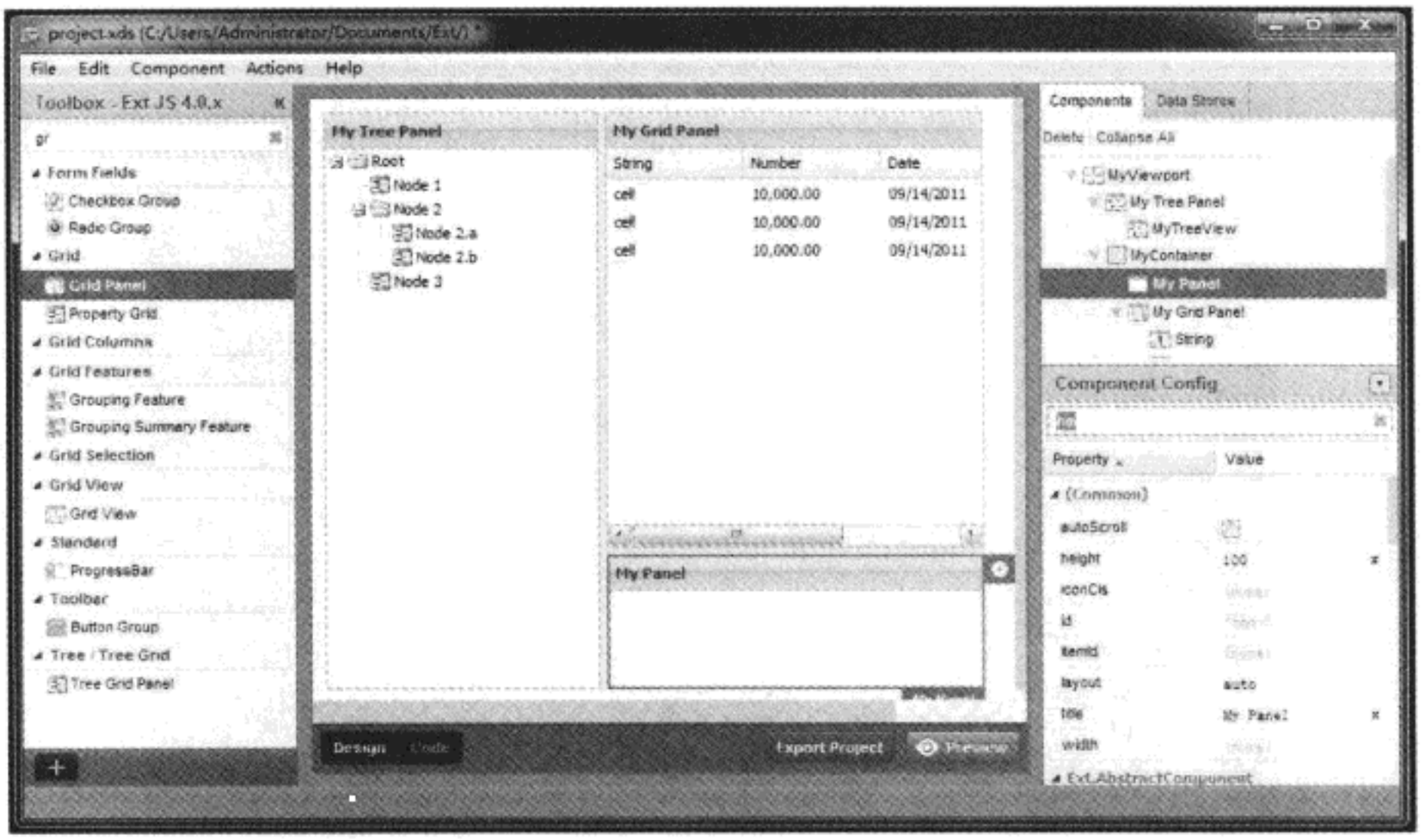


图 1-15　设计的最终效果

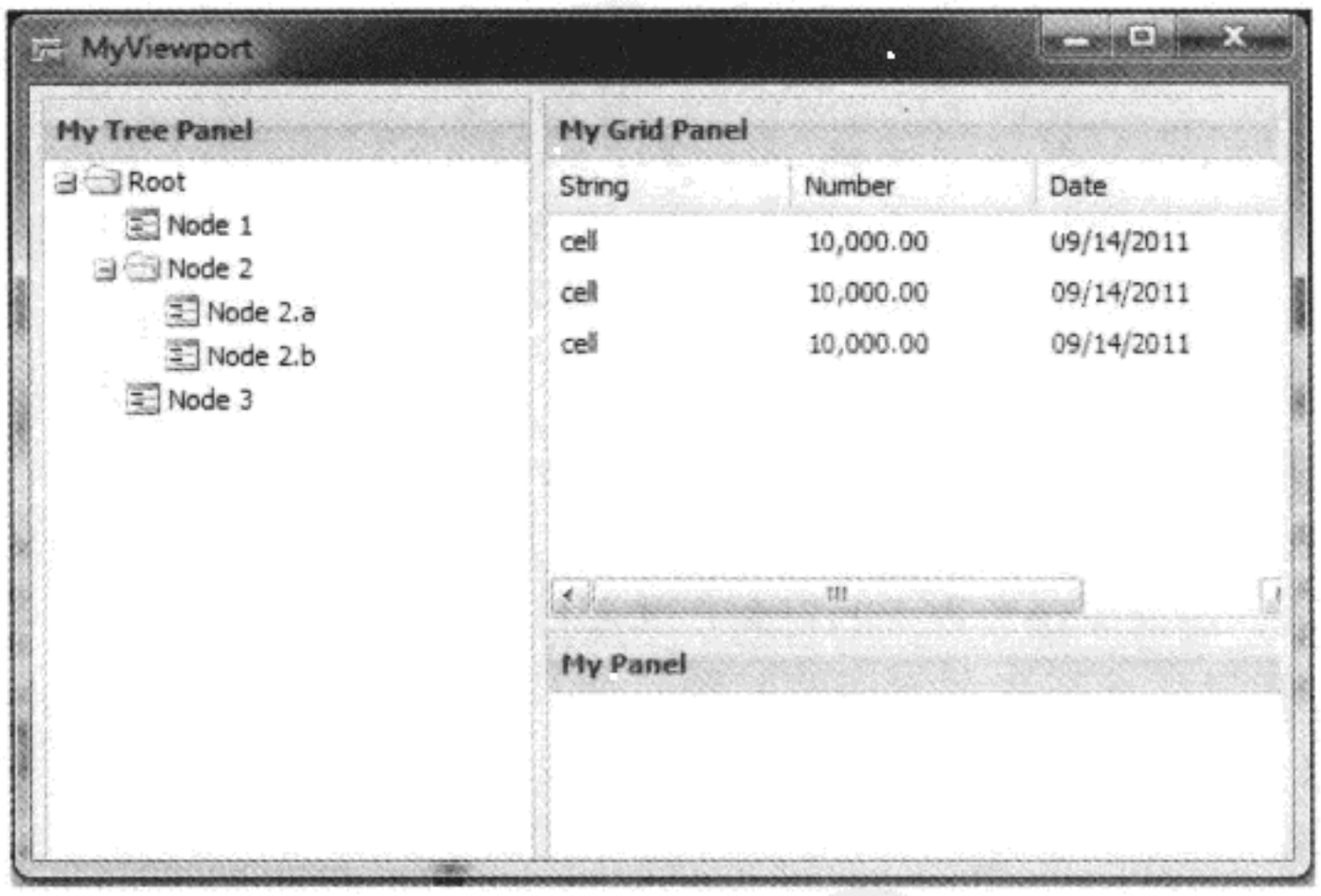


图 1-16　预览窗口

1.4.2　在 Visual Studio 中实现智能提示

Visual Studio（下面简称 VS）2008 年和 2010 年都有从 JS 库文件获取智能提示的功能，不过直接使用 Ext JS 的库文件的话，虽然也有提示，但是不全，应该说是大部分没有。主要原因是 VS 脚本提示功能是根据原生 JavaScript 对象的结构读取的，而 Ext JS 的对象定义与原生 JavaScript 对象完全不同，因而很难取得其属性和方法。因而，要完整支持 Ext JS 的智能提示，就必须将 Ext JS 对象的属性和方法提炼出来，然后组织成一个 VS 能读懂的 JavaScript 对

象结构，这样就能实现智能提示了。

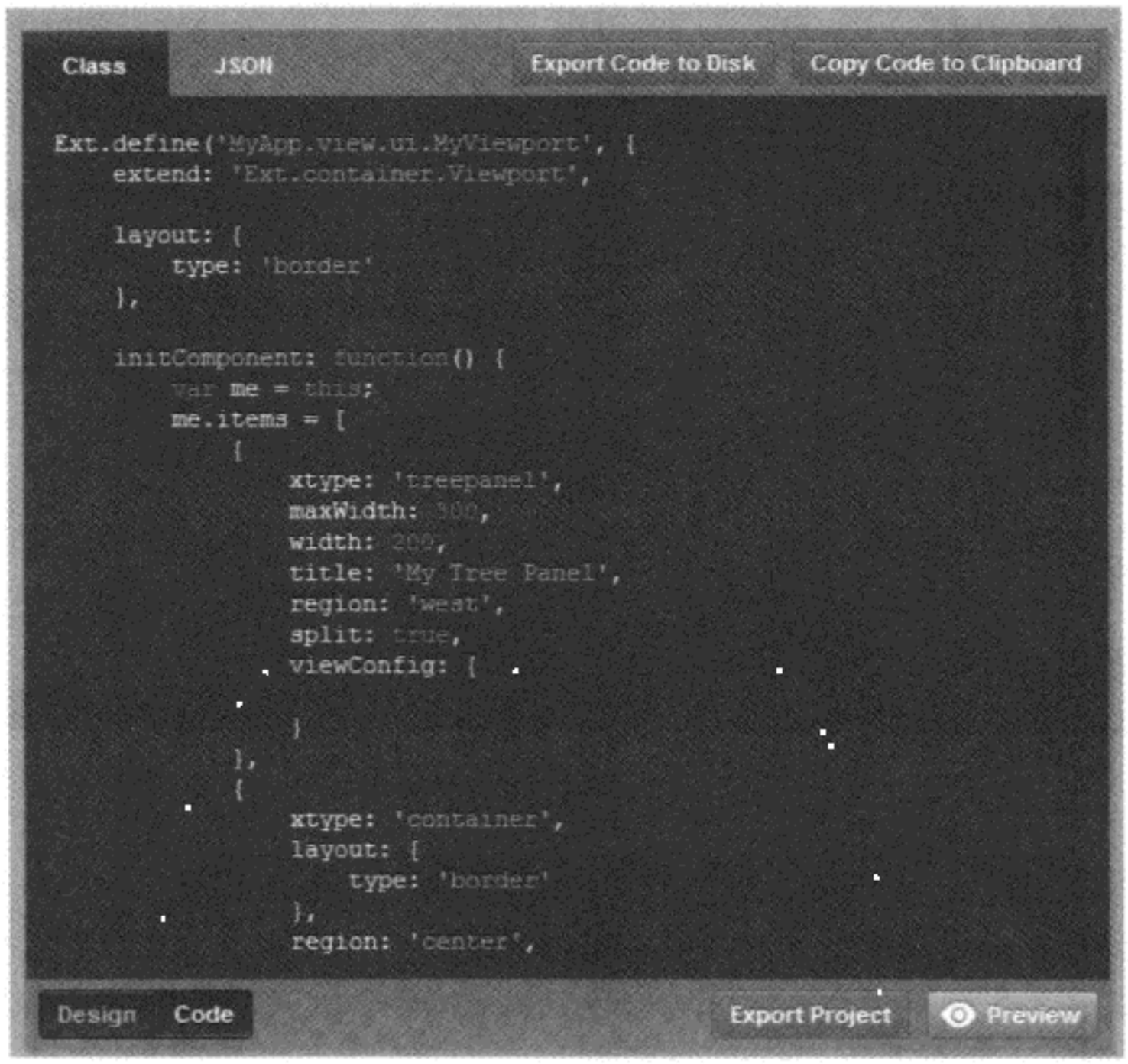


图 1-17　代码窗口

要将 Ext JS 对象的属性和方法提取出来不难，从图 1-18 中 DOM 树中的 Ext 对象结构可以看到，只要遍历一次 Ext 对象就可以取得所有对象的属性和方法。不过，实际上不是那么容易，原因是在 Ext JS 的对象中，有些是实例，有些是函数形式的对象，对于实例，其属性和方法直接在对象内，而对于函数形式的对象，其属性和方法却是在原型内，因而要区别对待。

还有一个重要问题就是命名空间的问题，如 Ext.menu.Item、Ext.data.Store 或 Ext.layout.container.VBox 这些 3 或 4 级的类名。采用递归遍历，判断该对象是 JavaScript 对象还是 Ext JS 方式定义的对象有难度，因而，最好的方式是使用附录中的类名列表，通过它去遍历对象，这样就简单很多了。在列表中，如果对象是实例，使用 Ext 的 isObject 方法，其值会是 true，否则，该对象就是函数式对象，需要实例化才能使用，因而属性和方法在原型内，不过这里要注意可能存在静态方法，为了避免遗漏，还得遍历对象本身的属性和方法。

Ext 对象本身需要另外加进去，主要原因是它不同于其他对象那样可直接通过对象本身或原型提取属性和方法，需要把在类列表中的类排除出去，这个可通过对象或函数是否存在“$className”属性解决。

还有对象的别名问题，这个可直接从 ClassManager 对象的 maps 对象中的 alternateToName 属性中取得，将别名直接指向对象 VS 就认识了。

最后的问题是如何将对象输出到页面，这个通过 encode 方法就可以自动转了。

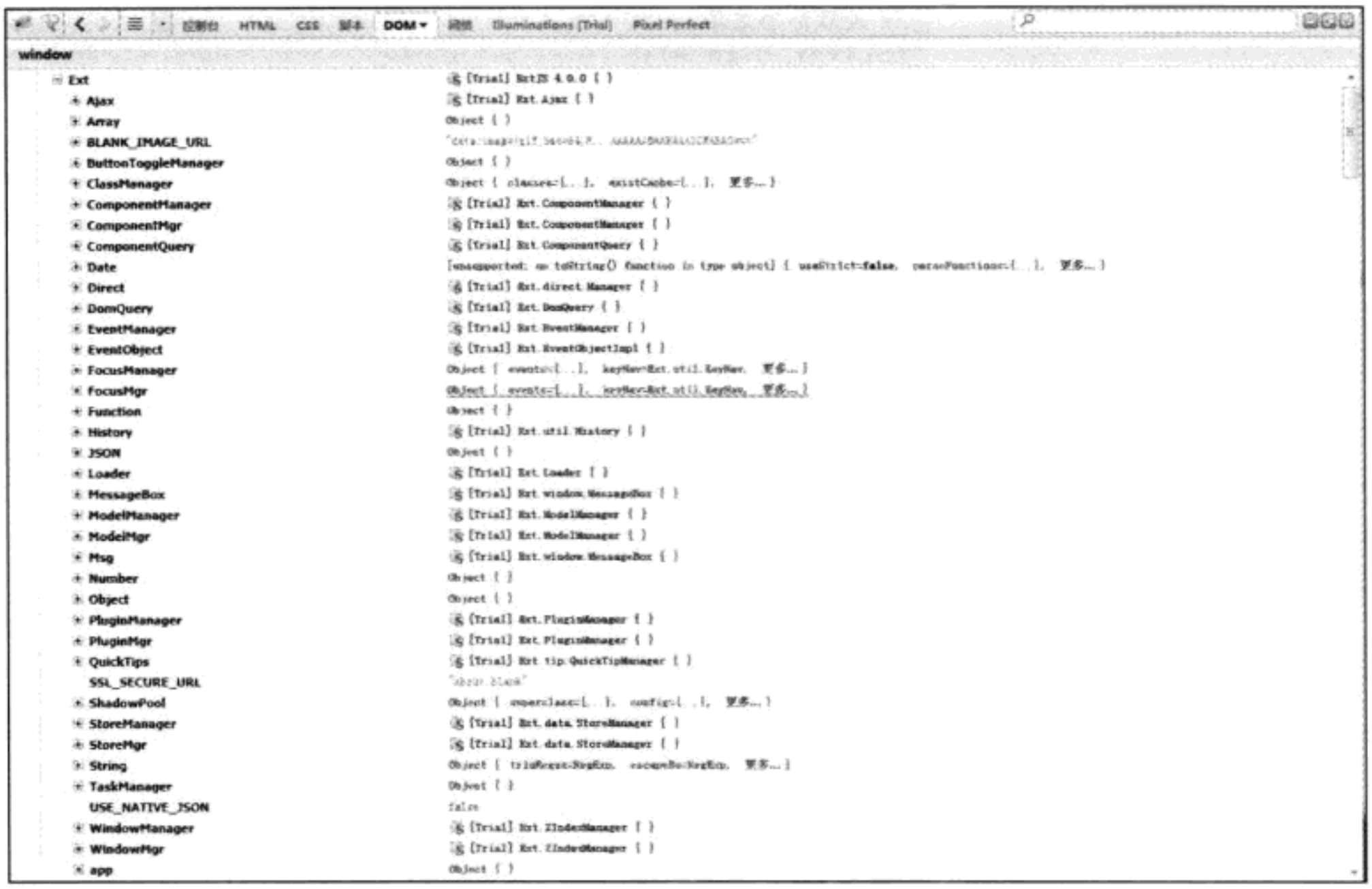


图 1-18　DOM 树中的 Ext 对象

命名空间的问题可通过 ns 方法解决，它会自动根据类名创建对象，只要别和 Ext 对象发生冲突就行了。

目标明确后，现在可以开始工作，创建一个名称为 VS-Ext.html 的文件，然后加入示例 1-1 所示的代码。

示例 1-1　VS-Ext.html 文件代码

```
    <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/
strict.dtd">
    <html>
    <head>
        <title>第 1 章 在 Visual Studio 中实现智能提示 </title>
         <link rel="stylesheet" type="text/css" href="../Ext4/resources/css/ext-all.
css"/>
        <script type="text/javascript" src="../Ext4/bootstrap.js"></script>
        <script type="text/javascript" src="class.js"></script>
</head>
<body>
   <script type="text/javascript">
           Ext.onReady(function(){

                   // 建立命名管道
                   Ext.ns("VS.Ext");

                   var processPrototype=function(s,d){
```

```
                    // 处理静态方法
                    for(var c in s){
                            if(s.hasOwnProperty(c)){
                                    d[c]="";
                            }
                    }
                    // 处理原型
                    for(var c in s.prototype){
                            d[c]="";
                    }
                    }

                    // 处理实例
                    var processInst=function(s,d){
                            var i=0;
                            for(var c in s){
                                    d[c]="";
                            }
                    }


// 处理 Ext 对象
for(var c in Ext){
        if(Ext.hasOwnProperty(c)){
                var p=Ext[c];
                if(Ext.isObject(p)){
                        if(!p["$className"]){
if( ["buildSettings","versions","lastRegisteredVersion"].indexOf(c)>=0 ){
        VS.Ext[c]=p;
                                }
                        }
                }else if(Ext.isFunction(p)){
                        if(!p["$className"]){
                                VS.Ext[c]="";
                        }

                }else{
                        VS.Ext[c]="";
                }
        }
}

        // 枚举对象
        Ext.Array.each(classList,function(classname){
Ext.ns("VS."+classname);
var d=classname.split("."),
        sobj=Ext[d[1]],
        dobj=VS.Ext[d[1]];
if(d.length >= 3){
        sobj=sobj[d[2]],
        dobj=dobj[d[2]];
}
if(d.length == 4){
        sobj=sobj[d[3]];
        dobj=dobj[d[3]];
```

```
}
if(sobj && dobj){
        if(Ext.isObject(sobj)){
                processInst(sobj,dobj);
        }else{
                if(sobj.prototype){
                        processPrototype(sobj,dobj);
                }
        }
}
    })

var html="Ext="+Ext.encode(VS.Ext)+"<br/>";
// 处理别名
for(var c in Ext.ClassManager.maps.alternateToName){
html+=c+"="+Ext.ClassManager.maps.alternateToName[c]+"</br>";
}
    Ext.getBody().dom.innerHTML=html;
})
    </script>
</body>
</html>
```

代码中，先创建新的命名空间“VS.Ext”，在其内放置对象的属性和方法。

函数 processPrototype 用于处理非实例的对象，第一个循环主要是遍历静态属性和方法。第二个循环用于遍历原型的属性和方法。

函数 processInst 用于遍历实例的属性和方法，在这里不能加 hasOwnProperty 方法检测属性或方法是不是它拥有的，不然会找不到属性和方法，除非是在实例创建后加的属性和方法。

接着处理 Ext 对象，如果属性指向的是对象，还要排除 grid、form 等对象干扰，这个要自己查看一下 Ext 的源代码，做一下处理。

接下来是枚举 class.js 中的类，先通过 ns 方法生成类的命名空间，然后根据小数点拆分一下类名，取得源对象和目的对象后根据 isObject 确认对象是实例还是非实例，进行相应的处理。

最后是把 VS 对象内的 Ext 对象和 ClassManager 对象内的别名对照表转换为文本，输出到页面。

在浏览器中打开页面，将看到一个 JSON 格式的输出文本，全选该文本，然后复制到一个名称为 Ext.js 的文件内，这样，通过该文件就可在 VS 内实现智能提示了。

要使用 Ext.js，应首先将文件复制到项目中，然后根据编辑的文件类型采用不同的方法。

如果是单独的脚本（扩展名为 js）文件，在脚本文件头部加入以下语句：

```
/// <reference path="[相对路径]/Ext.js" />
```

这里一定要注意路径，如果要编辑的脚本文件与 Ext.js 文件路径相同，语句为：

```
/// <reference path="Ext.js" />
```

最简单直接的方法，就是在解决方案资源管理器中把 Ext.js 文件直接拖动到编辑的脚本

文件中，这样会自动生成该语句。

有了该语句后，输入“Ext.”就可以看到如图 1-19 所示的智能提示了。

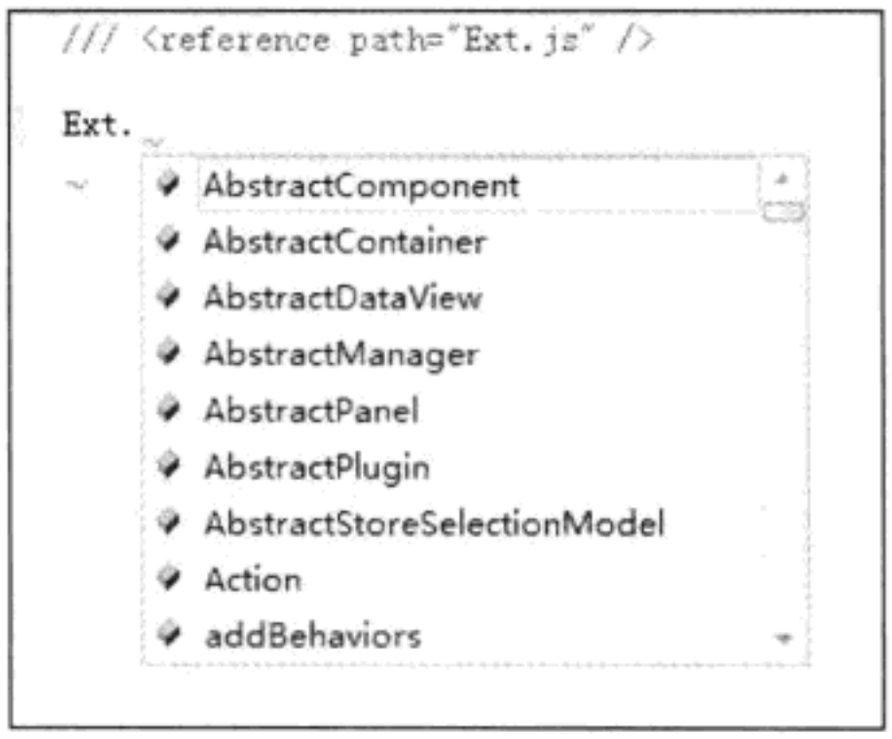


图 1-19　智能提示显示结果

如果是 HTML 文档，只要将 Ext.js 的引用加入文档就行了，**一定要将该文件放在 Ext JS 库文件的前面，这样才不会影响调试。文件编辑完成后，记得将该引用清除。**

1.4.3　Spket

Spket IDE 是一个功能强大的 JavaScript 和 XML 编辑器，是理想的 Ext JS 开发工具。不过，有点遗憾的是，它对非商业使用免费，如果是商业使用，则需要购买许可证。

可登录 http://spket.com/download.html 下载 Spket，本书使用的版本是 1.6.18。下载的是一个文件名为 spket-1.6.18.jar 的 jar 文件，需要 Java 环境支撑。所以在安装 Spket 前，需要在机器上安装并运行 Java 1.5 或更高版本。

Java 安装好以后，运行 spket-1.6.18.jar 文件将看到如图 1-20 所示的窗口。

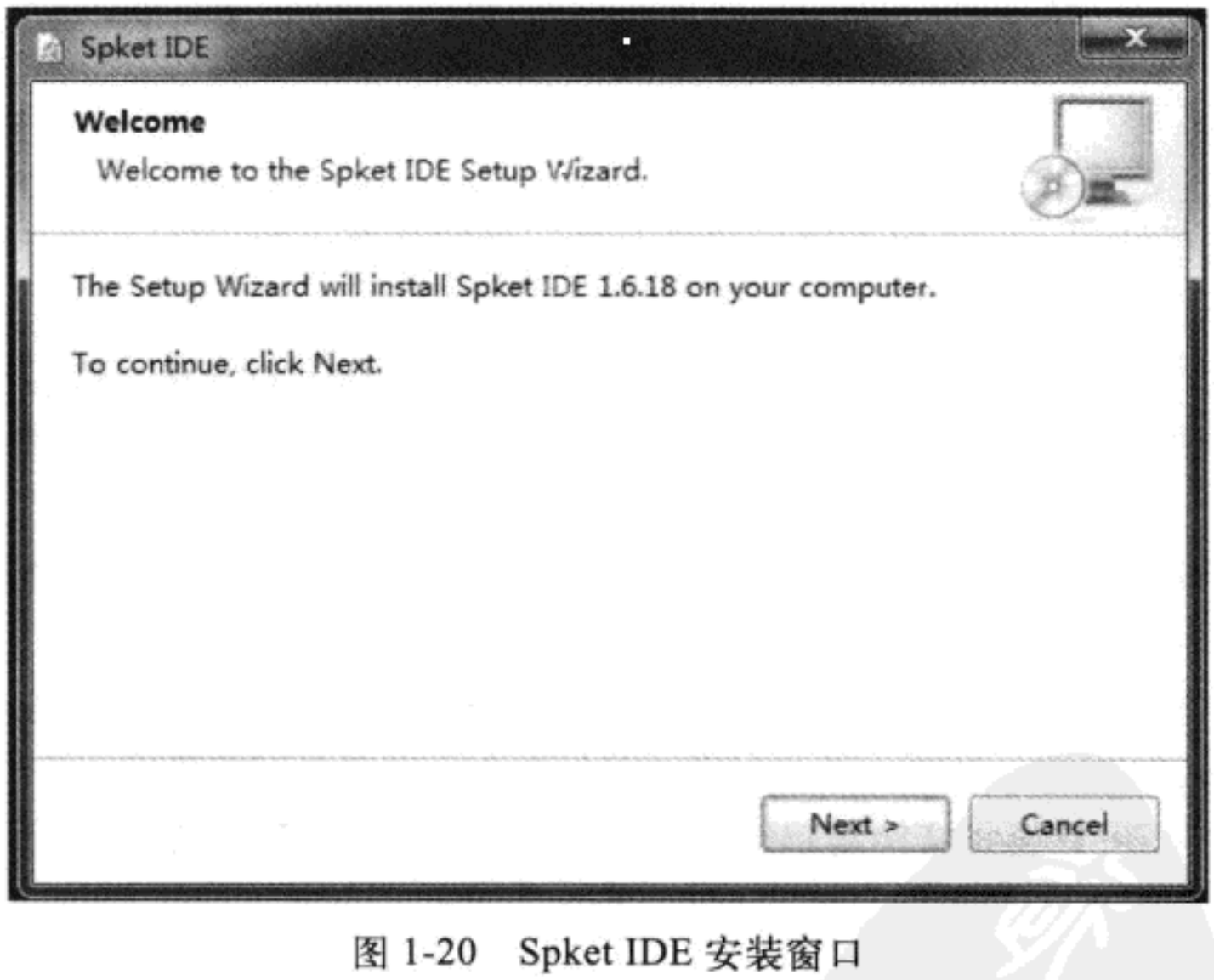


图 1-20　Spket IDE 安装窗口

单击“Next”按钮将看到如图 1-21 所示安装类型选择窗口。

在安装类型选择窗口中，可选择作为独立的应用程序安装，还是作为 Eclipse 的插件安装。不做任何修改，单击“Next”按钮将看到如图 1-22 所示的安装路径选择窗口。

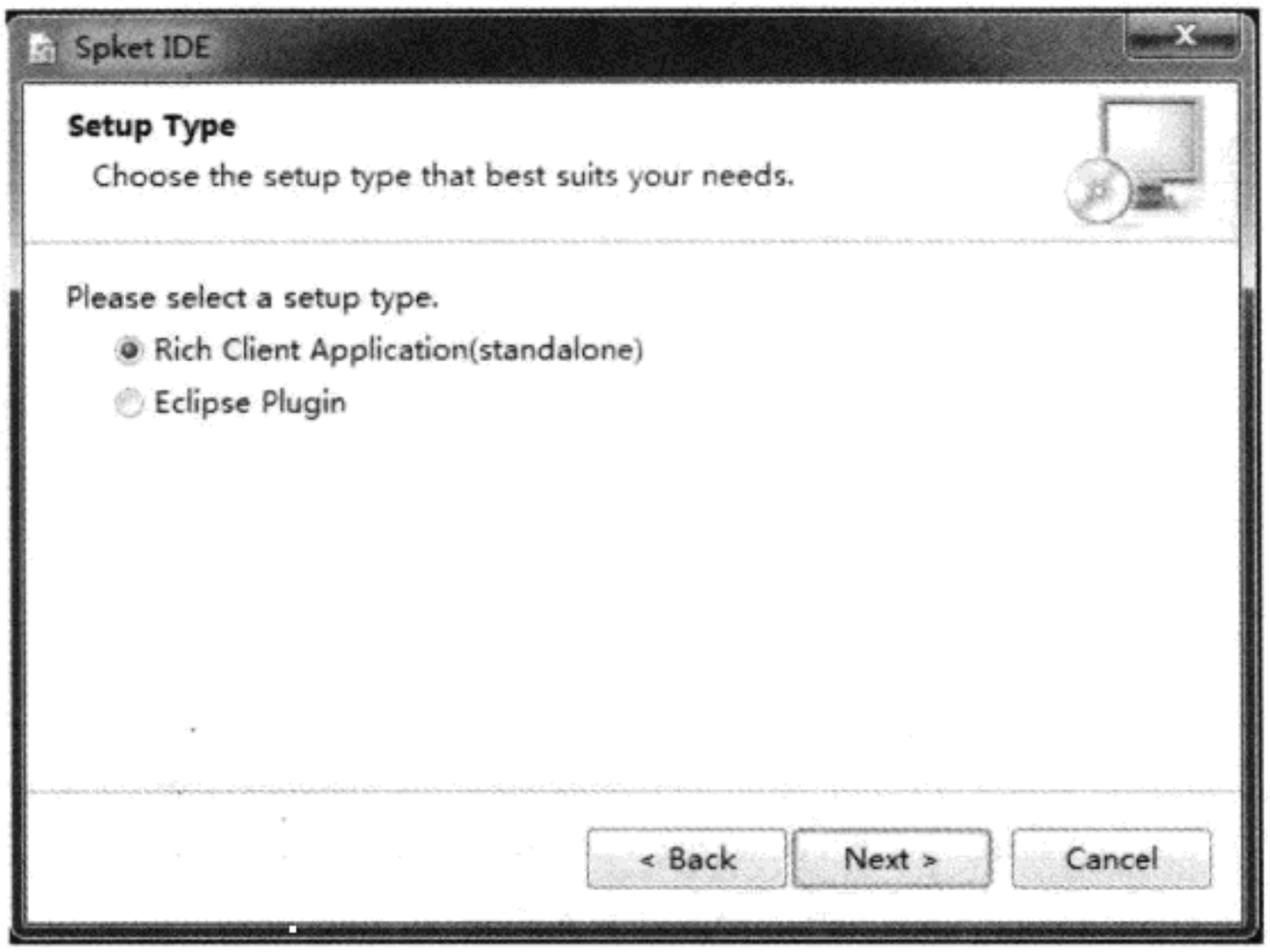


图 1-21　安装类型选择窗口

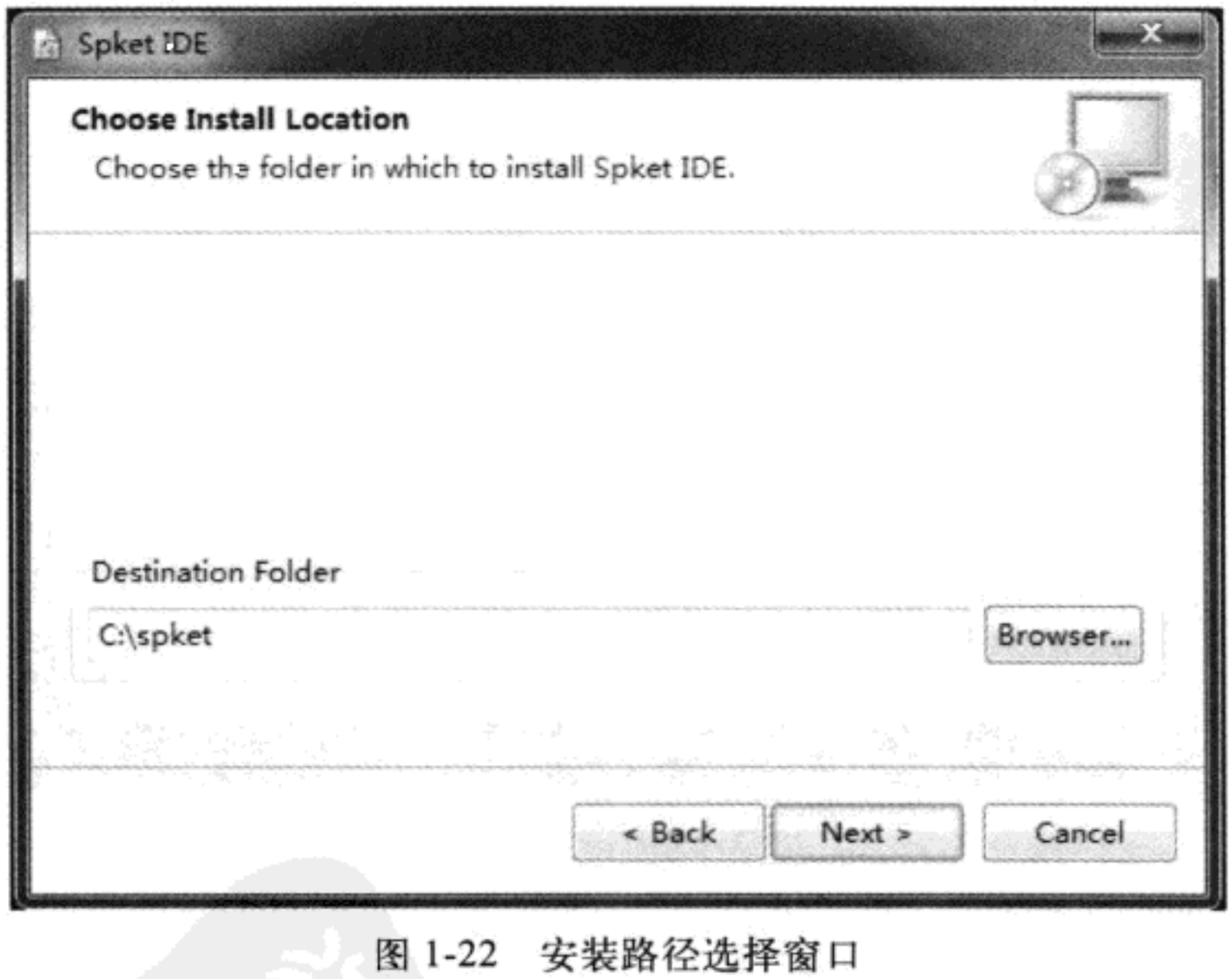


图 1-22　安装路径选择窗口

不修改路径，直接单击“Next”按钮进入如图 1-23 所示的预备安装窗口。

单击“Install”按钮等待安装完成。最后单击“Finish”按钮完成安装。

在安装目录下允许“spket.exe”文件，即可打开 Spket IDE。程序运行后会出现如图 1-24 所示的工作目录设置窗口。

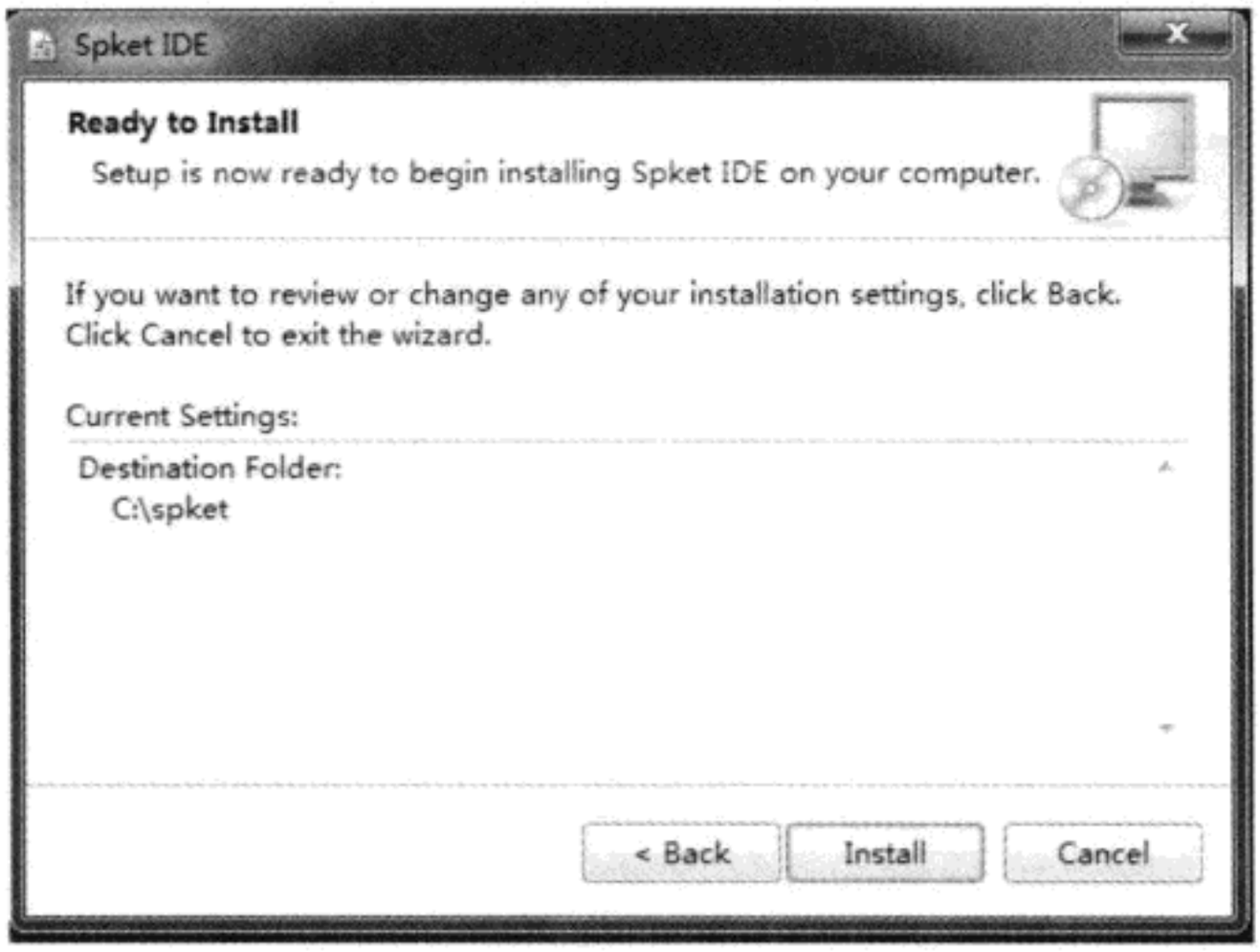


图 1-23　预备安装窗口

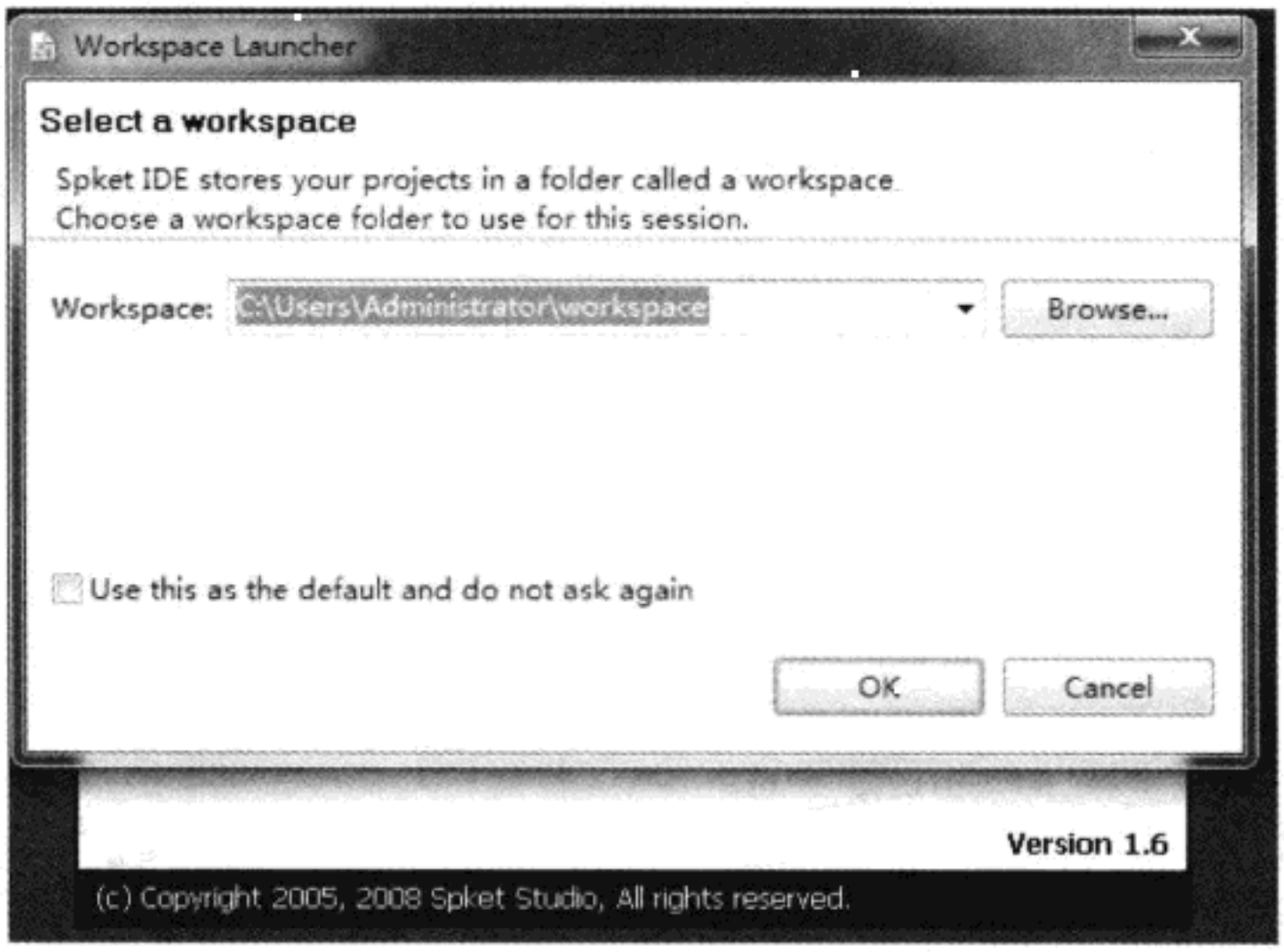


图 1-24　工作目录选择窗口

如果不想每次打开都选择一次工作目录，可把左下角的选择框选上。

进入如图 1-25 所示的 Spket IDE 主窗口后，要实现 Ext JS 的智能提示，需要做一些配置。

在 Window 菜单下选择 Preferences 子菜单，在弹出窗口中展开 Spket 节点，然后选择“JavaScript Profiles”节点，将看到如图 1-26 所示的设置窗口。单击右边的“New”按钮，在弹出窗口的输入框中输入“Ext JS 4”，单击“OK”按钮关闭窗口。在中间的列表窗口会多出一个“Ext JS 4”的列表项，选择该项，然后单击“Add Library”按钮，在弹出窗口中的下拉列表框中选择“Ext JS”并单击“OK”按钮返回。这时，在“Ext JS 4”列表项下会多出一个

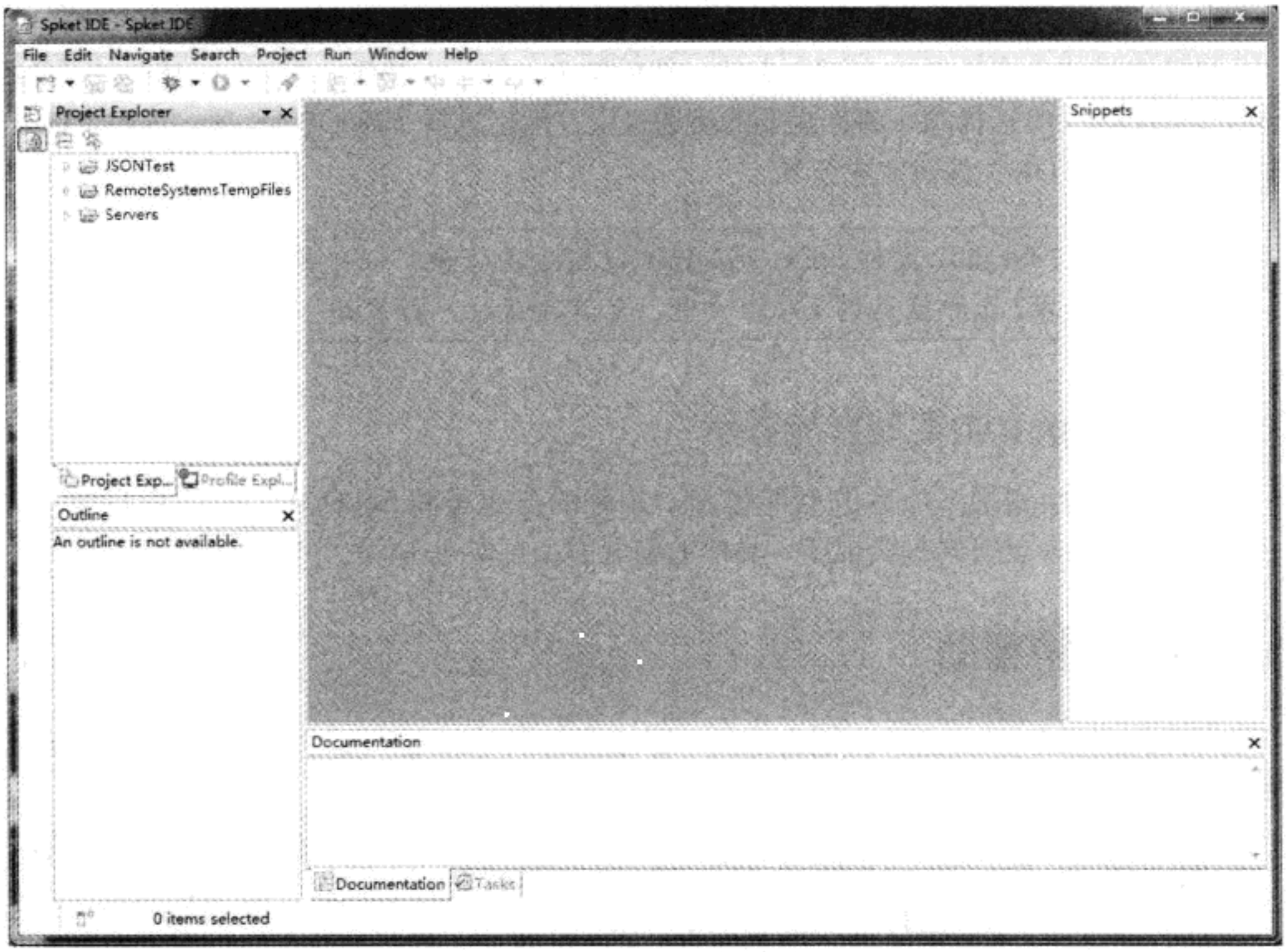


图 1-25　Spket IDE 主窗口

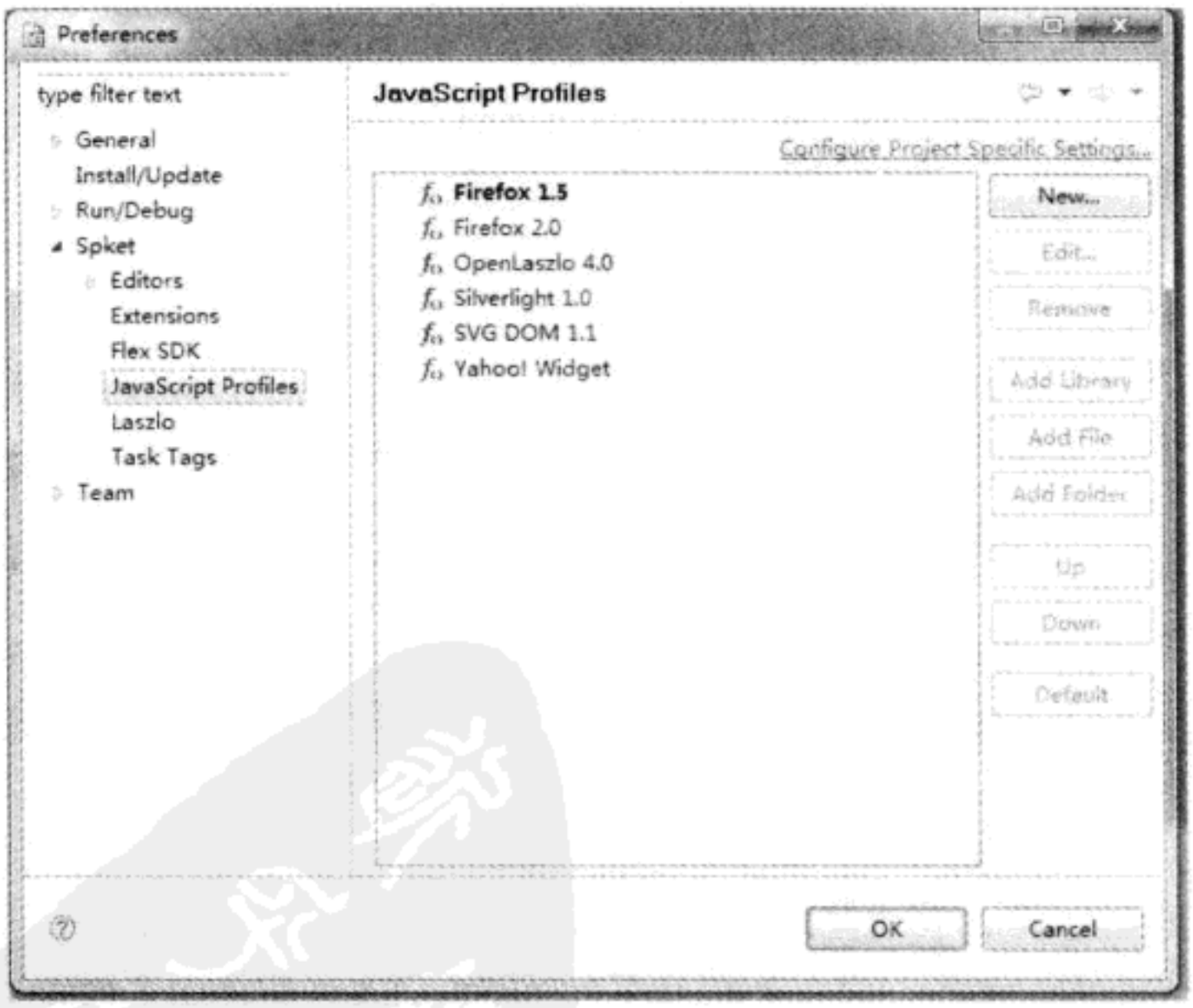


图 1-26　设置窗口

“Ext JS”的子选择项，选择该项并单击“Add File”按钮，在弹出的文件选择窗口中，在Ext JS目录中寻找并选择“ext.jsb2”文件，然后单击“打开”按钮。在“Ext JS”的子选择项下会增加Ext JS的类选择项，将你需要的类选择上，然后选择“Ext JS 4”项，单击“Default”按钮。最后单击“OK”按钮完成配置。

注意 如果找不到ext.jsb2文件（Ext JS4好像以新方法提供，一般情况下没有该文件），也可以使用1.4.2节中生成的Ext.js文件，笔者测试过，效果是一样的。

1.4.4 在Eclipse中实现智能提示

要在Eclipse中实现Ext JS的智能提示，最好的方式是将Spket IDE安装为Eclipse的插件，这样通过Spket就可轻松实现了。具体的配置方法可参考1.4.3小节。

1.5 如何获得帮助

学习和使用Ext JS或多或少都会遇到一时难以解决的问题，这时候就需要寻求帮助。寻求帮助的方法如下。

- 在线API文档：笔者认为，要提高自己的编程水平，很重要的一点就是经常看和查API文档。很多初学者习惯碰到问题就去论坛或者QQ群问人，这实在不是一个好的办法，因为缺乏思考过程。笔者认为，看API绝对可以提高你解决问题和寻找解决问题的能力，因为看和找的过程就是一个学习的过程。而去问人的过程，基本是一个复制过程，至于为什么要这样？为什么会这样？完全会被忽略。新的API还包含了不少使用指南，值得仔细研读。最好的办法是在自己的系统架一个Web服务器，然后把API架在服务器上，非常方便。
- Ext JS包中的示例：Ext JS包中的示例提供了大部分开发中的样板和组件的使用方法。只要花点时间研究一下示例的代码，绝对会受益匪浅。笔者就是这样学习的。
- 官方网站的学习中心：在官方网站的学习提供有很多关于Ext JS的学习文章，若仔细看一下，会有醍醐灌顶的感觉。官方网站的学习中心已经更新，是一个学习Ext JS非常好的地方，只是必须懂英文。官方网站学习中心的网址是http://www.sencha.com/learn/Ext JS/?4x。
- 官方论坛：官方论坛绝对是一个好去处，尤其是要寻找合适的组件的时候。更重要的是，里面有很多Ext JS牛人，总会有一个能给你提供必要的帮助。多去论坛逛逛，会有不一样的感觉喔。你英文不好？没关系，多用用翻译软件就好了。官方论坛的地址是http://www.sencha.com/forum/。
- 中文论坛及QQ群：在这些地方可能会有人帮助你立刻解决问题，但是对你开发水平的提高帮助不大。笔者的建议是在万不得已的情况才使用这个杀手锏。

- 以上一系列方法还没能解决你的问题？这时候，要思考一下你的设计了。

1.6　本章小结

本章主要介绍了学习 Ext JS 需要掌握的知识并对 Ext JS 4 进行了简单介绍，和大家分享了一些笔者学习 Ext JS 的经验，为下一章开始进入 Ext JS 奠定基础。在进入下一章之前，笔者建议大家架设好本地在线 API 文档，以便随时查阅。

第 2 章　从“Hello World”开始

“Hello World”几乎已经成为所有开发类图书的必用案例，本书也不能免俗。本章将通过编写“Hello World”程序来让大家对如何使用 Ext JS 进行开发有初步的了解，如 Ext JS 代码是如何运行的、代码书写风格是怎样的、如何实现本地化等。

2.1　获取 Ext JS 4

要下载 Ext JS 4，可访问地址：http://www.sencha.com/products/Ext JS/download/。

本书介绍的版本是 4.1，所以请下载 4.1 Beta 1 这个版本。

文件解压后，在根目录会看到以下目录和文件：

- builds 目录：包含沙盒、Core 和 foundation 3 个压缩好的脚本文件及其调试（debug）文件。
- docs 目录：API 文档目录。
- examples 目录：示例文件目录。
- locale：本地化文件目录。
- pkgs 目录：独立的包文件。
- resources 目录：Ext JS 4 的资源目录，包含了样式定义文件及其需要的图片文件。
- src 目录：Ext JS 4 的源文件目录。
- welcome 目录：Ext JS 欢迎页面所需的资源目录。
- bootstrap.js 文件：会根据 url 地址自动加载 ext-all.js 或 ext-all-debug.js 文件。此文本只在正式发布版本中有，可在 4.0.7 中找到。
- ext.js 文件：Ext Core 的库文件。
- ext-all.js 文件：Ext JS 的库文件（压缩）。
- ext-all-debug.js 文件：调试应用程序时使用的 Ext JS 的库文件（不带注释）。
- ext-all-debug-w-comments.js 文件：调试应用程序时使用的 Ext JS 的库文件（带注释）。
- ext-all-dev.js：用于诊断布局问题的 Ext JS 库文件。
- ext-all-dev-w-comments.js：带注释的诊断库文件。
- ext-neptune.js：使用海王星主题的 Ext JS 库文件。
- ext-debug.js 文件：调试使用 Ext Core 开发的程序时的 Ext Core 库文件。
- index.html 文件：Ext JS 4 欢迎页面。
- license.txt 文件：协议说明文本。
- release-notes.html 文件：Ext JS 4 版本发布说明。

2.2　配置使用 Ext JS 库

要使用 Ext JS，首先要做的是将 Ext JS 包里的 resources 目录、bootstrap.js 文件、ext-all.js 文件和 ext-all-debug.js 文件复制到项目目录。接着在页面中 head 标记部分添加 CSS 和脚本文件的引用：

```
<link rel="stylesheet" type="text/css" href="path/resources/css/ext-all.css"/>
<script type="text/javascript" src="path/bootstrap.js"></script>
```

要注意代码中的 path 是 CSS 文件或脚本文件相对于页面文件的路径。例如，页面文件在根目录，resoureces 目录在根目录下，bootstrap.js 在 js 目录下，就可这样写：

```
<link rel="stylesheet" type="text/css" href="resources/css/ext-all.css"/>
    <script type="text/javascript" src="js/bootstrap.js"></script>
```

如果熟悉 Ext JS 2 或 Ext JS 3 的，会发现 Ext JS 4 不是直接加载 ext-all.js 或 ext-all-debug.js，而是加载了 bootstrap.js，这有什么好处呢？先看看 bootstrap.js 的源代码：

```
(function() {

    var scripts = document.getElementsByTagName('script'),
        localhostTests = [
            /^localhost$/,                    /\b(25[0-5]|2[0-4][0-9]|[01]?[0-9]
                [0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4]
                [0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
                (:\d{1,5})?\b/ // IP v4
        ],
        host = window.location.hostname,
        isDevelopment = null,
        queryString = window.location.search,
        test, path, i, ln, scriptSrc, match;
    for (i = 0, ln = scripts.length; i < ln; i++) {
        scriptSrc = scripts[i].src;
        match = scriptSrc.match(/bootstrap\.js$/);
        if (match) {
            path = scriptSrc.substring(0, scriptSrc.length - match[0].length);
            break;
        }
    }

    if (queryString.match('(\\?|&)debug') !== null) {
        isDevelopment = true;
    }
    else if (queryString.match('(\\?|&)nodebug') !== null) {
        isDevelopment = false;
    }
    if (isDevelopment === null) {
        for (i = 0, ln = localhostTests.length; i < ln; i++) {
            test = localhostTests[i];
            if (host.search(test) !== -1) {
                isDevelopment = true;
                break;
```

```
        }
    }
}

if (isDevelopment === null && window.location.protocol === 'file:') {
    isDevelopment = true;
}

document.write('<script type="text/javascript" src="' + path + 'ext-all' +
    ((isDevelopment) ? '-debug' : '') + '.js"></script>');
})();
```

代码首先使用 getElementsByTagName 获取页面中所有带有 script 标记的元素，然后从中找出带有 bootstrap.js 的标记，最后将 bootstrap.js 的相对路径取出并保存在变量 path 中。

接着判断 url 的参数中是否有“debug”字符，例如，出现 http://localhost/index.html?debug，则设置 isDevelopment 为 true。否则检测是否有“nodebug”字符，如果有，设置为 false。如果以上两个条件都不符合，isDevelopment 还是初始值 null，就要判断 url 的域名是否是“localhost”或 IPv4 地址，如果是，则 isDevelopment 设置为 true。

如果 isDevelopment 是 null 且当前的 url 的协议是“file：”，则将 isDevelopment 设置为 true。如果 isDevelopment 为 true 时，则加载 ext-all-debug.js 文件，否则加载 ext-all.js 文件。

通过 bootstrap.js，可自动控制加载 ext-all-debug.js 文件或 ext-all.js 文件，不用我们自己去修改 script 标记，非常方便。但如果你觉得自己修改方便，也可以使用 Ext JS 旧版的方式加载脚本文件。不过 bootstrap.js 有个缺点，就是把所有 IPv4 地址都划归为使用调试文件的范围，不太符合国内的情况。因为在内网，应用也多半是使用 IP 地址访问的。不过，根据自己的情况去修改 bootstrap.js 也是很方便的。

如果想动态加载 Ext JS 库，就需要在页面中先加载 builds 目录下的 Ext-core.js 或 Ext.-core-debug.js 文件，然后在 onReady 函数外添加以下代码：

```
Ext.Loader.setConfig({enabled: true});
Ext.Loader.setPath({// 加载路径配置对象 });
Ext.require([
    'Ext.grid.*',
   ...
   // 需要加载的库
]);
Ext.onReady(function(){
    // 代码
});
```

Loader 对象的 setConfig 方法设置的 enabled 属性的作用是，开启动态加载的依赖加载功能。该功能的作用是在加载类库文件的时候，根据其依赖情况加载它需要的类库，其默认值是 false，这是因为一般情况下 Ext JS 的库文件都是一次性全部加载的，很少用到动态加载。试想一下，Ext JS 类库有 200 多个文件，在无法预知要加载多少类库的情况下，不断地向服务器请求 100 多个类库甚至全部 200 多个类库，那是很吓人的，不但增加了服务器的负担，客户也要一直等待类库的加载，这不是好的方法，所以不推荐使用此方法。

Loader 对象的 setPath 方法是用来设置加载路径的，这在 4.5 节中会讲述。接着就是使用 Ext.require 方法请求加载类库了。

2.3　编写“Hello World”程序

明白了 Ext JS 4 配置后，就可编写“Hello World”程序了。新建一个 HTML 文件 Hello_World.html，加入如代码清单 2-1 所示的代码。

代码清单 2-1　Hello World 程序

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/
    strict.dtd">
<html>
<head>
  <title>第 2 章 示例 2-1 Hello World</title>
  <link rel="stylesheet" type="text/css" href="../Ext4/resources/css/ext-all.
      css"/>
  <script type="text/javascript" src="../Ext4/bootstrap.js"></script>
  <script type="text/javascript" src="../Ext4/locale/Ext-lang-zh_CN.js"></script>
</head>
<body>
   <script type="text/javascript">
          Ext.onReady(function(){
                if(Ext.BLANK_IMAGE_URL.substr(0,4)!="data"){
                      Ext.BLANK_IMAGE_URL="./images/s.gif";
                }
                //Ext.create('Ext.Viewport',{
                new Ext.Viewport({
                      layout:'fit',
                      items:[{
                            xtype:"panel",
                            title:"欢迎",
                            html:"<h1  style='text-align:center;font-
                                size:60px;font-weight:bold;'>Hello  World</
                                h1>"
                      }]
                });
          })
   </script>
</body>
</html>
```

代码主要包括两部分：第一部分是在 head 部分配置了 Ext JS 库文件和样式文件的引用；第二部分是在 Ext.onReady 函数中使用 Viewport 定义界面，并在一个面板内显示“Hello World”。

在浏览器中打开页面将看到如图 2-1 所示的结果。

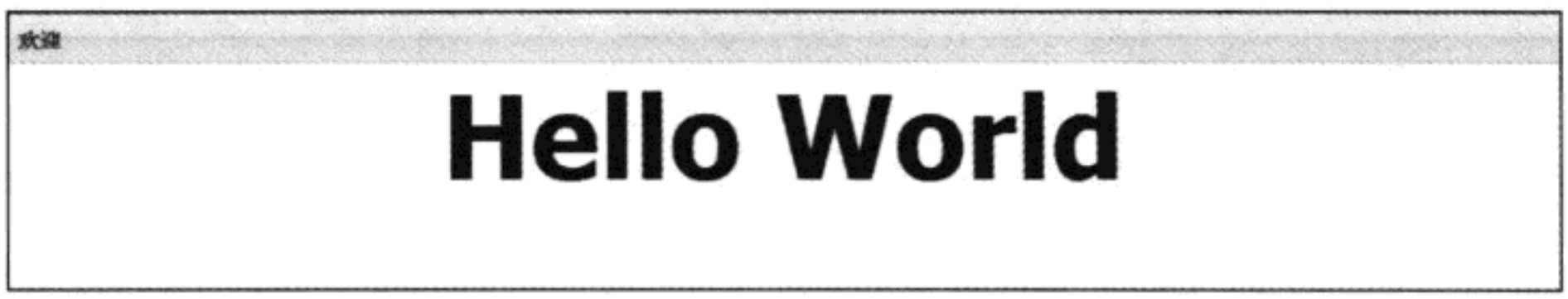


图 2-1　Hello World 程序显示结果

2.4　关于 Ext.onReady

代码为什么写在 Ext.onReady 中，而不是在 body 中添加一个 onload 事件并在 onload 事件中运行呢？主要原因是 Ext.onReady 在 DOM 模型加载完毕后即可进行操作，而无需像 onload 事件那样，等待页面的所有资源都加载完毕后才进行操作，尤其是在页面有大图片这类资源的时候。下面我们来看看 Ext.onReady 是如何做到这点的。

在 Loader.js 文件可以找到 Ext.onReady 的定义：

```
Ext.onReady = function(fn, scope, options) {
    Loader.onReady(fn, scope, true, options);
};
```

在这里调用了 Loader 对象的 onReady 方法，在 Loader.js 中可找到如下定义：

```
onReady: function(fn, scope, withDomReady, options) {
    var oldFn;

    if (withDomReady !== false && Ext.onDocumentReady) {
        oldFn = fn;

        fn = function() {
            Ext.onDocumentReady(oldFn, scope, options);
        };
    }

    fn.call(scope);
},
```

在上面的代码中，因为调用时 withDomReady 为 true，所以只需判断 Ext.onDocumentReady 是否存在，如果存在，就建立一个匿名函数 fn，准备执行 Ext.onDocumentReady 方法。最后是调用函数 fn，执行 Ext.onDocumentReady。

在 EventManger.js 中可找到 Ext.onDocumentReady 的定义：

```
onDocumentReady: function(fn, scope, options){
  options = options || {};
  var me = Ext.EventManager,
      readyEvent = me.readyEvent;

  options.single = true;
```

```
    readyEvent.addListener(fn, scope, options);
    if (Ext.isReady) {
        readyEvent.fire();
    } else if (document.readyState == 'complete') {
        me.fireDocReady();
    } else {
        me.bindReadyEvent();
    }
},
```

在上面的代码中，readyEvent 是 Ext.util.Event 的实例，options.single 的作用是规定 ready-Event 事件只执行一次。接着将函数添加到 Event 实例内的监听事件列表中，最后判断 DOM 模型是否已加载完成。如果已加载完成，则调用 fire 方法依次执行监听事件列表中的函数。这样做的目的是：当存在多个 onReady 方法时，能保证所有的函数都能执行。如果还没有加载完成，而 document 对象的 readyState 属性为“complete”，表示文档其实已经加载完成了，但是没有设置 isReady 属性为 true，那么可调用 fireDocReady 方法，其代码如下：

```
fireDocReady: function(){
    var me = Ext.EventManager;

    if (!Ext.isReady) {
        Ext.isReady = true;

        if (document.addEventListener) {
            document.removeEventListener('DOMContentLoaded', me.fireDocReady,
                false);
            window.removeEventListener('load', me.fireDocReady, false);
        } else {
            if (me.readyTimeout !== null) {
                clearTimeout(me.readyTimeout);
            }
            if (me.hasOnReadyStateChange) {
                document.detachEvent('onreadystatechange', me.checkReadyState);
            }
            window.detachEvent('onload', me.fireDocReady);
        }

        Ext.supports.init();
        me.onWindowUnload();
        me.readyEvent.fire();
    }
},
```

在上面的代码中，如果 isReady 不是 true，则将其设置为 true，然后移除文档的监听事件。首先调用 Ext.supports 的 init 方法检测当前运行环境的信息；然后调用 onWindowUnload 方法为文档绑定 unload 事件，触发后会删除页面的所有元素；最后再调用 readyEvent 的 fire 方法，开始执行我们定义的代码。

如果文档还没有加载完成，则执行 bindReadyEvent 方法，其代码如下：

```
bindReadyEvent: function(){
    var me = Ext.EventManager;
    if (me.hasBoundOnReady) {
        return;
    }

    if (document.addEventListener) {
        document.addEventListener('DOMContentLoaded', me.fireDocReady, false);
        window.addEventListener('load', me.fireDocReady, false);
    } else {
        if (!me.checkReadyState()) {
            document.attachEvent('onreadystatechange', me.checkReadyState);
            me.hasOnReadyStateChange = true;
        }
        window.attachEvent('onload', me.fireDocReady, false);
    }
    me.hasBoundOnReady = true;
},
```

看懂以上代码就应该很清楚整个执行过程了。在代码中，如果没有在页面中绑定监听事件，则绑定事件，非 IE 浏览器是绑定“DOMContentLoaded”事件，IE 是绑定 onload 事件。对于旧版本的 IE，会调用 checkReadyState 方法检查页面是否准备好，因为旧版本 IE 只能使用替代办法检查 DOMContentLoaded[⊖]事件。事件触发后执行 fireDocReady 方法。

2.5 关于 Ext.BLANK_IMAGE_URL

在 Ext-more.js 文件中可找到 BLANK_IMAGE_URL 属性的默认值和定义。其默认值是：

```
BLANK_IMAGE_URL : '
    AAABAAEAAAICRAEAOw=='
```

定义是：

```
BLANK_IMAGE_URL : (isIE6 || isIE7) ? 'http:/' + '/www.sencha.com/s.gif' :
    '
    EAOw=='
```

代码中以“data”开头的数据代表一个 1*1 像素的空白透明图片。因为 IE 6 和 IE 8 不支持这样格式的图片数据，所以必须为它指定一个图片路径，默认指向 http:///www.sencha.com/s.gif。

这个图片有什么用呢？在菜单项的源代码（Item.js）中可找到菜单项的渲染模版代码：

```
renderTpl: [
  '<tpl if="plain">',
      '{text}',
  '</tpl>',
  '<tpl if="!plain">',
      '<a class="' + Ext.baseCSSPrefix + 'menu-item-link" href="{href}"
```

⊖ 具体信息可访问 http://javascript.nwbox.com/IEContentLoaded/。

```
<tpl if="hrefTarget">target="{hrefTarget}"</tpl> hidefocus="true"
unselectable="on">',
'<img src="{icon}" class="' + Ext.baseCSSPrefix + 'menu-item-icon
    {iconCls}" />',
'<span class="' + Ext.baseCSSPrefix + 'menu-item-text" <tpl
    if="menu">style="margin-right: 17px;"</tpl> >{text}</span>',
'<tpl if="menu">',
    '<img src="' + Ext.BLANK_IMAGE_URL + '" class="' + Ext.baseCSSPrefix
        + 'menu-item-arrow" />',
        '</tpl>',
      '</a>',
  '</tpl>'
],
```

在上面的代码中，Ext.BLANK_IMAGE_URL 被用作显示图片。哈哈，有点想不通为什么用 1 个 1*1 像素的空白图片来作显示图片吧。别着急，看看它的 CSS 代码就清楚了。在 ext-all-debug.css 文件可找到有关“menu-item-arrow”的代码：

```
.x-menu-item-arrow {
  position: absolute;
  width: 12px;
  height: 9px;
  top: 9px;
  right: 0px;
  background: url('../themes/images/default/menu/menu-parent.gif') no-repeat
      center center;
}
```

真相出来了，Ext.BLANK_IMAGE_URL 只是一个占位符，目的是显示背景图片。这样做主要有以下两个目的：

- 在更换主题时如果图片是固定的，有两种方法更新图片，一种是覆盖旧图片，一种是重新定义图片路径。在你固定主题的情况下，覆盖图片的方法是可行的，但如果要由用户自定义主题就不行了。重新定义图片的方法不但增加开发人员工作量，而且也不能实现用户自定义主题功能。使用这种方法，只需要在 CSS 中指定背景图片就行了，既不需要覆盖图片，也不需要去重新定义图片路径，非常方便实用。
- 如果使用 div 标记或 span 标记代替 img 标记，那么在这两个标记外增加链接的时候，因为标记内的内容为空，所以链接就会出现问题。如果在 div 或 span 内加入空格呢？问题依旧一样，而且还会因为空格造成背景图片难于定位。有兴趣的可以自己做一下测试。

明白了 Ext.BLANK_IMAGE_URL 的作用，回头再研究一下 2.3 节的示例中有关 Ext.BLANK_IMAGE_URL 的三行代码。因为当浏览器是 IE 6 或 IE 7 时，图片地址会指向 http:///www.sencha.com/s.gif，如果访问不了 Sencha 网站，就会出现显示问题，所以需要将地址指向本地地址。如果不是 IE 6 或 IE 7，那么使用默认值就可以了。

2.6 关于字体

是不是觉得图 2-1 中的“欢迎”两字有点别扭？在目前所有的 Ext 版本中都存在这个问题。主要是因为在所有版本的 Ext 中，标题这些地方使用的字体大小都是“11px”的，将它们修改为“12px”就行了。修改方法是将你使用的 CSS 文件里的“11px”全部替换为“12px”。例如，你使用的是 ext-all.css 文件，那就将该文件里的“11px”全部替换为“12px”。

2.7 Ext JS 4 语法

1. 配置对象

Ext JS 的基本语法就是使用树状的配置对象来定义界面，其格式如下：

```
{
      config_options1:value1,
    config_options1:value2,
    ...
    config_optionsn:valuen,
    layout:{},
    items:[
   {},// 配置对象
   {}// 配置对象
   ...
    ],
    listeners:{
    // 定义事件（根据需要而定）
         click:function(){},
         dblclick:function(){}
         ...
    }
}
```

格式中从 config_options1、config_options2 到 config_optionsn 都是 API 文档中对象的配置项（config options）。**很多初学者会错误地认为 API 不全，找不到配置项。事实上 API 是完整的，只是有些布局隐含了面板或其他部件，但都在同一配置对象内定义。**例如在使用 Accordion 布局的时候，想消除布局标题栏右边的小图标，需要使用 hideCollapseTool 属性，而该属性是在 Panel 对象里的。在这方面，Ext JS 4 的 API 已经做了调整了，增加了一个其他配置项（Other Configs）的列表。

属性 layout 可以是对象，也可以是字符。该属性表示在当前容器内使用什么布局来填充子控件。如果没有特殊要求，直接使用布局的别名作为值，例如，2.3 节的示例中 Viewport 使用了 Fit 布局来放置子控件。如果有特殊要求，则要使用对象来定义该值。例如，如果使用 Hbox 布局，布局内的子控件需要居中对齐，则定义如下：

```
layout:{
    type:'hbox',
    align:'middle'
}
```

属性 items 是一个数组，可以在里面定义当前控件的子控件，里面可以是 1 个或多个配置项，根据你的需要而定。例如 2.3 节的示例，在 Viewport 下使用了一个 Panel 面板作为其面板。属性 listeners 是一个对象，可以在里面绑定事件，对象的属性就是事件名称，属性值就是要执行的函数。

2. 关于 xtype

在使用 Ext JS 编写应用时，很多时候通过定义 xtype 来指定该位置使用什么组件，例如 2.3 节的示例中的“xtype:"panel"”，这里指定使用面板作为当前位置的组件。这样做的主要目的是简化代码。如果没有 xtype，就得先使用变量指向一个组件，然后将其加入父组件中，或者直接在父组件的定义中加入一堆由 new 关键或 Ext.Create 方法创建的组件。这不但影响了书写代码的效率，还影响了代码的可读性。有 xtype 存在，就不用担心这些问题了。

在某些组件下，会默认其内部组件为某些组件，因而也就不需要书写 xtype 语句。例如，在 2.3 节的示例中把 xtype 去掉，代码也能正常运行，因为面板是 Viewport 内默认的组件。一般来说，没特别声明，使用的都是 Panel 组件。

定义 xtype，一般使用组件的别名。可在 API 文档的组件说明文档的顶部或 Component 对象的说明中找到组件的 xtype 值。

3. 使用 new 关键字创建对象

在 Ext JS 4 版本之前，一直使用 new 关键字创建对象，其语法如下：

```
new classname([config])
```

其中 calssname 是指类名；config 是可选参数，为类的配置对象（config options），类型为 JSON 对象。在 2.3 节的示例中，Ext.Viewport 就是使用该方法创建的。

4. 使用 Ext.create 方法创建对象

Ext.create 方法是新增的创建对象的方法，其语法如下：

```
Ext.create(classname,[config])
```

其中 classname 可以是类的全名、别名或备用名；config 是可选参数，为类的配置对象（config options），类型为对象。将 2.3 节的示例中的以下代码：

```
new Ext.Viewport({
```

修改为：

```
Ext.create('Ext.Viewport',{
```

效果是一样的。那为什么要增加这样一个方法呢？我们来研究一下 create 方法的源代码。

在 ClassManger.js 文件中，可以找到以下代码：

```
Ext.apply(Ext, {
    create: alias(Manager, 'instantiate'),
            ...
})
```

从代码可知 create 方法其实是 Ext.ClassManager 类的 instantiate 方法的别名，其代码如下：

```
instantiate: function() {
    var name = arguments[0],
        nameType = typeof name,
        args = arraySlice.call(arguments, 1),
        alias = name,
        possibleName, cls;

    if (nameType != 'function') {
        if (nameType != 'string' && args.length === 0) {
            args = [name];
            name = name.xclass;
        }

        // 省略调试代码
        cls = this.get(name);
    }
    else {
        cls = name;
    }
    if (!cls) {
        possibleName = this.getNameByAlias(name);

        if (possibleName) {
            name = possibleName;

            cls = this.get(name);
        }
    }

    if (!cls) {
        possibleName = this.getNameByAlternate(name);

        if (possibleName) {
            name = possibleName;

            cls = this.get(name);
        }
    }

    if (!cls) {
        // 省略调试代码
```

```
            Ext.syncRequire(name);

            cls = this.get(name);
        }
        // 省略调试代码
        return this.getInstantiator(args.length)(cls, args);
    },
```

首先将 name 变量指向从参数中获取的类名，然后判断 name 是不是函数，如果不是，且其不是字符串，则从 xclass 属性中获取类名。接着使用 get 方法获取对象，并将 cls 变量指向对象。

get 方法的源代码如下：

```
get: function(name) {
    var classes = this.classes;
    if (classes[name]) {
        return classes[name];
    }
    var root = global,
        parts = this.parseNamespace(name),
        part, i, ln;
    for (i = 0, ln = parts.length; i < ln; i++) {
        part = parts[i];
        if (typeof part != "string") {
            root = part;
        } else {
            if (!root || !root[part]) {
                return null;
            }
            root = root[part];
        }
    }
    return root;
},
```

代码中的 classes 对象包括了 Ext JS 的所有类，因而代码首先根据 name 从 classes 中寻找类，如果存在，则返回对象；如果不存在，则说明是用户自定义的类，需要从 Ext.global 中寻找。

查找工作首先要做的是使用 parseNamespace 方法拆解类名，其代码如下：

```
parseNamespace: function(namespace) {
            // 省略调试代码
    var cache = this.namespaceParseCache;

    if (this.enableNamespaceParseCache) {
        if (cache.hasOwnProperty(namespace)) {
            return cache[namespace];
        }
    }

var parts = [],
    rewrites = this.namespaceRewrites,
    root = global,
```

```
        rewrite, from, to, i, ln;

        for (i = 0, ln = rewrites.length; i < ln; i++) {
            rewrite = rewrites[i];
            from = rewrite.from;
            to = rewrite.to;

            if (namespace === from || namespace.substring(0, from.length) === from) {
                namespace = namespace.substring(from.length);
                if (typeof to !== "string") {
                    root = to;
                } else {
                    parts = parts.concat(to.split("."));
                }

                break;
            }
        }

        parts.push(root);

        parts = parts.concat(namespace.split("."));

        if (this.enableNamespaceParseCache) {
            cache[namespace] = parts;
        }

        return parts;
    },
```

代码中 namespaceParseCache 对象的作用是使用类名作为关键字并指向拆解后的类名数组，这样，当该类被多次使用时，就可以直接从 namespaceParseCache 对象中获取拆解的类名数组，而不需要再拆解一次，从而加快运行速度。

通过 enableNamespaceParseCache 属性可配置是否开启 namespaceParseCache 对象的缓存功能，默认是开启的。如果在 enableNamespaceParseCache 中已存在类名的拆解结果，则返回结果。

属性 namespaceRewrites 的定义如下：

```
    namespaceRewrites: [{
        from: 'Ext.',
        to: Ext
    }],
```

在其他文件中找不到为 namespaceRewrites 添加数据的代码，因而在循环中，如果类名是以“Ext.”开头的，root 会指向 Ext 对象；如果不是，root 就是初始值，指向 Ext.global。

接着，将 root 指向的对象存入 parts 数组，再将拆分类名产生的数组与 parts 数组合并。如果开启了缓存功能，在 namespaceParseCache 对象中，将以类名为关键字指向 parts 数组，最后

返回 parts 数组。数组返回后，通过循环的方式来查找类定义。因为返回数组（parts）的第 1 个数据不是 Ext 对象就是 Ext.golbal 对象，因而变量 root 在第一次循环时，会指向 Ext 对象或 Ext.golbal 对象。在后续循环中，会根据拆分的类名在 Ext 或 Ext.gdbal 对象中逐层找下去。如果期间“root[part]”不是对象，说明不存在该类，返回 null。如果找到了，返回 root 指向的对象。

如果 cls 不是指向对象，则尝试使用别名方法查找对象。查找时，首先使用 getNameByAlias 方法将别名转换为类名，其代码如下：

```
getNameByAlias: function(alias) {
        return this.maps.aliasToName[alias] || '';
},
```

在 maps 属性中保存了以下 3 个对象：

```
maps: {
    alternateToName: {},
    aliasToName: {},
    nameToAliases: {}
},
```

简单来说，maps 中的对象就是一个对照表，通过它就可轻松地找到类名和别名。对象 alternateToName 的作用是通过备用名获得类名，它以备用名作为关键字、类名作为值；对象 aliasToName 的作用是通过别名获取类名，它以别名作为关键字、类名作为值；对象 nameToAliases 的作用是通过类名获取别名，它以类名作为关键字、别名作为值。在创建类的时候，会把类名、别名和备用名写入对照表。

如果 getNameByAlias 方法返回的不是空字符串，说明变量 name 保存的是别名，需将 name 修改为类名，然后通过 get 方法获取对象。如果使用别名也找不到对象，则可尝试使用备用名来查找对象，执行代码与使用别名查找的方式类似。如果还没有找到，则可尝试使用 syncRequire 方法下载对象，其代码如下：

```
syncRequire: function() {
    this.syncModeEnabled = true;
    this.require.apply(this, arguments);
    this.refreshQueue();
    this.syncModeEnabled = false;
},
```

在上面的代码中，syncModeEnabled 方法可控制 Loader 对象的 require[⊖]方法通过同步的方式去下载对象。

如果实在是找不到，就会抛出异常。最后使用 getInstantiator 方法实例化对象，其代码如下：

```
getInstantiator: function(length) {
    if (!this.instantiators[length]) {
        var i = length,
```

⊖ 相关信息可阅读 4.4.4 节。

```
            args = [];

        for (i = 0; i < length; i++) {
            args.push('a['+i+']');
        }

        this.instantiators[length] = new Function('c', 'a', 'return new c('+args.
            join(',')+')');
    }

    return this.instantiators[length];
},
```

代码中的数组 instantiators 起缓存作用，因为同样的参数长度产生的匿名函数是一样，没必要每次都重新创建一次。最后返回的匿名函数如下：

```
(function anonymous(c, a) {return new c(a[0], a[1], … ,a[n]);})
```

代码中的 n 就是参数的长度，例如 length 的值为 5，则匿名函数如下：

```
(function anonymous(c, a) {return new c(a[0], a[1], a[2], a[3], a[4]);})
```

匿名函数返回后会立即执行，参数 c 指向对象的 cls，a 指向实例化对象时的参数。也就是说，对象是在匿名函数中实例化的，这样可以保证对象的作用域是安全的。

从以上的分析可以了解到，创建对象的新方法不但可以实现动态加载，而且可以保证作用域的安全，应该优先使用。

5. 使用 Ext.widget 或 Ext.createWidget 创建对象

Ext.widget 的作用是使用别名来创建对象，其语法与 2.7.2 节介绍的 Ext.create 是一样的，只是 classname 使用的是对象的别名。Ext.createWidget 是 Ext.widget 的另外一种使用方法而已，在源代码中的定义如下：

```
Ext.createWidget = Ext.widget;
```

Ext.widget 的代码如下：

```
widget: function(name) {
    var args = slice.call(arguments);
    args[0] = 'widget.' + name;

    return Manager.instantiateByAlias.apply(Manager, args);
},
```

也就是说，它使用 ClassManager 对象的 instantiateByAlias 方法创建对象，其代码如下：

```
instantiateByAlias: function() {
    var alias = arguments[0],
        args = slice.call(arguments),
        className = this.getNameByAlias(alias);

    if (!className) {
```

```
        className = this.maps.aliasToName[alias];

            // 省略调试代码

        Ext.syncRequire(className);
    }

    args[0] = className;

    return this.instantiate.apply(this, args);
},
```

代码先是根据别名寻找类名，如果找不到就抛出异常或尝试使用 syncRequire 方法加载类文件，最后调用 instantiate 方法创建对象。

6. 使用 Ext.ns 或 Ext.namespace 定义命名空间

在使用 C# 或 Java 做开发时，很多时候都会使用命名空间来组织相关类和其他类型。在 JavaScript 中并没有提供这样的方式，不过可以通过定义一个全局对象的方式来实现，譬如你要定义一个“MyApp”的命名空间，你可以这样：

```
MyApp ={};
```

这里要注意，不要使用 var 语句去定义全局对象。

在 Ext JS 中，使用 Ext.namespace 方法可创建命名空间，其语法如下：

```
Ext.namespace(namespace1,namespace2,…,namespacen)
```

其中 namespace1、namespace2 和 namespacen 都是字符串数据，是命名空间的名字，例如：

```
// 推荐用法
Ext.namespace("MyApp","MyApp.data","MyApp.user");
Ext.ns("MyApp","MyApp.data","MyApp.user");
// 或者，不建议使用
Ext.namespace(,"MyApp.data","MyApp.user");
Ext.ns("MyApp.data","MyApp.user");
```

Ext.ns 只是 Ext.namespace 的简单写法。

在 ClassManager.js 中可找到 Ext.namespace 的 createNamespace 方法的实现代码：

```
createNamespaces: function() {
  var root = global,
      parts, part, i, j, ln, subLn;

for (i = 0, ln = arguments.length; i < ln; i++) {
    parts = this.parseNamespace(arguments[i]);

    for (j = 0, subLn = parts.length; j < subLn; j++) {
        part = parts[j];

        if (typeof part !== 'string') {
            root = part;
```

```
            } else {
                if (!root[part]) {
                    root[part] = {};
                }

                root = root[part];
            }
        }
    }

    return root;
},
```

从上面代码可以看到，Ext.namespace 创建的命名空间是保存在 global 对象里的。**注意循环结构，数组长度都是先保存到一个局部变量再使用的，原因是 JavaScript 与 C#、Java 等语言不同，每次循环都要计算一次数组长度，这样会降低性能**。在第一个循环下，使用了 parseNamespace 方法拆分命名空间的字符串。

在 Ext JS 初始化时，this 指向的是 window 对象，因而 global 指向的是 window 对象。

数组返回到 createNamespace 方法后，开始执行完循环，最后在 window 对象下生成了以下结构的全局对象：

```
{
    MyApp:{
        Data:{}
    }
}
```

虽然生成的是全局对象，但是在作用域链上，它可以直接在 Ext JS 的作用域链内搜索对象，而不需要到最外层的全局作用域链搜索对象，这是最大的不同。

在任何编程语言里都不提倡使用全局变量，尤其是在 JavaScript 里，全局对象在每一层的作用域链里都搜索不到时，才会在全局作用域链里搜索，效率相当低。**因此，在 JavaScript 里不仅不推荐使用全局变量，而且建议当有一个变量不是在当前作用域定义的，并要多次使用时，建议将该变量保存在局部变量中，使用局部变量来进行操作，以避免在域链中多次搜索对象而降低性能**。

因此，建议的方法是在 Ext 对象下创建命名空间，譬如创建以下的命名空间：

```
Ext.ns("Ext.MyApp","Ext.MyApp.data","Ext.MyApp.user");
```

最好的方法是使用 Ext 已定义的命名空间“Ext.app”，如果是扩展或插件则使用“Ext.ux”。

7. 使用 Ext.define 定义新类

在 Ext JS 3 中，定义新类使用的是 Ext.extend 方法，因为 Ext JS 4 的类系统进行了重新架构，所以定义新类的方法也修改为了 Ext.define 方法，其语法如下：

```
Ext.define(classname,properties,callback);
```

其中：

- classname：要定义的新类的类名。
- properties：新类的配置对象，对象里包含了类的属性集对象，表 2-1 列出了常用的属性及其说明。
- callback：回调函数，当类创建完后执行该函数。

表 2-1　常用的属性集对象及其说明

属　性	说　明

extend	要继承的类的名称，譬如 subclass 继承自 superclass，则定义如下：
	`Ext.define("subclass",{`
	`extend:"superclass",`
	`…`
	`});`

alternateClassName	类的备用名称，例如：
	`Ext.define("subclass",{`
	`alternateClassName:"myclass",`
	`…`
	`});`
	这样 subclass 就有了一个备用名称“myclass”，在使用 Ext.create 创建对象时就可使用 myclass 这个名称创建 subclass 实例了

alias	类的别名，用法可参考 alternateClassName 属性

requires	需要使用到的类名数组，在动态加载时会根据该属性去下载类，譬如 subclass。需要使用到 Ext.EventManager 和 Ext.util.MixedCollection，则定义如下：
	`Ext.define("subclass",{`
	`requires:["Ext.EventManager","Ext.util.MixedCollection"],`
	`…`
	`});`

constructor	构造属性，一般用来初始化类的配置项和调用其父类的方法，例如：
	`Ext.define("subclass",{`
	`extend:"superclass",`
	`constructor:function(config){`
	`this.initConfig(config);`
	`…`
	`this.callParent("mehtodName");`
	`return this;`
	`},`
	`…`
	`});`

（续）

属　性	说　明

mixins	为类添加特殊的功能，例如要为 subclass 混合 Ext.util.Observable 和 Ext.util.Animate 功能，则代码如下：
	`Ext.define("subclass",{`
	`mixins:{`
	`ob:"Ext.util.Observable",`
	`fx:"Ext.util.Animate"`
	`},`
	`constructor:function(config){`
	`var me=this;`
	`me.initConfig(config);`
	`me.mixins.ob.constructor.call(me);`
	`...`
	`return me;`
	`},`
	`fx:function(){`
	`var me=this;`
	`...`
	`return me.mixins.fx.animate.apply(me,arguments);`
	`},`
	`...`
	`});`

config	定义类的配置项，创建时会自动为 config 里的每个属性添加 set 和 get 方法，例如：
	`Ext.define("subclass",{`
	`config:{`
	`width:100,`
	`height:100`
	`},`
	`constructor:function(config){`
	`this.initConfig(config);`
	`...`
	`return this;`
	`},`
	`...`
	`});`
	`var mysubclass=new subclass();`
	`mysubclass.setWidth(200);`
	`mysubclass.getHeight();`

（续）

属　性	说　明
config	代码中，initConfig 方法执行后就会为 congfig 的属性创建 set 和 get 方法，这样创建类的示例后，就可以直接通过 set 和 get 方法读写属性的值，譬如例子中用 setWidth 设置了配置中 width 的值，使用 getHeight 获取 height 的值

statics	定义静态方法，例如：
	`Ext.define("subclass",{`
	`statics:{`
	`method:function(args){`
	`return new this(args);`
	`}`
	`},`
	`constructor:function(config){`
	`this.initConfig(config);`
	`…`
	`},`
	`…`
	`});`
	`var myclass=subclass.method("class");`

类的创建过程将在第 4 章讲解，下面我们来实践一下。在 Firefox 中打开 2.3 节的示例，然后打开 Firebug，在控制台中输入以下代码：

```
Ext.define("Calculator",{
    constructor:function(){
        return this;
    },
    plus:function(v1,v2){
        return v1+v2;
    },
    minus:function(v1,v2){
        return v1-v2;
    },
    multiply:function(v1,v2){
        return v1*v2;
    },
    divid:function(v1,v2){
        return v1/v2
    }
});
var cal=new Calculator();
console.log(cal.plus(87,28)); //115
console.log(cal.minus(87,28)); //59
console.log(cal.multiply(7,8)); //56
console.log(cal.divid(28,7)); //4
```

代码中定义了一个名称为“Calculator”的类，它包含加、减、乘、除 4 个方法。运行后，

将 Firebug 标签页切换到 DOM 标签，可看到如图 2-2 所示的 Calculator 类。

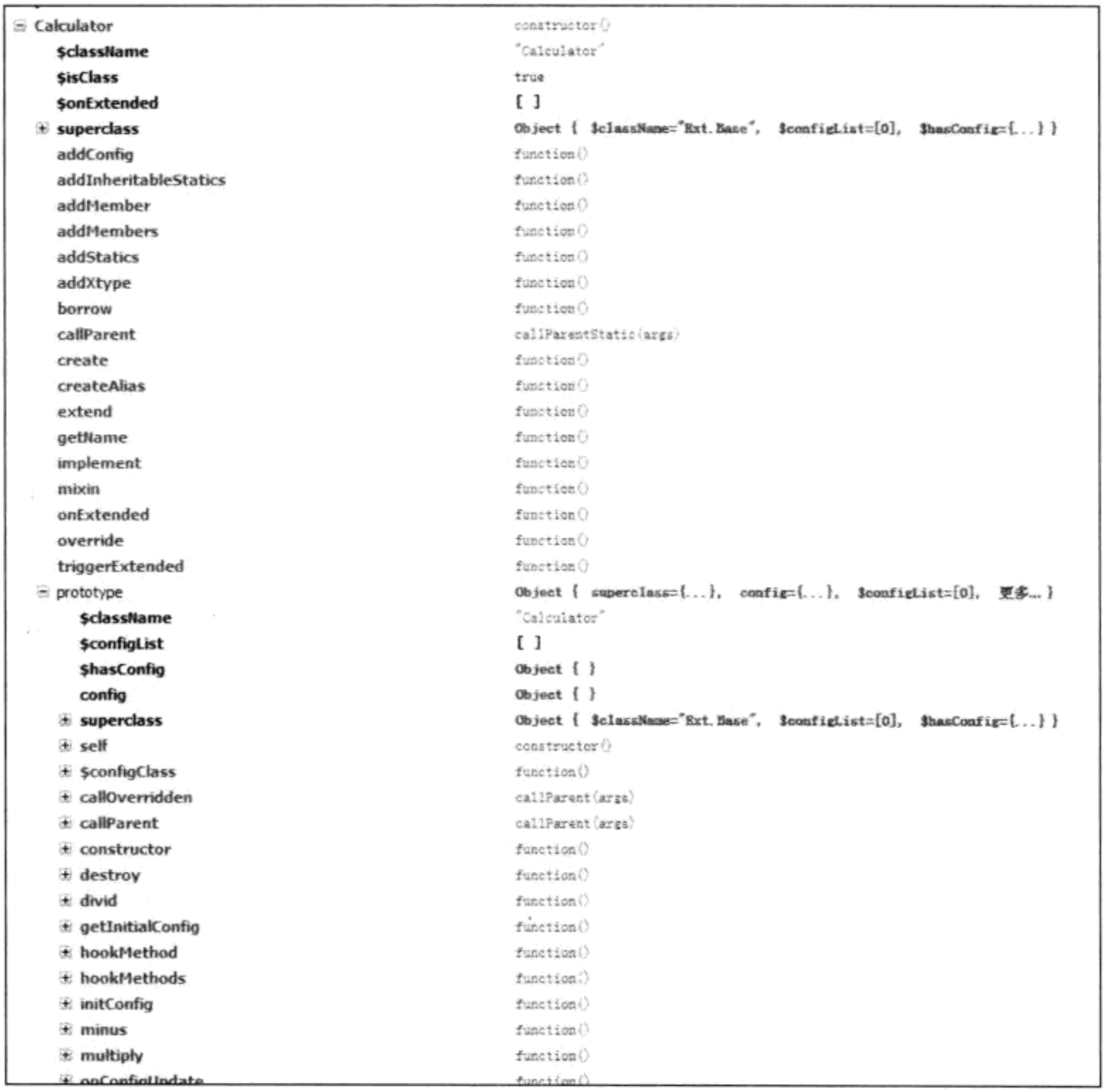


图 2-2　DOM 树中的 Calculator 类

继续创建一个继承自 Calculator 类的新类 NewCalculator，新类添加了十进制转换为十六进制的方法，代码如下：

```
Ext.define('NewCalculator',{
    extend:'Calculator',
    hex:function(v1){
        return v1.toString(16);
    }
});
var ncal=new NewCalculator();
console.log(ncal.hex(28)); //1c
```

代码中 extend 属性的值为 Calculator，表示 NewCalculator 将继承自 Calculator 类。方法 hex 是新增的进制转换方法。

运行后可在 DOM 树中看到如图 2-3 所示的 NewCalculator 类。

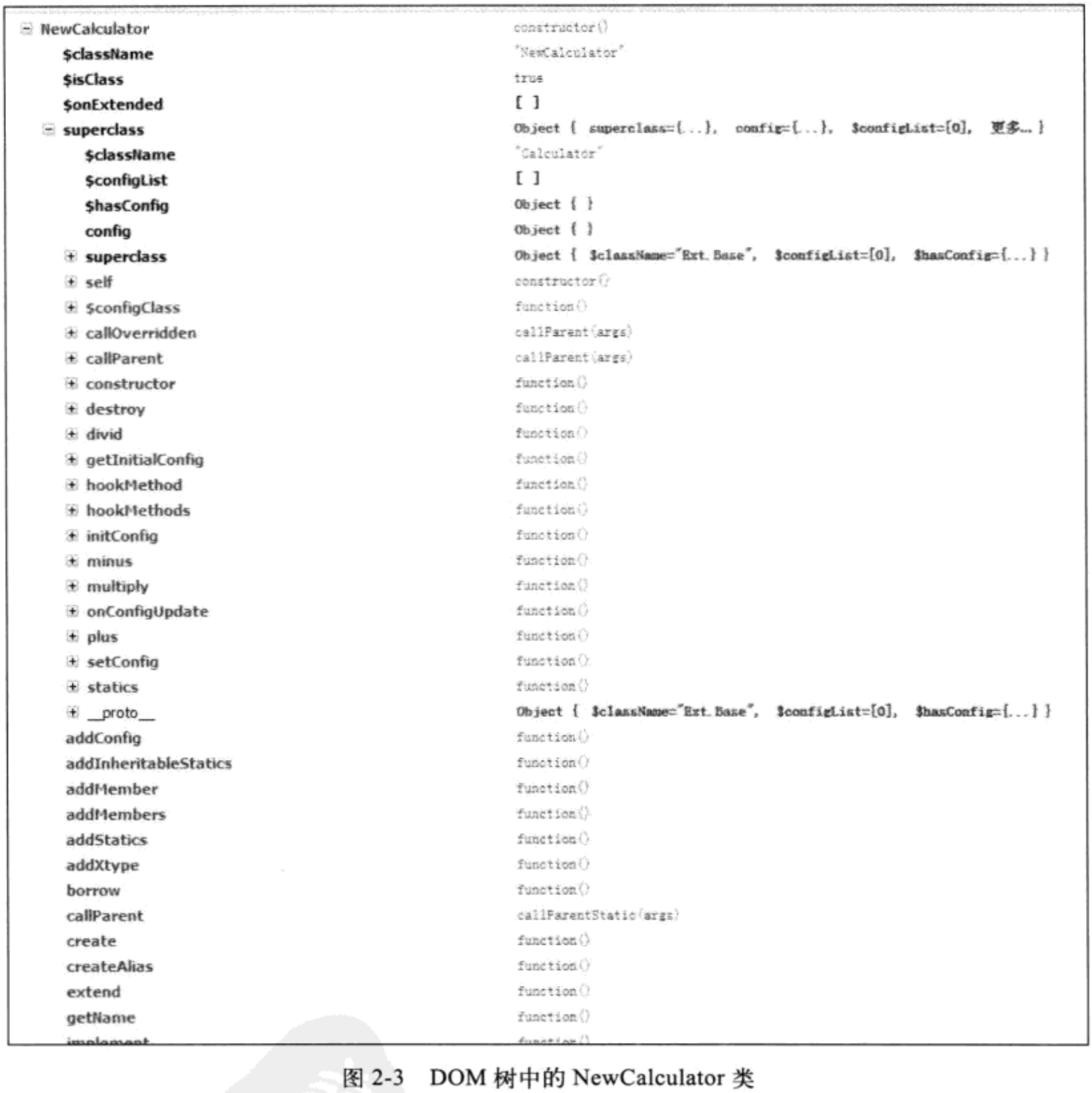


图 2-3　DOM 树中的 NewCalculator 类

比较一下图 2-3 与图 2-2，可看到 Calculator 类的超类（superclass）是 Ext.Base[⊖]，而 NewCalculator 类的超类是 Calculator，这说明，没有使用 extend 属性定义的类会默认从 Ext.Base 中继承。

⊖　相关信息可阅读 4.4.3 节。

继续我们的实践，这次要做的是将 HEX、BIN、OCT 这 3 个实现不同进制转换的类混合到 NewCalculator 类中。第一步要做的是定义 3 个实现进制转换的类：

```
Ext.define('HEX',{
    hex:function(v1){
        return v1.toString(16);
    }
});
Ext.define('BIN',{
    bin:function(v1){
        return v1.toString(2);
    }
});
Ext.define('OCT',{
    oct:function(v1){
        return v1.toString(8);
    }
});
```

然后将 HEX、BIN 和 OCT 三个功能混合到继承自 Calculator 类的 NewCalculator 类中，这样 NewCalculator 除了实现加减乘除功能外，还能实现十进制到二进制、八进制和十六进制的转换，代码如下：

```
Ext.define('NewCalculator',{
    extend:'Calculator',
    mixins:{
        Hex:'HEX',
        Bin:'BIN',
        Oct:'OCT'
    },
    convert:function(value,type){
        switch(type){
            case 2:
                return this.bin(value)
                break;
            case 8:
                return this.oct(value)
                break;
            default:
                return this.mixins.Hex.hex.call(this,value);
                break;
        }
    }
});
```

```
var ncal=new NewCalculator();
console.log(ncal.convert(25,2)); //11001
console.log(ncal.convert(25,8)); //31
console.log(ncal.convert(25,16)); //19
```

在代码中，使用 convert 方法可进行进制转换，第 2 个参数 type 表示要转换的进制类型。

调用 mixins 属性定义的混合功能有两种方法：第一种是二进制和八进制中使用的直接调

用的方法，第二种是十六进制中使用 call 方法调用的方法。

运行后，在 DOM 中可看到如图 2-4 所示的 NewCalculator 类。

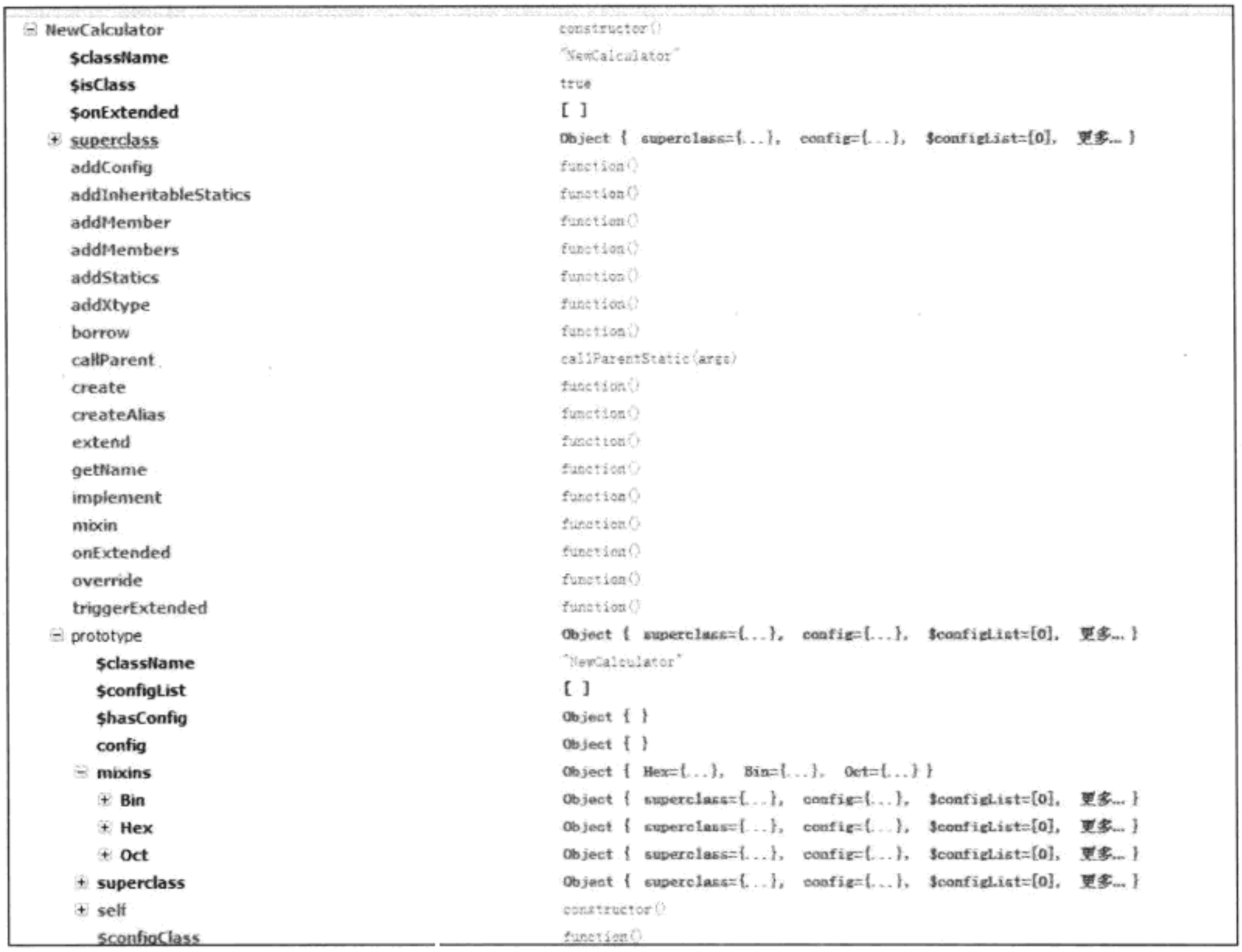


图 2-4　DOM 树中使用了 mixins 属性的 NewCalculator 类

在图 2-4 中的原型（prototype）节点里，可看到 mixins 对象中的 3 个属性都指向了对应的 BIN、HEX 和 OCT 对象，而这三个对象中的方法也直接附加到原型上了，这也就解释了为什么可以使用两种方法调用混合功能的方法。**要注意的是，如果存在同名方法，譬如将 BIN、HEX 和 OCT 定义的方法都修改为 convert，那么使用直接调用的方法就会出现错误，因为在 JavaScript 中，存在同名函数，前面的定义会被最后定义的函数覆盖，所以在不确定是否有同名方法的情况下，建议还是使用第二种 call 的方法。**

定义 3 个进制转换类实在麻烦，在一个 Convert 类中，预设好一个 type 参数，然后根据 type 定义的类型进行转换就方便多了，代码如下：

```
Ext.define("Convert",{
    config:{
    type:"hex"
    },
     type_num:16,
```

```
constructor:function(config){
        this.initConfig(config);
    return this;
},
applyType:function(type){
    this.type_num= type=="hex" ? 16 : ( type=="oct" ? 8 : 2);
    return type;
},
convert:function(v){
    return v.toString(this.type_num);
}
});
var cv=new Convert();
console.log(cv.convert(29)); //1d
cv.setType("oct");
console.log(cv.convert(29)); //35
cv.setType("bin");
console.log(cv.convert(29)); //11101
```

在 Convert 类中，type 属性在 initConfig 执行后会自动生成 applyType、setType、getType 和 resetType 这 4 个方法。当 type 的值发生改变时，会触发 applyType 方法，因而可以重写 applyType 方法以实现所需的功能。在 applyType 方法中，根据 type 的值修改了 type_num 的值，这样在 convert 方法中就可以使用 type_num 值直接进行进制转换了。

代码运行后在 DOM 中可看到如图 2-5 所示的 Convert 类。

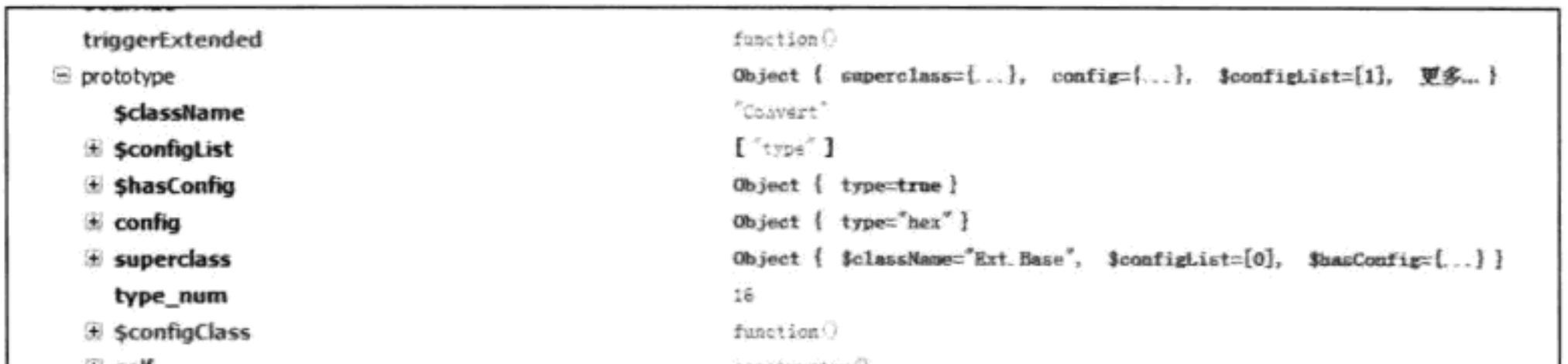


图 2-5　DOM 树中的 Convert 类

在图 2-5 的原型中，可看到与 Type 有关的 4 个方法。实现这么简单的功能，每次都要使用 new 关键字对象太麻烦了。如果是静态类就好了，所以修改代码如下：

```
Ext.define("Convert",{
    statics:{
        hex:function(v){
            return v.toString(16);
        },
        oct:function(v){
            return v.toString(8);
        },
        bin:function(v){
            return v.toString(2);
        }
    },
    constructor:function(config){
        return this;
    }
});
console.log(Convert.hex(29)); //1d
console.log(Convert.oct(29)); //35
console.log(Convert.bin(29)); //11101
```

在 Convert 类中，3 个进制转换方法都定义在 statics 属性中。运行后在 DOM 树中可看到如图 2-6 所示的 Convert 类。

图 2-6　DOM 树中 Convert 静态类

在图 2-6 中，3 个进制转换方法不在原型里，而是直接挂在类节点下。现在大家应该清楚静态类和其他类的区别了。

2.8 本地化

Ext JS 提供了 45 种语言的本地化文件，在 Ext JS 文件包的 local 中可以找到这些文件。如果要实现中文简体的本地化，可在页面的 head 部分加入以下语句：

```
<script type="text/javascript" src="../Ext4/locale/Ext-lang-zh_CN.js"></script>
```

打开语言包，会看到在 Ext.onReady 函数内有许多判断语句，其作用是判断 Ext 对象是否存在，如果存在则修改对象的属性，将属性带有的语言信息转换为对应的本地化语言信息。

2.9 为本书示例准备一个模板

为了便于示例的讲解，特意准备了这个模板。在没有特殊说明的情况下，本书后续的示例都会在此模板的基础上添加额外代码，该模板的代码将不会出现在示例的代码清单里。

新建一个 Templates.html 文件，加入如代码清单 2-2 所示的代码。

代码清单 2-2 模板代码清单

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/
strict.dtd">
    <html>
    <head>
        <title></title>
         <link rel="stylesheet" type="text/css" href="../Ext4/resources/css/ext-all.
css"/>
        <script type="text/javascript" src="../Ext4/bootstrap.js"></script>
         <script type="text/javascript" src="../Ext4/locale/Ext-lang-zh_CN.js"></
script>
        <style type="text/css">
              /* 在此添加样式代码 */
        </style>
    </head>
    <body>
       <!-- 在此添加 HTML 代码 -->
       <script type="text/javascript">
              Ext.onReady(function(){
                    if(Ext.BLANK_IMAGE_URL.substr(0,4)!="data"){
                          Ext.BLANK_IMAGE_URL="./images/s.gif";
                    }
                    // 在此添加 Ext JS 代码
              });
       </script>
    </body>
    </html>
```

模板主要包含了一个页面的完整结构，配置使用了 Ext JS 库，包含简体中文包，并已定义好 onReady 函数和图片检查。

2.10　本章小结

本章通过一个简单的例子学习了使用 Ext JS 的方法及其基本语法。尤其是在 2.7 节“使用 Ext.define 定义新类”中，我们实践了使用 Ext.define 定义新类的几种方法，目的是让大家熟悉 Ext JS 中的类定义并了解其中的变化。第 4 章会继续深入探讨类的生成过程，这对了解和使用 Ext JS 是非常有帮助的。定义模型和创建模型的语法会在第 9 章（数据模型）讲解。

第3章　调试工具及技巧

子曰："工欲善其事，必先利其器。"作为开发人员，好的调试工具是必不可少的。幸运的是，现在 JavaScript 的调试工具越来越多，也越来越方便。本章将介绍 Firebug、Debugbar、IE 9 等调试工具，根据你们的喜好选择一款吧，这是必需的。

3.1　使用 Firebug 进行调试

1.Firebug 概述

Firebug 是由 Joe Hewitt 开发的、基于 Firefox 的一个扩展，是目前最好的 Web 开发调试工具。它可以在客户端实时编辑、调试和检测任何页面的 CSS、HTML 和 JavaScript。

虽然也可以在其他浏览器中使用 Firebug lite，但是功能都不如在 Firefox 中强大，因而本书没有特别声明时说的 Firebug 都是基于 Firefox 环境的。

2011 年是 Mozilla"疯狂"的一年，Firefox 的版本一下子从 3.6 升级到 9，版本更新的速度越来越快，读者可以根据自己的喜好选择一个适合自己的版本。本节的安装过程是在 4.0 版进行的，与最新的版本可能会有差异，但基本上还是一致的。

2. Firebug 的安装

打开 Firefox 后，如图 3-1 所示，打开 Firefox 菜单，并选择图中红色圈住部分的"附加组件"。

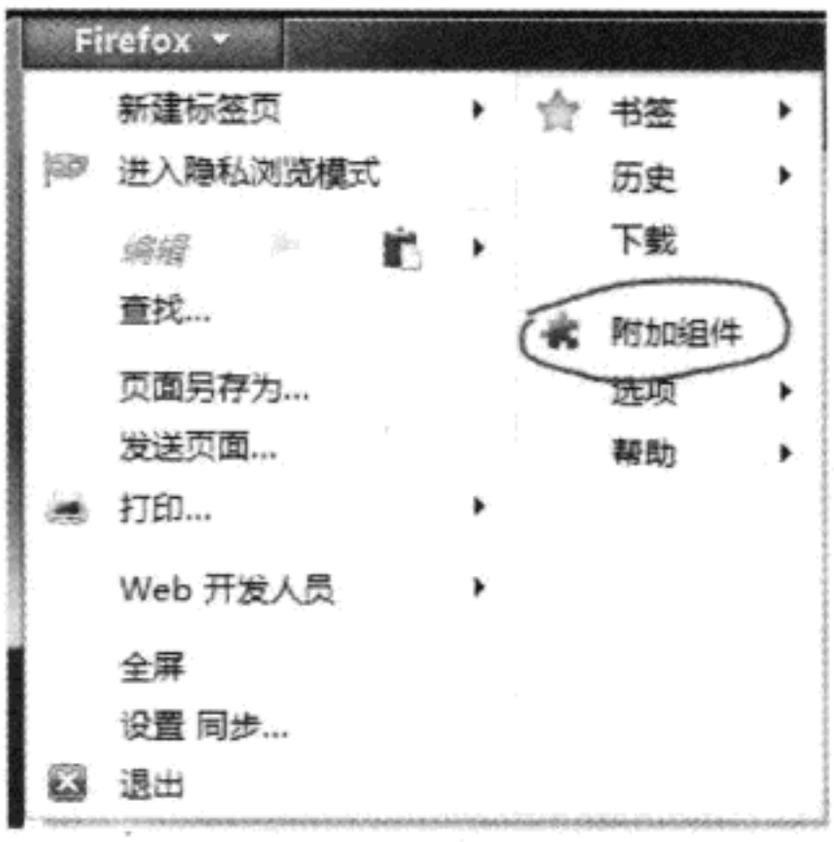


图 3-1　在 Firefox 菜单中选择附加组件

在如图 3-2 所示的附加组件管理器标签页的搜索框中输入 Firebug 并按下回车键。

在如图 3-3 所示的搜索结果出来后，单击 Firebug 1.7.0 条目中的"安装"按钮。

安装完成后将显示如图 3-4 所示的结果，单击"立即重启"重新启动 Firefox，即可完成 Firebug 的安装。

Firefox 重启后会如图 3-5 所示，在底部多了一个附加组件栏，栏中最右边多了一个臭虫图标，这是 Firebug 附加的工具按钮。单击可打开如图 3-6 所示的 Firebug 工作区域。

图 3-2　附加组件管理器

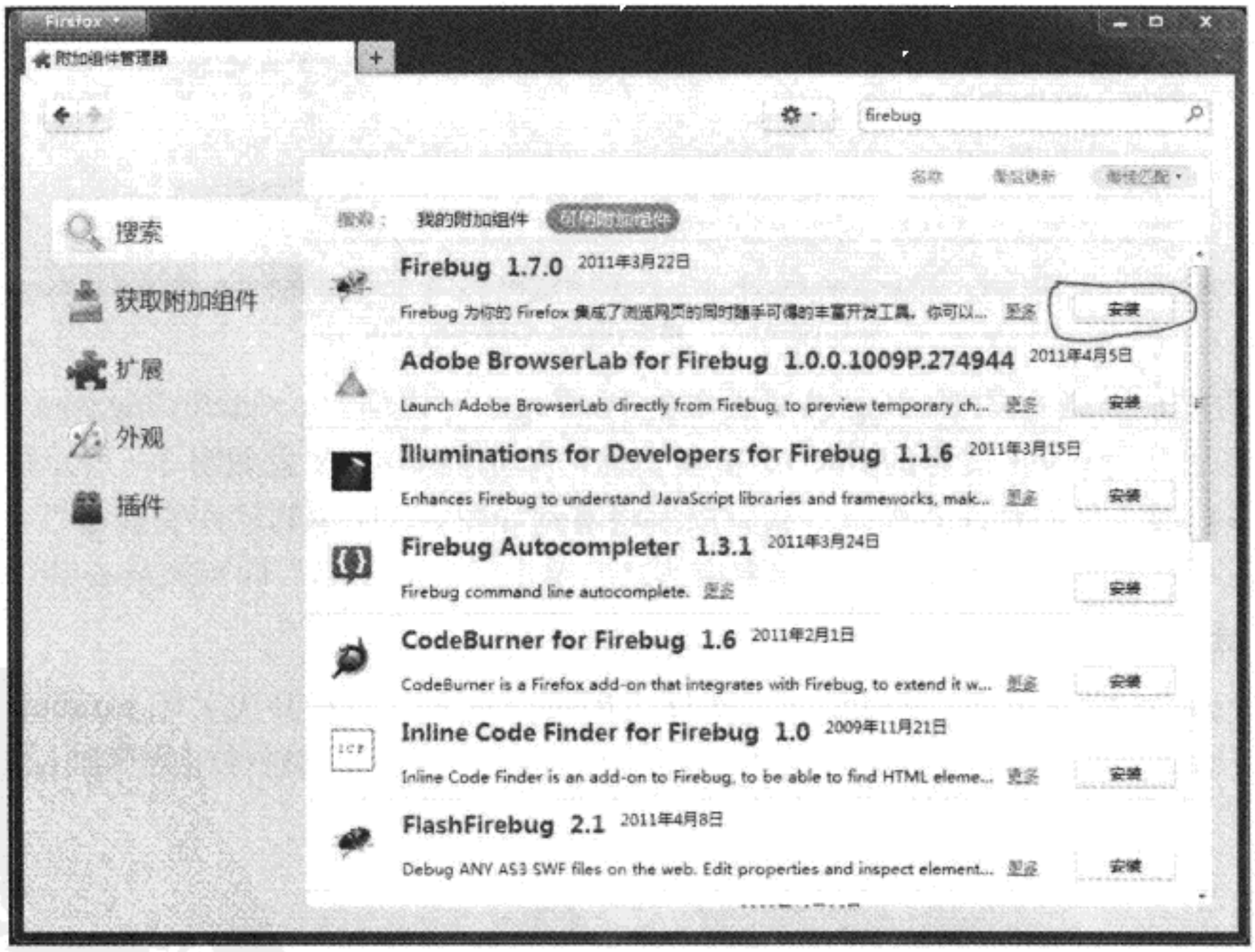


图 3-3　Firebug 搜索结果

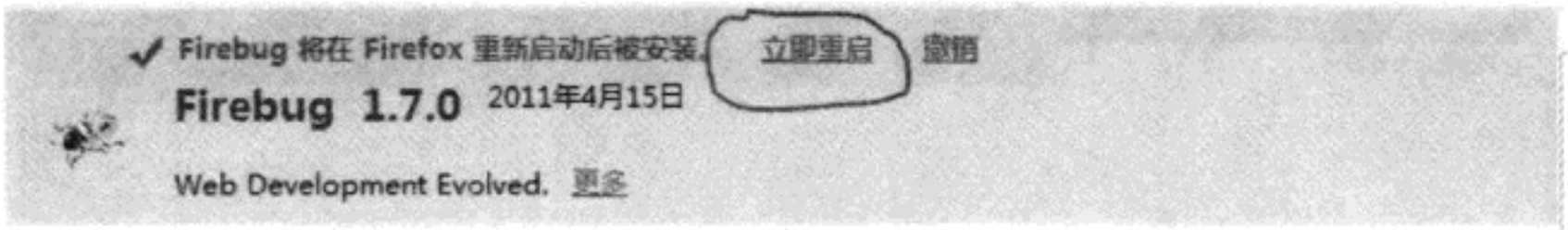


图 3-4 Firebug 安装完成后条目显示结果

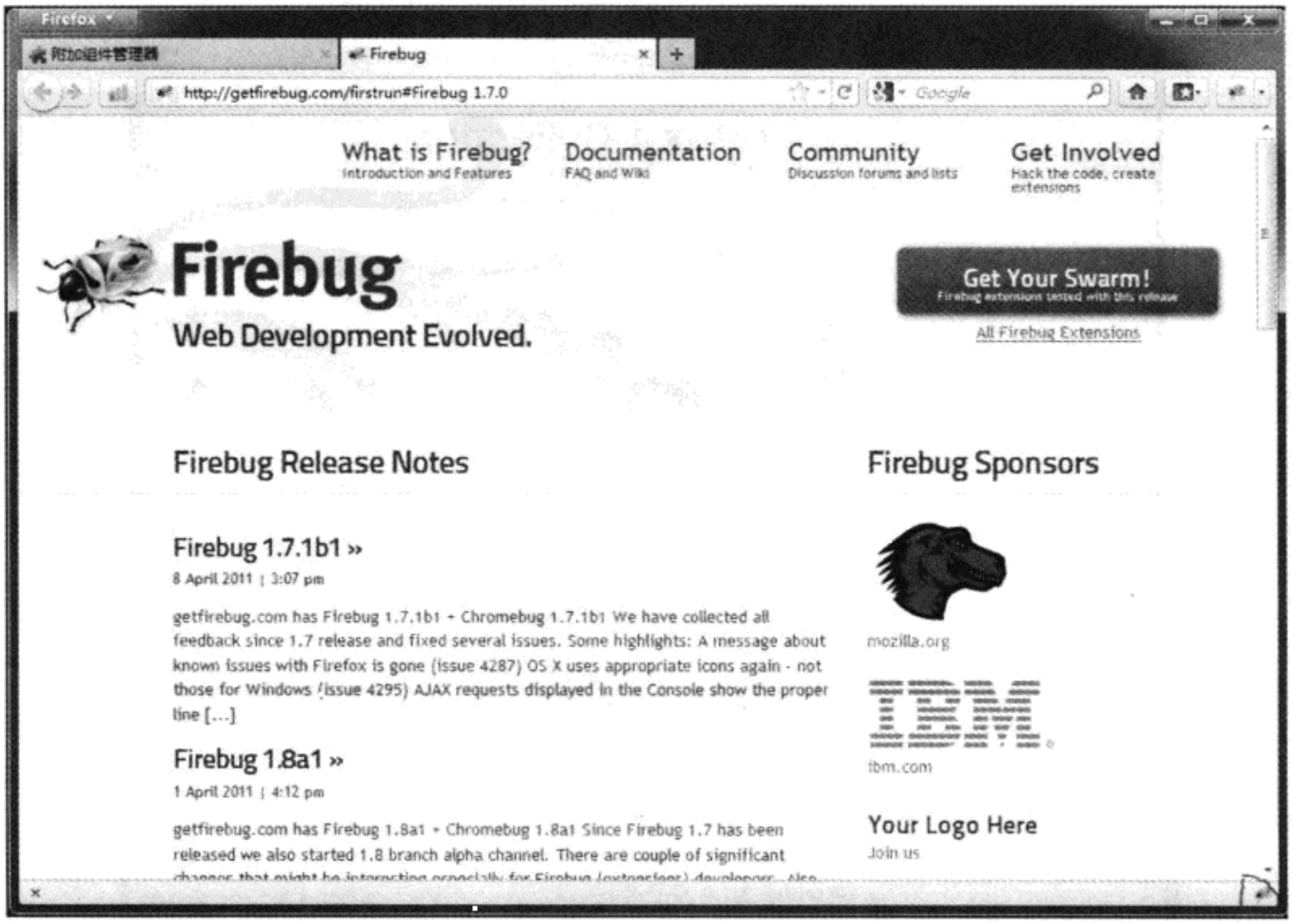


图 3-5 重启后的 Firefox

如果需要最新版本的 Firebug 或旧版本的 Firebug，可访问 http://getfirebug.com/downloads。网站提供的是以 xpi 为扩展名的 Firefox 扩展文件，单击后 Firefox 会如图 3-7 所示提示你是否允许安装该扩展。单击允许并等待下载完成后将看到如图 3-8 所示的软件安装窗口，单击“立即安装”后可看到如图 3-9 所示的立即重启信息。单击“立即重启”即可完成安装。

3. Firebug 的界面

通过 Firefox 附加组件栏的 Firebug 图标或用快捷键 F12 可打开或关闭 Firebug 窗口。从图 3-6 中可看到 Firebug 主要由工具栏和面板区两部分组成。工具栏按钮的详细说明如表 3-1 所示。

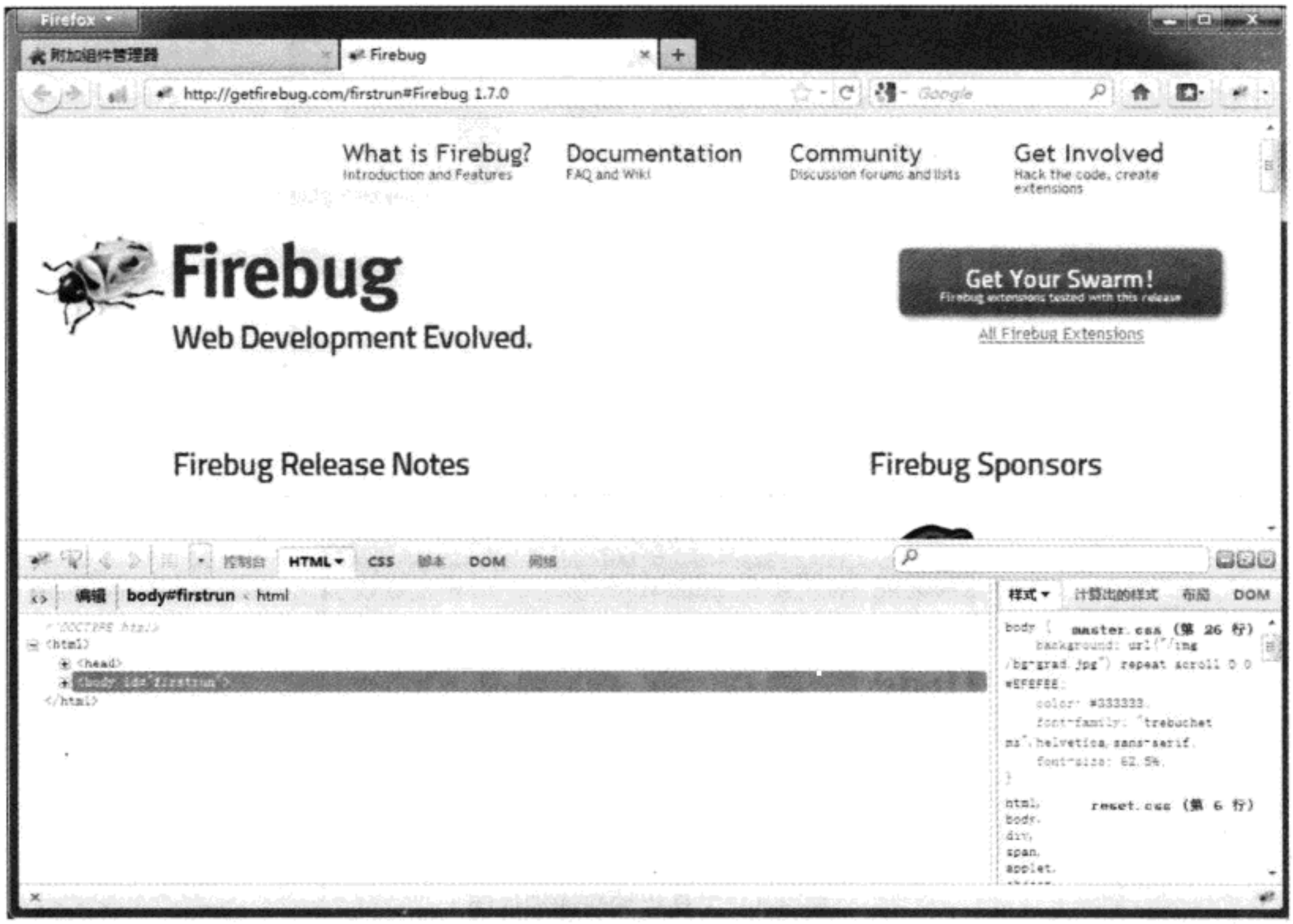


图 3-6　打开 Firebug 工作区域后的 Firefox

图 3-7　Firefox 提示是否运行安装扩展

图 3-8　软件安装窗口

图 3-9　立即重启提示

表 3-1　工具栏按钮详细说明

按　钮	说　明
	Firebug 主菜单
	单击后可在页面中获取元素，在 HTML 面板上会根据鼠标的移动切换到鼠标当前所指的源代码。单击选择页面元素后，HTML 源代码会固定在选元素上。这是非常实用的功能，会经常用到
	面板切换按钮
	在控制台面板以外的面板显示命令行
	面板选择菜单
控制台	切换到控制台面板，提供控制台面板菜单
HTML	切换到 HTML 面板，提供 HTML 面板菜单
CSS	切换到 CSS 面板，提供 CSS 面板菜单
脚本	切换到脚本面板，提供脚本面板菜单
DOM	切换到 DOM 面板，提供 DOM 面板菜单
网络	切换到网络面板，提供网络面板菜单
搜索框	在**面板**内（不是页面）搜索
	最小化 Firebug
	在一个新窗口内显示 Firebug
	关闭 Firebug 窗口或显示区域

Firebug 有控制台、HTML、CSS、脚本、DOM 和网络 5 个基本面板，切换到不同的面板时，面板区的显示会不同。有些基于 Firebug 的插件会在 Firebug 内增加自己的面板。

默认状态下只有 HTML、CSS 和 DOM 面板是处于活动状态的，而控制台、脚本和网络面板需要启动才能使用。这主要是因为开启这 3 个面板很占资源，而且容易造成 Firefox 崩溃，这需要大家做好心里准备。

4. 控制台面板

控制台面板的主要作用是显示各种的错误信息（可在面板按钮的下拉菜单中定义），输出脚本调试信息，使用命令行调试脚本，以及检测 JavaScript 执行时间的概况。

将面板切换到控制台将看到如图 3-10 所示的面板区。这时面板还没开启，如果要开启所有面板，可在附加组件栏的 Firebug 图标上单击鼠标右键，在菜单中启用所有面板。如果只开启当前面板，则单击面板中的“启用”，或在工具栏的面板按钮下拉菜单中选择“启用”。启动后的控制台面板将如图 3-11 所示，它包含工具栏、信息区和命令行 3 个区域。

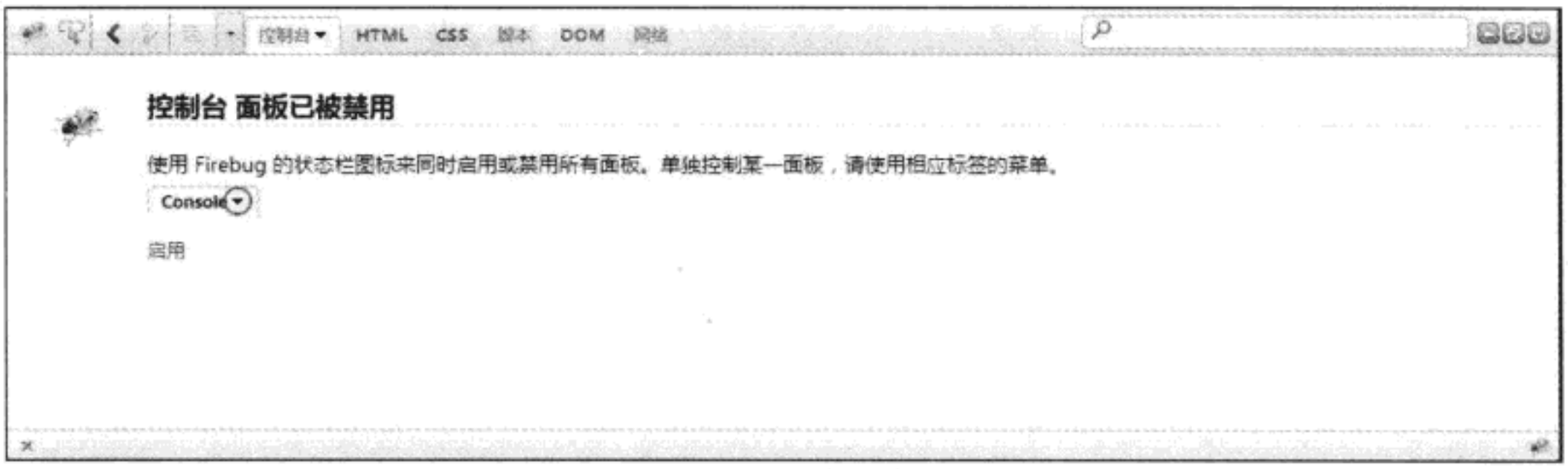


图 3-10　还没启动的控制面板

工具栏按钮的详细说明如表 3-2 所示。

图 3-11　启动后的控制台面板

表 3-2 工具栏按钮的详细说明

按 钮	说 明
	当脚本运行错误时，会在控制台面板中显示错误信息，启动该按钮后会在所有错误的地方加上断点以便调试
清除	清除信息区内的信息
保持	当页面重载时，不清空当前信息区，保留当前信息区的信息
概括	开启 JavaScript 脚本执行概括检测
所有	不过滤信息区的信息，显示全部信息
错误	过滤信息区的信息，只显示错误信息
警告	过滤信息区的信息，只显示警告信息
消息	过滤信息区的信息，只显示消息信息
调试信息	过滤信息区的信息，只显示调试信息

在命令行最左边和最右边都有一个小按钮，最左边的按钮可选择历史命令。单击最右边的按钮，可看到命令行如图 3-12 所示，显示在面板右边了。这样做的目的是方便你输入多行脚本，例如 2.7.6 小节的类定义代码，如果是单行命令行，输入就很麻烦了。单击命令行里的“运行”按钮可执行代码，单击最右边的小图标可恢复成单行命令行。

图 3-12 命令行显示在右边

5. HTML 面板

切换到 HTML 面板可看到如图 3-13 所示的面板。它和控制台面板差不多，分了工具栏、HTML 代码区和功能区 3 个部分。

在工具栏里有两个主要按钮。第一个按钮的作用是当 DOM 改变时中断脚本。“编辑”按钮的作用是将 HTML 代码区变成编辑区域，可直接编辑 HTML 代码。在“编辑”按钮的右边是以层次关系显示的 HTML 元素，在最右边的是 HTML 根标记，然后是 body 标记。

HTML 代码区看到的代码是最终源代码，也就是说由脚本生成的 HTML 代码也会显示。在源代码中移动鼠标，会在页面对应区域中出现一个色块，即鼠标所指源代码在页面中的位置。反过来，如果你使用 Firebug 工具栏中的选取元素按钮在页面中选定页面元素，对应的 HTML 源代码也会被选中。该功能对 HTML 调试相当有用。

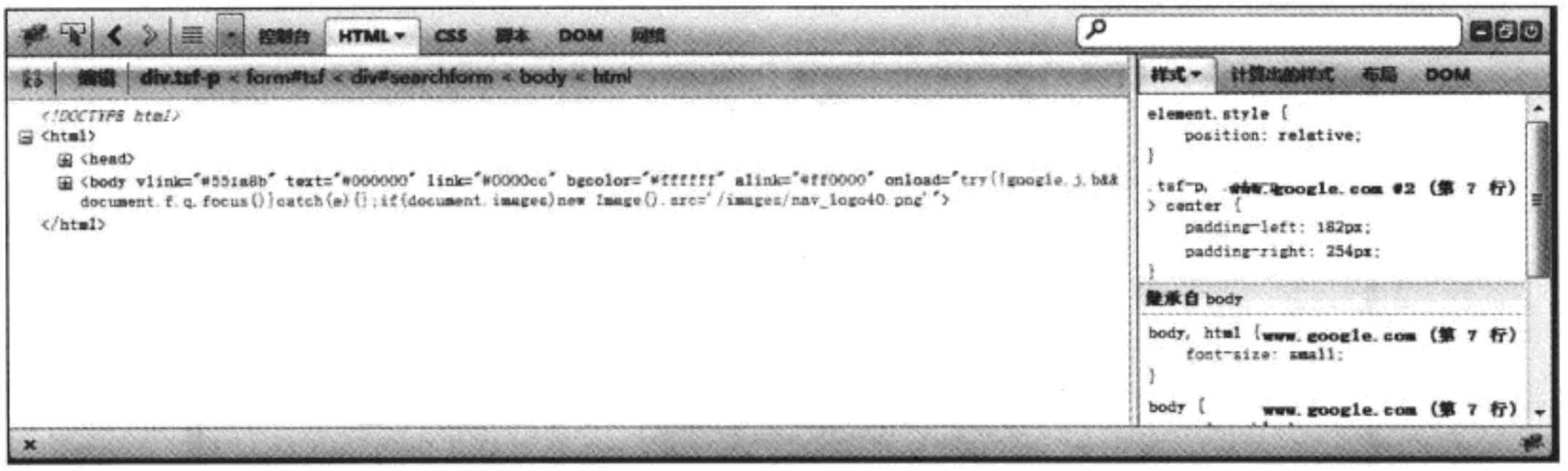


图 3-13　HTML 面板

在代码区单击元素的属性或属性值，可进入编辑状态，直接编辑属性和属性值。在鼠标右键菜单中还可为元素添加和删除属性。

选择一个元素后，在右边的功能区查看该元素的样式、计算出的样式、布局（盒子模型）和 DOM 节点。如图 3-14 所示，可以在样式中单击样式左边出现的小图标以屏蔽或开启某个样式，进行样式调试。也可以单击样式的属性或属性值，直接修改它们。通过右键菜单可添加样式。把鼠标移动到带有颜色或图片的样式上，会出现一个显示该颜色或图片的提示框。

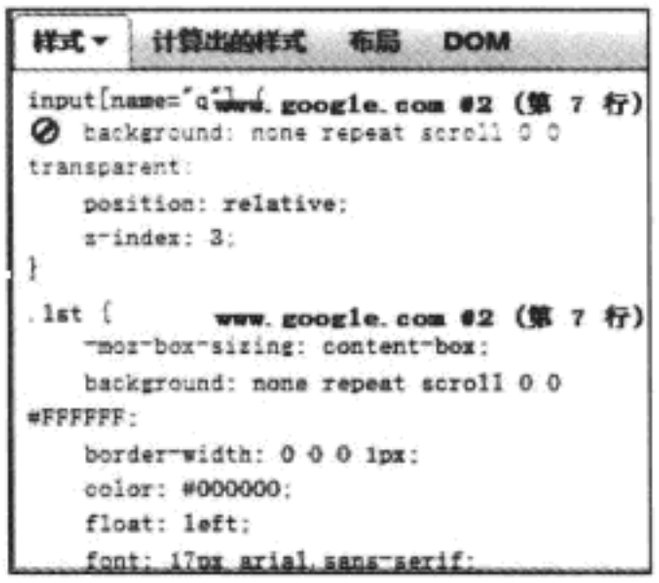


图 3-14　屏蔽样式

如果需要调整页面元素的布局，可直接修改布局功能区中的数字。不过调整好后，记得修改页面文件，不然辛苦调整好的布局在刷新后就没了。有关盒子模型的知识可阅读一下 CSS 方面的书籍[⊖]以及后面的内容。这里要注意的是，IE 的盒子模型和 Firefox 的盒子模型是不同的。在这里调整好的，在 IE 上看未必是这样的。在盒子模型上移动鼠标会如图 3-15 所示，在页面中出现标尺和色块。不同颜色的色块表示盒子模型中不同部分所占的区域。标尺和直线可让你轻松判断当前元素与页面其他元素的位置关系。

6. CSS 面板

如图 3-16 所示，CSS 面板比较简单，只有一个工具栏和一个信息区。

工具栏只有两个按钮，第 1 个按钮可将信息区变成编辑状态，直接编辑样式，在第 2 个按钮的下拉菜单中可选择要查看的样式文件。

CSS 面板与 HTML 面板功能区中的样式区域一样，可屏蔽或开启样式，可直接修改样式的属性和属性值，也可通过右键菜单添加属性。不过，CSS 面板多了一个功能，就是可通过右键菜单“新建规则”添加新的样式。

7. 脚本面板

脚本面板和控制台面板一样，也需要启动后才能使用，启动后将看到如图 3-17 所示的面板，面板也是由工具栏、信息区和功能区这 3 部分组成的。工具栏按钮的详细说明如

⊖　推荐阅读国内有本 CSS 3 著作《CSS 3 实战》（机械工业出版社，2011）。

表 3-3 所示。

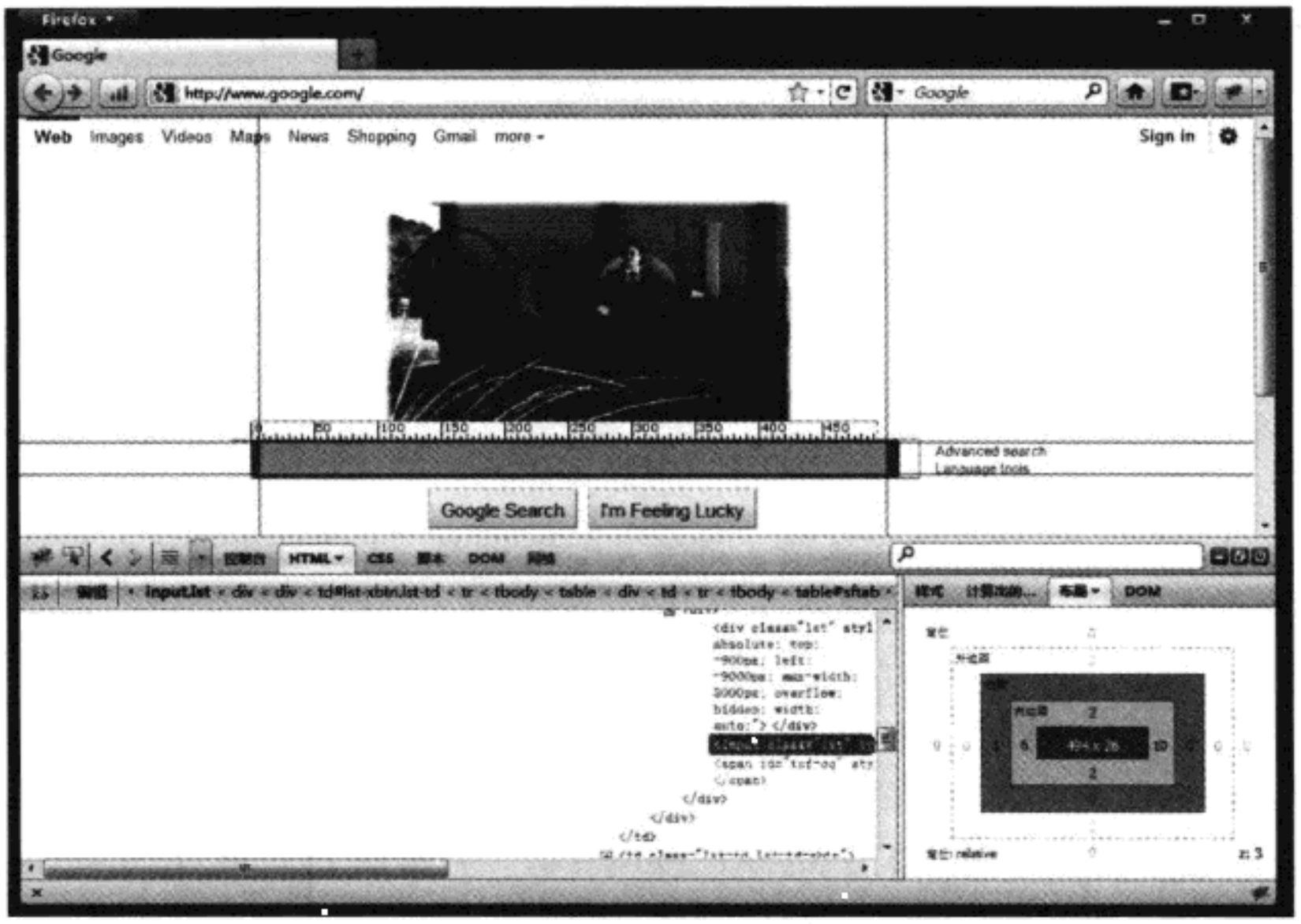


图 3-15　鼠标在盒子模型移动时在页面中显示的标尺

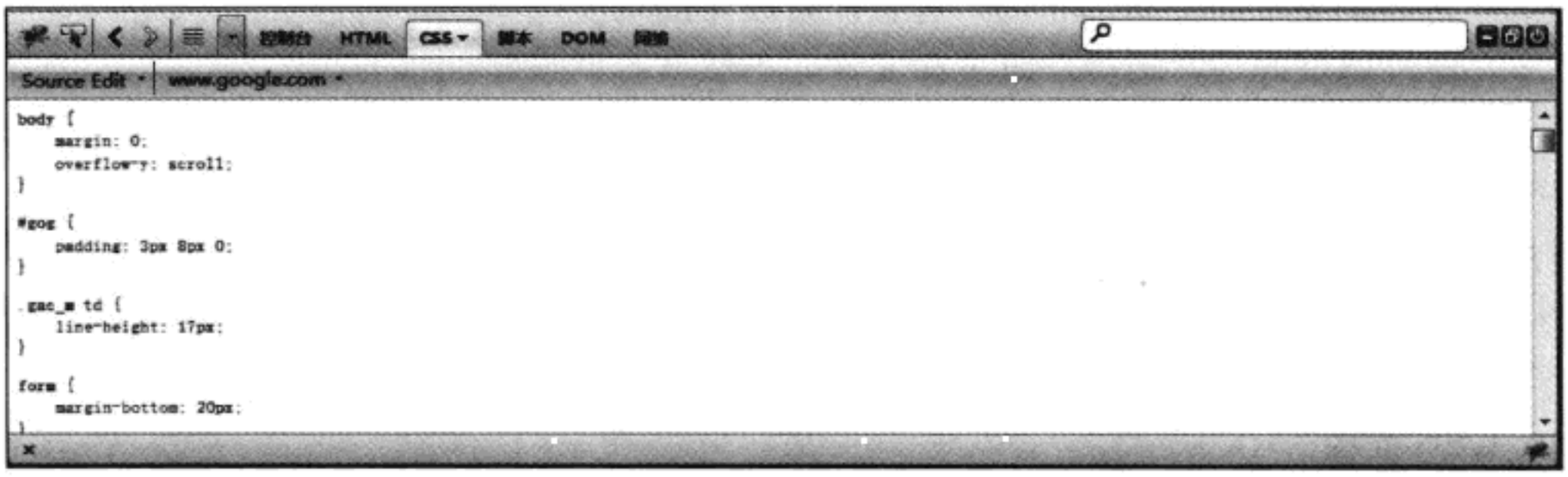


图 3-16　CSS 面板

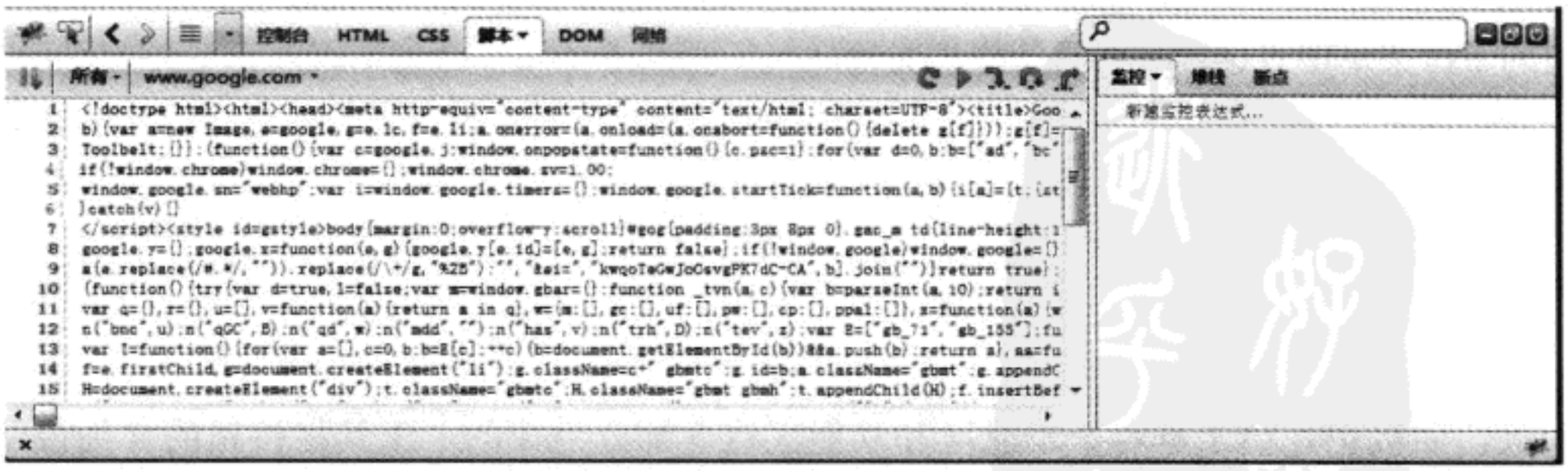


图 3-17　脚本面板

表 3-3　脚本面板工具栏按钮的详细说明

按　钮	说　明
	可在下一个语句中断脚本的执行
所有	可从下拉菜单中过滤脚本，只显示特定的脚本
[网站域名]	可选择要查看的脚本文件
	脚本调试按钮，返回代码跳转位置
	脚本调试按钮，继续执行代码。该功能的快捷键是 F8
	脚本调试按钮，单步执行，每次执行一条语句。该功能的快捷键是 F11
	脚本调试按钮，单步跳过。每次执行一条语句，与单步执行的区别在于调用函数时，不进入调函数内部进行跟踪。该功能的快捷键是 F10
	脚本调试按钮，单步退出。在函数内部跟踪时，如果想退出函数体，可使用该按钮。该功能的快捷键是 Shift+F11

工具栏的搜索框在脚本面板上有个特殊功能，就是在搜索框里输入“#n”（n ≥ 1）时，可直接跳转到脚本的第 n 行。

为了实践一下脚本调试功能，我们先做一个简单页面，页面除了基本元素只包含以下脚本：

```
<script type="text/javascript">
    for(var i=0;i<500;i++){
        console.log(i);
    }
</script>
```

脚本只有一个执行 500 次的循环，并在控制台中输出 i 值。

在浏览器中打开页面，然后打开 Firebug，启动脚本面板，再刷新一次页面将在脚本面中看到如图 3-18 所示的内容。

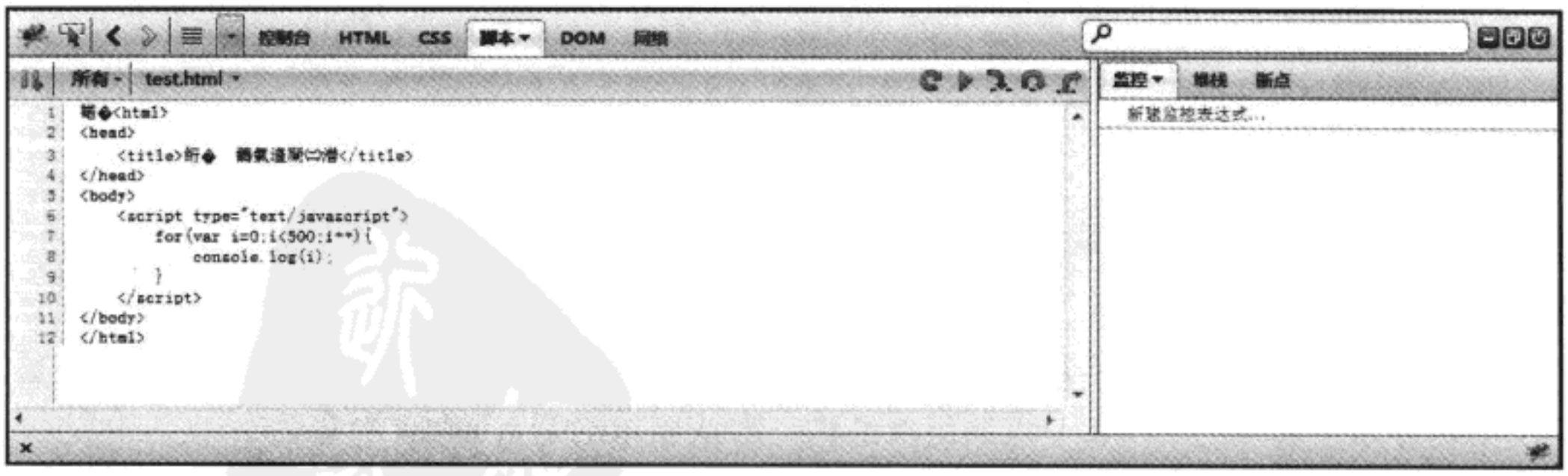


图 3-18　测试页面在脚本面板中的显示内容

在行号列，单击第 8 行会出现一个红点，表示已经在该位置设置了一个断点。再次刷新页面，脚本运行将停在断点位置。这时，在监控功能区会看到如图 3-19 所示的 DOM 节点情

况，而在堆栈功能区会看到如图 3-20 所示的堆栈情况，在断点功能区会看到如图 3-21 所示的断点情况。如果有多个断点，也都会在这里显示。单击断点左边的选择框，可屏蔽或开启断点，而单击最右边的删除图标可删除断点。

图 3-19 监控功能区显示的 DOM 节点情况

按下 F11（单步执行）键，可在监控功能区看到 i 值在不断地变化。如果不想继续监控，可按下 F8（继续执行）键。与其他 IDE 工具一样，在调试时，将鼠标移动到变量上方时，会以提示的方式显示变量的当前值。在第 8 行的行号列中单击鼠标右键会出现如图 3-22 所示的断点条件输入框。在输入框内输入“i=100”，那么在 i 值为 100 时，程序就会中断，停在断点。按下回车键后，在监控功能区可看到 i 值已经为 100 了，程序不再继续执行。

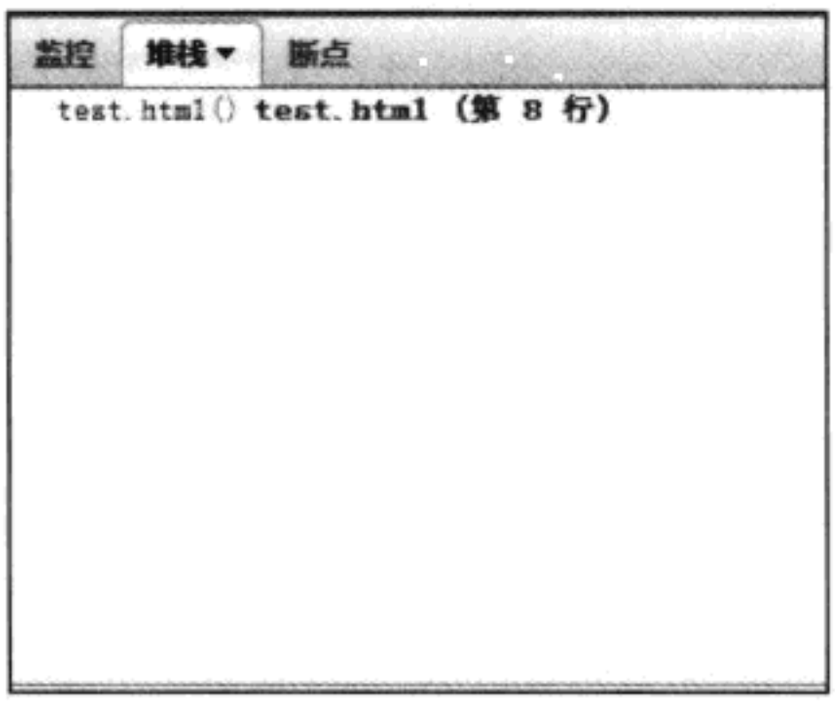


图 3-20 堆栈情况

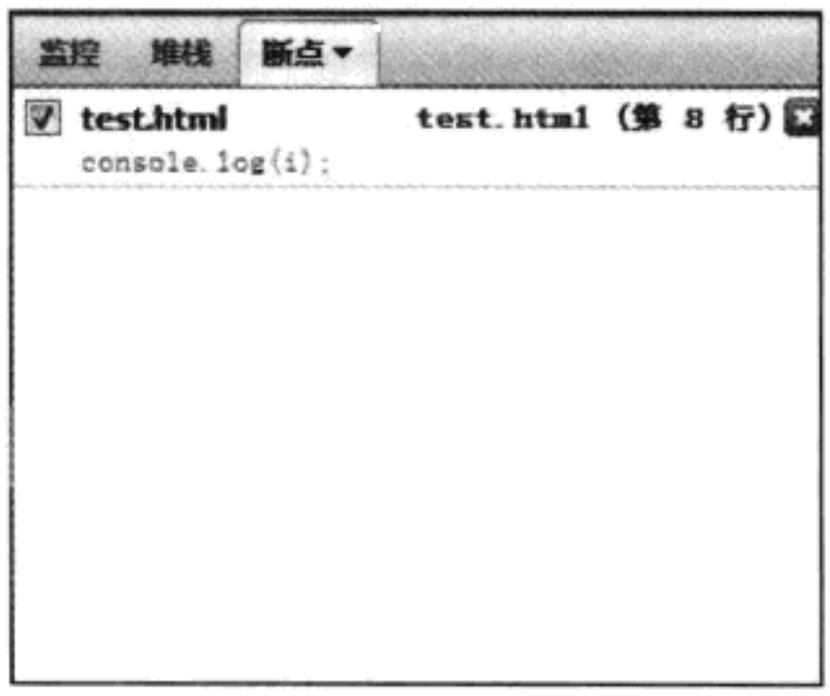


图 3-21 断点情况

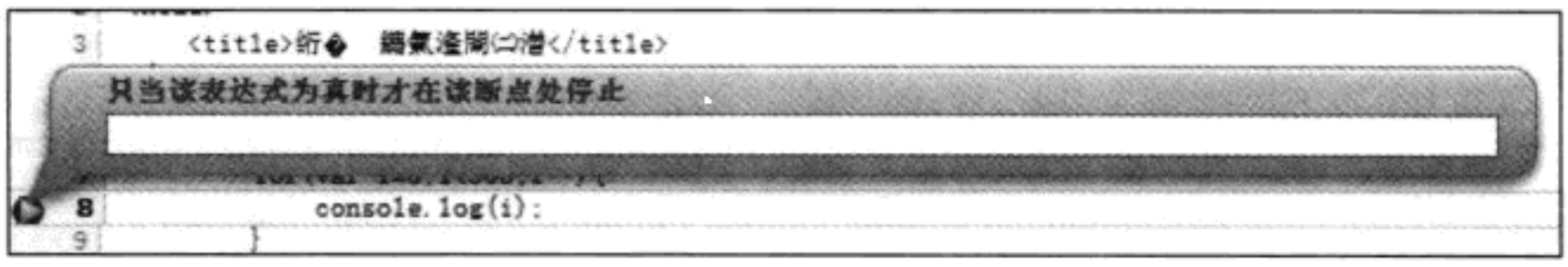


图 3-22 断点条件输入框

结合其他面板，脚本调试的功能会更强大，这有待大家在使用中发掘。

8. DOM 面板

DOM 面板和 CSS 面板一样，只有一个工具栏和信息区。工具栏没有任何功能按钮，只以层次结构显示所选择的 DOM 节点的层次结构。默认显示的是 window 对象。

在信息区可看到整个 DOM 树，通过搜索框可在 DOM 树中搜索对象。也可以在控制台中先用 console.log 命令显示该对象，然后单击控制台中的信息转移回 DOM 树中查看对象，例如，在控制台命令窗口中输入以下语句：

```
console.log(document);
```

在控制台会显示如下信息：

```
Document test.html
```

单击该信息会在 DOM 面板中看到如图 3-23 所示的结果，在工具栏可看到 document 对象的层次，而在信息区可看到 document 对象的属性和方法。

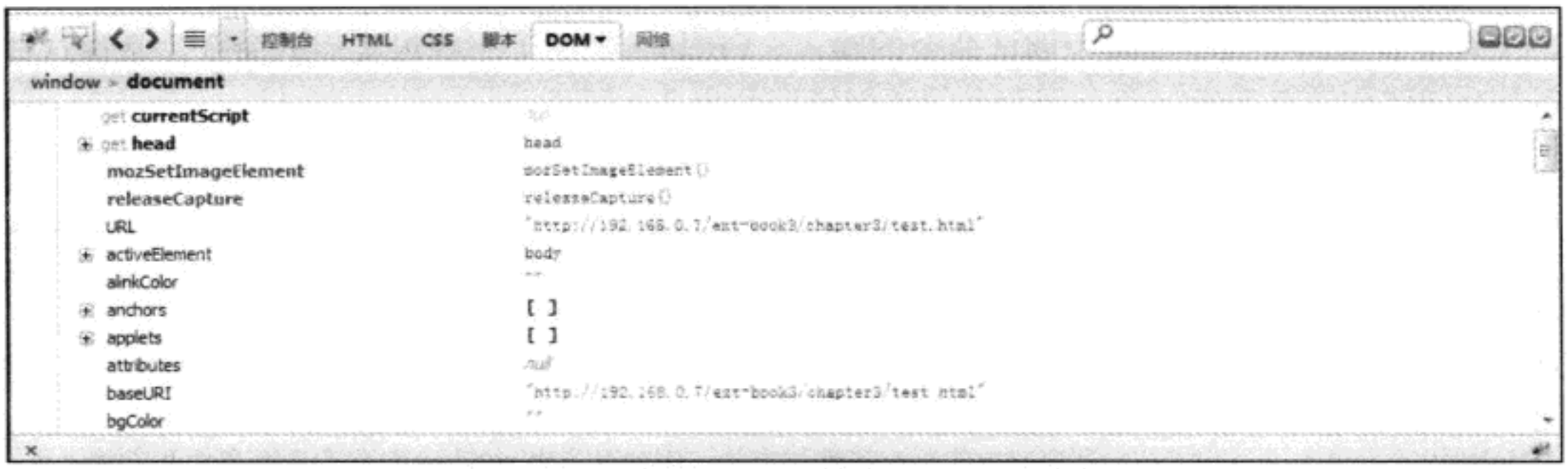


图 3-23　DOM 面板中查看 document 对象

需要熟练掌握该方法，因为很多时候需要查看 Ext JS 中的 Store 情况或其他对象情况，使用该方法比较方便。

通过 DOM 对象研究 Ext JS 也是一个不错的方法。第 1.4.2 节的方法就是这样研究出来的。

9. 网络面板

在网络面板可查看页面的下载情况。例如，我们打开 2.3 节的示例的页面，会看到如图 3-24 所示的显示。

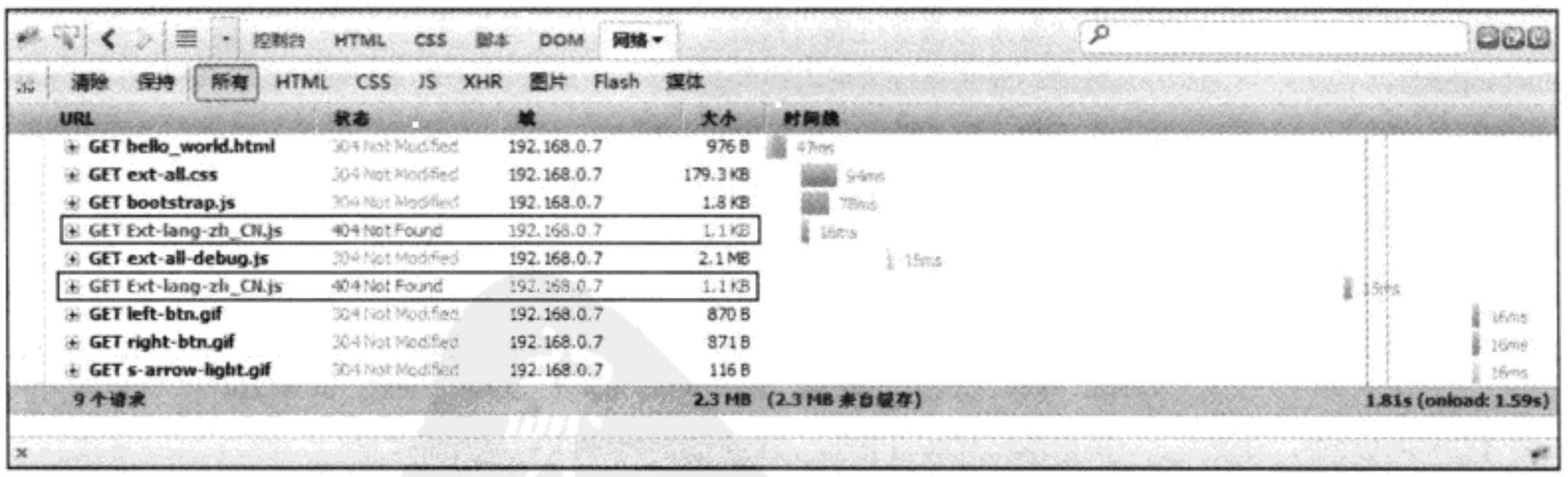


图 3-24　打开 2.3 节的示例页面后网络面板的显示

图 3-24 中加框部分表示这些文件下载出错，需要检查路径或文件状态。该功能很实用，很多时候页面不能正常显示就是因为图片无法下载或文件出错造成的，在网络面板上可对这些情况进行检查。

在每个文件最右边都会显示其下载速度和下载开始位置，在底部会有统计信息，有助于

你了解网页下载情况，并对其进行优化。

单击文件左边的小图标，可查看文件的头信息、提交的参数、响应和缓存等信息。这是检测 Ajax 提交的好方法，检查的内容包括提交的参数是否正确、返回的信息是否正确等。Ext JS 开发中常见的问题就是提交的参数没有搞清楚，或者是返回的数据格式不符合要求等，通过该方法可以轻松解决这些问题。

10. 命令行命令介绍

为了提高调试效率和减少调试命令的输入，Firebug 提供了一些很实用的命令，掌握这些命令对调试非常有帮助。命令行命令的详细说明如表 3-4 所示。

表 3-4　命令行命令的详细说明

命　令	说　明
$(id)	代替 document.getElementById 语句，获取页面元素
$$(selector)	使用选择器获取元素数组
$x(xpath)	使用 xpath 表达式获取元素数组
dir(object)	列出对象的所有属性和方法，例如要列出 window 对象的所有属性和方法，可输入： `dir(window)`
dirxml(node)	列出节点（node）的 HTML 或 XML 的源代码树，与在 HTML 面板查看该节点的效果一样
cd(window)	如果页面有 iframe，默认的 window 对象是主页面的，要操作 iframe 里的 window 对象必须从主页面的 window 对象开始引用。使用该命令，可将当前主页面的 window 对象转换为 ifarme 里的 window 对象，这样就可以方便地调试 iframe 里的页面了
clear()	清除信息区内的信息，与清除按钮作用相同
inspect(object[, 面板名])	在指定的面板监视对象（object）。如果不设置面板名称，则在控制台监视。面板名称的值可以为 HTML、CSS、Script 和 DOM
keys(object)	返回对象的属性组成的数组
value(object)	返回对象的属性值所组成的数组
debug(fn)	在函数（fn）的第一行增加一个断点
undebug(fn)	在函数（fn）中移除断点
monitor(fn)	跟踪函数（fn）的调用
unmonitor(fn)	取消对函数（fn）的调用跟踪
monitorEvents(object[,types])	跟踪对象的事件。types 的值可以为 composition、contextmenu、drag、focus、form、key、load、mouse、mutatuin、paint、scroll、text、ui 和 xul
unmonitorEvents(object[, types])	取消跟踪对象的事件。types 请参考 monitorEvents 命令

11. Illuminations for Developers 介绍

Illuminations for Developers 是一个专门用于调试 Ext JS 的小工具，它会在 Firebug 中添加一个 Illuminations 面板，在该面板内可以查看 Ext JS 对象。

Illuminations for Developers 和 Firebug 的安装过程一样。安装后打开 2.3 节的示例，可在 Illuminations 面板中看到如图 3-25 的结果。

图 3-25　打开 2.3 节的示例后 Illuminations 面板的显示结果

单击 Illuminations 面板工具栏中的 Widgets、Data 和 Element 三个按钮可分别查看页面包含的 Ext JS 组件、数据和生成的页面元素。

选择图中的 Ext JS 对象，可通过功能区面板查看对象的属性（Properties）、方法（Methods）、事件（Events）、数据（Records，如果对象带有 Store）、文档（Docs）、HTML 代码（HTML）、样式（Style）、计算出的样式（Computed）和布局（Layout）。

Illuminations 还扩展了 Firebug 工具栏的选取元素按钮，可直接通过该按钮选择 Ext JS 对象（Illuminations 工具栏必须选择为 Widgets），例如，在 2.3 节示例中单击“点选”按钮，然后选择面板的标题栏，将看到如图 3-26 所示的结果。从图 3-26 中可以看到，面板标题栏 Header 对象中包含一个 Component 对象控件。

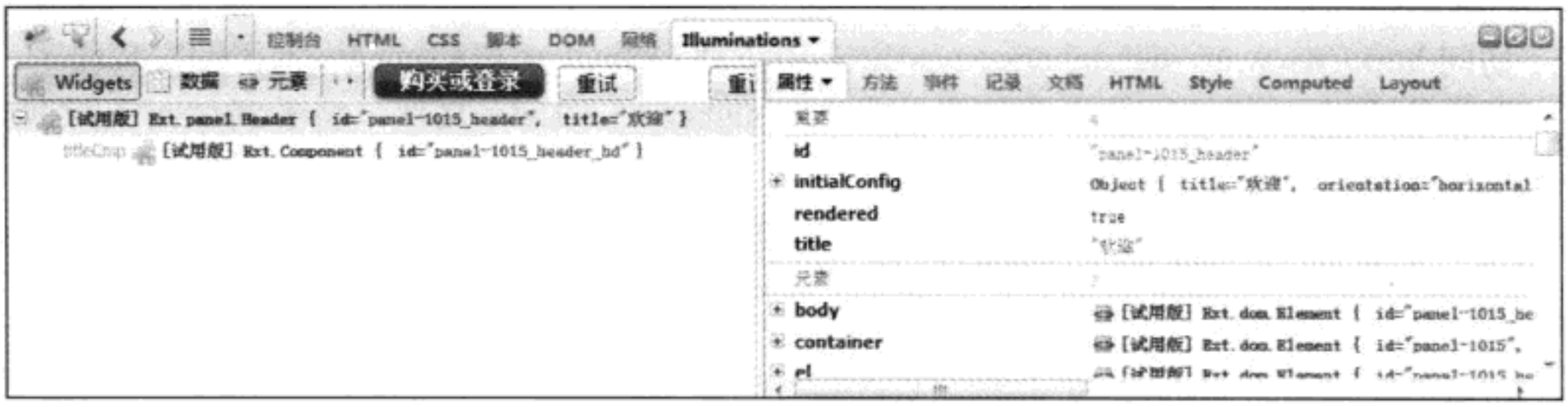


图 3-26　选取日期控件后 Illuminations 面板的显示结果

你也可以在页面中使用右键菜单实现该功能。

该工具绝对是调试 Ext JS 的好帮手，可惜，它不是完全免费的，看到工具栏的“购买或登录”按钮是不是心都凉了半截？不过，这东西也不算太贵，商业用户第一年需要 50 美金，以后每年 25 美金。个人用户第一年 25 美金，以后每年 12 美金。如果你觉得能给你带来收益，购买也未尝不可。

12. Firebug 实用扩展 Firebug AutoCompleter

Firebug 自带的自动完成功能只能在单行模式下使用，为了在多行模式下使用自动完成功能需要安装额外的扩展。Firebug AutoCompleter 就是这样的扩展。

Firebug AutoCompleter 的安装与 Firebug 的安装方式类似，在附加组件管理器中搜索到后，安装它并重启浏览器就可以使用了。这是非常实用的扩展，建议大家和 Firebug 一起安装。

3.2 在 IE 中调试

在 IE 8 之前，在 IE 中的调试就只有可怜的 alert 命令了，虽然可以在 Visual Studio 中进行调试，但太麻烦了。Firebug Lite 虽然也发布了支持 IE 的版本，但是需要在页面中加入 Firebug Lite 的脚本文件才行，而且在 Firebug 中的很多功能不能用。就目前来说，做得比较好的还是 Debugbar 工具，不过与 Firebug 比起来还是有很大的差距。

3.2.1 使用 Debugbar 和 Companion.js 调试

1. 介绍

Debugbar 虽然可以与 Firebug 一样获取页面元素、做源代码调试和 CSS 调试，可惜功能实在有限，只不过是聊胜于无而已。例如，如果要做类似 Firebug 那样的源代码调试，还需要使用 Companion.js 文件，而 Companion.js 是从 Firebug Lite 扩展出来的，所以功能还是有限的。

要注意，Debugbar 对个人用户是免费的，而对商业用户是收费的。

2. 安装

访问 http://www.my-debugbar.com/wiki/，可下载 Debugbar 和 Companion.js 的安装程序。

首先安装 Debugbar，运行下载的 install-debugbar-v5.4.1.exe 文件，将看到如图 3-27 所示的安装界面。

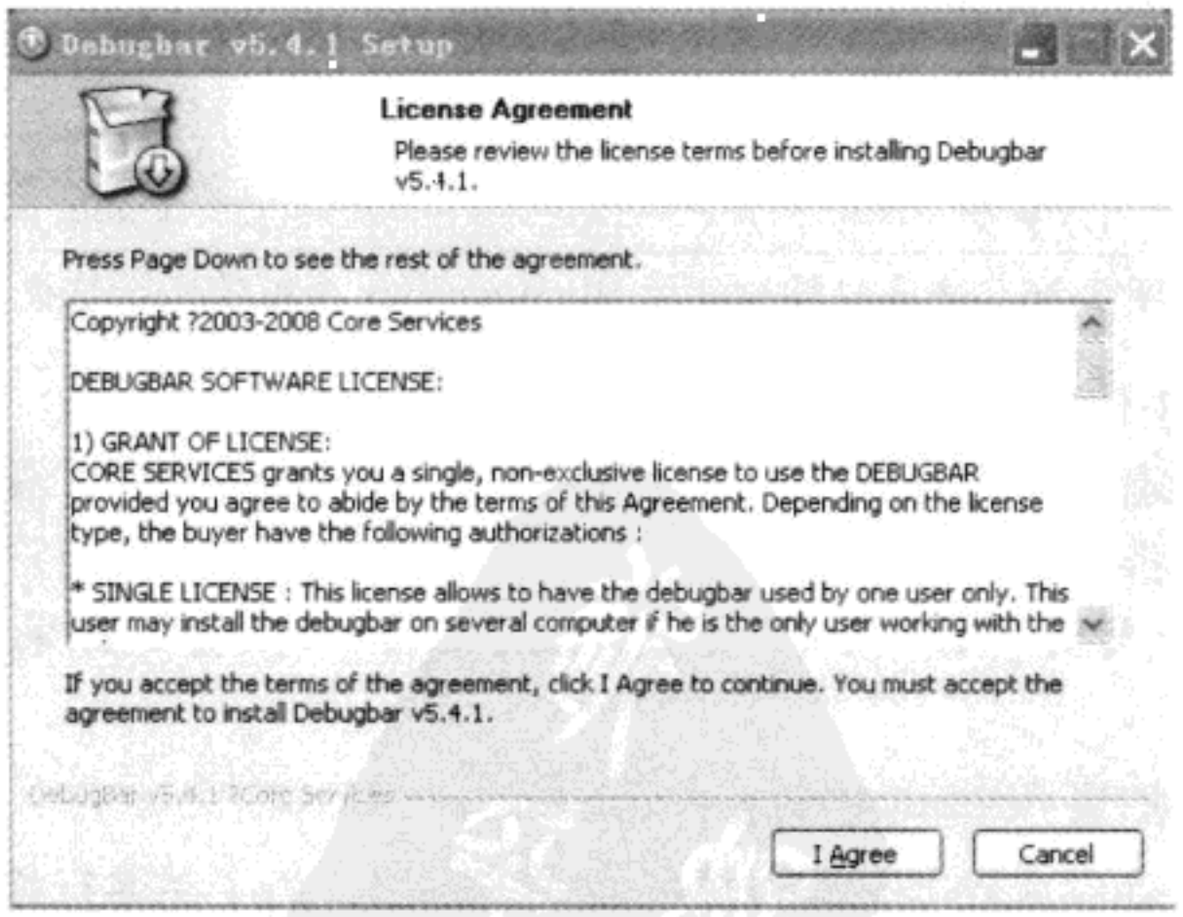


图 3-27 Debugbar 安装界面

单击“I Agree”按钮进入如图 3-28 所示的组件选择窗口。

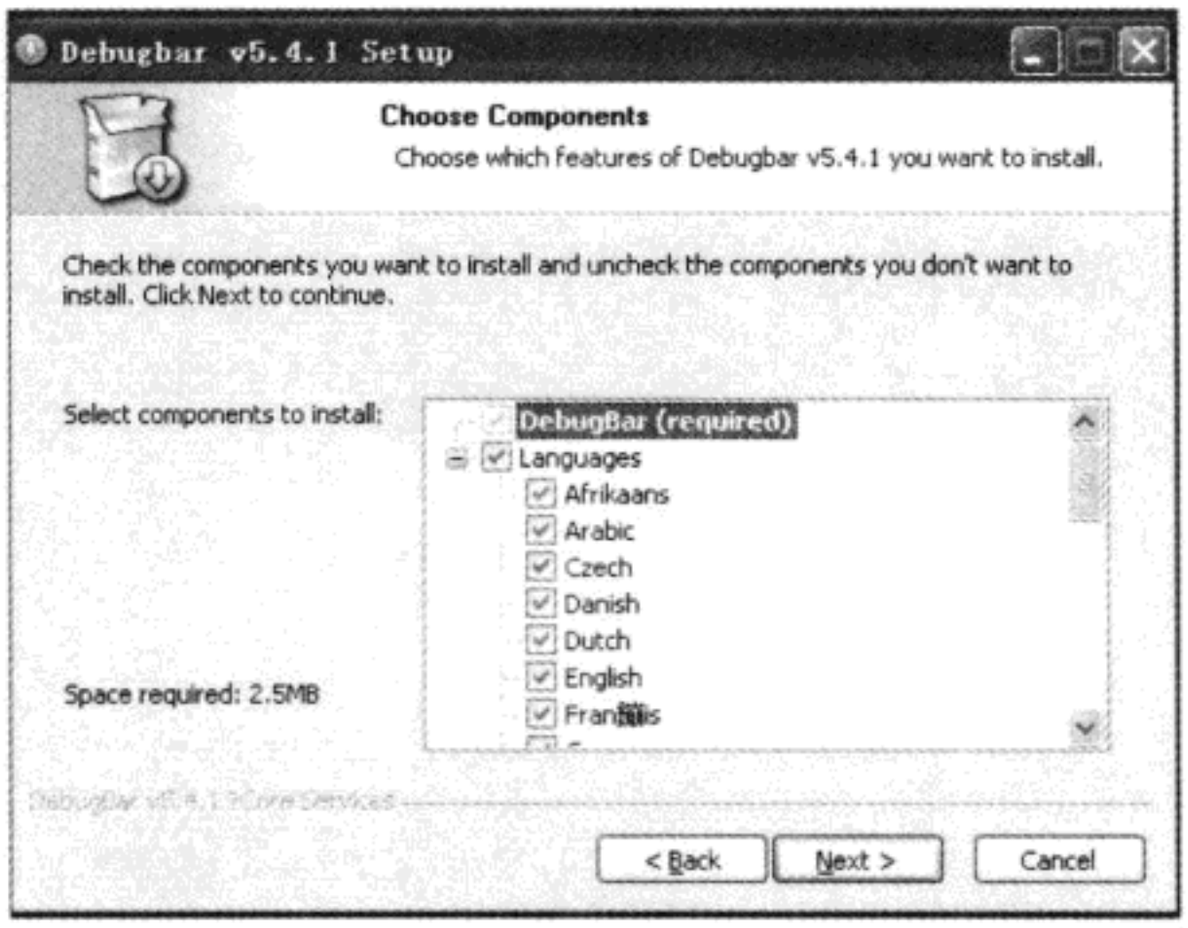


图 3-28　组件选择窗口

单击“Next”按钮进入安装目录选择窗口。

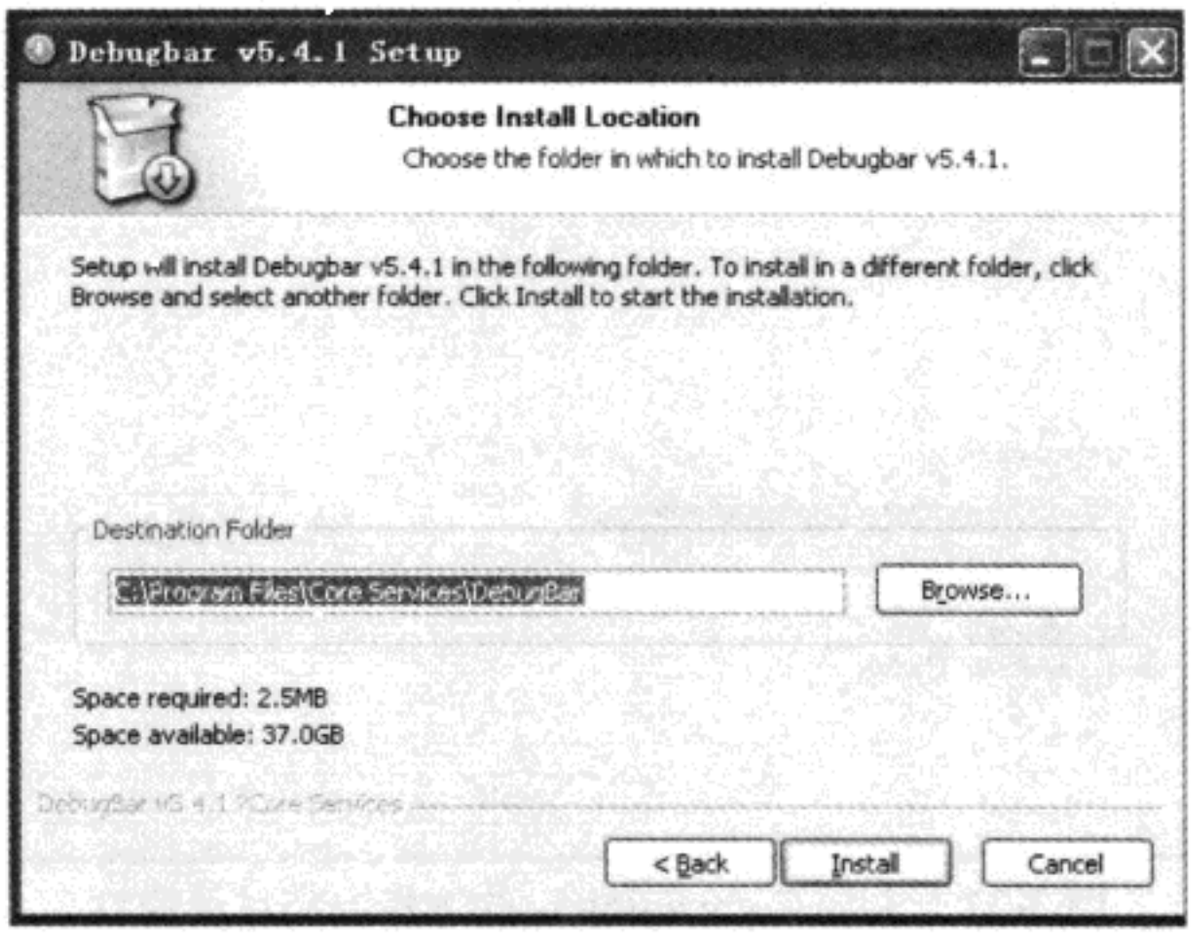


图 3-29　安装目录选择窗口

单击“Install”按钮开始安装，安装完成后会弹出一个 IE 窗口。

现在安装 Companion.js，运行下载的 install-companionjs-v0.5.5.exe 文件，将看到如图 3-30 所示的窗口。

单击“Next”按钮进入如图 3-31 所示的安装目录选择窗口。

单击“Install”按钮开始安装。等待出现了如图 3-32 所示的安装完成窗口，则单击“Close”按钮结束安装。

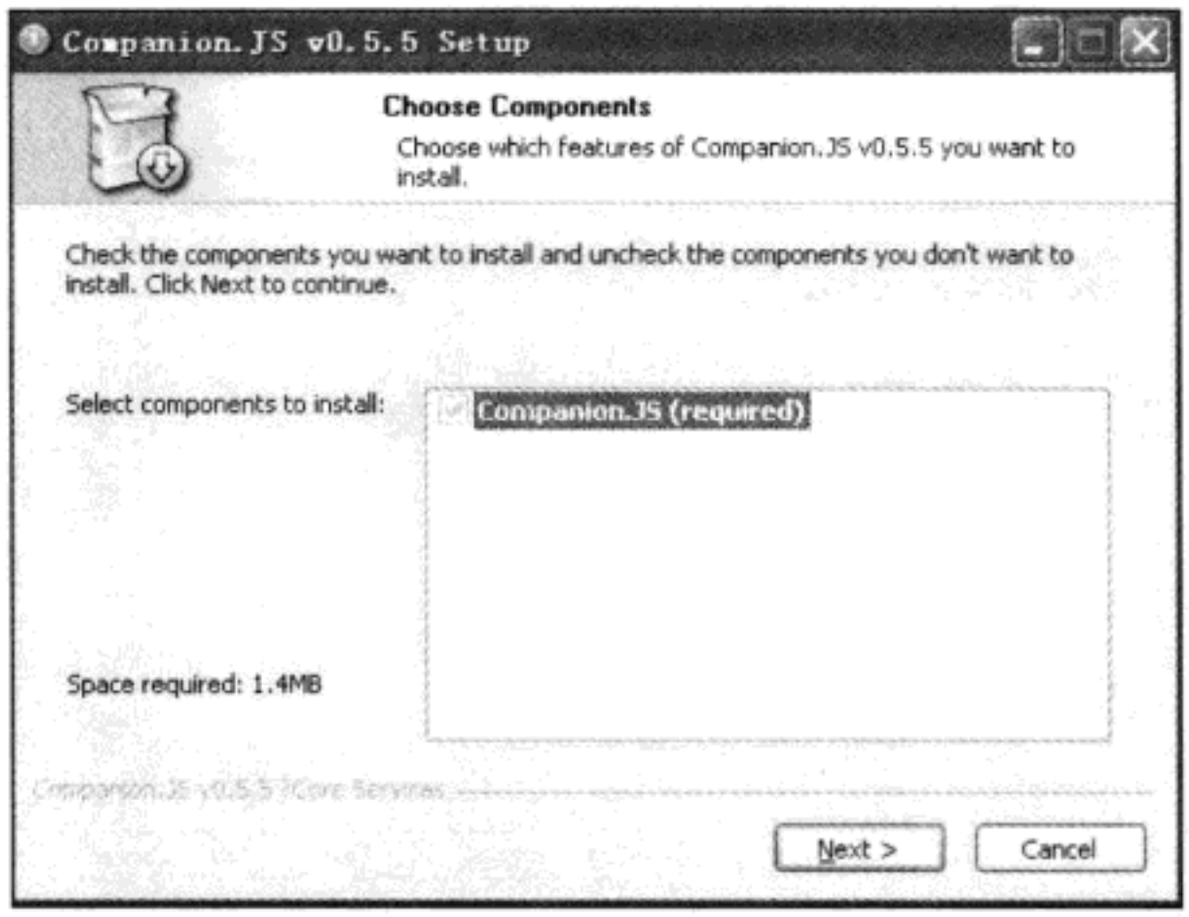


图 3-30　Companion.js 的安装窗口

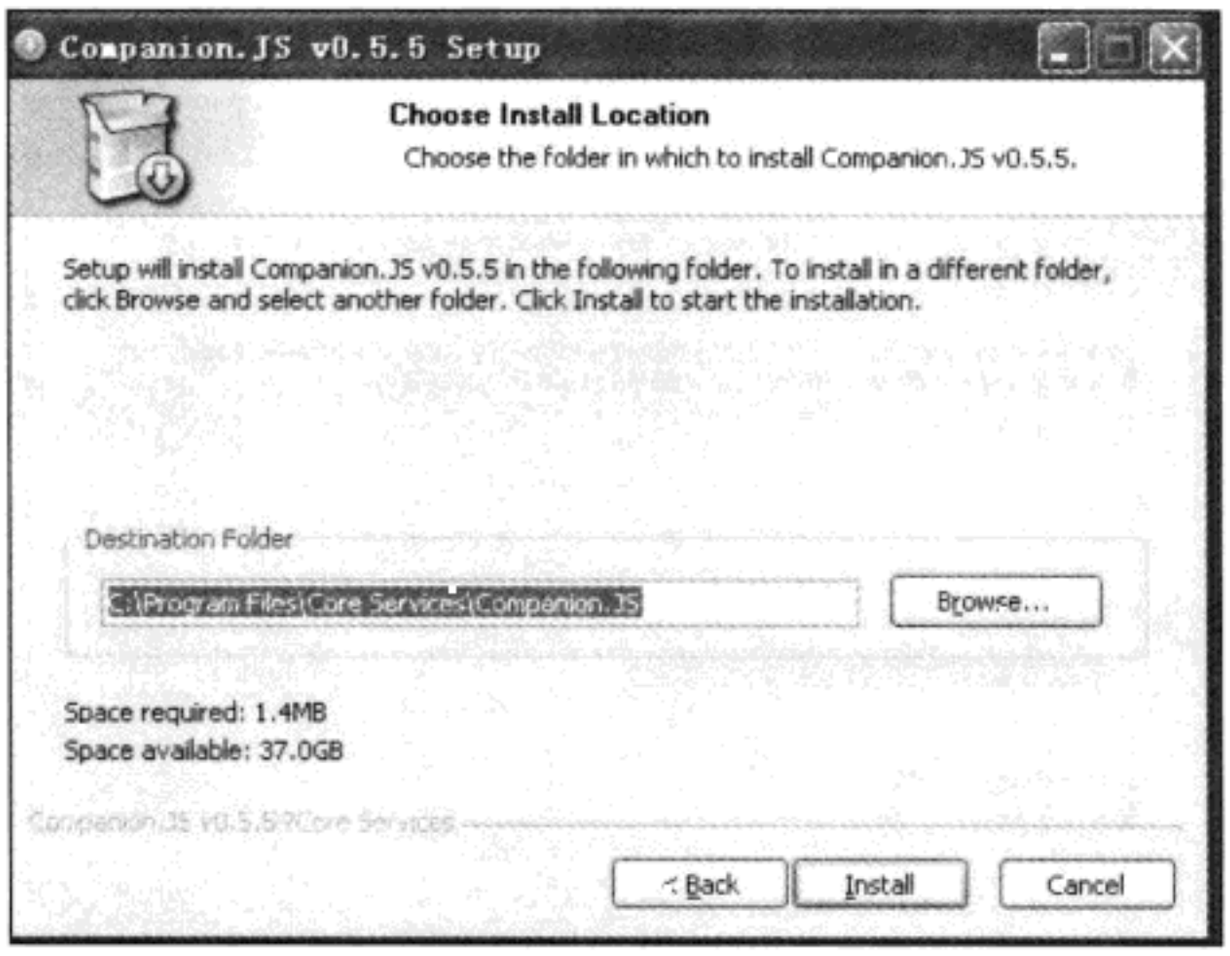


图 3-31　安装 Companion.js 的选择安装目录窗口

现在软件安装完了，可以使用了。

3. 使用

要在 IE 中开启脚本调试，首先要在“Internet 选项”的高级选项中将“禁用脚本调试（Internet Explorer）”和“禁用脚本调试（其他）”两个选项的勾去掉，如图 3-33 所示。

接着在 IE 工具栏里单击鼠标右键，在右键菜单中选择 Debugbar，打开 Debugbar 工具栏。在 IE 的查看菜单中打开浏览器栏的子菜单，然后选择“Companion.js”可打开“Companion.js”窗口，最终结果如图 3-34 所示。

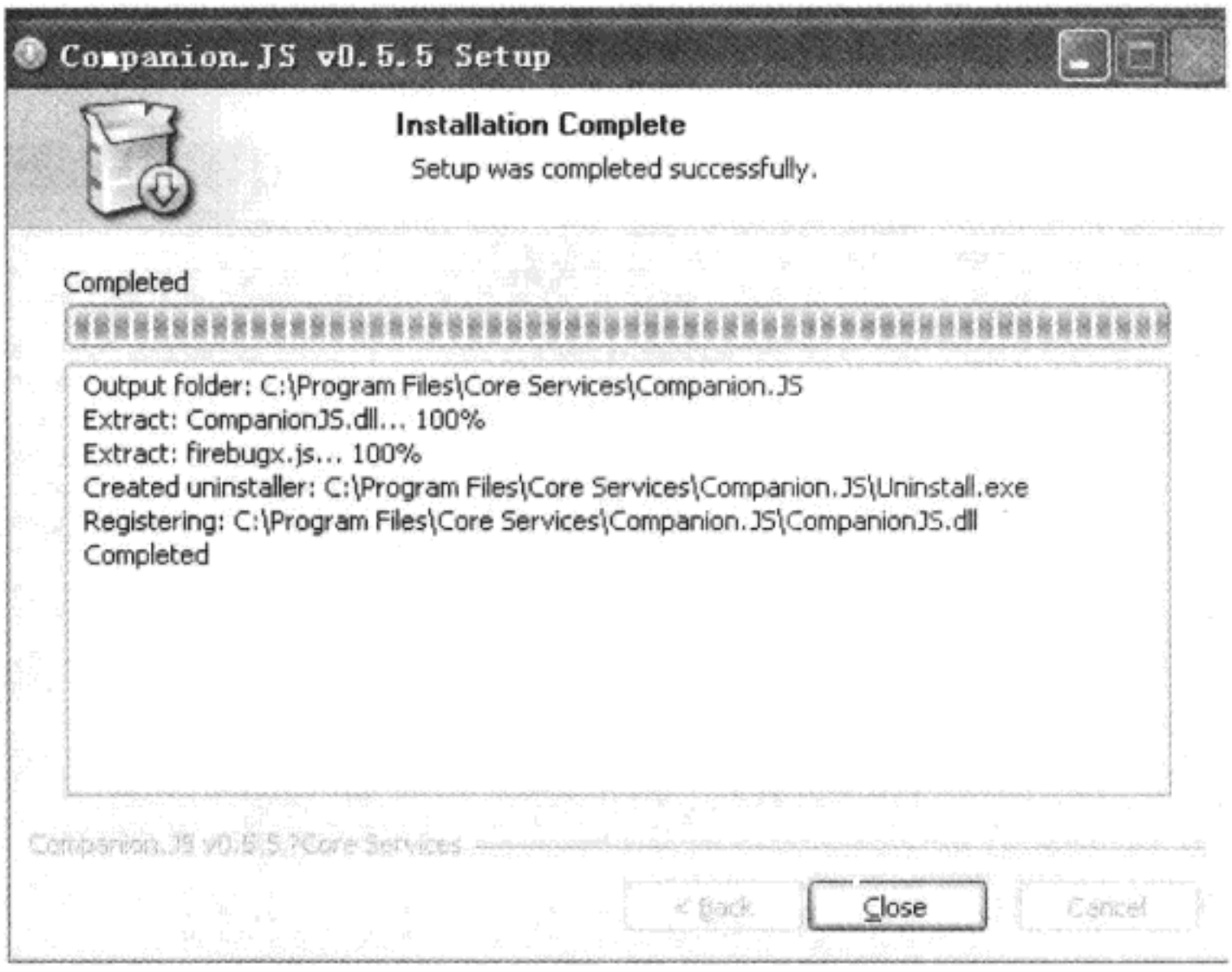


图 3-32　安装完成窗口

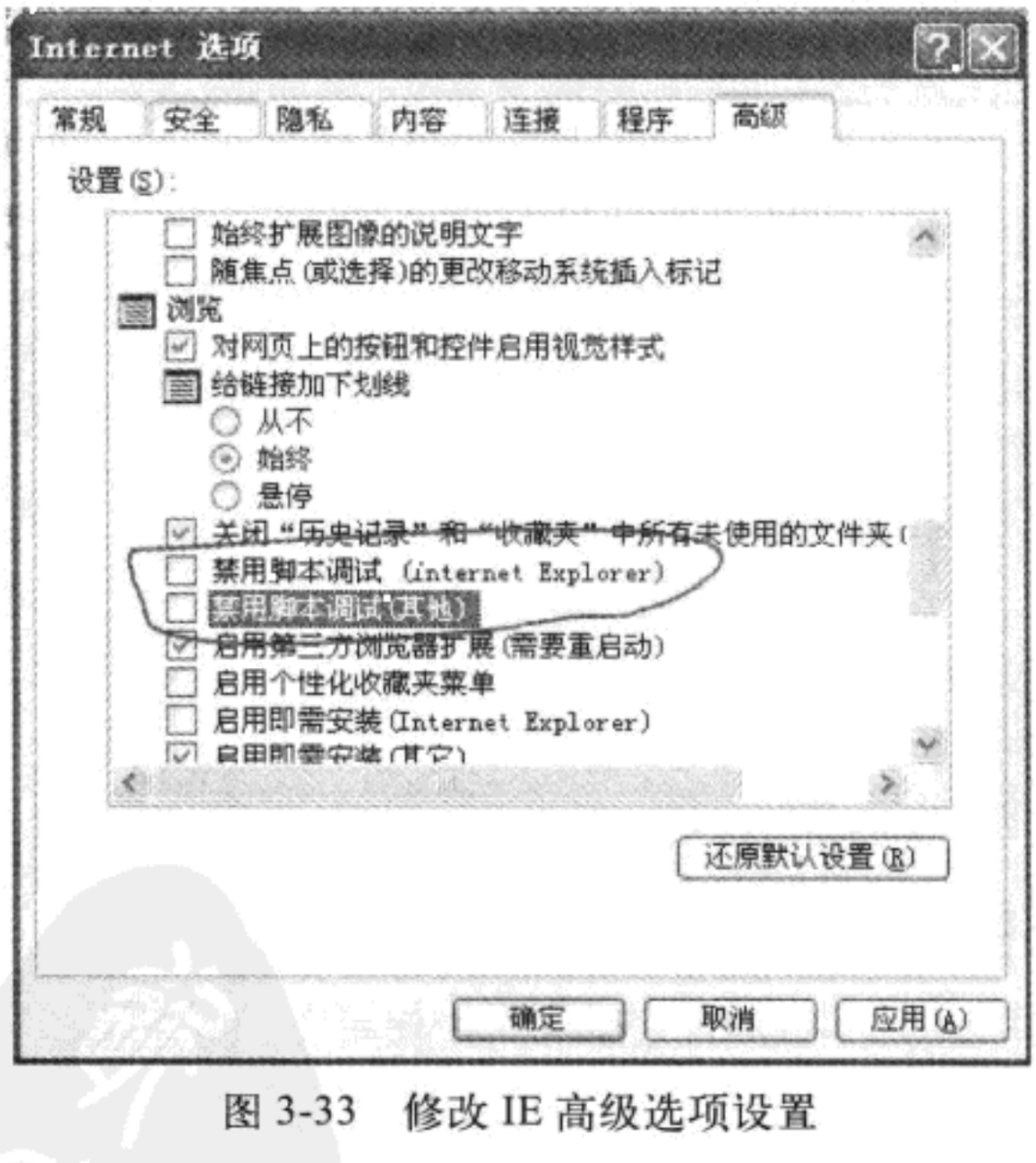


图 3-33　修改 IE 高级选项设置

如图 3-35 所示，在 DOM 选项卡中可查看页面的 DOM 节点。选择一个 DOM 节点后，可在下面的窗口查看节点的源代码（Source）、样式（Style）、计算出的样式（Comp. Style）、布局（Layout）和属性（Attrs）。通过拖动 DOM 标签下的图标可在页面中选取元素。除了在属

性窗口内可以修改属性值外，其余地方不允许进行任何修改，可以说基本不具备任何 HTML 和 CSS 调试功能。

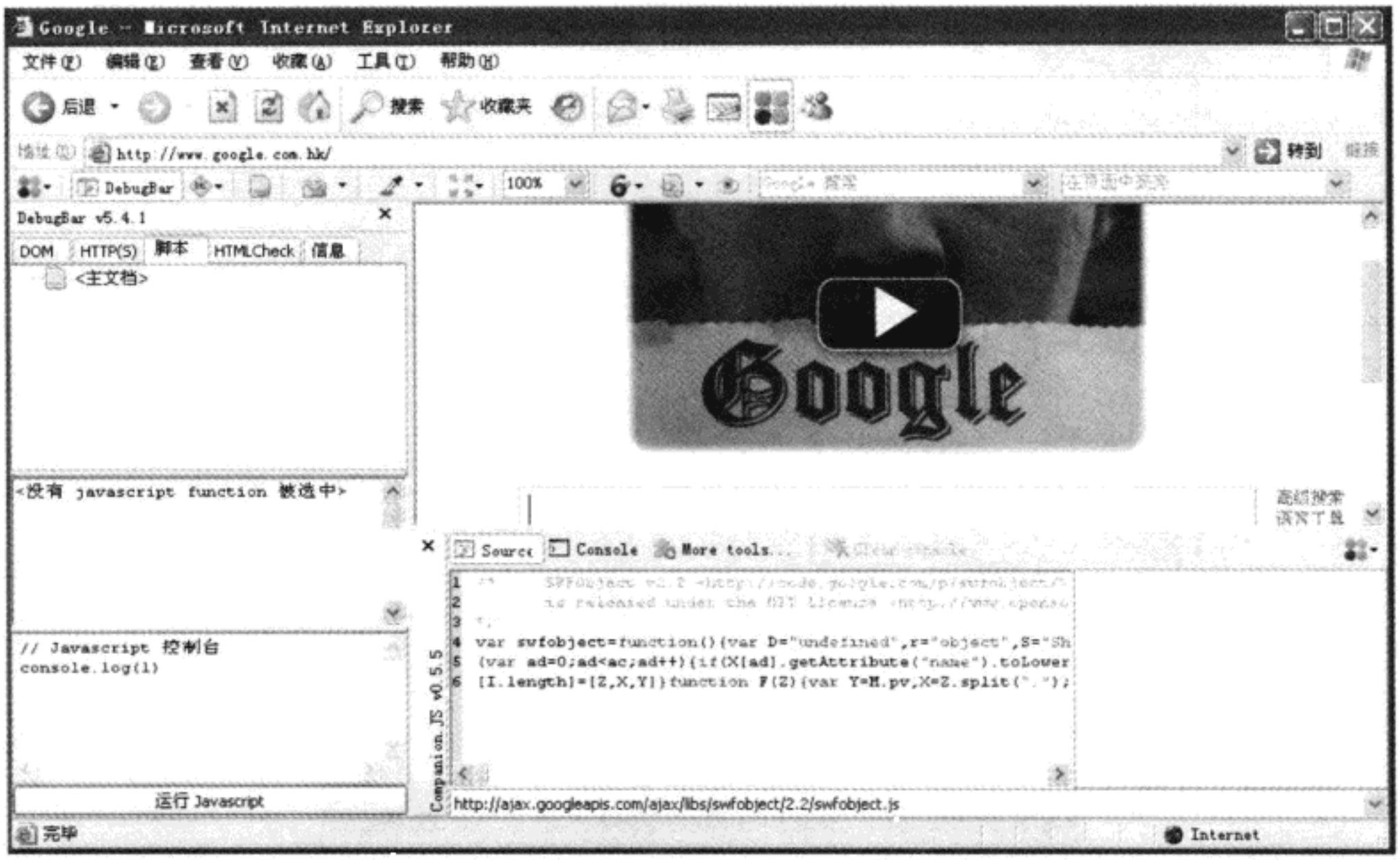


图 3-34 Debugbar 工具栏

HTTP（S）标签有点类似 Firebug 的网络面板，不过不太直观。脚本标签内不具备 Firebug 那样的单步调试功能，只是兼容了 Firebug 的调试语句。HTML Check 标签是进行 HTML 验证时用的。

信息标签显示的是页面的 URL、标题、页面大小、下载事件等信息。

总体来说，Debugbar 和 Companion.js 只能作为当 Firefox 中没有错误，而在 IE 中有错误时的辅助调试工具，主要调试工具还是应该使用 Firebug。

图 3-35 DOM 选项卡

3.2.2 使用 IEtester 测试

感谢 Debugbar 公司提供了 IEtester 这款工具，使用它就可以在一个软件内调试页面而在不同 IE 版本中显示结果了。

要使用 IEtester，请首先访问 http://www.my-debugbar.com/wiki/ 下载安装文件 install-ietester-v0.4.8.exe，然后运行该安装文件，将看到如图 3-36 所示的安装窗口。

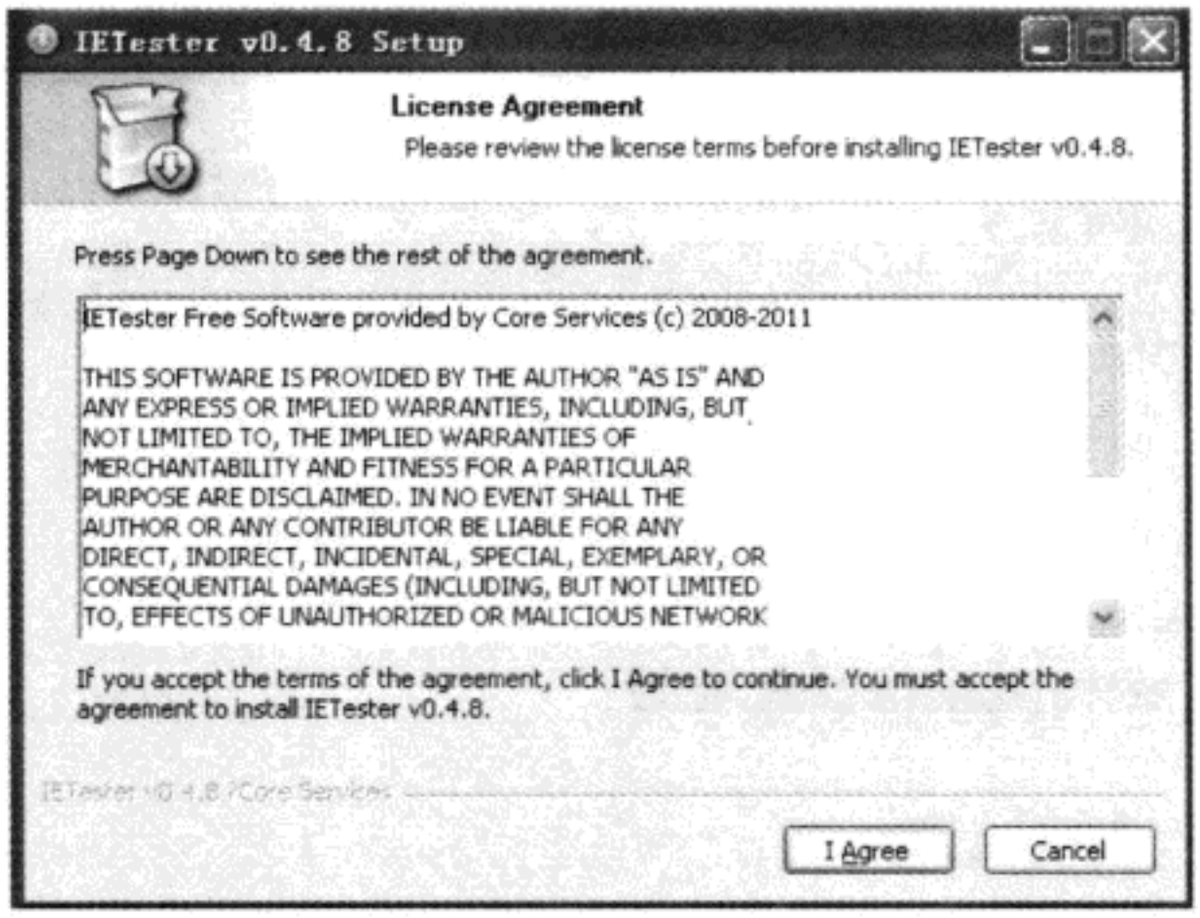


图 3-36　IEtester 安装界面

单击“I Agree”按钮，进入如图 3-37 所示的组件选择窗口。

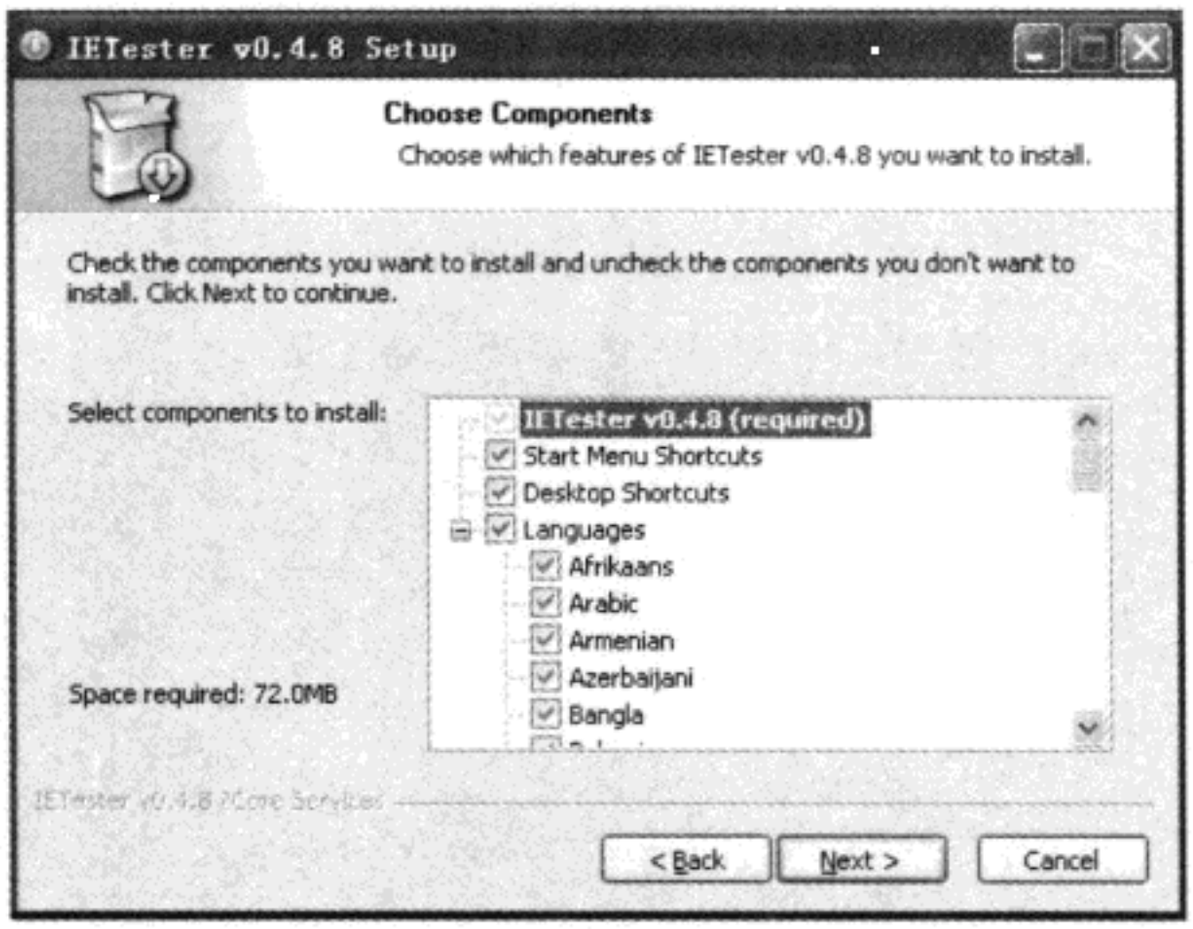


图 3-37　IEtester 安装的选择组件窗口

单击“Next”按钮进入如图 3-38 所示的安装目录选择窗口。

单击“Instal”按钮开始安装软件，安装完成后会在桌面看到一个 IEtester 的快捷方式，双击该快捷方式打开如图 3-39 所示的 IEtester 窗口。

因为 Windows XP 不支持 IE 9，所以在图 3-39 中，IE 9 的图标是灰色不可选择的。要在 IE 9 中调试，还是要安装 Vista 或 Win 7。

单击工具栏中各版本 IE 的按钮就可在标签中添加一个该版本的 IE 标签，例如，单击 IE 6 按钮，就会看到标签栏中多了一个如图 3-40 所示的标签页，标签的图标表示正在使用的 IE 版本。

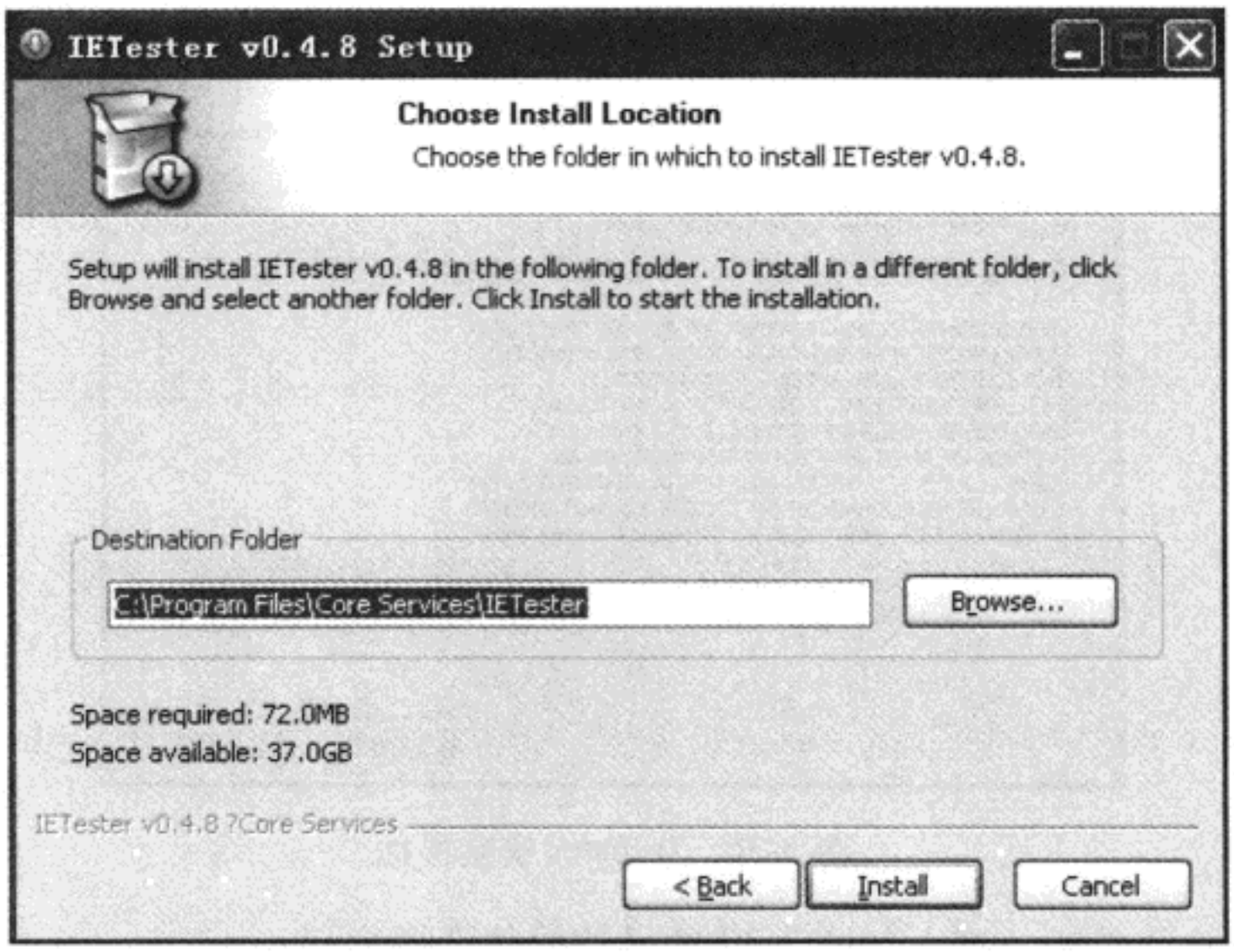


图 3-38　IEtester 安装的选择安装目录窗口

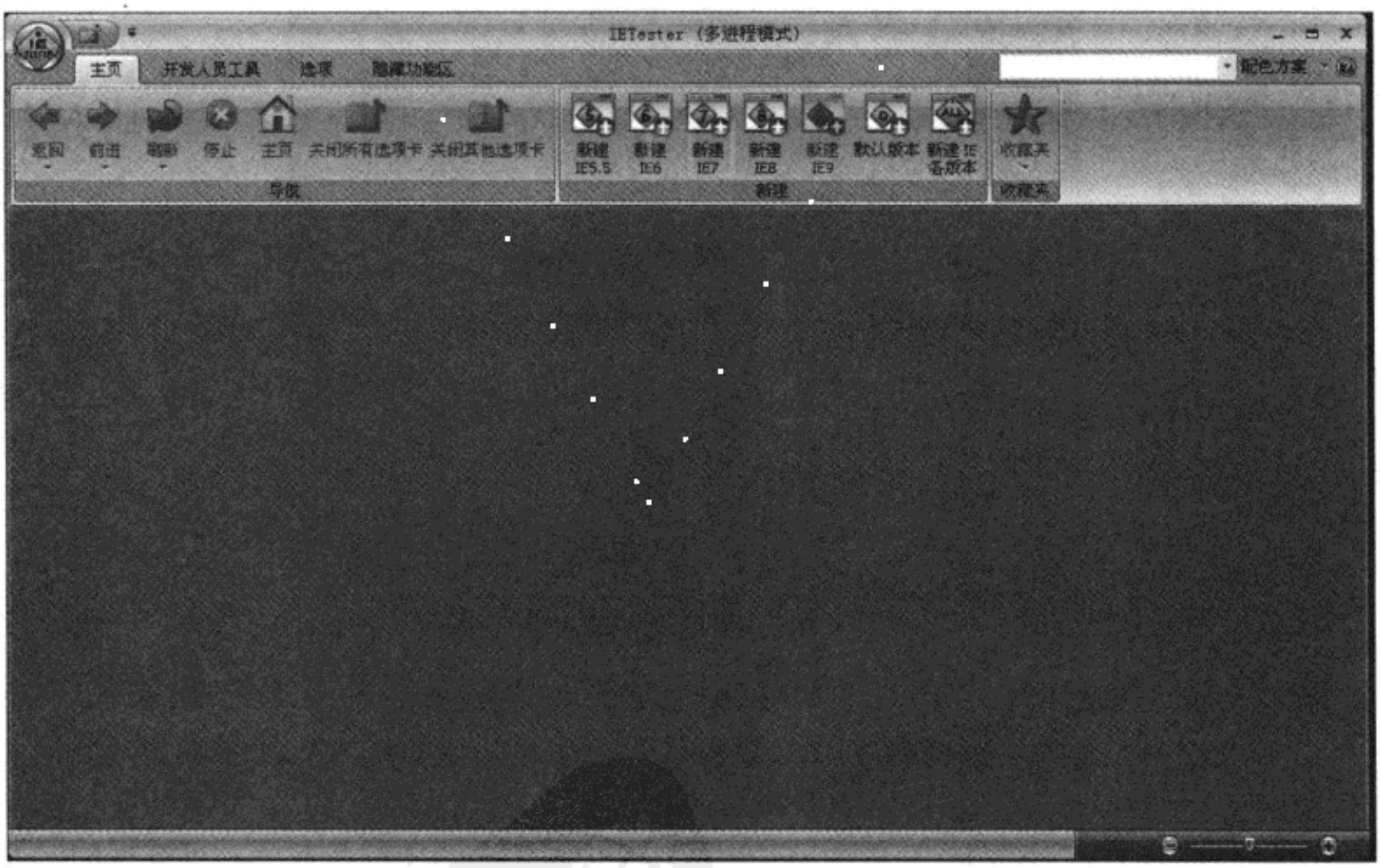


图 3-39　IEtester 窗口

图 3-40　IE 6 的标签

在一个系统内使用不同版本 IE 的引擎的原理很简单，IE 的引擎其实就是一些动态链接库（DLL），只要使用对应的动态链接库打开页面就相当于使用不同版本的 IE 打开页面。不过，这样存在兼容性问题，IEtester 时不时会崩溃。但问题不大，关键是能解决大问题。不然，你就需要安装多个虚拟机，然后在不同虚拟机上安装不同的 IE 进行调试了。

3.2.3　在 IE 8 或 IE 9 中调试

浏览器大战的压力让微软坐不住了，由于越来越多的开发者选择了 Firefox 和 Firebug，微软终于在 IE 8 中加入了调试工具。和 Firebug 一样，在 IE 8 按下 F12 可在浏览器中打开如图 3-41 所示的调试窗口。而在 IE 9 中，调试窗口会如图 3-42 所示。

图 3-41　IE 8 中的调试窗口

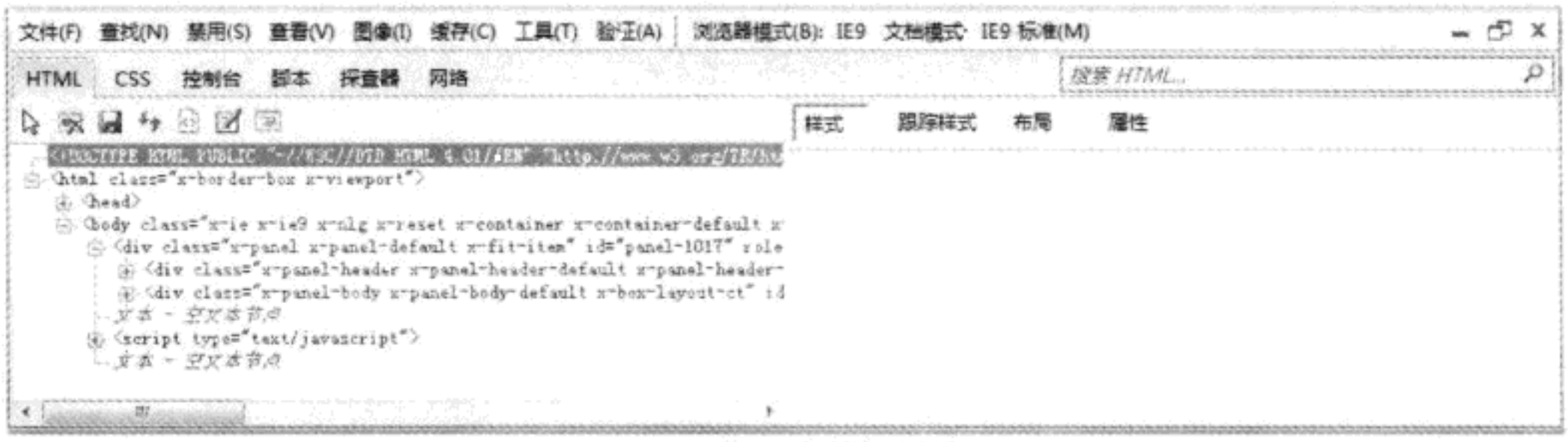


图 3-42　IE 9 中的调试窗口

对比一下 IE 8 和 IE 9，会发现多了控制台、网络两个标签页，IE 8 的控制台是作为脚本标签页的一个附加功能使用的。不过两个版本的调试工具都没有 DOM 查看器，其余的在外观上与 Firebug 差不多。

在 HTML 标签下，可看到页面的源代码，不过要单击工具栏的刷新按钮才可查看最终的源代码。HTML 标签中的功能和 Firebug 差不多，比 Debugbar 强，但就是微软风格太强烈。

单击标签下的箭头图标可以在页面中选择元素。在 HTML 标签和 CSS 标签的工具栏里都有一个“保存”按钮，可以保存修改，这在 Firebug 中需要通过附加扩展才能做。

在控制台中直接输入对象名，例如输入 window，可查看对象的部分信息，很大一部分信息使用省略号省略了。估计这是因为没有 DOM 面板造成的。至于为什么不提供 DOM 面板，笔者也实在想不明白。

脚本调试有个怪异的地方就是单击“启动调试”按钮后，必须将内嵌窗口变成独立窗口调试，而且只能开一个一个页面进行调试。不过，这也足够了。

探查器标签和 Firebug 控制台中的 JavaScript 执行时间概括功能相同，用来探查脚本的运行。

在 IE 9 中添加了网络标签，功能与 Firebug 的网络面板差不多。不过没有 Firebug 那么直观。

IE 的调试工具中还有个特殊功能，就是可以选择浏览器的模式，功能类似 IEtester，但不支持 IE 6。

总体来说，IE 自身的调试工具不如 Firebug，微软还有待努力。

3.3 在 Chrome 中调试

谷歌浏览器目前也很多人在使用，它自身带了一个如图 3-43 所示的调试工具，功能和 Firebug 大同小异。在 Chrome 主菜单中打开工具菜单，在子菜单中选择开发人员工具即可打开。与 Firebug 比较，它少了 DOM 查看器，不过在 Chrome 中查看 DOM 节点的功能比在 IE 中强，例如，在控制台输入 window，Chrome 中会以树结构显示 window 对象的所有属性，而不是像 IE 那样使用省略号。但控制台中没有多行命令行模式，非常不方便。不过对 Firebug 的命令支持比微软强大。

图 3-43 Chrome 中的调试工具

3.4 调试工具的总结

通过以上的介绍以及笔者的试用，**还是建议多使用 Firebug 做前期调试，做浏览器兼容性测试的时候再使用其他的调试工具**。

目前的 JavaScript 框架基本都是跨浏览器平台的，所以在 JavaScript 实现上不会存在太多的问题。只要在 Firefox 中调试通过，在其他浏览器中基本没有问题，不同之处可能具有页面

显示上的问题。如果在 Firefox 中通过了，而在 IE 9 之前的 IE 浏览器不能运行，那多半是在构建对象或数组的时候在结尾处多了一个逗号。只要在 IE 8 上做一下调试，应该就可以把问题找出来。

3.5　调试技巧

1. 使用 console 替代 alert

为什么要这样做？很简单，请看下面的代码：

```
var obj={data:[{name:'张三',age:30},{name:'李四',age:40}]};
```

如果你用 alert 调试，看到的是“[object,object]”，这对你根本没有任何帮助。如果你使用 console.log 输出，在 Firebug 控制台面板内可以看到“Object { data=[2]}”，单击对象可在 DOM 面板中看到如图 3-44 所示的结果。对比一下，console 命令的输出更具优势，尤其是在跟踪 Ext JS 数据的时候。

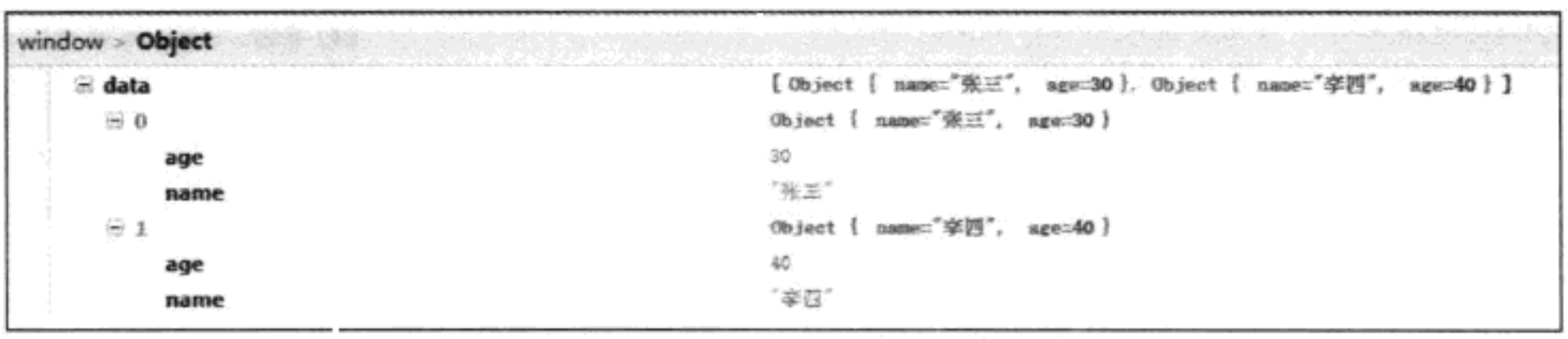


图 3-44　使用 console.log 后单击对象看到结果

再尝试一下使用 console.dir 输出 obj，可看到如图 3-46 所示的结果，这个命令比 log 更直观。

图 3-45　使用 console.dir 的输出结果

虽然这两个命令只有在 Firefox 中才能很好地使用，但这有什么关系呢？对于能提高生产力的事，何乐而不为？笔者认为装 Firefox 和 Firebug 的时间绝不会比你使用 alert 调试的时间多。

Alert 命令还存在一个很大的问题，就是在多次调用 alert 命令调试的时候，你得一次次地去按下“确定”按钮，而使用 console.log 则无需做任何事情，只要关注控制台就可以了。

console.log 的功能还不止于此，它还可以格式化输出，语法格式如下：

```
console.log('格式化字符串',arg1,arg2,…,argn);
```

可使用的格式字符如下：

- %s：字符串。
- %d，%i：数字。
- %f：浮点数。
- %o：对象。

例如，在命令行输入以下语句：

```
console.log('这是数字：%i\n这是 window 对象 :%o',3,window)
```

则在控制台的输出为：

```
这是数字：3
这是 window 对象 :Window index.html
```

console 命令还提供了类似信息对话框来使用图标区别不同提示信息的功能，例如，在命令行输入以下语句：

```
console.debug('无图标：console.debug!');
console.info('信息：Tconsole.info!');
console.warn('警告：console.warn!');
console.error('错误：console.error!');
```

运行后会看到如图 3-46 所示的结果。

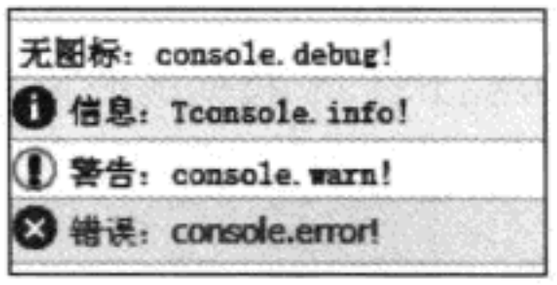


图 3-46　类似信息对话框的提示信息

console 还提供一些其他的命令，其详细说明如表 3-5 所示。

表 3-5　console 命令及其说明

命　令	说　明
console.assert(expression[,object,…])	断言。如果表达式为 false，则输出例外信息到控制台
console.dirxml(node)	列出 HTML 或 XML 元素的 XML 树
console.trace()	跟踪函数的调用入口
console.group(object[,object,..])	将信息分组再输出到控制台。通过 console.groupEnd() 结束分组
console.groupEnd()	结束分组输出
console.time(name)	创建一个名称为 name 的计时器，计算代码的执行事件。调用 console.timeEnd(name) 停止计时并输出执行事件
console.timeEnd(name)	停止名称为 name 的计时器的计时并输出执行事件
console.profile([title])	开始对脚本进行性能测试，title 为测试标题
console.profileEnd()	结束性能测试

2. 使用 debugger 命令

在脚本代码中插入 debugger 命令后，当代码执行到该位置时就会停止执行，等待用户操作，然后通过 Firebug 的脚本面板就可对代码进行单步调试了。

这是非常实用的命令，通过它可以跟踪代码的执行情况，并分析执行中出现的问题。

3. 显示问题的调试

当在页面中发现定义的 Ext JS 对象没有显示时，可按以下步骤调试：

- 检查代码是否正确运行了。

- 先在网络面板中检查需要的图片或数据是否已经下载。
- 检查数据是否正确。
- 如果只是在 IE 9 之前的 IE 中有问题，检查逗号问题。
- 使用元素选择功能检查元素是否已生成。如果已经生成，则可检查元素的宽度和高度，以及位置是否正确。如果没有生成，一般是因为代码没有正确执行。

显示问题最典型的情况就是 Grid 没显示数据，具体检查步骤如下：

- 在 Firebug 的网络面板中，检查服务器是否正确返回了数据，很多时候都是服务器端代码出了问题。
- 确认服务器代码能正确返回数据后，检查 Store 的 Reader 是否已正确将原始数据转换为以数据模型为基准的数据。如果没有正确转换，说明数据模型定义错误，可能是字段映射出了问题，也可能是返回的数据格式不正确，Reader 不能正确地从返回的文本提取数据。
- 如果以上都没问题，那么就是 Grid 的列定义与数据模型之间的映射存在问题，可以通过减少列定义的方法逐个排查。
- 如果不是列定义的问题，那就减少 Grid 的配置项，逐个排查 Grid 的配置项。
- 如果以上操作还不能解决问题，笔者只能说：God bless you ！

还有一个典型的例子是组件不显示或显示不正确，例如不应该换行的换行了，则检查步骤如下：

- 如果是动态加载的组件，在网络面板中检查服务器是否已正确返回数据，检查返回的数据是否正确。
- 在 HTML 面板中检查组件的 HTML 代码是否已正确生成，如果没有，说明组件没有正确加载，可能是因为代码没有被正确执行或代码存在错误。
- 如果 HTML 代码存在，那么请检查 HTML 代码的样式，看是否是宽度不够或被隐藏了。如果宽度不够，调整宽度；如果是隐藏了，将检查隐藏组件的配置对象。
- 如果还是存在问题，请减少组件的配置项，从尽可能小的配置项开始，逐个添加配置项，就可找出是哪个配置项导致了错误。

4. 数据加载问题

当碰到数据加载问题时，可按照以下步骤检查：

- 在网络面板中检查是否提交了请求，若没有，则说明数据没有提交，请检查提交请求的代码。
- 在网络面板中检查提交地址和文件是否正确。
- 检查提交参数是否正确。
- 检查后台文件是否获取了正确的提交参数。
- 检查返回格式是否正确。
- 检查是否只执行了异常回调函数，如果是，则检查为什么会执行异常回调函数，可通过 console.log 输出回调函数的参数来检查。
- 如果以上操作还不能解决问题，先将配置项减到最少，检查是否能正确加载。如果可

以，逐步添加配置项，找出有问题的配置项。

5. 布局问题

当碰到布局问题时，可按照以下步骤检查：

- 检查布局是否按你预想的方式生成了 HTML 代码。
- 检查布局生成的源代码，看是否 是宽度和高度不足。
- 检查是否使用了错误的布局。
- 检查是否使用了多余的布局。
- 将布局的配置项减到最少，然后进行测试，如果正确，逐步添加配置项，找出出现问题的配置项。

6. 对象或变量不存在的错误

当碰到对象或变量不存在的错误时，可按照以下步骤检查：

- 检查对象或变量是否已定义。
- 是否使用 scope 属性传递了作用域。
- 使用 console.log 跟踪处理的整个过程，检查过程中从哪个步骤开始出现对象或变量不存在的情况。
- 如果语句中使用了多个对象，就逐个检查问题出在哪个对象上，例如从 Grid 中获取选择的记录，就需要检查 Grid、GridView 和选择模型对象是否存在，以及选择模型是否正确。

7. 事件没有响应

当事件没有响应时，可按照以下步骤检查：

- 检查事件是否被正确绑定，方法是使用 console.log 在控制台中输出对象，然后在 DOM 面板中查看对象的 events 对象中对应的事件是否已绑定函数。也可以在 Illuminations 面板中选择控件，然后在 Events 子面板中查看是否绑定了事件。
- 在 API 中检查对象是否存在该事件，以及详细了解该事件的触发条件。
- 检查事件中的代码是否存在错误。

8. 诊断 Ext JS 4.1 的布局问题

因为 Ext JS 4.1 采用了批量渲染方式，因而会造成不能被发现的错误或出现轻微的视觉异常，有些时候会导致渲染失败，因而在 Ext JS 4.1 Beta 1 的文件包中提供了 ext-all-dev.js、Context.js 和 ContextItem.js 这三个文件用来诊断布局问题。使用方法是在页面中使用 ext-all-dev.js 文件代替 ext-all.js 文件，并加入另外两个文件的引用，代码如下：

```
<script type="text/javascript"src="/path/ext-all-dev.js"></script>
<script type="text/javascript"src="/path/src/diag/layout/Context.js"></script>
<script type="text/javascript"src="/path/src/diag/layout/ContextItem.js">
```

代码中 path 为文件所在目录与脚本目录的相对路径。加入这些脚本后，就可以在 Firebug 的控制台中看到诊断日志。如果是 IE，则需要在地址栏中输入以下代码来查看：

```
javascript:void(Ext.log.show())
```

该日志有行数限制，默认是 750 行，如果需要扩大行数，可加入以下代码（修改为 1500 行）：

```
Ext.log.max=1500;
```

9. 使用 JSLint 检测脚本

（1）概述

JSLint 是 Douglas Crockford 提供的一个在线的脚本语法检测工具，可以检测出脚本中隐含的错误以及规范书写代码。可访问 http://www.jslint.com/ 使用该检测工具。因为是在线的，所以需要你将脚本复制到网页中进行检测。

（2）在 Visual Studio 中使用 JSLint

要在 Visual Studio 2005 和 2008 版中使用 JSLint，可访问 http://jslint.codeplex.com/releases/view/22496 来下载 JSLintVS_bin.zip 文件。解压后，如果是 Visual Studio 2005，将文件复制到“%My Documents%/Visual Studio 2005/Addins”目录中；如果是 Visual Studio 2008，将文件复制到“%My Documents%/Visual Studio 2008/Addins”目录中。打开 Visual Studio 后，就可在右键菜单中找到 JSlint 了。

如果是 Visual Studio 2010，可访问 http://jslint4vs2010.codeplex.com/ 来下载，下载的是一个扩展名为 .vsix 的文件，双击该文件安装后即可在 Visual Studio 2010 中使用 JSLint 了。

（3）使用 jslint4java 检测脚本

如果你想使用 Java 版的 JSLint，可访问 http://code.google.com/p/jslint4java/ 来下载最新版软件。

使用方法是：

```
% java -jar jslint4java-1.4.7.jar your.js
```

其中 your.js 是你要验证的脚本文件名称。

3.6　本章小结

本章介绍了 Firebug、Debugbar 等基于不同浏览器的调试工具，目的是让大家选择合适的调试工具来提高生产力。笔者不得不再次建议大家将调试环境切换到 Firefox 上，因为目前最好的调试器还是 Firebug（不是 Firebug lite）。对于跨平台的框架，只要脚本能在 Firefox 中通过，基本上可以适用于大部分浏览器，只是界面显示有差异而已。应用程序的重点是商业逻辑，而不是界面显示的问题，显示可以微调，商业逻辑的修改那就是大问题了。

在调试技巧一节中，笔者针对自己使用 Ext JS 的经验列出了一些错误检测步骤，因为笔者不可能接触到所有的错误，所以这些步骤不可能解决大家碰到的所有问题，希望读者能根据自己情况做出调整。

第 4 章　Ext JS 的基础架构

任何 JavaScript 框架要实现跨平台运行，就要建立一个基础架构。在基础架构中要检测框架运行平台的情况，以便框架在使用时根据这些平台信息使用正确的语法和命令。基础架构还要提供一套扩展方法以便扩展平台，无论平台是自己扩展还是让用户进行扩展。框架运行平台的制造商会根据自己的开发理念定义一套事件模型，因而基础架构也要将这些不同的事件模型封装起来，实现一套与不同事件模型无关、标准统一的事件模型，从而实现框架的跨平台运行。和事件一样，各浏览器使用的 DOM 架构也有差异，因而基础架构还要将不同的 DOM 操作封装起来，实现跨平台的 DOM 操作。

本章将介绍 Ext JS 的基础架构，包括 Ext JS 的命名控件、支持类、类的创建和管理，以及事件和事件管理。

4.1　Ext JS 框架的命名空间：Ext

4.1.1　概述

Ext 是 Ext JS 的命名空间，为 Ext JS 框架提供了唯一的一个全局对象。这样不仅便于代码的维护，也避免了与其他代码发生冲突。

Ext 的定义包含 Ext.js 和 Ext-more.js 两个文件。在这两个文件里包含了 Ext JS 的一些基础定义、基本的属性和方法以及四个重要的方法：apply、applyIf、override 和 extend。

4.1.2　apply 和 applyIf 方法

方法 apply 的作用是将所有配置对象的成员复制到对象，示例代码如下：

```
Ext.apply = function(object, config, defaults) {
    if (defaults) {
        Ext.apply(object, defaults);
    }

if (object && config && typeof config === "object") {
    var i, j, k;

    for (i in config) {
        object[i] = config[i];
    }

    if (enumerables) {
        for (j = enumerables.length; j--;) {
```

```
                k = enumerables[j];
                if (config.hasOwnProperty(k)) {
                    object[k] = config[k];
                }
            }
        }
    }

    return object;
};
```

如果默认配置对象 defaults 存在，先递归调用 apply 方法进行复制。

如果对象存在、配置对象（config）存在，且配置对象的类型为“object”，开始进行复制。复制过程很简单，使用 for…in 语句遍历配置对象中的成员，然后将其复制到对象。

一些旧的浏览器的 for…in 语句不会枚举从 Object 对象继承的 hasOwnProperty、valueOf、isPrototypeOf、propertyIsEnumerable、toLocaleString、toString 和 constructor 等属性，因而需要进行额外的复制。而 enumerables 属性的作用就是判断是否要进行额外的复制，其相关代码如下：

```
toString = Object.prototype.toString,
enumerables = true,
enumerablesTest = { toString: 1 },
for (i in enumerablesTest) {
    enumerables = null;
}

if (enumerables) {
    enumerables = ['hasOwnProperty', 'valueOf', 'isPrototypeOf',
        'propertyIsEnumerable',
                'toLocaleString', 'toString', 'constructor'];
}
Ext.enumerables = enumerables;
```

如果浏览器 for…in 语句支持枚举 Object 对象原型的 toString 属性，说明不需要进行额外的复制，因而 enumerables 的值被修改为 null。如果 enumerables 为 true，说明要进行额外的复制，因而会指向一个数组，数组里包含了需要进行额外复制的 7 个属性。

如果 enumerables 属性不为 null，则要进行额外复制。在复制的 for 循环中，需要使用 hasOwnProperty 方法判断属性是否包含在配置对象中，如果包含，则复制。

在 apply 的 for…in 循环中，无论配置对象中的成员是否已在 object 对象中，都会进行复制，这就存在问题了，例如若配置对象已经将成员复制到对象，对象在初始化过程中会改变一些配置属性，如果使用 apply 方法，就会覆盖原来的值，从而造成不是开发人员预期的效果。因而这时候就需要使用 applyIf 方法，其代码如下：

```
applyIf: function(object, config) {
    var property;

    if (object) {
        for (property in config) {
```

```
                if (object[property] === undefined) {
                    object[property] = config[property];
                }
            }
        }

        return object;
    },
```

粗体代码的意思是指，只有在对象的属性值为 undefined（对象中不包含该关键字或其值为 undefined）的时候才进行复制。

方法 apply 和 applyIf 在 Ext JS 的类中使用广泛，因而了解它们对阅读 Ext JS 的源代码非常有用。

4.1.3 不推荐的 extend 方法

在 Ext JS 4 之前的版本中，所有类及用户自定义的扩展基本都是使用 extend 方法定义的。在 Ext JS 4 中，因为对类系统进行了重新架构，因而除了个别类外，基本都使用 define 方法来定义类，也就不再推荐使用 extend 方法。不过了解它对了解 Ext JS 还是有帮助的。

方法 extend 的源代码如下：

```
extend: function() {
    var objectConstructor = objectPrototype.constructor,
        inlineOverrides = function(o) {
    for (var m in o) {
        if (!o.hasOwnProperty(m)) {
            continue;
        }
            this[m] = o[m];
        }
};

return function(subclass, superclass, overrides) {
        if (Ext.isObject(superclass)) {
        overrides = superclass;
        superclass = subclass;
        subclass = overrides.constructor !== objectConstructor ? overrides.
            constructor : function() {
            superclass.apply(this, arguments);
        };
    }

    if (!superclass) {
   // 省略抛出异常代码
}

var F = function() {},
    subclassProto, superclassProto = superclass.prototype;
F.prototype = superclassProto;
subclassProto = subclass.prototype = new F();
subclassProto.constructor = subclass;
```

```
subclass.superclass = superclassProto;
if (superclassProto.constructor === objectConstructor) {
    superclassProto.constructor = superclass;
}
subclass.override = function(overrides) {
    Ext.override(subclass, overrides);
};
subclassProto.override = inlineOverrides;
subclassProto.proto = subclassProto;

            subclass.override(overrides);
            subclass.extend = function(o) {
                return Ext.extend(subclass, o);
            };

            return subclass;
        };
    }(),
```

首先要注意最后一行，这说明 extend 采用了闭包方式。

变量 objectPrototype 的定义如下：

```
objectPrototype = Object.prototype,
```

因而变量 objectConstructor 是指向 Object 对象原型的构造属性。

函数 inlineOverrides 的作用就是将对象 o 的成员复制给 this 指针。

在返回的函数中，代码首先使用 isObject [⊖]方法判断用户使用的是哪种定义方式。

Ext JS 扩展类的方式主要有以下两种：

```
subclass=Ext.extend(superclass,overrides);
Ext.extend(subclass,superclass,overrides);
```

在第 1 种定义方式中，参数 superclass 是配置对象，是 JavaScript 对象，因而返回值将会是 true。因而需要进行参数调整。在第 2 种方式中，superclass 是 Ext JS 对象，因而不需要调整。

在第 1 种方式中，如果配置对象的构造属性不同于 Object 对象原型的构造属性，则会将 subclass 指向配置对象的构造属性。否则会重新构造一个构造属性，函数内会调用执行其父类的构造属性。

接下来使用寄生组合式继承方式[⊜]创建子类。

首先是创建一个函数 F，将它的原型指向 superclass 的原型。然后使用 F 的实例，也就是 superclass 的原型实例作为 subclass 的原型。接着为原型添加构造属性，以弥补因原型重写而失去的默认的构造属性。如果 superclass 原型的构造属性与 Object 的构造属性相同，则将 superclass 原型的构造属性指向 superclass。

接着为 subclass 添加重写函数（override）。而 subclass 原型的重写函数使用在函数开始位置定义的 inlineOverrides 函数。两个函数的区别就是 inlineOverrides 函数只有当成员存在时才

⊖ 相关内容可阅读 4.1.4 节的第 3 小点。

⊜ 相关内容可阅读《JavaScript 高级程序设计（第 2 版）》的 6.2.6 节。

进行复制。直接将 subclass 原型的 proto 属性指向 subclass 的原型。

接下来的一步很重要，通过 override 方法将配置对象的成员复制到 subclass 的原型，这样继承于 subclass 的类就可以通过 subclass 的原型继承其成员了。

最后定义 subclass 的 extend 属性，返回 subclass。

以上内容可能不太好理解，建议多看看相关资料。

根据 ext-all-debug.js 文件的内容，在 4.1 中只剩 Version 对象是使用该方法扩展了。为了使旧的类兼容新的类系统，在 Class.js 中，重写了 extend 方法，其代码如下：

```
Ext.extend = function(Class, Parent, members) {
    if (arguments.length === 2 && Ext.isObject(Parent)) {
        members = Parent;
        Parent = Class;
        Class = null;
    }

    var cls;

    if (!Parent) {
        throw new Error("[Ext.extend] Attempting to extend from a class which
            has not been loaded on the page.");
    }

    members.extend = Parent;
    members.preprocessors = [
        'extend'
        ,'statics'
        ,'inheritableStatics'
        ,'mixins'
        ,'config'
    ];

    if (Class) {
        cls = new ExtClass(Class, members);
    }
    else {
        cls = new ExtClass(members);
    }

    cls.prototype.override = function(o) {
        for (var m in o) {
            if (o.hasOwnProperty(m)) {
                this[m] = o[m];
            }
        }
    };

    return cls;
};
```

还是要先检查类的定义格式，如果只有 2 个参数，则需要调整参数。

与旧的 extend 不同，调整后的 subclass 设置为 null，这是因为后续要按新类的创建方式

创建类。如果父类 Parent 不存在，抛出异常。不然，在配置对象内添加 extend 属性，并指向继承的对象 Parent。接着添加 5 个处理器：extend、statics、inheritableStatics、mixins 和 config。如果 Class 存在，把 Class 作为新的 Class 实例的第一个参数。否则配置对象作为新的 Class 的第一个参数。有关 Class 实例的过程请阅读 4.4 节类的创建及管理的内容。最后为新类的原型添加重写方法 override，然后返回。

4.1.4　数据及其类型检测

1. typeOf：检测变量的类型

(1) 语法

```
Ext.typeOf(v);
```

其中，v 是要检测的变量；表 4-1 列出了可能的返回值。

表 4-1　typeOf 的返回值

返回值	变量类型
undefined	如果变量值为 undefined
null	如果变量值是 null
string	如果变量值是字符串
number	如果变量值是数字
boolean	如果变量值是布尔值
date	如果变量值是日期对象
function	如果变量指向一个函数
object	如果变量值是一个对象
array	如果变量值是一个数组
regexp	如果变量值是正则表达式
element	如果变量值是 DOM 元素
textnode	如果变量值是 DOM 的文本节点且包含内容
whitspace	如果变量值是 DOM 的文本节点且内容为空

(2) 源代码

```
if (value === null) {
    return 'null';
}

var type = typeof value;

if (type === 'undefined' || type === 'string' || type === 'number' || type
   === 'boolean') {
    return type;
}

var typeToString = toString.call(value);
```

```
switch(typeToString) {
    case '[object Array]':
        return 'array';
    case '[object Date]':
        return 'date';
    case '[object Boolean]':
        return 'boolean';
    case '[object Number]':
        return 'number';
    case '[object RegExp]':
        return 'regexp';
}

if (type === 'function') {
    return 'function';
}

if (type === 'object') {
    if (value.nodeType !== undefined) {
        if (value.nodeType === 3) {
            return (/\S/).test(value.nodeValue) ? 'textnode' : 'whitespace';
        }
        else {
            return 'element';
        }
    }

    return 'object';
}

// 省略抛出异常代码
},
```

如果值等于 null，则返回“null”。

接着使用 JavaScript 的 typeof 方法检测类型，如果是“undefined”、“string”、“number”或“boolean”等值，直接返回。

如果不是以上值，需要先将 typeof 返回的值转换为字符串。

如果是数组、日期对象、布尔对象、数字对象或正则表达式，直接返回对应值。

如果是函数，返回“function”。

如果是对象，但是没有 nodeType 属性，返回“object”。

如果 nodeType 属性存在，说明是 DOM 节点，如果节点类型是 3，说明是文本节点。然后使用正则表达式测试其内容是否为空，如果不为空，返回“textnode”，否则返回“whitespace”。

如果节点类型不是文本结点，返回“element”。

最后还检测不出类型，就只有抛出异常了。

从代码可以看出，Ext 的 typeOf 方法比 JavaScript 的 typeof 方法功能加强了，检测更准确了。

2. isEmpty：检测值是否为空

（1）语法

```
Ext.isEmpty(v,allowEmptyString)
```

其中，v 是要检查的值；allowEmptyString 的默认值是 false，表示不允许空字符串，反过来，空字符串也返回 true；当返回值为 true 时表示值为空，false 则表示值不为空。

（2）源代码

```
isEmpty: function(value, allowEmptyString) {
    return (value === null) || (value === undefined) || (!allowEmptyString ? value
        === '' : false) || (Ext.isArray(value) && value.length === 0);
},
```

代码检测了 4 种为空的状态：null、undefined、空字符串及数组长度为 0。

3. isObject：检测值是否为 JavaScript 的对象

（1）语法

```
Ext.isObject(v);
```

其中，v 是要检查的值。如果是 JavaScript 的对象，返回 true，否则返回 false。

（2）源代码

```
isObject: (toString.call(null) === '[object Object]') ?
function(value) {
        return value !== null && value !== undefined && toString.call(value) ===
'[object Object]' && value.nodeType === undefined;
    } :
    function(value) {
        return toString.call(value) === '[object Object]';
    },
```

因为有些浏览器的 null 值也是对象，因而代码首先要使用 Object 对象原型的 toString 方法检查 null 的返回值是否“[object Object]”，从而避免 null 值造成的错误。两个函数中，检查的关键语句是“toString.call(value)”，Ext JS 的可扩展类的返回值是“[object Function]”，而 JavaScript 对象的返回值是“[object Object]”，根据这个就可判断该对象是 Ext JS 对象还是 JavaScript 对象。

4. isIterable：检测值是否可迭代的

（1）语法

```
Ext.isIterable(v);
```

其中，v 是要检查的值。如果值的类型是迭代的，返回 true，否则返回 false。

（2）源代码

```
isIterable: function(value) {
    var type = typeof value,
        checkLength = false;
```

```
if (value && type != 'string') {
    if (type == 'function') {
        if (Ext.isSafari) {
            checkLength = value instanceof NodeList || value instanceof
                HTMLCollection;
        }
    } else {
        checkLength = true;
    }
}
    return checkLength ? value.length !== undefined : false;
}
```

因为字符串存在 length 属性，所以首先要排除字符串。因为函数也有一个 length 属性，所以也需要把函数排除在外。但在 Safari 浏览器中，NodeList 和 HTMLCollection 返回的类型也是“function”，而这两个对象是可迭代的，因而要将 checkLength 设置为 true。最后如果 checkLength 为 true，且 length 属性值不为 undefined，则是可枚举的，否则返回 false。

5. isFunction：检测值是否为函数

（1）语法

```
Ext.isFunction(v);
```

其中，v 是要检查的值。如果值的类型是函数，返回 true，否则返回 false。

（2）源代码

```
isFunction:
(typeof document !== 'undefined' && typeof document.getElementsByTagName('body')
    === 'function') ? function(value) {
    return toString.call(value) === '[object Function]';
} : function(value) {
    return typeof value === 'function';
},
```

因为 Safari 3.x 和 4.x 使用 typeof 检查 NodeList 对象是返回“function”的，所以需要先判断是否存在这种情况，如果存在，则使用 toString 方法检测。否则直接使用 typeof 方法检测。

6. 其他的检测方法

在 Ext 中还有以下几个检测方法：

- isArray：检查值是否为数组。
- isDate：检查值是否为日期对象。
- isPrimitive：检查值是否是 JavaScript 基本数据类型（字符串、数字或布尔值）。
- isNumber 与 isNumeric ：检测值是否为数字。两者的主要区别是，isNumber 检查的是数据类型，而 isNumeric 检查值是否是数字值，例如值为字符串“1.1”，isNmuber 返回 false，isNumeric 返回 true。因而，如果值的数据类型是数字，且不是无穷大或无穷小值，isNumber 返回 true，否则返回 false。如果值是非数字、无穷大或无穷小，

isNumeric 返回 false，否则返回 true。

- isString：检查值是否是字符串。
- isBoolean：检查值是否是布尔值。
- isElement：检查值是否为元素。
- isTextNode：检查值是否为文本节点。
- isDefined：检查值是否已定义（非 undefined）。

以上方法都比较简单，有兴趣可以自己研究。

4.1.5　其他的基础方法

1. Iterate

对数组或对象进行迭代。

（1）语法

```
Ext.iterate(object,fn,scope);
```

其中，object 是要进行迭代操作的数组或对象；fn 是要进行迭代操作的函数；scope 是作用域；该方法没有返回值。

（2）源代码

```
iterate: function(object, fn, scope) {
    if (Ext.isEmpty(object)) {
        return;
    }

    if (scope === undefined) {
        scope = object;
    }

    if (Ext.isIterable(object)) {
        Ext.Array.each.call(Ext.Array, object, fn, scope);
    }
    else {
        Ext.Object.each.call(Ext.Object, object, fn, scope);
    }
}
```

首先使用 Ext.isEmpty 方法判断 object 是否为空，为空的话返回空值。然后判断作用域 scope 是否不存在，如果不存在，把作用域指向 object。如果 object 是可迭代的，调用 Ext 的 Array 对象的 each 方法进行迭代。否则，使用 Ext 的 Object[⊖]对象的 each 方法进行迭代。

2. Clone

可以克隆数组、对象、DOM 节点和日期等数据，以避免保持旧的指向。

⊖　相关内容请阅读 4.3.2 节。

（1）语法

```
Ext.clone(item);
```

其中，item：要复制的数组、对象、DOM 节点或日期；返回值为克隆后的数组、对象、DOM 节点或日期。

（2）源代码

```
clone: function(item) {
    if (item === null || item === undefined) {
        return item;
    }
    if (item.nodeType && item.cloneNode) {
        return item.cloneNode(true);
    }
    var type = toString.call(item);
    if (type === '[object Date]') {
        return new Date(item.getTime());
    }
    var i, j, k, clone, key;
    if (type === '[object Array]') {
        i = item.length;
        clone = [];
        while (i--) {
            clone[i] = Ext.clone(item[i]);
        }
    }
    else if (type === '[object Object]' && item.constructor === Object) {
        clone = {};
        for (key in item) {
            clone[key] = Ext.clone(item[key]);
        }
        if (enumerables) {
            for (j = enumerables.length; j--;) {
                k = enumerables[j];
                clone[k] = item[k];
            }
        }
    }
    return clone || item;
},
```

如果要克隆的对象 item 为空或 undefined，直接返回 item。如果对象 item 具有 nodeType 属性和 cloneNode 方法，则说明是 DOM 节点，使用 cloneNode 方法克隆对象。

接着使用 toString 方法获取对象的类型。

如果类型是日期，创建一个新的日期对象。

如果是数组，在 while 循环中递归调用 Clone 方法克隆数据到新建 clone 数组。

如果是对象，使用 for…in 循环逐个将对象的成员复制到新对象中。因为成员的值也可能是对象或数组，所以需要递归调用 Clone 方法复制成员值，例如克隆以下结构的数组：

```
[{name:'张三,id:1'},{name:'李四,id:2'}];
```

如果不把数组内的对象也一起克隆的话，那么新数组内的元素的指向还是原来的对象指向，这就等于没有克隆。所以在这里需要递归调用，保证指向不是旧的数组或对象指向。

克隆对象的时候还要根据浏览器情况，处理 for…in 循环不能处理的成员。

如果 item 不是对象也不是数组，直接返回 item。否则返回 clone。

3. id

产生 id 值。非常实用的功能。

（1）语法

```
Exd.id([el,prefix]);
```

其中，el 是可选参数，是要添加 id 的元素，值可为元素 id 值、HTMLElement 对象或 Element 对象；prefix 也是可选参数，是自定义的 id 前缀字符串；如果 el 已经有 id 值，直接返回 el 的 id 值，如果没有，返回生成的 id 值。

（2）源代码

```
id: function(el, prefix) {
    var me = this,
        sandboxPrefix = '';
    el = Ext.getDom(el, true) || {};
    if (el === document) {
        el.id = me.documentId;
    }
    else if (el === window) {
        el.id = me.windowId;
    }
    if (!el.id) {
        if (me.isSandboxed) {
            if (!me.uniqueGlobalNamespace) {
                me.getUniqueGlobalNamespace();
            }
            sandboxPrefix = me.uniqueGlobalNamespace + '-';
        }
        el.id = sandboxPrefix + (prefix || "ext-gen") + (++Ext.idSeed);
    }
    return el.id;
},
```

首先使用 getDom[⊖]方法返回元素 Dom 对象，如果不存在，el 指向一个空对象（{}）。

如果 el 是 document 对象，使用“ext-document”作为其 id。

如果 el 是 window 对象，使用“ext-window”作为其 id。

如果 el 不存在 id 值，则生成一个。如果是沙盒模式，使用沙盒模式的前缀作为前缀，避免冲突。如果提供了前缀字符串 prefix，则生成字节为前缀字符串与 Ext.idSeed 种子结合的 id 值。默认使用“ext-gen”与 Ext.idSeed 种子结合的 id 值。保证 id 值的唯一，依赖的是 Ext.idSeed 种子数的不断累加，其初始值是 1000。

最后返回结果。

⊖　相关内容请阅读 6.2.4 节。

4. getBody

返回当前 document 对象的 body 元素。

（1）语法

```
Ext.getBody();
```

要注意，返回值是一个 Element 对象。

（2）源代码

```
getBody: function() {
    var body;
    return function() {
        return body || (body = Ext.get(document.body));
    };
}(),
```

要注意，这里是使用 Ext.get[⊖]方法返回 Element 对象。

5. getHead

返回 document 对象的 head 元素。

（1）语法

```
Ext.getHead();
```

要注意，返回值是一个 Element 对象。

（2）源代码

```
getHead: function() {
    var head;

    return function() {
        if (head == undefined) {
            head = Ext.get(document.getElementsByTagName("head")[0]);
        }

        return head;
    };
}(),
```

该方法使用了一个闭包的方法，目的是防止用户在当前 head 元素之前添加一个 head 元素，这样返回的还是原来的 head 元素[⊖]。

要注意，该方法也是使用 Ext.get 方法返回 Element 对象，而且只返回页面中第 1 个 head 元素的 Element 对象。

6. getDoc

返回 document 对象。

⊖ 相关信息可阅读 6.2.2 节。

⊖ 相关信息可参阅《JavaScript 高级程序设计（第二版）》的 7.2.1 节。

（1）语法

```
Ext.getDoc();
```

要注意，返回值是一个 Element 对象。

（2）源代码

```
getDoc: function() {
    var doc;
    return function() {
        return doc || (doc = Ext.get(document));
    };
}(),
```

要注意，这里是使用 Ext.get 方法返回的对象。

7. Destroy

删除对象及其事件，从 DOM 中删除节点。如果对象存在 destroy 方法，执行它。

（1）语法

```
Ext.destroy(obj1,obj2…,objn);
```

其中，objn（n ≥ 1）是要删除的对象。该方法没有返回值。

（2）源代码

```
destroy: function() {
    var ln = arguments.length,
    i, arg;

    for (i = 0; i < ln; i++) {
        arg = arguments[i];
        if (arg) {
            if (Ext.isArray(arg)) {
                this.destroy.apply(this, arg);
            }
            else if (Ext.isFunction(arg.destroy)) {
                arg.destroy();
            }
            else if (arg.dom) {
                arg.remove();
            }
        }
    }
},
```

从代码可以看到，函数没有固定的参数，使用 arguments 对象获取参数。

通过遍历 arguments 对象，如果对象存在，分 3 种情况删除对象。如果是数组，则递归调用逐个删除数组中的元素。如果是函数，调用函数的 destroy 方法。如果是存在 dom 属性（其实就是 Element 对象），调用对象的 remove 方法。

8. urlAppend

为 url 追加查询字符串。

（1）语法

```
Ext.urlAppend(url,s)
```

其中，url 是要添加查询字符串的地址；s 是要添加的查询字符串；方法最后返回添加了查询字符串后的 url。

（2）源代码

```
urlAppend : function(url, s) {
    return Ext.String.urlAppend(url, s);
}
```

代码直接调用 Ext.String 的 urlAppend 方法生成字符串。

9. addBehaviors

若 document 对象已经准备好，则可为匹配选择符的元素绑定事件。

（1）语法

```
Ext.addBehaviors(o);
```

参数 o 为对象，其格式如下：

```
{
   'S@eventname':fn,
     ...
}
```

其中，S 是选择符[⊖]；eventname 是事件名称，如 click，dbclick；fn 是要绑定的函数；该方法没有返回值。

（2）源代码

```
addBehaviors : function(o){
    if(!Ext.isReady){
        Ext.onReady(function(){
            Ext.addBehaviors(o);
        });
    } else {
        var cache = {},
            parts,
            b,
            s;
        for (b in o) {
            if ((parts = b.split('@'))[1]) {
                s = parts[0];
                if(!cache[s]){
                    cache[s] = Ext.select(s);
                }
                cache[s].on(parts[1], o[b]);
            }
        }
        cache = null;
    }
},
```

⊖ 相关信息可阅读 6.1 节的内容。

代码首先检测 isReady 属性，看 Ext 是否已经初始化，如果没有，将 addBehaviors 方法放到 onReady 中，等待初始化完成再执行。

如果 Ext 已经初始化，则使用 for…in 循环遍历配置对象。如果配置对象属性按照“@”字符拆分的事件名存在，则在临时缓存 cache 中以选择符为名称保存选择出来的元素数组，然后使用 on 方法为元素绑定事件。

10. getScrollBarWidth

获取滚动条宽度。

（1）语法

```
Ext.getScrollBarWidth([force]);
```

其中，force 的作用是强迫重新计算滚动条宽度；该方法会返回滚动条的宽度。

（2）源代码

```
getScrollBarWidth: function(force){
    if(!Ext.isReady){
        return 0;
    }

    if(force === true || scrollWidth === null){
        var cssClass = Ext.isIE9 ? '' : Ext.baseCSSPrefix + 'hide-offsets';
        var div = Ext.getBody().createChild('<div class="' + cssClass
            + '" style="width:100px;height:50px;overflow:hidden;"><div
            style="height:200px;"></div></div>'),
            child = div.child('div', true);
        var w1 = child.offsetWidth;
        div.setStyle('overflow', (Ext.isWebKit || Ext.isGecko) ? 'auto' :
            'scroll');
        var w2 = child.offsetWidth;
        div.remove();
        scrollWidth = w1 - w2 + 2;
    }
    return scrollWidth;
},
```

如果 Ext JS 还没初始化完成，则返回 0。

Ext 的属性 scrollWidth 保存滚动条的宽度，如果其为 null 或者 force 为 true，则重新计算滚动条宽度。

因为 IE 9 通过元素偏移获取 offsetWidth 有 bug，所以不能使用“x-hide-offsets”样式。

计算原理就是在一个宽度为 100 像素，高度为 50 像素的 div 内，放置一个高度为 200 像素的 div。在刚开始时，设置第 1 个 div 的样式属性 overflow 为 hidden，这时候虽然第 2 个 div 的高度超过了第 1 个 div 的范围，但是不会显示滚动条，此时第 2 个 div 的 offsetWidth 属性值就是没有滚动条时的自身的宽度。将 overflow 属性修改为 auto 或 scroll 后，滚动条就会出现，此时第 2 个 div 的 offsetWidthdiv 属性值就是减去滚动条后的宽度。最后将第一次获得宽度值减去第二次获得宽度值就是滚动条的宽度了。

为了保证足够的显示空间，滚动条宽度增加了两个像素。

11. destroyMembers

删除对象的成员。

（1）语法

```
Ext.destroyMembers(o,name1,name2…namen);
```

其中，o 是要删除其成员的对象；namen（n ⩾ 1）是要删除的成员的关键字；该方法没有返回值。

（2）源代码

```
destroyMembers : function(o){
    for (var i = 1, a = arguments, len = a.length; i < len; i++) {
        Ext.destroy(o[a[i]]);
        delete o[a[i]];
    }
},
```

代码通过遍历 arguments 中的（注意是从 1 开始）关键字，使用 destroy 方法删除其值，然后再使用 delete 删除关键字。

12. copyTo

从一个对象复制属性名称到另一个对象。

（1）语法

```
Ext.copyTo(dest,source,names[,usePrototypeKeys]);
```

其中，dest 是要复制的目标对象；source 是要复制的源对象；names 是要复制的关键字，可为数组和字符串（使用逗号或分号分隔）；usePrototypeKeys 是布尔值的可选参数，当设置为 true 时，无论源对象是否存在关键字都进行复制，其默认值为 false，也就是说只有源对象存在关键字时才进行复制；该方法返回复制后的目标对象。

（2）源代码

```
copyTo : function(dest, source, names, usePrototypeKeys){
    if(typeof names == 'string'){
        names = names.split(/[,;\s]/);
    }
    Ext.each(names, function(name){
        if(usePrototypeKeys || source.hasOwnProperty(name)){
            dest[name] = source[name];
        }
    }, this);
    return dest;
},
```

如果 names 为字符串，将其拆分成数组。

使用 each 方法枚举 names 数组中的元素。如果 usePrototypeKeys 为 true，那么不用检测 source 是否存在关键字 name，直接进行复制。如果 usePrototypeKeys 为 false，需要使用 hasOwnProperty 检测 source 是否存在关键字 name，如果存在则进行复制。

13. 其他方法

还有以下几个方法：

- Ext.urlEncode ： 对 url 进行编码。在 Ext JS 4 中不推荐使用，建议用 Ext.Object 对象的 toQueryString 方法代替。
- Ext.urlDecode ： 对 url 进行解码。在 Ext JS 4 中不推荐使用，建议用 Ext.Object 对象的 fromQueryString 方法代替。

4.2　为框架顺利运行提供支持

4.2.1　平台检测工具：Ext.is

平台检测工具主要用于检测当前应用是运行在什么平台上，这个很重要，例如手机浏览器的可视范围和 PC 有很大区别，操作也有很大的不同，因而两个平台运行的应用是有区别的。

平台检测工具主要有以下 13 个属性：Android、Blackberry、Desktop、Linux、Mac、Phone、Standalone、Tablet、Windows、iOS、iPad、iPhone、和 iPod。

Ext.is 对象的源代码在 support.js 文件中，其代码如下：

```
Ext.is = {
    init : function(navigator) {
        var platforms = this.platforms,
            ln = platforms.length,
            i, platform;

        navigator = navigator || window.navigator;

        for (i = 0; i < ln; i++) {
            platform = platforms[i];
                this[platform.identity]  = platform.regex.test(navigator[platform.
property]);
        }

        this.Desktop = this.Mac || this.Windows || (this.Linux && !this.Android);
        this.Tablet = this.iPad;
        this.Phone = !this.Desktop && !this.Tablet;
        this.iOS = this.iPhone || this.iPad || this.iPod;
        this.Standalone = !!window.navigator.standalone;
    },

    platforms: [{
        property: 'platform',
        regex: /iPhone/i,
        identity: 'iPhone'
    },
    {
        property: 'platform',
        regex: /iPod/i,
```

```
            identity: 'iPod'
        },
        {
            property: 'userAgent',
            regex: /iPad/i,
            identity: 'iPad'
        },
        {
            property: 'userAgent',
            regex: /Blackberry/i,
            identity: 'Blackberry'
        },
        {
            property: 'userAgent',
            regex: /Android/i,
            identity: 'Android'
        },
        {
            property: 'platform',
            regex: /Mac/i,
            identity: 'Mac'
        },
        {
            property: 'platform',
            regex: /Win/i,
            identity: 'Windows'
        },
        {
            property: 'platform',
            regex: /Linux/i,
            identity: 'Linux'
        }]
};

Ext.is.init();
```

从 Ext.is 的定义可以看到，Ext.is 就是一个对象，init 方法为其初始化方法。

在 platforms 属性中使用数组形式包含了检测各种平台的正则表达式、检测用的属性名称和标记符号。这样做非常有利于扩展。例如有一种新的平台“ABC”可使用 platform 属性检测，就可在本地化文件中加入以下语句：

```
if(Ext.is){
   Ext.is.platforms.push({
        property: 'platform',
        regex: /ABC/i,
        identity: 'ABC'
   });
   Ext.is.init();
}
```

这样就可以在代码中使用以下代码检测 ABC 平台了：

```
if(Ext.is.ABC){}
```

在 init 方法中，使用 for 循环遍历 platforms 数组中的平台检测方法，然后将检测值作为 Ext.is 对象的成员，该成员的关键字由检测方法中的 identity 属性指定。

检测方法检测 window.navigator 对象中是否存在 property 属性指定的关键字，并匹配检测方法中的正则表达式，如果匹配，值为 true，否则为 false。

只要 Mac、Windows 或 Linux 中的一个值为 true，则 Desktop 为 true。

Tablet 的值与 iPad 的值相同。

只要不是 Desktop 和 Tablet，那么平台就是 Phone。

如果平台是 iPhone、iPad 或 iPod 中的一种，那么平台就是 iOS。

如果 window.navigator 存在属性 standalone，那么就是 standalone 平台。

4.2.2　当前运行环境检测工具：Ext.supports

新的浏览器支持 HTML 5 和 CSS 3，而旧的不支持，在框架中如果要使用 HTML 5 和 CSS 3 的功能，就必须先检测浏览器是否支持，不支持的话就得用其他方法实现，这样既可充分发挥浏览器的性能，也能向下兼容。最大的问题在于，是在每次使用新功能的时候都检测一遍，还是先检测好，并保存到一个变量或对象属性中，然后根据该变量或对象属性进行检测呢？答案自然是后一种方式好，因为这样可以避免多次运行不必要的检测过程，提高运行效率。Ext.supports 对象就是 Ext JS 框架实现该功能的对象。

Ext.supports 对象的定义与 Ext.is 对象类似，是对象结构定义，在 tests 属性中保存了检测当前运行环境的检测方法，通过 init 方法进行初始化。不过与 Ext.is 对象不同的是，它不能在定义后立即指向 init 方法进行初始化，看看 Ext.supports 的 init 方法的源代码就能了解个中缘由了。方法 init 的代码如下：

```
init : function() {
    var doc = document,
        div = doc.createElement('div'),
        tests = this.tests,
        ln = tests.length,
        i, test;

    div.innerHTML = [
        '<div style="height:30px;width:50px;">',
            '<div style="height:20px;width:20px;"></div>',
        '</div>',
        '<div style="width: 200px; height: 200px; position: relative; padding:
             5px;">',
            '<div style="position: absolute; top: 0; left: 0; width: 100%; height:
                100%;"></div>',
    '</div>',
    '<div style="float:left; background-color:transparent;"></div>'
].join('');

doc.body.appendChild(div);

for (i = 0; i < ln; i++) {
    test = tests[i];
```

```
            this[test.identity] = test.fn.call(this, doc, div);
        }

        doc.body.removeChild(div);
    },
```

在代码中，首先使用 createElement 方法创建一个 div 元素，然后使用 appendChild 方法将 div 元素追加到页面中。试想一下，如果 DOM 对象还没准备好，怎么可能在 DOM 对象内插入 div 元素呢？因此 Ext.supports 对象的初始化必须在 DOM 准备好的时候进行，并且要早于 onReady 函数运行。原因是必须在执行 onReady 方法进行界面渲染前做好检测工作，为界面渲染提供支持，不然这种支持就没任何意义了。根据这两个要求，Ext.supports 的 init 方法放在了 EventManager 的 fireDocReady 方法中。其位置正好是 DOM 对象初始化完成，释放加载事件后，开始执行 onReady 函数之前，而且保证了只运行一次。

把 div 元素追加到页面后就可进行检测工作了。通过循环将 tests 数组中保存的检测方法全部运行一次，然后使用检测对象的 identity 成员的值作为 supports 对象成员的关键字，检测结果为其属性值保存下来就行了。

检测完成后必须移除 div 元素，以免影响界面的渲染。

Ext.supports 对象提供了下列支持属性。

- ArraySort：如果 JavaScript 的数组排序没有 bug，该属性值为 true，否则为 false。其检测代码如下：

```
{
  identity: 'ArraySort',
  fn: function() {
      var a = [1,2,3,4,5].sort(function(){ return 0; });
      return a[0] === 1 && a[1] === 2 && a[2] === 3 && a[3] === 4 && a[4] === 5;
  }
},
```

使用 sort 方法做一次排序，看看排序结果是否正确。居然出现这样的 bug，令笔者汗颜。开发跨平台的框架实在不容易，单单收集各种浏览器的 bug 就是一件不容易的事。

- CSS3BorderRadius：如果当前运行环境支持 CSS 3 的 border-radius 属性，该属性为 true，否则为 false。其检测代码如下：

```
identity: 'CSS3BorderRadius',
fn: function(doc, div) {
  var domPrefixes = ['borderRadius', 'BorderRadius', 'MozBorderRadius',
      'WebkitBorderRadius', 'OBorderRadius', 'KhtmlBorderRadius'],
      pass = false,
      i;
  for (i = 0; i < domPrefixes.length; i++) {
      if (document.body.style[domPrefixes[i]] !== undefined) {
          return true;
      }
  }

  return pass;
}
},
```

数组 domPrefixes 保存了从各浏览器的 style 对象获取 border-radius 属性的 6 种属性名称（笔者不得不又一次感慨），只要有一种属性名称返回值不是 undefined，说明浏览器支持 border-radius，返回 true，否则返回 false。

❑ SVG：如果当前运行环境支持 SVG，该属性为 true，否则为 false。其检测代码如下：

```
{
  identity: 'Svg',
  fn: function(doc) {
      return !!doc.createElementNS && !!doc.createElementNS( "http:/" + "/www.
          w3.org/2000/svg", "svg").createSVGRect;
  }
},
```

如果 document 对象存在 createElementNS 方法，创建 svg 对象。如果 svg 对象存在 createSVGRect 方法，说明支持 SVG，返回 true，否则返回 false。

SVG 和 VML 属性对 Ext JS 4 来说相当重要，因为 Ext JS 的图表功能就是基于这两个技术实现的。如果浏览器这两个都不支持，基本就属于古董级浏览器了。一般情况下，IE 支持 VML，非 IE 支持 SVG。

❑ VML：如果当前运行环境支持 VML，该属性为 true，否则为 false。其检测代码如下：

```
{
  identity: 'Vml',
  fn: function(doc) {
      var d = doc.createElement("div");
      d.innerHTML = "<!--[if vml]><br><br><![endif]-->";
      return (d.childNodes.length == 2);
  }
},
```

创建一个 div 元素，并在里面添加与 VML 有关的代码，如果浏览器不支持 VML，会把整句作为一个注释，因而它只有注释一个子节点。如果浏览器支持 VML，中间的两个“
”会成 div 子节点，因而子节点数量会是 2。这样一来如果其子节点的数量为 2，说明支持 VML，返回 true，否则 false。

其他的支持属性：

❑ AudioTag：如果当前运行环境支持 HTML 5 的 audio 标记，该属性为 true，否则为 false。

❑ BoundingClientRect：如果当前运行环境支持元素的 getBoundingClientRect 方法，该属性为 true，否则为 false。

❑ CSS3BoxShadow：如果当前运行环境支持 CSS 3 的 box-shadow 属性，该属性为 true，否则为 false。

❑ CSS3DTransform：如果当前运行环境支持 CSS 3D 动画，该属性为 true，否则为 false。

❑ CSS3LinearGradient：如果当前运行环境支持线性渐变（linear gradients），该属性为 true，否则为 false。

- Canvas：如果当前运行环境支持画布，该属性为 true，否则为 false。
- ClassList：如果当前运行环境支持 HTML 5 的 classList API，该属性为 true，否则为 false。
- ComputedStyle：如果当初运行环境支持 document.defaultView.getComputedStyle()，该属性为 true，否则为 false。
- CreateContextualFragment：如果当前运行环境支持 CreateContextualFragment 方法，该属性为 true，否则为 false。
- DeviceMotion：如果当前运行环境支持设备动作，该属性为 true，否则为 false。
- Direct2DBug：如果当前运行环境通过 offsetWidth、offsetHeight 或 getBounding-ClientRect 等属性获取元素的尺寸，返回的子像素宽度舍入到最接近的像素，说明有 bug，该属性为 true，否则为 false。
- float：如果当前运行环境支持 CSS 的 float 属性，该属性为 true，否则为 false。
- GeoLocation：如果当前运行环境支持地理定位，该属性为 true，否则为 false。
- History：如果当前运行环境支持 HTML 5 的历史（history）记录功能，该属性为 true，否则为 false。
- MouseEnterLeave：如果当前运行环境支持 mouseenter 和 mouseleave 事件，该属性为 true，否则为 false。
- MouseWheel：如果当前运行环境支持鼠标滚轮，该属性为 true，否则为 false。
- Opacity：如果当前运行环境支持透明（opacity）属性，该属性为 true，否则为 false。
- OrientationChange：如果当前运行环境支持改变方向，该属性为 true，否则为 false。
- Placeholder：如果当前运行环境在 input 中支持 HTML 5 的占位符（placeholder）属性，该属性为 true，否则为 false。
- Range：如果当前运行环境支持 Range 对象，该属性为 true，否则为 false。
- RightMargin：如果当前运行环境支持 margin-right 属性，该属性为 true，否则为 false。
- Touch：如果当前运行环境支持触摸（touch）功能，该属性为 true，否则为 false。
- Transitions：如果当前运行环境支持 CSS 3 过渡（Transitions）效果，该属性为 true，否则为 false。
- TransparentColor：如果当前运行环境支持透明色，该属性为 true，否则为 false。

4.3 Ext JS 的静态方法

4.3.1 概述

为了扩展 JavaScript 原有的语言功能，方便数据处理，Ext JS 提供了 Ext.Date、Ext.Number、Ext.String、Ext.Error、Ext.Object、Ext.Function 和 Ext.Array 7 个对象。通过这些对象的静态方法，可以方便地进行数据转换、格式化数据输出和错误处理等工作。

Ext.Date、Ext.Number 和 Ext.String 代码比较简单，会在第 8 章格式化输出数据中介绍。

4.3.2　Ext.Object 中的静态方法

1. toQueryObjects

将对象转换为数组。

（1）语法

```
Ext.Object.toQueryObjects(name,object[,recursive]);
```

其中，name 是字符串，数组中子对象的关键字；object 是要转换的对象、数组、字符串或其他类型的数据；recursive 是可选参数，决定是否执行递归，默认值为 false，不执行递归，否则为 ture，执行递归；此方法返回转换后的数组。

（2）示例

在控制台输入以下命令：

```
console.dir(Ext.Object.toQueryObjects('student',{
    name:"张三",
    total:266,
    score:[
        {name:"数学",score:80},
        {name:"语文",score:90},
        {name:"英语",score:96}
    ]
},true))
```

运行后，可在控制台看到如图 4-1 所示的输出。

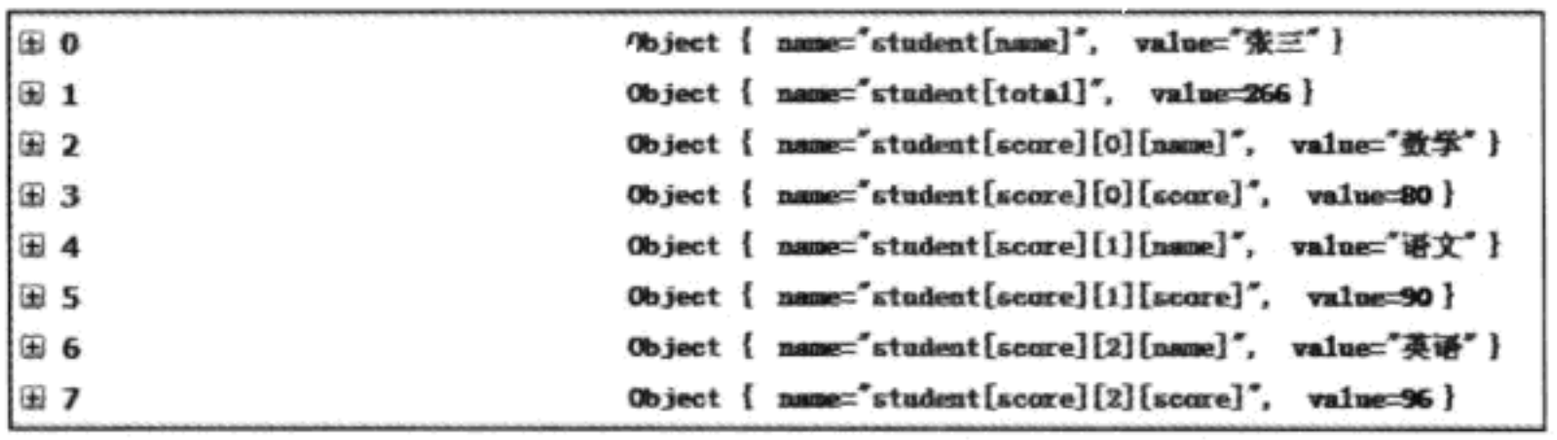


图 4-1　toQueryObjects 示例在控制台的输出结果

如果将递归参数取消，可看到如图 4-2 所示的结果。

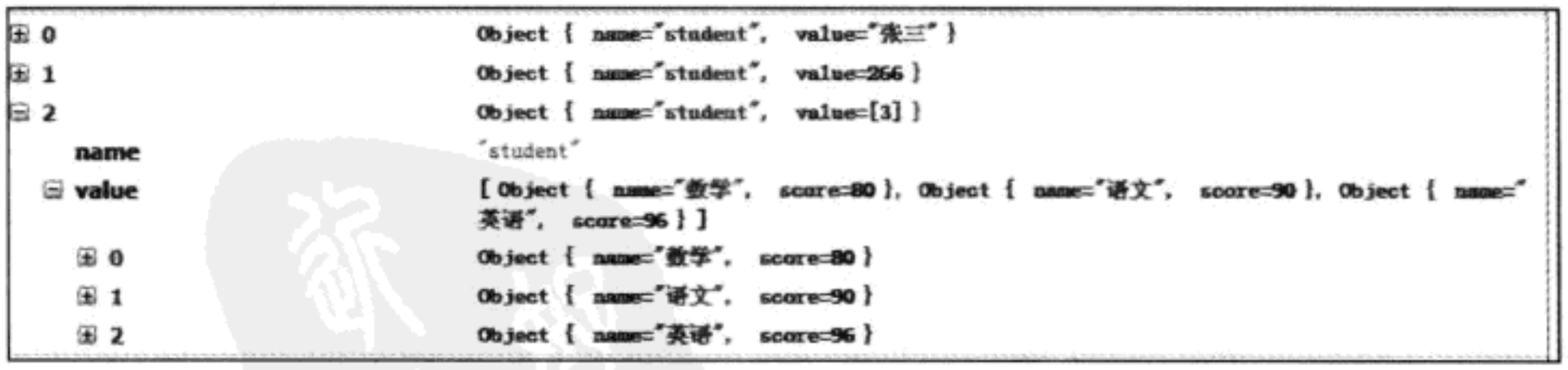


图 4-2　toQueryObjects 示例不使用递归在控制台的输出结果

对比图 4-1 和图 4-2，发现不使用递归的话，会丢失关键字，只转换第一层的数据。

（3）源代码

```
toQueryObjects: function(name, value, recursive) {
    var self = ExtObject.toQueryObjects,
        objects = [],
        i, ln;

    if (Ext.isArray(value)) {
        for (i = 0, ln = value.length; i < ln; i++) {
            if (recursive) {
                objects = objects.concat(self(name + '[' + i + ']', value[i],
                    true));
            }
            else {
                objects.push({
                    name: name,
                    value: value[i]
                });
            }
        }
    }
    else if (Ext.isObject(value)) {
        for (i in value) {
            if (value.hasOwnProperty(i)) {
                if (recursive) {
                    objects = objects.concat(self(name + '[' + i + ']', value[i],
                        true));
                }
                else {
                    objects.push({
                        name: name,
                        value: value[i]
                    });
                }
            }
        }
    }
    else {
        objects.push({
            name: name,
            value: value
        });
    }

    return objects;
},
```

如果要转换的是数组，且使用递归，则将名称修改为“name[i]”（$i \geqslant 0$），数组元素作为转换对象，递归参数为 true，递归调用 toQueryObject 方法。否则直接在结果数组尾部插入一个对象，对象包含 name 和 value 两个成员，成员 name 的值为参数 name，成员 value 的值为数组元素。

如果要转换的是对象，则遍历对象的关键字。如果使用递归方式，则将名称修改为“name[i]”（i 为对象关键字），值作为转换对象。递归参数为 true，递归调用 toQueryObject 方

法。否则直接在结果数组尾部插入一个对象。

如果要转换的不是数组或对象，则直接在结果数组尾部插入一个对象。

2. toQueryString

将对象转换为编码的查询字符串。

（1）语法

```
Ext.Object.toQueryString(object[,recusive]);
```

其中，object 是要转换的对象；recursive 是可选参数，决定是否使用递归，默认为 false，不使用递归，否则为 true，使用递归；方法返回编码后的查询字符串。

（2）示例

在控制台输入以下命令：

```
console.log(Ext.Object.toQueryString({
    id:"jack",
    msg:'Hello',
    option:["music","football","movie"]
},true));
```

运行后，可在控制台看到以下输出：

```
id=jack&msg=Hello&option%5B0%5D=music&option%5B1%5D=football&option%5B2%5D=movie
   //%5B 是 [,%5D 是 ]
```

如果没有递归，显示结果是：

```
id=jack&msg=Hello&option=music&option=football&option=movie
```

（3）源代码

```
toQueryString: function(object, recursive) {
    var paramObjects = [],
        params = [],
        i, j, ln, paramObject, value;

    for (i in object) {
        if (object.hasOwnProperty(i)) {
            paramObjects = paramObjects.concat(ExtObject.toQueryObjects(i,
                object[i], recursive));
        }
    }

    for (j = 0, ln = paramObjects.length; j < ln; j++) {
paramObject = paramObjects[j];
value = paramObject.value;

if (Ext.isEmpty(value)) {
    value = '';
}
else if (Ext.isDate(value)) {
    value = Ext.Date.toString(value);
```

```
            }

            params.push(encodeURIComponent(paramObject.name) + '=' + encodeURICompone
                nt(String(value)));
        }

        return params.join('&');
    },
```

代码首先使用 for…in 循环将 object 内的关键字用 toQueryObjects 方法转换成数组并插入到临时数组。经过 toQueryObjects 方法处理后的数据都是包含 name 和 value 两个成员的对象。

接着处理一下数组里的每个对象的空值和日期值的情况，经过 encodeURIComponent 编码后组装成查询字符串的格式，并插入到结果数组中。

最后将数组用“&”符号连接成字符串并返回。

3. fromQueryString

将查询字符串解码转换为对象。

(1) 语法

```
Ext.Object.fromQueryString(string[,recursive]);
```

其中，string 是要转换的字符串；recursive 是可选参数，决定是否使用递归，默认为 false，不使用递归，否则为 true，使用递归；此方法返回转换后的对象。

(2) 源代码

```
fromQueryString: function(queryString, recursive) {
    var parts = queryString.replace(/^\?/, '').split('&'),
        object = {},
        temp, components, name, value, i, ln,
        part, j, subLn, matchedKeys, matchedName,
        keys, key, nextKey;

    for (i = 0, ln = parts.length; i < ln; i++) {
        part = parts[i];

        if (part.length > 0) {
            components = part.split('=');
            name = decodeURIComponent(components[0]);
            value = (components[1] !== undefined) ? decodeURIComponent(components
                [1]) : '';

            if (!recursive) {
if (object.hasOwnProperty(name)) {
    if (!Ext.isArray(object[name])) {
        object[name] = [object[name]];
    }
    object[name].push(value);
}
else {
    object[name] = value;
}
```

```
}
else {
    matchedKeys = name.match(/(\[):?([^\]]*)\]/g);
    matchedName = name.match(/^([^\[]+)/);

    if (!matchedName) {
      // 省略抛出异常代码
    }

    name = matchedName[0];
    keys = [];
    if (matchedKeys === null) {
        object[name] = value;
        continue;
    }

    for (j = 0, subLn = matchedKeys.length; j < subLn; j++) {
        key = matchedKeys[j];
        key = (key.length === 2) ? '' : key.substring(1, key.length-1);
        keys.push(key);
    }

    keys.unshift(name);
    temp = object;
    for (j = 0, subLn = keys.length; j < subLn; j++) {
        key = keys[j];

        if (j === subLn - 1) {
            if (Ext.isArray(temp) && key === '') {
                temp.push(value);
            }
            else {
                temp[key] = value;
            }
        }
        else {
            if (temp[key] === undefined || typeof temp[key] ===
               'string') {
                nextKey = keys[j+1];

                temp[key] = (Ext.isNumeric(nextKey) || nextKey === '')
                    ? [] : {};
            }

            temp = temp[key];
        }
    }

    return object;
},
```

这个比 toQueryString 复杂多了。

首先把问号及其前面的字符替换掉，然后根据“&”字符将字符串拆分成数组。

接着逐个根据“=”符号拆分出关键字和值，接着解码拆分出来的关键字和值。

如果不使用递归就简单了。如果不是数组，在对象里添加以 name 为关键字、value 为值的成员就行了。如果是数组，也就是在对象中已存在同名的关键字，则转换成数组，再在数组中插入一个值就行了。

如果要使用递归，则要使用正则表达式根据“[]”拆分关键字。

如果关键字不带“[]”符号，直接在对象里添加以 name 为关键字、value 为值的成员就行了。

因为 matchedKeys 拆分出来的数组元素的格式是“[key]”，因而要检查其长度是否为 2，如果为 2，说明是空字符串，否则去掉“[]”，然后将结果存到 keys 数组中。

接着把 name 插入到 keys 数组的开始位置。

如果不是数组最后一个元素，检查以 key 为名称的属性是否存在，如果存在，但其值是字符串，则说明对象或数组还没创建，接着取下一个值，检查该值是数字还是空，如果都不是，则将创建一个空对象，否则创建一个数组。

接着将变量 temp 指向该对象。

如果是数组最后一个元素，则检查 temp 是不是数组，同时检查 key 是否为空，如果是，再将值添加到数组，否则它是对象，则创建一个以 key 为关键字、值为 value 的对象。

4. Each

枚举对象的回调函数。

（1）语法

```
Ext.Object.each(object,fn[,scope]);
```

其中，object 是要枚举的对象；fn 是枚举操作要执行的回调函数，每一次枚举会依次提供关键字、值和对象本身这 3 个参数，在回调函数内返回 false，可中止枚举过程；scope 是可选参数，为函数运行的作用域；该方法没有返回值。

（2）源代码

```
each: function(object, fn, scope) {
    for (var property in object) {
        if (object.hasOwnProperty(property)) {
            if (fn.call(scope || object, property, object[property], object) ===
                false) {
                return;
            }
        }
    },
```

代码遍历对象中的关键字，然后使用 call 方法执行回调函数，如果没有设置回调函数，则使用对象的作用域代替。如果回调函数返回值为 false，则会中止枚举过程。

5. Merge

以递归方式合并任何数量的对象，而且取消对它们的指向。

（1）语法

```
Ext.Object.merge(source[,obj1,obj2,…,objn]);
Ext.merge(source[,obj1,obj2,…,objn]);  // 简写
```

其中，source 是要合并的源对象；objn（n ⩾ 1）是要合并的对象；方法返回合并后的对象。

（2）源代码

```
merge: function(source) {
    var i, ln, object, key, value;

    for (i = 1,ln = arguments.length; i < ln; i++) {
        object = arguments[i];

        for (key in object) {
            value = object[key];

            if (value && value.constructor === Object) {
                if (source[key] && source[key].constructor === Object) {
                    ExtObject.merge(source[key], value);
                }
                else {
                    source[key] = Ext.clone(value);
                }
            }
            else {
                source[key] = value;
            }
        }
    }

    return source;
},
```

代码中循环遍历参数中要合并的对象（注意循环是从 1 开始，第 1 个对象为源对象）。获取对象后，通过 for…in 循环遍历对象中的成员，如果其值存在且是对象，则继续判断源对象中是否存在同关键字的成员，如果存在，则递归调用 merge 方法合并对象。否则，克隆一个对象作为源对象成员的值。

如果不是对象，则直接进行合并。

最后返回 source。

6. 其他的方法

在 Ext.Object 中还有以下几个方法：

- chain：根据指定的对象创建一个新对象作为原型链。
- getValues：获取对象内所有的值，以数组形式返回。
- getKey：在对象内根据指定的值返回其关键字。

- ❑ getKeys：获取对象所有关键字，以数组形式返回。
- ❑ getSize：返回对象的关键字的数量。

以上方法都比较简单，有兴趣可以自己研究。

4.3.3 Ext.Function 中的静态方法

1. flexSetter

封装一个只接收两个参数的函数。封装后，函数的参数会变得灵活。

（1）语法

```
Ext.Function.flexSetter(fn);
```

其中，fn 是要封装的函数；该方法返回封装后的函数。

（2）源代码

```
flexSetter: function(fn) {
    return function(a, b) {
        var k, i;
        if (a === null) {
            return this;
        }
        if (typeof a !== 'string') {
            for (k in a) {
                if (a.hasOwnProperty(k)) {
                    fn.call(this, k, a[k]);
                }
            }
            if (Ext.enumerables) {
                for (i = Ext.enumerables.length; i--;) {
                    k = Ext.enumerables[i];
                    if (a.hasOwnProperty(k)) {
                        fn.call(this, k, a[k]);
                    }
                }
            }
        } else {
            fn.call(this, a, b);
        }

        return this;
    };
},
```

从第 2 个判断语句下的代码可以看到，参数 a 由原来只能处理字符串，变成现在可以灵活地处理对象了。原来函数的两个参数变成了对象中的关键字和值。

如果参数 a 是字符串，则使用原来的方式执行函数。

2. Bind

非常有用的绑定函数的方法，主要作用是保持作用域。

（1）语法

```
Ext.Function.bind(fn[,scope,args,appendArgs]);
Ext.bind(fn[,scope,args,appendArgs]); // 简写
```

其中，fn 是要绑定的函数；scope 是可选参数，是函数运行的作用域，如果没有指定，默认是 window 对象；args 是可选参数，是调用时重写参数的参数数组。appendArgs 是可选参数，值可以为布尔值或数字，如果为 true，则将 agrs 追加到原有参数，而不是重写原有参数。如果是数字，则在原有参数由该数字指定的位置插入 args 参数。该函数返回绑定后的函数。

（2）源代码

```
bind: function(fn, scope, args, appendArgs) {
    if (arguments.length === 2) {
        return function() {
            return fn.apply(scope, arguments);
        };
    }

    var method = fn,
        slice = Array.prototype.slice;

    return function() {
        var callArgs = args || arguments;

        if (appendArgs === true) {
            callArgs = slice.call(arguments, 0);
            callArgs = callArgs.concat(args);
        }
        else if (typeof appendArgs == 'number') {
            callArgs = slice.call(arguments, 0);
            Ext.Array.insert(callArgs, appendArgs, args);
        }

        return method.apply(scope || Ext.global, callArgs);
    };
},
```

如果只有两个参数，表示绑定的函数没有其他参数，返回一个最简单的闭包函数。否则返回带参数的闭包函数。

如果 args 参数存在，则变量 callArgs 指向 args，否则指向 arguments。

如果是追加方式，则在 arguments 后追加 args。

如果是插入方式，则在丢弃了第一个参数的参数数值 callArgs 内，插入 appendArgs 和 args。

最后执行 fn 函数，注意作用域，如果没有设置就使用 window 对象作为其作用域。

3. Pass

封装一个新函数，在调用旧函数时，将预设的参数插入到新函数的参数前面作为旧函数的参数。该方法主要用于创建回调函数。

（1）语法

```
Ext.Function.pass(fn,args[,scope]);
```

其中，fn 是要封装的函数；args 是数组，为预设的参数；scope 是可选参数，为函数的作用域；该方法返回封装好的函数。

（2）源代码

```
pass: function(fn, args, scope) {
    if (!Ext.isArray(args)) {
        args = Ext.Array.clone(args);
    }

    return function() {
        args.push.apply(args, arguments);
        return fn.apply(scope || this, args);
    };
},
```

如果参数 args 不是数组，使用 Ext.Array 对象的 clone 方法将 args 克隆为数组。在返回的函数内部调用旧函数时，args 被合并到了原来参数的前面。

4. Alias

为方法创建一个别名。

（1）语法

```
Ext.Function.alias(object,methodName);
```

其中，object 是要创建别名的对象；methodName 是字符串，为创建别名的方法名称；该方法返回要创建别名的函数。

（2）源代码

```
alias: function(object, methodName) {
    return function() {
        return object[methodName].apply(object, arguments);
    };
},
```

在返回函数内部，使用 apply 方法调用 object 以 methodName 为名称的方法。

5. createInterceptor

创建一个拦截函数。其作用是在调用原始函数时，先使用拦截函数检查数据，如果拦截函数返回 false，不调用原始函数，直接返回参数 returnValue 或 null。否则，执行原始函数。该功能主要用于验证输入，验证通过后才修改原始值。

（1）语法

```
Ext.Function.createInterceptor(origFn,newFn[,scope,returnValue])
```

其中，origFn 是原始函数；newFn 是拦截函数；scope 是可选参数，为函数的作用域；returnValue 是可选参数，是拦截函数返回 false 时的返回值，可以是任何数据类型，如果没

有设置，则返回 null；该方法返回新建的函数。

（2）源代码

```
createInterceptor: function(origFn, newFn, scope, returnValue) {
    var method = origFn;
    if (!Ext.isFunction(newFn)) {
        return origFn;
    }
    else {
        return function() {
            var me = this,
                args = arguments;
            newFn.target = me;
            newFn.method = origFn;
            return (newFn.apply(scope || me || window, args) !== false) ? origFn.
                apply(me || window, args) : returnValue || null;
        };
    }
},
```

如果拦截函数不是函数，直接返回原始函数。

在返回函数的内部，如果拦截函数返回的不是 false，则执行原始函数，否则返回 returnValue 或 null。

6. createDelayed

创建一个延时执行的回调函数。

（1）语法

```
Ext.Function.createDelayed(fn,delay[,scope,args,appends]);
```

其中，fn 是回调函数；delay 是延迟的时间，其单位是微秒；scope 是可选参数，指函数的作用域，如果没有指定，默认是 window 对象。args 是可选参数，为调用时重写参数的参数数组；appendArgs 是可选参数，为布尔值或数字，如果为 true，则将 agrs 追加到原有参数，而不是重写原有参数；如果是数字，则在原有参数由该数字指定的位置插入 args 参数；该方法返回新建的函数。

（2）源代码

```
createDelayed: function(fn, delay, scope, args, appendArgs) {
    if (scope || args) {
        fn = Ext.Function.bind(fn, scope, args, appendArgs);
    }
    return function() {
        var me = this;
        setTimeout(function() {
            fn.apply(me, arguments);
        }, delay);
    };
},
```

如果存在作用域和要追加的参数，先使用 bind 方法创建绑定函数。

在返回函数内，直接使用 setTimeout 方法设置延时执行。

7. Defer

在指定时间后执行函数。

（1）语法

```
Ext.Function.defer(fn,delay[,scope,args,appends]);
Ext.defer(fn,delay[,scope,args,appends]); // 简写
```

其中，fn 是回调函数；delay 是延迟的时间，其单位是微秒；scope 是可选参数，为函数作用域，如果没有指定，默认是 window 对象；args 是可选参数，为调用时重写参数的参数数组；appendArgs 是可选参数，为布尔值或数字，如果为 true，则将 agrs 追加到原有参数，而不是重写原有参数；如果是数字，则在原有参数由该数字指定的位置插入 args 参数；该方法返回新建的函数。

（2）源代码

```
defer: function(fn, millis, obj, args, appendArgs) {
    fn = Ext.Function.bind(fn, obj, args, appendArgs);
    if (millis > 0) {
        return setTimeout(fn, millis);
    }
    fn();
    return 0;
},
```

首先使用 bind 方法创建绑定函数，然后检查时间是否大于 0，如果大于 0，开始延时任务；否则直接执行回调函数。

8. createSequence

创建一个函数，在执行原始函数之后执行指定的函数。

（1）语法

```
Ext.Function.createSequence(origFn,newFn[,scope]);
```

其中，origFn 是先执行的原始函数；newFn 是在原始函数执行完后再执行的函数；scope 是可选参数，为函数的作用域；该方法返回新建的函数。

（2）源代码

```
createSequence: function(originalFn, newFn, scope) {
    if (!newFn) {
        return originalFn;
    }
    else {
        return function() {
            var result = originalFn.apply(this, arguments);
            newFn.apply(scope || this, arguments);
            return result;
        };
    }
},
```

如果 newFn 没有定义，则直接返回原始函数。否则返回一个先执行原始函数、后执行指定函数的函数。

9. createBuffered

创建一个缓冲函数，在指定的时间内，如果函数再次被触发，则重新开始计时。只有在缓存期内没有再被触发时，才会执行指定的函数。该方法主要用于定义键盘事件，例如在缓冲期内，用户再没有输入任何字符，则可以做一些处理，如去数据库搜索数据，在下拉列表内显示。

（1）语法

```
Ext.Function.createBuffered(fn,buffer,scopr,args);
```

其中，fn 是缓冲时间过后要执行的函数；buffer 是缓冲时间，其单位是微秒；scope 是可选参数，为函数的作用域，如果没有指定，默认是 window 对象；args 是可选参数，是调用时重写参数的参数数组；该方法返回新建的函数。

（2）源代码

```
createBuffered: function(fn, buffer, scope, args) {
    var timerId;

    return function() {
        var callArgs = args || Array.prototype.slice(arguments),
            me = scope || this;

        if (timerId) {
            clearTimeout(timerId);
        }

        timerId = setTimeout(function(){
            fn.apply(me, callArgs);
        }, buffer);
    };
},
```

变量 timerId 保存的是 setTimeout 方法的句柄，若在缓冲期内又触发了函数，会使用 clearInterval 方法清理之前 setTimeout 定义的事件，然后再重新开始计时。

10. createThrottled

封装那些会被多次且迅速调用的函数，如鼠标移动事件，只有在距离上次调用时间达到指定间隔后才执行操作。

（1）语法

```
Ext.Function.createThrottled(fn,interval[,scope]);
```

其中，fn 是缓冲时间过后要执行的函数；interval 是指定的时间间隔，其单位是微秒；scope 是可选参数，为函数的作用域，如果没有指定，默认是 window 对象。该方法返回新建的函数。

（2）源代码

```
createThrottled: function(fn, interval, scope) {
    var lastCallTime, elapsed, lastArgs, timer, execute = function() {
        fn.apply(scope || this, lastArgs);
        lastCallTime = new Date().getTime();
    };

    return function() {
        elapsed = new Date().getTime() - lastCallTime;
        lastArgs = arguments;

        clearTimeout(timer);
        if (!lastCallTime || (elapsed >= interval)) {
            execute();
        } else {
            timer = setTimeout(execute, interval - elapsed);
        }
    };
}
```

代码首先构造了一个函数 execute。在其内部会执行函数 fn，而且记录当前的执行时间。

在返回的函数内，会计算当前时间和上次执行时间的时间间隔。

接着清理之前定义的 setTimeout 事件。

如果是第一次执行，或者间隔时间大于指定时间，执行 execute 函数；否则添加一个时间事件，事件触发时间由指定时间 interval 减去时间间隔的差值决定。

11. Clone

为指定的函数创建一个克隆函数。

（1）语法

```
Ext.Function.clone(fn);
```

其中，fn 是要克隆的函数。

（2）源代码

```
clone: function(method) {
    return function() {
        return method.apply(this, arguments);
    };
},
```

代码只是返回一个匿名函数，而匿名函数内会执行指定的函数。

12. interceptBefore

在函数的原始操作之前执行一个指定的操作。

（1）语法

```
Ext.Function.interceptBefore(object, methodName, fn[, scope]);
```

其中，object 为目标对象；methodName 为要重写的方法；fn 为新的操作；scope 是可选

参数，为函数的作用域，如果没有指定，默认是 window 对象。该方法返回新建的函数。

（2）源代码

```
interceptBefore: function(object, methodName, fn, scope) {
    var method = object[methodName] || Ext.emptyFn;

    return (object[methodName] = function() {
        var ret = fn.apply(scope || this, arguments);
        method.apply(this, arguments);

        return ret;
    });
},
```

该方法会重写对象的原有方法，新的方法会在执行原方法之前先执行指定的函数。

13. interceptAfter

在函数的原始操作之后执行一个指定的操作。

（1）语法

```
Ext.Function.interceptAfter(object, methodName, fn[, scope]);
```

其中，object 为目标对象；methodName 为要重写的方法；fn 为新的操作；scope 是可选参数，为函数的作用域，如果没有指定，默认是 window 对象。该方法返回新建的函数。

（2）源代码

```
interceptAfter: function(object, methodName, fn, scope) {
    var method = object[methodName] || Ext.emptyFn;

    return (object[methodName] = function() {
            method.apply(this, arguments);
            return fn.apply(scope || this, arguments);
    });
},
```

该方法会重写对象的原有方法，新的方法会在执行原方法之后执行指定的函数。

4.3.4　Ext.Array 中的静态方法

1. each

枚举数组。

（1）语法

```
Ext.Array.each(array,fn[,scope,reverse]);
Ext.each(array,fn[,scope,reverse]); // 简写
```

其中，array 是要枚举的数组。fn 是枚举每个项时要执行的回调函数。函数依次序可接收 item、index 和 items 三个参数。其中，item 是枚举的数组元素，index 是数组的当前索引，items 是数组本身。scope 是可选参数，为函数的作用域。reverse 是可选参数，为布尔值，如果为 true，数组将从最后一个元素开始枚举，默认为 false，数组从第一个元素开始枚举。如

果回调函数返回 false，则返回当前索引，否则返回 true。

（2）源代码

```
each: function(array, fn, scope, reverse) {
    array = ExtArray.from(array);

    var i,
        ln = array.length;

    if (reverse !== true) {
        for (i = 0; i < ln; i++) {
            if (fn.call(scope || array[i], array[i], i, array) === false) {
                return i;
            }
        }
    }
    else {
        for (i = ln - 1; i > -1; i--) {
            if (fn.call(scope || array[i], array[i], i, array) === false) {
                return i;
            }
        }
    }

    return true;
},
```

如果 reverse 的值为 fales，循环从 0 开始，否则，从“ln-1”开始。

如果 fn 返回 false，则返回当前索引 i。

如果循环执行完毕，没有中断，则返回 true。

2. forEach

遍历一个数组，并使用指定函数处理数组的每一个元素。

（1）语法

```
Ext.Array.forEach(array,fn[,scope]);
```

其中，array 是要遍历的数组；fn 是处理函数，函数依次序可接收 item、index 和 items 三个参数，其中，item 是枚举的数组元素，index 是数组的当前索引，items 是数组本身；scope 是可选参数，为函数作用域；该方法没有返回值。

（2）源代码

```
forEach: function(array, fn, scope) {
    if (supportsForEach) {
        return array.forEach(fn, scope);
    }

    var i = 0,
        ln = array.length;

    for (; i < ln; i++) {
```

```
            fn.call(scope, array[i], i, array);
        }
    },
```

如果 JavaScript 的 Array 对象有 forEach 方法，使用 JavaScript 的 Array 对象的 forEach 方法处理。

如果没有 forEach 方法，则遍历数组，然后使用 fn 处理数组元素。

3. toArray

将可迭代的数据转换为数组。

（1）语法

```
Ext.Array.toArray(iterable[,start,end]);
Ext.toArray(iterable[,start,end]); // 简写
```

其中，iterable 的数据类型为任何可迭代的数据，表示将要转换为数组的数据；start 是可选参数，为数字值，表示转换的开始位置；end 是可选参数，为数字值，表示转换的结束位置；该方法返回转换后的数组。

（2）源代码

```
toArray: function(iterable, start, end){
    if (!iterable || !iterable.length) {
        return [];
    }

    if (typeof iterable === 'string') {
        iterable = iterable.split('');
    }

    if (supportsSliceOnNodeList) {
        return slice.call(iterable, start || 0, end || iterable.length);
    }

    var array = [],
        i;

    start = start || 0;
    end = end ? ((end < 0) ? iterable.length + end : end) : iterable.length;

    for (i = start; i < end; i++) {
        array.push(iterable[i]);
    }

    return array;
},
```

如果数据不存在，或不可迭代（不存在 length 属性），则返回空数组。

如果数据类型是字符串，则将它拆分成字母数组。

因为 IE 6 到 IE 8 版本中 Array 对象原型的 slice 方法不支持节点列表，因而要通过属性 supportsSliceOnNodeList 检查浏览器是否完全支持 slice 方法，如果支持，使用 slice 方法返回

数组。

如果不完全支持，就只能自己用循环处理了。

4. pluck

根据指定的属性，获取数组内每个对象指定关键字的值。

（1）语法

```
Ext.Array.pluck(array,name);
Ext.pluck(array,name); // 简写
```

其中，array 是获取数据的数组；name 是字符串，为指定的关键字；该方法返回由指定关键字的值所组成的数组。

（2）示例

在浏览器中打开 Ext JS 4.0 的 API 文档的首页，然后在控制台输入以下命令就可获取主页面中所有的类名了：

```
console.dir(Ext.pluck(Ext.query(".links a"),"innerHTML"));
```

运行后，可在控制台看到以下输出：

```
0                "Ext"
1                "Ext.Base"
...
308              "Ext.util.Observable"
309              "Ext.util.TaskRunner"
```

（3）源代码

```
pluck: function(array, propertyName) {
    var ret = [],
        i, ln, item;

    for (i = 0, ln = array.length; i < ln; i++) {
        item = array[i];

        ret.push(item[propertyName]);
    }

    return ret;
},
```

代码不难，遍历一次数组，然后将属性值保存到 ret 数组就行了。

5. from

将一个值转换为数组。

(1) 语法

```
Ext.Aarray.from(value[,newReference]);
```

其中，value 是要转换为数组的值；newReference 是可选参数，决定使用数组的元素是否使用新的指向，默认值为 false，不使用新指向，否则，克隆数组，使用新指向。

如果 value 不能转换为数组，返回空数组。如果 value 本身就是数组，根据 newReference 参数决定是直接返回数组还是返回新指向的数组。如果是可迭代的值，则返回 toArray 方法生成的数组。如果以上都不是，返回由 value 构成的只有一个元素的数组。

（2）源代码

```
from: function(value, newReference) {
    if (value === undefined || value === null) {
        return [];
    }

    if (Ext.isArray(value)) {
        return (newReference) ? slice.call(value) : value;
    }

    if (value && value.length !== undefined && typeof value !== 'string') {
        return Ext.toArray(value);
    }

    return [value];
},
```

如果返回值为 undefined 或 null，返回空数组。如果值是数组，且要使用新指向，使用 slice 方法返回数组，否则直接返回值。

如果值是可迭代的，返回由 toArray 方法生成的数组。

如果以上都不是，返回以值 Value 作为唯一元素的数组。

6. sort

对数组中的元素进行排序。如果要对包含中文字符串的数组进行排序，一定要定义自定义排序函数，不然会出现错误。

（1）语法

```
Ext.Array.sort(array[,fn]);
```

其中，array 是要对元素进行排序的数组；fn 是可选参数，为自定义的排序函数，如果要对中文字符串排序，一定要定义该函数；该方法返回排序后的数组。

（2）示例

首先在命令行中输入以下代码：

```
var array=["张三","李四","王五","刘六"];
console.log(Ext.Array.sort(array));
```

运行后，可在控制台看到以下输出：

```
["刘六", "张三", "李四", "王五"]
```

结果是错误的，并不是预期的排序结果。将命令修改为以下命令：

```
var ar=["张三","李四","王五","刘六"];
console.log(Ext.Array.sort(ar,function(a,b){
    if(typeof a === 'string'){
```

```
            return a.localeCompare(b);
        }else{
            return a<b;
        }
}));
```

运行后，可在控制台看到以下输出：

```
["李四", "刘六", "王五", "张三"]
```

这才是正确的排序结果。主要改变是在排序函数中对字符串使用了 localeCompare 比较大小，而不是直接比较大小。

（3）源代码

```
sort: function(array, sortFn) {
    if (supportsSort) {
        if (sortFn) {
            return array.sort(sortFn);
        } else {
            return array.sort();
        }
    }

    var length = array.length,
        i = 0,
        comparison,
        j, min, tmp;

    for (; i < length; i++) {
        min = i;
        for (j = i + 1; j < length; j++) {
            if (sortFn) {
                comparison = sortFn(array[j], array[min]);
                if (comparison < 0) {
                    min = j;
                }
            } else if (array[j] < array[min]) {
                min = j;
            }
        }
        if (min !== i) {
            tmp = array[i];
            array[i] = array[min];
            array[min] = tmp;
        }

        return array;
    },
```

代码先检查 JavaScript 的 Array 对象是否支持 sort 方法，如果支持，则使用 JavaScript 的 Array 对象的 sort 方法。虽然某些浏览器的 sort 方法会使用正确的中文排序，但是为了安全起见，还是建议使用自定义的排序方法，不然在某些浏览器上还是会出错。

排序使用了冒泡排序法。如果没有自定义排序函数，会直接使用“array[j] < array[min]”语句进行比较，这是造成中文字符排序出错的主要原因。

7. 其他方法

在 Ext.Array 对象中还有以下方法：

- indexOf：查询元素在数组中的位置，如果找不到元素，返回 –1。
- contains：检查数组是否包含指定元素。
- map：与 forEach 方法有点类似，不同的是 map 方法会把处理结果以数组形式返回。
- every：与 each 方法有点类似，使用回调函数处理数组的每一个元素，如果回调函数返回 false，中断执行并返回 false。否则返回 true。
- some：与 every 方法正好相反，回调函数返回 true 时，中断执行并返回 true，否则返回 false。
- clean：清理数组中的空值。
- unique：清理数组中的重复项。
- filter：使用过滤函数验证函数中的每一个元素，如果过滤函数返回 true，则该元素保留，否则移除元素。最后返回过滤后的数组。
- remove：从数组中删除一个元素。
- include：如果数组不包含指定元素，将该元素加入数组。
- clone：克隆数组，会使用新的指向。
- merge：合并数组，并去除重复项。
- intersect：将存在于所有数组的项组合成一个新数组。
- difference：将数组 B 中不存在于数组 A 的项依次插入到数组 A 中。
- flatten：使用递归方式将多维数组转换为一维数组。
- min：返回数组元素中的最小值。与 sort 方法一样，如果包含中文字符号，一定要使用自定义函数来获取值。
- max：返回数组元素中的最大值。与 sort 方法一样，如果包含中文字符号，一定要使用自定义函数来获取值。
- mean：计算数组中所有项的平均值。
- sum：计算数组中所有项的和。

以上方法都比较简单，若有兴趣可以自己研究。

4.3.5　Ext.Error 中的静态方法

Ext.Error 类的作用是为 Ext 应用程序提供一种有用的调试方法，最主要的功能就是自动显示发生错误的源类和方法，这样，通过错误信息就能知道错误发生在哪个类中的哪个方法。当然，如果应用不大、类不多，可以使用习惯的错误处理方式来进行处理，但如果是类似 Ext JS 这样的框架或应用，有 200 多个类的情况，一般的错误处理方式就很难满足要求了。

Ext.Error 类主要有两个属性和两个静态方法。其主要属性如下：

- ignore：布尔值，默认值为 false，显示错误报告。如果设置为 true，则不显示错误报告。这样的好处是，在类中可包含调试代码，通过该开关，可在调试时显示错误报告，而在运营时关闭错误报告，而无须删除类内的调试代码。
- notify ：作用与 ignore 属性相同，区别是它不会影响异常的抛出。而且它在正式版中不能使用。

Ext.Error 主要的静态方法是 raise 方法，用于抛出错误信息。方法 handle 的作用是实现自定义错误处理，比较少用到。下面看看 raise 是如何工作的，其源代码如下：

```
raise: function(err){
    err = err || {};
    if (Ext.isString(err)) {
        err = { msg: err };
    }

    var method = this.raise.caller;

    if (method) {
        if (method.$name) {
            err.sourceMethod = method.$name;
        }
        if (method.$owner) {
            err.sourceClass = method.$owner.$className;
        }
    }

    if (Ext.Error.handle(err) !== true) {
        var msg = Ext.Error.prototype.toString.call(err);

        Ext.log({
            msg: msg,
            level: 'error',
            dump: err,
            stack: true
        });

        throw new Ext.Error(err);
    }
},
```

代码先判断参数 err 是否存在，如果不存在，设置其为空对象，如果是字符串，则将其转换为对象成员 msg 的值。

接着从 caller 属性获取调用 raise 方法的调用方法。如果方法存在，从其“$name”属性中获取方法名称，并将其作为错误对象 sourceMethod 成员的值。如果方法存在“$owner”属性（方法拥有者），则将拥有者的类名作为错误对象 sourceClass 成员的值。

接着调用 handle 方法处理错误信息，默认它会返回 ignore 属性的值。如果 handle 方法为 false，则调用 Ext.Error 类原型的 toString 方法根据 err 对象组合错误信息并返回。在 Ext.log 中登记错误信息后，调用 throw 语句抛出错误信息。

从代码中可以看到，从 Ext JS 4.1 beta 1 开始，内置了日志功能用于监测错误。

通过源代码分析，可以看到要使用 raise 方法很简单，其格式如下：

```
Ext.Error.raise(err)
```

其中 err 可以是字符串，也可以是带 msg 成员的对象。

打开模板页，在控制台中输入以下代码：

```
Ext.Error.raise("我的错误信息");
```

会在控制台看到如图 4-3 所示的输出。

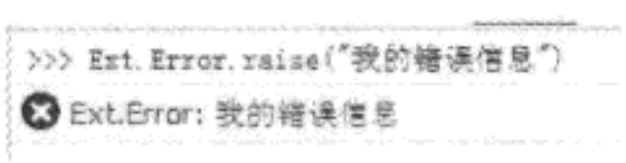


图 4-3　rasie 的输出效果

Ext.Error 是很好的调试工具，很实用，不过目前功能还比较单一，而且有 bug，需要继续完善。

4.4　深入了解类的创建及管理

4.4.1　开始创建类

要了解 Ext JS，必须先了解其类系统。本节所讲的类的创建过程是指使用 Ext.define 定义的类的创建过程，而不是使用旧的 extend 方法创建类的过程。

在 Ext JS 4 中使用 Ext.define 创建类，其代码在 ClassManager.js 中，代码如下：

```
define: function (className, data, createdFn) {
    if (data.override) {
        return Manager.createOverride.apply(Manager, arguments);
    }

    return Manager.create.apply(Manager, arguments);
},
```

如果参数 data 存在成员 override，则调用 ClassManager 对象的 createOverride 方法重写类，否则调用 create 方法创建类。因为 createOverride 方法只是重写类，因而这里只讨论 create 方法，其代码如下：

```
create: function(className, data, createdFn) {
// 省略抛出异常代码
data.$className = className;

return new Class(data, function() {
    var postprocessorStack = data.postprocessors || Manager.default-
        Postprocessors,
        registeredPostprocessors = Manager.postprocessors,
        index = 0,
        postprocessors = [],
```

```
        postprocessor, process, i, ln, j, subLn, postprocessorProperties,
            postprocessorProperty,
        alternateNames;

    delete data.postprocessors;

    for (i = 0,ln = postprocessorStack.length; i < ln; i++) {
        postprocessor = postprocessorStack[i];

        if (typeof postprocessor == 'string') {
            postprocessor = registeredPostprocessors[postprocessor];
            postprocessorProperties = postprocessor.properties;

            if (postprocessorProperties === true) {
                postprocessors.push(postprocessor.fn);
            }
            else if (postprocessorProperties) {
                for (j = 0,subLn = postprocessorProperties.length; j < subLn; j++) {
                    postprocessorProperty = postprocessorProperties[j];

                    if (data.hasOwnProperty(postprocessorProperty)) {
                        postprocessors.push(postprocessor.fn);
                        break;
                    }
                }
            }
        }
        else {
            postprocessors.push(postprocessor);
        }
    }

    process = function(clsName, cls, clsData) {
        postprocessor = postprocessors[index++];

        if (!postprocessor) {
            Manager.set(className, cls);

            if (createdFn) {
                createdFn.call(cls, cls);
            }

            Manager.triggerCreated(className);
            return;
        }

        if (postprocessor.call(this, clsName, cls, clsData, process) !==
            false) {
            process.apply(this, arguments);
        }
    };

    process.call(Manager, className, this, data);

    Manager.applyOverrides(className);
```

```
            alternateNames = Manager.maps.nameToAlternates[className];

            for (i = 0, ln = alternateNames && alternateNames.length || 0; i < ln;
                ++i) {
                Manager.applyOverrides(alternateNames[i]);
            }
        });
    },
```

代码先在配置对象中添加一个关键字为“$className”的成员，其值为类名。接着使用 new 关键字创建一个 Class 对象的实例。

4.4.2　创建类的类：Ext.Class

创建 Class 对象的实例后，将执行如下的 Class 对象的初始化代码：

```
Ext.Class = ExtClass = function(Class, data, onCreated) {
    if (typeof Class != 'function') {
        onCreated = data;
        data = Class;
        Class = null;
    }

    if (!data) {
        data = {};
    }

    Class = ExtClass.create(Class, data);

    ExtClass.process(Class, data, onCreated);

    return Class;
};
```

为了兼容旧的方式，Class 对象的入口参数还是 3 个，以下是这 3 个参数的说明：

- Class：要创建的类。
- data：配置对象。
- onCreated：类创建完成的回调函数。

因使用 Ext.define 创建的类只传递两个参数，因而需要调整参数。

如果 Class 的参数类型是函数，说明是使用旧方法创建的新类，不需要调整。该过程与 extend 方法差不多。

如果 Class 的参数类型不是函数，则需要调整参数，onCreated 将指向正确的回调函数，也就是第 2 个参数 data。参数 data 则指向正确的配置对象 Class。参数 Class 则设置为 null。

接着调用 Ext.Class 对象的 create 方法，其代码如下：

```
create: function(Class, data) {
    var name, i;

    if (!Class) {
```

```
        Class = makeCtor(
            data.$className
        );
    }

    for (i = 0; i < baseStaticMemberLength; i++) {
        name = baseStaticMembers[i];
        Class[name] = Base[name];
    }

    return Class;
},
```

如果参数 Class 不存在，则调用 makeCtor 方法为其创建一个构造函数。

接着将 Ext.Base 对象中的基本静态成员复制到新类中，简单来说，就是将 Ext.Base 对象的属性和方法复制到新类。

返回后，调用 process 方法继续类的创建过程，其代码如下：

```
process: function(Class, data, onCreated) {
    var preprocessorStack = data.preprocessors || ExtClass.defaultPreprocessors,
        registeredPreprocessors = this.preprocessors,
        hooks = {
            onBeforeCreated: this.onBeforeCreated
        },
        index = 0,
        preprocessors = [],
        preprocessor, preprocessorsProperties,
        i, ln, j, subLn, preprocessorProperty, process;

    delete data.preprocessors;

    for (i = 0,ln = preprocessorStack.length; i < ln; i++) {
        preprocessor = preprocessorStack[i];

        if (typeof preprocessor == 'string') {
            preprocessor = registeredPreprocessors[preprocessor];
            preprocessorsProperties = preprocessor.properties;

            if (preprocessorsProperties === true) {
                preprocessors.push(preprocessor.fn);
            }
            else if (preprocessorsProperties) {
                for (j = 0,subLn = preprocessorsProperties.length; j < subLn; j++)
{
                preprocessorProperty = preprocessorsProperties[j];

                if (data.hasOwnProperty(preprocessorProperty)) {
                    preprocessors.push(preprocessor.fn);
                    break;
                }
            }
        }
    }
}
else {
```

```
            preprocessors.push(preprocessor);
        }
    }

    hooks.onCreated = onCreated ? onCreated : Ext.emptyFn;

    process = function(Class, data, hooks) {
        preprocessor = preprocessors[index++];

        if (!preprocessor) {
            hooks.onBeforeCreated.apply(this, arguments);
            return;
        }

        if (preprocessor.call(this, Class, data, hooks, process) !== false) {
            process.apply(this, arguments);
        }
    };

    process.call(this, Class, data, hooks);
},
```

在这里有两个重要的变量，要先搞清楚其内容才知道接下来会进行什么处理。第 1 个是 preprocessorStack，如果配置对象中没有定义 preprocessors 属性，则会使用 Ext.Class 对象的 defaultPreprocessors 属性的值，那么它的值是什么呢？想知道也简单，打开“Hello World”页面，然后在控制台输入以下命令：

```
console.log(Ext.Class.defaultPreprocessors)
```

在控制台可看到以下输出：

```
["className", "loader", "extend", "statics", "inheritableStatics", "config",
    "mixins", "xtype", "alias"]
```

数组里的字符串是不是有点眼熟？先别管，现在研究一下变量 registeredPreprocessors，其默认值为 Ext.Class 对象的 preprocessors 属性值，同样在控制台输入：

```
console.log(Ext.Class.preprocessors);
```

控制台的输出是：

```
Object { extend={...}, statics={...}, inheritableStatics={...}, 更多 ...}
```

这是一个对象，在控制台单击对象进去将看到如图 4-4 所示的结果。

从图 4-4 中可以看到，每个对象都有 properties、name 和 fn 三个属性。现在继续往下看代码。

在删除了配置对象的 preprocessors 属性后，进入一个循环。循环会取数组 preprocessorStack 中的值，然后以该值作为关键字在 registeredPreprocessors 中找到对象。如果对象的 properties 属性为 true，则直接将 fn 指向的函数放到 preprocessors 数组中。如果不为 true，则检查配置对象中是否有定义，有定义的才加入到数组中。

之后将 hooks 对象的 onCreated 属性指向回调函数 onCreated（如果存在）。

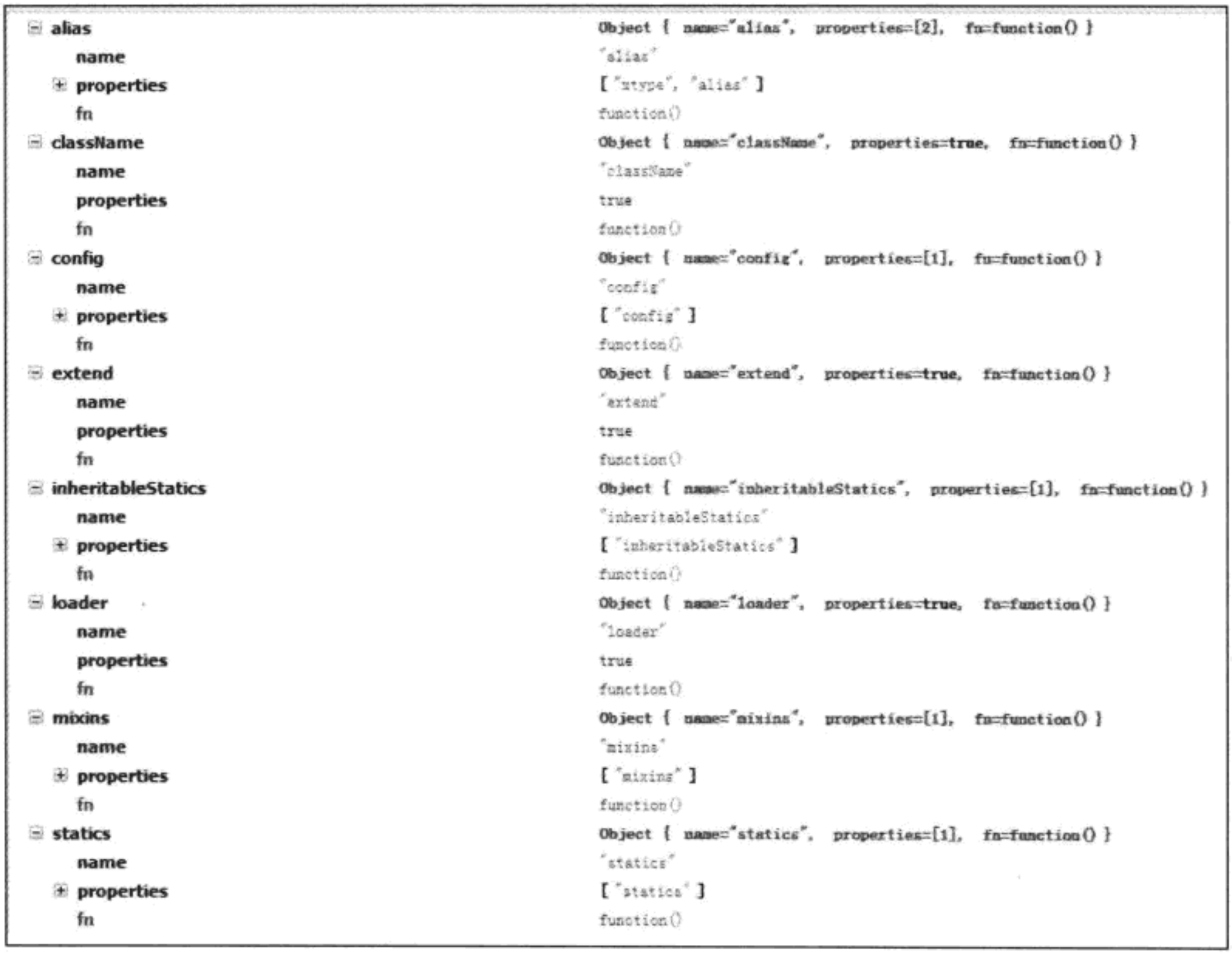


图 4-4　preprocessors 对象的内容

接着是将变量 process 指向一个匿名函数，并开始调用该函数。在函数内，会从 preprocessors 数组中取出函数，并将变量 preprocessor 指向该函数，也就是从 registeredPreprocessors 对象中获取处理函数。

如果 preprocessor 不存在，表示数组内的函数已经执行完毕，调用 onBeforeCreated 方法并返回，这个我们之后再研究。

接着执行 preprocessor 指向的函数。如果返回值不为 false，则递归调用 process 函数。也就是说如果没有中断，process 函数会一直执行到 preprocessors 数组再没有数据可取时才停止，即在检查到 preprocessor 不存在时停止。在停止前会执行配置对象的 onBeforeClassCreated 函数。

现在不急着往下执行，先了解一下 className、loader、extend、statics、inheritableStatics、config、mixins 和 alias 这 8 个函数到底做了什么操作。这就需要从 Class 对象的 preprocessors 属性入手，因为是它提供了 8 个函数的操作。

在 Class.js 中，可找到 preprocessors 的定义：

```
preprocessors: {}
```

定义时为 1 个空对象，那是怎么为它添加属性的呢？在 Class.js 文件中有一个名称为 registerPreprocessor 的方法，其代码如下：

```
registerPreprocessor: function(name, fn, properties, position, relativeTo) {
    if (!position) {
        position = 'last';
    }

    if (!properties) {
        properties = [name];
    }

    this.preprocessors[name] = {
        name: name,
        properties: properties || false,
        fn: fn
    };

    this.setDefaultPreprocessorPosition(name, position, relativeTo);

    return this;
},
```

如果参数 position 不存在，则默认为 last。

如果参数 properties 也不存在，则指向一个由参数 name 构建的数组。

接着创建一个有 name、properties 和 fn 这三个成员的对象，再调用 setDefaultPreprocessorPosition 方法根据 position 和 relativeTo 的值将 name 加入到 defaultPreprocessors 数组中。

现在按顺序先找使用 registerPreprocessor 方法注册的 className 对象，其代码如下（在 ClassManager.js 文件中）：

```
Class.registerPreprocessor('className', function(cls, data) {
    if (data.$className) {
        cls.$className = data.$className;
        cls.displayName = cls.$className;
    }
}, true, 'first');
```

代码很简单，如果配置对象存在 $className 属性，则将新类的 $className 属性赋值为配置对象 $className 属性的值，也就是将新类的 $className 属性设置为类名。新类的 dispalyName 的值也会是类名。

注意最后的参数是 true，也就是 properties 的值为 true，为必须执行的过程，且其指向位置是 first，说明它是最先处理的过程。

现在已经指定 className 对象的作用是处理类名。之后是 loader，这暂时跳过，将在 4.4.4 节进行讲解。接着是 extend，其代码如下：

```
ExtClass.registerPreprocessor('extend', function(Class, data) {
    var Base = Ext.Base,
        basePrototype = Base.prototype,
        extend = data.extend,
```

```
        Parent, parentPrototype, i;

    delete data.extend;

    if (extend && extend !== Object) {
        Parent = extend;
    }
    else {
        Parent = Base;
    }

    parentPrototype = Parent.prototype;

    if (!Parent.$isClass) {
        for (i in baseProtoType) {
            if (!parentPrototype[i]) {
                parentPrototype[i] = baseProtoType[i];
            }
        }
    }

    Class.extend(Parent);

    Class.triggerExtended.apply(Class, arguments);

    if (data.onClassExtended) {
        Class.onExtended(data.onClassExtended);
        delete data.onClassExtended;
    }

}, true);
```

将代码与 extend 方法的代码比较一下，会发现除了变量名不同外，其他基本上是一样的，也就是说，在这里也是使用寄生组合式继承来实现类的继承。所以呼吁大家先熟悉寄生组合式继承再来阅读以下内容。

代码首先将变量 base 指向 Base 对象；baseProtoType 指向 Base 对象的原型；将 extend 变量指向配置对象的 extend 属性值，也就是指向要继承的类。

注意 经过 loader 后，配置对象的 extend 属性值已经指向父类，而不是字符串了。

删除配置对象的 extend 成员后，通过判断 extend 是否为对象，决定新类的父类是由 extend 指定的类还是使用 Base 对象作为父类。从这里就可以知道，Base 类可以说是许多 Ext JS 类的基类。

接着就开始进行原型继承了，变量 parentPrototype 会指向父类的原型，如果父类不存在“$isClass”属性，则需要从 Base 对象继承，将 Base 对象原型中的成员复制到父类的原型中。

之后调用 Class 的 extend 方法，这个方法是在调用 create 方法时从 Base 对象中以静态成员的形式复制过来的，其代码如下：

```
extend: function(parent) {
```

```
var parentPrototype = parent.prototype,
    basePrototype, prototype, i, ln, name, statics;

prototype = this.prototype = Ext.Object.chain(parentPrototype);
prototype.self = this;

this.superclass = prototype.superclass = parentPrototype;

if (!parent.$isClass) {
    basePrototype = Ext.Base.prototype;

    for (i in basePrototype) {
        if (i in prototype) {
            prototype[i] = basePrototype[i];
        }
    }
}

statics = parentPrototype.$inheritableStatics;

if (statics) {
    for (i = 0,ln = statics.length; i < ln; i++) {
        name = statics[i];

        if (!this.hasOwnProperty(name)) {
            this[name] = parent[name];
        }
    }

    if (parent.$onExtended) {
        this.$onExtended = parent.$onExtended.slice();
    }

    prototype.config = new prototype.$configClass;
    prototype.$configList = prototype.$configList.slice();
    prototype.$hasConfig = Ext.Object.chain(prototype.$hasConfig);
},
```

对比一下旧的 extend 方法可以发现，基本原理是一样的，不过多了几样东西。如果是从 Base 对象继承的类，会把 Base 对象原型的成员复制到新类的原型中；把父类原型“$inheritableStatics”中的静态方法复制到新类中；如果父类存在“$onExtended”属性，会复制到新类；在原型中加入成员 config、“$configList”和“$hasConfig”。

在代码中使用了两次 Ext.Object 的 chain 方法创建原型链，其源代码如下：

```
chain: function (object) {
    TemplateClass.prototype = object;
    var result = new TemplateClass();
    TemplateClass.prototype = null;
    return result;
},
```

这是寄生组合式继承创建原型链的过程。

回到 extend 的处理过程，接着调用 triggerExtended 方法（从 Base 对象复制过来的静态方法），其代码如下：

```
triggerExtended: function() {
    var callbacks = this.$onExtended,
        ln = callbacks.length,
        i, callback;

    if (ln > 0) {
        for (i = 0; i < ln; i++) {
            callback = callbacks[i];
            callback.fn.apply(callback.scope || this, arguments);
        }
    }
},
```

从代码可以看到，主要是调用“$onExtended”中保存的回调函数。这个“$onExtended”中的回调函数是怎么来的呢？回到 extend 处理过程，继续分析下面的代码就清楚了。如果配置对象定义了 onClassExtended 属性，那么就会调用从 Base 对象复制过来的 onExtended 方法，将 onClassExtended 属性指向的回调函数保存到“$onExtended”数组中。

至此，extend 处理过程就执行完了，接着进入 statics 的处理，其代码如下：

```
ExtClass.registerPreprocessor('statics', function(Class, data) {
    Class.addStatics(data.statics);
    delete data.statics;
});
```

直接调用 addStatics 方法，其代码如下：

```
addStatics: function(members) {
    var member, name;
    var className = Ext.getClassName(this);

    for (name in members) {
        if (members.hasOwnProperty(name)) {
            member = members[name];
            if (typeof member == 'function') {
                member.displayName = className + '.' + name;
            }
            this[name] = member;
        }
    }

    return this;
},
```

该处理的主要作用是将配置对象中定义的静态属性和方法复制到类中。代码首先从配置对象中取得 statics 对象，然后遍历对象中的属性和方法，并将其复制到类中。注意，是直接复制到类中，而不是类的原型中。最后删除配置对象的 statics 属性。如果是复制到原型中，那么必须使用 new 语句创建实例后才能调用这些方法，也就不是静态方法了。

接着进行 inheritableStatics 处理，其代码如下：

```
ExtClass.registerPreprocessor('inheritableStatics', function(Class, data) {
    Class.addInheritableStatics(data.inheritableStatics);
    delete data.inheritableStatics;
});
```

直接调用 addInheritableStatics 方法，其代码如下：

```
addInheritableStatics: function(members) {
    var inheritableStatics,
        hasInheritableStatics,
        prototype = this.prototype,
        name, member;

inheritableStatics = prototype.$inheritableStatics;
hasInheritableStatics = prototype.$hasInheritableStatics;

if (!inheritableStatics) {
    inheritableStatics = prototype.$inheritableStatics = [];
    hasInheritableStatics = prototype.$hasInheritableStatics = {};
}

var className = Ext.getClassName(this);

for (name in members) {
    if (members.hasOwnProperty(name)) {
        member = members[name];
        if (typeof member == 'function') {
            member.displayName = className + '.' + name;
        }
        this[name] = member;

        if (!hasInheritableStatics[name]) {
            hasInheritableStatics[name] = true;
            inheritableStatics.push(name);
        }
    }
}

    return this;
},
```

该处理的作用是将可继承的静态属性和方法复制到类中，并在其原型的 $inheritableStatics 中保存要继承的静态属性和方法的名称。**在这里要注意 statics 属性与 inheritableStatics 属性的区别：statics 属性是用来定义静态类的，而静态类是不能继承的；inheritableStatics 属性是用来定义类中的静态属性和方法的，而这些属性和方法是可以被子类继承的。**

代码首先检查类的原型中是否有要继承的静态属性和方法，如果没有，则将原型中的 $inheritableStatics 属性指向一个空数组，然后将配置对象在 inheritableStatics 对象中定义的属性和方法复制到类中，同时 $inheritableStatics 数组中记录这些属性和方法名称。最后删除 inheritableStatics 属性，结束处理。

接着进行的处理是 congfig，其代码如下：

```
ExtClass.registerPreprocessor('config', function(Class, data) {
    var config = data.config,
        configNameCache = ExtClass.configNameCache,
        prototype = Class.prototype,
        emptyFn = Ext.emptyFn;

    delete data.config;

    Ext.Object.each(config, function(name) {
        var capitalizedName, customIniter, customGetter;

    if (!configNameCache[name]) {
        capitalizedName = name.charAt(0).toUpperCase() + name.substr(1);

        configNameCache[name] = {
            internal: '_' + name,
            apply: 'apply' + capitalizedName,
            update: 'update' + capitalizedName,
            'set': 'set' + capitalizedName,
            'get': 'get' + capitalizedName,
            init: 'init' + capitalizedName
        };
    }
    var nameMap = configNameCache[name],
        internalName = nameMap.internal,
        applyName = nameMap.apply,
        updateName = nameMap.update,
        setName = nameMap.set,
        getName = nameMap.get,
        initName = nameMap.init,
        optimizedGetter;

    if (!(setName in prototype) && !data.hasOwnProperty(setName)) {
        data[setName] = function(value) {
            var oldValue = this[internalName],
                applier = this[applyName],
                updater = this[updateName],
                initer = this[initName];

            if (initer !== emptyFn) {
                this[initName] = emptyFn;
            }

            if (typeof applier == 'function') {
                value = applier.call(this, value, oldValue);
            }

    if (typeof value != 'undefined') {
        this[internalName] = value;

        if (typeof updater == 'function' && value !== oldValue &&
            !(value === null && oldValue === undefined)) {
            updater.call(this, value, oldValue);
        }
    }
```

```
            return this;
        };
    }

    if (!(getName in prototype) || data.hasOwnProperty(getName)) {
        customGetter = data[getName] || false;

        if (customGetter) {
            optimizedGetter = function() {
                return customGetter.apply(this, arguments);
            };
        }
        else {
            optimizedGetter = new Function('return this.'+internalName);
        }

        data[getName] = function() {
            var initer = this[initName],
                currentGetter;

            if (initer !== emptyFn) {
                this[initName] = emptyFn;
                initer.call(this, this.config[name]);
            }

            currentGetter = this[getName];

            if ('$previous' in currentGetter) {
                currentGetter.$previous = optimizedGetter;
            }
            else {
                this[getName] = optimizedGetter;
            }

            return optimizedGetter.apply(this, arguments);
        };
    }

    if (data.hasOwnProperty(initName)) {
        customIniter = data[initName];
        data[initName] = function(value) {
            this[initName] = emptyFn;
            customIniter.call(this, value);
        };
    }
    else if (!(initName in prototype)) {
        data[initName] = function(value) {
            this[initName] = emptyFn;
            this[setName](value);
        };
    }
});

Class.addConfig(config);
});
```

通过 2.7.7 节使用 Ext.define 定义新类一节，我们已经指定在 congfig 中定义的配置项会自动添加 get、set 和 apply 方法，而 congfig 处理就是实现该功能的。

代码首先使用 Ext.Objext 的 each 方法枚举配置项。在枚举执行的函数中，首先将配置项的名称首字符变成大写，然后构造出对应的方法名，例如配置项的名称为 width，那么就构造出 applyWidth、updateWidth、setWidth、getWidth 和 initWidth 这五个方法名。

接着检查在原型中是否已经有同名的方法，且配置对象中是否已定义该方法，如果都没有，则在配置对象中添加该方法，其指向为函数，函数会返回传递过来的 val 参数。

接下来是生成 set 方法，该方法会执行一次 apply 方法，如果其返回值存在，就将返回值作为属性值。

之后生成 get 方法，该方法直接返回属性值。接着生成 init 方法。最后调用 addConfig 方法，其代码如下：

```
addConfig: function(config) {
    var prototype = this.prototype,
        hasConfig = prototype.$hasConfig,
        configList = prototype.$configList,
        defaultConfig = prototype.config,
        name;

    for (name in config) {
        if (config.hasOwnProperty(name)) {
            if (!hasConfig[name]) {
                hasConfig[name] = true;
                configList.push(name);
            }
        }
    }

    Ext.merge(defaultConfig, config);

    prototype.$configClass = Ext.Object.classify(defaultConfig);
},
```

从代码可以看到，类原型的“$hasConfig”的值可决定是否需要生成方法（true 要生成），而“$configList”则保存了有生成方法的配置项。

接着使用 Ext.Objext 的 merge 方法将配置对象内的属性和方法合并到新类原型的 congfig 属性中，并删除配置对象的 config 属性。

最后将“$configClass”属性指向由 classify 方法生成的函数，该函数可以查询配置项及将生成的方法实例化到类中。

接着要进行的是 mixins 处理，其代码如下：

```
ExtClass.registerPreprocessor('mixins', function(Class, data, hooks) {
    var mixins = data.mixins,
        name, mixin, i, ln;

    delete data.mixins;
```

```
Ext.Function.interceptBefore(hooks, 'onCreated', function() {
    if (mixins instanceof Array) {
        for (i = 0,ln = mixins.length; i < ln; i++) {
            mixin = mixins[i];
            name = mixin.prototype.mixinId || mixin.$className;

            Class.mixin(name, mixin);
        }
    }
    else {
        for (name in mixins) {
            if (mixins.hasOwnProperty(name)) {
                Class.mixin(name, mixins[name]);
            }
        }
    }
});
```

代码会调用 interceptBefore 方法修改 hooks 对象中的 onCreated 方法，在调用 onCreated 之前先执行指定的函数。函数内，会根据配置对象的 mixins 属性的值是数组还是对象分别使用不同的循环方式调用 mixin 方法，其代码如下：

```
mixin: function(name, mixinClass) {
    var mixin = mixinClass.prototype,
        prototype = this.prototype,
        key;

    for (key in mixin) {
        if (mixin.hasOwnProperty(key)) {
            if (typeof prototype[key] == 'undefined' && key !== 'mixins' && key !=
'mixinId') {
                prototype[key] = mixin[key];
            }
            else if (key === 'config') {
                this.addConfig(mixin[key]);
            }
        }
    }

    if (typeof mixin.onClassMixedIn != 'undefined') {
        mixin.onClassMixedIn.call(mixinClass, this);
    }

    if (!prototype.hasOwnProperty('mixins')) {
        if ('mixins' in prototype) {
            prototype.mixins = Ext.Object.chain(prototype.mixins);
        }
        else {
            prototype.mixins = {};
        }
    }

    prototype.mixins[name] = mixin;
},
```

代码的主要功能就是把混入类原型中的成员和配置（config）复制到新类中。如果混入类定义了 onClassMixedIn 方法，执行它。还要把混入对象记录在原型的 mixins 属性下。

接着要处理的是 xtype，其代码如下（在 classManager.js）：

```
Class.registerPreprocessor('xtype', function(cls, data) {
    var xtypes = Ext.Array.from(data.xtype),
        widgetPrefix = 'widget.',
        aliases = Ext.Array.from(data.alias),
        i, ln, xtype;

    data.xtype = xtypes[0];
    data.xtypes = xtypes;

    aliases = data.alias = Ext.Array.from(data.alias);

    for (i = 0,ln = xtypes.length; i < ln; i++) {
        xtype = xtypes[i];

        //省略调试代码

        aliases.push(widgetPrefix + xtype);
    }

    data.alias = aliases;
});
```

代码主要是把配置对象的 xtype 属性中的值与 alias 中的值合并在一起。在合并前，会在 xtype 的值前加上前缀“widget”，这也是定义别名时必须用到的前缀。最后把结果保存在配置对象的 alias 属性中。

最后要处理的是 alias，其代码如下（在 classManager.js）：

```
Class.registerPreprocessor('alias', function(cls, data) {
    var aliases = Ext.Array.from(data.alias),
        xtypes = Ext.Array.from(data.xtypes),
        widgetPrefix = 'widget.',
        widgetPrefixLength = widgetPrefix.length,
        i, ln, alias, xtype;

    for (i = 0, ln = aliases.length; i < ln; i++) {
        alias = aliases[i];

        //省略调试代码

        if (alias.substring(0, widgetPrefixLength) === widgetPrefix) {
            xtype = alias.substring(widgetPrefixLength);
            Ext.Array.include(xtypes, xtype);

            if (!cls.xtype) {
                cls.xtype = data.xtype = xtype;
            }
        }
    }
```

```
        data.alias = aliases;
        data.xtypes = xtypes;
    });
```

这里的代码把别名（alias 属性指向的数组）逐个拆出来，然后把前缀去掉，将结果保存到配置对象的 xtypes 属性中。过程中会使用 include 方法处理掉重复项。

至此，在 Ext.Class 中的处理过程就完成了。因为 preprocessor 再也找不到数据，不存在了，就执行 onBeforeClassCreated 方法，其代码如下：

```
onBeforeCreated: function(Class, data, hooks) {
    Class.addMembers(data);
    hooks.onCreated.call(Class, Class);
},
```

代码先调用 addMembers 方法，其代码如下：

```
addMember: function(name, member) {
    if (typeof member == 'function' && !member.$isClass && member !== Ext.emptyFn) {
        member.$owner = this;
        member.$name = name;
        member.displayName = (this.$className || '') + '#' + name;
    }

    this.prototype[name] = member;
    return this;
},
```

代码主要是为类添加“$owner”、“$name”和 displayName 等成员。

接着调用 hooks 对象的 onCreated 方法，也就是在本节开头的 create 方法中，使用 new 关键字创建新类中的回调函数，将返回到 ClassManager 的处理过程。

4.4.3　所有继承类的基类：Ext.Base

在上一节中可以看到，在使用 define 方法定义新类的时候会把 Base 类的成员复制到新类，而这些成员大部分都会在类的创建过程中被调用，是名副其实的基类。

4.4.4　实现动态加载：Ext.Loader

Loader 对象在 Class 对象中注册一个 loader 处理器，用来实现类的动态加载，其代码如下：

```
Class.registerPreprocessor('loader', function(cls, data, hooks, continueFn) {
    var me = this,
    dependencies = [],
    className = Manager.getName(cls),
    i, j, ln, subLn, value, propertyName, propertyValue;

for (i = 0,ln = dependencyProperties.length; i < ln; i++) {
    propertyName = dependencyProperties[i];

    if (data.hasOwnProperty(propertyName)) {
        propertyValue = data[propertyName];
```

```
if (typeof propertyValue == 'string') {
    dependencies.push(propertyValue);
}
else if (propertyValue instanceof Array) {
    for (j = 0, subLn = propertyValue.length; j < subLn; j++) {
        value = propertyValue[j];

        if (typeof value == 'string') {
            dependencies.push(value);
        }
    }
    else if (typeof propertyValue != 'function') {
        for (j in propertyValue) {
            if (propertyValue.hasOwnProperty(j)) {
                value = propertyValue[j];

                if (typeof value == 'string') {
                    dependencies.push(value);
                }
            }
        }
    }
}

if (dependencies.length === 0) {
    return;
}

var deadlockPath = [],
    requiresMap = Loader.requiresMap,
    detectDeadlock;

if (className) {
    requiresMap[className] = dependencies;
    if (!Loader.requiredByMap) Loader.requiredByMap = {};
    Ext.Array.each(dependencies, function(dependency){
        if (!Loader.requiredByMap[dependency]) Loader.requiredByMap[dep-
            endency] = [];
        Loader.requiredByMap[dependency].push(className);
    });
    detectDeadlock = function(cls) {
        deadlockPath.push(cls);

        if (requiresMap[cls]) {
            if (Ext.Array.contains(requiresMap[cls], className)) {
                throw new Error("Deadlock detected while loading dependencies!
                    '" + className + "' and '" +
                        deadlockPath[1] + "' " + "mutually require each other.
                            Path: " +
                        deadlockPath.join(' -> ') + " -> " + deadlockPath[0]);
            }
```

```
                for (i = 0,ln = requiresMap[cls].length; i < ln; i++) {
                    detectDeadlock(requiresMap[cls][i]);
                }
            }
        };

        detectDeadlock(className);
    }


    Loader.require(dependencies, function() {
        for (i = 0,ln = dependencyProperties.length; i < ln; i++) {
            propertyName = dependencyProperties[i];

            if (data.hasOwnProperty(propertyName)) {
                propertyValue = data[propertyName];

                if (typeof propertyValue == 'string') {
                    data[propertyName] = Manager.get(propertyValue);
                }
                else if (propertyValue instanceof Array) {
                    for (j = 0, subLn = propertyValue.length; j < subLn; j++) {
                        value = propertyValue[j];

                        if (typeof value == 'string') {
                            data[propertyName][j] = Manager.get(value);
                        }
                    }
                }
                else if (typeof propertyValue != 'function') {
                    for (var k in propertyValue) {
                        if (propertyValue.hasOwnProperty(k)) {
                            value = propertyValue[k];

                            if (typeof value == 'string') {
                                data[propertyName][k] = Manager.get(value);
                            }
                        }
                    }
                }
            }
        }

        continueFn.call(me, cls, data, hooks);
    });

    return false;
}, true, 'after', 'className');
```

通过代码最后一行的参数 true，可以知道该处理器是必须执行的，而且必须在 className 处理器之后执行。

在代码中，变量 dependencyProperties 的定义如下：

```
dependencyProperties = ['extend', 'mixins', 'requires'],
```

也就是说，配置对象的 extend、mixins 和 requires 属性中定义的对象都是要检查是否已经加载的。

循环中，会根据 extend、mixins 和 requires 属性的值是字符串、数组或对象，使用不同的处理方式将对象名取出并保存到变量 dependencies 指向的数组中。

如果 dependencies 的长度为 0，说明不需要检查对象是否已加载，直接返回。

变量 className 是使用 ClassManager 对象的 getName 方法返回的新类的类名。

对象 requiresMap 可以说是一个缓存对象，在对象中保存了以类名作为关键字、它所依赖的类名组成的数组作为值的依赖表。通过依赖表就可轻松地实现按需动态加载类了。

如果类名存在，首先在依赖表中添加一个类名为关键字、依赖数组为值的成员。

接着构建一个递归函数 detectDeadlock，通过该递归函数可以根据类名从依赖表中找出所有依赖的类，例如 A 类依赖于 B 类，而 B 类依赖 C 和 D 类，而 C 类依赖 E 类，那么要正确定义 A 类，必须能访问 B、C、D 及 E 等类。不过该方法会存在死锁情况，就是依赖情况可能出现环形链，因而要特别小心。

所有这些找到的类名都会保存到 deadlockPath 指向的数组中。

找齐所有依赖类后，就执行 Loader 对象的 require 方法加载类，其代码如下：

```
require: function(expressions, fn, scope, excludes) {
    var excluded = {},
        included = {},
        queue = this.queue,
        classNameToFilePathMap = this.classNameToFilePathMap,
        isFileLoaded = this.isFileLoaded,
        excludedClassNames = [],
        possibleClassNames = [],
        classNames = [],
        references = [],
        callback,
        syncModeEnabled,
        filePath, expression, exclude, className,
        possibleClassName, i, j, ln, subLn;

    if (excludes) {
        excludes = arrayFrom(excludes);

        for (i = 0,ln = excludes.length; i < ln; i++) {
            exclude = excludes[i];

            if (typeof exclude == 'string' && exclude.length > 0) {
                excludedClassNames = Manager.getNamesByExpression(exclude);

                for (j = 0,subLn = excludedClassNames.length; j < subLn; j++) {
                    excluded[excludedClassNames[j]] = true;
                }
            }
        }
    }

    expressions = arrayFrom(expressions);
```

```
if (fn) {
    if (fn.length > 0) {
        callback = function() {
            var classes = [],
                i, ln, name;

            for (i = 0,ln = references.length; i < ln; i++) {
                name = references[i];
                classes.push(Manager.get(name));
            }

            return fn.apply(this, classes);
        };
    }
    else {
        callback = fn;
    }
}
else {
    callback = Ext.emptyFn;
}

scope = scope || Ext.global;

for (i = 0,ln = expressions.length; i < ln; i++) {
    expression = expressions[i];

    if (typeof expression == 'string' && expression.length > 0) {
        possibleClassNames = Manager.getNamesByExpression(expression);
        subLn = possibleClassNames.length;

        for (j = 0; j < subLn; j++) {
            possibleClassName = possibleClassNames[j];

            if (excluded[possibleClassName] !== true) {
                references.push(possibleClassName);

                if (!Manager.isCreated(possibleClassName) && !included-
                    [possibleClassName]) {
                    included[possibleClassName] = true;
                    classNames.push(possibleClassName);
                }
            }
        }
    }
}

if (classNames.length > 0) {
    if (!this.config.enabled) {
        throw new Error("Ext.Loader is not enabled, so dependencies cannot be
            resolved dynamically. " +
                "Missing required class" + ((classNames.length > 1) ? "es" :
                    "") + ": " + classNames.join(', '));
    }
```

```
        }
        else {
            callback.call(scope);
            return this;
        }

        queue.push({
            requires: classNames.slice(),
            callback: callback,
            scope: scope
        });

        for (i = 0,ln = classNames.length; i < ln; i++) {
            className = classNames[i];

            if (!isFileLoaded.hasOwnProperty(className)) {
                isFileLoaded[className] = false;
                filePath = this.getPath(className);
                classNameToFilePathMap[className] = filePath;
                this.numPendingFiles++;
                syncModeEnabled = this.syncModeEnabled;
                this.loadScriptFile(
                    filePath,
                    pass(this.onFileLoaded, [className, filePath], this),
                    pass(this.onFileLoadError, [className, filePath]),
                    this,
                    syncModeEnabled
                );
                if (ln === 1 && syncModeEnabled) {
                    return Manager.get(className);
                }
            }
        }
        return this;
    },
```

代码先将把参数 excludes（要排除的类）转换到 excluded 对象中。

接着把要加载的类转换为数组（expressions），并处理回调函数。

之后把要加载的类与排除的类做比较，将实际要加载的类放到 classNames 数组中。

如果没有开启动态加载（Loader 对象的 enabled 属性不为 true），那么抛出异常。如果没有要加载的类，直接执行回调函数并返回。

如果有需要加载的类，在加载队列（queue）中加入一个包含 requires（要加载的类）、callback（回调函数）和 scope（作用域）这三个成员的对象。

接着要做的就是遍历依赖数组中的类名，在 isFileLoaded 对象中记录当前文件的加载状态为 false，然后获取类文件路径（getPath），加载计数器 numPendingFiles 加 1，最后使用 loadScriptFile 方法开始加载文件。

当文件加载成功会调用 onFileLoaded 方法，其代码如下：

```
onFileLoaded: function(className, filePath) {
    this.numLoadedFiles++;
```

```
            this.isFileLoaded[className] = true;
            this.numPendingFiles--;
            if (this.numPendingFiles === 0) {
                this.refreshQueue();
            }

        // 省略调试信息代码
        },
```

如果加载成功，文件加载数量计数器 numLoadedFiles 会加 1，对象 isFileLoaded 中该类的加载状态设置为 true，计数器 numPendingFiles 减 1，如果 numPendingFiles 的值为 0，说明全部文件已加载完成，使用 refreshQueue 方法刷新队列，其代码如下：

```
refreshQueue: function() {
    var ln = this.queue.length,
        i, item, j, requires;
    if (ln === 0) {
        this.triggerReady();
        return;
    }
    for (i = 0; i < ln; i++) {
        item = this.queue[i];
        if (item) {
            requires = item.requires;
            if (requires.length > this.numLoadedFiles) {
                continue;
            }
            j = 0;
            do {
                if (Manager.isCreated(requires[j])) {
                    requires.splice(j, 1);
                }
                else {
                    j++;
                }
            } while (j < requires.length);

            if (item.requires.length === 0) {
                this.queue.splice(i, 1);
                item.callback.call(item.scope);
                this.refreshQueue();
                break;
            }
        }
    }

    return this;
},
```

代码会遍历队列中的对象，如果依赖类（requires）数组的长度少于文件加载数量，则进入 while 循环。循环中，使用 ClassManager 对象的 isCreated 方法检查类是否已经被创建，如果已创建，则将该类移出数组，否则索引加 1。如果最后的依赖类数组长度为 0，说明已完成依赖类的加载，要将该队列对象移出队列，接着执行队列对象中的回调函数，最后重新

执行 refreshQueue 方法，并中断 for 循环的执行。

在 loadScriptFile 方法中，如果是异步方式（该方式可通过 syncModeEnabled 配置项设置），会使用 SCRIPT 标记加载类，如果是同步方式，则会使用 Ajax 方式加载，其代码如下：

```
loadScriptFile: function(url, onLoad, onError, scope, synchronous) {
    var me = this,
        noCacheUrl = url + (this.getConfig('disableCaching') ? ('?' + this.getCon
            fig('disableCachingParam') + '=' + Ext.Date.now()) : ''),
        fileName = url.split('/').pop(),
        isCrossOriginRestricted = false,
        xhr, status, onScriptError;

    scope = scope || this;

    this.isLoading = true;

    if (!synchronous) {
        onScriptError = function() {
            onError.call(scope, "Failed loading '" + url + "', please verify that
                the file exists", synchronous);
        };

        if (!Ext.isReady && Ext.onDocumentReady) {
            Ext.onDocumentReady(function() {
                me.injectScriptElement(noCacheUrl, onLoad, onScriptError, scope);
            });
        }
        else {
            this.injectScriptElement(noCacheUrl, onLoad, onScriptError, scope);
        }
    }
    else {
          // 忽略采用同步模式（Ajax）加载模式加载代码
    }
},
```

如果 DOM 树还没初始化完成，则在 onDocumentReady 中加入一个函数，等到 DOM 树初始化完成时再执行加载操作。否则，直接执行 injectScriptElement 方法，加载类文件，其代码如下：

```
injectScriptElement: function(url, onLoad, onError, scope) {
    var script = document.createElement('script'),
        me = this,
        onLoadFn = function() {
            me.cleanupScriptElement(script);
            onLoad.call(scope);
        },
        onErrorFn = function() {
            me.cleanupScriptElement(script);
            onError.call(scope);
        };

script.type = 'text/javascript';
```

```
script.src = url;
script.onload = onLoadFn;
script.onerror = onErrorFn;
script.onreadystatechange = function() {
    if (this.readyState === 'loaded' || this.readyState === 'complete') {
        onLoadFn();
    }
};

this.documentHead.appendChild(script);
    return script;
},
```

代码会创建一个 script 元素，并为其绑定 onload、onErrorFn 和 onreadystatechange 三个事件。只要加载完成都会触发参数 onLoad 指向的回调函数，也就是 onFileLoaded 方法。

现在返回调用 require 方法时定义的回调函数代码。

代码会遍历 extend、mixins 和 requires 属性中的属性，并使用 ClassManager 对象的 get 方法返回的对象作为其值，也就是说，在这里，原本是类名的关键字已经指向对象了。

在 Loader 对象中，还在 ClassManager 对象中注册了一个 uses 处理，目的是加载配置对象的 uses 属性中的类，其代码如下：

```
Manager.registerPostprocessor('uses', function(name, cls, data) {
    var uses = Ext.Array.from(data.uses),
        items = [],
        i, ln, item;

    for (i = 0, ln = uses.length; i < ln; i++) {
        item = uses[i];

        if (typeof item === 'string') {
            items.push(item);
        }
    }

    Loader.addOptionalRequires(items);
});
```

代码先将配置对象 uses 属性中的类名提取出来保存到变量 items 指向的数组，然后使用 addOptionalRequires 将它们加入到 Loader 对象的 optionalRequires 数组。而 optionalRequires 数组中的类会在 triggerReady 方法中加载。

回到 refreshQueue 方法，可以看到，当队列的长度为 0 时，才会执行 triggerReady 方法，也就是说，只有在队列中的依赖类加载完成后，才会加载可选的类。

至此，类的创建过程已经完成了大半，余下的工作就是在 ClassManager 对象中注册类了。

4.4.5　管理类的类：Ext.ClassManager

在 Ext.Class 的处理过程完成后，会执行回调函数中，进行余下的处理，该处理过程与 Ext.Class 中的一样，保存在 defaultPostprocessors 数组中，从控制台的输出可以看到其包含的

内容如下：

```
["alias", "singleton", "alternateClassName", "uses"]
```

在 Ext.Class 中，别名（alias）的处理只是将别名整理好存回配置对象的 alias 属性中，而实际的处理是在 ClassManager 中完成的，其处理代码如下：

```
Manager.registerPostprocessor('alias', function(name, cls, data) {
    var aliases = data.alias,
        i, ln;

    for (i = 0,ln = aliases.length; i < ln; i++) {
        alias = aliases[i];
        this.setAlias(cls, alias);
    }
}, ['xtype', 'alias']);
```

在这里调用了 setAlias 方法，其代码如下：

```
setAlias: function(cls, alias) {
    var aliasToNameMap = this.maps.aliasToName,
        nameToAliasesMap = this.maps.nameToAliases,
        className;

    if (typeof cls == 'string') {
        className = cls;
    } else {
        className = this.getName(cls);
    }

    if (alias && aliasToNameMap[alias] !== className) {
        //省略调试信息
        aliasToNameMap[alias] = className;
    }

    if (!nameToAliasesMap[className]) {
        nameToAliasesMap[className] = [];
    }

    if (alias) {
        Ext.Array.include(nameToAliasesMap[className], alias);
    }

    return this;
},
```

从代码中可以看到，setAlias 方法会在别名、备用名与类名之间建立一个映射表（maps），映射表的结构如下：

```
maps: {
    alternateToName: {},
    aliasToName: {},
    nameToAliases: {}
},
```

通过该表，就可通过备用名找到类名；通过别名找到类名；通过类名找到别名。完成别名处理后，要进行 singleton 处理，也就是单件模式的类的处理，其代码如下：

```
Manager.registerPostprocessor('singleton', function(name, cls, data, fn) {
    fn.call(this, name, new cls(), data);
    return false;
});
```

从代码可以看到，如果是单件模式的类，则会返回类的一个实例。

接着进行的是 alternateClassName 处理，其代码如下：

```
Manager.registerPostprocessor('alternateClassName', function(name, cls, data) {
    var alternates = data.alternateClassName,
        i, ln, alternate;

    if (!(alternates instanceof Array)) {
        alternates = [alternates];
    }

    for (i = 0, ln = alternates.length; i < ln; i++) {
        alternate = alternates[i];

        //省略调试代码
        this.set(alternate, cls);
    }
});
```

代码会调用 set 方法处理类的别名，其代码如下：

```
set: function(name, value) {
    var targetName = this.getName(value);
    this.classes[name] = this.setNamespace(name, value);
    if (targetName && targetName !== name) {
        this.maps.alternateToName[name] = targetName;
    }
    return this;
},
```

从代码可以看到，set 方法会在 classes 指向的对象内添加一个以参数 name 为关键字、类为值的成员。这也是类注册的方法。因而通过 classes 属性，可由类名、别名找到类。

最后要做的是 uses 处理，该处理过程在 Loader 内，具体可查看 4.4.4 节。

至此，一个类的处理过程就全部完成了。

4.4.6　类创建的总结

从以上分析可以总结出类的创建主要是如图 4-5 所示的流程。

在流程中使用了 13 个处理器，依次分别处理了类名、动态加载、继承、静态类、静态方法、配置项、混入、xtype、别名、单件模式、备用名和可选类等配置项。基本涵盖了 Ext JS 4 中所有类的功能。

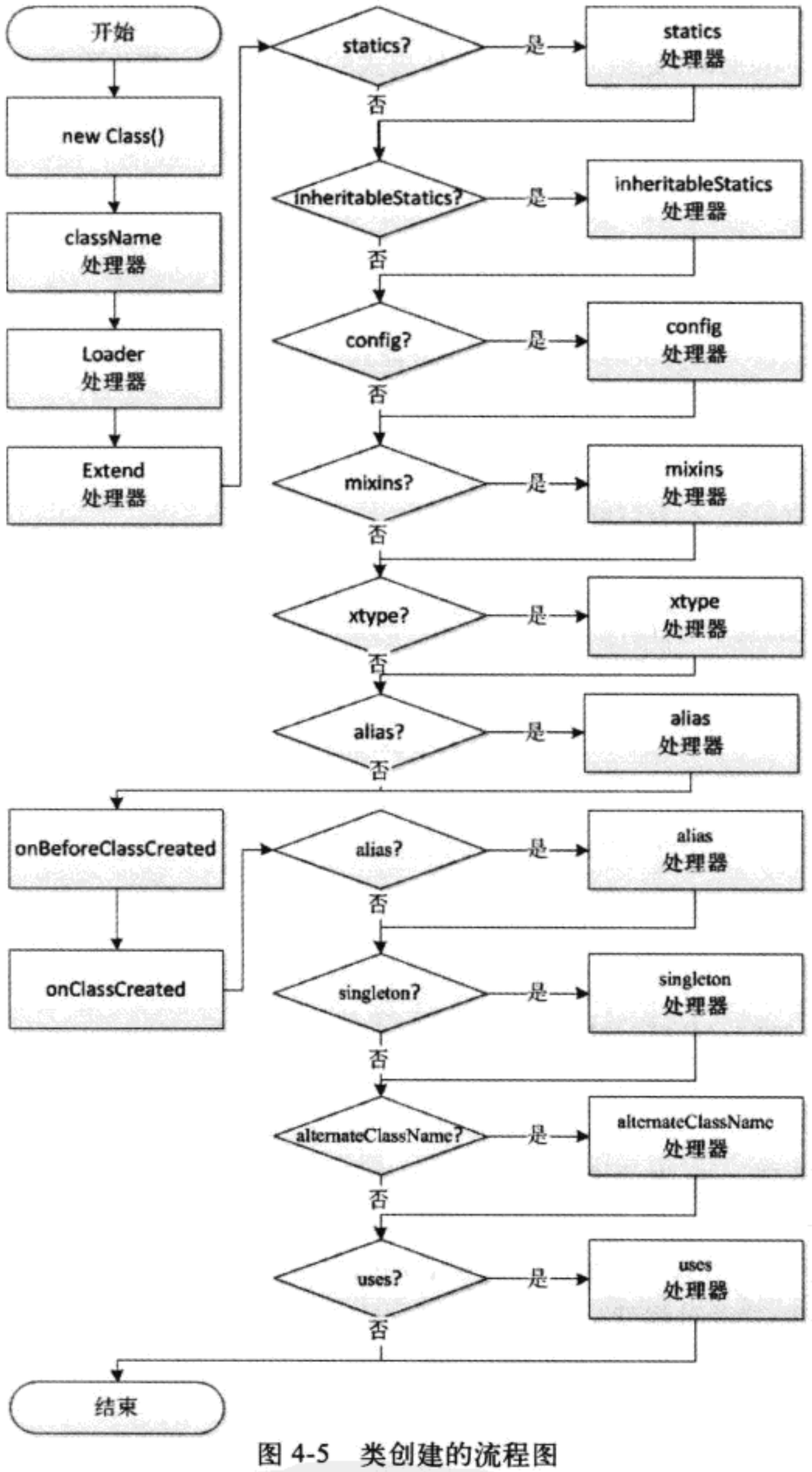


图 4-5　类创建的流程图

该方法有一个主要优点，就是易于扩展。当有新的处理时，可以注册进处理器列表，这样在配置项有对应配置的时候，就可以启用处理器进行处理。其缺点是比 Ext JS 3 的处理过程复杂，从而降低了性能。不过，随着浏览器的发展，这些性能的损失是可以弥补的，更易于扩展的类模型的优势足以弥补性能降低的劣势。

4.5　动态加载的路径设置

在 4.4.4 节，动态加载是使用 getPath 方法获取下载路径的，其代码如下：

```
getPath: function(className) {
    var path = '',
        paths = this.config.paths,
        prefix = this.getPrefix(className);

    if (prefix.length > 0) {
        if (prefix === className) {
            return paths[prefix];
        }

        path = paths[prefix];
        className = className.substring(prefix.length + 1);
    }

    if (path.length > 0) {
        path += '/';
    }

    return path.replace(/\/\.\//g, '/') + className.replace(/\./g, "/") + '.js';
},
```

从变量 paths 的定义可知，预设路径保存在 Loader 对象的 config 对象的 paths 对象中，其默认配置如下：

```
paths: {
    'Ext': '.'
}
```

也就是说，默认 Ext 的加载路径是根目录。

代码先使用 getPrefix 方法获取类名的前缀，其代码如下：

```
getPrefix: function(className) {
    var paths = this.config.paths,
        prefix, deepestPrefix = '';

    if (paths.hasOwnProperty(className)) {
        return className;
    }

    for (prefix in paths) {
        if (paths.hasOwnProperty(prefix) && prefix + '.' === className.
        substring(0, prefix.length + 1)) {
            if (prefix.length > deepestPrefix.length) {
                deepestPrefix = prefix;
            }
        }
    }

    return deepestPrefix;
},
```

代码会检查 paths 对象中是否包含有以类名为属性名称的属性，如果有，说明该类有直接路径，直接返回就可以了。例如，定义一个类名为“My.App.User”的类，在 paths 对象存在以下定义：

```
'My.App.User':'../app/user.js'
```

说明“My.App.User”类不用计算其下载路径，直接根据 paths 对象中的定义去下载文件就可以了。

如果不存在直接路径，就先找带有路径的前缀，例如“My.App.User”类，如果在 paths 对象中只存在“My”的路径定义，则返回前缀“My”。如果 paths 对象中同时存在“My”和“My.App”的路径定义，则返回前缀“My.App”。如果以上都不存在，则返回空字符串。

回到 getPath 方法，如果返回的前缀不是空字符串，则检查前缀是否与类型相同。如果相同，从 paths 对象中取出路径直接返回。否则取出前缀的路径，并把类名的前缀部分去掉。

接着判断路径（path）是否存在，如果不存在，给路径加一个“/”。

最后把类名转换为带“.js”的文件名加上路径返回。要注意的是，如果类名中还带有“.”，那么会将“.”转换为“/”，也就是当成路径的一部分。

要预设加载路径，可使用 Loader 对象的 setPath 方法。代码很简单，就是将配置对象的成员复制到 paths 路径里，在此就不讲述了。

下面在浏览器中打开 Hello_World.html 文件，在控制台输入以下命令：

```
Ext.Loader.setPath({
    "My":"./app",
    "My.app":"./app",
    "Ext":'./lib/src'
})
console.dir(Ext.Loader.config.paths);
```

命令使用 setPath 方法预设了“My”、“My.app”和“Ext”的加载路径。运行后，在控制台中可看到以下输出：

```
Ext     "./lib/src/"
My      "./app"
My.app   "./app/"
```

该设置是根据一般的项目结构设置的路径。一般情况下都会加载 ext-all.js，因而不用设置 Ext 目录也行，但是如果你喜欢全部动态加载，也可以按示例代码进行设置，把 Ext JS 的源代码放到 lib 的 src 目录里。一般项目中，自定义的类可以放到 app 目录下，类名的命名空间可以随便设置，只要在 paths 对象中指定其路径就可以了。例如若定义了“My.base”、“My.app.user”和“My.app.product”等类，使用“Ext.ux.plugin”等 Ext 插件或扩展，使用 getPath 获得的路径将是：

```
./app/base.js
./app/user.js
./app/product.js
./lib/scr/ux/plugin.js
```

现在的问题是插件也要放到 src 目录下，因而为了方便，可以另外定义插件的目录，例如：

```
Ext.Loader.setPath("Ext.ux","./lib/ux");
```

这样就可以把插件放到 lib 下的 ux 目录了。

经过以上的设置，在 OnReady 方法前使用 Ext.require 方法就可顺利加载库文件了，例如：

```
Ext.Loader.setConfig({enabled: true});
Ext.Loader.setPath({
    "My":"./app",
    "My.app":"./app",
    "Ext":'./lib/src',
   "Ext.ux":"./lib/ux"
});
Ext.require([
    'My.base',
    'My.app.user',
    'My.app.product',
    'Ext.ux.plugin'
]);
Ext.onReady(function(){
    // 应用代码；
});
```

4.6　综合实例：页面计算器

（1）功能描述

使用 addBehaviors 方法实现简单功能的计算器。

（2）实现代码

计算器按钮比较多，因而最大的难题在于如何为按钮添加事件。

如果利用浏览器的事件传递机制，可以在包裹按钮的元素上绑定一个事件，但是这存在一个分支要检查很多的问题，而且一个函数内的代码也比较多，所以不太适合。

如果为每个按钮都绑定一个事件，一个个去写，也比较麻烦，例如，数字键功能雷同，加、减、乘或除这 4 个按钮功能也单一，因而可以分组处理。使用 addBehaviors 方法，可分组为它们添加事件，这样就方便多了。现在要做的就是确定怎么分组，10 个数字键可以分在一组，然后是 4 个运算符号可以为一组，小数点和正负符号键也可以划一组，最后就是清除按钮和计算按钮归一组了。

分组后要考虑的问题是如何通过选择器选择对应的元素。这个不难，给每个组的元素加一个区别于其他组元素的标记就行。而最好的方法就是利用 class 属性，这样选择符也简单。

这些都想好后就可以开始写代码了，先使用模版生成一个 cal.html 文件，然后添加以下样式：

```
table{margin:20px 0 0 20px;font-size:20px;line-height:40px;border:1px solid
    #000;padding:3px;}
th{text-align:center;}
#Calculator{border:1px solid #000;}
#result{width:156px;}
.cal{width:40px;height:40px;text-align:center;}
```

```
.number{width:40px;height:40px;text-align:center;}
.op{width:40px;height:40px;text-align:center;}
.sign{width:40px;height:40px;text-align:center;}
.cmd{width:80px;height:40px;text-align:center;}
```

样式 number 代表数字组，样式 op 代表运算符号组，样式 sign 代表小数点和正负号这一组，样式 cmd 代表“清除”按钮和“计算”按钮这一组。

接着添加以下 HTML 代码：

```
<table cellpadding="1" cellpadding="1" border="0">
<tr style="border:1px solid #000;background:#2159C2;color:#fff">
   <th colspan="4" >计算器</th>
</tr>
<tr>
   <td colspan="4" align="center"><input id="result" readonly="true" style="text-
       align:right;" type="text" value="0" /></td>
</tr>
<tr>
<td colspan="2"><input class="cmd" type="button" value="="/></td>
<td colspan="2"><input class="cmd" type="button" value="C"/></td>
</tr>
<tr>
   <td><input class="number" type="button" value="7"/></td>
   <td><input class="number" type="button" value="8"/></td>
   <td><input class="number" type="button" value="9"/></td>
   <td><input class="op" type="button" value="+"/></td>
</tr>
<tr>
   <td><input class="number" type="button" value="4"/></td>
   <td><input class="number" type="button" value="5"/></td>
   <td><input class="number" type="button" value="6"/></td>
   <td><input class="op" type="button" value="-"/></td>
</tr>
<tr>
   <td><input class="number" type="button" value="1"/></td>
   <td><input class="number" type="button" value="2"/></td>
   <td><input class="number" type="button" value="3"/></td>
   <td><input class="op" type="button" value="*"/></td>
</tr>
<tr>
   <td><input class="number" type="button" value="0"/></td>
   <td><input class="sign" type="button" value="-/+"/></td>
   <td><input class="sign" type="button" value="."/></td>
   <td><input class="op" type="button" value="/"/></td>
</tr>
</table>
```

最后添加以下代码：

```
var cal=function(){
   switch(op){
           case "-":
                  first=parseFloat(first)-parseFloat(second);
                  break;
```

```
case "*":
        first=parseFloat(first)*parseFloat(second);
        break;
case "/":
        second=parseFloat(second)
        if(second!=0)
                first=parseFloat(first)/second;
        break;
default:
        first=parseFloat(first)+parseFloat(second);
        break;
    }
    op="";
    if(arguments.length>0)op=arguments[0];
    second="";
    result.value=first;
}

var first="";
var second="";
var op="";
var result=Ext.getDom("result");
Ext.addBehaviors({
    "input.number@click":function(e,el){
            if(Ext.isEmpty(op)){
                    if(!(el.value==0 && first==0)){
                            first= first+el.value;
                            result.value=first;
                    }
            }else{
                    if(!(el.value==0 && second==0)){
                            second=second+el.value;
                            result.value=second;
                    }
            }
    },
    "input.cmd@click":function(e,el){
            if(el.value=="C"){
                    if(Ext.isEmpty(op)){
                            first="";
                    }else{
                            second="";
                    }
                    result.value="0";
            }else{
                    cal();
            }
    },
    "input.sign@click":function(e,el){
            if(el.value=="."){
                    if(Ext.isEmpty(op)){
                            if(first.toString().indexOf(".")==-1){
                                    first=first+".";
                                    result.value=first;
                            }
```

```
                }else{
                        if(second.toString().indexOf(".")==-1){
                                second=second+".";
                                result.value=second;
                        }                                                    }
        }else{
                if(Ext.isEmpty(op)){
                        first=first*-1;
                        result.value=first;
                }else{
                        second=second*-1;
                        result.value=second;
                }
        }
    },
    "input.op@click":function(e,el){
            if(Ext.isEmpty(op) || Ext.isEmpty(second)){
                    op=el.value;
                    result.value="0";
            }else{
                    cal(el.value);
            }
    }
});
```

因为要实现单击运算符号后可以进行连续运算的功能，所有要将执行运算的函数 cal 独立出来，允许单击等号或单击运算符号后执行运算。

在代码中，first 用来保存第一个数字，second 用来保存第二个数字，op 用来保存运算符号，result 用来指向结果文本框的 DOM 对象。

如果是单击运算符执行的运算，会传递给 cal 一个参数，因而可通过 arguments 属性的长度判断是否有参数，如果有，修改 op 的值。

代码中粗体代码使用的是 addBehaviors 方法为元素添加事件的字符串格式。这样按“@”拆分后，就可以使用选择器选择到元素，再绑定事件了。

事件中的代码比较简单，在此就不一一说明了。

（3）页面效果

在浏览器中打开页面，并计算“87×29”将看到如图 4-6 所示的效果。

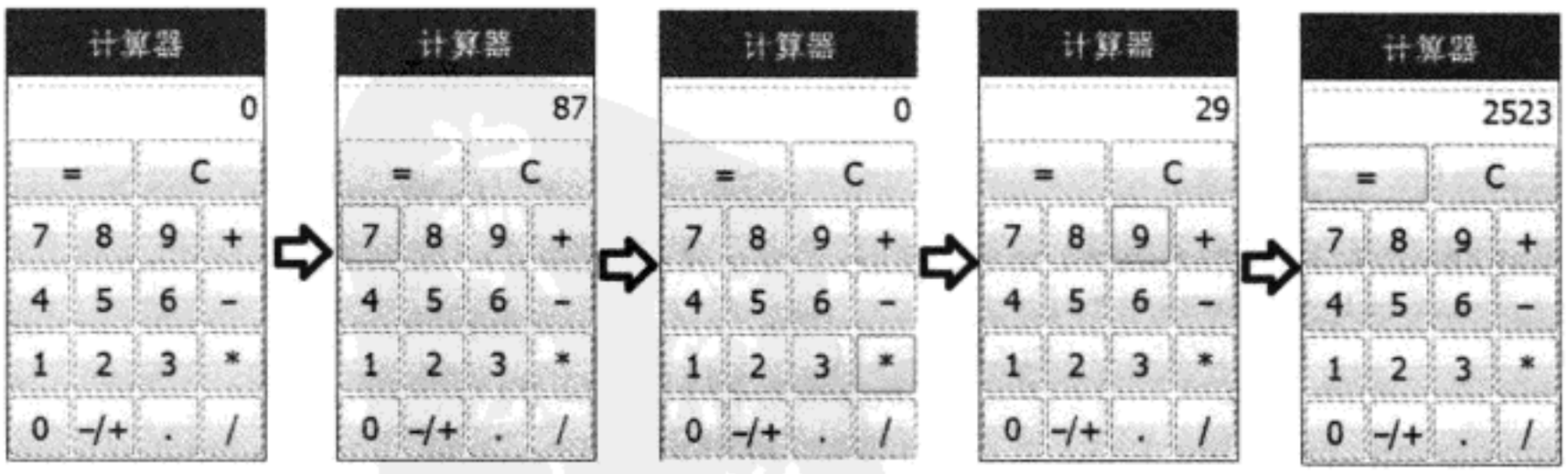


图 4-6　计算器计算“87×29”的效果

4.7　本章小结

每个 JavaScript 框架都有自己的一套基础架构。了解整个基础架构的运行，既可以了解架构的设计理念，又可以从其设计理念中知道如何去学习和使用框架，还可以从架构代码中提高自己写代码的水平。例如，jQuery 是一个函数式架构的框架，因而学习和使用 Jquery 就可从函数这个思路去学习，一切皆函数。而 Ext JS 是面向对象式的架构，因而要就以面向对象的方式去学习和使用。

本章只讲述了基础架构中的平台检测方法和扩展方法及类的创建过程，还剩下事件、DOM 操作这两个基础架构将在下面两章进行讲述。

第 5 章　Ext JS 的事件及其应用

在当前以事件驱动为主流的 Web 应用开发中，其基础当然是事件了。而各浏览器厂商因自身需要使用了不同的事件模型，开发人员如果要开发跨浏览器平台的 Web 应用，则需要处理不同的事件模型兼容的问题。作为一个跨浏览器平台的框架，就需要将这些处理封装起来，为内部的组件和开发人员提供一个公共的接口。因而本章主要讲述的内容包括：Ext JS 如何封装事件模型、如何提供公共接口以及如何使用这些公共接口。

5.1　概述

Ext JS 中包含浏览器事件和内部事件。浏览器事件主要是指用户通过操作页面中的元素而触发的事件，如按钮的单击事件、鼠标移动事件等；内部事件则是指组件之间、组件与浏览器之间的联动等事件。例如，如果用户在 Grid 中单击了标题栏进行排序，那么要处理的事件就包括标题栏单击事件、GridView 的刷新事件、Store 的排序事件等许多相关事件。

Ext JS 封装浏览器事件和提供接口主要是使用 EventManager 和 EventObject 这两个对象，而内部事件主要使用 Event 和 Observable 这两个对象。

5.2　浏览器事件

5.2.1　绑定浏览器事件的过程：Ext.EventManager

要为元素绑定事件，通常会使用 EventManager 的 on 方法，其语法如下：

```
Ext.EventManager.on(el,eventNmae,fn[,scope,options]);
```

其中 el 是要绑定的事件的元素，可以为元素的 id、Element 对象或 HTMLElement 对象。eventName 一般情况下是事件的名称，但在一次定义多个事件时，也可以是一个对象。当 eventName 是事件名称时，fn 就是事件要触发的函数，当 eventName 是对象时，参数 fn 会被忽略。scope 是可选参数，为函数的作用域，当 eventName 为对象时，该参数也会被忽略。options 是可选参数，为事件触发函数的配置对象，表 5-1 列出了它包含的配置项及说明，当 eventName 为对象时，该参数也会被忽略。

表 5-1　options 配置对象的配置项

配置项	说　明
scope	事件触发函数的作用域，默认作用域是触发事件的元素
delegate	简单选择符，用于过滤目标或寻找目标的子节点
stopEvent	如果为 true，会停止事件传播及阻止默认行为，一般用于屏蔽浏览器右键弹出菜单
preventDefault	如果为 true，会阻止默认行为
stopPropagation	如果为 true，会阻止事件传播
normalized	如果为 false，传递给触发函数的是浏览器事件，而不是 EventObject 对象
delay	指定一个时间，单位为毫秒。事件触发后，在该指定时间后才执行触发函数
single	决定触发函数是否只执行一次。如果为 true，则在事件触发后会移除该事件以保证触发函数只执行一次
buffer	指定一个时间，单位为毫秒。事件触发后，在该指定时间内再次触发事件时不会执行触发函数。只有在该指定时间过后触发事件时，才会执行触发函数
target	只有在触发事件的元素是 target 指定的元素时，才会执行触发函数。如果是 target 指定的元素的子元素通过事件传播机制触发的事件，不执行触发函数

下面来研究一下这个事件绑定过程是怎样的。首先在 EventManager.js 找到 on 方法的定义：

```
Ext.EventManager.on = Ext.EventManager.addListener;
```

从代码中可以知道，on 方法其实就是 addListener 方法的别名，而 addListener 的代码如下：

```
addListener: function(element, eventName, fn, scope, options){
    if (typeof eventName !== 'string') {
        this.prepareListenerConfig(element, eventName);
        return;
    }

    var dom = element.dom || Ext.getDom(element),
        bind, wrap;

    //省略调试信息
    options = options || {};

    bind = this.normalizeEvent(eventName, fn);
    wrap = this.createListenerWrap(dom, eventName, bind.fn, scope, options);

    if (dom.attachEvent) {
        dom.attachEvent('on' + bind.eventName, wrap);
    } else {
        dom.addEventListener(bind.eventName, wrap, options.capture || false);
    }

    if (dom == document && eventName == 'mousedown') {
        this.stoppedMouseDownEvent.addListener(wrap);
    }
```

```
        this.getEventListenerCache(element.dom ? element : dom, eventName).push({
            fn: fn,
            wrap: wrap,
            scope: scope
        });
    },
```

代码首先检查 evnetName（事件名称）是否是字符串，如果不是，执行 prepareListenerConfig 方法，其代码如下：

```
prepareListenerConfig: function(element, config, isRemove){
var me = this,
    propRe = me.propRe,
    key, value, args;

for (key in config) {
    if (config.hasOwnProperty(key)) {
        if (!propRe.test(key)) {
            value = config[key];
            if (typeof value == 'function') {
                args = [element, key, value, config.scope, config];
            } else {
                args = [element, key, value.fn, value.scope, value];
            }
            if (isRemove === true) {
                me.removeListener.apply(this, args);
            } else {
                me.addListener.apply(me, args);
            }
        }
    }
}
},
```

在 addListener 方法中传递过来的参数只有 element 和 eventName，因而 prepareListenerConfig 方法接收的参数 config 就是 eventName，而 isRemove 为 undefined。

代码用 for…in 循环遍历 config 中的成员，而且会使用正则表达式 propRe 过滤掉 scope、delay、buffer、single、stopEvent、preventDefault、stopPropagation、normalized、args、delegate 和 freezeEvent 等非事件配置项。如果成员 key 的值为函数，则可直接构建参数数组。否则，把 value 作为一个嵌套定义的配置对象，将其 fn 的值作为 addListener 方法的的触发函数，其 scope 作为作用域，自身作为配置对象。

如果 isRemove 为 true，表示要移除事件，使用 removeListener 方法移除事件，否则使用 addListener 方法绑定事件。

从以上代码可以了解到，绑定事件可以有 3 种形式，下面通过一个例子来演示这 3 种方式。首先，使用示例模板创建一个名称为 5-1.html 的页面文件，然后在 body 中加入 3 个文本框：

```
<input id="input1" type="text" value=" 请输入 "/>
<input id="input2" type="text" value=" 请输入 "/>
<input id="input3" type="text" value=" 请输入 "/>
```

这 3 个文本框除了 id 不同外，其他定义是一样的。接着分别用 3 种绑定事件的方式各为一个文本框绑定 focus 事件和 blur 事件。事件 focus 触发后，如果值为“请输入”，则清空文本框。事件 blur 触发后，如果文本框的值为空，则将其值设置为“请输入”。

因为 3 个文本框的 focus 事件和 blur 事件的执行代码一样，因而可以预先定义好两个函数：

```
var focus=function(e,el){
   if(el.value==" 请输入 ") el.value="";
}
var blur=function(e,el){
   if(el.value=="")el.value=" 请输入 ";
}
```

接着，使用一次只绑定一个事件的方式为 id 为 input1 的文本框绑定两个事件：

```
Ext.EventManager.on("input1","focus",focus);
Ext.EventManager.on("input1","blur",blur);
```

然后，使用一次绑定多个事件，但使用相同配置项的方式为 id 为 input2 的文本框绑定事件：

```
Ext.EventManager.on("input2",{
   focus:focus,
   blur:blur
})
```

最后，使用一次绑定多个事件，使用不同配置项的方式为 id 为 input3 的文本框绑定事件：

```
Ext.EventManager.on("input3",{
   focus:{fn:focus},
   blur:{fn:blur}
})
```

在浏览器中打开页面后，在 3 个文本框中切换焦点，可看到 3 个文本框的行为是一样的。也就是说，三种绑定事件的方式的效果是一样的，至于采用什么样的定义方式，可根据具体情况灵活决定。例如，如果要一次定义多个事件，而且每个事件都有自己的专属行为，那么可以使用第三种方式，否则使用第二种方式更方便。

现在回到 addListener 方法，继续研究源代码。如果 eventName 是字符串，继续执行下面的代码。如果参数 element 存在 dom 属性，则将变量 dom 指向它，否则调用 getDom 方法获取 DOM 节点。接着调用 normalizeEvent 方法绑定事件，其代码如下：

```
normalizeEvent: function(eventName, fn){
    if (this.mouseEnterLeaveRe.test(eventName) && !Ext.supports.MouseEnterLeave) {
        if (fn) {
            fn = Ext.Function.createInterceptor(fn, this.contains, this);
        }
        eventName = eventName == 'mouseenter' ? 'mouseover' : 'mouseout';
    } else if (eventName == 'mousewheel' && !Ext.supports.MouseWheel && !Ext.
        isOpera){
        eventName = 'DOMMouseScroll';
    }
    return {
```

```
            eventName: eventName,
            fn: fn
        };
    },
```

从代码可以看出，normalizeEvent 方法主要是处理浏览器事件的差异，将浏览器事件标准化，而浏览器事件的主要差异在 mouseenter、mouseleave 和 mousewheel 这 3 个事件上。有些浏览器不支持 mouseenter 和 mouseleave 事件，而要使用 Function 对象的 createInterceptor 方法建立一个拦截函数，在触发 mouseenter 或 mouseover 事件后，再执行触发函数，也就是模拟出 mouseenter 或 mouseleave 事件。对于 mousewheel 事件，有些浏览器使用的事件名称是 DOMMouseScroll，所以要更换事件名称。

在创建拦截函数时，使用了 contains 属性指向的函数，其代码如下：

```
contains: function(event){
    var parent = event.browserEvent.currentTarget,
        child = this.getRelatedTarget(event);

    if (parent && parent.firstChild) {
        while (child) {
            if (child === parent) {
                return false;
            }
            child = child.parentNode;
            if (child && (child.nodeType != 1)) {
                child = null;
            }
        }
    }
    return true;
},
```

函数会使用 getRelatedTarget 方法获取与事件的目标节点相关的节点，其代码如下：

```
getRelatedTarget: function(event) {
    event = event.browserEvent || event;
    var target = event.relatedTarget;
    if (!target) {
        if (this.mouseLeaveRe.test(event.type)) {
            target = event.toElement;
        } else if (this.mouseEnterRe.test(event.type)) {
            target = event.fromElement;
        }
    }
    return this.resolveTextNode(target);
},
```

代码直接使用浏览器事件的 relatedTarget 属性返回与事件的目标节点相关的节点。如果节点不存在，则检查事件类型；如果是 mouseout 或 mouseleave 事件，则使用浏览器事件的 toElement 属性返回移入鼠标的元素；如果是 mouseover 或 mouseenter，则使用浏览器事件的 fromElement 属性返回移出鼠标的元素。

节点获取后，使用 resolveTextNode 方法处理浏览器的文本节点之间的差异后再返回，其代码如下：

```
resolveTextNode: Ext.isGecko ?
    function(node) {
        if (!node) {
            return;
        }
        var s = HTMLElement.prototype.toString.call(node);
        if (s == '[xpconnect wrapped native prototype]' || s == '[object
           XULElement]') {
            return;
        }
            return node.nodeType == 3 ? node.parentNode: node;
    }: function(node) {
            return node && node.nodeType == 3 ? node.parentNode: node;
    },
```

如果浏览器是使用 Gecko 引擎的，则执行第一个函数，否则执行第二个函数。

在第一个函数内，如果节点不存在，则直接返回，否则，使用 HTMLElement 对象原型的 toString 方法返回节点的文本信息。若返回的是“[xpconnect wrapped native prototype]”或“[object XULElement]”，则返回 null；若不是，则根据节点类型（nodeType）判断，节点类型是文本节点（值为 3），则返回节点的父节点，否则直接返回节点。

在第二个函数内，如果节点存在且是文本节点，则返回节点的父节点；否则直接返回节点。

在拦截函数 contains 内，要检查与事件的目标节点相关的节点是否包含目标节点，如果包含，返回 false，不执行触发函数；否则，返回 true，执行触发函数。

这可能有点难理解。对于 mouseover 事件来说，属性 relatedTarget 返回的是鼠标指针移出的节点，当属性 relatedTarget 返回的节点包含目标节点，意味着鼠标既在目标节点上，又在离开节点上，也就是说，鼠标实际上没有离开过目标节点，因而不会执行触发函数。而对于 mouseout 事件来说，属性 relatedTarget 返回的是移入节点，当 relatedTarget 属性包含目标节点，实际上鼠标并没有真正移入目标节点，只是在目标节点上移动鼠标，因而也不会执行触发函数。

回到 addListener 方法，接着要做的是使用 createListenerWrap 方法创建封装函数，其代码如下：

```
createListenerWrap : function(dom, ename, fn, scope, options) {
    options = options || {};
    var f, gen;

return function wrap(e, args) {
    if (!gen) {
        f = ['if(!Ext) {return;}'];

        if(options.buffer || options.delay || options.freezeEvent) {
            f.push('e = new Ext.EventObjectImpl(e, ' + (options.freezeEvent ?
                'true' : 'false' ) + ');');
        } else {
```

```
            f.push('e = Ext.EventObject.setEvent(e);');
        }
        if (options.delegate) {
            f.push('var t = e.getTarget("' + options.delegate + '", this);');
            f.push('if(!t) {return;}');
        } else {
            f.push('var t = e.target;');
        }
        if (options.target) {
            f.push('if(e.target !== options.target) {return;}');
        }
        if(options.stopEvent) {
            f.push('e.stopEvent();');
        } else {
            if(options.preventDefault) {
                f.push('e.preventDefault();');
            }
            if(options.stopPropagation) {
                f.push('e.stopPropagation();');
            }
        }
        if(options.normalized === false) {
            f.push('e = e.browserEvent;');
        }
        if(options.buffer) {
            f.push('(wrap.task && clearTimeout(wrap.task));');
            f.push('wrap.task = setTimeout(function(){');
        }
        if(options.delay) {
            f.push('wrap.tasks = wrap.tasks || [];');
            f.push('wrap.tasks.push(setTimeout(function(){');
        }
        f.push('fn.call(scope || dom, e, t, options);');
        if(options.single) {
            f.push('Ext.EventManager.removeListener(dom, ename, fn, scope);');
        }
        if(options.delay) {
            f.push('}, ' + options.delay + '));');
        }
        if(options.buffer) {
            f.push('}, ' + options.buffer + ');');
        }
        gen = Ext.cacheableFunctionFactory('e', 'options', 'fn', 'scope',
            'ename', 'dom', 'wrap', 'args', f.join('\n'));
    }
    gen.call(dom, e, options, fn, scope, ename, dom, wrap, args);
};
},
```

代码中，变量 f 是构建封装函数语句的数组。

如果在配置对象中设置了 buffer、dealy 或 freezeEvent 等属性，则要使用新的 EventObjectImpl 实例。因为如果不使用新的 EventObjectImpl 实例，那么现有的 EventObject 对象就是一个共享对象，事件内的属性会随着操作的改变而改变，所以在触发函数执行时，使用的

是最后的事件结果，这样会出现错误。我们可以通过一个实验来观察一下这两者的区别。

使用示例模板创建一个名称为 5-2.html 的页面文件，然后加入以下 HTML 代码：

```
<div id="div1" style="width:200px;height:200px;
display:block;background:red"></div>
```

然后为 div 绑定一个单击事件：

```
Ext.EventManager.on("div1","click",function(e){
   console.log(e.getXY())
},this,{delay:3000});
```

事件延迟了 3 秒才执行触发事件，触发事件执行后使用 getXY 方法获取单击点的坐标，并在控制台中输出。在浏览器中打开页面，然后在红色区域内的 10 个不同的点单击鼠标，获取 10 个不同的坐标点，笔者获取的数据如下：

```
[49, 35][49, 85][43, 132][123, 153][159, 113][142, 45][105, 58]
[73, 109][89, 152][173, 163]
```

然后在 ext-all-debug.js 中找到 createListenerWrap 方法，将函数内的第一个判断语句屏蔽掉，只保留“f.push('e = Ext.EventObject.setEvent(e);');”这句，然后刷新页面。

先在脚本面板内的文件列表中选择 ext-all-debug.js 文件，然后使用搜索框查找 create-ListenerWrap 方法，确认代码是否已经更改。如果已更改，同样在红色区域内的 10 个不同的点单击鼠标，笔者获取的数据如下：

```
[845, 384][846, 385][846, 385][846, 385][846, 385][846, 385]
[846, 385][846, 385][846, 385][846, 385]
```

比较两组数据，会发现第二组的数据基本是一样的，说明触发函数接收的是最后一次单击时的事件，这在对事件属性有严格要求的环境下会导致错误发生。因而在使用 buffer、delay 或 freezeEvent 设置事件时，要使用新的 EventObjectImpl 实例。

将 ext-all-debug.js 文件修改回原来的样子，继续分析下面的代码。下面的代码就是根据配置对象的配置项添加函数代码。所有代码都已构建好后，可使用 Ext 对象的 cacheable-FunctionFactory 方法构建函数，其代码如下：

```
cacheableFunctionFactory: function() {
    var me = this,
        args = Array.prototype.slice.call(arguments),
        cache = me.functionFactoryCache,
        idx, fn, ln;

    if (Ext.enableSandbox) {
        ln = args.length;
        if (ln > 0) {
            ln--;
            args[ln] = 'var Ext=window.' + me.getUniqueGlobalNamespace() + ';' +
                args[ln];
        }
    }
    idx = args.join('');
```

```
        fn = cache[idx];
        if (!fn) {
            fn = Function.prototype.constructor.apply(Function.prototype, args);

            cache[idx] = fn;
        }
        return fn;
    },
```

方法 cacheableFunctionFactory 会在函数前加一行“var Ext=window.ExtSandbox1”代码，这主要是为在页面中同时使用旧版本的 Ext JS 而做的兼容性设置。

如果没有 buffer、delay 或 freezeEvent 配置项，functionFactory 方法一般会产生以下函数：

```
(function anonymous(e, options, fn, scope, ename, dom, wrap, args) {
    var Ext = window.ExtSandbox1;
    if (!Ext) {return;}
    e = Ext.EventObject.setEvent(e);
    var t = e.target;
    fn.call(scope || dom, e, t, options);}
)
```

如果定义了 buffer 配置项，会生成以下函数：

```
(function anonymous(e, options, fn, scope, ename, dom, wrap, args) {
    var Ext = window.ExtSandbox1;
    if (!Ext) {return;}
    e = new Ext.EventObjectImpl(e, false);
    var t = e.target;
    wrap.task && clearTimeout(wrap.task);
    wrap.task = setTimeout(function () {fn.call(scope || dom, e, t, options);}, 100);
})
```

其他配置项定义生成的函数就不列举了，有兴趣的可以自己研究一下。

生成的函数主要工作是将浏览器的事件转换为 EventObject 对象，再将 EventObject 对象作为触发函数的事件参数传递给触发函数。在这里可以清晰地看到，如果不创建新的 EventObjectImpl 实例，fn 的事件参数将是事件最后一次封装的 EventObject 对象。

最后使用闭包方式将生成的函数封装后返回。

返回 addListener 方法后，将根据浏览器支持的方式使用 attachEvent 方法或 addEventListener 方法将封装好的函数 wrap 绑定到元素。

如果绑定事件的是 document 对象且事件是 mousedown，则需要在 Ext JS 内部事件内绑定该事件，以确保 stopEvent 调用时仍然会触发事件。

最后使用 getEventListenerCache 方法将事件对象保存到缓存中，其代码如下：

```
getEventListenerCache : function(element, eventName) {
    var elementCache, eventCache, id;
    if (!element) {
        return [];
    }

    if (element.$cache) {
```

```
        elementCache = element.$cache;
    } else {
        elementCache = Ext.cache[id = this.getId(element)] || (Ext.cache[id] = {});
    }
    eventCache = elementCache.events || (elementCache.events = {});

    return eventCache[eventName] || (eventCache[eventName] = []);
},
```

如果参数 element 不存在，则直接返回空数组。

如果参数 element 指向的元素有“$cache”属性，则使用该属性缓存事件；否则以元素的 ID 为关键字，在 Ext 的缓存（cache 对象）中创建一个成员用来缓存事件。缓存的事件包括 fn（触发函数）、wrap（封装函数）和 scope（作用域）这三个关键字。这样做的目的是便于在移除事件时找到事件并进行移除操作。

至此，绑定一个浏览器事件就完成了。

5.2.2　封装浏览器事件：Ext.EventObject

在上一节中，我们知道了浏览器事件都需要封装成 EventObjectImpl 对象的实例才能传递给触发函数，因而深入了解一下 EventObjectImpl 对象的运作过程非常有必要。

在上一节中，还了解了 EventObject 是 Ext.EventObjectImpl 对象的实例，相当于一个共享的浏览器事件封装对象，使用 setEvent 方法将浏览器事件封装成 EventObject 对象后再传递给触发函数，从而实现跨平台的处理。因而，setEvent 方法是封装浏览器事件的关键方法。

在讲述 setEvent 方法之前，先了解一下 EventObject 对象的产生过程。使用 new 关键字创建 EventObjectImpl 对象的实例，首先会执行的是构造函数，其代码如下：

```
constructor: function(event, freezeEvent){
    if (event) {
        this.setEvent(event.browserEvent || event, freezeEvent);
    }
},
```

如果参数 event 存在，执行 setEvent 方法；如果不存在时，则不做任何操作。Event-Object 对象的创建代码如下：

```
Ext.EventObject = new Ext.EventObjectImpl();
```

只是创建了一个实例而已，并没有做什么。

以下是 setEvent 方法的代码：

```
setEvent: function(event, freezeEvent){
    var me = this, button, options;

    if (event == me || (event && event.browserEvent)) { // already wrapped
        return event;
    }
    me.browserEvent = event;
    if (event) {
```

```
        button = event.button ? me.btnMap[event.button] : (event.which ? event.
            which - 1 : -1);
        if (me.clickRe.test(event.type) && button == -1) {
            button = 0;
        }
        options = {
            type: event.type,
            button: button,
            shiftKey: event.shiftKey,
            ctrlKey: event.ctrlKey || event.metaKey || false,
            altKey: event.altKey,
            keyCode: event.keyCode,
            charCode: event.charCode,
            target: Ext.EventManager.getTarget(event),
            relatedTarget: Ext.EventManager.getRelatedTarget(event),
            currentTarget: event.currentTarget,
            xy: (freezeEvent ? me.getXY() : null)
        };
    } else {
        options = {
            button: -1,
            shiftKey: false,
            ctrlKey: false,
            altKey: false,
            keyCode: 0,
            charCode: 0,
            target: null,
            xy: [0, 0]
        };
    }
    Ext.apply(me, options);
    return me;
},
```

代码首先检查传递过来的事件是否是已封装好的事件，如果是，则直接返回。如果不是，则将 browserEvent 属性指向浏览器事件。如果浏览器事件存在，先使用按钮对照表（btnMap）标准化按钮值（button）。如果是双击事件，将按钮的值设置为 0。接着在 options 对象内保存事件类型、按钮、shift 键、ctrl 键、alt 键、键盘代码、字符代码、事件目标节点、与事件目标节点相关的节点、当前事件目标节点和事件坐标等信息。如果事件不存在，则使用默认值设置 options 对象。

然后使用 apply 方法将 options 对象的成员复制给 EventObject 对象，最后返回 EventObject 对象。

EventObjectImpl 对象除了提供了事件的基本属性外，还提供了一系列事件处理方法：

- getCharCode：返回事件的字符编码。
- getKey：返回标准化后的键盘代码。
- getPageX 和 getX：返回事件的 x 坐标。getPageX 是旧版本的方法，不建议使用。
- getPageY 和 getY：返回事件的 y 坐标。getPageY 是旧版本的方法，不建议使用。
- getXY：以数组格式返回事件坐标点。

- getPoint：以 Point ⊖对象格式返回事件的坐标点。
- getRelatedTarget：返回与事件目标节点相关的节点。
- getTarget：返回事件目标节点。
- getWheelDelta：返回标准化后的鼠标滚轮滚动值。
- hasModifier：如果键盘的 ctrl、meta、shift 或 alt 键被按下，该值返回 true。
- injectEvent ：使用 EventObject 对象的数据和一个新目标（可选）注入一个 DOM 事件。这是一个多层次的技术，不建议在应用中使用。该方法主要用于 HTMLEditor。
- isNavKeyPress：如果键盘按下的是 PageDown、PageUp、End、Home、箭头键、回车键、Tab 键或 ESC 键，该值返回 true。
- isSpecialKey ：如果键盘按下的是 isNavKeyPress 方法中的键、ctrl、backspace、shift、alt、pause、caps lock、print screen、insert 或 delte 等键，该值返回 true。
- preventDefault：阻止浏览器的默认事件处理。
- stopEvent ：停止事件，执行 preventDefault 方法和 preventDefault 方法。主要用于阻止浏览器右键菜单，以便使用自己的弹出菜单。
- stopPropagation：取消事件传递。
- within：检查事件目标是否是指定元素的子元素，如果是则返回 true，否则返回 false。

5.2.3　移除浏览器事件

EventManager 提供了 un（removeListener）、removeAll 和 purgeElement 这 3 种移除浏览器事件的方法。

1. un（removeListener）

从元素中移除一个事件。

（1）语法

```
Ext.EventManager.un(el,eventNmae,fn,scope);
Ext.EventManager.removeListener(el,eventNmae,fn,scope);
```

其中，el 是要移除事件的元素，它可以为元素的 id、Element 对象或 HTMLElement 对象。eventName 与绑定事件一样，可以为事件名称或配置对象；fn 为要移除的触发函数向，这个必须有，否则无法移除事件；scope 是函数的作用域，必须与绑定事件时的指向一致。方法 un 是 removeListener 的简写。

(2) 源代码

```
removeListener : function(element, eventName, fn, scope) {
    if (typeof eventName !== 'string') {
        this.prepareListenerConfig(element, eventName, true);
        return;
    }

    var dom = Ext.getDom(element),
```

⊖ 相关信息请阅读 16.4.5 节。

```
        el = element.dom ? element : Ext.get(dom),
        cache = this.getEventListenerCache(el, eventName),
        bindName = this.normalizeEvent(eventName).eventName,
        i = cache.length, j,
        listener, wrap, tasks;


    while (i--) {
        listener = cache[i];

        if (listener && (!fn || listener.fn == fn) && (!scope || listener.scope
    === scope)) {
    wrap = listener.wrap;
    if (wrap.task) {
        clearTimeout(wrap.task);
        delete wrap.task;
    }
    j = wrap.tasks && wrap.tasks.length;
    if (j) {
        while (j--) {
            clearTimeout(wrap.tasks[j]);
        }
        delete wrap.tasks;
    }
    if (dom.detachEvent) {
        dom.detachEvent('on' + bindName, wrap);
    } else {
        dom.removeEventListener(bindName, wrap, false);
                    }
                    if (wrap && dom == document && eventName == 'mousedown') {
                        this.stoppedMouseDownEvent.removeListener(wrap);
                    }
                    Ext.Array.erase(cache, i, 1);
                }
            }
        },
```

与绑定事件一样，如果 eventName 是字符串，则先使用 prepareListenerConfig 方法处理事件的配置项，然后逐个移除事件。接着是获取 DOM 节点，使用 getEventListenerCache 方法返回事件缓存，调用 normalizeEvent 方法获取事件的绑定名称。再接着要做的就是从事件缓存中逐个提取绑定事件，如果删除事件和绑定事件是同一事件，则从事件缓存中移除该事件。

比较的方法是触发函数的指针与缓存中触发函数的指针相同，作用域的指针与缓存中的作用域指针相同。

如果事件是 buffer 事件，需要使用 clearTimeout 移除 timeout 事件，最后使用 delete 语句删除 wrap.task。如果事件是 delay 事件，需要使用循环移除 timeout 事件，最后使用 delete 语句删除 wrap.tasks。接着是根据浏览器的可用方式使用 detachEvent 或 removeEventListener 方法将事件从节点上移除。最后是使用 Ext.Array 的 erase 方法将事件从缓存中删除。

2. removeAll

从元素中移除所有事件。

（1）语法

```
Ext.EventManager.removeAll(el);
```

其中，el 为要移除所有事件的元素。

（2）源代码

```
removeAll : function(element){
    var el = element.dom ? element : Ext.get(element),
        cache, events, eventName;

    if (!el) {
        return;
    }
    cache = el.$cache;
    events = cache.events;

    for (eventName in events) {
        if (events.hasOwnProperty(eventName)) {
            this.removeListener(el, eventName);
        }
    }
    cache.events = {};
},
```

从事件缓存中取出元素的事件对象，然后使用 removeListener 方法逐个移除事件。

3. purgeElement

使用递归方式移除元素及其子节点的事件。

（1）语法

```
Ext.EventManager.purgeElement(el[,eventName]);
```

其中，el 为要移除事件的元素；eventName 为可选参数，是事件名称，如果不设置，则移除所有事件，否则只移除指定事件名称的事件。

（2）源代码

```
purgeElement : function(element, eventName) {
    var dom = Ext.getDom(element),
        i = 0, len;

    if(eventName) {
        this.removeListener(dom, eventName);
    }
    else {
        this.removeAll(dom);
    }

    if(dom && dom.childNodes) {
        for(len = element.childNodes.length; i < len; i++) {
            this.purgeElement(element.childNodes[i], eventName);
        }
    }
},
```

如果事件名称存在，使用 removeListener 方法移除事件名称指定的事件，否则使用 removeAll 移除全部事件。

如果元素存在子节点，则遍历所有子节点，递归调用 purgeElement 方法移除事件。

5.3 内部事件

5.3.1 内部事件对象：Ext.util.Event

与 EventObjectImpl 对象一样，内部事件也需要一个对象来记录自己的信息，这个对象就是 Event 对象。它为内部事件提供了事件名称、observable 实例、事件列表等信息，还提供了绑定事件、移除事件、触发事件和清理缓存或延迟事件等方法。要注意的是，Event 对象提供的方法只是框架内部使用的接口，而不是外部接口，因而不要尝试直接使用这些方法。

1. addListener

绑定内部事件。其源代码如下：

```
addListener: function(fn, scope, options) {
    var me = this,
        listener;
        scope = scope || me.observable;

    // 省略 debug 代码

    if (!me.isListening(fn, scope)) {
        listener = me.createListener(fn, scope, options);
        if (me.firing) {
            me.listeners = me.listeners.slice(0);
        }
        me.listeners.push(listener);
    }
},
```

代码先使用 isListening 检查触发函数是否已经绑定，其代码如下：

```
isListening: function(fn, scope) {
    return this.findListener(fn, scope) !== -1;
},
```

转到 findListener 方法，其代码如下：

```
findListener: function(fn, scope) {
    var listeners = this.listeners,
    i = listeners.length,
    listener,
    s;

    while (i--) {
        listener = listeners[i];
        if (listener) {
```

```
                s = listener.scope;
                if (listener.fn == fn && (s == scope || s == this.observable)) {
                    return i;
                }
            }
        }

        return - 1;
    },
```

代码使用 while 循环来遍历监听事件数组内的触发函数，如果触发函数的指向相同，且作用域指向相同或作用域等于 observable 对象的作用域，则返回触发函数在数组中的位置 i，否则返回 –1。

回到 addListener 方法，如果还有没绑定，则使用 createListener 方法绑定事件，其代码如下：

```
createListener: function(fn, scope, o) {
    o = o || {};
    scope = scope || this.observable;
    var listener = {
            fn: fn,
            scope: scope,
            o: o,
            ev: this
        },
        handler = fn;
    if (o.single) {
        handler = createSingle(handler, listener, o, scope);
    }
    if (o.delay) {
        handler = createDelayed(handler, listener, o, scope);
    }
    if (o.buffer) {
        handler = createBuffered(handler, listener, o, scope);
    }

    listener.fireFn = handler;
    return listener;
},
```

代码先建立了一个包含 fn、scope、o 和 ev 的事件监听对象 listener。

如果配置对象的 single 属性为 true，则使用 createSingle 方法封装触发函数，其代码如下：

```
function createSingle(handler, listener, o, scope) {
    return function() {
        listener.ev.removeListener(listener.fn, scope);
        return handler.apply(scope, arguments);
    };
}
```

与浏览器事件的 single 配置项一样，创建只执行一次的事件。在封装的函数内，会先使用 removeListener 方法移除事件，然后再执行触发函数。

如果配置项 delay 为 true，则使用 createDelayed 方法封装触发函数，其代码如下：

```
function createDelayed(handler, listener, o, scope) {
    return function() {
        var task = new Ext.util.DelayedTask();
        if (!listener.tasks) {
            listener.tasks = [];
        }
        listener.tasks.push(task);
        task.delay(o.delay || 10, handler, scope, Ext.Array.toArray(arguments));
    };
}
```

在封装函数内，首先是创建一个 DelayedTask[⊖]对象的实例，然后在监听对象 listener 内添加成员 tasks，并指向一个空数组，然后将 DelayedTask 对象的实例保存到数组中，最后调用 delay 方法执行延时任务。

如果配置对象的 buffer 属性为 true，则使用 createBuffered 方法封装触发函数，其代码如下：

```
function createBuffered(handler, listener, o, scope) {
    listener.task = new Ext.util.DelayedTask();
    return function() {
        listener.task.delay(o.buffer, handler, scope, Ext.Array.
            toArray(arguments));
    };
}
```

因为 buffer 的运行规律是在指定的时间间隔后才执行一次，所以这里不需要像 delay 那样，使用数组保存触发过的事件，只要保存一次就行了。接着将监听对象 listener 的 fireFn 属性指向最终的触发函数并返回监听对象。然后检查当前事件是否在触发状态，如果是，为了不打扰监听循环，使用 slice 方法返回旧数组的元素并建立一个新的数组，并使 listeners 属性指向新的数组，最后在监听数组中加入新的监听对象。

2. removeListener

移除事件。其源代码如下：

```
removeListener: function(fn, scope) {
    var me = this,
        index,
        listener,
        k;
    index = me.findListener(fn, scope);
    if (index != -1) {
        listener = me.listeners[index];
        if (me.firing) {
            me.listeners = me.listeners.slice(0);
        }
        if (listener.task) {
            listener.task.cancel();
            delete listener.task;
        }
        k = listener.tasks && listener.tasks.length;
```

⊖ 相关信息请阅读 5.4.1 节。

```
        if (k) {
            while (k--) {
                listener.tasks[k].cancel();
            }
            delete listener.tasks;
        }
        me.listeners.splice(index, 1);
        return true;
    }

    return false;
},
```

代码首先使用 findListener 方法从监听事件数组中找到事件索引。如果索引大于等于 0，则从监听数组中返回监听对象。如果当前事件在触发状态，则使用 slice 方法复制一个监听数组。接着的移除过程与移除浏览器事件是一样的。最后要做的就是从监听数组中移除监听对象。

3. clearListeners

使用迭代方式停止所有缓存或延时事件。其源代码如下：

```
clearListeners: function() {
    var listeners = this.listeners,
        i = listeners.length;

    while (i--) {
        this.removeListener(listeners[i].fn, listeners[i].scope);
    }
},
```

通过 while 循环逐个使用 removeListener 方法移除监听对象。

4. Fire

触发事件。其源代码如下：

```
fire: function() {
    var me = this,
        listeners = me.listeners,
        count = listeners.length,
        i,
        args,
        listener;

    if (count > 0) {
        me.firing = true;
        for (i = 0; i < count; i++) {
            listener = listeners[i];
            args = arguments.length ? Array.prototype.slice.call(arguments, 0) :
                [];
            if (listener.o) {
                args.push(listener.o);
            }
            if (listener && listener.fireFn.apply(listener.scope || me.observable,
                args) === false) {
```

```
                return (me.firing = false);
            }
        }
    }
    me.firing = false;
    return true;
}
```

如果监听数组有监听事件，则使用循环逐个调用监听对象的 fireFn 所指向的触发函数。

5.3.2 为组件添加事件接口：Ext.util.Observable

在 Ext JS 中，很多组件都有自己特定的事件，但这些事件不是浏览器事件，不需要绑定到 DOM 节点上。因而既不能使用 EventObject 来处理这些事件，也不能使用 EventManager 对象为它们绑定事件。简单来说，这些事件是 Ext JS 内部的虚拟事件，需要一套内部的事件模型机制来处理这些事件，这个事件机制要实现的功能包括为组件添加事件接口、绑定事件、触发事件、处理混合事件及移除事件等。

在 Ext JS 中实现此功能的是 Ovservable 对象，这是一个使用观察者模式的对象。Ext JS 4 对 Ovservable 对象进行了标准化重构，不再像 Ext JS 3 那样直接使用函数原型定义该对象，而是使用 define 方法定义该对象。而且，组件也不再像 Ext JS 3 那样，直接通过从 Ovservable 对象继承来实现事件接口，而是使用 mixins 功能将 Ovservable 对象混合到组件中，从而增加了组件的灵活性。回顾一下 4.4.2 中混入（mixins）处理器的代码，可以知道，组件混合 Ovservable 对象，也就是把 Ovservable 对象原型的成员复制到组件中，这样组件也就具有了 Ovservable 对象的属性和方法。

在混合了 Ovservable 对象的组件的构造函数中，一般都会看到以下代码：

```
me.addEvents(
    'beforeactivate',
     'activate',
     ...
)
```

这些代码的作用是使用 Ovservable 对象的 addEvents 方法为组件创建事件接口。方法 addEvents 的源代码如下：

```
addEvents: function(o) {
    var me = this,
        events = me.events || (me.events = {}),
        arg, args, i;

if (typeof o == 'string') {
    for (args = arguments, i = args.length; i--; ) {
        arg = args[i];
        if (!events[arg]) {
            events[arg] = true;
        }
    }
} else {
```

```
            Ext.applyIf(me.events, o);
        }
    },
```

代码会先检查参数 o 是否是字符串。如果是，则通过 arguments 对象获取所有参数，然后在对象的 events 对象中添加以参数为关键字、值为 true 的成员。否则，则直接调用 applyIf 将其复制到 events 对象。

至此，事件接口就准备好了，下一步要探讨的是如何绑定事件。

注意　因为 events 是在构造函数中创建的，所以只有在实例化组件时才会创建。

5.3.3　为组件绑定事件

为组件绑定事件有两种方式：一种是在创建组件时，在配置对象中配置 listeners 属性；另一种就是在创建组件后，使用 on 方法绑定。

因为 listeners 属性是在配置对象中定义的，因而会在组件初始化时使用该属性。也就是说，使用 listeners 属性绑定的事件，会在构造函数中进行绑定。在混合了 Observable 对象的组件中，在其构造函数中都有如下这行代码：

```
me.mixins.observable.constructor.apply(me, arguments);
```

这行代码的作用就是执行 Observable 对象的构造函数，初始化 Observable 对象。Observable 对象的构造函数如下：

```
    constructor: function(config) {
        var me = this;

        Ext.apply(me, config);
        if (me.listeners) {
            me.on(me.listeners);
            delete me.listeners;
        }
        me.events = me.events || {};

        if (me.bubbleEvents) {
            me.enableBubble(me.bubbleEvents);
        }
    },
```

上述代码首先是复制配置对象到实例，然后检查配置对象的 listeners 属性是否存在，如果存在，则使用 on（addListener 方法的简写）方法绑定事件。要注意，因为这里是使用 apply 方法调用的构造函数，把实例的作用域传递过来了。这里的 me 也就是 this 指针，是指实例而不是 Observable 对象。因而，如果实例化的对象其原型重写了 addListener 方法，这里执行的是覆写后的 addListener 方法，而不是 Observable 对象的方法。例如 AbstractComponent 组件，就重写了 addListener 方法。重写的原因，看看 AbstractComponent

组件的 addListener 方法就知道了，其代码如下：

```
addListener : function(element, listeners, scope, options) {
    var me = this,
        fn,
        option;

    if (Ext.isString(element) && (Ext.isObject(listeners) || options && options.
        element)) {
        if (options.element) {
            fn = listeners;
            listeners = {};
            listeners[element] = fn;
            element = options.element;
            if (scope) {
                listeners.scope = scope;
            }
            for (option in options) {
                if (options.hasOwnProperty(option)) {
                    if (me.eventOptionsRe.test(option)) {
                        listeners[option] = options[option];
                    }
                }
            }
        }

        if (me[element] && me[element].on) {
            me.mon(me[element], listeners);
        } else {
            me.afterRenderEvents = me.afterRenderEvents || {};
            me.afterRenderEvents[element] = listeners;
        }
    }
    return me.mixins.observable.addListener.apply(me, arguments);
},
```

看不明白？再看看 Observable 对象的 addListener 方法，其代码如下：

```
addListener: function(ename, fn, scope, options) {
    var me = this,
        config,
        event;

    if (typeof ename !== 'string') {
        options = ename;
        for (ename in options) {
            if (options.hasOwnProperty(ename)) {
                config = options[ename];
                if (!me.eventOptionsRe.test(ename)) {
                    me.addListener(ename, config.fn || config, config.scope ||
                        options.scope, config.fn ? config : options);
                }
            }
        }
    }
    else {
```

```
            ename = ename.toLowerCase();
            me.events[ename] = me.events[ename] || true;
            event = me.events[ename] || true;
            if (Ext.isBoolean(event)) {
                me.events[ename] = event = new Ext.util.Event(me, ename);
            }
            event.addListener(fn, scope, Ext.isObject(options) ? options : {});
        }
    },
```

比较一下它们之间的参数，会发现 AbstractComponent 对象的第一个参数是 element。也就是说，组件的内部事件也可能是用户操作页面元素触发的，因而需要在为元素绑定事件的同时触发内部事件。例如单击 Grid 的标题栏进行排序，标题栏的元素单击事件是浏览器事件，而触发排序操作是内部事件，所以这里需要一个接口。重写 addListener 方法的目的就在此。

注意，在 AbstractComponent 对象的 addListener 方法中，如果组件还没渲染，则“me[element]”是不存在的，也就不可能绑定事件，因而需要先在 afterRenderEvents 对象中保存事件，等渲染后再绑定事件。

无论是立刻绑定还是渲染后绑定，都会使用 Observable 对象的 mon（addManagedListener 方法的别名）方法为元素绑定事件，该方法通过混合功能已从 Observable 对象复制到了 AbstractComponent 对象，因而可以直接使用“me.mon”调用，其代码如下：

```
addManagedListener : function(item, ename, fn, scope, options) {
    var me = this,
        managedListeners = me.managedListeners = me.managedListeners || [],
        config;
    if (typeof ename !== 'string') {
        options = ename;
        for (ename in options) {
            if (options.hasOwnProperty(ename)) {
                config = options[ename];
                if (!me.eventOptionsRe.test(ename)) {
                    me.addManagedListener(item, ename, config.fn || config,
                        config.scope || options.scope, config.fn ? config :
                        options);
                }
            }
        }
    }
    else {
        managedListeners.push({
            item: item,
            ename: ename,
            fn: fn,
            scope: scope,
            options: options
        });

        item.on(ename, fn, scope, options);
    }
},
```

上述代码先检查参数 ename 是不是字符串。如果不是，则递归调用 addManagedListener 方法。如果是，则在 managedListeners 数组中加入一个监听对象，对象包含了 Element 对象（item）、事件名称（ename）、触发函数（fn）、作用域（scope）和配置对象（options）等信息，最后使用 Element 对象的 on（addListener 方法的别名）方法为元素绑定事件，其代码如下：

```
addListener: function(eventName, handler, scope, opt) {
    var els = this.elements,
            len = els.length,
            i, e;

    for (i = 0; i < len; i++) {
        e = els[i];
        if (e) {
            Ext.EventManager.on(e, eventName, handler, scope || e, opt);
        }
    }
    return this;
},
```

上述代码会通过循环调用 EventManager 的 on 方法逐个为 elements 数组内的元素绑定事件。

在 AbstractComponent 组件的 addListener 方法的最后一句，因为 Observable 对象的 addListener 方法已被重写，所以需要调用 mixins 对象中的 Observable 对象的 addListener 方法绑定事件。

回到 Observable 对象的 addListener 方法，可以看到代码也是先检查 ename 是否是对象，如果是，则递归调用 addListener 方法。如果不是，从 events 对象中取出以 ename 为属性名称的属性，若其值是布尔值，则为其创建一个 Event 对象，然后使用 Event 对象的 addListener 方法绑定事件。也就是说，在实例的 events 对象内会保存事件对应的 Event 对象。

至此，内部事件的绑定就完成了。

5.3.4 内部事件的触发过程

事件绑定后，如何触发呢？如果仅仅只是内部事件，使用 fireEvent 方法触发，其代码如下：

```
fireEvent: function(eventName) {
    var name = eventName.toLowerCase(),
        events = this.events,
        event = events && events[name],
        bubbles = event && event.bubble;

    return this.continueFireEvent(name, Ext.Array.slice(arguments, 1), bubbles);
},
```

代码首先从 events 对象中取出事件，然后调用 continueFireEvent 方法，其代码如下：

```
continueFireEvent: function(eventName, args, bubbles) {
    var target = this,
        queue, event,
        ret = true;
```

```
do {
    if (target.eventsSuspended === true) {
        if ((queue = target.eventQueue)) {
            queue.push([eventName, args, bubbles]);
        }
        return ret;
    } else {
        event = target.events[eventName];
        // Continue bubbling if event exists and it is `true` or the handler
             didn't returns false and it
        // configure to bubble.
        if (event && event != true) {
            if ((ret = event.fire.apply(event, args)) === false) {
                break;
            }
        }
    }
} while (bubbles && (target = target.getBubbleParent()));
return ret;
},
```

如果事件是冒泡的，则会在循环中不断冒泡，直到 getBubbleParent 方法返回值与目标（target）相同为止，其代码如下：

```
getBubbleParent: function(){
    var me = this, parent = me.getBubbleTarget && me.getBubbleTarget();
    if (parent && parent.isObservable) {
        return parent;
    }
    return null;
},
```

代码会调用 getBubbleTarget 方法返回冒泡的目标，该方法是在各组件内定义的，在此不讨论，有兴趣可以自己研究一下。

如果对象存在且支持 Observable 对象，则返回。

在循环内，如果组件的当前事件处于暂停状态，则将事件参数保存到事件队列数组（eventQueue）中并返回。如果处于非暂停状态，则会检查事件是否存在，如果存在，就调用 Event 对象的 fire 方法触发事件。如果 fire 返回 false，则中断循环。

如果是用户操作了浏览器元素触发的内部事件，处理过程也差不多，通过 Grid 的列标题单击事件可以了解其过程，在 Column.js（列标题定义组件）文件中可找到列标题的单击事件（onElClick）：

```
onElClick: function(e, t) {
    var me = this,
        ownerHeaderCt = me.getOwnerHeaderCt();
    if (ownerHeaderCt && !ownerHeaderCt.ddLock) {
        if (me.triggerEl && (e.target === me.triggerEl.dom || t === me.triggerEl.
            dom || e.within(me.triggerEl))) {
            ownerHeaderCt.onHeaderTriggerClick(me, e, t);
        } else if (e.getKey() || (!me.isOnLeftEdge(e) && !me.isOnRightEdge(e))) {
            me.toggleSortState();
            ownerHeaderCt.onHeaderClick(me, e, t);
        }
```

```
        }
    },
```

注意加粗的代码，直接调用了 HeaderContainer（标题容器，\grid\header\container.js）对象的 onHeaderClick 事件，其代码如下：

```
onHeaderClick: function(header, e, t) {
    this.fireEvent("headerclick", this, header, e, t);
},
```

在这里就使用了 Observable 对象的 fireEvent 方法去触发 headerclick 事件。

5.3.5 移除事件

如果组件存在浏览器事件，则移除事件时要同时移除浏览器事件，该操作会使用到 Observable 对象的 removeManagedListener 方法，其代码如下：

```
removeManagedListener : function(item, ename, fn, scope) {
    var me = this,
        options,
        config,
        managedListeners,
        length,
        i;

    if (typeof ename !== 'string') {
        options = ename;
        for (ename in options) {
            if (options.hasOwnProperty(ename)) {
                config = options[ename];
                if (!me.eventOptionsRe.test(ename)) {
                    me.removeManagedListener(item, ename, config.fn || config,
                        config.scope || options.scope);
                }
            }
        }
    }

    managedListeners = me.managedListeners ? me.managedListeners.slice() : [];

    for (i = 0;length = managedListeners.length;i < length; i++) {
        me.removeManagedListenerItem(false, managedListeners[i], item, ename, fn,
            scope);
        }
    }
},
```

代码会先检查 ename 是不是字符串，如果不是，则递归调用 removeManagedListener 方法。然后再遍历 managedListeners 数组中记录的浏览器事件，调用 removeManaged-ListenerItem 方法逐个移除事件，其代码如下：

```
removeManagedListenerItem: function(isClear, managedListener, item, ename, fn,
    scope){
    if (isClear || (managedListener.item === item && managedListener.ename ===
        ename && (!fn || managedListener.fn === fn) && (!scope || managedListener.
```

```
            scope === scope))) {
            managedListener.item.un(managedListener.ename, managedListener.fn,
                managedListener.scope);
            if (!isClear) {
                Ext.Array.remove(this.managedListeners, managedListener);
            }
        }
    },
```

如果事件对象、事件名称、触发函数、作用域都相同，则在数组中移除该项，并调用 Element 对象的 un 方法移除事件。与 EventManager 对象一样，Observable 对象也提供了移除所有浏览器事件的 clearManagedListeners 方法，其代码如下：

```
    clearManagedListeners : function() {
        var managedListeners = this.managedListeners || [],
            i = 0,
            len = managedListeners.length;

        for (; i < len; i++) {
            this.removeManagedListenerItem(true, managedListeners[i]);
        }

        this.managedListeners = [];
    },
```

遍历监听数组中的浏览器事件，调用 removeManagedListenerItem 方法逐个移除事件。要移除内部事件，需要使用 removeListener 方法，其代码如下：

```
    removeListener: function(ename, fn, scope) {
        var me = this,
            config,
            event,
            options;

        if (typeof ename !== 'string') {
            options = ename;
            for (ename in options) {
                if (options.hasOwnProperty(ename)) {
                    config = options[ename];
                    if (!me.eventOptionsRe.test(ename)) {
                        me.r  emoveListener(ename, config.fn || config, config.scope ||
                            options.scope);
                    }
                }
            }
        } else {
            ename = ename.toLowerCase();
            event = me.events[ename];
            if (event.isEvent) {
                event.removeListener(fn, scope);
            }
        }
    },
```

代码仍然会先检查 ename 是否是字符串，如果不是，则递归调用 removeListener 方法。移除过程比较简单，从 events 对象中取出事件，然后使用 event 对象的 removeListener 方法移除就行了。

5.4 特定功能的事件对象

5.4.1 延时任务：Ext.util.DelayedTask

DelayedTask 对象主要用于实现输入缓冲的功能，在指定的时间间隔内，如果用户没有操作触发事件，就会执行实际触发函数。

DelayedTask 对象提供了以下两个方法：

- cancel：取消最后的队列任务。
- delay：取消当前的任务，开始一个新的任务。

下面通过一个示例来学习如何使用 DelayedTask 对象。

（1）功能描述

示例的主要功能是：如果在 1 秒内没有向文本框内输入任何数字，就会根据文本框中的数值把列表（有 50 个数字）中数值小于该输入值的数值过滤掉。

（2）实现代码

使用示例模板创建一个文件名称为 5-3.html 的页面文件。

首先在页面中加入以下样式代码：

```
#numList span{height:20px;width:100px;display:block;}
```

接着加入以下 HTML 代码：

```
<input id="input1" value="" />
<div id="numList"></div>
```

然后要做的就是创建一个包含 50 个随机数字的列表：

```
html="";
for(var i=0;i<50;i++){
   html+="<span>"+Math.floor(Math.random()*10000)+"</span>";
}
var el=Ext.getDom("numList");
el.innerHTML=html;
```

每个数字都加了一个 span 标签，因而可以很方便地使用 childNode 属性获取所有节点，从而获取里面的数字值。

接下来要创建一个 DelayedTask 对象的实例：

```
var task=new Ext.util.DelayedTask(function(){
   var els=Ext.getDom("numList").childNodes;
   var value=parseInt(Ext.getDom("input1").value);
   for(var i=els.length-1;i>=0;i--){
```

```
            var v=parseInt(els[i].innerHTML)
            if(v < value){
                    els[i].style.display="none";
            }else{
                    els[i].style.display="block";
            }
    }
});
```

任务中的函数很简单，遍历 numList 的子节点，在比较大小之后设置其 display 属性是否显示。下一步要为文本框绑定 keypres 事件：

```
Ext.EventManager.on("input1","keypress",function(e,el){
    var key=e.getKey();
    if(key<e.ZREO || key > e.NINE) e.stopEvent();
    task.delay(1000);
})
```

代码首先使用 getKey 方法返回事件中的键盘代码，如果键盘代码不是数字，则停止事件。最后是使用 delay 方法设置任务，其时间间隔为 1 秒。

（3）页面效果

在浏览器中打开页面并在文本框内输入 6000，等待 1 秒将看到如图 5-1 所示的结果。

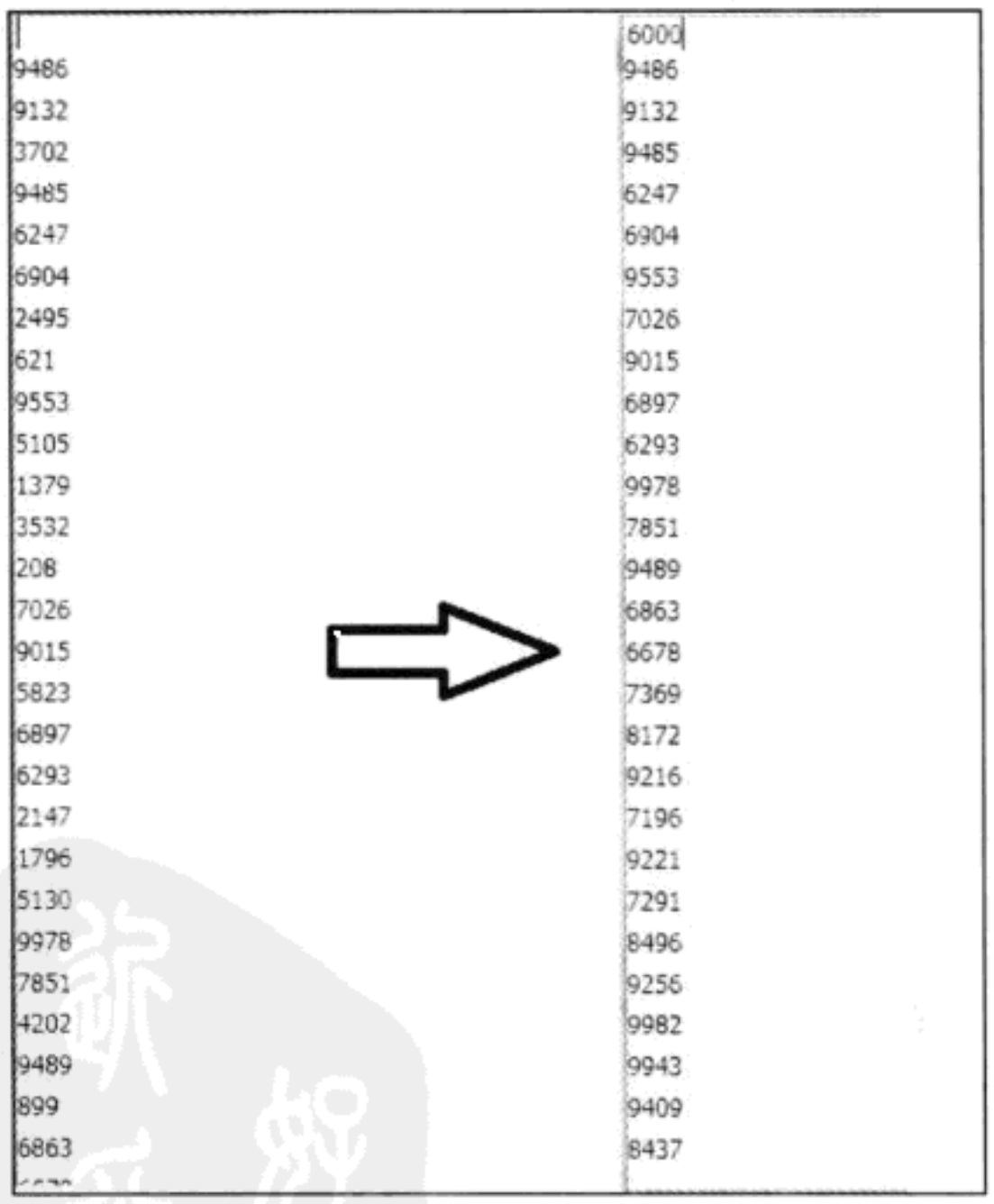


图 5-1　示例的效果

从示例中可以了解到，DelayedTask 非常适合用于用户输入搜索数据时动态显示过滤数据的情况。

5.4.2 一般任务：Ext.util.TaskRunner 与 Ext.TaskManager

TaskRunner 对象的主要作用是执行一些任务，例如定时更新股票信息、定时到服务器上查询是否有最新信息等。简单来说，它就是将 JavaScript 的 setInterval 方法封装起来，以便使用。

TaskManager 对象是 TaskRunner 对象的一个实例，因而可以直接使用 TaskManager 对象执行任务，如果有需要，也可以新建一个 TaskRunner 对象的实例来运行任务。

一个任务其实就是一个配置对象，有以下 6 个参数：

- run：任务触发时执行的函数。
- interval：触发任务的时间间隔，单位是毫秒。
- args：可选值，为传递给触发函数的参数数组。
- scope：可选值，为触发函数的作用域，默认值是任务的配置对象。
- duration：可选值，任务运行时间超过该值指定的时间后，任务会自动停止，其单位为毫秒。
- repeat：可选值，为任务可执行的次数，超过此值指定的次数后，任务会自动停止。

TaskRunner 对象提供了 start（开始任务）、stop（停止任务）和 stopAll（停止所有任务）3 个方法操作任务。

要使用任务很简单，下面通过一个简单倒计时示例来演示如何使用任务。

（1）功能描述

该示例主要是让用户输入一个时间值，单击“开始”按钮后，开始进入倒计时，倒计时结束会弹出一个对话框提示用户时间到。

（2）实现代码

使用示例模板新建一个文件名称为 5-4.html 的页面文件，在页面中加入以下样式代码：

```
#div1{
        width:160px;
        height:160px;
        line-height:80px;
        padding:0;
        font-size:80px;
        text-align:center;
        font-weight:blod;
        display:none;
}
```

然后在页面中各添加一个文本框、按钮和一个隐藏的 div：

```
<div style="padding:20px 0 0 20px;">
   <input id="input1" type="text" value="0"/>
   <input id="stopButton" type="button" value="开始">
   <div id="div1"></div>
</div>
```

接着要做的是用变量指向文本框、按钮和 div 的 DOM 节点，以便操作：

```
var input=Ext.getDom("input1");
```

```
var div=Ext.getDom("div1");
var button=Ext.getDom("stopButton");
```

文本框用于输入倒计时的值，按钮用于开始任务或中途停止任务，div 的作用是显示倒计时的数字。

接着要定义一个任务，任务的触发函数会根据返回的计数器改变 div 的显示，如果计时结束，则恢复文本框的输入、隐藏 div、修改按钮显示，最后显示计时结束提示，其代码如下：

```
var task={
    run:function(count){
        var v=parseInt(input.value);
        var d=v-count;
        if(v>count){
                div.innerHTML=v-count;
        }else{
                input.removeAttribute("readonly");
                div.style.display="none";
                button.value="开始";
                Ext.MessageBox.alert("信息","时间到！")
        }
    },
    interval:1000,
    duration:1000
}
```

代码中 duration 的值会在单击按钮后被修改，这里可以任意设置。

注意　提示信息不能使用 alert 命令，因为 alert 会停止当前代码的执行，本该在触发函数执行完毕后执行的后续代码不能执行，这样就不能顺利移除任务，任务会多执行一次，最后出现两次提示信息。

接着为文本框绑定 keypres 事件，禁止输入非数字字符：

```
Ext.EventManager.on("input1","keypress",function(e,el){
    var key=e.getKey();
    if(key<e.ZREO || key > e.NINE) e.stopEvent();
});
```

最后为按钮绑定单击事件：

```
Ext.EventManager.on("stopButton","click",function(e,el){
    var t=task;
    if(el.value=="停止"){
        el.value="开始";
        input.removeAttribute("readonly");
        div.style.display="none";
        Ext.TaskManager.stop(task);
    }else{
        var v=parseInt(input.value);
        if(v>0){
            t.duration=v*1000;
            input.readOnly="true"
```

```
                    div.innerHTML=v;
                    div.style.display="block";
                    el.value=" 停止 ";
                    Ext.TaskManager.start(task);
                }
            }
        });
```

当按钮值为“停止”时，修改按钮的显示值为“开始”，文本框移除只读属性、隐藏div，然后停止任务。

如果不是按钮，则从文本框获取倒计时值，如果值大于0，则修改duration属性，并设置文本为只读，然后设置显示倒计时值的div的初始值并设置其为可见，修改按钮值为“停止”，最后开始任务。

（3）页面效果

在浏览器中打开页面并在文本框内输入20，单击开始并等待倒计时结束将看到如图5-2所示的结果。

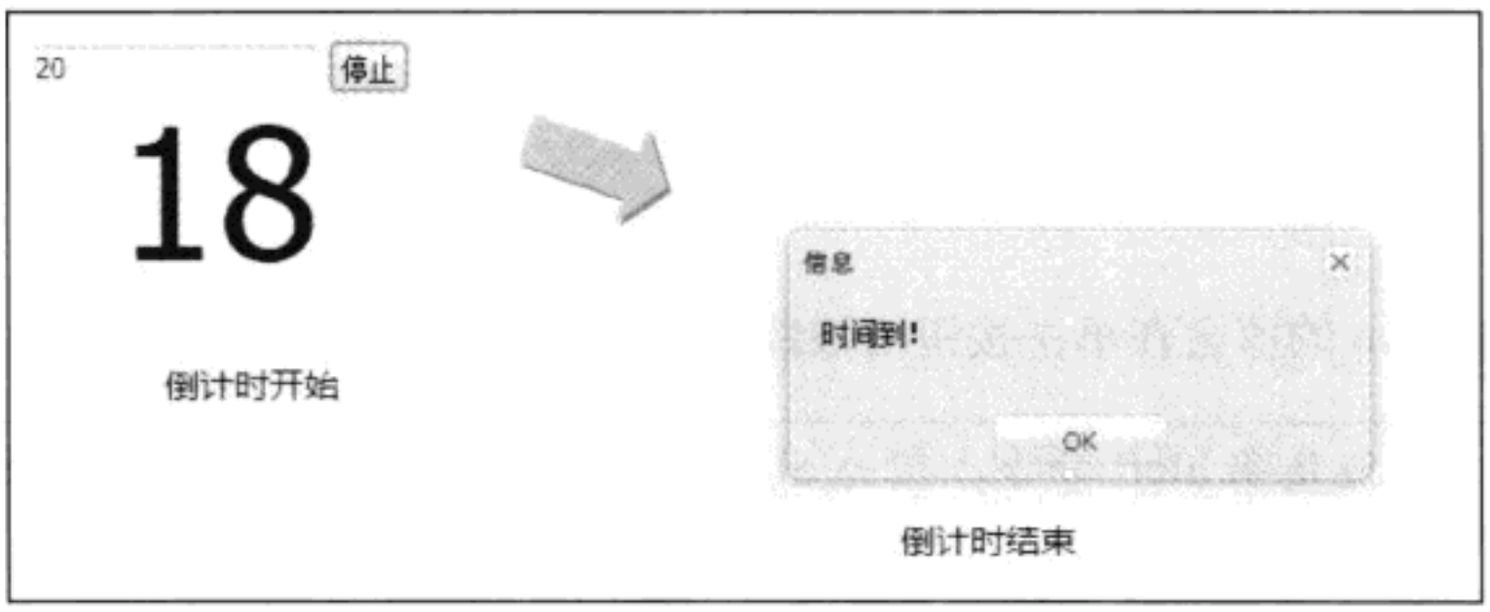


图5-2　倒计时程序页面效果

5.4.3　封装好的单击事件：Ext.util.ClickRepeater

封装单击事件的作用是简化代码的书写以及简单实现一些特殊功能，例如在按钮（button.js）的定义中有以下代码：

```
repeater = Ext.create('Ext.util.ClickRepeater', btn, Ext.isObject(me.repeat) ?
    me.repeat: {});
me.mon(repeater, 'click', me.onRepeatClick, me);
```

在这里就省略了获取元素、为元素绑定事件、将元素的浏览器单击事件接到内部事件等代码，全部让ClickRepeater对象代理了。还有就是设置pressedCls属性，这样就可轻松实现按钮的单击特殊效果。

ClickRepeater对象是从Observable对象继承的对象，因而拥有Observable对象的所有成员。它也定义了自己独有的配置项：

- pressedCls：单击时应用在元素上的样式名称，可实现特殊效果。
- accelerate：布尔值，如果设置为true，interval和delay配置项会被忽略，操作会慢慢

开始并逐渐加速。

- dealy：在该值指定时间后才会再次触发事件。
- interval：指定触发事件之间的间隔，默认值是 20 毫秒。
- listeners：可选值，为对象，可为事件配置对象，可在这里绑定事件。
- preventDefault：布尔值，如果为 true，阻止默认的单击事件。
- stopDefaul：布尔值，如果为 true，停止默认的单击事件。

ClickRepeater 对象的使用很简单，将 5.2 节的第二个示例（5-2.html）中为按钮绑定事件的代码修改为以下代码即可：

```
Ext.create("Ext.util.ClickRepeater","stopButton",{
   pressedCls:'pressed',
   listeners:{
          click:function(e){
                 var el=button;
                 var t=task;
                 if(el.value==" 停止 "){
                        el.value=" 开始 ";
                        input.removeAttribute("readonly");
                        div.style.display="none";
                        Ext.TaskManager.stop(task);
                 }else{
                        var v=parseInt(input.value);
                        if(v>0){
                               t.duration=v*1000;
                               input.readOnly="true"
                               div.innerHTML=v;
                               div.style.display="block";
                               el.value=" 停止 ";
                               Ext.TaskManager.start(task);
                        }
                 }
          }
   }
})
```

加粗代码的作用是在按钮单击时，增加一个类似 Ext JS 按钮的单击效果，使用的就是 Ext JS 按钮的单击样式。注意触发函数参数的变化，因为当前函数的作用域是 ClickRepeater 对象，因而需要增加“var el=button;”语句，以保证 el 为按钮的 DOM 对象。

5.5　键盘事件

5.5.1　为元素绑定键盘事件：Ext.util.KeyMap

KeyMap 对象可以为元素设置一些按键，从而实现一些特殊功能，例如可以在工具栏的按钮上添加向下箭头的按键，让它打开子菜单。

要使用 KeyMap 对象，可以在创建实例时定义好配置对象或配置对象数组，也可以在创

建后使用 addBinding 方法来添加，配置对象主要有以下配置项：

- key：触发事件的键盘代码。可以是单一的键盘代码，也可以是键盘代码的数组。
- shift：布尔值，为 true 时，可实现与 shift 键一起的组合键功能。
- alt：布尔值，为 true 时，可实现与 alt 键一起的组合键功能。
- ctrl：布尔值，为 true 时，可实现与 ctrl 键一起的组合键功能。
- handler（fn）：事件的触发函数，fn 为别名。
- scope：触发函数的作用域。
- defaultEventAction：事件的默认操作。可以设置为 stopEvent、stopPropagation 和 preventDefault。如果不设置，则没有默认操作。

可以使用 disable、enable 和 setDisabled 开启 KeyMap 对象，而方法 isEnabled 可返回 KeyMap 的状态，如果为开启状态，返回 true。

下面通过一个示例演示一下如何绑定键盘事件。

（1）功能描述

页面中有一个表单，当用户在文本框、复选框等控件上按下回车键的时候，焦点将跳到下一个输入框。

（2）实现代码

要实现回车后将焦点切换到下一个输入框的功能，一般的思路是为每个输入框定义一个 keypres 方法，然后判断输入键盘代码是否为回车，如果是，跳转到下一输入框。不过，在输入框比较多的情况下，这样做是比较累人的。如果利用事件的传播机制，只要在表单标记上定义一个事件就可以了。本示例会使用 KeyMap 对象将回车键绑定到表单标记上。

在表单上绑定事件唯一的困难是如何找到当前焦点的元素和下一焦点的元素。解决办法就是将所有 input 标记和 textarea 标记的元素保存到一个数组中，直接在数组中找到当前的焦点元素，那么下一个元素就是要获得焦点的元素。

思路确定后，可以开始工作了，使用模板页创建一个文件名称为 5-5.html 的页面文件，然后在文件中输入以下样式代码：

```
#form1 span{height:30px;width:200px;display:block;}
```

接着设计一个简单的表单：

```
<div style="margin:20px 0 0 20px;">
    <form id="form1">
    <span>姓名：<input name="name" type="text" value=""></span>
    <span>性　别：<input type="radio" name="sex" value="男">男   
        <input type="radio" name="sex" value="女">女</span>
    <span>电子邮件：<input type="text" name="email" value=""></span>
    <span>个人简介：<textarea name="intro"></textarea></span>
    </form>
</div>
```

现在开始写代码，首先要做的是建立一个元素数组：

```
var els=Ext.query("#form1 input,textarea");
```

这个比较简单，接着就是创建 KeyMap 对象了：

```
Ext.create("Ext.util.KeyMap","form1",{
    key:13,
    fn:function(key,e){
            var el=e.target,target=els,ln=target.length;
            for(var i=0;i<ln;i++){
                    if(target[i]==el){
                            if(i+1==ln){
                                    target[0].focus();
                            }else{
                                    target[i+1].focus();
                            }
                    }
            }

    },
    scope:Ext.getDom("form1")
})
```

创建 KeyMap 对象时一定要注意作用域，代码中的作用域是表单标记，因为只有这样，触发函数返回的事件才是元素的事件，否则返回的事件对象将是 KeyMap 事件对象。

触发函数代码不难，从事件中获取触发事件的元素，然后与数组中的元素比较，如果元素相同，则判断索引是否为最后一个，若是，第一个输入框将获得焦点；否则触发事件的元素的下一个元素将获得焦点。

对自上往下排列的输入框而言，这段代码是没有问题的，但是如果表单比较复杂，例如输入框被分了很多列，且每列都在不同的容器中，那么顺序就会变成先跳转完第一列，再跳转第二列的情况，所以这需要根据具体情况做调整。还有一个问题就是没有注意隐藏输入框，只需要在获取元素数组时做一下过滤就行了，不难解决。

（3）页面效果

在浏览器中打开页面并从第一个文本框开始，连续按 3 次回车键，将看到如图 5-3 所示的结果。

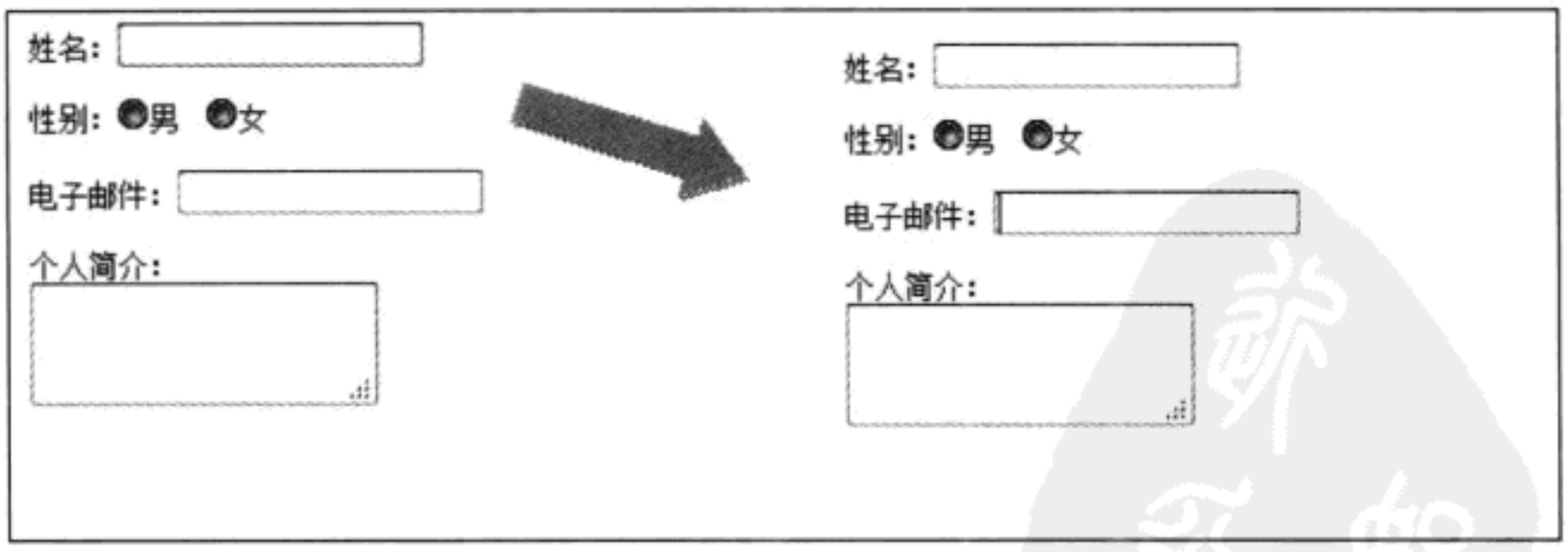


图 5-3　5.4.3 节的示例从第一个文本框开始连续按 3 次回车键后的焦点改变情况

5.5.2 键盘导航：Ext.util.KeyNav

KeyNav 的作用是封装键盘导航事件，从而可以轻松地在页面中设置键盘导航。可用的导航键包括回车键、空格键、4 个箭头键、tab 键、esc 键、PageUp 键、PageDown 键、del 键、backspace 键、home 键和 end 键。

与 KeyMap 对象一样，KeyNav 也可以通过 disabled、enbale 和 setDisabled 三个方法开启或关闭其功能。

下面通过一个示例来演示一下如何实现键盘导航。

（1）功能描述

本示例要实现的功能：在浏览图片时，通过键盘在列表的小图片中移动，然后根据选择刷新大图片。

（2）实现代码

相同的功能用单击事件也可以实现，而且实现起来不难。使用键盘导航与单击事件类似，只是不像单击事件那样是随机的，而是有一定的顺序，通过一个索引变量就可轻松解决。

使用模板创建一个文件名称为 5-6 的页面文件，在样式代码中添加以下代码：

```
#div1{margin:20px 0 0 20px;width:650px;display:block;}
span{width:165px;display:block;padding:1px;float:left;}
 .select{border:2px solid #cdcdcd;}
```

样式 select 的作用是表示当前选择的小图片。

下面创建 HTML 代码：

```
<div tabIndex="1" id="div1">
   <span>
<img width="160" class="smallImg select" src="../images/s1.jpg"/>
<img width="160" class="smallImg" src="../images/s2.jpg"/>
   <img width="160" class="smallImg" src="../images/s3.jpg"/>
   </span>
   <span style="width:478px;">
   <img height="320" id="bigImg" src="../images/1.jpg"/>
   </span>
</div>
```

这里要注意，一定要给 div 加上 tabIndex 属性，不然 div 就不能获取焦点，绑定导航事件会因没有焦点而不能执行。

下面开始编码，首先要做的是使用变量记录小图片的 Element 对象，还要记录下大图片的节点，以便更换图片，最重要的当然是设置好索引值，其代码如下：

```
var els=Ext.select(".smallImg");
var bigImg=Ext.getDom("bigImg");
var index=0;
```

因为键盘的上下箭头只是让索引加 1 或减 1，因而可以使用同一个函数做处理，定义如下：

```
var fn=function(v){
   var ell=els;
```

```
    ell.item(index).removeCls("select")
    index=index+v;
    if(index<0){
            index=2;
    }else if(index>2){
            index=0
    };
    var el=ell.item(index);
    el.radioCls("select")
    var src=el.dom.src;
bigImg.src=src.substr(0,src.length-6)+src.substr(src.length-5);
}
```

代码使用到了 Element 对象的 removeCls 方法和 radioCls 方法为元素添加样式类。要注意当索引值超出数组范围时，需要进行调整。最后是创建 KeyNav 对象：

```
Ext.create("Ext.util.KeyNav","div1",{
    up:function(){
            fn(-1);
    },
    down:function(){
            fn(1);
    },
    scope:this
});
```

如果 div1 没定义 tabIndex 属性，那么必须使用 Ext 的 getDoc 方法返回 document 对象作为绑定事件的元素，否则绑定了事件也无法工作。

（3）页面效果

在浏览器中打开页面后，先用单击小图片或按 tab 键的方法将焦点切换到 div 内，再按下两次向下箭头键后，将看到如图 5-4 所示的结果。

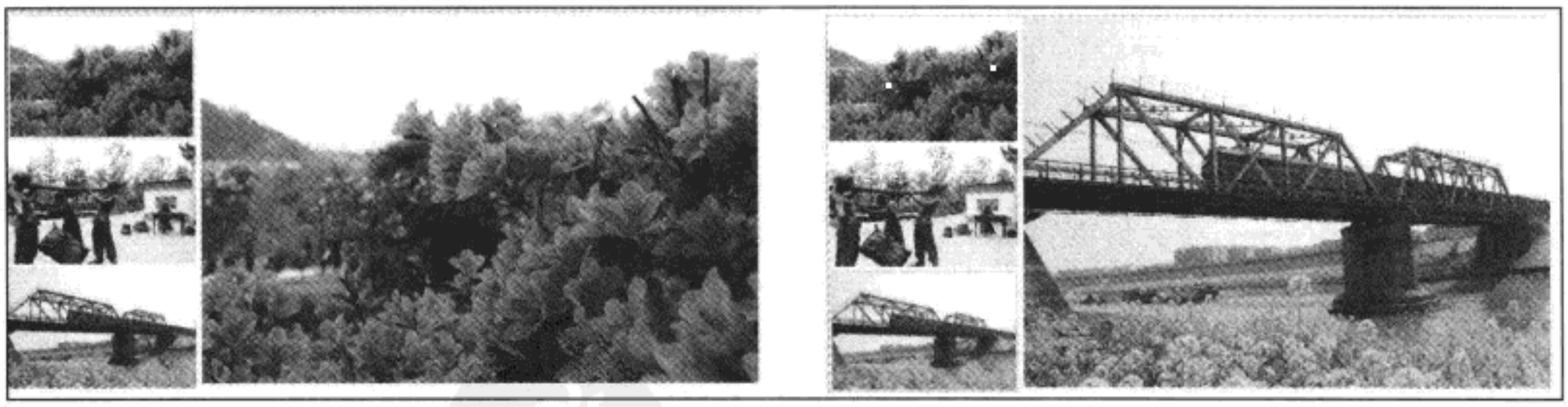


图 5-4　键盘导航事件的示例效果

5.6　综合实例：股票数据的实时更新

（1）功能描述

本示例主要结合前几章的知识，定义一个名称为“My.HKStock”的类，通过动态加载的方法加载类，然后根据页面设置的股票代码去新浪网获取股票信息，接着刷新股票代码列表、

股票的分时图、日 K 线图、周 K 线图或月 K 线图。

（2）实现代码

在开始工作之前，首先要确定如何去新浪网获取股票的即时信息以及如何更新信息。习惯了 Web 应用开发，第一时间会想到用 Ajax 的方式去取数据，不过这里涉及跨域的问题，所以 Ajax 的方式行不通。一般跨域的问题有两种解决办法，使用 iframe 或者添加新的 script 标记。新浪股票返回的数据是脚本，使用 iframe 不好处理，因而动态 script 标记的方法是理想选择。

为了实现动态加载，需要先创建一个与类名同名的脚本文件，也就是需要首先创建 HKstock.js 文件。接着写下类的定义结构：

```
Ext.define("My.HKStock",{
});
```

然后要预定义一些属性。要实现键盘选择股票行和切换曲线图，所以需要记录当前选择行的位置和当前曲线图的位置，还有新浪股票的接口地址、曲线图片地址等信息。为了可以自定义股票刷新事件，任务的刷新时间也可以作为一个定义，在内部需要通过一个数组来记录股票信息。最后定义的是对象的渲染模板。要定义的属性如下：

```
selectIndex:0,
activeIndex:0,
url:'http://hq.sinajs.cn/?_dc=',
imgurl:"http://image.sinajs.cn/newchart/hk_stock",
imgtype:["min","daily","weekly","monthly"],
interval:3000,
elements:[],
renderTpl:[
    '<div style="margin:20px 0 0 20px;">',
    '<div id="op">',
    '<a id="stopButton">停止刷新</a>',
    '<span id="updatetime"></span>',
    '</div>',
    '<div class="clear"></div>',
    '<table id="stocklist">',
    '<tr>',
            '<th>代码</th><th>名称</th><th>最新价</th><th>涨跌额</th><th>涨跌幅</th>',
            '<th>买入</th><th>卖出</th><th>昨收</th><th>今开</th><th>最高</th>',
            '<th>最低</th><th>成交量/万</th><th>成交额/万</th>',
    '</tr>',
    '<tpl for=".">',
    '<tr class="">',
            '<td class="center">{.}</td><td class="center"></td>',
            '<td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td>',
            '<td></td><td></td><td></td><td></td>',
'</tr>',
'</tpl>',
'</table><br/><br/>',
'<div style="width:700px;text-align:center">',
'<div id="tabs">',
        '<ul>',
                '<li class="active">分时图</li>',
                '<li>日K线</li>',
```

```
                '<li>周 K 线</li>',
                '<li>月 K 线</li>',
            '</ul>',
    '</div>',
    '<div id="tabContent">',
            '<img id="img1" width="545" border="0" src=""/>',
    '<div>',
    '</div>'
],
```

因为图片有 4 种，分别在不同的路径下，所以需要使用 imgtype 来记录这些信息。在表格的标题列下使用了一个 for 循环用来输出股票行。虽然用的是标签页，但是内容只有图片，所以只用一个内容页就行了。

接着要定义的是构造函数，即创建类时要做什么。首先是检查配置，尤其是股票代码是否已经设置了；然后是渲染模板，渲染后要把一些需要更新的元素在对象中记录下来，以便于更新，其代码如下：

```
constructor:function(config){
    var me = this,
    config = config || {};
    code=config["code"];
    if(code){
        if(Ext.isString(code)){
            me.code=[code];
        }else if(Ext.isArray(code)){
            me.code=code;
        }else if(Ext.isIterable(code)){
            me.code=Ext.Array.toArray(code);
        }else{
        alert("请输入正确的股票代码! ")
        return;
        }
    if(config["interval"]){
        me.interval=config["interval"];
        }
        me.task={
            run: me.getStock,
            interval: me.interval,
            scope:me
        };
        this.render();
    }else{
        alert("请输入要查看的股票代码! ")
        return;
    }

},
```

检查股票的代码比较啰嗦，还有待改进。HKStock 对象可以使用由股票代码组成的字符作为数据，也可以使用数组等可以迭代的数据，在配置项 code 中进行定义。渲染后就要开始刷新任务，因而要先设置好任务。注意任务作用域要设置为 me，不然在 getStock 中就找

不到对象了。接着要完成的是渲染方法 render：

```
render:function(){
    var me=this,el,childs;
    me.tpl = Ext.create('Ext.XTemplate', me.renderTpl);
    me.el=me.tpl.overwrite(Ext.getBody(),me.code);
    childs=Ext.getDom("stocklist").childNodes[0].childNodes;
    for(var i=1;ln=childs.length,i<ln;i++){
            me.elements.push(childs[i]);
    }
    me.tabs=Ext.select("#tabs li",true);
    me.button=Ext.getDom("stopButton");
    me.updatetime=Ext.getDom("updatetime");
    me.image=Ext.getDom("img1");
    me.rowSelect(0);
    Ext.create("Ext.util.ClickRepeater",me.button,{
            pressedCls:'pressed',
            listeners:{
                    click:me.buttonClick,
                    scope:me
            }
    });
    Ext.fly("stocklist").on("click",me.listClick,me);
    Ext.fly("tabs").on("click",me.tabClick,me);
    me.keynav=Ext.create("Ext.util.KeyNav",Ext.getDoc(),{
            up:me.onKeyUp,
            down:me.onKeyDown,
            left:me.onKeyLeft,
            right:me.onKeyRight,
            scope:me
    });
    me.TaskManager=new Ext.util.TaskRunner();
    me.TaskManager.start(me.task);
},
```

这里使用了 XTemplate[⊖]对象的 overwrite 方法来渲染 HTML 代码。

虽然表格里没有定义 tbody 标记，但是在生成时会自动加上去，因而需要使用两次 childNodes 属性才能获取到 tr 标记的 DOM 对象。第 1 个 tr 对象是标题栏，因而可以忽略掉，所有循环都要从 1 开始计数，将余下 tr 元素保存到 elements 数组就行了。

接着是记录曲线图的标签、停止刷新按钮、刷新时间和图片等元素。方法 rowSelect 的作用是选择表格行，先执行一次以默认选择第一行。

接着是使用 ClickRepeater 对象为“停止刷新”按钮绑定单击事件 buttonClick。

为了实现表格和标签既可以用键盘导航，也可以用鼠标选择，需要为表格和 tab 标签定义单击事件和键盘导航事件。为了简便，单击事件采用了事件传播机制。如果同样使用 ClickRepeater 对象定义，那么就不能知道是哪行或哪个 tab 标签进行了单击，因而这里还是使用标准的浏览器事件定义来绑定单击事件。为了一次就定义好 4 个导航键，将导航事件绑定到页面的 document 对象了。接着创建一个任务，并开始刷新任务。

接着要做的是完成事件函数了，首先是“停止刷新”按钮的，按钮要实现既可以停止，

⊖ 相关信息请阅读 8.1.4 节。

又可以开始功能，因而定义如下：

```
buttonClick:function(){
    var me=this,el=me.button;
    if(el.innerHTML==" 停止刷新 "){
            el.innerHTML=" 开始刷新 ";
            me.TaskManager.stop(me.task);
    }else{
            el.innerHTML=" 停止刷新 ";
            me.TaskManager.start(me.task);
    }
},
```

接着是表格的单击事件 listClick：

```
listClick:function(e,el){
    var me=this,t=me.elements;
    for(var i=t.length-1;i>=0;i--){
            if(el.parentNode==t[i]){
                    me.rowSelect(i-me.selectIndex);
            }
    }
},
```

这里要注意，如果表格内除了表格元素 table，tr 或 td 等，没有其他的 HTML 元素，那么一般触发事件为元素的 td，因而需要使用 ParentNode 属性返回其中元素（tr 元素），再将其与 element 数值中的元素作对比，如果是相同元素，说明需要选择该行。然后根据当前元素在数组中的位置（i），减去当前选择行，就是要移动的行数了。

接着完成标签的单击事件 tabClick：

```
tabClick:function(e,el){
    var me=this,t=me.tabs.elements;
    for(var i=t.length-1;i>=0;i--){
            if(el==t[i].dom){
                    me.tabSelect(i-me.activeIndex);
            }
    }
},
```

原理和表格的单击事件一样，不过 tabs 保存的就是响应事件的元素，所以直接比较就行了。不过 tabs 数组是使用 select ㊀方法选择出来的数组，是 CompositeElement ㊁对象，因而需要与其 dom 属性比较。

接着要完成的是任务触发函数 getStock 了，在这里要创建一个 script 标记，并将其增加到页面，其代码如下：

```
getStock:function(){
    var me=this,
            oldscript=Ext.get("dynscript");
    if(oldscript) oldscript.remove();
    var script = document.createElement('script');
```

㊀ 相关信息请阅读 6.5.1 节。

㊁ 相关信息请阅读 6.5.2 节。

```
script.setAttribute("type",'text/javascript');
script.setAttribute("src",me.url+(new Date().getTime())+"&list=hk"+me.code.
    join(",hk"));
script.setAttribute("id","dynscript");
var cb=Ext.Function.bind(me.callback,me);
script.onload=cb;
script.onreadystatechange = function() {
        if (this.readyState == 'complete') {
                cb();
        }
    }
    Ext.getHead().appendChild(script);
},
```

为了不让 script 标记泛滥（默认 3 秒 1 个，1 分钟就 20 了，虽然还不知道会有怎样的后果），在新建之前，要把旧的删除（如果存在的话）。为了便于删除，会为标记添加一个 id 属性。为什么不能使用修改 src 方式呢？对于图片是可以的，但对于脚本有时候会罢工，这和浏览器有关，为了保险起见，还是新增比较安全。直接使用数组合成地址就行了，不难。最重要的一步是为标记添加回调函数 callback，当脚本下载完成后，要通过回调函数获取数据并更新表格和曲线图。因为浏览器对脚本下载完成的事件不同，因而需要绑定两个方法以保证能执行回调函数。为了保证作用域，需要使用 Ext.Function 对象 bind 方法绑定回调函数。这里应该可以改进一下，没必要每次都进行绑定，读者有兴趣可以自己研究一下。

接着完成回调函数 callback：

```
callback:function(){
    var me=this;
    for(var i=me.code.length-1;i>=0;i--){
            var code=me.code[i];
            var v=window["hq_str_hk"+code];
            if(v){
                    vv=v.split(",");
                    //var el=me.elements[i].query("td");
                    var el=me.elements[i].childNodes;
                    el[1].innerHTML=vv[1];
                    el[2].innerHTML=vv[6];
                    el[3].innerHTML=vv[7];
                    el[4].innerHTML=vv[8]+"%";
                    el[5].innerHTML=vv[9];
                    el[6].innerHTML=vv[10];
                    el[7].innerHTML=vv[3];
                    el[8].innerHTML=vv[4];
                    el[9].innerHTML=vv[4];
                    el[10].innerHTML=vv[5];
                    el[11].innerHTML=vv[12]/10000;
                    el[12].innerHTML=vv[11]/10000;
                    var cls= vv[7].substr(0,1)=="-" ? "red" :
                            (parseFloat(vv[7])>0? "green" : "");
                    me.elements[i].style.color=cls;
            }
    }
me.updatetime.innerHTML="刷新时间："+Ext.Date.format(new Date(), 'Y-m-d H:i:s');
me.updateImage();
```

```
},
```

新浪股票返回的数据格式如下：

```
var hq_str_hk00941="CHINA MOBILE, 中国移动 ,69.100,69.000,69.300,68.850,69.250,0.250
    ,0.362,69.200,69.250,1029756963,14921069,9.851,3.747,84.800,68.500,2011/05/20,
    16:00";
```

脚本下载后，就会再执行，因而需要将 window 对象中数据从变量中取出来。取出来后就是要数据更新。

因表格的列和返回的数据列的规律对不上号，只能一个个写了。变量 cls 的作用是改变行的文字颜色，如果股票是跌的，字体为红色；是涨的用绿色；没有变化就用黑色。

最后要做的是更新刷新事件。因为图片更新其他方法也要使用到，所以独立成一个方法，其代码如下：

```
updateImage:function(){
    var me=this;
    me.image.src=[me.imgurl,me.imgtype[me.activeIndex],me.code[me.selectIndex]+".
        gif?_dc="+Ext.Date.format(new Date(), 'Y-m-d H:i:s')].join("/");

},
```

用数组生成路径的方法不错，够简捷，值得推荐。

接着要完成的是 4 个导航按钮的事件，这个简单，其代码如下：

```
onKeyUp:function(){
    var me=this;
    me.rowSelect(-1);
},

onKeyDown:function(){
    var me=this;
    me.rowSelect(1);
},

onKeyLeft:function(){
    var me=this;
    me.tabSelect(-1);
},

onKeyRight:function(){
    var me=this;
    me.tabSelect(1);
},
```

接着要完成行选择方法 rowSelect 了：

```
rowSelect:function(v){
    var me=this;
    me.elements[me.selectIndex].className="";
    me.selectIndex=me.selectIndex+v;
    if(me.selectIndex<0){
            me.selectIndex=me.code.length-1;
```

```
        }else if(me.selectIndex>(me.code.length-1)){
                me.selectIndex=0
        };
        me.elements[me.selectIndex].className="selected";
        me.updateImage();
    },
```

代码没什么特别的地方，就是将当前选择行的样式类去掉，然后计算出新的选择行，为新的选择行添加选择样式。别忘记更新图片。

最后完成的是标签选择方法 tabSelect 了：

```
tabSelect:function(v){
    var me=this,tabs=me.tabs.elements;
    tabs[me.activeIndex].removeCls("active");
    me.activeIndex=me.activeIndex+v;
    if(me.activeIndex<0){
            me.activeIndex=3;
    }else if(me.activeIndex>3){
            me.activeIndex=0
    };
    tabs[me.activeIndex].radioCls("active")
    me.updateImage();
}
```

这与 rowSelect 方法大同小异，就不多说了。

这样，HKStock 对象的定义就完成了，现在要做的是怎么在页面中实现动态加载并加载对象了。

使用模板页创建一个名称为 5-7.html 的页面文件，然后加入以下样式代码：

```
#op{
    width:1000px;
    height:20px;
    line-height:20px;
    display:block
}
#op
    a{float:left;color:blue;cursor:pointer;text-decoration:underline}
#op span{float:right;}
.clear{clear:both;line-height:0;height:0;overflow:hidden;}
tr{border-bottom:1px solid #cdcdcd;}
th{background:#CDCDCD;text-align:center;}
td{width:80px;text-align:right;}
td.center{text-align:center;}
.selected{background:#DDFFFF}
#tabs{
    width:600px;
    height:20px;
    font-size:14px;
    line-height:20px;
    display:block;
    margin:auto;
    background:#D9D9D9;
    border-bottom:1px solid #e0e0e0;
```

```
}
#tabs ul{list-style: none}
#tabs li{float:left;width:80px;text-align:center;display:block;cursor:pointer;}
#tabs .active{
   border-left:1px solid #e0e0e0;
   border-right:1px solid #e0e0e0;
   border-top:1px solid #e0e0e0;
   background:white;
}
#tabContent{
   width:600px;
   margin:auto;
   border-left:1px solid #e0e0e0;
   border-right:1px solid #e0e0e0;
   border-bottom:1px solid #e0e0e0;
   height:330px;
   display:block;
   text-align:center;
}
```

样式有点多，要做的事情也比较多，例如，选择行的样式、tab 标签的样式等。但这个不是重点，就不多说了。

没有 HTML 代码，因为 HKStock 对象可以做这事，它会把 HTML 代码渲染到 body 上。

要实现动态加载，记得要在 onReady 方法前做 3 件事：开启动态加载的依赖加载功能、定义类的加载路径以及使用 require 方法加载类。

接着的事情就简单了，使用 create 方法创建 HKStock 对象就行了，完成的代码如下：

```
Ext.Loader.setConfig({enabled: true});
Ext.Loader.setPath({
    "My":"",
});
Ext.require([
   "My.HKStock"
]);
Ext.onReady(function(){
   if(Ext.BLANK_IMAGE_URL.substr(0,4)!="data"){
          Ext.BLANK_IMAGE_URL="./images/s.gif";
   }
   // 在此添加 Ext JS 代码
   Ext.create("My.HKStock",{
          code:["00941","00762","00728","01288","01398"]
   });
});
```

这里要注意，“My”的路径不能设置为根目录，因为当前的目录不一定是根目录，应该把根目录设置为空，这样 Loader 对象会在与页面文件相同的目录中加载脚本。

至此，示例就完成了，可以在浏览器查看效果了。

（3）页面效果

在浏览器中打开页面后，先按下箭头，再按右箭头将看到如图 5-5 所示的效果。

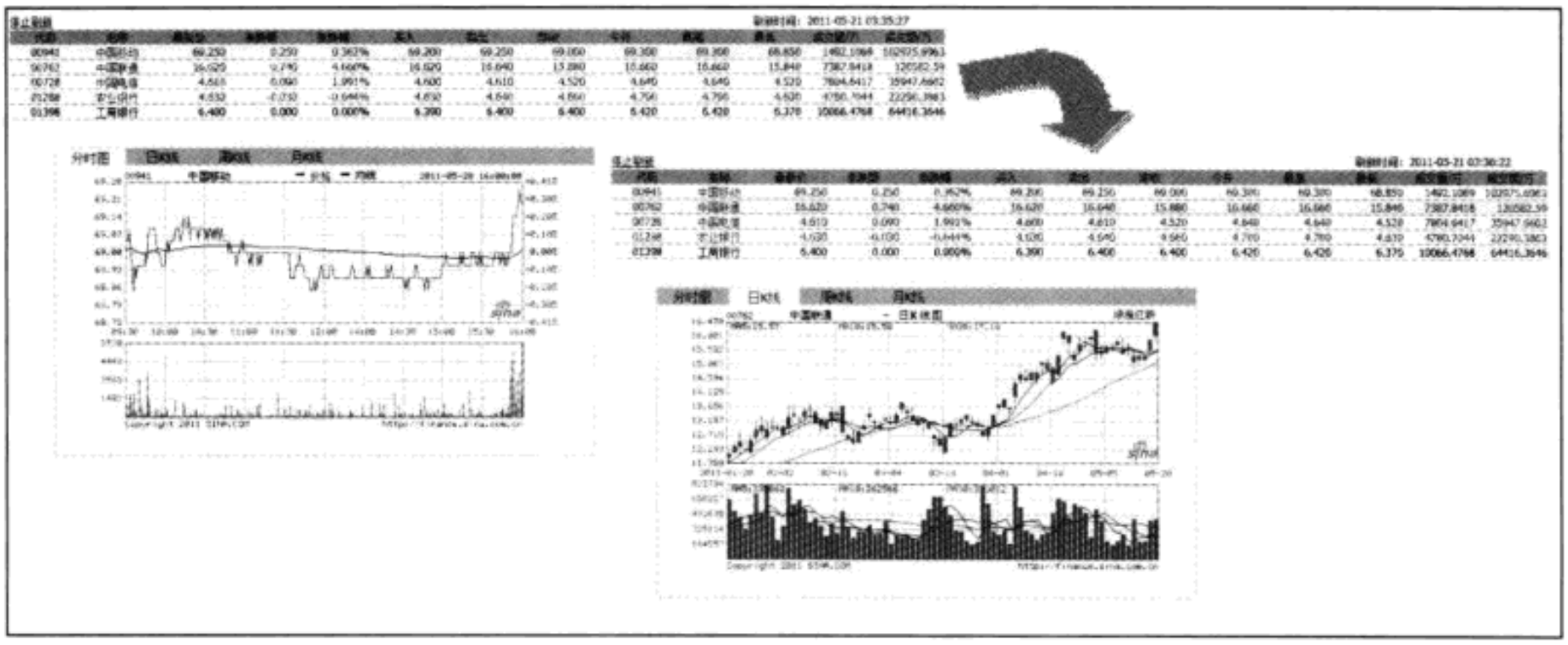


图 5-5　按下箭头和右箭头的效果

5.7　本章小结

本章主要讲述了 Ext JS 的事件模型及其应用，掌握这些知识，在使用 Ext JS 时会更得心应手，因为一切皆事件。现代的编程模式基本都是基于事件驱动开发的，因而现在学习编程的，基本都离不开事件。

很多读者可能都会认为，只要了解组件提供了什么事件就足够了，它怎么运作的就不必在意了。但要明白的是，事出皆有因，尤其是事件问题。例如，笔者写该章的时候，就有人问为什么 stopEvent 方法不起作用了，如果明白了事件的内部原理，要查找这样的错误，就简单得多了。

第 6 章　选择器与 DOM 操作

文档对象模型（DOM）是 JavaScript 与页面元素的沟通桥梁，但标准的 DOM 操作语句书写比较麻烦，而且浏览器之间还存在差异。作为跨平台的框架，不仅必须简化书写，还要兼顾各浏览器之间的差异，所以会根据框架自身要求重新封装选择器和 DOM 操作。本章将讲述 Ext JS 中的选择器和 DOM 操作，大家可以从中了解 Ext JS 的选择器和 DOM 操作与其他框架会有什么不同，又会有什么特点。

6.1　Ext JS 的选择器：Ext.DomQuery

6.1.1　选择器的作用

脚本如果想修改页面中元素的属性和行为，首先要做的就是定位这些需要修改的元素，然后对其进行修改。当然，可以通过 id 使用 getElementById 方法定位到某个元素，但是，如果是一组菜单链接或者是表格中的行呢？总不能为页面中的每个元素都加一个 id 吧？所以这就需要使用到选择器了。

选择器的作用就是通过元素的标签名、属性名、CSS 属性值对页面元素进行快速、准确的定位及选择。

6.1.2　使用 Ext.query 选择页面元素

1. 基本语法

```
Ext.query(path[,root,type]);
```

其中：

- path：查询使用的选择符或 xpath。
- root ：查询开始的节点或节点 id。如果不设置，默认为 document 对象。为了提高查询速度，建议不要从 document 对象开始查询。
- type ：查询类型。值可以为 select 和 simple。当值为 simple 时，使用 id 或标签名属性值等做简单查询。

当开始节点已通过 Ext.get 方法获得时，也可以使用以下语法：

```
var el = Ext.get(rootID); //rootID 为节点 id
el.quert(path);
```

2. 示例

（1）功能描述

页面中包含常见的文章列表，每篇文章都带有一个值为文章 id 的复选框，用来选择文章，而在标题栏有一个“全选”复选框，用来执行全部选择或全部取消选择文章的操作，这就需要使用选择器将文章列表中所有的复选框选择出来，然后将它们的值设置为“全选”复选框的 checked 值。

（2）实现代码

使用模板新建一个名称为 6-1.html 的页面文件，接着把以下样式代码加入到样式定义里：

```
body{padding:30px 0 0 30px;}
h1{font-size:16px;font-weight:bold;line-height:40px;}
table{
    font-size:14px;
    line-height:30px;
}
th{text-align:center;}
.check{text-align:center;}
.title{padding-left:10px;}
```

接着加入 HTML 代码，就是一个表格，代码如下：

```
<h1>示例 6-1 使用 Ext.query 选择页面元素 </h1>
<table cellpadding="2" cellspacing="2" border="1">
    <tr>
            <th width="40"><input type="checkbox" id="selectAll" name="selectAll"
            value=" 全选 "></th>
        <th width="200"> 文章标题 </th>
</tr>
<tr>
        <td class="check"><input type="checkbox" name="articleId" value="1"></td>
        <td class="title"> 文章标题 1</td>
</tr>
<tr>
        <td class="check"><input type="checkbox" name="articleId" value="2"></td>
        <td class="title"> 文章标题 2</td>
</tr>
<tr>
        <td class="check"><input type="checkbox" name="articleId" value="3"></td>
        <td class="title"> 文章标题 3</td>
</tr>
<tr>
        <td class="check"><input type="checkbox" name="articleId" value="4"></td>
        <td class="title"> 文章标题 4</td>
</tr>
<tr>
        <td class="check"><input type="checkbox" name="articleId" value="5"></td>
        <td class="title"> 文章标题 5</td>
</tr>
<tr>
        <td class="check"><input type="checkbox" name="articleId" value="6"></td>
        <td class="title"> 文章标题 6</td>
```

```
    </tr>
</table>
```

最后加入脚本：

```
Ext.fly("selectAll").on("click",function(e,el){
    var els=Ext.query("input[name=articleId]");
    for(var i=0;ln=els.length,i<ln;i++){
            els[i].checked=el.checked;
    }
})
```

代码为“全选”复选框绑定了一个单击事件。当“全选”复选框单击事件触发时，使用 Ext.query 获取所有 name 属性值为 aritcleId 的复选框。因为 Ext.query 返回的是一个由 HTMLElement 对象组成的数组，所以需要通过循环，根据“全选”复选框的选择情况设置数组内 HTMLElement 对象的 checked 属性。

（3）页面效果

在浏览器打开页面将看到如图 5-1 所示的结果。单击标题栏的“全选”复选框可改变文章的选择状态。

示例6-1 使用Ext.query选择页面元素

	文章标题
	文章标题1
	文章标题2
	文章标题3
	文章标题4
	文章标题5
	文章标题6

图 6-1　使用 Ext.query 选择页面元素的运行结果

3. 源代码分析

在 query.js（在目录 \src\dom 下）文件中可找到 Ext.query 的源代码：

```
Ext.query = Ext.DomQuery.select;
```

也就是说 Ext.query 是 DomQuery 对象 select 方法的简写，其代码如下：

```
select : document.querySelectorAll ? function(path, root, type) {
    root = root || document;
    if (!Ext.DomQuery.isXml(root)) {
    try {
        var cs = root.querySelectorAll(path);
        return Ext.Array.toArray(cs);
    }
    catch (ex) {}
    }
    return Ext.DomQuery.jsSelect.call(this, path, root, type);
} : function(path, root, type) {
    return Ext.DomQuery.jsSelect.call(this, path, root, type);
},
```

代码首先判断浏览器是否支持 W3 的原生选择器 querySelectorAll。如果支持，则执行第一个函数；否则，执行第二个函数。

在第一个函数中，首先指定 root 的指向，默认是指向 document 对象。接着通过 isXML 方法判断 root 是否为 XML 文档对象。isXML 方法的代码如下：

```
isXml: function(el) {
    var docEl = (el ? el.ownerDocument || el : 0).documentElement;
    return docEl ? docEl.nodeName !== "HTML" : false;
},
```

isXML 代码中，简单来说就是通过 el 是否存在根元素（owerDocument）、根节点（document-Element）以及其节点名称是否为“HTML”来判断 el 是否为 XML 文档对象的。

如果 root 不是 XML 文档对象，则使用原生的 querySelectorAll 查询方法选择元素，然后将查询结果转换为数组返回。

如果 root 是 XML 文档对象或者 querySelectorAll 执行失败，则使用 Ext JS 定义的 jsSelect 方法选择元素。这里使用了 call 方法调用，目的是保持 this 仍然是当前执行上下文的 this 指向。

如果原生的 querySelectorAll 不存在，也使用 jsSelect 方法选择元素，其代码如下：

```
jsSelect: function(path, root, type){
    root = root || document;

    if(typeof root == "string"){
        root = document.getElementById(root);
    }
    var paths = path.split(","),
        results = [];

    for(var i = 0, len = paths.length; i < len; i++){
        var subPath = paths[i].replace(trimRe, "");
        if(!cache[subPath]){
            cache[subPath] = Ext.DomQuery.compile(subPath);
            if(!cache[subPath]){
                    //省略抛出异常代码
            }
        }
        var result = cache[subPath](root);
        if(result && result != document){
            results = results.concat(result);
        }
    }
    if(paths.length > 1){
        return nodup(results);
    }
    return results;
},
```

代码首先初始化 root 的指向，如果 root 为 undefined 或空，则为 document 对象；如果 root 为字符串，则使用 getElementById 方法获取 HTMLElement 对象。

接着将参数 path 根据逗号分拆成数组，并将变量 paths 指向该数组。

接着通过循环遍历数组内的字符串查询元素。

循环中，首先将字符串取出并通过正则表达式去掉多余的空字符串。然后在 DomQuery 的缓存（cache，初始化时为空对象）中查询对象是否已查询过。如果没有，则使用 compile 方法生成一个匿名函数，并将 cache 中以 subpath 为名称的属性指向该匿名函数。接着执行该匿名函数，并将 result 指向执行结果。

下一步判断 result 是否有结果且不是 document 对象。条件符合则将结果与 results 数组合并。

如果不止查询一次，则需要执行 nodup 方法去除重复的元素再返回。

只执行了一次查询，则直接返回 results。

下面看看 compile 方法是如何生成匿名函数的，其代码如下：

```
compile : function(path, type){
    type = type || "select";

    var fn = ["var f = function(root){\n var mode; ++batch; var n = root ||
        document;\n"],
        mode,
        lastPath,
        matchers = Ext.DomQuery.matchers,
        matchersLn = matchers.length,
        modeMatch,
        lmode = path.match(modeRe);

if(lmode && lmode[1]){
    fn[fn.length] = 'mode="'+lmode[1].replace(trimRe, "")+'";';
    path = path.replace(lmode[1], "");
}

while(path.substr(0, 1)=="/"){
    path = path.substr(1);
}

while(path && lastPath != path){
    lastPath = path;
    var tokenMatch = path.match(tagTokenRe);
    if(type == "select"){
        if(tokenMatch){
            if(tokenMatch[1] == "#"){
                fn[fn.length] = 'n = quickId(n, mode, root, "'+tokenMatch
                    [2]+'");';
            }else{
                fn[fn.length] = 'n = getNodes(n, mode, "'+tokenMatch[2]+'");';
            }
            path = path.replace(tokenMatch[0], "");
        }else if(path.substr(0, 1) != '@'){
            fn[fn.length] = 'n = getNodes(n, mode, "*");';
        }
    }else{
        if(tokenMatch){
            if(tokenMatch[1] == "#"){
                fn[fn.length] = 'n = byId(n, "'+tokenMatch[2]+'");';
            }else{
                fn[fn.length] = 'n = byTag(n, "'+tokenMatch[2]+'");';
            }
            path = path.replace(tokenMatch[0], "");
        }
    }
}
while(!(modeMatch = path.match(modeRe))){
    var matched = false;
    for(var j = 0; j < matchersLn; j++){
        var t = matchers[j];
        var m = path.match(t.re);
        if(m){
```

```
                fn[fn.length] = t.select.replace(tplRe, function(x, i){
                    return m[i];
                });
                path = path.replace(m[0], "");
                matched = true;
                break;
            }
        }

        ...// 省略部分 debug 信息
    }
        if(modeMatch[1]){
            fn[fn.length] = 'mode="'+modeMatch[1].replace(trimRe, "")+'";';
            path = path.replace(modeMatch[1], "");
        }
    }
    fn[fn.length] = "return nodup(n);\n}";

    eval(fn.join(""));
    return f;
},
```

代码首先定义了查询类型（type），默认是 select。接着通过一个数组 fn 定义了函数体的开始部分。

接着判断 path 是否包含“/”、“>”、“+”或“~”等符号，如果有，则在数组 fn 内添加：

```
mode='/';  //“/”也可以是 >、+、~ 等字符
```

然后将 path 里的符号替换掉。

接着将 path 里前面的“/”符号全部去掉。

接着使用 while 循环拆分 path。这里首先使用正则表达式 tagTokenRe 拆解 path，将结果保存到 tokenMatch 中。接着会根据 type 的值和拆分结果在数组 fn 内添加不同的代码，这里通过示例就比较好理解了。

假设 path 值是“#divId input”，则拆分后 tokenMatch 的值为：

```
["#divId", "#", "divId"]
```

当 type 的值是 select 时，会添加以下语句：

```
n = quickId(n, mode, root,'divId');
```

当 type 的值是 simple 时，会添加以下语句：

```
n = byId(n, 'divId');
```

假设 path 的值是“input[name=articleId]”，则拆分后的 tokenMatch 的值为：

```
["input", undefined, "input"]
```

当 type 的值是 select 时，会添加以下语句：

```
n = getNodes(n, mode,"input");
```

当 type 的值是 simple 时，会添加以下语句：

```
n = byTag(n, "input");
```

在以上的几种情况，path 的值会使用替换掉拆分出来的部分，例如值是“#divId input”时，替换后为“input”。

假设 path 的值是“@input[name=articleId]”，拆分后的 tokenMatch 的值为 null。

那么当 type 的值是 select 时，会添加：

```
n = getNodes(n, mode, "*");
```

当 type 的值是 simple 时，不做任何处理，程序继续执行。

接着继续使用 modeRe 进行拆分，如果拆分不成功，则进入 while 循环拆分余下部分。余下部分的拆分需要使用 DomQuery 对象的 matchers 数组，数组中保存了 5 种类型的查询方式，其代码如下：

```
matchers : [{
        re: /^\.([\w-]+)/,
        select: 'n = byClassName(n, " {1} ");'
    }, {
        re: /^\:([\w-]+)(?:\(((?:[^\s>\/]*|.*?))\))?/,
        select: 'n = byPseudo(n, "{1}", "{2}");'
    },{
        re: /^(?:([\[\{])(?:@)?([\w-]+)\s?(?:(=|.=)\s?['"]?(.*?)["']?)?[\]\}])/,
        select: 'n = byAttribute(n, "{2}", "{4}", "{3}", "{1}");'
    }, {
        re: /^#([\w-]+)/,
        select: 'n = byId(n, "{1}");'
    },{
        re: /^@([\w-]+)/,
        select: 'return {firstChild:{nodeValue:attrValue(n, "{1}")}};'
    }
],
```

从代码可以看到，数组内的 re 关键字的作用是利用其定义的正则表达式对 path 进行验证和拆分工作，而 select 关键字的作用是根据 path 的值，为将要创建的匿名函数中添加的语句。表 6-1 详细说明了这 5 种类型查询方式及其执行 match 方法后的返回值。

拆分完成后，判断是否包含 /、>、+ 或 ~4 种符号，如果有，则与代码开始部分一样，加入一个语句。

最后为匿名函数加入最后两行：

```
return nodup(n);
}
```

然后通过数组 join 方法将数组转换为字符串，再使用 eval 将字符串转换为可执行的函数，然后将函数 f 返回。

例如，示例的查询会生成以下代码：

表 6-1 查询方式及其说明

索引	执行的查询	说明
0	byClassName	根据元素的 class 值进行选择 例如：.check 返回：n = byClassName(n, " check "); 根据伪对象进行选择 例如：:first 返回：n = byPseudo(n, "first", "undefined"); 例如：:nth-child(evan): 返回：n = byPseudo(n, "nth-child", "evan");
1	byPseudo	
2	byAttribute	根据属性值进行选择，“{”符合代码 CSS 属性值，“[”符合代表元素的属性值 例如：:{display=none} 返回：n = byAttribute(n, "display", "none", "=", "{"); 例如：[name=articleId] 返回：n = byAttribute(n, "name", "articleId", "=", "[");
3	byId	根据 id 属性进行选择 例如：#selectAll 返回：n = byId(n, "selectAll");
4	返回一个对象	例如：@name 返回：return {firstChild:{nodeValue:attrValue(n, "name")}};

```
var f = function (root) {
    var mode;
    ++batch;
    var n = root || document;
    n = getNodes(n, mode, "input");
    n = byAttribute(n, "name", "articleId", "=", "[");
    return nodup(n);
};
```

从以上的分析可以看到，查询基本是分两步走，第一步是使用 quickId、byId、byTag 或 getNodes 等方法查询获取元素，然后使用 byAttribute、byClassName、byPseudo 或 byId 等方式过滤元素。

在 DomQuery 对象中有 4 个排除重复元素的方法：quickDiff、quickDiffIEXml、nodup 和 nodupIEXml。其中 quickDiffIEXml 和 nodupIEXml 是基于 IE 的。

nodup 和 nodupIEXml 的工作原理就是遍历结果数组中的元素，检验其 _nodup 属性是否与当前计数器（var d=++key）的值相同，如果不相同，则将计数器的值赋给 _nodup 属性，这是关键。这样，当数组中的元素再次出现该元素的时候，其 _nodup 属性值与计数器的值相同，这就说明了该元素出现了重复，需要剔除。剔除方法就是构建一个新的数组，保存下重复元素出现之前的元素，然后判断余下的元素是否与计数器值相同，如果不相同则保留，其实现代码如下。

```
r = [];
for(var j = 0; j < i; j++){
    r[++ri] = cs[j];
}
for(j = i+1; cj = cs[j]; j++){
    if(cj._nodup != d){
        cj._nodup = d;
        r[++ri] = cj;
    }
}
return r;
```

quickDiff 和 quickDiffIEXml 方法的工作原理与 nodup 一样。只不过它们检验的是 _qdiff 属性，且是对两个数组进行比较。它会先将第一数组的 _qdiff 属性赋值，然后在第二个数组中检验 _qdiff 属性值是否与计数器相同，如果不同，则将其加入结果数组。

6.1.3　基本选择符

1. *：选择任何元素

（1）语法

```
Ext.query("*")
```

（2）示例

要选择 6.1.2 节的示例中页面的所有元素，可在控制台输入以下命令：

```
console.log(Ext.DomQuery.jsSelect('*')):
```

在这里直接使用 jsSelect 方法目的是避免使用 W3 原生的选择器。要想知道是否使用了原生选择，可在语句之前输入：

```
debugger;
```

这等于在当前行设置了一个断点，程序会在此位置暂停，进入调试模式，然后使用 F11 键单步执行代码就可了解代码是否使用了原生的选择器了。

运行后，可在控制台看到以下输出：

```
[html.x-border-box, head, title, link, script ../Ext4/bootstrap.js, script ext-all-debug.js, style, body#ext-gen1009.x-gecko, h1, table, tbody, tr, th, input#selectAll 全选, th, tr, td.check, input 1, td.title, tr, td.check, input 2, td.title, tr, td.check, input 3, td.title, tr, td.check, input 4, td.title, tr, td.check, input 5, td.title, tr, td.check, input 6, td.title, script, script#_firebugCommand-LineInjector]
```

注意　将鼠标移动到数组中的元素上面，在页面中会出现一个色块标示该标记在页面中的位置与大小，单击可切换到 HTML 面板，并会选中该标记的代码。

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "*");
```

在 getNodes 中将执行以下代码：

```
for(var i = 0, ni; ni = ns[i]; i++){
    cs = ni.getElementsByTagName(tagName);
    for(var j = 0, ci; ci = cs[j]; j++){
        result[++ri] = ci;
    }
}
```

代码中，ns 是根节点数组，通过循环直接使用 DOM 的 getElementsByTagName 选择根节点下的元素，然后保存到 result 数组，并返回。

2. E：根据元素标记 E 选择元素

（1）语法

Ext.query("E") //E 为元素标记，如 input、div

（2）示例

要选择 6.1.2 节的示例中的所有 td 元素。可在命令行中输入以下代码：

```
console.log(Ext.DomQuery.jsSelect("td"))
```

运行后，可在控制台看到以下输出：

```
[td.check, td.title, td.check, td.title, td.check, td.title, td.check, td.title,
td.check, td.title, td.check, td.title]
```

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "E"); ////E 为元素标记，如 input、div
```

在 getNodes 中执行的代码与“*”方式相同。

3. E F：选择包含在标记 E 中的标记 F

（1）语法

```
Ext.query("E F") //E、F 均为元素标记，如 input、div
```

（2）示例

要选择 td 下所有 input，可在控制台输入以下代码：

```
console.log(Ext.DomQuery.jsSelect("td input"));
```

运行后，可在控制台看到以下输出：

```
[input 1, input 2, input 3, input 4, input 5, input 6]
```

因为“全选”复选框是在标记 th 内的，所有没有被选择。

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "E");
mode = "";
n = getNodes(n, mode, "F") //E、F 均为元素标记，如 input、div
```

这里执行了两次 getNodes 方法，执行的代码与“*”的方式是一样的。不过要注意的是第二次传递的参数 n 已经在第一次的执行完成后修改了。因而循环中选择元素的根节点已经是第一次选择出来的元素了。

4. E>F：选择包含在标记 E 中的直接子标记 F

（1）语法

```
Ext.query("E>F") //E、F 均为元素标记，如 input、div
```

（2）示例

在 Firebug 中单击工具栏第二个按钮，然后在页面中用鼠标选择“文本标题 6”旁的复选框，然后在 HTML 面板中单击“编辑”按钮进入编辑状态，将编辑的代码修改为：

```
<span><input type="checkbox" value="6" name="articleId"></span>
```

只是在 input 标记外添加了一个包裹它的 span 标记。再次单击“编辑”按钮完成编辑。现在要使用“E>F”选择符选择 td 内的 input，在控制台中输入以下代码：

```
console.log(Ext.DomQuery.jsSelect("td>input"));
```

运行后，可在控制台看到以下输出：

```
[input 1, input 2, input 3, input 4, input 5]
```

因为经过修改后的值为 6 的 input 不是 td 的直接子元素，所有没有被选择。

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "E");
mode = ">";
n = getNodes(n, mode, "F"); //E、F 均为元素标记，如 input、div
```

第一次执行的 getNodes 方法与“*”方式一样，第二次因为 mode 为“>”，所以会执行以下代码：

```
var utag = tagName.toUpperCase();
for(var i = 0, ni, cn; ni = ns[i]; i++){
    cn = ni.childNodes;
    for(var j = 0, cj; cj = cn[j]; j++){
        if(cj.nodeName == utag || cj.nodeName == tagName  || tagName == '*'){
            result[++ri] = cj;
        }
    }
}
```

与“*”方式的区别在于，这里不直接使用 getElementsByTagName 返回子元素，而是使用 childNodes 属性返回根节点的所有元素，然后判断这些子元素的标记来选取子元素。这样就

可避免了选择子元素下带有相同标记的元素，保证只选择根元素下的直接子元素。

5. E+F：选择所有紧接在元素 E 后的元素 F

（1）语法

```
Ext.query("E+F") //E、F 均为元素标记，如 input、div
```

（2）示例

在控制台输入以下语句：

```
console.log(Ext.DomQuery.jsSelect("body+table"));
console.log(Ext.DomQuery.jsSelect("h1+table"));
```

运行后，可在控制台看到以下输出：

```
[]
[table]
```

因为 table 和 body 之间隔着 h1，所以不是紧接在 body 的元素，从而第 1 个语句选择结果为空，第 2 个语句的结果为 table。

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "E");
mode = "+";
n = getNodes(n, mode, "F"); //E、F 均为元素标记，如 input、div
```

与“E>F”方式类似，第二次 getNodes 会执行以下代码：

```
var utag = tagName.toUpperCase();
for(var i = 0, n; n = ns[i]; i++){
    while((n = n.nextSibling) && n.nodeType != 1);
    if(n && (n.nodeName == utag || n.nodeName == tagName || tagName == '*')){
        result[++ri] = n;
    }
}
```

DOM 的 nextSibling 属性可返回根元素后紧接着的元素（处于同一树层级中）。因为 nextSibling 会返回文本、注释等节点，因而要通过 nodeType 属性（1 表示节点为元素）保证返回的是元素才执行标记的比较。找到根元素后的第一个元素后，如果“n.nodeType != 1”的值为 false，循环中断，这样保证了是紧接在根元素后的元素。

6. E~F：选择在元素 E 之后同层的元素 F

（1）语法

```
Ext.query("E~F") //E、F 均为元素标记，如 input、div
```

（2）示例

如果想选择在 h1 之后且同 script 元素，可在控制台输入以下命令：

```
console.log(Ext.DomQuery.jsSelect("h1~script"));
```

运行后，可在控制台看到以下输出：

```
[script]
```

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "E");
mode = "~";
n = getNodes(n, mode, "F"); //E、F 均为元素标记，如 input、div
```

第二次 getNodes 会执行以下代码：

```
var utag = tagName.toUpperCase();
for(var i = 0, n; n = ns[i]; i++){
    while((n = n.nextSibling)){
        if (n.nodeName == utag || n.nodeName == tagName || tagName == '*'){
            result[++ri] = n;
        }
    }
}
```

代码与“E+F”方式不同在于 while 循环缺少了“n.nodeType != 1”的判断，因而循环会执行到再也找不到根元素的后续元素才终止。

7. #ID：选择 id 属性值为 ID 的元素

（1）语法

```
Ext.query("#ID") //ID 为元素的 id
```

（2）示例：

如果要选择“全选”复选框，可在命令行输入：

```
console.log(Ext.DomQuery.jsSelect("#selectAll"));
```

运行后，可在控制台看到以下输出：

```
[input#selectAll 全选 ]
```

（3）源代码

经过 compile 方法编译后的关键代码：

```
document;n = quickId(n, mode, root, "ID") //ID 为元素的 id
```

代码直接执行 quickId 方法查找元素，其代码如下：

```
function quickId(ns, mode, root, id){
    if(ns == root){
       var d = root.ownerDocument || root;
       return d.getElementById(id);
    }
    ns = getNodes(ns, mode, "*");
    return byId(ns, id);
}
```

如果 ns 等于 root，一般情况下是相等，那么 d 指向 root 的根元素或 root 本身，最后使用 getElementById 方法返回元素。如果不等，则使用 getNodes 方法获取 ns 下的全部元素，使用 byId 方法过滤元素其代码如下：

```
function byId(cs, id){
    if(cs.tagName || cs == document){
        cs = [cs];
    }
    if(!id){
        return cs;
    }
    var result = [], ri = -1;
    for(var i = 0, ci; ci = cs[i]; i++){
        if(ci && ci.id == id){
            result[++ri] = ci;
            return result;
        }
    }
    return result;
}
```

从代码可以看到，byId 方法主要通过逐个判断已获取元素的 id 属性来选择元素。

8. .classname：选择 CSS 类名为 classname 的元素

（1）语法

```
Ext.query("E.classname")
//E 为元素标记，如 input、div
//classname 为 CSS 类名
```

（2）示例

如果想选择类名为 check 的 td，可在命令行中输入以下代码：

```
console.log(Ext.DomQuery.jsSelect(".check"));
// 或
console.log(Ext.DomQuery.jsSelect("td.check"));
```

运行后，可在控制台看到以下输出：

```
[td.check, td.check, td.check, td.check, td.check, td.check]
[td.check, td.check, td.check, td.check, td.check, td.check]
```

这表明在只有 td 设置了类名 check 的情况下，示例中的两个语句结果是一样的。但是，如果有其他不同的标记使用一样的类名，结果就不同了。例如，在 HTML 面板，选择 h1 的 HTML 代码，在右键菜单选择“新增属性”，然后输入 class，回车后再输入“check”，切换到控制台面板后，运行代码，在控制台可看到以下输出：

```
[h1.check, td.check, td.check, td.check, td.check, td.check, td.check]
[td.check, td.check, td.check, td.check, td.check, td.check]
```

结果已经不同了，**因而大家在使用类名选择符的时候，一定要尽可能先缩小自己的选择范围，以避免不必要的错误。**

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "E");
n = byClassName(n, " classname ");
//E 为元素标记，如 input、div
//classname 为 CSS 类名
```

代码在选择出元素后，会执行 byClassName 过滤元素，其代码如下：

```
function byClassName(nodeSet, cls){
    if(!cls){
        return nodeSet;
    }
    var result = [], ri = -1;
    for(var i = 0, ci; ci = nodeSet[i]; i++){
        if((' '+ci.className+' ').indexOf(cls) != -1){
            result[++ri] = ci;
        }
    }
    return result;
};
```

代码很简单，判断元素的 classNames 属性是否包含参数 cls 就行了。

6.1.4　属性选择符

1．[attribute]：选择带有属性 attribute 的元素

（1）语法

```
Ext.query("[attribute]")
Ext.query("E[attribute]") //E 为元素标记，如 input、div
```

（2）示例

如果要选择 input 中带 id 属性的元素，可在命令行中输入：

```
console.log(Ext.DomQuery.jsSelect("input[id]"));
```

运行后，可在控制台看到以下输出：

```
[input#selectAll 全选 ]
```

（3）源代码

经过 compile 后的关键代码：

```
n = getNodes(n, mode, "*");
n = byAttribute(n, "attribute", "undefined", "undefined", "[");
```

如果选择符是以标记 E 开始，则上面代码的“*”会变成“E”。代码首先使用 getNodes 方法获取元素，然后再使用 byAttribute 方法过滤元素，执行的代码如下：

```
function byAttribute(cs, attr, value, op, custom){
```

```
    var result = [],
        ri = -1,
        useGetStyle = custom == "{",
        fn = Ext.DomQuery.operators[op],
        a,
        xml,
        hasXml;

    for(var i = 0, ci; ci = cs[i]; i++){
        if(ci.nodeType != 1){
            continue;
        }
    }
    if(!hasXml){
        xml = Ext.DomQuery.isXml(ci);
        hasXml = true;
    }

    if(!xml){
        if(useGetStyle){
            a = Ext.DomQuery.getStyle(ci, attr);
        } else if (attr == "class" || attr == "className"){
            a = ci.className;
        } else if (attr == "for"){
            a = ci.htmlFor;
        } else if (attr == "href"){
            a = ci.getAttribute("href", 2);
        } else{
            a = ci.getAttribute(attr);
        }
        }else{
            a = ci.getAttribute(attr);
        }
        if((fn && fn(a, value)) || (!fn && a)){
            result[++ri] = ci;
        }
    }
    return result;
}
```

代码中，DomQuery 对象的 operators 属性根据操作符定义了对应的比较函数，其代码如下：

```
operators : {
    "=" : function(a, v){
        return a == v;
    },
    "!=" : function(a, v){
        return a != v;
    },
    "^=" : function(a, v){
        return a && a.substr(0, v.length) == v;
    },
    "$=" : function(a, v){
        return a && a.substr(a.length-v.length) == v;
    },
    "*=" : function(a, v){
```

```
        return a && a.indexOf(v) !== -1;
    },
    "%=" : function(a, v){
        return (a % v) == 0;
    },
    "|=" : function(a, v){
        return a && (a == v || a.substr(0, v.length+1) == v+'-');
    },
    "~=" : function(a, v){
        return a && (' '+a+' ').indexOf(' '+v+' ') != -1;
    }
```

从上面代码可以看到 operators 属性可以非常容易地扩展。

因为当前方式的 op 参数为 undefined，所以 fn 也为 undefined。

循环中，先判断节点的类型是否为元素，如果不是，则继续执行循环。

然后判断循环中的第一个节点是否 XML 节点。如果是，直接使用 getAttribute 方法获取元素属性值。如果不是，则按以下 5 种情况获取属性值：

- 如果通过 CSS 值选择元素（custom 参数值为“{”），则使用 DomQuery 对象 getStyle 方法获取 CSS 值得，实际就是使用 Element 对象的 getStyle 方法获取 CSS 值。
- 获取元素的 className 属性值。
- 获取元素的 for 属性值。
- 因为直接使用 getAttribute 方法获取 href 属性值在不同浏览器获取的值不同，因而需要单列出来通过第二参数确保获取的值是正确值。
- 直接使用 getAttribute 方法获取属性值。

如果 fn 不为 undefined，则执行 operators 属性中对应参数 op 的操作。如果 fn 为 undefined，则判断属性值 a 是否存在。

如果 operators 属性中的操作函数返回 true，或者属性值 a 存在，则将元素保存到 result 数组中。

2．[attribute=value]：选择 attribute 的属性值为 value 的元素

（1）语法

```
Ext.query("[attribute=vale]")
Ext.query("E[attribute=value]") //E 为元素标记，如 input、div
```

（2）示例

如果要选择 input 中 name 为 articleId 的元素，可在命令行中输入：

```
console.log(Ext.DomQuery.jsSelect("input[name=articleId]"));
```

运行后，可在控制台看到以下输出：

```
[input 1, input 2, input 3, input 4, input 5, input 6]
```

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "*"); //如果存在标记E，则*会为E
n = byAttribute(n, "attribute", "vale", "=", "[");
```

与“[attribute]”方式不同的地方就是参数 value 变成了选择符中的值（value），而参数 op 为选择符中的等号（=）操作符。

执行代码中的 fn 指向了 operators 属性中的等号（=）操作。

3．[attribute^=value]：选择 attribute 的属性值以 value 开头的元素

（1）语法

```
Ext.query("[attribute^=vale]")
Ext.query("E[attribute^=value]") //E为元素标记，如input、div
```

（2）示例

该选择符一般应用于选择有共同前缀的元素，例如，页面中有两个表单，表单内的编辑控件都是以表单的名字开头，使用该选择符就可快速选择这些元素。

在 6.1.2 节的示例中，在页面中选择“示例 6-1 使用 Ext.query 选择页面元素”，然后单击编辑按钮，在 h1 标记后加入以下代码：

```
<form name="form1">
    <input name="form1_name" type="text">
    <input name="form1_sex" type="text">
</form>
<form name="form2">
    <input name="form2_name" type="text">
    <input name="form2_sex" type="text">
</form>
```

代码定义了两个表单，每个表单都有两个编辑控件，而且名称都是以表单的名称开始的。这样，在命令行中输入以下命令就可以选择表单内的编辑控件了：

```
console.log(Ext.DomQuery.jsSelect("input[name^=form1]"));
console.log(Ext.DomQuery.jsSelect("input[name^=form2]"));
```

运行后，可在控制台看到以下输出：

```
[input, input]
[input, input]
```

可将鼠标移动数组的元素上，以验证选择的元素。

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "*"); //如果存在标记E，则*会为E
n = byAttribute(n, "attribute", "vale", "^=", "[");
```

执行代码中的 fn 指向了 operators 属性中的“^=”操作。

4．[attribute$=value]：选择 attribute 的属性值以 value 结尾的元素

（1）语法

```
Ext.query("[attribute$=vale]")
Ext.query("E[attribute$=value]") //E 为元素标记，如 input、div
```

（2）示例

如果要选择上面两个表单中以 name 结尾的 input 元素，可在命令行中输入：

```
console.log(Ext.DomQuery.jsSelect("input[name$=name]"));
```

运行后，可在控制台看到以下输出：

```
[input, input]
```

单击数组中的元素，查看一下 HTML 代码，可验证选择是否正确。

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "*"); // 如果存在标记 E，则 * 会为 E
n = byAttribute(n, "attribute", "vale", "$=", "[");
```

执行代码中的 fn 指向了 operators 属性中的“$=”操作。

5．[attribute*=value]：选择 attribute 的属性值包含 value 的元素

（1）语法

```
Ext.query("[attribute*=vale]")
Ext.query("E[attribute*=value]") //E 为元素标记，如 input、div
```

（2）示例

如果想选择包含“form”的 input，可在命令行中输入：

```
console.log(Ext.DomQuery.jsSelect("input[name*=form]"));
```

运行后，可在控制台看到以下输出：

```
[input, input, input, input]
```

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "*"); // 如果存在标记 E，则 * 会为 E
n = byAttribute(n, "attribute", "vale", "*=", "[");
```

执行代码中的 fn 指向了 operators 中的“*=”操作。

6．[attribute%=value]：选择 attribute 的属性值能整除 value 的元素

（1）语法

```
Ext.query("[attribute%=vale]")
Ext.query("E[attribute%=value]") //E 为元素标记，如 input、div
```

（2）示例

如果要选择 value 属性能整除 2 的 input 元素，可在命令行中输入以下命令：

```
console.log(Ext.DomQuery.jsSelect("input[value%=2]"));
```

运行后，可在控制台看到以下输出：

```
[input, input, input, input, input 2, input 4, input 6]
```

两个表单中的 input，因为没有设置 value，所以当成 0 处理了，因而能整除 2，会被选择。

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "*"); // 如果存在标记 E，则 * 会为 E
n = byAttribute(n, "attribute", "vale", "%=", "[");
```

执行代码中的 fn 指向了 operators 属性中的“%=”操作。

7．[attribute!=value]：选择 attribute 的属性值不等于 value 的元素

（1）语法：

```
Ext.query("[attribute!=vale]")
Ext.query("E[attribute!=value]") //E 为元素标记，如 input、div
```

（2）示例

如果要选择 type 类型不是 text 的 input，可在命令中输入以下代码：

```
console.log(Ext.DomQuery.jsSelect("input[type!=text]"));
```

运行后，可在控制台看到以下输出：

```
[input#selectAll 全选 , input 1, input 2, input 3, input 4, input 5, input 6]
```

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "*"); // 如果存在标记 E，则 * 会为 E
n = byAttribute(n, "attribute", "vale", "!=", "[");
```

执行代码中的 fn 指向了 operators 属性中的“$=”操作。

6.1.5 CSS 属性值选择符

CSS 属性选择符的语法可参考属性选择符的语法，不同的地方主要如下：

- 使用大括号（{}）代替属性选择符中的中括号（[]）。
- 不要使用没有属性值的语法，例如“Ext.query(‘input{display}’)”，原因是 CSS 属性与元素属性不同，一直是存在的，所以使用该语法与“Ext.query(‘input’)”结果是一样的。
- 经过 compile 方法编译后的匿名函数中，byAttribute 方法调用的 custom 参数值为“{”。
- 在 byAttribute 方法中，会调用 DomQuery 对象的 getStyle 方法获取 CSS 属性值。
- 注意 padding、background 等可以合并定义的 CSS 属性，查询时必须拆分成单一的属性，如 padding-left、padding-top 或 background-color。

在浏览器中打开 6.1.2 节的示例，在命令行中输入以下命令：

```
// 选择文本居中对齐的元素
console.log(Ext.DomQuery.jsSelect("td{text-align=center}"));
// 选择字体大小是 16px 的元素
console.log(Ext.DomQuery.jsSelect("{font-size=16px}"));
// 选择 padding 属性以 30px 开头的元素
console.log(Ext.DomQuery.jsSelect("{padding^=30px}"));
// 选择 padding 属性以 30px 结尾的元素
console.log(Ext.DomQuery.jsSelect("{padding-left$=30px}"));
// 选择 padding 属性包含 0 的元素
console.log(Ext.DomQuery.jsSelect("{padding-left*=3}"));
// 获取行高不是 30px 的元素
console.log(Ext.DomQuery.jsSelect("{line-height!=30px}"));
```

运行后，可在控制台看到以下输出：

```
[td.check, td.check, td.check, td.check, td.check, td.check]
[html.x-border-box, head, title, link, script ../Ext4/bootstrap.js, script ext-
    all-debug.js, style, style, h1]
[]
[body#ext-gen1009.x-gecko]
[body#ext-gen1009.x-gecko]
[html.x-border-box, head, title, link, script ../Ext4/bootstrap.js, script ext-
    all-debug.js, style, style, body#ext-gen1009.x-gecko, h1, form, input, input,
    form, input, input, input#selectAll 全选 , input 1, input 2, input 3, input 4,
    input 5, input 6, script]
```

第三句命令直接使用 padding 作为属性名称，因而结果为空。

6.1.6　伪类选择符

1．E:first-child

选择元素 E，且其为父节点的第一子节点。

（1）语法

```
Ext.query("E:first-child");  //E 是元素标记，如 input、div
```

（2）示例

如果要将 table 中的标题行的字体改为白色，背景色改为蓝色，可在命令行中输入：

```
var els=Ext.DomQuery.jsSelect('tr:first-child');
els[0].setAttribute('style','color:#fff;background-color:blue;');
```

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "E");
n = byPseudo(n, "first-child", "undefined");
```

代码首先使用 getNode 获取所有符合要求的元素，然后使用 byPseudo 方法过滤元素。byPseudo 的源代码如下：

```
function byPseudo(cs, name, value){
    return Ext.DomQuery.pseudos[name](cs, value);
}
```

代码很简单，调用 DomQuery 对象的 pseudos 属性中由参数 name 指定的方法。与 operators 的定义一样，扩展容易，调用方便。

该伪类方法传入的 name 值是 first-child，所以会执行以下代码：

```
"first-child" : function(c){
    var r = [], ri = -1, n;
    for(var i = 0, ci; ci = n = c[i]; i++){
        while((n = n.previousSibling) && n.nodeType != 1);
        if(!n){
            r[++ri] = ci;
        }
    }
    return r;
},
```

代码中，previousSibling 属性会返回待选元素的前一个节点（同一层级），如果节点的节点类型（nodeType）为元素，循环会停止。如果待选元素之前没有元素，previousSibling 属性会返回 null，循环也会结束。

循环结束后，如果 n 为 null，说明待选元素是其父的第一子节点，要将其保存到结果数组。否则，剔除该元素。

2. E:last-child

选择标记为 E，且其为父节点的最后一个子节点的元素。

（1）语法

```
Ext.query("E:last-child");  //E 是元素标记，如 input、div
```

（2）示例

如果要将 table 的最后一行的背景色修改为黄色，可在命令行中输入：

```
var els=Ext.DomQuery.jsSelect('tr:last-child');
els[0].setAttribute('style','background-color:yellow;');
```

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "E");
n = byPseudo(n, "last-child", "undefined");
```

执行的代码：

```
"last-child" : function(c){
    var r = [], ri = -1, n;
    for(var i = 0, ci; ci = n = c[i]; i++){
        while((n = n.nextSibling) && n.nodeType != 1);
        if(!n){
            r[++ri] = ci;
```

```
        }
    }
    return r;
},
```

除了使用 nextSibling 方法获取下一节点外，代码基本与“first-child”代码一样。

3．E:nth-child(n)

选择标记为 E，且其为父节点的第 n（n ⩾ 1）个子节点的元素。

（1）语法

```
Ext.query("E:nth-child(n)");
//E 是元素标记，如 input、div
//n 为整数，n ⩾ 1
```

（2）示例

如果要将 table 的第三行的背景色修改为绿色，可在命令行中输入：

```
var els=Ext.DomQuery.jsSelect('tr:nth-child(3)');
els[0].setAttribute('style','background-color:green;');
```

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "E");
n = byPseudo(n, "nth-child", "n");
```

执行的代码：

```
"nth-child" : function(c, a) {
    var r = [], ri = -1,
        m = nthRe.exec(a == "even" && "2n" || a == "odd" && "2n+1" || !nthRe2.
             test(a) && "n+" + a || a),
        f = (m[1] || 1) - 0, l = m[2] - 0;
    for(var i = 0, n; n = c[i]; i++){
        var pn = n.parentNode;
        if (batch != pn._batch) {
            var j = 0;
            for(var cn = pn.firstChild; cn; cn = cn.nextSibling){
                if(cn.nodeType == 1){
                   cn.nodeIndex = ++j;
                }
            }
            pn._batch = batch;
        }
        if (f == 1) {
            if (l == 0 || n.nodeIndex == l){
                r[++ri] = n;
            }
        } else if ((n.nodeIndex + l) % f == 0){
            r[++ri] = n;
        }
    }
}
```

```
        return r;
    },
```

代码中粗体代码为选择的关键语句，表 6-2 列出了不同的 a 值所对应的 m、f 和 l 值。

表 6-2　不同的 a 值所对应的 m、f 和 l 值

a	m	f=(m[1] || 1)-0	l=m[2] - 0
even	["2n", "2", ""]	2	0
odd	["2n+1", "2", "1"]	2	1
具体数值，例如 8	["n+8", "", "8"]	1	8

在遍历待选元素的数组中，首先获取元素的父节点，然后遍历父节点下的子节点。为了防止重复节点，会为每个节点添加一个 _batch 属性，其值会设置为计数器的值，如果值与当前计数器值相等，说明是重复节点，不需要做操作。如果节点类型（nodeType）是元素，且不重复，则为其添加 nodeIndex 属性，用来保存节点在父节点下的索引值。

“nth-child(n)”方式是选择第 n 个节点，是具体数值，所以 f 为 1，判断节点索引是否等于 l 值。如果相等，则将该元素保存到结果数组中。

如果是“even”或“odd”，则使用整除法判断元素是否符合奇偶条件。要注意，如果是选择偶数行，则按正常的 nodeIndex 值（l 值为 0）进行整除计算就行了。如果是奇数行，则要加 1（l 值为 1），将奇数行变成偶数行再进行整除计算。

4．E:nth-child(odd) 或 E:odd

选择标记为 E，且其为父节点的奇数子节点的元素。

（1）语法

```
Ext.query("E:nth-child(odd)");    //E 是元素标记，如 input、div
Ext.query("E:odd");
```

“E:odd”是“E:nth-child(odd)”的简写。

（2）示例

如果要将 table 的奇数行的背景色修改为灰色，可在命令行中输入：

```
var els=Ext.DomQuery.jsSelect('tr:odd');
for(var i=1;ln=els.length,i<ln;i++){
    els[i].setAttribute('style','background-color:gray;');
}
```

循环从 1 开始是为了避免修改标题行。

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "E");
n = byPseudo(n, "nth-child", "odd");
```

执行过程与“nth-child(n)”方式一样，只是第三个参数改为“odd”。

5. E:nth-child(even) 或 E:even

选择标记为 E，且其为父节点的偶数子节点的元素。

（1）语法

```
Ext.query("E:nth-child(even)");  //E 是元素标记，如 input、div
Ext.query("E:even");
```

“E:even” 是 “E:nth-child(even)” 的简写。

（2）示例

如果要将 table 的偶数行的背景色修改为银色，可在命令行中输入：

```
var els=Ext.DomQuery.jsSelect('tr:even');
for(var i=0;ln=els.length,i<ln;i++){
    els[i].setAttribute('style','background-color:Silver;');
}
```

经过以上几个伪类选择符的示例，已经将 6.1.2 节的示例的表格实现了隔行变色。如果结合 CSS 和 Element 对象的样式操作，代码会更简单。

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "E");
n = byPseudo(n, "nth-child", "even");
```

执行过程与 “nth-child(n)” 方式一样，只是第三个参数改为 “even” 了。

6. E:only-child

选择标记为 E，且其为父节点的唯一子节点的元素。

（1）语法

```
Ext.query("E:only-child");  //E 是元素标记，如 input、div
```

（2）示例

例如要选择是父节点的唯一子节点的 input，可在命令行中输入：

```
console.log(Ext.DomQuery.jsSelect('input:only-child'));
```

运行后，可在控制台看到以下输出：

```
[input#selectAll 全选 , input 1, input 2, input 3, input 4, input 5, input 6]
```

两个表单的 input 没有被选择是因为它们都不是父节点的唯一子节点。

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "E");
n = byPseudo(n, "only-child", "undefined");
```

执行的代码：

```
"only-child" : function(c){
    var r = [], ri = -1;;
    for(var i = 0, ci; ci = c[i]; i++){
        if(!prev(ci) && !next(ci)){
            r[++ri] = ci;
        }
    }
    return r;
},
```

代码中 prev 方法代码如下：

```
function prev(n){
    while((n = n.previousSibling) && n.nodeType != 1);
    return n;
}
```

其作用就是返回待选元素的前一元素。

代码中 next 方法的代码如下：

```
function next(n){
    while((n = n.nextSibling) && n.nodeType != 1);
    return n;
}
```

其作用就是返回紧跟待选元素后的下一元素。

如果 prev 和 next 方法都返回 null，说明待选元素是节点的唯一子节点，需将元素保存到结果数组。

7．E:checked

选择标记为 E，且其 checked 属性值为 true 的元素。

（1）语法

```
Ext.query("E:checked");  //E 是元素标记，如 input、div
```

（2）示例

将“文章标题 2”旁的复选框选上，然后在命令中输入以下代码：

```
console.log(Ext.DomQuery.jsSelect('input:checked'));
```

运行后，可在控制台看到以下输出：

```
[input 2]
```

（3）源代码

经过 compile 方法编译后的关键代码

```
n = getNodes(n, mode, "E");
n = byPseudo(n, "checked", "undefined");
```

执行的代码：

```
"checked" : function(c){
```

```
    var r = [], ri = -1;
    for(var i = 0, ci; ci = c[i]; i++){
        if(ci.checked == true){
            r[++ri] = ci;
        }
    }
    return r;
},
```

代码直接检查元素的 checked 属性值，如果为 true，则保存到结果数组中。

8. E:first

选择标记为 E 的元素集合中的第一个元素。

（1）语法

```
Ext.query("E:first");  //E 是元素标记，如 input、div
```

（2）示例

如果要选择第一个表单，可在命令行中输入：

```
console.log(Ext.DomQuery.jsSelect('form:first'));
```

运行后，可在控制台看到以下输出：

```
[form]
```

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "E");
n = byPseudo(n, "first", "undefined");
```

执行的代码：

```
"first" : function(c){
    return c[0] || [];
},
```

直接返回数组 c 中的第一元素就行了。如果 c 不存在，则返回空数组。

9. E:last

选择标记为 E 的元素集合中的最后一个元素。

（1）语法

```
Ext.query("E:last");  //E 是元素标记，如 input、div
```

（2）示例

如果要选择最后一个 input，可在命令行中输入以下代码：

```
console.log(Ext.DomQuery.jsSelect('input:last'));
```

运行后，可在控制台看到以下输出：

```
[input 6]
```

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "E");
n = byPseudo(n, "last", "undefined");
```

执行的代码：

```
"first" : function(c){
    return c[c.length-1] || [];
},
```

直接返回数组 c 中的最后一元素就行了。如果 c 不存在，则返回空数组。

10. E:nth(n)

选择标记为 E 的元素集合中的第 n（n ⩾ 1）个元素。

（1）语法

```
Ext.query("E:nth(n)");  //E 是元素标记，如 input、div
```

（2）示例

如果要选择 table 中的第 3 行，可在命令行中输入以下命令：

```
console.log(Ext.DomQuery.jsSelect('tr:nth(3)'));
```

运行后，可在控制台看到以下输出：

```
[tr]
```

（3）源代码

经过 compile 后的关键代码：

```
n = getNodes(n, mode, "E");
n = byPseudo(n, "nth", "n");
```

经过 compile 方法编译后的关键代码：

```
"nth" : function(c, a){
    return c[a-1] || [];
},
```

因为数组索引是从 0 开始，因而要选择第 n 个元素，需要将 n 减 1 作为数组索引。如果 c 不存在，或数组中的 n–1 个元素为 undefined，返回空数组。

11. E:contains(str)

选择标记为 E，且其 innerHTML 属性值包含字符串 str 的元素。

（1）语法

```
Ext.query("E:contains(str)");  //E 是元素标记，如 input、div
```

（2）示例

该选择符一般用于高亮显示查询文本，例如，要将 6.1.2 节的示例中的文章标题中查找“标题”并高亮显示，可在命令行中输入：

```
var els=Ext.DomQuery.jsSelect('td:contains( 标题 )');
var re=/ 标题 /g;
for(var i=0;ln=els.length,i<ln;i++){
    els[i].innerHTML=els[i].innerHTML.replace(re,"<font color='red'> 标题 </font>")
}
```

因为标题栏的标记是 th，所以不会改变标题栏的显示。

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "E");
n = byPseudo(n, "contains", "str");
```

执行的代码：

```
"contains" : function(c, v){
    var r = [], ri = -1;
    for(var i = 0, ci; ci = c[i]; i++){
        if((ci.textContent||ci.innerText||'').indexOf(v) != -1){
            r[++ri] = ci;
        }
    }
    return r;
},
```

因为 IE 和 Firefox 用来返回文本的属性不同，因而为了实现跨平台，就出现了粗体代码中的情况了。如果是 IE，innerText 属性会返回文本，textContent 为 undefined。如果是 Firefox，textContent 属性会返回文本，而 innerText 为 undefined。取得文本后，就可判断其是否包含 v 了。如果包含则将待选元素保存到结果数组中。

12. E:nodeValue(value)

选择标记为 E，且其文本节点的 nodeValue 属性值等于 value 的元素。

（1）语法

```
Ext.query("E:nodeValue(value)");  //E 是元素标记，如 input、div
```

（2）示例

在命令行中输入语句：

```
console.log(Ext.DomQuery.jsSelect('td:nodeValue( 文章标题 2)'));
```

运行后，返回的会是空数组。这是因为经过高亮显示后，td 的文本节点的 nodeValue 已经改变了，在命令行中输入以下代码：

```
console.log(Ext.DomQuery.jsSelect('td:nth(2)')[0].firstChild.nodeValue);
```

运行后的结果是“文章”。**因而使用该选择符的时候一定要小心，在无法确定元素的内容是纯文本的时候，不要轻易使用该选择符。**

如果将命令换成：

```
console.log(Ext.DomQuery.jsSelect('th:nodeValue( 文章标题 )'));
```

就会得到期望的结果了。因为第二个 th 的内容是纯文本的，所以可以得到正确的选择。

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "E");
n = byPseudo(n, "nodeValue", "value");
```

执行的代码：

```
"nodeValue" : function(c, v){
    var r = [], ri = -1;
    for(var i = 0, ci; ci = c[i]; i++){
        if(ci.firstChild && ci.firstChild.nodeValue == v){
            r[++ri] = ci;
        }
    }
    return r;
},
```

代码首先通过 firstChild 属性判断文本节点是否存在，如果存在，则使用文本节点的 nodeValue 属性值与 v 比较，若相等，则将元素保存到结果数组中。

13. E:not(S)

选择标记为 E，且与简单选择符 S 不匹配的元素。

（1）语法

```
Ext.query("E:not(S)");  //E 是元素标记，如 input、div
```

（2）示例

如果选择显示标题的 td，可在命令行中输入：

```
console.log(Ext.DomQuery.jsSelect('td:not(.check)'));
```

运行后，可在控制台看到以下输出：

```
[td.title, td.title, td.title, td.title, td.title, td.title]
```

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "E");
n = byPseudo(n, "not", "S");
```

执行的代码：

```
"not" : function(c, ss){
```

```
    return Ext.DomQuery.filter(c, ss, true);
},
```

代码直接转到 Ext.DomQuery.filter 方法去了，其代码如下：

```
filter : function(els, ss, nonMatches){
    ss = ss.replace(trimRe, "");
    if(!simpleCache[ss]){
        simpleCache[ss] = Ext.DomQuery.compile(ss, "simple");
    }
    var result = simpleCache[ss](els);
    return nonMatches ? quickDiff(result, els) : result;
},
```

代码先将简单选择符去掉前后的空格，然后在缓存 simpleCache 检查是否已经存在编译好的匿名函数，如果没有，使用 compile 方法编译成匿名函数并保存到缓存中。

然后执行匿名函数选择元素，最后根据参数 nonMatches 判断是否使用 quickDiff 方法剔除重复元素。

14. E:has(S)

选择标记为 E，且包含与简单选择符 S 匹配的元素。

（1）语法

```
Ext.query("E:has(S)");  //E 是元素标记，如 input、div
```

（2）示例

如果选择包含 input 的 td，可在命令行中输入：

```
console.log(Ext.DomQuery.jsSelect('td:has(input)'));
```

运行后，可在控制台看到以下输出：

```
[td.check, td.check, td.check, td.check, td.check, td.check]
```

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "E");
n = byPseudo(n, "has", "S");
```

执行的代码：

```
"has" : function(c, ss){
    var s = Ext.DomQuery.select,
        r = [], ri = -1;
    for(var i = 0, ci; ci = c[i]; i++){
        if(s(ss, ci).length > 0){
            r[++ri] = ci;
        }
    }
    return r;
},
```

代码会使用待选元素作为开始元素，执行 DomQuery 对象的 select 方法查询是否与简单选择符 ss 匹配。如果匹配，则返回的不是空数组，数组长度必定大于 0，表示元素符合要求，保存到结果数组中。

15. E:next(S)

选择标记为 E，且其下一节点（同一层级）与简单选择符 S 匹配的元素。

（1）语法

```
Ext.query("E:next(S)");  //E 是元素标记，如 input、div
```

（2）示例

例如要选择与 tabel 同层的 h1，可在命令行中输入：

```
console.log(Ext.DomQuery.jsSelect('h1:next(table)'));
```

运行后，可在控制台看到以下输出：

```
[h1]
```

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "E");
n = byPseudo(n, "next", "S");
```

执行的代码：

```
"next" : function(c, ss){
    var is = Ext.DomQuery.is,
        r = [], ri = -1;
    for(var i = 0, ci; ci = c[i]; i++){
        var n = next(ci);
        if(n && is(n, ss)){
            r[++ri] = ci;
        }
    }
    return r;
},
```

代码首先使用 next 方法获取待选元素的下一元素 n，然后判断 n 是否存在，以及使用 DomQuery 对象的 is 方法判断元素 n 是否匹配选择符 ss。

DomQuery 对象 is 方法的代码如下：

```
is : function(el, ss){
    if(typeof el == "string"){
        el = document.getElementById(el);
    }
    var isArray = Ext.isArray(el),
        result = Ext.DomQuery.filter(isArray ? el : [el], ss);
    return isArray ? (result.length == el.length) : (result.length > 0);
},
```

代码使用 filter 方法来过滤元素，如果过滤后的元素还存在，则表示符合匹配要求，返回 true。否则，返回 false。

16. E:prev(S)

选择标记为 E，且其上一节点（同一层级）与简单选择符 S 匹配的元素。

（1）语法

```
Ext.query("E:prev(S)");  //E 是元素标记，如 input、div
```

（2）示例

例如要选择与 h1 同层的 table，可在命令行中输入：，

```
console.log(Ext.DomQuery.jsSelect('table:prev(h1)'));
```

运行后，可在控制台看到以下输出：

```
[table]
```

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "E");
n = byPseudo(n, "prev", "S");
```

执行的代码：

```
"prev" : function(c, ss){
    var is = Ext.DomQuery.is,
        r = [], ri = -1;
    for(var i = 0, ci; ci = c[i]; i++){
        var n = prev(ci);
        if(n && is(n, ss)){
            r[++ri] = ci;
        }
    }
    return r;
}
```

代码除了使用 prev 方法外，与“E:prev(S)”基本相同。

17. E:any(S1|S2…|Sn)

选择标记为 E，且与简单选择符 S1 到 Sn 中的任一个能匹配的元素。

（1）语法

```
Ext.query("E:any(S1|S2…|Sn)");  //E 是元素标记，如 input、div
```

（2）示例

如果要选择全部 td，可在命令行中输入：

```
console.log(Ext.DomQuery.jsSelect('td:any(.check|.title)'));
```

运行后，可在控制台看到以下输出：

```
[td.check, td.title, td.check, td.title, td.check, td.title, td.check, td.title,
     td.check, td.title, td.check, td.title]
```

（3）源代码

经过 compile 方法编译后的关键代码：

```
n = getNodes(n, mode, "E");
n = byPseudo(n, "any", "S1|S2\u2026|Sn");
```

执行的代码：

```
"any" : function(c, selectors){
    var ss = selectors.split('|'),
        r = [], ri = -1, s;
    for(var i = 0, ci; ci = c[i]; i++){
        for(var j = 0; s = ss[j]; j++){
            if(Ext.DomQuery.is(ci, s)){
                r[++ri] = ci;
                break;
            }
        }
    }
    return r;
},
```

代码首先使用 split 方法拆分选择符，然后通过循环，使用 DomQuery 对象的 is 方法检验元素，只要与选择符中的一个匹配，则保存元素到结果数组中，并中断检验循环，检验下一元素。

6.1.7 扩展选择器

通过 DomQuery 对象的 matchers、operators 和 pseudos 3 个属性可轻松的扩展选择器的功能。

1. 扩展选择符规则

例如，希望使用“&”符号作为 id 选择符号，可在 matchers 属性中添加一个匹配规则，其代码如下：

```
Ext.DomQuery.matchers.push(
    {
        re: /^&([\w-]+)/,
        select: 'n = byId(n, "{1}");'
    }
);
```

这样你就可以使用以下语法通过 id 来选择元素了：

```
Ext.query("&MyId"); //MyId 为元素 id
```

2. 扩展操作

例如，希望添加一个属性值大于 value 的操作，可在 operators 属性内添加一个名称为

“>=”的操作，其代码如下：

```
Ext.DomQuery.operators[">="]=function(a, v){
    return a > v;
}
```

这样你就可以使用以下语法来选择属性值大于 value 的操作了：

```
Ext.query("E[attribute>=value]")
```

3. 扩展伪操作符

例如，希望有一个能返回指定范围元素的操作符，可在 pseudos 属性中添加一个名称为 range 的属性，其代码如下：

```
Ext.DomQuery.pseudos.range=function(c,v){
   var r=[],
        ri=-1,
        v=v.split("-"),
        start= v[0]-1 || 0,
        end= v[1] || c.length;
   for(var i=start;i<end;i++){
          r[++ri] = c[i];
   }
   return r;
}
```

这里要注意，v 不能使用逗号，因为在 jsSelect 方法中，会首先使用逗号拆分选择符，这样就会发生错误了。代码中变量 start 是选择元素的起始位置，end 是结束位置。

添加 range 方法，就可使用以下语法来获取指定范围的元素了：

```
Ext.query("E:range(start-end)")
//start 为起始位置
//end 为结束位置
例如：
Ext.query("li:range(2-4)")
// 获取从第 2 个元素开始到第 4 个元素结束的 li
```

6.1.8　Ext.DomQuery 的使用方法

1. compile

将选择符或 XPATH 编译成一个可重复使用的函数。

（1）语法

Ext.DomQuery.compiel(S[,type])

其中

- S：选择符。
- type：可以为 select 或 simple。默认值为 select。
- 返回值：匿名函数。

（2）示例

在命令行中输入以下代码：

```
console.log(Ext.DomQuery.compile('input'));
```

会在控制台中看到以下输出：

```
function()
```

单击“function”，会在脚本面板看到以下代码：

```
var f = function (root) {var mode;++batch;var n = root || document;n = getNodes(n,
    mode, "input");return nodup(n);};
/* !eval(new String(fn.join("");)) */
```

这就是 compile 方法编译出来的匿名函数。Firebug 可谓神通广大，对了解和分析代码相当的实用，必须好好掌握。

（3）源代码

该方法在 6.1.2 节有详细描述。

2. filter

使用简单选择符过滤元素数组。

（1）语法

```
var el=Ext.DomQuery.filter(els,S,nonMatches);
```

其中：

- els：需要过滤的元素数组。
- S：过滤选择符。
- nonMatches：如果为 true，则选择那些不匹配的元素，否则选择匹配的元素。
- 返回值：元素数组。

（2）源代码

该方法在 6.1.6 节的标题 13 中有详细描述。

3. is

判断元素是否匹配简单选择符。

（1）语法

```
var v=Ext.DomQuery.is(el,S);
```

其中：

- el：可以是字符串、HTMLElement 或元素数组。
- S：选择符。
- 返回值：如果元素匹配简单选择符，返回 true，否则返回 false。

（2）源代码

该方法在 6.1.6 节的标题 15 中有详细描述。

4. jsSelect

根据选择符选择元素。
这是 Ext JS 选择器的入口，可阅读 6.1.2 节的介绍。

5. selectNode

选择一个元素。
（1）语法

```
var el = Ext.DomQuery.selectNode(S[,root]);
```

其中：

- S：选择符。
- root：可选，查询开始的节点或节点 id。
- 返回值：HTMLElement。

（2）源代码

```
selectNode : function(path, root){
    return Ext.DomQuery.select(path, root)[0];
},
```

使用 DomQuery 对象的 select 方法选择到元素后，返回第一个元素就行了。

6. selectValue

选择节点的值。
（1）语法

```
var value = Ext.DomQuery.selectValue(S,[root],defaultValue)
```

其中：

- S：选择符。
- root：可选，查询开始的节点或节点 id。
- defaultValue：默认值。如果找不到节点，返回默认值。
- 返回值：字符串。

（2）源代码

```
selectValue : function(path, root, defaultValue){
    path = path.replace(trimRe, "");
    if(!valueCache[path]){
        valueCache[path] = Ext.DomQuery.compile(path, "select");
    }
    var n = valueCache[path](root), v;
    n = n[0] ? n[0] : n;

    if (typeof n.normalize == 'function') n.normalize();

    v = (n && n.firstChild ? n.firstChild.nodeValue : null);
    return ((v === null||v === undefined||v==='') ? defaultValue : v);
},
```

该方法也会根据选择符建立一个值缓存（valueCache）用来保存编译后的匿名函数。通过匿名函数选择元素后，判断返回的是数组还是单一元素，如果是数组，n 就指向数组的第一个元素。执行 normalize 方法的作用是克服最大文本大小的限制。

如果元素存在 firstChild 属性，则 v 等于 nodeValue 值，否则 v 等于 null。

如果 v 为 null、undefined 或空，则返回默认值。

7. selectNumber

选择节点的数值。

（1）语法

```
var value = Ext.DomQuery.selectValue(S,[root],defaultValue)
```

其中：

- S：选择符。
- root：可选，查询开始的节点或节点 id。
- defaultValue：默认值。如果找不到节点，返回默认值。
- 返回值：数值。

（2）源代码

```
selectNumber : function(path, root, defaultValue){
    var v = Ext.DomQuery.selectValue(path, root, defaultValue || 0);
    return parseFloat(v);
},
```

代码使用 Ext.DomQuery.selectValue 返回值，然后使用 parseFloat 将其转换为数值。

6.1.9 Ext JS 选择器的总结

Ext JS 选择器最大的特点就是容易扩展。而且使用非常方便，通过选择符的组合，可以更好地实现功能。尤其是具有缓存功能，可以使查询速度更快捷。

6.2 获取单一元素：Ext.dom.Element

6.2.1 从错误开始

初次使用 Ext JS 的，多数会犯以下错误：

```
Ext.get('MyID').innerHTML = '内容';
Ext.fly('MyID').innerHTML = '内容';
```

正确的写法应该是：

```
Ext.get('MyID').dom.innerHTML = '内容';
Ext.fly('MyID').dom.innerHTML = '内容';
```

造成这个问题的主要原因是使用 Ext.get 方法或 Ext.fly 方法返回的对象是 Element 对象，

而不是 HTMLElement 对象。因而，你不能直接访问 HTMLElement 的属性，而必须先调用 dom 属性，再调用 HTMLElement 的属性。

要直接操作元素的 dom 属性，最简捷的方法是使用 Ext.getDom 方法，例如上面的例子可修改为：

```
Ext.getDom('MyID').innerHTML = '内容';
```

6.2.2　使用 Ext.get 获取元素

（1）语法

```
var el=Ext.get(id);
```

其中，id 为可以为元素的 id 属性值、HTMLElement 对象、Element 对象、Composite-ElementLite 对象、数组或 document 对象。

（2）示例

例如页面中有一个 div 元素的代码如下：

```
<div id='title'>我爱 Ext JS</div>
```

要获取 div 元素，代码如下：

```
var el=Ext.get('title');
```

如果获取或修改 HTMLElement 的属性，必须使用 dom 属性调用，或者使用 Ext.getDom 方法，例如要修改 innerHTML 属性的值，代码如下：

```
el.dom.innerHTML='修改后的文本';
```

或

```
Ext.getDom('title').innerHTML='修改后的文本';
```

（3）源代码

在 AbstractElement.js（目录 \src\core\src\dom 下）文件中，可找到以下语句：

```
Ext.get = function(el) {
    return Ext.dom.Element.get(el);
};
```

代码中，调用的是 Element 的 get 方法，而该方法是继承自 AbstractElement 对象的，其代码如下：

```
get: function(el) {
    var me = this,
        El = Ext.dom.Element,
        cache,
        extEl,
        dom,
        id;
```

```
if (!el) {
    return null;
}

if (typeof el == "string") {
    if (el == Ext.windowId) {
        return El.get(window);
    } else if (el == Ext.documentId) {
        return El.get(document);
    }

cache = Ext.cache[el];
if (cache && cache.skipGarbageCollection) {
    extEl = cache.el;
    return extEl;
}

if (!(dom = document.getElementById(el))) {
    return null;
}

if (cache && cache.el) {
    extEl = cache.el;
    extEl.dom = dom;
} else {
    extEl = new El(dom);
}
return extEl;
} else if (el.tagName) { // dom element
    if (!(id = el.id)) {
        id = Ext.id(el);
    }
    if (Ext.cache[id] && Ext.cache[id].el) {
        extEl = Ext.cache[id].el;
        extEl.dom = el;
    } else {
        extEl = new El(el);
    }
    return extEl;
} else if (el instanceof me) {
    if (el != me.docEl && el != me.winEl) {
        el.dom = document.getElementById(el.id) || el.dom;
    }
    return el;
} else if (el.isComposite) {
    return el;
} else if (Ext.isArray(el)) {
    return me.select(el);
} else if (el === document) {
    if (!me.docEl) {
        me.docEl = Ext.Object.chain(El.prototype);
        me.docEl.dom = document;
        me.docEl.id = Ext.id(document);
        me.addToCache(me.docEl);
    }
```

```
        return me.docEl;
    } else if (el === window) {
        if (!me.winEl) {
            me.winEl = Ext.Object.chain(El.prototype);
            me.winEl.dom = window;
            me.winEl.id = Ext.id(window);
            me.addToCache(me.winEl);
        }
        return me.winEl;
    }
    return null;
},
```

代码首先判断了传来的参数是否为空或 undefined 等值，如果是，返回 null。

接着判断 el 是否字符串，如果是，使用 getElementById 获取元素。如果不存在，返回 null。接着在缓存 EC（Ext.cache）中寻找元素是否存在。如果存在，从缓存获取元素，并将元素的 dom 属性指向元素的 DOM 对象；如果不存在，再创建一个新的 Element 对象，并使用 addToCache 方法将其添加到缓存 EC 中。

我们看看 addToCache 方法是怎么把 Element 对象添加到缓存的，其代码如下：

```
El.addToCache = function(el, id) {
    if (el) {
        id = id || el.id;
        EC[id] = {
            el: el,
            data: {},
            events: {}
        };
    }
    return el;
};
```

从上面代码中可以看到，addToCache 方法在缓存中添加了一个以元素对象 id 为关键字的对象，对象里包含 el、data 和 events 三个成员，成员 el 指向元素对象，data 和 events 都为空对象。这为以后记录或操作元素对象提供了方便。但是，这会增加了内存的开销，**因而，一般的建议是只对元素进行一次性操作的时候，使用 Ext.fly 而不是 Ext.get 来获取元素。**

回到 get 方法代码继续往下看，获取了元素后，返回元素。

如果参数不是 id，则继续判断是否存在 tagName 属性。如果存在，且 id 属性为 undefined，则使用 Ext.id 方法为其生成一个 id 值。然后在缓存 EC 中查找元素是否存在，不存在则使用 addTOCache 方法将其添加到缓存，并返回元素。

最后判断参数本身是否为元素对象。如果是，则继续判断浏览器是否为 IE，且元素 id 不存在或为空。若是，则重新设置元素的 dom 属性，若不是，则重新使用 getElementById 方法获取 DOM 对象。最后返回元素。

如果以上都不是，通过 isComposite 属性判断参数是否为 Ext.CompositeElementLite 对象，若是，直接返回参数指向的对象。

还不是，则判断是否为数组，若是，则使用 select 方法寻找元素并返回。在 Composite-

ElementLite.js 文件中可找到 select 方法，代码如下：

```
if(Ext.DomQuery){
    Ext.dom.Element.selectorFunction = Ext.DomQuery.select;
}

Ext.dom.Element.select = function(selector, root){
    var elements;
    if(typeof selector == "string"){
        elements = Ext.dom.Element.selectorFunction(selector, root);
    }else if(selector.length !== undefined){
        elements = selector;
    }else{
        // 省略调试信息
    }
    return new Ext.CompositeElementLite(elements);
};
```

代码中前 3 句是将 selectorFunction 方法指向 DomQuery 对象的 select 方法，也就是使用选择器查询。因为参数是在数组时才执行该方法的，所以将变量 el 指向参数后，返回 Ext.CompositeElementLite[⊖]对象。

继续回到 get 方法的代码，如果 el 是 document 对象，则创建一个临时的 Element 对象表示 document 对象并返回。

如果以上都不是，则返回 null 值。

6.2.3 使用 Ext.fly 获取元素

Ext.fly 的语法可参考 Ext.get 的语法，不过在实现上有很大区别，其代码（AbstractElement.js 文件中）如下：

```
Ext.fly = function() {
    return AbstractElement.fly.apply(AbstractElement, arguments);
};
```

从代码可以看到，fly 方法直接调用了 AbstractElement 对象的 fly 方法，其代码如下：

```
fly: function(el, named) {
    var fly = null,
        _flyweights = AbstractElement._flyweights;

    named = named || '_global';

    el = Ext.getDom(el);

    if (el) {
        fly = _flyweights[named] || (_flyweights[named] = new AbstractElement.
            Fly());
        fly.dom = el;
        fly.data = {};
    }

    return fly;
}
```

⊖ 相关信息请阅读 6.5 节。

代码中，AbstractElement 对象的 _flyweights 属性指向一个空对象。而粗体代码会新建一个 Element 对象实例，也就是变量 fly 会指向该实例。而 fly 的 dom 属性会指向获取的元素，然后返回。

从代码可以看到，如果没有传递参数 named，就会总是使用“_global”作为关键字，指向 Element 对象实例，也就是说，fly 指向的对象随时会变。如果不注意，就会发生错误，从而产生错误。例如，在页面中有 id 分别为 id1、id2 的两个 div，其代码如下：

```
<div id='id1'>1</div>
<div id='id2'>2</div>
```

然后再执行以下代码：

```
var el=Ext.fly('id1');
console.log(el.dom);
Ext.fly('id2').dom.innerHTML=' 我是 id2';
console.log(el.dom);
el.dom.innerHTML = ' 我是 id1';
```

你会发现 id 为 id1 的 div 的内部文本并没有改变，而 id2 的 div 内部文本是“我是 id1”。而在控制台看到的输出是：

```
<div id="id1">
<div id="id2">
```

这说明 el 在执行第二个 Ext.fly 时已经被改变了，从而说明 **Ext.fly 不适合在多次调用的场合使用，只适合一次性的场合使用**。这样做的好处是为了减少内存的消耗，因为 get 方法会在缓存中记录对象，而当这些元素只需要使用一次，没必要记录的时候，使用 get 方法就会浪费内存。

6.2.4　使用 Ext.getDom 获取元素

使用 Ext.getDom 返回的是 HTMLElement 对象，这个要和 Ext.get 和 Ext.fly 区分清楚。其语法如下：

```
Ext.getDom(id[,strict]);
```

其中 id 可以为元素的 id、Element 对象或 HTMLElement 对象。strict 为可选参数，其作用是是否做一个严格的 id 检查，这是因为 IE 可通过 name 属性获取对象，而不必是 id 属性。这个看看 Ext.getDom 的源代码就很清楚，其源代码如下：

```
getDom : function(el, strict) {
    if (!el || !document) {
        return null;
    }
    if (el.dom) {
        return el.dom;
    } else {
        if (typeof el == 'string') {
            var e = document.getElementById(el);
```

```
            if (e && isIE && strict) {
                if (el == e.getAttribute('id')) {
                    return e;
                } else {
                    return null;
                }
            }
            return e;
        } else {
            return el;
        }
    },
```

代码首先判断 el 是否为空或不存在，或者 document 对象是否存在，如果都不存在，则返回 null。接着判断 el 是否 Element 对象，如果是，返回其 dom 属性。如果不是，判断 el 是否为字符串，若是，则使用 getElementById 方法返回 HTMLElement 对象。接着这句就很清楚了，如果浏览器是 IE（isIE）且确认要强制检测 id 属性，判断 el 是否等于 e 对象的 id 属性。如果等于，则返回 e，如果不是则返回 null。

如果 el 不是字符串，则直接返回 el。

6.2.5 获取元素的总结

从 6.2.2 节和 6.2.3 节的介绍，可总结出使用 Ext.get 和 Ext.fly 的几个要点：

- Ext.get 比较耗内存，尽量避免使用。
- Ext.fly 虽然比较省内存，但是只能一次性使用。
- 如果你偏爱 Ext.fly，那么在使用时，务必定义 name 参数以示区别。
- 如果只是简单的调用元素的 DOM 属性，尽量使用 Ext.getDom 方法。

6.3 元素生成器：Ext.dom.Helper

6.3.1 概述

不建议使用 DOM 生成 HTML 代码，原因是代码量比较大的时候，性能比较低。而且自己组装 HTML 字符串比较乱。全部使用模版又无法适应特殊要求。因而，JavaScript 框架都会提供生成 HTML 代码的函数或对象，以方便操作。在 Ext JS 中，实现此功能的 DomHelper 对象。

DomHelper 对象可根据需要生成 HTML 代码、DOM 对象或模板。

6.3.2 使用 createHtml 或 markup 方法生成 HTML 代码

使用 createHtml 方法可创建 HTML 代码，其源代码如下：

```
createHtml: function(spec) {
    return this.markup(spec);
},
```

直接调用 markup 方法（\src\core\src\dom 目录下的 AbstractHelper.js 文件），其代码如下：

```
markup: function(spec) {
    if (typeof spec == "string") {
        return spec;
    }

    var buf = this.generateMarkup(spec, []);
    return buf.join('');
},
```

如果参数 spec 是字符串，直接返回。否则调用 generateMarkup 方法生成标记数组，其代码如下：

```
generateMarkup: function(spec, buffer) {
    var me = this,
        attr, val, tag, i, closeTags;

    if (typeof spec == "string") {
        buffer.push(spec);
    } else if (Ext.isArray(spec)) {
        for (i = 0; i < spec.length; i++) {
            if (spec[i]) {
                me.generateMarkup(spec[i], buffer);
            }
        }
    } else {
        tag = spec.tag || 'div';
        buffer.push('<', tag);

        for (attr in spec) {
            if (spec.hasOwnProperty(attr)) {
                val = spec[attr];
                if (!me.confRe.test(attr)) {
                    if (typeof val == "object") {
                        buffer.push(' ', attr, '="');
                        me.generateStyles(val, buffer).push('"');
                    } else {
                        buffer.push(' ', me.attribXlat[attr] || attr, '="', val, '"');
                    }
                }
            }
        }

        if (me.emptyTags.test(tag)) {
            buffer.push('/>');
        } else {
            buffer.push('>');

            if ((val = spec.tpl)) {
                val.applyOut(spec.tplData, buffer);
            }
            if ((val = spec.html)) {
                buffer.push(val);
```

```
            }
            if ((val = spec.cn || spec.children)) {
                me.generateMarkup(val, buffer);
            }

            closeTags = me.closeTags;
            buffer.push(closeTags[tag] || (closeTags[tag] = '</' + tag + '>'));
        }
    }

    return buffer;
},
```

如果参数 spec 是字符串，直接将其放入 buffer 数组内。

如果参数 spec 是数组，则逐个递归调用 generateMarkup 数组元素生成标记。

如果 spec 既不是字符串也不是数组，则使用配置对象方式生成 HTML 代码。

配置对象主要有以下 4 个属性：

- tag：元素的标记，如 div、input。如果没有设置该属性，则默认使用 div 作为元素标记。
- cls：元素样式名称。
- html：元素内部的 HTML 代码（innerHTML）。
- children 或 cn：要创建的元素下的子元素。可根据需要嵌套。

代码首先组装了 HMTL 代码的标记部分，如果对象 spec 的 tag 属性不存在，使用 div 作为标记。

接着使用循环遍历 spec 中的成员。

如果属性 attr 不是 tag、html、children 或 cn 等主要属性，就判断其值（val）是不是对象。如果是对象，则将 attr 加入数组，然后调用 generateStyles 生成样式。最终的结果代码格式如下：

```
attr="key1:val1;key2:val2;…keyn:valn"
```

如果 val 不是对象，则根据 atrr 的值生成以下 3 种代码：

- 如果 atrr 的值等于“cls”，生成以下代码：

```
class="val 的值 "
```

- 如果 attr 的值等于“htmlFor”，生成以下代码：

```
for="val 的值 "
```

- 如果 attr 的值不等于“cls”或“htmlFor”，生成以下代码：

```
attr="val 的值 "
```

最后一种情况说明可以直接使用 HTML 元素的属性作为配置对象的属性。例如，以下配置对象：

```
{tag:'input',type:'raido',value:'true'}
```

会输出：

```
<input type="radio" value="1"/>
```

其中 type 和 value 就直接使用 input 的属性作为配置对象的属性。

完成了 HTML 的代码的中间部分，现在要做的是怎样闭合 HTML 代码。首先要判断元素的标记是不是 input、img 这些不需要闭合标记的元素，如果是，直接使用“/>”闭合标记。

如果不是，先使用“>”闭合标记。然后判断是否带 tpl（模板）属性，如果是，则调用模板的 applyOut 方法将模板写入数组。如果 spec 带 html 属性，则将 html 代码写入数组。最后判断是否有 children 或 cn 属性，如果有，则递归调用 generateMarkup 方法生成子元素。最后在 buffer 数组中加入闭合标记。

最后将数组返回 buffer 数组。再将数组转换为字符串返回 createHtml 方法完成标记的生成。

明白了 createHtml 方法的执行过程，使用起来就得心应手多了。例如，要创建可折叠的面板 Accordion 中的报表面板，可这样写：

```
var html=Ext.dom.Helper.createHtml([
   {tag:"a",cls:"titlebar",html:"报表"},
   {cn:[
          {tag:"p",cn:[
          {tag:"a",html:"月度"},
          {tag:"a",html:"季度"},
          {tag:"a",html:"年度"}
          ]}
   ]}
]);
console.log(html);
// 结果：<a class="titlebar">报表</a><div><p><a>月度</a><a>季度</a><a>年度</a></
p></div>
```

当然，实际中不会这样应用，只有结合了 DomHelper 对象提供的操作方法，才会显示出 DomHelper 的功效。

6.3.3　使用 createDOM 方法生成 DOM 对象

一般情况下，是不建议使用 createDOM 方法的，因为 DOM 原生操作的性能实在不佳，其代码如下：

```
function createDom(o, parentNode){
    var el,
        doc = document,
        useSet,
        attr,
        val,
        cn;

if (Ext.isArray(o)) {
    el = doc.createDocumentFragment();
    for (var i = 0, l = o.length; i < l; i++) {
        createDom(o[i], el);
    }
} else if (typeof o == 'string') {
    el = doc.createTextNode(o);
} else {
```

```
el = doc.createElement( o.tag || 'div' );
useSet = !!el.setAttribute;
for (attr in o) {
    if(!confRe.test(attr)){
        val = o[attr];
        if(attr == 'cls'){
            el.className = val;
        }else{
            if(useSet){
                el.setAttribute(attr, val);
            }else{
                el[attr] = val;
            }
        }
    }
}
Ext.DomHelper.applyStyles(el, o.style);

if ((cn = o.children || o.cn)) {
    this.createDom(cn, el);
} else if (o.html) {
    el.innerHTML = o.html;
}
}
if(parentNode){
   parentNode.appendChild(el);
}
return el;
}
```

与 createHTML 方法有个主要区别，就是这里要传入一个父节点（parentNode）的参数。

与 createHTML 方法一样，首先判断参数 o 是不是数组，如果是，要先使用 createDocument-Fragment 方法创建一个文档碎片。使用文档碎片的好处是可以避免立即更新页面，而是等所有元素组件完成后一次性添加到页面，这和 innerHTML 属性的原理类似。直接通过循环递归调用 createDom 方法将元素组装到文件碎片中。

如果是字符串，则使用 createTextNode 方法创建一个文本节点。

如果不是字符串或数组，则按配置对象创建元素。首先使用 createElement 方法根据 tag 属性生成一个 HTMLElement 对象。如果 tag 属性不存在，则使用 div 作为默认标记。因为有些元素在 IE 上不支持 setAttribute 方法，所以需要使用 useSet 来确定使用什么方法为元素附加属性。

在遍历对象 o 的属性的循环中，如果属性 attr 不是 tag、html、children 或 cn 等主要属性，接着要判断 attr 的值是否是“cls”，如果是，将元素的 classname 属性设置为 val 值；如果不是，且元素支持 setAttribute 方法，则使用 setAttribute 方法为元素添加属性。否则，使用 JavaScript 对象添加属性的方法为元素添加属性。这里与 createHTML 方法不同的地方就是配置对象定义时不能使用 htmlFor 属性，要直接使用 for 属性。

接着使用 DomHelper 对象的 applyStyles 方法为元素添加样式。

如果有 children 或 cn 属性，递归调用 createDom 方法继续添加子元素。如果存在 html 属性，使用 innerHTML 属性为元素添加 HTML 代码。

如果存在父节点（parentNode），则使用 appendChild 方法将元素追加到父节点。

6.3.4　使用 createTemplate 方法创建模板

使用 createTemplate 方法，可通过配置对象来创建模板，其代码如下：

```
createTemplate : function(o){
    var html = this.markup(o);
     return new Ext.Template(html);
}
```

从代码中可以看到，createTemplate 方法其实就是使用 markup 方法生成模板代码，然后再使用模板生成 HTML 代码。

6.3.5　Helper 对象的使用方法

1. insertHtml

在指定元素的指定位置插入 HTML 代码。

（1）语法

```
Ext.dom.Helper.insertHtml(where,el,html);
```

其中：

- where ：插入位置，值可以为 beforeBegin（元素开始标记之前）、afterBegin（元素开始标记之后）、beforeEnd（元素结束标记之前）或 afterEnd（元素结束标记之后）。
- el：要插入 HTML 代码的元素，可为 HTMLElement 对象或 TextNode 对象。
- html：字符串，要插入的 HTML 代码。
- 返回值：新节点的 HTMLElement 对象。

（2）示例

如果想在 6.1.2 节的示例的表格的标题栏下添加一行数据，可在命令行输入以下代码：

```
var el=Ext.query("tr:first-child")[0];
var dh=Ext.dom.DomHelper;
dh.insertHtml('afterend',el,
    dh.markup({
        tag:'tr',
        cn:[
            {tag:'td',cls:'check',cn:[
     {tag:'input',type:'checkbox',name:'articleId',value:'8'}]},
            {tag:'td',cls:'title',html:'新插入文章'}
        ]
    })
)
```

也可以使用以下命令：

```
var el=Ext.query("tr:nth(2)")[0];
var dh=Ext.dom.DomHelper;
dh.insertHtml('beforebegin',el,
    dh.markup({
        tag:'tr',
        cn:[
            {tag:'td',cls:'check',cn:[
     {tag:'input',type:'checkbox',name:'articleId',value:'8'}]},
            {tag:'td',cls:'title',html:'新插入文章'}
        ]
    })
)
```

在 HTML 面板中，可看到在标题栏下多了以下 HTML 代码：

```
<tr><td class="check"><input type="checkbox" value="8" name="articleId"></td><td
    class="title">新插入文章</td></tr>
```

（3）源代码

```
insertHtml : function(where, el, html){
    var hash = {},
        hashVal,
        range,
        rangeEl,
        setStart,
        frag,
        rs;

    where = where.toLowerCase();
    hash[beforebegin] = ['BeforeBegin', 'previousSibling'];
    hash[afterend] = ['AfterEnd', 'nextSibling'];

    if (el.insertAdjacentHTML) {
        if(tableRe.test(el.tagName) && (rs = insertIntoTable(el.tagName.
            toLowerCase(), where, el, html))){
            return rs;
        }

        hash[afterbegin] = ['AfterBegin', 'firstChild'];
        hash[beforeend] = ['BeforeEnd', 'lastChild'];
        if ((hashVal = hash[where])) {
            el.insertAdjacentHTML(hashVal[0], html);
            return el[hashVal[1]];
        }
    } else {
        if (Ext.isTextNode(el)) {
            where = where === 'afterbegin' ? 'beforebegin' : where;
            where = where === 'beforeend' ? 'afterend' : where;
        }
        range = Ext.supports.CreateContextualFragment ? el.ownerDocument.
            createRange() : undefined;
        setStart = 'setStart' + (endRe.test(where) ? 'After' : 'Before');
```

```
if (hash[where]) {
    if (range) {
        range[setStart](el);
        frag = range.createContextualFragment(html);
    } else {
        frag = createContextualFragment(html);
    }
    el.parentNode.insertBefore(frag, where == beforebegin ? el :
        el.nextSibling);
    return el[(where == beforebegin ? 'previous' : 'next') + 'Sibling'];
} else {
    rangeEl = (where == afterbegin ? 'first' : 'last') + 'Child';
    if (el.firstChild) {
        if (range) {
            range[setStart](el[rangeEl]);
            frag = range.createContextualFragment(html);
        } else {
            frag = createContextualFragment(html);
        }

        if(where == afterbegin){
            el.insertBefore(frag, el.firstChild);
        }else{
            el.appendChild(frag);
        }
    } else {
        el.innerHTML = html;
    }
                return el[rangeEl];
            }
        }
    // 省略抛出异常代码
},
```

对象 hash 的作用是保存插入位置对应使用到的查找节点方法，它的属性名称就是插入位置的名称。

首先判断元素是否支持 insertAdjacentHTML 方法。如果支持，再检查 el 的标记是否为表格元素的标记，尝试使用 insertIntoTable 方法在表格内插入代码，其代码如下：

```
function insertIntoTable(tag, where, el, html) {
    var node,
        before;

    tempTableEl = tempTableEl || document.createElement('div');

if(tag == 'td' && (where == afterbegin || where == beforeend) ||
   !tableElRe.test(tag) && (where == beforebegin || where == afterend)) {
    return null;
}
before = where == beforebegin ? el :
         where == afterend ? el.nextSibling :
         where == afterbegin ? el.firstChild : null;

if (where == beforebegin || where == afterend) {
```

```
        el = el.parentNode;
    }

    if (tag == 'td' || (tag == 'tr' && (where == beforeend || where ==
         afterbegin))) {
        node = ieTable(4, trs, html, tre);
    } else if ((tag == 'tbody' && (where == beforeend || where == afterbegin)) ||
               (tag == 'tr' && (where == beforebegin || where == afterend))) {
        node = ieTable(3, tbs, html, tbe);
    } else {
        node = ieTable(2, ts, html, te);
    }
    el.insertBefore(node, before);
    return node;
}
```

变量 tempTableEl 是 DomHelper 对象内的全局变量。如果为 null，会指向一个 div 元素。

接着，如果标记是 td，且插入位置是 afterbegin 或 beforeend，又或者标记不是 td、tr 或 tbody，而插入位置是 beforebegin 或 afterend，则不能插入，返回 null。

如果插入位置是开始标记之前，那么 before 指向元素 el ；如果插入位置是结束标记之后，那么 before 指向紧跟元素 el（同层）的下一元素；如果插入位置是开始标记之后，那么 before 指向元素的第一个子节点；如果都不是，before 为 null 值。

如果插入位置是开始标记之前或结束标记之后，则元素 el 修改为元素的父元素。

无论以上何种情况，都会调用 ieTable 方法执行插入操作，其代码如下：

```
function ieTable(depth, s, h, e){
    tempTableEl.innerHTML = [s, h, e].join('');
    var i = -1,
        el = tempTableEl,
        ns;
    while(++i < depth){
        el = el.firstChild;
    }
    ns = el.nextSibling;
    if (ns){
        var df = document.createDocumentFragment();
        while(el){
            ns = el.nextSibling;
            df.appendChild(el);
            el = ns;
        }
        el = df;
    }
    return el;
}
```

表 6-3 详细说明了不同情况下 tempTableEl 元素内 HTML 代码情况。

表 6-3　tempTableEl 元素生成的 HTML 代码

标记和插入位置	s	e	[s, h, e].join('')
td 或 tr（beforeend、afterbegin）	<table><tbodby><tr>	</tr></tbody></table>	<table><tbody><tr>html</tr></tbody></table>
tbody（beforeend、afterbegin）或 tr（beforebegin、afterend）	<table><tbody>	</tbody></table>	<table><tbody>html</tbody></table>
其他情况	<table>	</table>	<table>html</table>

- 其实就是构造一个包含插入代码的表格，然后通过 while 循环，依据插入代码在构造的表格中的深度，使用 firstChild 属性获取插入代码所在节点。

因为要插入的代码可能存在同层结构的元素，例如插入“ 1 2 ”，那么使用 firstChild 获取的将是第 1 个 div 元素，如果这时候将 el 返回，那么就会漏了第 2 个 div，这时候，就要使用 nextSibling 属性返回紧跟第 1 个子节点后的元素，如果存在，说明要将这些也包括在返回的 el 中。因而先使用 createDocumentFragment 方法创建一个文档碎片，通过 while 循环，使用 nextSibling 不断返回同层的元素，使用 appendChild 方法将其添加到文档碎片中，最后返回这个文档碎片。

节点返回后，使用 insertBefore 方法插入节点。

如果 insertIntoTable 方法返回的不是 null，说明插入成功，返回 rs 就行了。

如果以上情况都行不通，就继续往下执行。

如果 hash 对象包含 where 指定的插入位置，则 hashVal 不为 null，可使用 insert-Adjacent-HTML 方法直接插入代码，并返回插入节点。

如果元素不支持 insertAdjacentHTML 方法，则先判断元素 el 是否文本节点。因为不能在文本节点内插入代码，所以需要修改插入位置，afterbegin 修改为 beforebegin，或 beforeend 修改为 afterend。

如果浏览器支持创建 Range 对象，则使用 createRange 方法在元素 el 的文档中创建 Range 对象。

如果插入位置包含字符串“end”，说明插入位置在结束标记，因而 setStart 的值为“setStartAfter”，否则为“setStartBefor”。

这里要注意，hash 对象内只有 afterbegin 和 beforeend 属性。因而当插入位置是 afterbegin 或 beforeend 时，如果可使用 Range 对象，则通过 Range 对象的 setStartAfter 方法在 el 元素后设置开始范围，或通过 setStartBefor 方法在 el 元素前设置开始范围。然后使用 Range 对象的 createContextualFragment 方法直接将代码转换为文档碎片。

如果不支持 Range 对象，则使用内部的 createContextualFragment 方法生成文档碎片，其代码如下：

```
function createContextualFragment(html){
    var div = document.createElement("div"),
        fragment = document.createDocumentFragment(),
        i = 0,
        length, childNodes;
```

```
        div.innerHTML = html;
        childNodes = div.childNodes;
        length = childNodes.length;

        for (; i < length; i++) {
            fragment.appendChild(childNodes[i].cloneNode(true));
        }

        return fragment;
    }
```

代码先生成一个 div 元素，然后将插入代码编程 div 的内容。然后通过获取 div 元素的子节点的办法，将子节点的副本（cloneNode）追加到文档碎片中。这里要注意，cloneNode 方法的参数只有为 true 时才会复制其下的子节点。

文档碎片生成好后，就可以使用 insertBefore 方法插入了。DOM 原生操作只有一个 insertBefore 方法确实麻烦，无时无刻都要根据插入位置去查找符合 insertBefore 方法的元素。

插入完成后返回插入节点。

如果插入位置是 afterbegin 或 beforeend，则根据插入位置确定是在元素第一个子节点（afterbegin）还是最后一个子节点（beforeend）插入。

如果元素 el 的 firstChild 属性不为 null，说明元素有子节点。接着检查是否可以使用 Range 对象，如果可以，根据插入位置锁定开始位置在第一个子节点或最后一个子节点，最后使用 Range 对象的 createContextualFragment 方法将插入代码转换为文档碎片。

如果不支持 Range 对象，使用内部 createContextualFragment 方法将插入代码转换为文档碎片。

如果插入位置是 afterbegin，在元素第一子节点前插入就行了（insertBefore），这个简单。如果是 beforeend，在元素最后一个子节点追加就行了。如果元素没有子节点，更加简单了，直接使用 innerHTML 属性就可以了。最后根据插入位置选择元素第一子节点或最后一个子节点返回即可。

通过以上的代码分析可以看到，如果不经过封装，使用 DOM 的原生操作方法为一个元素插入 html 代码是多么的繁琐，更何况还要兼顾不同浏览器的差异。这需要花费相当多的时间去研究才行。不过，有了框架，一切都将变得简单。

2. append

创建新的元素，并追加到指定的元素下。

（1）语法

```
Ext.dom.Helper.append(el,o[,returnEl]);
```

其中：

- el：要插入新元素的元素。
- o：创建新元素的配置对象，可以是数组、字符串或对象。
- returnEl：可选参数。如果为 true 返回 Element 对象。默认值为 false，返回 HTML-

Element 对象。

❑ 返回值：HTMLElement 对象或 Element 对象。

（2）示例

如果想在示例 6-1 的表格中追加一行数据，可在命令中输入以下代码：

```
var el=Ext.query("table")[0];
var dh=Ext.dom.Helper;
dh.append(el,
    {
            tag:'tr',
                    cn:[
                    {tag:'td',cls:'check',cn:[
     {tag:'input',type:'checkbox',name:'articleId',value:'8'}]},
                    {tag:'td',cls:'title',html:'新插入文章'}
            ]
        }
)
```

此代码与 innerHTML 相比，最突出的地方就是可以直接使用配置对象，而不需要先生成 HTML 代码。

（3）源代码

```
append : function(el, o, returnElement){
    return doInsert(el, o, returnElement, beforeend, '', true);
},
```

代码执行了 doInsert 方法，其代码如下：

```
function doInsert(el, o, returnElement, pos, sibling, append){
    el = Ext.getDom(el);
    var newNode;
    if (pub.useDom) {
        newNode = createDom(o, null);
        if (append) {
            el.appendChild(newNode);
        } else {
            (sibling == 'firstChild' ? el : el.parentNode).insertBefore(newNode,
                el[sibling] || el);
        }
    } else {
        newNode = Ext.dom.Helper.insertHtml(pos, el, Ext.dom.DomHelper.
            createHtml(o));
    }
    return returnElement ? Ext.get(newNode, true) : newNode;
}
```

代码首先使用 getDom 获取指定元素的 DOM 对象。

属性 useDom 的默认值是 false，表示不使用 createDom 方法生成节点。如果你希望使用 createDom 方法生成节点，可使用以下代码设置 useDom 属性：

```
Ext.dom.Helper.useDom = true;
```

不过不建议这样。

如果 useDom 为 true，则使用 createDom 方法创建节点。因为当前是追加操作，所以插入点 pos 的值是“beforeend”。

如果是追加方式，则使用 appendChild 方法添加节点；否则使用 insertBefore 插入节点。

如果 useDom 为 false，则使用 insertHtml 方法插入代码。

最后根据 returnElement 的值决定是返回新节点的 HTMLElement 对象还是 Element 对象。

3. insertAfter

创建新的元素，并插入到指定的元素后。

（1）语法

该方法语法请参考 append 方法。

（2）示例

在 6.1.2 节的示例的表格的标题栏下插入一行的代码可修改为：

```
var el=Ext.query("tr:first-child")[0];
var dh=Ext.dom.Helper;
dh.insertAfter(el,
{tag:'tr',
        cn:[
            {tag:'td',cls:'check',cn:[
       {tag:'input',type:'checkbox',name:'articleId',value:'8'}]},
            {tag:'td',cls:'title',html:'新插入文章'}
        ]
    }
)
```

与使用 innerHTML 方法相比，此代码简单了不少。

（3）源代码

```
insertAfter : function(el, o, returnElement){
   return doInsert(el, o, returnElement, afterend, 'nextSibling');
},
```

与 append 方法一样，直接调用 doInsert 方法，插入位置变成了 afterend，sibling 变成了“nextSibling”。

4. insetBefor

创建新的元素，并插入到指定元素前。

（1）语法

该方法语法请参考 append 方法。

（2）示例

在 6.1.2 节示例的表格的标题栏下插入一行的代码可修改为：

```
var el=Ext.query("tr:nth(2)")[0];
var dh=Ext.dom.Helper;
dh.insertAfter(el,
{tag:'tr',
```

```
        cn:[
            {tag:'td',cls:'check',cn:[
        {tag:'input',type:'checkbox',name:'articleId',value:'8'}]},
            {tag:'td',cls:'title',html:'新插入文章'}
        ]
    }
)
```

与使用 innerHTML 方法相比，此代码简单了不少。

（3）源代码

```
insertBefore : function(el, o, returnElement){
    return doInsert(el, o, returnElement, beforebegin);
},
```

与 append 方法一样，直接调用 doInsert 方法，插入位置变成了 beforebegin，省略了 sibling 参数。

5. insertFirst

创建新的元素，并作为指定元素的第一子节点插入到指定元素中。

（1）语法

该方法语法请参考 append 方法。

（2）示例

如果要在 6.1.2 节示例的表格内插入一行并将其作为第一行，可在命令行中输入以下代码：

```
var el=Ext.query("table")[0];
var dh=Ext.dom.Helper;
dh.insertFirst(el,
{tag:'tr',
        cn:[
            {tag:'td',cls:'check',cn:[
        {tag:'input',type:'checkbox',name:'articleId',value:'8'}]},
            {tag:'td',cls:'title',html:'新插入文章'}
        ]
    }
)
```

（3）源代码

```
insertFirst : function(el, o, returnElement){
    return doInsert(el, o, returnElement, afterbegin, 'firstChild');
},
```

与 append 方法一样，直接调用 doInsert 方法，插入位置变成了 afterbegin，sibling 参数修改为“firstChild”。

6. overview

创建新的元素，并替换指定元素的内容。

（1）语法

该方法语法请参考 append 方法。

（2）示例

如果要替换 6.1.2 节示例的表格中的第二行，可在命令行中输入以下代码：

```
var el=Ext.query("tr:nth(2)")[0];
var dh=Ext.dom.Helper;
dh.overwrite(el,[
   {tag:'td',cls:'check',cn:[
           {tag:'input',type:'checkbox',name:'articleId',value:'8'}]},
           {tag:'td',cls:'title',html:'新插入文章'}
 ]
)
```

这里要注意，因为是替换，所以要保留 tr 标记并替换其内容，而不能直接在 table 标记内替换。

（3）源代码

```
overwrite : function(el, o, returnElement){
    el = Ext.getDom(el);
    el.innerHTML = createHtml(o);
    return returnElement ? Ext.get(el.firstChild) : el.firstChild;
},
```

代码很简单，使用 getDom 方法获取元素的 DOM 对象，然后替换 innerHTML 属性的内容就行了。

7. applyStyles

设置元素的 style 属性。

（1）语法

```
Ext.dom.Helper.applyStyles(el,styles);
```

其中：

- el：应用样式的元素。
- styles：要应用的样式定义，可为字符串、对象或函数。
- 返回值：无。

（2）示例

如果要将 6.1.2 节示例中的表格的标题栏背景设置为蓝色，字体设置为白色，可在命令行中输入以下代码：

```
var el=Ext.query("tr:first-child")[0];
var dh=Ext.dom.Helper;
dh.applyStyles(el,"background-color:blue;color:white");
```

在 HTML 代码中可看到标题栏的 tr 标记以附加以下代码：

```
<tr style="background-color: blue; color: white;">
```

（3）源代码

```
applyStyles : function(el, styles){
```

```
if (styles) {
    el = Ext.fly(el);
    if (typeof styles == "function") {
        styles = styles.call();
    }
    if (typeof styles == "string") {
        styles = Ext.dom.Element.parseStyles(styles);
    }
    if (typeof styles == "object") {
        el.setStyle(styles);
    }
},
```

代码首先判断参数 styles 是否有值，如果有，继续检查。
如果是函数，则使用 call 方法调用函数，并返回样式值。
如果 styles 是字符串，使用 Element 对象的 parseStyles 方法处理字符。
如果 styles 是对象，使用 Element 对象的 setStyle 方法处理对象。

6.4　元素的操作

1. 概述

Ext.dom.Element 提供了 171 个方法，可谓相当丰富，根据操作类型基本可以分为查询、DOM 操作、样式操作、对齐、尺寸大小、定位、拖放、滚动、键盘、动画和杂项 9 类。

2. 查询

Element 对象一共有以下 13 个查询操作：

- ❑ contains：判断元素是否包含另一个元素。
- ❑ .child：从元素的直接子元素中选择与选择符匹配的元素。
- ❑ down：选择与选择符匹配的元素的子元素。
- ❑ first：选择元素第一个子元素。
- ❑ findParent：查找与简单选择符匹配的元素的父元素。
- ❑ findParentNode 或 up：查找与简单选择符匹配且在规定深度内的元素的父元素。
- ❑ is：判断元素是否匹配选择符。
- ❑ last：选择元素的最后一个子元素。
- ❑ next：选择与元素同层的下一个元素。
- ❑ parent：返回元素的父元素。
- ❑ prew：选择与元素同层的上一元素。
- ❑ query：根据选择符获取元素子元素。
- ❑ select：根据选择符获取元素的子元素集合。

3.DOM 操作

Element 对象一共有以下 12 个 DOM 操作方法：

- appendTo：将当前元素追加到指定元素中。
- appendChild：在当前元素中追加元素。
- createChild：在元素中插入由 DomHelper 对象创建的元素。
- insertAfter：将元素插入到指定元素之后。
- insertBefore：将元素插入到指定元素之前。
- insertFirst：在当前元素插入（或创建）一个元素作为当前元素的第一个子元素。
- insertSibling：在当前元素前或后插入（或创建）元素（同层）。
- insertHtml：在当前元素内插入 HTML 代码。
- remove：移除当前元素。
- replace：使用当前元素替换指定的元素。
- replaceWith：使用创建的元素替换当前元素。
- wrap：创建一个元素，并将当前元素包裹起来。

4. 样式操作

Element 对象一共有以下 18 个样式操作方法：

- addCls：增加 CSS 样式到元素，重复的样式会自动过滤。
- applyStyles：设置元素的 style 属性，比 setStyle 灵活。
- boxWrap：使用 boxMarkup 定义的 HTML 代码包装元素。
- clearOpacity：清理不透明度（opacity）设置。
- getColor：返回 CSS 颜色属性的值，返回值为 6 位数组的 16 进制颜色值。
- getMargin ：返回一个具有 top、left、right 和 bottom 属性的对象，属性值为相应元素 margin 值。
- getStyle：返回元素的当前样式和计算样式。
- getStyles：返回一个包含元素样式的对象，对象属性为样式名称、值为样式值。
- getStyleSize：返回元素的样式尺寸。
- parseStyles：将样式字符串转换为以样式名称为属性名称，样式值为属性值的对象。
- parseBox：将数字或字符串转换为对象样式的 margin 值。
- radioCls：为当前元素添加样式，并删除其兄弟节点的元素。
- removeCls：移除元素的样式。
- replaceCls：替换元素的样式。
- set：设置元素属性。
- setOpacity：设置不透明度。
- setStyle：为元素设置样式。
- unitizeBox：将表示 margin 大小的对象转换为字符串。

5. 对齐

Element 对象提供了以下 4 个对齐方法：

- alignTo：将当前元素对齐到另外一个元素。定位位置的选择是基于对齐元素的 tl（左上角）、t（顶边中心）、tr（右上角）、l（左边中心）、c（元素中心）、r（右边中心）、bl（左下角）、b（底线中心）和 br（右下角）这 9 个定位点。
- anchorTo：当窗口调整大小时，将当前元素锚到指定元素并重新调整。
- getAlignToXY：返回当前元素对齐指定元素是的 xy 坐标。
- removeAnchor：移除当前元素的任何锚定位。

6. 尺寸大小

Element 对象提供了以下 20 个操作元素尺寸大小的方法：

- clip：存储元素当前的 overflow 设置并裁剪溢出。
- getBorderWidth：返回边界宽度。
- getComputedWidth：返回计算出来的 CSS 宽度。
- getComputedHeight：返回计算出来的 CSS 高度。
- getDocumentWidht：返回文档宽度。
- getDocumentHeight：返回文档高度。
- getFrameWidth：返回合计了 padding 和 border 的宽度。
- getHeight：返回 offsetHeight 值。
- getPadding：返回 padding 的宽度。
- getSize：返回元素的大小。
- getTextWidth：返回文本宽度。
- getViewportHeight：返回窗口的可视高度。
- getViewporWidth：返回窗口的可视宽度。
- getViewSize：返回元素可以用来放置内容的区域大小。
- getWidth：返回 offsetWidth 值。
- isBorderBox：主要用来检测盒子模型，与 IE 6 和 IE 7 有关。
- setHeight：设置元素高度。
- setSize：设置元素大小。
- setWidth：设置元素宽度。
- unclip：在 clip 被调用前将剪裁值（溢出）还原为原始值。

7. 定位

Element 对象提供了以下 33 个定位方法：

- clearPositioning：当文档加载完成后，清理定位回到默认值。
- fromPoint：返回指定坐标的顶层元素。
- getBottom：返回右下角的 Y 坐标。
- getBox：返回一个包含元素位置的对象，对象包括元素左上角的坐标值、右下角的坐

标值、宽度和高度。

- getCenterXY：返回元素的当前坐标。
- getLeft：返回左边的 x 坐标。
- getOffsetsTo：返回元素相对于指定元素的偏移值。
- getPageBox：返回一个包含元素位置的对象。
- getPositioning：返回一个包含 CSS 位置属性的对象。
- getRegion：返回元素所在区域。
- getRight：返回元素的右边 X 坐标。
- getTop：返回元素顶部的 Y 坐标。
- getViewRegion：返回元素的内容区域。
- getX：返回元素当前的 X 坐标。
- getXY：返回元素当前的 XY 坐标。
- getY：返回元素当前的 Y 坐标。
- move：移动元素。
- moveTo：将元素移动到指定的 XY 坐标上。
- position：初始化元素的位置。
- setBottom：设置元素的 bottom 样式。
- setBounds：设置元素的位置和大小。
- setBox：设置元素的位置大小。
- setLeft：设置元素坐标的 X 坐标。
- setLeftTop：设置元旦左上角坐标。
- setLocation：设置元素位置。
- setPositioning：设置元素位置。
- setRegion：在指定区域设置元素的位置和大小。
- setRight：设置元素 right 的样式值。
- setTop：设置元素的顶部 Y 坐标。
- setX：设置元素的 X 坐标。
- setXY：设置元素的位置。
- setY：设置元素的 Y 坐标。
- translatePoints：转换元素的页面坐标为 CSS 的 left 和 top 值。

8. 滚动操作

Element 对象提供了 5 个滚动操作的方法：

- getScroll：返回元素当前滚动条的位置。
- isScrollable：如果元素允许滚动，返回 true。
- scroll：滚动到指定位置。
- scrollIntoView：将元素滚动到指定容器的可视区域。

- scrollTo：将元素滚动到指定的点。

9. 杂项

Element 对象还要以下 21 个方法：

- addKeyMap：为元素创建一个 KeyMap 对象。
- addKeyListener：为 KeyMap 绑定事件。
- blur：使元素失去焦点。
- center：使元素居中。
- clean：清理空白的文本节点。
- createProxy：为这个元素创建一个代理元素。
- createShim：为元素创建一个 iframe 垫片以保证选择或其他对象跨越时是可见的。
- focus：使元素获得焦点。
- getLoader：返回 ElementLoader[⊖]对象。
- getOrientation：返回当前窗口的方法。是根据窗口的高度和宽度出来的，返回值是 portrait（竖向）和 landscape（横向）。
- getValue：如果元素有 value 属性，返回其值。
- normalize：将 CSS 属性中的连接符号去掉，例如将“border-width”转换为 borderWidth。
- on：绑定事件。
- load：直接调用 ElementLoader 的 load 方法为元素加载内容。
- mark：遮障当前元素，屏蔽用户操作。
- repaint：强迫浏览器重新绘画元素。
- serializeForm：序列化表单为 URL 编码的字符串。
- un：移除事件。
- unmark：移除遮障。
- unselectable：禁用文本选择。
- update：更新元素的 innerHTML 属性。

6.5　获取元素集合：Ext.CompositeElementLite 与 Ext.CompositeElement

6.5.1　使用 Ext.select 获取元素集合

Ext.select 的作用是通过选择符获取元素集合，其语法如下：

```
Ext.select(selector, unique, root);
```

其中：

- selector：选择符。注意，选择符内不能包含 id 选择符。
- unique：是否为每一个元素创建唯一的 Element 对象。默认值是 false，表示不创建，返

⊖ 相关信息请阅读 7.1.5 节。

回 CompositeElementLite 对象。如果设置为 ture，表示创建，返回 CompositeElement 对象。

- root：查询开始的节点或节点 id。默认值是 document。建议在知道根节点的情况设置根节点，以加快查询速度。

例如，页面中存在以下代码：

```
<div id="root">
        <a id="id1">1</a>
        <a id="id2">2</a>
        <a id="id3">3</a>
        <a id="id4">4</a>
</div>
```

如果要选择 id 为 root 的 div 内的 a 元素，其代码如下：

```
var els1=Ext.select('a',false,'root'); //'root' 为 div 的 id
console.log(els1.elements);
```

代码运行后可在控制台看到以下结果：

```
[a#id1, a#id2, a#id3, a#id4]
```

如果先获取根节点，再使用根节点的 select 方法获取根节点下的元素集合，其代码如下：

```
var el=Ext.get("root");
var els2=el.select('a');
console.log(els2.elements);
```

代码运行后在控制会看到与 Ext.select 查询一样的结果。

结果一样的原因是 Ext.select 方法其实是 Element 的 select 方法的简写，在 Composite-Element.js 文件中可看到其定义：

```
Ext.select = Ext.Element.select;
```

而 Ext.dom.Element 的 select 方法代码如下：

```
Ext.dom.Element.select = function(selector, unique, root){
    var elements;
    if(typeof selector == "string"){
        elements = Ext.dom.Element.selectorFunction(selector, root);
    }else if(selector.length !== undefined){
        elements = selector;
    }else{
    // 省略抛出异常代码
    }
    return (unique === true) ? new Ext.CompositeElement(elements) : new Ext.
        CompositeElementLite(elements);
};
```

代码与 6.1.2 节的 select 方法除了参数多了一个，返回结果不同外，代码基本是一样的，在此就不再赘述了。

6.5.2　Ext.dom.CompositeElement 与 Ext.dom.CompositeElementLite 的区别

首先看看 Ext.CompositeElement 的源代码：

```
Ext.define('Ext.dom.CompositeElement', {
    alternateClassName: 'Ext.CompositeElement',

    extend: 'Ext.dom.CompositeElementLite',
    getElement: function(el) {
        return el;
    },
    transformElement: function(el) {
        return Ext.get(el);
    }

}, function() {
    // 省略 Ext.select 定义代码
});
```

从类定义内的 extend 定义可知 CompositeElement 继承自 .CompositeElementLite，并且复写了 getElement、transformElement 方法。

与 CompositeElementLite 的 getElement 方法和 transformElement 方法代码对比一下：

```
getElement: function(el) {
    return this.el.attach(el);
},
transformElement: function(el) {
    return Ext.getDom(el);
},
```

可以发现在 CompositeElementLite 对象返回的是 HTMLElement 对象。而 Ext.CompositeElement 对象返回的是使用 Ext.get 方法所返回的 Element 对象。而这也是两个之间的主要区别。

了解了 CompositeElementLite 与 .CompositeElement 的区别后，**在平时的应用中为了节省内存，提高性能，应该多使用 Ext.CompositeElementLite 对象**。

6.5.3　操作元素集合

CompositeElementLite 对象提供了以下 13 个方法：

- add：增加元素到 CompositeElementLite 对象。
- clear：清除所有元素。
- contains：如果 CompositeElementLite 对象包含指定元素，返回 true。
- each：枚举元素。
- fill：清除所有元素，并增加指定的元素到 CompositeElementLite 对象。
- filter：使用选取符过滤元素。
- first：返回第一个元素。

- getCount：返回元素的总数。
- indexOf：返回指定元素在 CompositeElementLite 对象的索引值。
- item：根据索引返回元素。
- last：返回最后一个元素。
- removeElement：移除指定的元素。
- replaceElement：使用指定的元素替换 CompositeElementLite 对象中的元素。

CompositeElement 对象继承于 CompositeElementLite 对象，但没有定义自己的方法，因而只有从 CompositeElementLite 对象继承的方法。

6.6 综合实例：可折叠的面板 Accordion

（1）功能描述

Accordion 主要功能就是单击标题可切换面板，简单来说，就是隐藏已显示的面板，显示当前的标题下的面板。

（2）实现代码

在实现代码前，首先要考虑的是每个面板的 HTML 代码格式，只有固定了格式，才能方便地通过选择器选择面板元素，然后对其操作。

Accordion 面板的主要有以下两个特征：

- 标题：用来显示面板标题，需显示手型图标，允许单击操作。
- 内容面板：显示内容，由标题控制面板的显示。

根据以上的特征，可简单地将面板的格式定义如下：

```
<a> 标题 </a>
<div>
    <p> 内容 </p>
</div>
```

现在要考虑的是面板标题的单击事件，面板的标题可能会很多，一个个为它们绑定事件比较麻烦。使用 DOM 的事件传播机制可以很好地解决这个问题，只要在面板的父节点绑定单击事件，就可捕获面板标题的单击事件了。

实现的难点已经解决，现在可以实现代码了。使用模板新建一个名称为 Accordion.html 的页面文件，先加入以下样式代码：

```
.accordion:{}
.accordion  {width: 300px;border: 1px solid black;}
.accordion div {background-color: #eee;}
.accordion p {
    margin-bottom : 10px;
    border: none;
    text-decoration: none;
    font-weight: bold;
    font-size: 10px;
    margin: 0px;padding: 10px;
```

```
}
.titlebar {
    cursor:pointer;
    display:block;
    padding:5px;
    margin-top: 0;
    text-decoration: none;
    font-weight: bold;
    font-size: 12px;
    color: black;
    background-color: #00a0c6;
    border-top: 1px solid #FFFFFF;
    border-bottom: 1px solid #999;
}
.titlebar:hover {background-color: white;}
.titlebar .selected {color: black;background-color: #80cfe2;}
```

接着加入以下 HTML 代码：

```
<h1>第 6 章 可折叠的面板 Accordion</h1>
 <div class="accordion"  id="MyAccordion">
     <a class="titlebar">用户 </a>
     <div>
        <p>
                <a>张三 </a><br/>
                <a>李四 </a>
        </p>
     </div>
<a  class="titlebar">采购 </a>
<div >
    <p>
            <a>进货 </a><br/>
            <a>库存 </a>

    </p>
</div>
<a  class="titlebar">报表 </a>
<div>
    <p>
            <a>月度 </a><br/>
            <a>季度 </a><br/>
            <a>年度 </a>
    </p>
</div>
<a  class="titlebar">系统管理 </a>
<div>
        <p>
                <a>用户管理 </a><br/>
                <a>系统设置 </a>
        </p>
    </div>
</div>
```

接下来为 id 为 MyAccordion 的绑定单击事件，通过事件传递机制来处理面板的单击动作：

```
var el=Ext.get("MyAccordion");
el.on("click",function(e,el){
    var hideEl=Ext.query("#MyAccordion>div{display!=none}");
    if(hideEl){
            hideEl[0].setAttribute("style","display:none");
    }
    var showEl=Ext.query("+div",el);
    if(showEl){
            showEl[0].setAttribute("style","");
    }
},this);
```

在单击事件中，使用了一个复合选择符“#MyAccordion>div{display!=none}”查询当前显示的面板。“#MyAccordion>”查询可保证获取的 div 元素是面板的 div，而不是面板内容里的 div。而“{display!=none}”选择则确保选择的是正在显示的面板。根据 Accordion 的特征，一次只会显示一个面板，因而可以直接访问元素数组的第一个元素，将其隐藏。

因为内容面板是紧跟在标题后的 div，而 el 是触发事件的标题元素，因而使用“+div”就可选择紧跟其后的 div 元素，然后将其显示出来即可。

接着要对面板初始化，将第一个面板之后的面板设置为隐藏：

```
var els=Ext.query("#MyAccordion>a+div");
for(var i=1;ln=els.length,i<ln;i++){
    els[i].setAttribute("style","display:none");
}
```

组合查询符“#MyAccordion>a+div”的作用是选择出所有内容面板的 div 元素，然后通过循环初始化它们的显示。初始状态，第一个内容面板是显示状态的，所以 for 循环是从 1 开始的。

（3）页面效果

在浏览器中打开页面，并单击“报表”面板将看到如图 6-2 的效果。

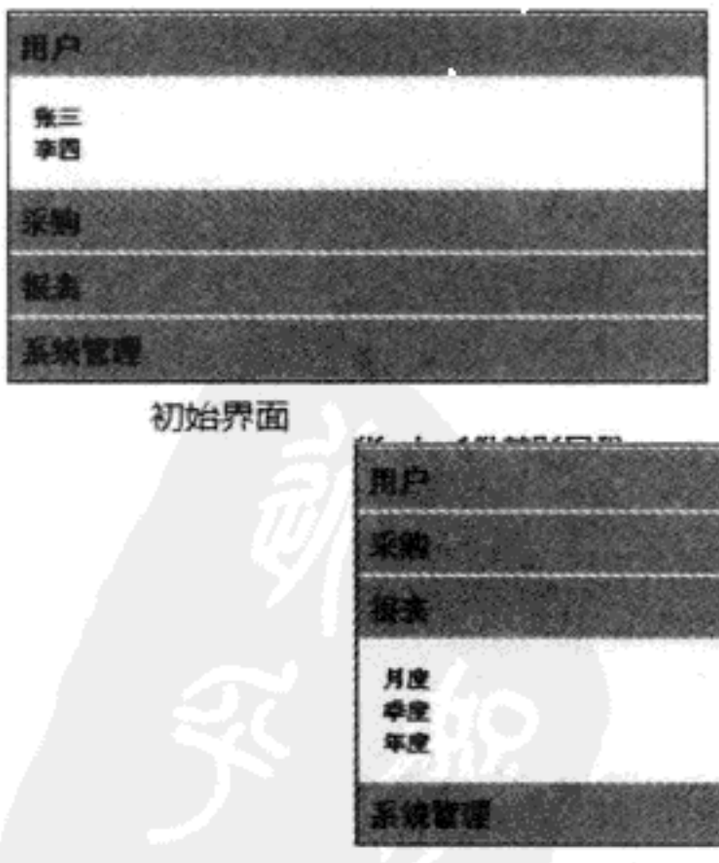


图 6-2　可折叠的面板 Accordion 的页面效果

6.7　本章小结

本章主要讲述了 Ext JS 的选择器和 DOM 操作，这些都是 Ext JS 组件工作的基础，组件基本是通过 Element 对象来操作节点的，因而充分了解 Element 对象及选择器，对理解组件的操作是非常有帮助的。

第7章　数据交互

Web 2.0 的兴起，皆因 Ajax，有了它，可以实现一种无刷新的页面技术，从而增进用户的体验效果，也让实现更多有趣的功能成为一种可能。更重要的是，它是 Web 应用程序数据交换桥梁，有了 Ajax，使数据交换更容易，更轻松。不过，各浏览器提供的 Ajax 技术不尽相同，因而作为一个跨浏览器的框架，当然要对其进行封装。本章主要讲述的内容就是 Ext JS 是如何封装 Ajax 的，以及提供了一些可用的便于数据交换的对象。

7.1　数据交互基础

7.1.1　Ajax 概述

Ajax 是“Asynchronous JavaScript and XML”（异步的 JavaScript 与 XML 技术）的简写。它不是什么新东西，早在 IE 5 年代就已存在了，不过因为只支持 IE，不怎么流行而已。后来随着越来越多的浏览器（其实浏览器品种不算多）的支持，尤其是 Google Maps 的出现，才使其流行起来，从而出现了一次互联网上的 Web 2.0 革命。

Ajax 使用的是 XMLHttpRequest 对象，看对象名称就知道，其本来目的是加载 XML 文件的，但是，XML 文件的繁杂和难于处理，使 JOSN 成为了事实上的标准。

有关 Ajax 的信息在互联网上可以说是“汗牛充栋”，笔者也不在这里画蛇添足了。不过，这个是必须要了解清楚的，不然你在使用 Ext JS 框架时，还在使用页面跳转刷新的方式提供数据，那就太“落伍”了。

言归正传，本章开头说了，因浏览器处理 XMLHttpRequest 对象的不同，因而要对其进行封装，在 Ext JS 中，封装该类的是 Ext.data.Connection 对象。

7.1.2　封装 Ajax：Ext.data.Connection 与 Ext.Ajax

你没看错，封装 Ajax 的类不是 Ext.Ajax，而是 Connection 对象。Ajax 对象只是 Connection 对象的一个子类，其定义只有几行代码：

```
Ext.define('Ext.Ajax', {
    extend: 'Ext.data.Connection',
    singleton: true,
    autoAbort : false
});
```

根据 singleton 的定义，可以知道 Ajax 对象是一个单件模式的类，根据第 4 章介绍的 singleton 处理器的功能，它返回的对象是一个实例对象，因而，可以说，Ajax 对象就是

Connection 对象的一个实例。这样做的原因在 API 里说清楚了，是为应用中不同的请求共享一些公共设置，例如 timeout、disableCaching 或 method 等公共属性，还有就是 beforrequest、requestcomplete 和 requestexception 等内部事件。笔者认为 3 个内部事件是主要原因，因为使用 Class 对象的定义方式，很难将 Observable 对象混合进 Ajax 对象，而采用现在这种方式，通过 mixins 方式就可以轻松实现了。

Connection 对象的构造函数很简单，主要是定义了一个 requests 对象，用以保存所有请求。

Connection 对象中最重要的当然是 request 方法，这是 Ajax 的请求入口，其源代码如下：

```
request : function(options) {
    options = options || {};
    var me = this,
        scope = options.scope || window,
        username = options.username || me.username,
        password = options.password || me.password || '',
        async,
        requestOptions,
        request,
        headers,
        xhr;

    if (me.fireEvent('beforerequest', me, options) !== false) {
        requestOptions = me.setOptions(options, scope);
        if (this.isFormUpload(options) === true) {
            this.upload(options.form, requestOptions.url, requestOptions.data,
                options);
            return null;
        }
        if (options.autoAbort === true || me.autoAbort) {
            me.abort();
        }
        xhr = this.getXhrInstance();
        async = options.async !== false ? (options.async || me.async) : false;
        if (username) {
            xhr.open(requestOptions.method, requestOptions.url, async, username,
                password);
        } else {
            xhr.open(requestOptions.method, requestOptions.url, async);
        }
        headers = me.setupHeaders(xhr, options, requestOptions.data, request-
            Options.params);
        request = {
            id: ++Ext.data.Connection.requestId,
            xhr: xhr,
            headers: headers,
            options: options,
            async: async,
            timeout: setTimeout(function() {
                request.timedout = true;
                me.abort(request);
            }, options.timeout || me.timeout)
        };
        me.requests[request.id] = request;
```

```
            if (async) {
                xhr.onreadystatechange = Ext.Function.bind(me.onStateChange, me,
                    [request]);
            }
            xhr.send(requestOptions.data);
            if (!async) {
                return this.onComplete(request);
            }
            return request;
        } else {
            Ext.callback(options.callback, options.scope, [options, undefined,
                undefined]);
            return null;
        }
    },
```

代码首先触发 beforerequest 事件，如果返回 false，代码会停止执行，并使用 Ext 的 callback 方法调用回调函数。

首先要处理的是使用 setOptions 方法处理配置对象的提交参数、提交地址、提交方式以及提交的数据类型，最后返回以下格式的数据给 requestOptions 变量：

```
{
    url: url,// 提交参数的提交地址
    method: method, // 提交方式
    data: data // 提交参数或数据类型
}
```

如果是使用 Ajax 上传文件，则执行 upload 方法，其代码如下：

```
upload: function(form, url, params, options){
    form = Ext.getDom(form);
    options = options || {};

    var id = Ext.id(),
            frame = document.createElement('iframe'),
            hiddens = [],
            encoding = 'multipart/form-data',
            buf = {
                target: form.target,
                method: form.method,
                encoding: form.encoding,
                enctype: form.enctype,
                action: form.action
            }, hiddenItem;
    Ext.fly(frame).set({
        id: id,
        name: id,
        cls: Ext.baseCSSPrefix + 'hide-display',
        src: Ext.SSL_SECURE_URL
});

document.body.appendChild(frame);
if (document.frames) {
   document.frames[id].name = id;
```

```
}

Ext.fly(form).set({
    target: id,
    method: 'POST',
    enctype: encoding,
    encoding: encoding,
    action: url || buf.action
});

if (params) {
    Ext.iterate(Ext.Object.fromQueryString(params), function(name, value){
        hiddenItem = document.createElement('input');
        Ext.fly(hiddenItem).set({
            type: 'hidden',
            value: value,
            name: name
        });
        form.appendChild(hiddenItem);
        hiddens.push(hiddenItem);
    });
}

Ext.fly(frame).on('load', Ext.Function.bind(this.onUploadComplete, this,
    [frame, options]), null, {single: true});
form.submit();

Ext.fly(form).set(buf);
Ext.each(hiddens, function(h) {
        Ext.removeNode(h);
    });
},
```

代码一开始就创建了一个 iframe，说明文件上传是提交到 iframe 的，而不使用 XMLHttp-Request 对象。在提交前要先使用 buf 对象记录下表单的原始设置。

接着要做的事就是修改表单中提交目标为 iframe，然后将提交参数的变成表单的隐藏输入框，还要把 iframe 的加载事件绑定到 onUploadComplete 方法。最后提交表单。

表单提交后要将 buf 对象中记录的表单原始设置恢复给表单，还要移除隐藏输入框。

继续看看提交完成后会做什么操作，onUploadComplete 的代码如下：

```
onUploadComplete: function(frame, options){
    var me = this,
        response = {
            responseText: '',
            responseXML: null
        }, doc, firstChild;

    try {
        doc = frame.contentWindow.document || frame.contentDocument || window.
            frames[id].document;
        if (doc) {
            if (doc.body) {
                if (/textarea/i.test((firstChild = doc.body.firstChild || {}).
```

```
                    tagName)) {
                    response.responseText = firstChild.value;
                } else {
                    response.responseText = doc.body.innerHTML;
                }
            }
            response.responseXML = doc.XMLDocument || doc;
        }
    } catch (e) {
    }

    me.fireEvent('requestcomplete', me, response, options);

    Ext.callback(options.success, options.scope, [response, options]);
    Ext.callback(options.callback, options.scope, [options, true, response]);

    setTimeout(function(){
        Ext.removeNode(frame);
    }, 100);
},
```

响应对象 response 会包含 responseText 和 responseXML 两种格式的数据，因而使用者可根据自己的定义选择使用文本格式数据还是 XML 格式的数据。

接着是触发 requestcomplete 事件。同时定义了 success 和 callback 回调函数的要注意了，这两个回调函数都会被调用，因而只能选择其中一种方式，不然代码可能会执行两次。

最后还要延时删除 iframe。

回到 requet 方法的代码，如果设置配置对象的 autoAbort 为 true，或已经设置了 Ajax 的 autoAbort 属性为 true，则会使用 abort 方法清理那些状态不是 0 或 4 的请求，也就是说该属性会中止那些已经发送、但还没完成的求。因而，设置这个参数要考虑清楚，不然会出现同时有多个请求，但是只响应了一个情况。

接着是使用 getXhrInstance 方法创建 XMLHttpRequest 对象的实例，其代码如下：

```
getXhrInstance: (function(){
    var options = [function(){
        return new XMLHttpRequest();
    }, function(){
        return new ActiveXObject('MSXML2.XMLHTTP.3.0');
    }, function(){
        return new ActiveXObject('MSXML2.XMLHTTP');
    }, function(){
        return new ActiveXObject('Microsoft.XMLHTTP');
    }], i = 0,
    len = options.length,
    xhr;

for(; i < len; ++i) {
    try{
        xhr = options[i];
        xhr();
        break;
    }catch(e){}
```

```
        }
        return xhr;
    })(),
```

该函数使用了闭包方式，因而在 Connection 对象创建时已经设置好了使用哪种方式返回 XMLHttpRequest 对象的实例。无论怎么说，IE 都是麻烦制造者，它居然有 3 种创建 XMLHttpRequest 对象的方式，用现在时髦的网络语言说：坑爹啊！在 IE 6 时代，只要一个判断语句就可以创建了 XMLHttpRequest 对象了，现在居然要使用数组方式逐个去判断，如果 IE 不断升级，这数组得多长啊！还好，微软没那么勤快。

接着取异步值。

如果存在用户名（username），需要为请求添加用户名和密码。

接着是使用 setupHeaders 方法设置请求的头信息。

接着是创建一个请求对象，以 id 作为属性名称保存在 requests 对象里，以便中止。

如果使用异步方式（默认是异步方式），也就是可在下载完成之前向调用程序返回控制权，因而必须为 XMLHttpRequest 对象绑定一个状态改变事件，以便下载完成后触发回调函数，不然不知道何时才执行回调函数，在这里绑定 onStateChange 方法，其代码如下：

```
onStateChange : function(request) {
    if (request.xhr.readyState == 4) {
        this.clearTimeout(request);
        this.onComplete(request);
        this.cleanup(request);
    }
},
```

在状态改变为 4 时，也就是下载完成状态，清理任务，调用 onComplete 方法，其代码如下：

```
onComplete : function(request) {
    var me = this,
        options = request.options,
        result = me.parseStatus(request.xhr.status),
        success = result.success,
        response;

    if (success) {
        response = me.createResponse(request);
        me.fireEvent('requestcomplete', me, response, options);
        Ext.callback(options.success, options.scope, [response, options]);
    } else {
        if (result.isException || request.aborted || request.timedout) {
            response = me.createException(request);
        } else {
            response = me.createResponse(request);
        }
        me.fireEvent('requestexception', me, response, options);
        Ext.callback(options.failure, options.scope, [response, options]);
    }
    Ext.callback(options.callback, options.scope, [options, success, response]);
```

```
        delete me.requests[request.id];
        return response;
    },
```

首先使用 parseStatus 检查响应状态，状态为 200 到 300（不包括 300）或 304 时，为成功，这时 success 值为 true。了解这个很重要，通过这个值，就可以知道为什么有时候明明是成功的，却都转到失败去了。

如果成功，使用 createResponse 方法创建响应对象，其代码如下：

```
createResponse : function(request) {
var xhr = request.xhr,
    headers = {},
    lines = xhr.getAllResponseHeaders().replace(/\r\n/g, '\n').split('\n'),
    count = lines.length,
    line, index, key, value, response;

while (count--) {
    line = lines[count];
    index = line.indexOf(':');
    if(index >= 0) {
        key = line.substr(0, index).toLowerCase();
        if (line.charAt(index + 1) == ' ') {
            ++index;
        }
        headers[key] = line.substr(index + 1);
    }
}

request.xhr = null;
delete request.xhr;

response = {
    request: request,
    requestId : request.id,
    status : xhr.status,
    statusText : xhr.statusText,
    getResponseHeader : function(header){ return headers[header.toLowerCase
        ()]; },
    getAllResponseHeaders : function(){ return headers; },
    responseText : xhr.responseText,
    responseXML : xhr.responseXML
};

xhr = null;
return response;
},
```

代码会通过 XMLHttpRequest 对象的 getAllResponseHeaders 方法返回所有响应的头信息，然后记录在 headers 对象中，接着删除 XMLHttpRequest 对象，并创建一个包含以下成员的响应对象：

- request：请求对象。
- requestId：请求对象的 id。

- status：XMLHttpRequest 对象的响应状态码。
- statusText：XMLHttpRequest 对象的响应状态文本信息。
- getResponseHeader：根据属性名称返回其头信息。
- getAllResponseHeaders：所有响应的头信息。
- responseText：如果请求返回的是非 XML 格式数据，可通过该属性获取响应信息。
- responseXML：如果请求返回的是 XML 格式的数据，可通过该属性获取响应信息。

最后是将变量 xhr 设置为 null，返回响应对象。

回到 onComplete 方法，创建响应对象后，先触发 requestcomplete 事件，接着执行配置对象的 success 属性指向的函数。

如果响应不成功是异常、超时或中止等情况，则通过 createException 方法创建一个异常对象；否则，创建通过一个 createResponse 方法响应对象。接着触发 requestexception 事件，并执行配置对象中定义的 failure 函数。

最后执行配置对象中定义的 callback 函数，并在 requests 对象中删除当前请求。

回到 onStateChange 方法，最后使用 cleanup 方法将 request 对象中的 XMLHttpRequest 对象设置为 null，并使用 delete 语句删除，主要作用是释放内存。

回到 requet 方法，接着要做的就是发送请求了。如果是同步的，会等到请求完成后再执行下面代码，也就是执行 onComplete 方法。如果是异步的，则直接返回。

至此，一个请求就完成了。

7.1.3 使用 Ajax

使用 Ajax 很简单，语法如下：

```
Ext.Ajax.request(options);
```

其中 options 为配置对象，一般包括以下配置项：

- url：请求地址。如果不设置，使用 Ajax 对象设置的默认 url 地址。
- params：可选参数，为提交的参数，可以是字符串、对象或函数。
- method：可选参数，为提交方式，值可以为 GET 或 POST，全为大写字母。
- callback：可选参数，为回调函数，如果使用 success 和 failure 就不要使用该设置。函数依次序可接收请求的配置对象、是否成功（true 表示成功）和响应对象[⊖]等 3 个参数。
- success：可选参数，为请求成功后执行的函数。从 7.1.2 节可以知道，请求的状态码为 200 到 300（不包含 300）或 304 时才会执行该属性指定的函数。函数依次序可接收响应对象和配置对象两个参数。
- failure：可选参数，为请求失败时执行的函数。函数可接收的参数与 success 一样。
- scope：可选参数，可通过该属性设置回调函数的作用域。
- timeout：可选参数，为请求的超时时间，默认值是 30 秒，单位是微秒。
- form：可选参数，为要提交的表单的 id、Element 对象或 HTMLElement 对象。

⊖ 相关信息可阅读 7.1.2 节。

- ❑ isUpload，可选参数，布尔值，是否上传文件。只有设置了 form 属性该属性才有效。为 true 时，说明要上传文件。
- ❑ headers：可选参数，为请求的头信息对象。
- ❑ xmlData：可选参数，要提交的 XML 数据。
- ❑ jsonData：可选参数，要提交的 JSON 数据。
- ❑ disableCaching：可选参数，布尔值。默认值为 false，Ajax 会在发送请求时添加一个“dc_”的参数，以确保链接的唯一性，从而避免使用浏览器缓存中的数据。如果为 true，则不添加额外参数，而极有可能使用浏览器缓存中的数据，所以设置该值时要三思。

下面通过一个示例来学习如何使用 Ajax 对象。

（1）功能描述

本示例将通过两个按钮分别使用 Ajax 加载非 XML 文件和 XML 文件，然后在页面中显示响应对象的信息。因为 success 和 failure 只是将 callback 的响应根据请求结果拆分出来的，因而示例中只使用 callback 函数显示结果。

（2）实现代码

首先使用示例模板页创建一个文件名为 7-1.html 的页面文件，然后先添加两个按钮，分别用来加载非 XML 文件和 XML 文件：

```
<input id="button1" type="button" value="加载非 XML 文件 ">  
<input id="button2" type="button" value="加载 XML 文件 ">
```

接着定义一个 div 用来显示响应对象：

```
<div id="ajaxinfo"></div>
```

下面要做的是定义一个回调函数 callback，两次加载都使用该回调函数对响应对象进行处理：

```
var callback=function(o,s,r){
var html=[];
html.push("<p>success: "+s);
console.log(r["responseText"])
for(var i in r){
   if(r.hasOwnProperty(i)){
   switch(i){
          case "getAllResponseHeaders":
                var d=r['getAllResponseHeaders']();
                for(var j in d){
                if(d.hasOwnProperty(j)){
html.push("<p>response[getAllResponseHeaders]["+j+"]: "+d[j]);
                }
                }
   break;
   case "getResponseHeader":
          break;
   case "responseXML":
   var doc=r[i];
      if(doc){
```

```
var t="";
        if(doc.xml)
        t=Ext.htmlEncode(r[i].xml);
        else
        t=Ext.htmlEncode((new XMLSerializer()).serializeToString(doc));
        html.push("<p>response["+i+"]: "+t);

        }
        break;
case "options" :
            break;
        default:
        html.push("<p>response["+i+"]: "+r[i]);

        }
    }
}
Ext.getDom("ajaxinfo").innerHTML=html.join("");
}
```

代码没有处理配置对象这个参数，因为其实它就是你定义请求时的配置对象。这里使用了一个数组来保存显示信息。

首先加入数组的是第二个参数 success 的值。

接着是处理响应对象。因为 options 属性就是配置对象，getResponseHeader 是根据头信息的属性获取其值，在 getAllResponseHeaders 信息中已包含，因而省略了不显示。因为 getAllResponseHeaders 在创建响应对象时是函数，因而要执行一次，然后再遍历返回的对象内的属性和属性值。加载 XML 文件时，返回的 responseXML 对象，不能直接输出文本，因而需要转换一下。其余的直接加入数组就行了。

余下的工作就是为两个按钮绑定单击事件了：

```
Ext.fly("button1").on("click",function(){
    Ext.Ajax.request({
            url:"test.js",
            callback:callback
    })
});
Ext.fly("button2").on("click",function(){
    Ext.Ajax.request({
            url:"test.xml",
            callback:callback
    })
});
```

很简单，配置对象除了 url 和回调函数，没其他的定义。

要使示例顺利运行，还要创建 test.js 和 testx.xml 两个文件。首先创建 test.js 文件，内容如下：

```
{
    success:true,
    msg:"ok"
}
```

最后创建 test.xml 文件，其代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<data>
    <success>true</success>
    <message>ok</message>
</data>
```

好了，可以测试示例了。

(3) 示例效果

在浏览器打开页面会看到两个按钮，单击“加载非 XML 文件”，会看到页面显示以下结果：

```
success: true
response[request]: [object Object]
response[requestId]: 1
response[status]: 200
response[statusText]: OK
response[getAllResponseHeaders][content-length]: 34
response[getAllResponseHeaders][date]: Sun, 22 May 2011 18:46:59 GMT
response[getAllResponseHeaders][x-powered-by]: ASP.NET
response[getAllResponseHeaders][server]: Microsoft-IIS/7.5
response[getAllResponseHeaders][etag]: "db59e935f18cc1:0"
response[getAllResponseHeaders][accept-ranges]: bytes
response[getAllResponseHeaders][last-modified]: Sun, 22 May 2011 09:06:46 GMT
response[getAllResponseHeaders][content-type]: application/x-javascript
response[responseText]: { success:true, msg:"ok" }
```

单击“加载 XML 文件”，会看到页面显示以下结果：

```
success: true
response[request]: [object Object]
response[requestId]: 2
response[status]: 200
response[statusText]: OK
response[getAllResponseHeaders][content-length]: 116
response[getAllResponseHeaders][date]: Sun, 22 May 2011 18:48:37 GMT
response[getAllResponseHeaders][x-powered-by]: ASP.NET
response[getAllResponseHeaders][server]: Microsoft-IIS/7.5
response[getAllResponseHeaders][etag]: "c5e0ea196018cc1:0"
response[getAllResponseHeaders][accept-ranges]: bytes
response[getAllResponseHeaders][last-modified]: Sun, 22 May 2011 09:10:31 GMT
response[getAllResponseHeaders][content-type]: text/xml
response[responseText]: true ok
response[responseXML]: <?xml version="1.0" encoding="UTF-8"?> <data> <success>true
    </success> <message>ok</message> </data>
```

对比一下两个结果，可以看到主要区别在“content-type”。还有一个特殊的地方，返回 XML 数据，在 responseText 中也会有数据，不过数据格式是文本，而不是 XML 对象。

注意 如果返回的数据是 JSON 数据，可以使用 JOSN 对象的 decode[⊖]方法将其转换为对象。

⊖ 相关信息请阅读 16.4.2 节。

7.1.4 跨域获取数据：Ext.data.JsonP

在5.6节综合示例中就涉及了跨域取数据的问题，示例中是使用创建script脚本去加载数据的，而在Web应用开发中，多少都会涉及跨域获取数据的问题，但让开发人员自己去处理这个问题，显然不太合适。例如，代理要跨域获取数据，这也需要封装，这就是JsonP对象。

在5.6节的示例中不能使用JsonP对象，笔者很懊恼，原因是它非要在提交参数后加上“callback=Ext.data.JsonP.callback1”（callback1是回调函数的名称），这样会造成去新浪获取数据时产生错误。

不知道是有意还是无意的，JsonP对象没有为script标记绑定事件，使其能在脚本下载完成时触发回调函数，而是需要开发人员自己根据提交的callback参数（或自己定义的回调参数变量名称），在返回的脚本中直接调用回调函数，例如，提交的代码包含以下与回调有关的参数：

```
callback=Ext.data.JsonP.callback1
```

则在返回的脚本中必须有以下代码：

```
Ext.data.JsonP.callback1({// 返回的数据 });
```

如果想修改回调函数提交参数名称及回调函数名，可在配置对象中定义callbackKey和callbackName两个配置项，例如：

```
Ext.data.JsonP.request({
    callbackKey: 'cb',
    callbackName: 'mycb',
    ...
});
```

则提交的参数会修改为：

```
cb=Ext.data.JsonP.mycb;
```

笔者建议修改callbackName配置项，不然默认回调函数名称将是“callback”加上由计数器累加产生的数字组成的名称，例如callback1、callback2或callback8等，这样你就要在服务器端提取名称后才能写回调函数，如果要加载脚本等静态文件，就基本不可能了。

除了以上两个配置外，JsonP对象的配置项与Ajax对象的配置项一样，因而这里就不再重复讲述了。

JsonP对象与Ajax对象一样，是一个单件模式的对象，可直接使用，无须创建实例（本身就是实例）。

7.1.5 为Element对象提供加载功能：Ext.ElementLoader

在Element对象中有一个load方法，是用来为元素加载内容的，其代码如下：

```
load : function(options) {
    this.getLoader().load(options);
```

```
        return this;
    },
```

方法 getLoader 的代码如下：

```
getLoader : function() {
    var dom = this.dom,
        data = Ext.core.Element.data,
        loader = data(dom, 'loader');

    if (!loader) {
        loader = Ext.create('Ext.ElementLoader', {
            target: this
        });
        data(dom, 'loader', loader);
    }
    return loader;
},
```

代码先在 Element 对象中看 ElementLoader 对象是否存在，如果不存在，则创建一个，然后将其保存到 Element 对象中。这里要注意，ElementLoader 对象的 target 指向了当前 Element 对象。

ElementLoader 对象的 load 方法会使用 Ajax 对象加载数据，如果加载成功会执行 Renderer 对象的 Html 方法进行渲染，其代码如下：

```
statics: {
    Renderer: {
        Html: function(loader, response, active){
            loader.getTarget().update(response.responseText, active.scripts === true);
            return true;
        }
    }
},
```

代码还使用 Element 对象本身定义的 update 方法刷新内容。

因 load 方法是使用 Ajax 对象加载内容的，因而其配置对象与 Ajax 对象的 request 方法是一样。不过，ElementLoader 对象也有一个很有特色的配置项 loadMask，其作用是决定在加载时是否在元素上显示正在加载的信息，默认值是 false，不显示。如果要显示，需要在配置对象中设置其值为 true 或要显示的文本。

7.1.6 为组件提供加载功能：Ext.ComponentLoader

在 Ext JS 3.3 中，Panel 对象这类可以加载内容的容器都会有一个 load 方法，在 Ext JS 4 中已经取消了，由 loader 对象或者 getLoader 方法取代。在所有 UI 组件的父类 AbstractComponent 的构造函数中，都可以找到以下代码：

```
me.loader = me.getLoader();
```

而 getLoader 方法代码如下：

```
getLoader: function(){
    var me = this,
        autoLoad = me.autoLoad ? (Ext.isObject(me.autoLoad) ? me.autoLoad : {url:
            me.autoLoad}) : null,
        loader = me.loader || autoLoad;

    if (loader) {
        if (!loader.isLoader) {
            me.loader = Ext.create('Ext.ComponentLoader', Ext.apply({
                target: me,
                autoLoad: autoLoad
            }, loader));
        } else {
            loader.setTarget(me);
        }
        return me.loader;

    }
    return null;
},
```

和 Element 对象的 getLoader 方法有点类似，不过这里创建的是 ComponentLoader 对象，而不是 ElementLoader 对象。

实际上，ComponentLoader 对象是 ElementLoader 对象的子类，它扩展了 ElementLoader 对象的 Renderer 对象，添加了 Data 和 Component 两个方法，其代码如下：

```
Renderer: {
Data: function(loader, response, active){
    var success = true;
    try {
        loader.getTarget().update(Ext.decode(response.responseText));
    } catch (e) {
        success = false;
    }
    return success;
},

Component: function(loader, response, active){
    var success = true,
        target = loader.getTarget(),
        items = [];

    //省略调试代码
    try {
        items = Ext.decode(response.responseText);
    } catch (e) {
        success = false;
    }
    if (success) {
        if (active.removeAll) {
            target.removeAll();
        }
        target.add(items);
    }
```

```
            return success;
        }
    }
```

从代码可以看到，Data 方法使用调用对象本身的 update 方法渲染数据，而 Component 方法则使用了添加组件的方式渲染新组件，这就为动态改变容器内的组件提供了很好的方法，例如，当前容器内显示的是用户信息的 Grid，当单击产品后，就可以使用 Component 方式更换容器内的 Grid，让它显示产品信息。粗体代码部分表示如果定义了 removeAll 属性为 true，则会使用 removeAll 方法清除容器内的组件。

下面用一个示例来实践一下这个效果。

（1）功能描述

示例主要在页面中创建一个自动加载面板，首先，面板会自动加载一个面板到面板内，然后通过两个按钮分别试验增加面板和重写面板两个功能。

（2）实现代码

首先使用模板页创建一个名称为 7-2.html 的页面文件。

要实现动态加载，需设置面板的 autoLoad 属性，不然不会创建 ComponentLoader 对象，也就是不能使用 loader 属性进行动态加载。下面开始创建一个面板：

```
var panel=Ext.create("Ext.Panel",{
    renderTo:Ext.getBody(),
    layout:"auto",
    height:500,
    width:400,
    autoLoad:{
            url:"Component.js",
            renderer:"component"
    },
    renderer:"component",
    tbar:[
            {text:"加载组件",scope:panel,handler:function(){
                    panel.loader.load({
                            url:"Component1.js",
                            renderer:"component"
                    })
            }},
            {text:"加载组件(removeAll)",scope:panel,handler:function(){
                    panel.loader.load({
                            url:"Component1.js",
                            renderer:"component",
                            removeAll:true
                    })
            }}
    ]
});
```

代码中，autoLoad 属性是要按照 Ajax 的配置对象进行配置，附加的 renderer 属性可决定使用何种方式渲染数据，这里会使用 Component 方法渲染数据。在面板上工具栏还定义了两个按钮，第一个按钮是使用 Component 方法添加组件，第二个按钮则是使用加载的组件替

换掉面板内原来的组件。

要完成示例还要创建两个文件，首先创建一个名称为 Component.js 的文件，在其中添加以下代码：

```
{xtype:'panel',height:100,width:200,html:" 原有的组件 "}
```

这是使用 JSON 格式定义的一个组件，也是 Component 方法能识别的格式。

最后创建 Component1.js 的文件，在其中添加以下代码：

```
{xtype:'panel',height:100,width:90,html:" 新增的组件 "}
```

两个面板的定义差不多，只是长度、宽度和内容不同。

（3）示例效果

在浏览器中打开页面，并依次单击第一个按钮和第二个按钮，将看到如图 7-1 所示的效果。

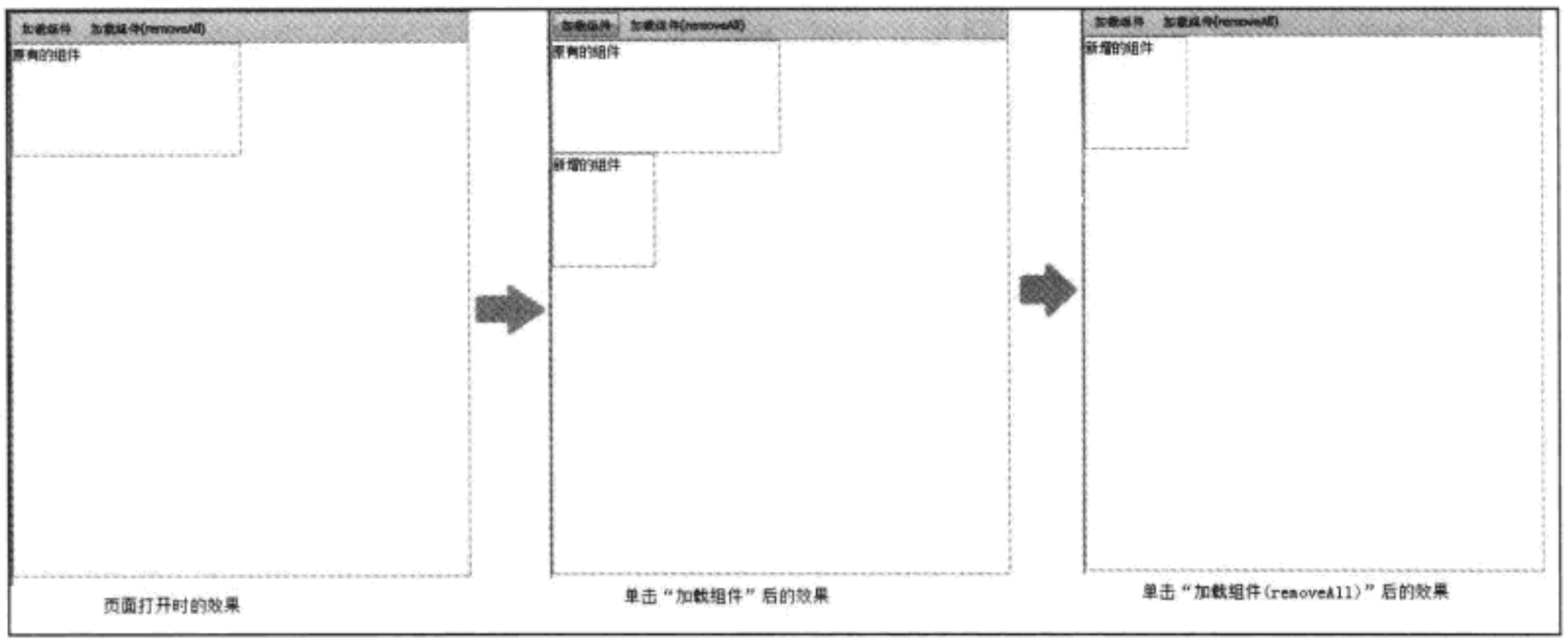


图 7-1 自己加载组件的示例效果

从图 7-1 中可以看到，程序按预想的情况实现了组件的动态加载。

不过，这个方法也有个 bug，就是不会清除纯文本，因为文本不是组件。例如面板开始时加载的是 HTML 代码不是组件，那么使用组件方式添加组件，它不会自动移除那些 HTML 代码，有兴趣可以自己尝试验证一下。

7.2 代理

7.2.1 代理概述

为什么需要代理？很简单，就是为了分层。分层的优点在这里就不多说了，有兴趣可以去阅读相关资料。代理属于三层中的数据访问层，它会根据模型去加载和保存数据。

代理一共有 11 个类，各类的成员及其之间的关系如图 7-2 所示。从图 7-2 中可以看到，Proxy 是代理的基类，根据数据所在位置派生出了 ServerProxy 和 ClientProxy 两个类。而

ServerProxy 根据存取数据方式派生出 AjaxProxy、ScriptTagProxy（JsonP）和 DirectProxy 三个类。ClientProxy 则根据数据存储类型又派生出 MemoryProxy 和 WebStorageProxy 两个类。为了实现 RESTfu 操作，又从 AjaxProxy 派生出了 RestProxy。 为了读取 sessionStorage 或 LocalStorage 中的数据，从 WebStorageProxy 派生出了 SessionStorageProxy 和 LocalStorageProxy。

下面开始为大家介绍这些代理类。

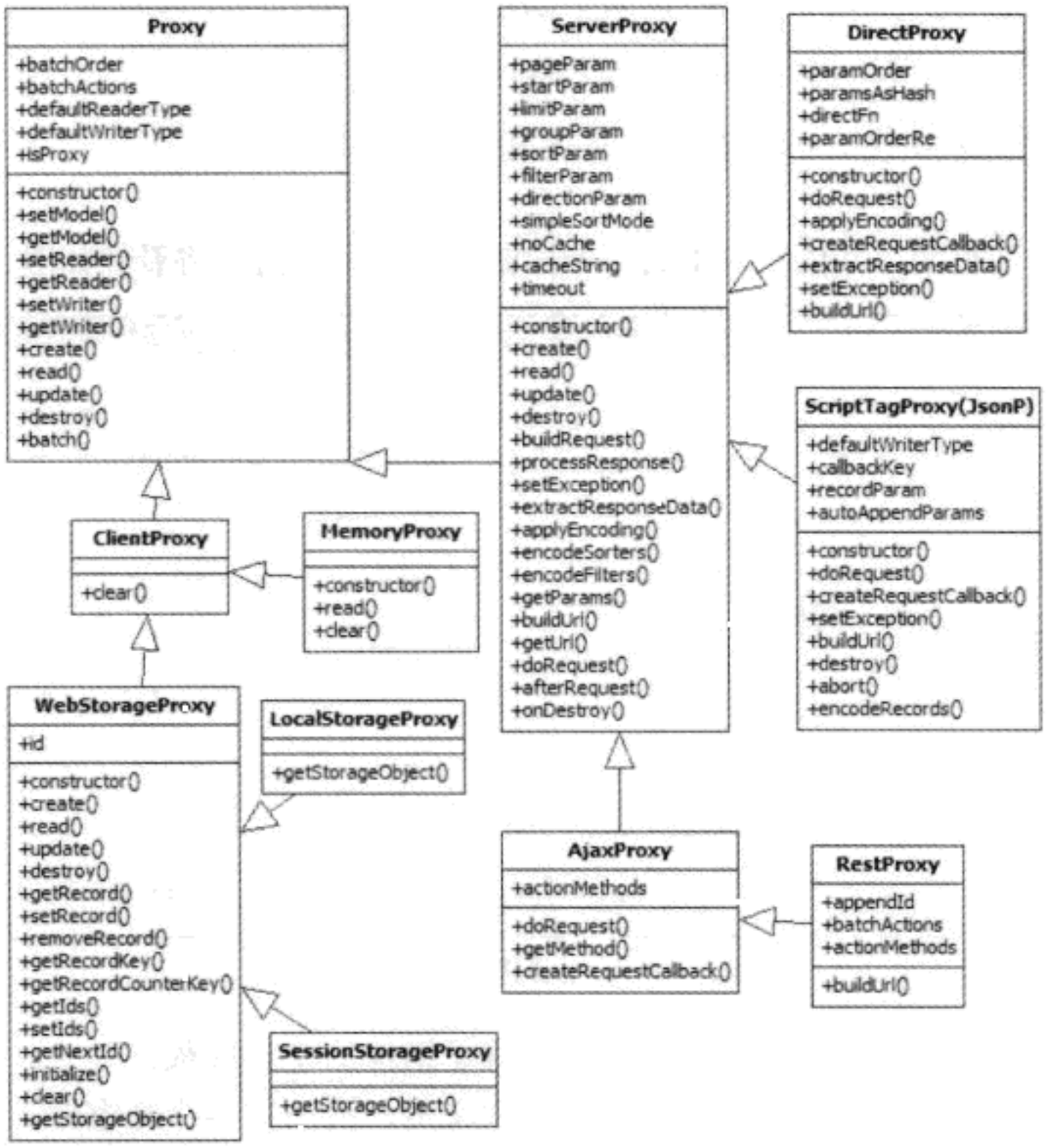


图 7-2　代理类图

7.2.2　基本的代理：Ext.data.proxy.Proxy

从图 7-2 可以看到，Proxy 类提供了 5 个属性和 11 个方法（不包含构造函数，下同），因混合了 Observable 对象，API 文档中很多方法是 Observable 对象的。因采用了不同的存储方式，其 create、read、update 和 destroy 方法都不同，所以在 Proxy 中，这 4 个方法都被定义为空函数（Ext.emptyFn），让子类进行重写。因而实际可用的只有 7 个方法。

我们先了解一下 Proxy 的运作过程，从构造函数开始：

```
constructor: function(config) {
    config = config || {};
```

```
        if (config.model === undefined) {
        delete config.model;
    }
    this.mixins.observable.constructor.call(this, config);

    if (this.model !== undefined && !(this.model instanceof Ext.data.Model)) {
        this.setModel(this.model);
    }
},
```

先检查配置对象的是否存在模型的定义，如果没有，删除配置对象中的 model 属性。接着是初始化 Observable 对象；如果存在模型，使用 setModel 方法处理模型。因为代理一般情况下是伴随 Store 或模型的创建而创建的，不单独创建，所以不用担心模型问题，系统内部会处理。在这里只是了解一下其运作过程。

下面是 setModel 方法的代码：

```
setModel: function(model, setOnStore) {
    this.model = Ext.ModelManager.getModel(model);

    var reader = this.reader,
        writer = this.writer;

    this.setReader(reader);
    this.setWriter(writer);

    if (setOnStore && this.store) {
        this.store.setModel(this.model);
    }
},
```

代码首先从 ModelManager 对象中通过 getModel 方法获取模型，接着使用 setReader 方法创建 reader[⊖]，其代码如下：

```
setReader: function(reader) {
    var me = this;

    if (reader === undefined || typeof reader == 'string') {
        reader = {
            type: reader
        };
    }

    if (reader.isReader) {
        reader.setModel(me.model);
    } else {
        Ext.applyIf(reader, {
            proxy: me,
            model: me.model,
            type : me.defaultReaderType
        });
```

⊖ 相关信息请阅读 7.3 节内容。

```
            reader = Ext.createByAlias('reader.' + reader.type, reader);
        }

        me.reader = reader;
        return me.reader;
    },
```

如果 reader 存在，使用其 setModel 方法设置模型；否则，创建一个新的 reader。其操作基本上就是让 reader 知道如何将读取的数据写到模型中。同样，setWriter 的作用是让 writer 知道如何根据模型结果去提交数据。

回到 Proxy 的 setModel 方法方法，最后判断是否存在 Store，如果是，使用其 setModel 方法设置模型。

至此，Proxy 的创建过程就结束，从过程中可以了解到，Proxy 的主要工作就是将模型和 reader 及 writer 捆绑。

在 Proxy 还提供了一个 batch 方法用来批量处理数据，其代码如下：

```
batch: function(operations, listeners) {
    var me = this,
        batch = Ext.create('Ext.data.Batch', {
            proxy: me,
            listeners: listeners || {}
        }),
        useBatch = me.batchActions,
        records;

    Ext.each(me.batchOrder.split(','), function(action) {
        records = operations[action];
        if (records) {
            if (useBatch) {
                batch.add(Ext.create('Ext.data.Operation', {
                    action: action,
                    records: records
                }));
            } else {
                Ext.each(records, function(record){
                    batch.add(Ext.create('Ext.data.Operation', {
                        action : action,
                        records: [record]
                    }));
                });
            }
        }
    }, me);

    batch.start();
    return batch;
}
```

方法会接收两个参数，第一个参数是要进行批处理的记录，第二个参数是要监听的 batch 的事件。第一个参数写法如下：

```
{
    create:[rec1,rec2],
    update:[rec3],
    destroy:[rec4]
}
```

成员 create 表示新增的记录，update 表示要更新的记录，destroy 表示要删除的记录。无论是要处理一个记录还是多个记录，都要使用数组形式定义记录集。

Batch 对象有 complete（全部操作完成）、exception（某个操作发生错误时触发）和 operationcomplete（一个操作完成时触发）三个事件。

代码首先会创建一个 Batch 对象，这个会在下一节讲述。

在枚举操作里使用到两个预设的属性 batchOrder 和 batchActions。属性 batchOrder 的作用根据操作名称去获取操作的记录并使用对应的操作处理记录。属性 batchActions 的作用是将相同操作记录一次性提交还是逐个提交。默认值为 true，会将记录一次性提交。从 batch 的代码也可以看到，当其为 true 时，会将所有记录作为一个操作。而为 false 时，则枚举每一个记录生成一个操作。

最后 Batch 对象的 start 方法开始执行批量操作。

批量操作适合那些需要让用户确认保存时才做更新的操作。

7.2.3 进行批量操作：Ext.data.Batch 与 Ext.data.Operation

在上一节的 batch 方法中会创建一个 Batch 对象的实例，其作用就是控制批量操作的进行。我们先看看 Batch 对象是怎么运作的，其构造函数如下：

```
constructor: function(config) {
    var me = this;
    me.addEvents(
      'complete',
      'exception',
      'operationcomplete'
    );
    me.mixins.observable.constructor.call(me, config);
    me.operations = [];
},
```

在这里定义了 3 个事件，从第 4 章的介绍可以知道，在执行 Observable 对象的构造函数时会绑定 listeners 定义的事件。最后是定义了一个操作数组，用来存放批量处理的操作。

再看看 Batch 对象的 add 方法：

```
add: function(operation) {
    this.total++;
    operation.setBatch(this);
    this.operations.push(operation);
},
```

每在 operations 数组添加一个操作，总数（total）就会加 1。Operation 对象 setBatch 的作用是将其 batch 属性指向当前 Batch 对象，以便访问。

开始批量操作是从 start 方法开始，其代码如下：

```
start: function() {
    this.hasException = false;
    this.isRunning = true;
    this.runNextOperation();
},
```

首先设置了一些状态，hasException 表示当前没有错误，isRunning 表示批量操作正在运行。最后调用了 runNextOperation 操作，其代码如下：

```
runNextOperation: function() {
    this.runOperation(this.current + 1);
},
```

这里不使用循环的目的是为了等一个操作完成后再执行另外一个操作，直到全部操作都完成，不然，一次并发操作太多，服务器负荷过重，后果难以预料。方法 runOperation 的代码如下：

```
runOperation: function(index) {
    var me = this,
        operations = me.operations,
        operation  = operations[index],
        onProxyReturn;

    if (operation === undefined) {
        me.isRunning  = false;
        me.isComplete = true;
        me.fireEvent('complete', me, operations[operations.length - 1]);
    } else {
        me.current = index;

        onProxyReturn = function(operation) {
            var hasException = operation.hasException();

            if (hasException) {
                me.hasException = true;
                me.fireEvent('exception', me, operation);
            } else {
                me.fireEvent('operationcomplete', me, operation);
            }

            if (hasException && me.pauseOnException) {
                me.pause();
            } else {
                operation.setCompleted();
                me.runNextOperation();
            }
        };
        operation.setStarted();
        me.proxy[operation.action](operation, onProxyReturn, me);
    }
}
```

代码先从 operations 数组中取出操作，然后判断操作是否存在，如果不存在，说明操作已经全部完成，设置批量操作运行状态为 false，完成状态为 true，并触发 complete 事件。如果存在，修改当前索引（current），然后使用闭包方式定义一个回调函数 onProxyReturn。

在函数 onProxyReturn 内，会根据操作对象（operation）的 hasException 属性检查操作执行是否存在错误，如果存在，修改代理对象的 hasException 属性为 true，并触发 exception 事件。否则，触发 operationcomplete 事件。如果存在错误，且属性 operationcomplete（错误时停止）为 true，则会调用 pause 方法设置状态为停止状态，停止批量操作。否则，调用操作对象的 setCompleted 方法设置操作状态为完成状态，运行状态为停止状态。

接着使用操作对象的 setStarted 方法设置操作的开始状态为 true，运行状态为 true。

最后一步就是根据操作的类型调用 Proxy 对象内对应的函数执行操作，例如当前操作类型为 create，代理的类型为 AjaxProxy，则会执行 ServerProxy 对象中的 create 方法。

简单地说，Batch 对象就是一个事务队列，会自动执行队列中的事务，直到没有事务为止。

在 runOperation 方法中，Operation 对象的操作基本都是一些状态设置的操作，因而可以说，Operation 对象就是一个状态对象，用来记录操作的状态及数据。这样的好处是可以把 Operation 对象当做一个配置对象，作为其他方法的参数，例如 AjaxProxy 的 doRequest 方法，就可以从操作对象中取得要提交的记录。

Operation 对象的定义相当简单，基本上是一些操作状态属性和设置状态属性的方法，在此就不再讲述了，有兴趣可以自己了解一下。

7.2.4 服务器端代理：Ext.data.proxy.Server

虽然 Ajax 与 JsonP 提交数据的方式不同，但是提交前的数据处理还是一样的，因而有了 ServerProxy 对象。

ServerProxy 对象定义了 7 个很重要的提交参数，非常重要，要谨记，表 7-1 列出了这些提交参数及其说明。

表 7-1 ServerProxy 对象提交参数及其说明

属性	值	说明
pageParam	page	如果数据是分页的，该参数指定取第几页的数据
startParam	start	指定提取数据的开始位置
limitParam	limit	每页的数据总数

groupParam	group	由进行分组的列名和排序方式构成的分组信息，其基本格式为：
		`[{"property":"col1","direction":"dir1"},…{"property":"coln","direction":"dirn"}]`
		其中 coln（n ⩾ 1）为要分组的列名。dirn（n ⩾ 1）为排序方式，为“ASC”或“DESC”。
		例如要对列 province 进行分组，且升序排列，则提交的 group 的值如下：
		`[{"property":"province","direction":"ASC"}]`
		在服务器端，最简单的方法是将值转换为 JSON 对象再处理，当然，也可以使用字符串拆分的方式获取值

（续）

属性	值	说明
sortParam	sort	如果只有一个排序的列，该值会是排序的列名。如果有多个排序列，则提交的格式与 group 的格式一样
filterParam	filter	由进行过滤的列名及过滤值做成的过滤信息，其基本格式与 group 一样，只是 direction 替换为 value，如： `[{"property":"province","value":"广东"}]` 以上代码表示对 province 进行过滤，过滤值是“广东”
directionParam	dir	如果只有一个排序的列，该值为该列的排序方式，为“ASC”或“DESC”

表 7-1 中的值是提交时在服务器端提取值的参数名称。

ServerProxy 对象的构造函数只是添加了一个 exception 事件及初始化了 extraParams、api 和 nocache 等属性，没特别的地方。

属性 api 的作用是定义数据的 CURD 操作地址，例如：

```
api: {
    read:"/user/",
    create:"/user/add",
    update:"/user/edit",
    destroy:"/user/del"
}
```

以上定义表示读取数据时，使用地址“/user/”；创建新记录时，使用地址“/user/add”；更新记录时使用地址“/user/edit”；删除数据时使用地址“/user/del”。

如果没有定义 api，则直接使用 url 定义的地址。这些定义要根据项目使用何种技术灵活配置，千万不要一刀切，不然痛苦的是自己。

方法 create、read、update 和 destroy 方法都是直接执行 doRequest 方法开始发送请求，在操作对象中会有请求类型，所以使用统一的 doRequest 方法就行了。因发送请求地方不同，实际的 doRequest 方法会在各子类中定义。

当请求完成后，会执行 processResponse 方法，其代码如下：

```
processResponse: function(success, operation, request, response, callback, scope){
    var me = this,
        reader,
        result,
        records,
        length,
        mc,
        record,
        i;

    if (success === true) {
        reader = me.getReader();
        result = reader.read(me.extractResponseData(response));
        records = result.records;
        length = records.length;
```

```
if (result.success !== false) {
    mc = Ext.create('Ext.util.MixedCollection', true, function(r) {return
          r.getId();});
    mc.addAll(operation.records);
    for (i = 0; i < length; i++) {
        record = mc.get(records[i].getId());

        if (record) {
            record.beginEdit();
            record.set(record.data);
            record.endEdit(true);
        }
    }

    Ext.apply(operation, {
        response: response,
        resultSet: result
    });

    operation.setCompleted();
    operation.setSuccessful();
} else {
    operation.setException(result.message);
    me.fireEvent('exception', this, response, operation);
}
} else {
    me.setException(operation, response);
    me.fireEvent('exception', this, response, operation);
}

if (typeof callback == 'function') {
    callback.call(scope || me, operation);
}

me.afterRequest(request, success);
},
```

如果请求成功，会使用 Reader 对象 read 方法将数据从返回的数据格式化为标准的模型数据。因为 Store 最后的数据格式是 MixedCollection[⊖]对象，因而需要将数据转换为 MixedCollection 对象数据格式，还需要使用模型的 set 方法设置数据初始状态，例如 dirty 属性。

接着会将响应对象和返回的记录集复制到操作对象内，并设置操作对象的完成状态为 true，操作状态为成功。

如果请求不成功，则设置操作对象的意外状态为 true，并触发 exception 事件。

如果存在回调函数，执行回调函数。

最后执行 afterRequest 方法，不过目前还是一个空函数。

ServerProxy 对象余下的方法主要作用是为提交请求处理提交地址及提交参数，在此就不多说了，有兴趣可以自己研究。

⊖ 相关信息请阅读 7.4.4 节。

7.2.5 使用 Ajax 处理数据的代理：Ext.data.proxy.Ajax 与 Ext.data.proxy.Rest

AjaxProxy 对象的作用是使用 Ajax 方式存取数据。代码不多，重点是 doRequest 方法，其代码如下：

```
doRequest: function(operation, callback, scope) {
    var writer  = this.getWriter(),
        request = this.buildRequest(operation, callback, scope);

    if (operation.allowWrite()) {
        request = writer.write(request);
    }

    Ext.apply(request, {
        headers        : this.headers,
        timeout        : this.timeout,
        scope          : this,
        callback       : this.createRequestCallback(request, operation, callback, scope),
        method         : this.getMethod(request),
        disableCaching: false
    });

    Ext.Ajax.request(request);

    return request;
},
```

使用 buildRequest 方法处理完提交地址和提交参数后，使用 Writer[⊖]的 write 方法将提交的记录处理成合适的提交方式后，再将必要的数据复制到请求对象，就可以调用 Ajax 对象的 request 方法提交请求了。要注意的是，回调函数是使用 createRequestCallback 方法创建的闭包函数，实际上还是调用 processResponse 方法。

REST 是 Representational State Transfer 的简写，大概意思就是表征状态转换，是 Roy Fielding 博士在 2000 年他的博士论文中提出来的一种软件架构风格[⊜]。说得有点玄乎了，简单点来说，就是根据请求的方式来决定进行什么处理，一般情况下，有 POST、GET、PUT 和 DELETE 这 4 种请求方式，分别对应的 Web 服务是新建、获取、更新和删除 4 种操作。还不明白就看下面的示例。

（1）功能描述

本示例主要验证 RestProxy 的功能。

（2）实现代码

首先使用模版页创建一个名称为 7-3.html 的页面文件。

要使用 RestProxy，需要先定义一个模型：

```
Ext.define('Test', {
    extend: 'Ext.data.Model',
    fields: ['id', 'TestName'],
```

⊖ 相关信息请阅读 7.3.4 节。

⊜ 详细信息请参阅 http://zh.wikipedia.org/wiki/REST

```
    proxy: {
        type: 'rest',
        format:"aspx", //java 程序请去掉
        url : 'test'
    }
});
```

模型很简单，只有两个字段，这个不重要。关键是 proxy 的定义，type 属性为 rest 则表示使用 RestProxy,。

下面定义代码：

```
var test="";

var panel=Ext.create("Ext.Panel",{
   renderTo:Ext.getBody(),
   height:500,
   width:400,
   tbar:[
          {text:"读取",scope:panel,handler:function(){
          Test.load(1,{
                 success: function(d,o) {
                 test=d;
                 panel.update(o.response.responseText);
                 }
          });
          }},
          {text:"新增",scope:panel,handler:function(){
          var t=Ext.ModelManager.create({id:"",TestName:"新增"},"Test");
          t.save({
          success: function(d,o) {
                 panel.update(o.response.responseText);
                 }
          });
          }},
          {text:"更新",scope:panel,handler:function(){
          test.save({
                 success: function(d,o) {
                 panel.update(o.response.responseText);
                 }
          })
          }},
          {text:"删除",scope:panel,handler:function(){
          test.destroy({
          success: function(d,o) {
                 panel.update(o.response.responseText);
                 }
                 })
                 }}
    ]
});
```

变量 test 的作用是保存单击读取按钮后返回的 Test 对象示例，这样才能执行后续的更新和删除操作。页面中会显示一个面板，面板上有读取、新增、更新和删除按钮分别用来验证

REST 的 4 个的提交，提交成功后，将返回的文本显示在面板中。

接着是完成服务器端的代码 .

```
C#:
public void ProcessRequest (HttpContext context) {
    context.Response.ContentType = "text/javascript";
    string method = context.Request.HttpMethod;
    switch (method)
    {
        case "POST":
            context.Response.Write("{id:2,TestName:' 新增 '}");
            break;
        case "PUT":
            context.Response.Write("{id:2,TestName:' 更新 '}");
            break;
        case "DELETE":
            context.Response.Write("{id:2,TestName:' 删除 '}");
            break;
        default: //GET
            context.Response.Write("{id:1,TestName:' 获取 '}");
            break;

    }
}
```

Java:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
   response.setContentType("text/html; charset=utf-8");
   response.getWriter().write("{id:2,TestName:' 获取 '}");
}

protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
   response.setContentType("text/html; charset=utf-8");
   response.getWriter().write("{id:2,TestName:' 新增 '}");
}

protected void doPut(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
   response.setContentType("text/html; charset=utf-8");
   response.getWriter().write("{id:2,TestName:' 更新 '}");
}

protected void doDelete(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
   response.setContentType("text/html; charset=utf-8");
   response.getWriter().write("{id:2,TestName:' 删除 '}");
}
```

如果客户端提取请求的方式是 GET，返回的 TestName 值是“读取”；如果是 POST，返回的 TestName 值得是“新增”；如果是 PUT，返回的 TestName 值是“更新”；如果是 Delete，返回的 TestName 值是“删除”。

（3）页面效果

在浏览器中打开页面将看到如图 7-3 所示的页面。

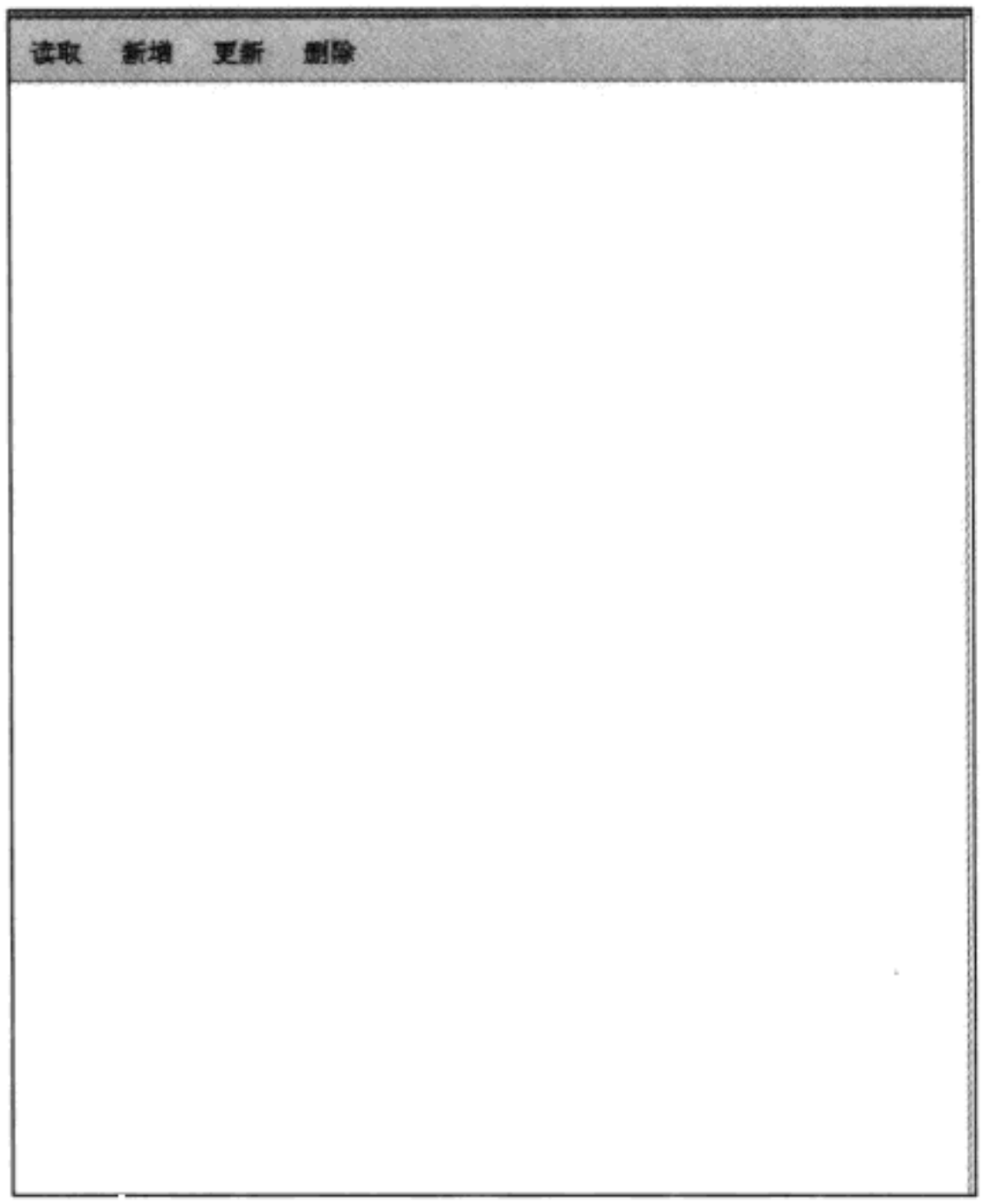


图 7-3 示例 RESTful 的页面效果

要打开 Firebug 观察提交请求。

首先单击“读取”按钮，会在控制台中看到以下请求：

```
C#:
GET http://localhost/extjs4/test/1.aspx?_dc=1306345615527&id=1

Java:
GET http://localhost:8080/Restful/test/1?_dc=1306345105207&id=1
```

在面板中可看到“{id:1,TestName:'获取'}”。说明服务器程序已争取处理了 GET 方式的请求。

接着单击“新增”按钮，会在控制台中看到以下请求：

```
C#:
POST http://localhost/extjs4/test.aspx?_dc=1306345651406

Java:
POST http://localhost:8080/Restful/test?_dc=1306345361753
```

在面板中可看到“{id:1,TestName:'新增'}”。说明服务器程序已争取处理了 POST 方式的请求。

接着单击“更新”按钮，会在控制台中看到以下请求：

```
C#:
PUT http://localhost/extjs4/test/1.aspx?_dc=1306345689014

Java:
PUT http://localhost:8080/Restful/test/2?_dc=1306345742337
```

在面板中可看到“{id:1,TestName:'更新'}”。说明服务器程序已争取处理了 PUT 方式的请求。

最后单击“删除”按钮，会在控制台中看到以下请求：

```
C#:
DELETE http://localhost/extjs4/test/2.aspx?_dc=1306345859843

Java:
DELETE http://localhost:8080/Restful/test/2?_dc=1306345903297
```

在面板中可看到“{id:1,TestName:'删除'}”。说明服务器程序已争取处理了 DELETE 方式的请求。

从示例可以看到，服务器根据提交方式就可正确处理数据的读取、新增、更新和操作，相当的方便。使用 Java 会更方便，一种方式就是一个方法，而且提交路径设置也方便。使用 C# 首先要设置 URL 重写，还要修改程序映射，为 aspx 和 ashx 文件添加 PUT 和 DELETE 谓词。

7.2.6 跨域处理数据的代理：Ext.data.proxy.JsonP

如果要跨域处理数据，就需要使用 ScriptTagProxy，它会使用 JsonP 对象进行数据处理，也就是说，你的服务器端代码必须直接执行回调函数。这可能是相当恼人的事情，但要做跨域，一定要考虑清楚这个问题。

ScriptTagProxy 与 AjaxProxy 一样，需要重写 doRequest 方法，因提交参数的差异，还要重写 buildUrl 方法。代码雷同，在此就不多说了，有兴趣可以自己研究一下。

7.2.7 为 Ext.Direct[⊖]服务的代理：Ext.data.proxy.Direct

DirectProxy 是为 Ext.Direct 服务的代理，因而有其特殊性，注意特殊的地方在提交参数必须是有次序的，因为服务器端的方法接收参数是依次序接收的。设置变量次序的属性是 paramOrder，其定义格式如下：

```
paramOrder:['param1','param2',…,'paramn'];
paramOrder:['param1,param2,…,paramn'];
paramOrder:['param1 param2 … paramn'];
paramOrder:['param1|param2|…|paramn'];
```

以上数组、使用空格、逗号或“|”分隔开的字符串等 4 种方式都是允许的格式。

加载数据时的排序信息、过滤信息、分页的信息等是一定要定义好次序，不然服务器端

⊖ 相关信息请阅读第 17 章。

不知道如何获取这些参数，例如：

```
paramOrder:['page,start,limit,sort,dir,filter'];
```

还有一个必须定义的属性是 directFn，该属性的作用是用来指定请求对象，也就是要执行服务器端的哪个方法，如果没有指定，那就不知道将数据提交到哪里了。

因为 DirectProxy 是直接调用服务器端方法，所以不需要处理 URL，参数则要严格依据顺序处理，因而其 doRequest 方法与 AjaxProxy 和 ScriptTagProxy 有所不同：

```
doRequest: function(operation, callback, scope) {
var me = this,
    writer = me.getWriter(),
    request = me.buildRequest(operation, callback, scope),
    fn = me.api[request.action]  || me.directFn,
    args = [],
    params = request.params,
    paramOrder = me.paramOrder,
    method,
    i = 0,
    len;

// 省略 debug 代码
if (operation.allowWrite()) {
    request = writer.write(request);
}

if (operation.action == 'read') {
    method = fn.directCfg.method;

    if (method.ordered) {
        if (method.len > 0) {
            if (paramOrder) {
                for (len = paramOrder.length; i < len; ++i) {
                    args.push(params[paramOrder[i]]);
                }
            } else if (me.paramsAsHash) {
                args.push(params);
            }
        }
    } else {
        args.push(params);
    }
} else {
    args.push(request.jsonData);
}

    Ext.apply(request, {
        args: args,
        directFn: fn
    });
    args.push(me.createRequestCallback(request, operation, callback, scope), me);
    fn.apply(window, args);
},
```

代码依然会使用 buildRequest 方法建立请求对象。然后根据请求的操作类型从 API 中获取操作方法，如果不存在，则直接使用 directFn 定义的方法。

如果操作是读取数据的话，则检查服务器端方法的参数是否需要排序，例如获取产品的详细信息，只接收产品编号这唯一参数，则不需要排序。如果要排序，则根据 paramOrder 定义的顺序将参数加入数组；如果不需要，直接将提交参数加入数组就行了。

如果是其他操作，则将请求对象中的数据加入数组，因为其他方式一般是以表单形式获取数据的，所以不需要排序，但需要数据。

接着就是将 args 和 fn 加入请求对象，将回调函数加入参数，然后执行服务器端映射回来的方法，提交数据。

7.2.8 客户端代理：Ext.data.proxy.Client

HTML 5 新增了 Web 存储用来存储本地数据，因而在 Ext JS 4 中增加了 ClientProxy 用来存储本地不同的数据。

在 ClientProxy 中只简单的定义了一个需要重写的 clear 方法。其作用是作为所有本地存储的基类。

7.2.9 从变量中提取数据的代理：Ext.data.proxy.Memory

MemoryProxy 的作用就是将变量指向的数据读到模型中，因而只有简单的 read 方法，代码比较简单，在此就不多说了，有兴趣可以自己研究一下。

7.2.10 使用浏览器存储的代理：Ext.data.WebStorageProxy、Ext.data.SessionStorageProxy 和 Ext.data.proxy.LocalStorage

Web 存储提供了 localStorage 和 sessionStorage 两种存储，在 localStorage 存储中的数据是长久有效的，类似时永不过期的 Cookies、sessionStorage 则会在浏览器关闭的时候被清除。LocalStorageProxy 和 SessionStorageProxy 主要区别是取数据的地方不同，LocalStorageProxy 是从 localStorage 读取数据，而 SessionStorageProxy 是从 sessionStorage 读取数据，它们的操作都是一样的，因而 LocalStorageProxy 和 SessionStorageProxy 只定义了 getStorageObject 方法，用来获取数据，它们的操作都是从其父类 WebStorageProxy 继承的。

与 ServerProxy 一样，WebStorageProxy 也提供了数据的读取（read）、创建（create）、更新（update）和删除（destroy）等 4 个方法用来操作数据，不过因为都是本地数据，不需要提交，因而这些操作都无需使用 Reader 对象和 Writer 对象，直接操作记录集就行了。

Web 存储是不错的东西，使用 sessionStorage 可以用来保存用户的权限信息，用 localStorage 则可以保存用户的主题等信息，且其操作比 Cookies 的简单，如果不考虑浏览器问题，值得用来代替 Cookies。

7.3 读取和格式化数据

7.3.1 概述

从服务器端取回来的数据可能是数组、JSON 格式的数据或 XML 数据，让用户自己去处理这些数据虽然可行。但是，UI 组件存在一个问题，它们必须在内部内置数据处理模块，来处理这些不同格式的数据，这对 UI 的开发明显是不利的，试想要改一下数据处理方式，就得修改所有 UI 组件，这是相当吃力不讨好的事情。所以，需要将数据封装成标准的数据格式，这就是数据模型，而不同格式的数据就需要一个工具将其转换为标准的数据格式，这就是 Reader。

7.3.2 数据的转换过程：Ext.data.reader.Xml、Ext.data.reader.Json 和 Ext.data.reader.Array

Ext JS 格式化后的数据格式就是模型，因而数据转换的关键就是如何将不同格式数据的字段对应到模型的字段上。数组只有索引，因而只能根据索引对应字段，而 JSON 数据和 XML 都可根据映射名称或字段名称进行对应。

在 Reader 创建的时候，会依据每个字段的次序创建一个数据提取器的函数并保存在 extractorFunctions 指向的数组中，提取器的作用就是根据字段从数据中提取字段值。创建提取器的方法是 buildExtractors，不同的数据格式生成的提取器也是不同的。

ArrayReader 的 buildExtractors 方法如下：

```
buildExtractors: function() {
    this.callParent(arguments);
    var fields = this.model.prototype.fields.items,
        length = fields.length,
        extractorFunctions = [],
        i;
    for (i = 0; i < length; i++) {
        extractorFunctions.push(function(index) {
            return function(data) {
                return data[index];
            };
        }(fields[i].mapping || i));
    }
    this.extractorFunctions = extractorFunctions;
}
```

代码从模型的原型中取到字段数组后，就逐个根据它们的索引建立提取器，这里采用了闭包的方式返回一个匿名函数，注意，如果字段定义了 mapping 属性，会根据 mapping 指定的索引去取值。

JsonReader 的 buildExtractors 方法代码如下：

```
buildExtractors : function() {
    var me = this;
```

```
        me.callParent(arguments);

        if (me.root) {
            me.getRoot = me.createAccessor(me.root);
        } else {
            me.getRoot = function(root) {
                return root;
            };
        }
    },
```

它会首先调用父类的 buildExtractors 方法，其代码如下：

```
buildExtractors: function(force) {
    var me          = this,
        idProp      = me.getIdProperty(),
        totalProp   = me.totalProperty,
        successProp = me.successProperty,
        messageProp = me.messageProperty,
        accessor;
    if (force === true) {
        delete me.extractorFunctions;
    }
    if (me.extractorFunctions) {
        return;
    }
    if (totalProp) {
        me.getTotal = me.createAccessor(totalProp);
    }
    if (successProp) {
        me.getSuccess = me.createAccessor(successProp);
    }
    if (messageProp) {
        me.getMessage = me.createAccessor(messageProp);
    }
    if (idProp) {
        accessor = me.createAccessor(idProp);
        me.getId = function(record) {
            var id = accessor.call(me, record);
            return (id === undefined || id === '') ? null : id;
        };
    } else {
        me.getId = function() {
            return null;
        };
        me.buildFieldExtractors();
    },
```

因为 JSON 格式的数据可能还包含总记录数（totalProperty）、返回记录的状态（success-Property）、提示信息（messageProperty）等信息，因而也要为它们提供提取器，如果定义了 id 属性（idProperty），那么也要构建 id 的提取器，否则返回 null。如果是字段，执行 buildFieldExtractors 方法构建字段的提取器。构建这些提取器都需要使用 createAccessor 方

法，其代码如下：

```
createAccessor: function() {
    var re = /[\[\.]/;

    return function(expr) {
        if (Ext.isEmpty(expr)) {
            return Ext.emptyFn;
        }
        if (Ext.isFunction(expr)) {
            return expr;
        }
        if (this.useSimpleAccessors !== true) {
            var i = String(expr).search(re);
            if (i >= 0) {
                return Ext.functionFactory('obj', 'return obj' + (i > 0 ? '.' : '')
                    + expr);
            }
        }
        return function(obj) {
            return obj[expr];
        };
    };
}()
```

方法 createAccessor 是一个闭包函数，默认情况下，生成的提取器会直接使用粗体代码返回字段值。如果提取器的属性形式为“f1.f2”这样的形式，那么返回形式就会变成“obj.f1.f2”。

XmlReader 直接使用 Reader 对象的 buildExtractors 方法，因而最终也是使用其内部的 createAccessor 方法创建提取器，其代码如下：

```
createAccessor: function(expr) {
    var me = this;

    if (Ext.isEmpty(expr)) {
        return Ext.emptyFn;
    }

    if (Ext.isFunction(expr)) {
        return expr;
    }

    return function(root) {
        var node = Ext.DomQuery.selectNode(expr, root),
            val = me.getNodeValue(node);

        return Ext.isEmpty(val) ? null : val;
    };
},
```

从代码可以看到，提取器函数会使用 DomQuery 对象找到节点，然后使用 getNodeValue 方法返回值，其代码如下：

```
getNodeValue: function(node) {
    var val;
    if (node && node.firstChild) {
        val = node.firstChild.nodeValue;
    }
    return val || null;
},
```

在 XML 中，文本是作为一个文本节点存在的，因而使用 firstChild 取得文本节点，返回其节点值就是需要的值了。

提取器构建好了，就可以处理数据了。无论是从内存中获取的数据，还是从服务器获取的数据，都会调用 Reader 对象的 read 方法处理返回的数据，其代码如下：

```
read: function(response) {
    var data = response;

    if (response && response.responseText) {
        data = this.getResponseData(response);
    }

    if (data) {
        return this.readRecords(data);
    } else {
        return this.nullResultSet;
    }
},
```

从代码可以很清楚看到，如果是从服务器返回的数据，都要使用 getResponseData 处理返回的数据。如果服务器返回的是 JSON 数据，responseText 是文本格式，所以需要使用 Ext.JSON 对象的 decode 将其转换为对象。而 XML 数据，则直接使用 responseXML 属性返回 XML 文档对象即可。

如果数据存在，则使用 readRecords 方法读取数据。JsonReader 和 XmlReader 都重写了 Reader 对象的 readRecords 方法，但最终是执行 Reader 对象的 readRecords 方法处理数据的，其代码如下：

```
readRecords: function(data) {
    var me  = this;

    if (me.fieldCount !== me.getFields().length) {
        me.buildExtractors(true);
    }
    me.rawData = data;
    data = me.getData(data);
    var root    = Ext.isArray(data) ? data : me.getRoot(data),
        success = true,
        recordCount = 0,
        total, value, records, message;

    if (root) {
        total = root.length;
    }
```

```
if (me.totalProperty) {
    value = parseInt(me.getTotal(data), 10);
    if (!isNaN(value)) {
        total = value;
    }
}
if (me.successProperty) {
    value = me.getSuccess(data);
    if (value === false || value === 'false') {
        success = false;
    }
}
if (me.messageProperty) {
    message = me.getMessage(data);
}
if (root) {
    records = me.extractData(root);
    recordCount = records.length;
} else {
    recordCount = 0;
    records = [];
}
return Ext.create('Ext.data.ResultSet', {
    total   : total || recordCount,
    count   : recordCount,
    records: records,
    success: success,
    message: message
});
},
```

如果字段发生改变，则需要重新构建提取器。

接着将 rawData 属性指向原始数据。

如果 data 是数组，它的根节点就是数据本身；如果是 JSON 数据，则会使用 getRoot 方法（由 createAccessor 方法生成）提取数据；如果是 XML 数据，则在 XML 文档中寻找到数据的根节点。

接着是处理记录总数、返回记录的状态、提示信息等数据。

接着是使用 extractData 处理数据，其代码如下：

```
extractData : function(root) {
    var me = this,
        values  = [],
        records = [],
        Model   = me.model,
        i       = 0,
        length  = root.length,
        idProp  = me.getIdProperty(),
        node, id, record;

if (!root.length && Ext.isObject(root)) {
    root = [root];
    length = 1;
```

```
        }

        for (; i < length; i++) {
            node   = root[i];
            values = me.extractValues(node);
            id     = me.getId(node);


            record = new Model(values, id, node);
            records.push(record);

            if (me.implicitIncludes) {
                me.readAssociated(record, node);
            }
        }

        return records;
    },
```

取数据的过程就是遍历每行数据，使用 extractValues 方法将数据按字段名称重新组织成一个对象，其代码如下：

```
    extractValues: function(data) {
        var fields = this.getFields(),
            i      = 0,
            length = fields.length,
            output = {},
            field, value;

        for (; i < length; i++) {
            field = fields[i];
            value = this.extractorFunctions[i](data);

            output[field.name] = value;
        }

        return output;
    },
```

从代码可以很清楚地看到，根据字段的提取器提取值，然后将以字段名称为属性、提取值为值添加到对象 output 中。

将对象 output 返回后，会使用该对象创建一个新模型，模型将在 7.4 节进行讲解，这里先跳过。

接着将模型对象放到 records 数组中。

如果模型存在关联，则使用 readAssociated 方法读取关联数据。如果关联数据已存在返回的数据中，则从返回的数据中取；如果不存在，则使用关联的数据模型的 Reader 对象读取。

最后将 records 数组返回到 readRecords 方法。在 readRecords 方法，又用 ResultSet 对象封装一次数据后返回。ResultSet 对象包含了数据的记录总数、当前记录数、记录、success 标志和返回结果等内容。

至此，数据就转换完成了。

7.3.3 Reader 对象的配置项

Reader 对象的配置项决定了如何从返回的数据中提取数据，因而要想正确的返回数据，必须清楚这些配置项。Reader 对象提供了以下 6 个配置项：

- idProperty：字符串，指定哪个字段为每个记录的唯一标识字段。如果模型中定义了唯一标识字段，则使用模型的定义，否则为 undefined。如果设置值与模型指定的唯一标识字段不同，则会使用该属性指定的字段重写模型的唯一标识字段。
- totalProperty：从返回数据获取数据库记录总数的属性名称，默认值为 total。
- successProperty：从返回数据获取 success 属性的属性名称，默认值是 success。这个很重要，如果返回数据不包含 success 属性，那么就不会触发 success 回调函数，也不会读取数据；
- root：一定要设置该配置项，其作用就是从返回数据中提取数据的属性名称。其默认值为空字符串，所以如果不设置该配置项，将获取不了数据。
- messageProperty：其作用是从返回数据获取信息的属性名称，例如，服务器端处理时发现提交的参数不符合要求，就可通过该配置指定的属性名称返回提示信息。
- implicitIncludes：布尔值，默认值为 true，意味着会自动处理数据中的关系模型数据。

在 JsonReader、XmlReader 和 ArrayReader 中还有一个 record 属性用来读取格式比较特殊的数据。

根据以上属性描述，可以知道与返回数据相关的属性主要有 totalProperty、successProperty、messageProperty 和 root 这 4 个，其中，root 是必须有的，因而可根据这 4 个属性确定如何组织返回数据，例如定义了以下属性：

```
{
    root:'data',
    messageProperty:'msg'
}
```

则 JSON 格式的返回数据标准格式是：

```
{
    total:100,
    success:true,
    msg:'成功',
    data:[
            {// 数据 1}
            {// 数据 2}
            ...
    ]
}
```

因为没有定义 totalProperty 和 successProperty 属性，所以使用默认的属性名称。相应的 XML 数据格式为：

```
<?xml version="1.0" encoding="UTF-8"?>
<total>100</total>
<success>true</success>
```

```
<msg>成功</msg>
<data>
    <data1></data1>
    <data2></data2>
    ...
</data>
```

配置项 record 的作用，看一下源代码会比较清楚，以下是 JsonReader 提取数据的方法。extractData 的代码：

```
extractData: function(root) {
    var recordName = this.record,
        data = [],
        length, i;

    if (recordName) {
        length = root.length;

        for (i = 0; i < length; i++) {
            data[i] = root[i][recordName];
        }
    } else {
        data = root;
    }
    return this.callParent([data]);
},
```

代码中 recordName 获取 record 配置项的值。如果 recordName 存在，则粗体代码会将数据从 root 中取出存放到 data 数组中，而取出数据的关键字就是 record 配置项指定的值。

再看看 XmlReader 的 extractData 方法：

```
extractData: function(root) {
    var recordName = this.record;

    //省略调试代码
    if (recordName != root.nodeName) {
        root = Ext.DomQuery.select(recordName, root);
    } else {
        root = [root];
    }
    return this.callParent([root]);
},
```

注意粗体代码，数据变成了从 record 配置项指定的节点上的数据了。

也就是说，只要指定了 record 就不是从 root 指定的位置取数据，而是由 record 指定的位置取数据。

7.3.4 格式化提交数据：Ext.data.writer.Writer、Ext.data.writer.JSON 和 Ext.data.writer.Xml

因为 UI 组件中使用的是格式化后的数据，因而要将数据提交给服务器，需要一个逆过程

将数据转换为提交格式。数据的提交格式只有 JSON 和 XML 两种。

要提交数据，首先要做的就是通过 getRecordData 方法获取改变过的数据，其代码如下：

```
getRecordData: function(record) {
    var isPhantom = record.phantom === true,
        writeAll = this.writeAllFields || isPhantom,
        nameProperty = this.nameProperty,
        fields = record.fields,
        data = {},
        changes,
        name,
        field,
        key;

    if (writeAll) {
        fields.each(function(field){
            if (field.persist) {
                name = field[nameProperty] || field.name;
                data[name] = record.get(field.name);
            }
        });
    } else {
        changes = record.getChanges();
        for (key in changes) {
            if (changes.hasOwnProperty(key)) {
                field = fields.get(key);
                name = field[nameProperty] || field.name;
                data[name] = changes[key];
            }
        }
        if (!isPhantom) {
            data[record.idProperty] = record.getId();
        }
    }
    return data;
}
```

如果是将全部字段值都提交的话（writeAllFields 属性为 true），处理很简单，直接从使用模型的 get 方法获取值，然后以字段名为关键字、其值为值的成员添加到 data 对象就行了。

如果只提交修改值，则首先通过模型的 getChanges 方法返回哪些字段被修改过，然后将修改过的值取出来加入 data 对象。很重要的是要在记录中加入 id 值，不然服务器不知道修改哪条记录。

数据获取后，就可以使用 writeRecords 将记录写到请求对象里。

JsonWriter 的 writeRecords 代码如下：

```
writeRecords: function(request, data) {
    var root = this.root;

    if (this.allowSingle && data.length == 1) {
        data = data[0];
    }
```

```
if (this.encode) {
    if (root) {
        request.params[root] = Ext.encode(data);
    } else {
        // 省略抛出异常代码
    }
} else {
    request.jsonData = request.jsonData || {};
    if (root) {
        request.jsonData[root] = data;
    } else {
        request.jsonData = data;
    }
}
return request;
}
```

当 allowSingle 为 ture 且提交数据的长度为 1（数组只有 1 个数据）时，需要将数据从数组中取出来，转换为非数组形式。如果 enccde 属性为 true，则将记录使用 Ext.JSON 对象的 encode 方法转换为字符串后，作为查询字符串提交。否则，将请求对象的 JsonData 属性指向数据，让提交对象自行处理后提交。

XmlWriter 对象 writeRecords 方法代码如下：

```
writeRecords: function(request, data) {
    var me = this,
        xml = [],
        i = 0,
        len = data.length,
        root = me.documentRoot,
        record = me.record,
        needsRoot = data.length !== 1,
        item,
        key;
xml.push(me.header || '');

if (!root && needsRoot) {
    root = me.defaultDocumentRoot;
}
if (root) {
    xml.push('<', root, '>');
}
for (; i < len; ++i) {
    item = data[i];
    xml.push('<', record, '>');
    for (key in item) {
        if (item.hasOwnProperty(key)) {
            xml.push('<', key, '>', item[key], '</', key, '>');
        }
    }
    xml.push('</', record, '>');
}
if (root) {
    xml.push('</', root, '>');
```

```
        }
        request.xmlData = xml.join('');
        return request;
    }
```

整个流程就是使用一个数组将数据封装成 XML 格式的字符串，然后将数组生成的字符串复制给请求对象的 xmlData 属性，让提交对象提交。

Writer 要的工作就是将提交数据组织起来。

7.3.5 Writer 对象的配置项

Writer 对象提供两个配置项：

- writeAllFields：布尔值，默认值为 true，表示会将所有字段都提交到服务器。如果设置为 false，则只提交修改过的字段。
- nameProperty：提交的数据的关键字，默认值是 name，使用数据模型中字段定义中 name 的值为关键字。也可以另外定义关键字用来提交，例如，定义了 mapping 属性，可以使用 mapping 的值作为关键字。

JsonWriter 提供了 3 个配置项：

- root：从上一节的源代码可以知道，如果 encode 为 true，则 root 的值将作为提交变量的关键字；如果 encode 为 false，则作为对象的属性名称，其值指向提交数据。而其默认值是 undefined，因而提交数据将直接提交。
- encode：布尔值，默认值为 false，不对提交数据进行编码。从上一节的源代码可以知道，只有设置了 root 值之后，才会对数据进行编码。
- allowSingle：布尔值，默认值为 true，表示数据只有 1 个时，才能不以数组形式提交数据。如果为 false，则无论提交数据是 1 个还是多个，都以数组形式提交。

XmlWriter 提供了 4 个配置项：

- documentRoot：字符串，提交的 XML 文档根元素，默认值是 xmlData。
- defaultDocumentRoot：如果 documentRoot 的值为空，则使用该值作为提交的 XML 文档的根元素。默认值是 xmlData。
- header：字符串，设置 XML 文档头部，如 encoding 或 version 等，默认值为空字符串。
- record：提交的 XML 文档中包含提交数据的节点，默认值是 record。

这里我们要清楚一点，JsonWriter 默认的提交数据的 Content-Type 是 application/JSON，因而使用习惯的查询字符串获取数据的方式是取不到数据的，如果是 C#，需要使用 InputStream 来读取，Java 则需要使用 getInputStream 读取。如果不熟悉这些方法，一定要设置 root 配置项和设置 encode 配置项为 true，这样就提交数据的 Content-Type 是 application/x-www-form-urlencoded，也就可以使用习惯的方法通过 root 设置的名称获取数据了。

XmlWriter 提交的 XML 文档，因而只能使用 InputStream 或 getInputStream 读取数据。

7.4 数据模型

7.4.1 概述

在 Ext JS 4 引入了数据模型的概念，以替代之前版本的 Record 对象，它与 Record 对象的主要区别是引入了关联、验证等新的内容。

简单来说，一个数据模型实例相当于数据库的一条记录，它由字段组成。通过数据模型，可以添加记录、删除记录、编辑记录和更新记录等。

在使用数据模型之前，必须先了解数据模型中的一些基本元素，如字段、验证、关系等。

7.4.2 数据类型及排序类型：Ext.data.Types 与 Ext.data.SortTypes

JavaScript 不是强类型的开发语言，因而对数据排序或过滤等处理，会因数据类型的不同而出现差异。例如，数字在按字符串排序与按整型来排序结果是不同。因而，为了能正确对数据进行排序或过滤等操作，必须强制其按指定的数据类型进行运作，而这就是 Types 对象和 SortTypes 对象的作用。

SortTypes 对象可指定数据按什么方式进行排序，预设了 7 种类型：

- ❑ none：根据数据的原来类型进行排序，就是返回未经处理的原数据排序。
- ❑ asText：该方式会去除文本中的 HTML 标记再进行排序。
- ❑ asUCText：该方式与 asText 作用一样，不过它不区分大小写。
- ❑ asUCString：对字符串不区分大小写进行排序。
- ❑ asDate：按日期进行排序。
- ❑ asFloat：按浮点数进行排序。
- ❑ asInt：按整型进行排序。

Types 对象提供了 6 种数据类型对象，每个对象都包含 convert（转换函数）、sortType（排序类型）和 type（类型）三个成员：

- ❑ AUTO ：如果字段定义没有标明数据类型，那么其类型就为 AUTO 类型，AUTO 类型的转换函数不做任何转换，直接返回原数据，因而其排序类型为 none。
- ❑ STRING ：字符串。如果定义字段时设置了 useNull 为 true，则转换函数在值不存在时会返回 null 值，否则返回空字符串。其类型为 string。默认排序类型为 asUCString。
- ❑ INT（INTEGER）：整型。它的转换函数很有意思，会自动替换掉货币符号“$”和百分号“%”，然后在再使用 parseInt 方法转换为整数。如果字段定义了 useNull 为 true，在原值不能转换为整数的情况下，返回 null 值，否则返回 0。其类型为 int。默认排序类型为 none。
- ❑ FLOAT（NUMBER）：浮点型。转换函数与整型类似，只是转换使用的是 parseFloat 方法。其类型为 float。默认排序类型为 none。
- ❑ BOOL（BOOLEAN）：布尔值。转换函数如果允许返回 null，则当值不存在或为空字

符串时返回 null，否则返回 true、字符串“true”或 1。其类型为 bool。默认排序类型为 none。

- DATE：日期。日期会根据字段定义的日期格式（dateFormat）将原数据转换为日期返回。其类型为“date”。默认排序类型为 asDate。

Types 对象和 SortTypes 对象都是可以扩展的，而且相当容易，下面通过一个示例来演示如何扩展 Types 对象和 SortTypes 对象。

（1）功能描述

示例主要演示如何自定义数据类型和排序类型。主要数据如下：

```
var data=[
    {name:"张三", nationality:{country:"中国",img:"cn.gif"},salary:"10000"},
    {name:"李四", nationality:{country:"中国",img:"cn.gif"},salary:"14000"},
    {name:"王五", nationality:{country:"中国",img:"cn.gif"},salary:"10010"},
    {name:"Alice", nationality:{country:"美国",img:"us.gif"},salary:"8000"},
    {name:"Josn", nationality:{country:"美国",img:"us.gif"},salary:"10000.50"},
    {name:"Chapin", nationality:{country:"法国",img:"fr.gif"},salary:"9000.60"},
    {name:"Sargent", nationality:{country:"法国",img:"fr.gif"},salary:"9300"}
];
```

数据中 nationality 是一个对象，包含国家名称和图片名称，示例将定义一个排序类型，让显示国旗图标的列按国家（country）排序。salary 列则要定义一个新类型，让其在显示时加上“￥”符号，但是排序时按数字进行排列。

（2）实现代码

首先使用示例模板创建一个名称为 7-4.html 的页面文件。

为了做一个排序的对比，需要显示两个 Grid，因而要加入两个 div 分别绑定两个 Grid：

```
<div id="div1"></div>
<div id="div2"></div>
```

首先定义新的排序类型，实现按国家排序：

```
Ext.apply(Ext.data.SortTypes, {
    asCountry: function(v){
        return v.country;
    }
});
```

当然，你也可以直接使用“Ext.data.SortTypes.asCountry=function”进行定义，看自己喜好。

接着定义一个新的数据类型“YUAN”：

```
Ext.data.Types.YUAN={
   convert:function(v,data){
        v = (v !== undefined && v !== null && v !== '') ? parseFloat(v) :  0;
           return Ext.String.format("￥{0}",Ext.Number.toFixed(v,2));
   },
    sortType: function(v){
           return parseFloat(v.replace(/￥/,""),10);
    },
```

```
        type: 'yuan'
    };
```

转换函数会将数字转换为以“￥”符号开头，显示小数点后两位的格式化数值。排序类型没有使用自定义类型，而是直接定义了，这充分提现了其灵活性。排序用的数据是替换符号“￥”后的浮点数。

以上代码可以放在 OnReady 内，也可以放在 OnReady 外。接着将数据写到 OnReady 内。

接着要做的是先定义一个函数用来显示国旗图片：

```
function ShowImage(v){
   return "<img src='../images/flag/"+v.img +"' />";
}
```

接着创建使用了 asCountry 排序的 Store 和 Grid：

```
var store1=Ext.create('Ext.data.Store', {
    fields:[
           {name: 'name',type:'string'},
           {name: 'nationality',sortType:"asCountry"},
           {name: 'salary',type:"yuan"}
    ],
    data:data
});
Ext.create('Ext.grid.Panel', {
    title: '使用 asCountry 排序的 Grid',
    store: store1,
    columns: [
        {header:'姓名',dataIndex:'name'},
        {header:'国籍',dataIndex:'nationality',renderer:ShowImage},
        {header:'薪水',dataIndex:'salary',align:'right'}
    ],
    height: 200,
    width: 400,
    renderTo: "div1"
});
```

注意，粗体代码定义了 sortTpye，这样字段就会根据该定义进行排序。

Grid 中第二列的定义了 renderer，目的是使用 ShowImage 函数将数据转换为图片标记输出。

最后创建没有使用 asCountry 排序的 Store 和 Grid：

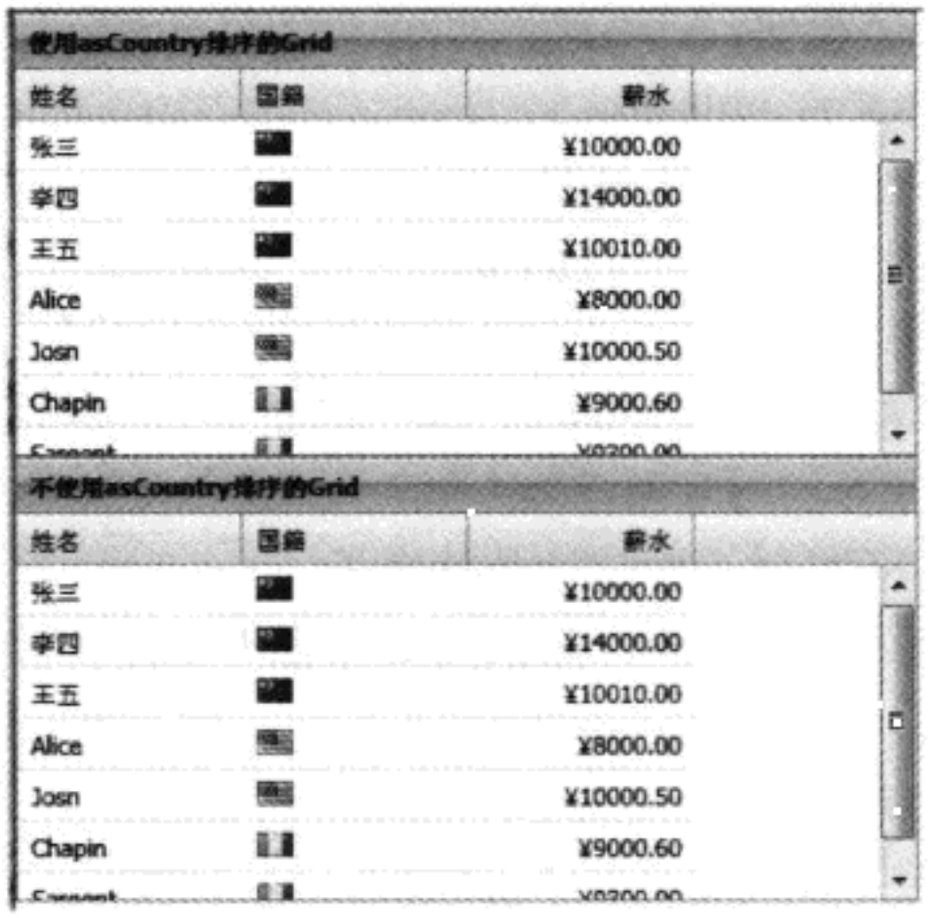
```
var store2=Ext.create('Ext.data.Store', {
    fields:[
           {name: 'name',type:'string'},
           {name: 'nationality'},
           {name: 'salary',type:"yuan"}
    ],
    data:data
});

Ext.create('Ext.grid.Panel', {
    title: '不使用 asCountry 排序的 Grid',
    store: store2,
    columns: [
```

```
        {header:'姓名',dataIndex:'name'},
        {header:'国籍',dataIndex:'nationality',renderer:ShowImage},
        {header:'薪水',dataIndex:'salary',align:'right'}
    ],
    height: 200,
    width: 400,
    renderTo: "div2"
});
```

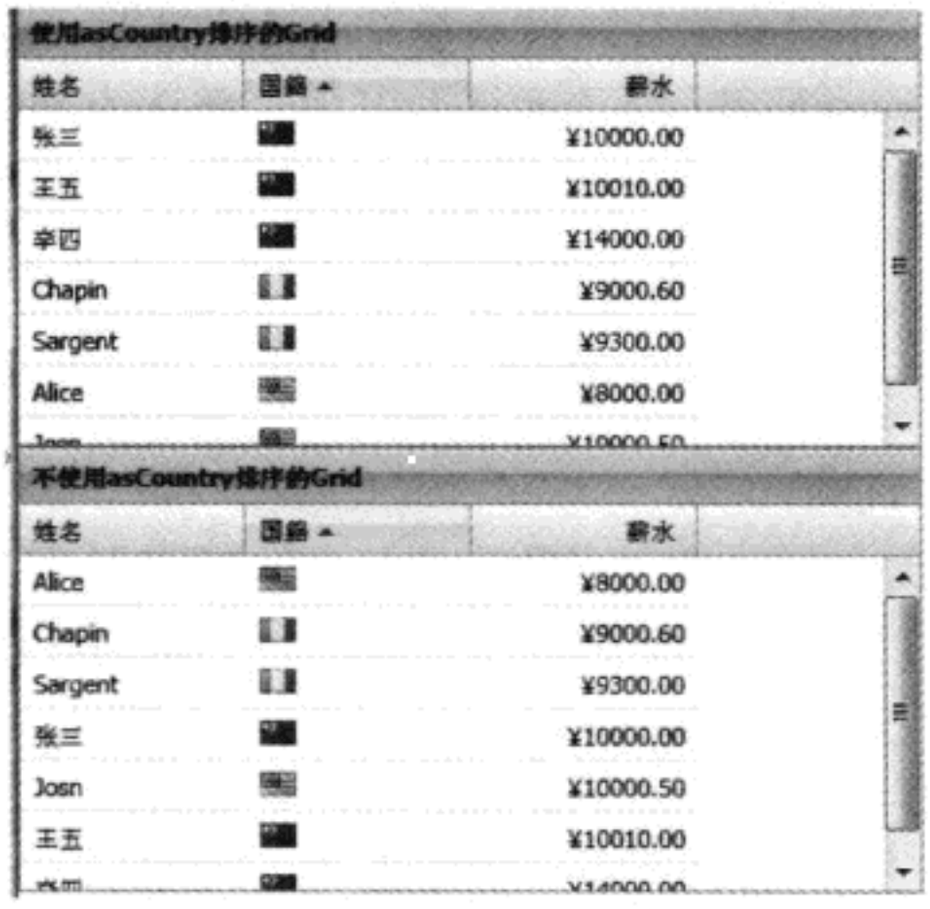
至此，代码就完成了，可以测试了。

（3）示例效果

在浏览器中打开示例，然后分别单击两个 Grid 的薪水列标题，再分别单击两个 Grid 的国籍列标题，将看到如图 7-4 所示的效果。



页面打开时的显示效果



单击国籍后进行排序的显示效果

图 7-4 单击薪水进行排序后的显示效果

要自定义数据类型，必须考虑的问题是，提交给后台的数据格式也是以自定义数据类型的格式提交的，例如本示例中的数据类型 YUAN，其提交的数据是以符号“￥”开头的，因而在后台要注意处理掉符号。

7.4.3 数据模型的骨架——字段：Ext.data.Field

如果数据模型没有字段，那数据模型也就不是模型了。有了字段，才知道数据是做什么用的，它会存储什么样的数据。

Field 对象提供了 10 个配置属性：

- convert：转换函数，默认为数据类型的转换函数，也可以自定义转换函数。
- dateFormat ： 日期格式，这里要注意，定义的日期格式必须与返回的日期的格式一样，不然会造成转换函数无法转换日期，访问不了数据。
- useNull：是否允许 null 值，默认为 false，不允许 null 值。
- defaultValue：字段的默认值，不设置的话，默认为空字符串。
- mapping ： 将数据映射到字段，如果是数组格式，该值为数组的索引。如果是 JSON 数据，该值为关键字。如果是 XML 数据，则该值为节点名称。如果字段的名称是数据中的关键字或节点名称，又或者字段的次序与数组中的数据次序是一样的，则可不指定该值。
- name：字段名称，在任何情况下都要定义，不然就不知道这是什么了。
- persist ： 如果该值为 false，则说明该字段值不允许编辑，数据提交时也不会提交该值。默认值为 true。
- sortDir ： 初始化排序方向，值可以为 ASC（降序排序）或 DESC（升序排序），字母必须全部为大写，默认值得是 ASC。
- srotType：排序类型，值可为 SortTypes 定义的排序函数，或自定义排序函数。
- type：字段类型，值可为 Type 对象定义的类型，如果不设置，默认为 AUTO 类型。

7.4.4 数据集：Ext.util.AbstractMixedCollection 与 Ext.util.MixedCollection

如果用 JavaScript 的数组或对象作为数据集，那么对其排序或通过关键字获取数据都是比较困难的。因而需要重新建立一个数据集对象，可以方便地添加、删除数据，并能对数据进行排序，也可以通过索引或关键字查询数据，MixedCollection 对象就有这样的作用。在 MixedCollection 对象或 AbstractMixedCollection 对象中，数据的关键字是唯一的，这点要注意。

MixedCollection 对象继承于 AbstractMixedCollection 对象，主要是在 AbstractMixedCollection 对象的基础上混合了 Sortable[⊖]对象，为数据集合提供排序功能。要将两个类拆开是因为在 Sortable 对象中要使用 AbstractMixedCollection 对象记录排序中信息，例如排序的字段、该字段的排序类型、排序函数等，这也是一个数据集，但是它不需要排序。

⊖ 相关信息请阅读 7.5.8 节。

MixedCollection 对象主要通过 items 数组、keys 数组和 map 对象来管理数据。items 数组的作用是放置数据，而 keys 数组的作用则是保存数据的关键字，一般是其 id 属性值。而 map 对象的作用是作为一个映射表将 keys 中的关键字指向 items 中的数据。这样的好处是，无论 items 中的数组的顺序如何变化，都可以通过关键字快速定位到数据。可以观察 AbstractMixedCollection 对象的 add 方法，看数据是如何保存到这些对象的，其代码如下：

```
add : function(key, obj){
    var me = this,
        myObj = obj,
        myKey = key,
        old;

    if (arguments.length == 1) {
        myObj = myKey;
        myKey = me.getKey(myObj);
    }
    if (typeof myKey != 'undefined' && myKey !== null) {
        old = me.map[myKey];
        if (typeof old != 'undefined') {
            return me.replace(myKey, myObj);
        }
        me.map[myKey] = myObj;
    }
    me.length++;
    me.items.push(myObj);
    me.keys.push(myKey);
    me.fireEvent('add', me.length - 1, myObj, myKey);
    return myObj;
},
```

如果参数只有 1 个，说明没有提供关键字，需要使用 getKey 方法从对象中提取关键字，也就是提取对象的 id 值而已。如果在 map 对象中已经存在与关键字对应的对象，则使用 replace 方法替换该对象。其实就是用 indexOfKey 方法定位到 items 内的对象，然后替换掉对象，并替换掉 map 对象里关键字指向的对象。

如果参数不存在，则先在 map 对象内添加对象。然后计数器 length 加 1，将对象加入 items 数组，将关键字加入 keys 数组。最后触发 add 事件，并返回对象。

AbstractMixedCollection 对象主要就是操作 items 数组、keys 数组和 map 对象，主要有以下 25 个操作方法：

- add：增加一个条目到数据集。
- addAll：将数组或对象的数据添加到数据集。
- getKey ：返回对象的关键字，一般情况下，返回的是对象的 id。也可以重定义该函数，根据需要的方式返回关键字。
- replace：替换数据集中的条目。
- each：枚举数据集中的条目。
- eachKey：枚举数据集的关键字。
- findBy：返回指定的函数运算后返回 true 值的第一条目。

- insert：在指定位置插入一个条目。
- remove：删除一个条目。
- removeAll：删除数组内指定的数据。
- removeAt：删除指定位置的条目。
- removeAtKey：删除指定关键字的条目。
- getCount：返回数据总数。
- indexOf：返回指定条目的位置。
- indexOfKey：返回指定关键字的位置。
- get：根据指定关键字或索引返回条目。
- getAt：根据索引返回条目。
- contains：如果数据集包含指定的条目，返回 true。
- containsKey：如果指定的关键字在数据集的关键字集合中，返回 true。
- clear：删除所有数据。
- first：返回数据集的第一个条目。
- last：返回数据集的最后一个条目。
- sum：合计数据集中指定的属性的值。
- collect：在数据集中收集指定属性的唯一值。
- getRange：返回指定范围的数据。

MixedCollection 对象在 AbstractMixedCollection 对象的基础上添加了 4 个排序方法：

- sort：根据指定的字段进行排序。
- sortBy：使用指定的函数进行排序。
- sortByKey：根据关键字进行排序。
- reorder：基于旧索引到新索引的映射进行重新排序。

MixedCollection 对象是 Ext JS 中一个很重要的对象，很多数据结构都是以 MixedCollection 对象为基准的，因而掌握好该对象及其属性和方法相当重要。

7.4.5 数据验证及错误处理：Ext.data.validations 与 Ext.data.Errors

数据验证对象 validations 是新增的对象，其作用是当数据模型不使用表单时，为模型提供验证机制，例如直接使用 Grid 修改数据并提交，就不需要表单验证。

目前 validations 对象提供 5 种验证方式：

- presence：验证值是否存在。如果值存在，返回 true。
- length：验证值是否在指定 min 和 max 值之间。如果值在 min 和 max 之间，返回 true。
- format：验证值是否符合正则表达式给定的格式。如果符合，返回 true。
- inclusion：验证值是否在 list 指定的数组内。如果存在，返回 true。
- exclusion：验证值是否在 list 指定的数组内。如果不存在，返回 true。

因为没有表单的错误处理机制，所以必须使用另外的错误机制处理验证，而这就是 Errors 对象的工作。Errors 对象继承自 MixedCollection 对象，因而一个错误会作为数据集的一个条

目存在。Errors 对象提供了两个检查错误的方法：

- isValid：如果没有错误，该方法返回 true。
- getByField：根据指定的字段名称返回该字段的所有错误信息。

下面实践一下如何使用验证以及进行错误处理，在浏览器打开 7.4.2 节示例，然后在控制台先定义一个待验证的模型：

```
Ext.define('Feedback', {
    extend: 'Ext.data.Model',
    fields: [
        {name: 'name',type: 'string'},
        {name: 'email',  type: 'string'},
        {name: 'types',  type: 'string'},
        {name: 'content',type: 'string'},
    ],
    validations: [
        {type: 'presence',  field: 'name'},
        {type: 'presence',  field: 'email'},
        {type: 'presence',  field: 'types'},
        {type: 'presence',  field: 'content'},
        {type: 'length',    field: 'content',max: 400},
        {type: 'inclusion', field: 'types', types: ['投诉', '问题','建议']},
        {type: 'format',     field: 'email', matcher: /^(\w+)([\-+.][\w]+)*@(\w[\-
            \w]*\.){1,5}([A-Za-z]){2,6}$/}
    ]
});
```

代码定义了一个反馈信息模型，包括反馈人姓名（name）、反馈类型（types）、反馈人电子邮件（email）和反馈内容 4 个字段。在 validations 中，定义了一个验证数组，包括了 4 个字段不能为空，内容字符长度不能超过 400 个字符，反馈类型必须是投诉、问题或建议等值中的一个以及电子邮件必须符合格式等验证信息。

下面创建一个空数据，然后获取错误信息：

```
var fb = Ext.ModelManager.create({}, 'Feedback');
var err = fb.validate();
console.log(err.isValid());
```

控制台将显示“false”。

继续输入以下命令获取与电子邮件有关的错误信息：

```
console.dir(err.getByField("email"));
```

控制台将看到以下信息：

```
0  Object { field="email", message="must be present"}
1  Object { field="email", message="is the wrong format"}
```

在本地化文件中，没有将 validations 对象的信息本地化，因而 message 显示的还是英文信息，如果需要，你可以自己添加。打开文件 ext-lang-zh_CN.js，然后添加以下代码即可：

```
if(Ext.data.validations){
   Ext.apply(Ext.data.validations,{
```

```
            presenceMessage: '不能为空',
            lengthMessage: '的字符长度不符合要求',
            formatMessage: '的格式错误',
            inclusionMessage: '的值不在规定的值内',
            exclusionMessage: '的值是不允许的',
        });
    }
```

不过这样的信息直接显示也不太准确，还是需要自己处理一下。

使用 MixedCollection 对象的 each 方法，可以枚举错误信息，在命令行输入以下命令：

```
err.each(function(item,index,len){
    console.log(item);
});
```

在控制台将看到以下输出：

```
Object { field="name", message="不能为空"}
Object { field="email", message="不能为空"}
Object { field="types", message="不能为空"}
Object { field="content", message="不能为空"}
Object { field="types", message="的值不在规定的值内"}
Object { field="email", message="的格式错误"}
```

也可以直接使用 for 循环遍历错误信息，在命令行输入以下代码：

```
for(var i=0;ln=err.items.length,i<ln;i++){
    console.log(err.items[i]);
}
```

在控制台的输出结果与 each 方法是一样的。

不用说，validations 对象与数据类型对象一样也是可以扩展的，例如要添加一个验证电子邮件的方法，可在命令行中输入以下代码：

```
Ext.apply(Ext.data.validaticns,{
    emailMessage:"请输入正确的电子邮件",
    email:function(config,value){
        return  !!(/^(\w+)([\-+.][\w]+)*@(\w[\-\w]*\.){1,5}([A-Za-z]){2,6}$/.
            test(value));
    }
});
```

代码中要注意的是验证的错误提示信息的属性名称是验证的方法名称加上“Message”，例如代码中方法名称为 email，则提示信息的属性名称为 emailMessage。

修改一下 Feedback 的定义，将 validations 中 type 值为 format 的验证定义中的 format 修改为 email，就可以使用 email 方法进行验证了。

7.4.6 模型的关系：Ext.data.Association、Ext.data.HasManyAssociation 和 Ext.data.BelongsToAssociation

熟悉关系数据库的开发人员对数据之间的关系应该不会陌生。Association 对象的目的就

是在数据模型之间关联。根据不同的关系，从 Association 对象派生出了 HasManyAssociation 对象和 BelongsToAssociation 对象。HasManyAssociation 对象的作用是实现一对多的关系，而 BelongsToAssociation 对象的作用是实现多对一的关系。

Association 对象的重点是提供了一个静态的 Create 方法，用于创建 HasManyAssociation 对象或 BelongsToAssociation 对象的实例。代码很简单，在这里不列出来了。

HasManyAssociation 对象是一对多的关系，也就是常说的主从关系，因而该对象会为子数据创建一个 Store，用来存放和访问子数据。

BelongsToAssociation 对象是多对一关系，也就是它是某个主模型的子模型，因而其主要的操作就是返回父模型。因此，它会生成 get 和 set 两个方法，用来获取或设置父模型的数据。方法的名称默认是 get 或 set 加上关联的数据模型名称，例如，关联的模型名称为 Order，那么生成的方法名称就是 getOrder 和 setOrder。在 getOrder 方法，可能会使用 Ajax 加载数据，因而可以在方法内加入 Ajax 的回调函数 callback、success 和 failure，还可以设置作用域（scope）。

要定义数据模型的关系，标准的做法是在模型中使用 associations 配置项，例如：

```
Ext.define("Order",{
    …,
    associations:[
    {// 关系配置对象 1},
    {// 关系配置对象 2},
    …
    ]
})
```

关系的配置对象一般包括以下配置项：

- type：关系类型，值可以为 hasMany 或 belongsTo。
- model：关联的模型名称。
- primaryKey：可选项，为主键，默认值是 id。
- foreignKey：可选项，为外键，默认值是模型名称后加“_id”。
- autoLoad：可选项，是否自动加载，默认值为 false，不自动加载。
- associationKey：可选项，在数据中读取关系数据的属性名称。如果没有定义，默认是关系模型的名称的复数，例如，关系模式的名称为 Order，则可通过 Orders 属性读取关系模型的数据。
- filterProperty：可选项，这是 HasManyAssociation 对象特有的配置项，主要作用不是通过外键查询子数据，而是通过过滤的方式查询子数据。这个会比较难理解，把它看做是过滤后数据就容易理解了，事实上也是这样，只是通过关系来实现而已。

除了使用 associations 配置项来配置关系外，还可以直接使用 belongsTo 或 hasMany 配置项，这两个配置项可以是字符串、数组或关系配置对象。为字符串时，字符串为数据模型的名称；为数组时，数组项可以是数据模型的名称；还可以是关系配置对象。关系配置对象的配置项与 associations 的关系配置对象的配置项是一样的。

7.4.7 管理数据模型：Ext.AbstractManager 与 Ext.ModelManager

ModelManager 对象的作用是注册数据模型以及记录实例化的数据模型，以实现对数据模型及其实例的管理。具有相似管理功能的还有 PluginManager 对象和 ComponentManager 对象，因而就有了为这些管理对象提供的 AbstractManager 对象。

AbstractManager 对象提供了 HashMap[⊖]对象和 types 对象这两种方式来注册并管理对象。HashMap 对象记录的是实例化的对象。这时候，实例化对象的 id 就是关键字，通过 id 就可以找到这个实例化对象。而 types 对象记录由 Ext.define 定义的对象，例如定义的数据模型会存储在 types 对象中，通过数据模型名称就可以检索到该数据模型。

AbstractManager 对象提供了 9 个方法用来操作 HashMap 对象和 types 对象：

- get：通过 id 获取实例化的对象（HashMap 对象）。
- register：注册实例化的对象（HashMap 对象）。
- unregister：在管理器中注销实例化的对象（HashMap 对象）。
- registerType：注册对象（types）。
- isRegistered：检查对象是否已经注册，如果已注册返回 true（types）。
- create：在 types 对象中找到对象，并实例化该对象。
- onAvailable：注册一个函数，该函数会在向管理器添加一个实例化对象时执行。
- each：枚举实例化的对象。枚举函数会依次序接收 key、value 和 length 三个参数。其中 key 是实例化对象在 HashMap 对象中的关键字，value 就是实例化对象，length 是 HashMap 对象中条目的总数。
- getCount：返回 HashMap 对象的条目总数。

ModelManager 对象在 AbstractManager 对象增加和重写了以下 6 个方法：

- registerType：注册定义的数据模型。
- onModelDefined：模型定义后触发的回调函数，主要用来处理关系。
- registerDeferredAssociation：如果关系模型不存在，会在 associationStack 数组中保存该关系，以便关系模型注册后再注册关系。
- getModel：根据模型名称返回模型。
- create：实例化模型，也就是创建一条记录。
- regModel：注册模型，实际上是调用 registerType 注册模型。

7.4.8 定义数据模型：Ext.data.Model

说是定义数据模型而不是创建，是因为数据模型是使用 Ext.define 定义的一个数据模型类，而不是使用 Ext.create 创建的数据模型的实例。数据模型与其他类的区别是它直接或间接继承自 Model 类。直接继承容易理解，就是在定义数据模型时加入“extend: 'Ext.data.Model'”这个配置项。间接继承的意思是当前类的父类或父类的父类是继承于 Model 类的，例如，首先从 Model 类派生出车这个类，然后又从车派生出汽车、卡车或火车等类。

⊖ 相关信息请阅读 16.4.7 节。

定义数据模型使用的是 Ext.define 方法，因而其语法格式可阅读 2.7.7 节使用 Ext.define 定义新类一节。当然了，数据模型是一个比较特殊的类，因而有以下一些特殊配置项：

- extend：如果是直接继承于 Model 类，其值为“Ext.data.Model”；如果是间接继承 Model 类，则其值为父类的类名，例如定义了 Vehicle 类，则 Truck 类的 extend 值为 Vehicle。
- fields：必须定义的项，为数据模型的字段，它是一个由 Field 对象组成的数组。如果字段没有特别要求，可以直接以字段名称为一个数组条目。如果包含字段类型（type）、默认值（defaultValue）等内容，则必须以 JSON 对象为数组条目，具体的字段属性可阅读 7.5.4 节。
- validations：可选项，为数组，在这里可定义数据模型的验证项，具体信息可阅读 7.4.5 节。
- associations：可选项，为数组，在这里可定义数据模型的关系，具体信息可阅读 7.4.6 节。
- belongsTo：可选项，可为字符、数组或对象，在这里可定义数据模型的多对一关系，具体信息可阅读 7.4.6 节。
- hasMany：可选项，可为字符、数组或对象，在这里可定义数据模型的一对多关系，具体信息可阅读 7.4.6 节。

除了以上的配置项外，与类定义一样，还可以为模型添加必要的方法，例如添加一些处理字段值的方法。

7.4.9 数据模型的定义过程

因数据模型是使用 Ext.define 定义的类，因而其定义过程与 4.4 节介绍的类的创建过程基本是一样的。不过要处理验证、关联、字段等信息还是有区别的，主要区别在 Model 类定义了 onClassExtended 方法，根据图 4-5 类创建的流程图，它会在 Class 对象的处理器完成工作后执行，其代码如下：

```
onClassExtended: function(cls, data) {
    var onBeforeClassCreated = data.onBeforeClassCreated;
    data.onBeforeClassCreated = function(cls, data) {
        var me = this,
            name = Ext.getClassName(cls),
            prototype = cls.prototype,
            superCls = cls.prototype.superclass,
            validations = data.validations || [],
            fields = data.fields || [],
            associations = data.associations || [],
            belongsTo = data.belongsTo,
            hasMany = data.hasMany,
            fieldsMixedCollection = new Ext.util.MixedCollection(false,
                function(field) {
                return field.name;
            }),
            associationsMixedCollection = new Ext.util.MixedCollection(false,
                function(association) {
                return association.name;
            }),
```

```
        superValidations = superCls.validations,
        superFields = superCls.fields,
        superAssociations = superCls.associations,
        association, i, ln,
        dependencies = [];
    cls.modelName = name;
    prototype.modelName = name;
    if (superValidations) {
        validations = superValidations.concat(validations);
    }

    data.validations = validations;

    if (superFields) {
        fields = superFields.items.concat(fields);
    }
    for (i = 0, ln = fields.length; i < ln; ++i) {
        fieldsMixedCollection.add(new Ext.data.Field(fields[i]));
    }
    data.fields = fieldsMixedCollection;
    if (belongsTo) {
        belongsTo = Ext.Array.from(belongsTo);
        for (i = 0, ln = belongsTo.length; i < ln; ++i) {
            association = belongsTo[i];
            if (!Ext.isObject(association)) {
                association = {model: association};
            }
            association.type = 'belongsTo';
            associations.push(association);
        }
        delete data.belongsTo;
    }
    if (hasMany) {
        hasMany = Ext.Array.from(hasMany);
        for (i = 0, ln = hasMany.length; i < ln; ++i) {
            association = hasMany[i];
            if (!Ext.isObject(association)) {
                association = {model: association};
            }
            association.type = 'hasMany';
            associations.push(association);
        }
        delete data.hasMany;
    }
    if (superAssociations) {
        associations = superAssociations.items.concat(associations);
    }
    for (i = 0, ln = associations.length; i < ln; ++i) {
        dependencies.push('association.' + associations[i].type.
            toLowerCase());
    }
    if (data.proxy) {
        if (typeof data.proxy === 'string') {
            dependencies.push('proxy.' + data.proxy);
        }
```

```
        else if (typeof data.proxy.type === 'string') {
            dependencies.push('proxy.' + data.proxy.type);
        }
    }
    Ext.require(dependencies, function() {
        Ext.ModelManager.registerType(name, cls);
        for (i = 0, ln = associations.length; i < ln; ++i) {
            association = associations[i];
            Ext.apply(association, {
                ownerModel: name,
                associatedModel: association.model
            });
            if (Ext.ModelManager.getModel(association.model) === undefined) {
                Ext.ModelManager.registerDeferredAssociation(association);
            } else {
        associationsMixedCollection.add(Ext.data.Association.
            create(association));
            }
        }
        data.associations = associationsMixedCollection;
        onBeforeClassCreated.call(me, cls, data);
        cls.setProxy(cls.prototype.proxy || cls.prototype.defaultProxyType);
        Ext.ModelManager.onModelDefined(cls);
    });
};
},
```

要做的事情很多，所以代码比较长。不过重点主要如下：

- 为字段创建一个数据集（fieldsMixedCollection）。
- 为关系创建一个数据集（associationsMixedCollection）。
- 合并父类的验证项。间接继承时，父类可能会定义一些验证项，因而子类必须继承这些验证项。
- 合并父类的字段。
- 将字段添加到字段数据集。
- 将分开定义的一对多关系或多对一关系全部统一到 association 定义中。在这里，笔者建议定义关系还是使用 association 定义比较好，可避免不必要的操作。
- 合并父类的关系。
- 根据关系，确定当前数据模型依赖的数据模型。
- 动态加载依赖的数据模型。
- 加载完成后，就使用 ModelManager 对象的 registerType 方法注册数据模型，以便查询。然后检查依赖的关系模型是否存在，如果存在，创建一个关系实例并加入关系数据集。如果不存在，则使用 ModelManager 对象的 registerDeferredAssociation 方法延迟依赖关系模型的注册。
- 最后是执行 ModelManager 的 onModelDefined 方法。

从以上的分析可以了解到模型包含两个重要的数据集（MixedCollection 对象），一个是记录字段信息的 fields 属性，一个是记录关系信息的 associations 属性。定义的每个数据模型都

会在 ModelManager 对象内进行注册。

7.4.10 数据模型的创建

创建一个数据模型实例，就是创建一个记录，必须牢记，这些关系搞不清楚，对了解 Store 会有障碍。

我们先来研究一下模型实例是如何创建的。模型是继承自 Model 对象的，如果没有重写构造函数，则在创建实例时会执行 Model 对象的构造函数，其代码如下：

```
constructor: function(data, id, raw, convertedData) {
    data = data || {};

    var me = this,
        fields,
        length,
        field,
        name,
        value,
        newId,
        persistenceProperty,
        i;

    me.internalId = (id || id === 0) ? id : Ext.data.Model.id(me);
    me.raw = raw;
    if (!me.data) {
        me.data = {};
    }
    me.modified = {};
    if (me.persistanceProperty) {
          //省略调试代码
        me.persistenceProperty = me.persistanceProperty;
    }
    me[me.persistenceProperty] = convertedData || {};
    me.mixins.observable.constructor.call(me);
    if (!convertedData) {
        fields = me.fields.items;
        length = fields.length;
        i = 0;
        persistenceProperty = me[me.persistenceProperty];

        if (Ext.isArray(data)) {
            for (; i < length; i++) {
                field = fields[i];
                name  = field.name;
                value = data[i];

                if (value === undefined) {
                    value = field.defaultValue;
                }
                if (field.convert) {
                    value = field.convert(value, me);
                }
                persistenceProperty[name] = value ;
```

```
        }

    } else {
       for (; i < length; i++) {
            field = fields[i];
            name  = field.name;
            value = data[name];
            if (value === undefined) {
                value = field.defaultValue;
            }
            if (field.convert) {
                value = field.convert(value, me);
            }
            persistenceProperty[name] = value ;
        }
    }
}
me.stores = [];
if (me.getId()) {
    me.phantom = false;
} else if (me.phantom) {
    newId = me.idgen.generate();
    if (newId !== null) {
        me.setId(newId);
    }
}
me.dirty = false;
me.modified = {};
if (typeof me.init == 'function') {
    me.init();
}
me.id = me.idgen.getRecId(me);
},
```

代码不少，但是重点有以下几个：

- 为每个模型实例创建一个 internalId 来识别实例，该 id 也可以在创建实例时自定义。
- 如果是使用 Reader 对象读取的数据，则使用 raw 属性指向原始数据。
- 定义 modified 对象，用来记录更改过的字段及其旧值。
- 定义一个对象用来记录字段及其值，如果没有修改 persistanceProperty 属性值，则是 data，也就是通过访问模型实例的 data 属性就可以访问到模型实例的数据。后面的循环会将字段名称作为关键字、默认值作为值，加入到对象中。如果创建实例时带数据，则使用 set 方法替换字段的值。
- 定义 stores 数组，用来记录模型绑定到了哪些 Store。
- 如果实例存在 id，则说明是与服务器端数据同步的，因而设置 phantom 属性为 false。否则根据 id 设置 phantom 的值。
- 设置 dirty 属性为 false，表示数据未被修改。
- 如果定义了 init 方法，执行 init 方法。
- 最后生成模型的 id。

从以上代码分析中，可以知道，模型实例的数据可以在 data（没有修改 persistanceProperty

属性，也不建议修改）属性中找到，通过字段名称就可从对象中获取该字段的值。通过 modified 对象，可知道哪些字段被修改过。

创建数据模型，也就是实例化数据模型，一般有以下三种方法：

```
Ext.create(ModeName,config);
new ModeName(config);
Ext.ModelManager.create(config,ModeName);
```

其中 Modename 为模型名称，config 为配置对象。在配置对象里可为模型的字段设置值。

例如，打开 7.4.2 节示例，先在命令行执行 7.5.5 节的 Feeback 定义，然后在命令行中输入如下命令：

```
var f1=Ext.create("Feedback",{name:"王六"});
var f2=new Feedback({name:"王六"});
var f3=Ext.ModelManager.create({name:"王六"},"Feedback");
console.log(f1)
console.log(f2)
console.log(f3)
```

将会在控制台看到以下输出：

```
[Trial] Feedback <Feedback> { id="Feedback-ext-record-15", internalId="ext-record-15"}
[Trial] Feedback <Feedback> { id="Feedback-ext-record-16", internalId="ext-record-16"}
[Trial] Feedback <Feedback> { id="Feedback-ext-record-17", internalId="ext-record-17"}
```

因为没有定义 id，为了保持记录的唯一性，Ext JS 会自动为每条记录添加一个 id。

模型还提供了一个通过 Proxy 加载数据生成实例的静态方法 load，例如，要通过远程为 Feedback 加载数据，可以执行以下代码：

```
var FB=Ext.ModelManager.getModel("Feedback");
var fb;
FB.load(1,{
    url:'fb.js',
    success:function(m){
        fb=m;
        console.log(fb)
    },
    scope:this
})
```

如果不能确定当前作用域是否可以访问到模型对象，最好还是使用 ModelManager 对象返回模型，然后再使用 load 方法。load 方法的第 1 个参数是查询数据用的唯一值，是定义模型时定义的 id 属性，如果没有定义，默认值是 id。第 2 个参数就是 Ajax 的配置对象了。因为没有配置 Proxy 对象，所以默认使用 JSONReader 获取数据，因而数据的返回格式是 JSON 格式，例如：

```
{
    id:1,
    name:"刘六"
}
```

JOSN 的属性名称为字段名称，属性值为数据库中的值就行了。将以上代码保存为 fb.js。然后运行以上的代码，会在控制台看到以下输出：

```
GET http://localhost/ext-book3/chapter7/fb.js?_dc=1307255617023&id=1 200 OK 18ms
    ext-all-debug.js (第 20573 行)
undefined
[Trial] Feedback <Feedback> { id="Feedback-1", internalId=1}
```

注意发送请求，有个查询字符串是“id=1”，这就是 load 方法定义的第 1 个参数值与 id 属性组合的查询字符串。根据 id 的值，就可以从数据库中获取数据，并将数据组合成 JSON 格式返回。

从 console.log 语句输出的结果，可以知道 success 函数的第 1 个参数返回的就是一个模型实例。

模型创建或修改后，可以使用 save 方法提交到服务器，但前提是必须设置了 Proxy 对象。

7.4.11 数据模型的配置项、属性和方法

从上面的介绍可以知道，一个实例化的模型就是一条记录，因而需要提供一些属性和方法来操作记录，并记录其状态。

1. 配置项

- idProperty：用于定义模型的唯一 id，其值为 fields 中定义的字段名称，默认值为 id。
- persistenceProperty：模型内用于访问数据的属性名称，默认值为 data。

2. 属性

- defaultProxyType：设置代理类型，默认类型是 Ajax。
- dirty：只读属性，布尔值，返回值为 true 时表示记录已经被修改。
- editing：只读属性，布尔值，如果为 true，表示模型实例正在被编辑。
- fields：只读属性，为 MixedCollection 对象，可从中获取字段信息。
- modified：只读属性，一个保存了已编辑过的字段及修改值的对象，在这里可以检查哪些字段被修改了。对象的属性名称就是字段名称。
- phantom：只读属性，如果记录在数据库中不存在，此属性为 true。任何记录只要有一个数据库主键作为其 id 属性，那么它就不是 phantom，phantom 属性为 false。
- raw：只读属性，如果是使用 reader 读取的数据，使用该属性可查询读取时的原始数据。
- store：只读属性，使用该属性可以获取模型实例所属的 Store。

3. 方法

- get：返回指定字段的值。
- set：设置指定字段的值。可以只设置 1 个字段，也可以使用对象格式数据一次设置几个字段的值。**如果关联了 UI，该方法会自动刷新 UI 的显示。**
- isEqual：检查两个值是否相等。
- beginEdit：进入编辑状态。

- cancelEdit：取消所有改变。
- endEdit：结束编辑状态。
- getChanges：获取被修改过的值。就是将 modified 对象的成员复制到一个新对象返回。
- isModified：检查指定字段是否被修改过。
- setDirty：标记记录已被修改。
- reject：取消所有修改，字段值恢复到最后一次 commit 操作的状态或原始状态。
- commit：确认修改，要注意，确认修改后无法恢复之前的值。
- copy：复制当前模型实例。
- setProxy：设置 Proxy 对象。
- getProxy：返回 Proxy 对象。
- validate：校验模型的数据。
- isValid：检查模型是否通过了验证。返回 true 表示模型数据已验证，没有错误。
- save：使用 Proxy 对象提交数据到服务器端进行保存操作。
- destroy：使用 Proxy 对象通知服务器删除当前模型实例对应的数据。
- getId：返回模型实例的 id 字段的值。
- setId：设置模型实例的 id 字段值。
- join：设置模型实例的 store 属性。表示该模型实例添加到一个 store 里。
- unjoin：删除模型实例的 store 属性。表示该模型实例已被移除 store。
- getAssociatedData：获取模型实例的关系数据。

7.5 Store

7.5.1 概述

在 Ext JS 4 中 有 AbstractStore、Store、ArrayStore、JsonStore、JsonPStore、XmlStore、DirectStore、PropertyStore、TreeStore 和 NodeStore 10 个对象，它们的关系如图 7-5 所示。

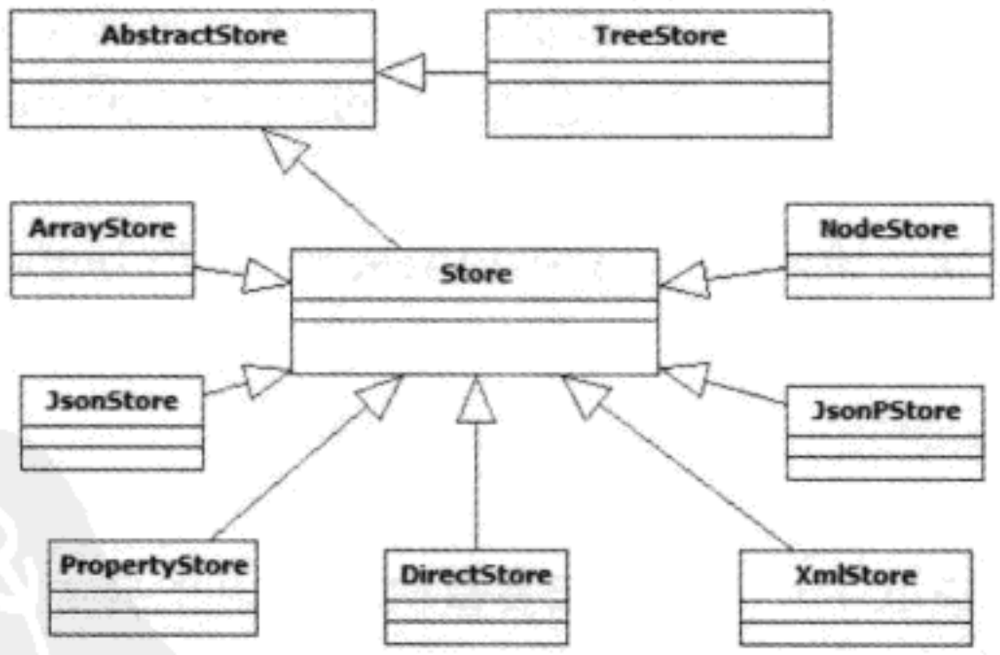


图 7-5 Store 之间的关系

为什么要使用 Store？初学者一般都有这样的疑问。是因为 Store 的一个重要作用，这也是很多初学者容易搞混淆的问题，就是把数据与 UI 组件分离。这样的好处，了解三层架构的都会明白的。如果不明白，就好好补上三层架构这一课。清楚这一点非常重要，不然会出现太多问题，例如经常有初学者问怎么获取 Grid 中某个单元格的数据，怎么更新单元格里的数据等问题。因而，笔者不得不再强调一次，**Ext JS 的是采用三层架构的，数据与 UI 组件是分离的，要访问的数据在 Store 里，不在 UI 组件里。因而，要操作数据，请使用 Store 对象的属性和方法**。不过 Store 对象与 TreeStore 对象有点差异，在 Store 对象中，数据在 data 属性中的，而 TreeStore 对象的数据是在 tree 属性中。

还有一点也非常重要，一般来说，数据模型的实例是一条记录，而 Store 对象就是这些记录的集合。Store 对象是使用 MixedCollection 对象存储这些模型实例的，因而可以通过 MixedCollection 对象的属性和方法访问 Store 对象的数据。对于 TreeStore 对象，模型实例是一个节点（数据节点，不是树 UI 中的节点，也不是 DOM 中的节点，7.5 节内的节点没特别说明都是指数据节点），而节点是有层次的，因而其操作必须便于节点操作，所以要使用 TreeData 对象存储这些节点，而不能用 MixedCollection 对象。

从图 7-5 可以看到，TreeStore 对象不是从 Store 对象派生的，而是直接从 AbstractStore 派生的，其原因就是因为 TreeStore 对象的模型实例是一个节点，且存储结构为 TreeData 对象。

NodeStore 对象让笔者产生了困惑，API 里说是从 AbstractStore 派生的，但是类定义的时候却是从 Store 派生的，不过它运行得很好，因而笔者还是根据定义划分到 Store 对象下。NodeStore 对象是一个很有趣的对象，它的 data 属性是 MixedCollection 对象，记录着树列表中可见的节点，而其 ncde 属性就是树的跟节点，有着节点的属性和方法，通过其可遍历树中的所有节点。这样的目的是为了方便操作，无须从全部节点中找出可见的节点，从而方便操作。这个特性主要是方便显示，因而其一般用在 TreeView 对象。

从 ArrayStore、JsonStore、XmlStore 和 DirectStore 的名字就可以知道，这是由于处理数据的方法不同而派生的对象，实际上也是这样，它们的源代码也就是设置了不同的 Proxy、Reader 或 Writer 对象。

PropertyStore 对象是为 PropertyGrid 打造的 Store，因其数据结构只有属性名称和值两个字段，因而其数据模型是固定的，为 Ext.grid.property.Property。其数据操作方式与 Store 对象也有些小区别，因而重写了 Store 对象的一些方法，而其余的操作则与 Store 对象没有区别。

7.5.2 Store 对象的实例化过程

Store 对象的构造函数如下：

```
constructor: function(config) {
    config = Ext.Object.merge({}, config);

    var me = this,
        groupers = config.groupers || me.groupers,
        groupField = config.groupField || me.groupField,
        proxy,
        data;
```

```
if (config.buffered || me.buffered) {
    me.prefetchData = new Ext.util.MixedCollection(false, Ext.data.Store.
        recordIndexFn);
    me.pendingRequests = [];
    me.pagesRequested = [];

    me.sortOnLoad = false;
    me.filterOnLoad = false;
}
data = config.data || me.data;

me.data = new Ext.util.MixedCollection(false, Ext.data.Store.recordIdFn);

if (data) {
    me.inlineData = data;
    delete config.data;
}

if (!groupers && groupField) {
    groupers = [
        {
            property : groupField,
            direction: config.groupDir || me.groupDir
        }
    ];
}
delete config.groupers;

me.groupers = new Ext.util.MixedCollection();
me.groupers.addAll(me.decodeGroupers(groupers));

this.callParent([config]);

if (me.groupers.items.length) {
    me.sort(me.groupers.items, 'prepend', false);
}

proxy = me.proxy;
data = me.inlineData;
if (!me.buffered && !me.pageSize) {
    me.pageSize = 25;
}

if (data) {
    if (proxy instanceof Ext.data.proxy.Memory) {
        proxy.data = data;
        me.read();
    } else {
        me.add.apply(me, [data]);
    }

    me.sort();
    delete me.inlineData;
} else if (me.autoLoad) {
```

```
            Ext.defer(me.load, 10, me, [typeof me.autoLoad === 'object' ? me.autoLoad:
                undefined]);
        }
    },
```

代码先使用 Ext.Object 对象的 merge 方法将配置对象和空对象合并。使用 merge 方法的目的是克隆一份配置对象，保留原始的配置对象。

如果设置了 buffered 为 true，则要创建一个预取数据的对象：MixedCollection 对象实例。还要创建两个数组 pendingRequests 和 pagesRequested，pendingRequests 数组用来记录预取请求，pagesRequested 用来记录预取的页数。

接着把配置对象中的配置项 data 的数据保存到 data 属性中，不然下面的操作会把数据丢失。

接着是**最重要的一个步骤，将 data 属性指向一个 MixedCollection 对象实例。Reader 在读取数据后，会将读取数据转换后保存到该对象中，也就是说，通过 data 属性可以访问和操作 Store 中的数据，这很重要。**

接着是将配置对象中定义的数据放到 inlineData 属性中，并删除配置对象的成员 data。

接着是规范化分组数据，将其转换成数组，数组由分组对象组成。每个分组对象包括 property 和 direction 这两个成员，property 记录分组的字段名称，direction 记录排序方向。

删除配置对象的 groupers 成员后，又创建一个 MixedCollection 对象实例用来记录分组信息，并将刚才规范化后的分组数据编码后添加到实例中。通过 groupers 属性就可以操作这些分组信息。

接着是调用父类的构造函数了，也就是 AbstractStore 对象的构造函数，其代码如下：

```
    constructor: function(config) {
        var me = this,
            filters;

        Ext.apply(me, config);
        me.removed = [];
        me.mixins.observable.constructor.apply(me, arguments);
        me.model = Ext.ModelManager.getModel(me.model);
        Ext.applyIf(me, {
            modelDefaults: {}
        });
        if (!me.model && me.fields) {
            me.model = Ext.define('Ext.data.Store.ImplicitModel-' + (me.storeId || Ext.
id()), {
                extend: 'Ext.data.Model',
                fields: me.fields,
                proxy: me.proxy || me.defaultProxyType
            });

            delete me.fields;

            me.implicitModel = true;
        }
// 省略调试代码
        me.setProxy(me.proxy || me.model.getProxy());
        me.proxy.on('metachange', me.onMetaChange, me);
```

```
if (me.id && !me.storeId) {
    me.storeId = me.id;
    delete me.id;
}
if (me.storeId) {
    Ext.data.StoreManager.register(me);
}
me.mixins.sortable.initSortable.call(me);
filters = me.decodeFilters(me.filters);
me.filters = Ext.create('Ext.util.MixedCollection');
me.filters.addAll(filters);
},
```

代码先将配置对象的成员复制到 Store 对象实例。

接着创建一个 removed 数组用来记录移除的数据。

接着指向 Observable 对象的构造函数初始化 Observable 对象并将其混入 Store 实例。

接着从模型管理器中获取模型对象。然后定义一个 modelDefaults 对象并复制到 Store 实例。如果没有定义 model，只定义了 fields 配置项，则使用 fields 配置项创建一个模型。

无论是定义了 model 配置还是定义 fields 配置项，都会调用 setProxy 方法创建代理实例并将 proxy 属性指向该实例。还要为代理绑定 metachange 事件并触发 metachange 事件。

定义了 id 配置项的 Store 对象，都会将 id 值转换为 storeId 属性的值，然后删除 id 成员。

如果 storeId 属性存在，则在 StoreManager 中注册该 Store 实例。

接着调用 initSortable 方法初始化 Sortable 对象。

最后与分组信息一样，创建一个 MixedCollection 对象实例用来记录过滤信息，通过 filters 属性可操作 Store 对象的过滤信息。

返回 Store 对象的构造函数，如果存在分组，则调用 sort 方法进行排序。

如果没有定义 buffered 属性和 pageSize 属性，则默认每页只取 25 条记录。

如果在定义 Store 时定义了数据，且使用的是 MemoryProxy，则调用 read 方法读取数据；否则，调用 add 方法添加数据。数据读取后，调用 sort 方法进行排序，并删除 inlineData 成员。

如果没有数据，且定义了 autoLoad 配置项，则延时调用 load 方法加载数据。

至此，Store 的实例化过程就完成了。

通过 Store 对象的创建过程，可清楚了解到，要访问 Store 中的数据，则使用 data 属性；要访问排序信息，则使用 sorters 属性（在 initSortable[⊖]方法中创建的）；要访问过滤信息，则使用 filters 属性；要访问分组信息，则使用 groupers 属性。以上 4 个属性，除了 sorters 属性是 AbstractMixedCollection 对象实例外，其余都是 MixedCollection 对象实例，原因是 Mixed-Collection 对象有排序属性，需要使用 Sorter 对象。如果 Sorter 对象又使用 Mixed-Collection 对象，那么就会出现问题，所以一定要先定义。

7.5.3 TreeStore 对象的实例化过程

TreeStore 对象的构造函数如下：

⊖ 相关信息可阅读 7.5.8 节。

```
constructor: function(config) {
    var me = this,
        root,
        fields;


    config = Ext.apply({}, config);
    fields = config.fields || me.fields;
    if (!fields) {
        config.fields = [{name: 'text', type: 'string'}];
    }
    me.callParent([config]);
    me.tree = Ext.create('Ext.data.Tree');
    me.relayEvents(me.tree, [
          //省略事件代码
    ]);
    me.tree.on({
        scope: me,
        remove: me.onNodeRemove,
        beforeexpand: me.onBeforeNodeExpand,
        beforecollapse: me.onBeforeNodeCollapse,
        append: me.onNodeAdded,
        insert: me.onNodeAdded,
        sort:me.onNodeSort
    });
    me.onBeforeSort();
    root = me.root;
    if (root) {
        delete me.root;
        me.setRootNode(root);
    }
    // 省略版本改变后属性名称改变的代码
},
```

如果没有定义模型或字段，那么就使用默认配置，只有一个 text 字段。然后调用 AbstractStore 对象的构造函数生成一个模型。**如果需要自定义字段，如保存着地址的 url 这些字段，必须自定义 field 配置项或使用模型，这样你就可通过模型实例访问到这些字段了。**

通过上一节 AbstractStore 对象的构造函数可以了解到，除了建立数据模型外，还会创建 sorters 属性和 filters 属性。

接着是**很关键一步，创建一个 TreeData 对象实例，用来记录节点，也就是说，通过 tree 属性，可以访问和操作树的所有节点。**

接着为 TreeData 对象绑定一些事件，主要是节点的处理事件。

因为树节点很多，不可能为每一个节点都绑定事件，因而需要使用事件传播机制，所以就会使用 relayEvents 方法来添加事件，以表明这些事件是可以传播的。而不需要进行传播的 rootchange 事件则使用 addEvents 方法添加。

接着调用 onBeforeSort 方法是为数据排序做准备。

接着要做的是设置根节点，将 root 属性指向根节点。通过 root 属性，也就是根节点，你可以按层次访问所有节点，具体请阅读 7.5.11 节。

通过 TreeStore 的创建过程，**要了解 tree 和 root 这两个属性，通过它们可访问到所有节点信息。**

7.5.4 Ext.data.Store 加载数据的方法

Store 对象有 load、loadData、loadRecords 和 loadPage 等 4 种加载数据的方法。

方法 loadData 的作用是加载本地数据，其代码如下：

```
loadData: function(data, append) {
    var model = this.model,
        length = data.length,
        i,
        record;
    for (i = 0; i < length; i++) {
        record = data[i];
        if (! (record instanceof Ext.data.Model)) {
            data[i] = Ext.ModelManager.create(record, model);
        }
    }

    this.loadRecords(data, {addRecords: append});
},
```

从 for 循环可以知道，参数 data 必须是数组。从判断语句可以看到，如果数据只是单纯的对象，需要将其转换为模型实例，然后调用 loadRecords 方法将其加载到 Store。

方法 loadRecords 的代码如下：

```
loadRecords: function(records, options) {
    var me     = this,
        i      = 0,
        length = records.length,
        start = (options = options || {}).start,
        snapshot = me.snapshot;

    if (!options.addRecords) {
        delete me.snapshot;
        me.clearData(true);
    } else if (snapshot) {
        snapshot.addAll(records);
    }

    me.data.addAll(records);

    if (typeof start != 'undefined') {
        for (; i < length; i++) {
            records[i].index = start + i;
            records[i].join(me);
        }
    } else {
        for (; i < length; i++) {
            records[i].join(me);
        }
    }
```

```
me.suspendEvents();

if (me.filterOnLoad && !me.remoteFilter) {
    me.filter();
}

if (me.sortOnLoad && !me.remoteSort) {
    me.sort();
}

    me.resumeEvents();
    me.fireEvent('datachanged', me, records);
    me.fireEvent('refresh', me);
},
```

代码首先判断配置项（options）的 addRecords 属性是否为 false，如果是，说明不是增加记录，而是重新加载记录，因而要删除快照和清理 data 中的数据；否则，在快照中调用 addAll 加载记录。从 loadData 方法调用 loadRecords 方法的代码可以看到，loadData 方法的第 2 个参数 append 控制着数据是添加还是重新加载。

因为 Store 对象的 data 属性指向的是 MixedCollection 对象，因而使用 addAll 方法就可以将数据加载到 data 了。

如果 start 的值不为 undefined，则为每个记录加一个索引值，再调用模型的 join 方法；否则，直接调用模型的 join 方法。方法 join 的作用是把当前 Store 实例加入到模型的 stores 数组中。

接着暂停所有事件。如果设置了在加载后执行过滤且不使用远程过滤，则执行过滤操作。如果设置了加载后执行排序且不是使用远程排序，则执行排序操作。

最后是恢复事件，并触发 datachanged 和 refresh 事件。

从代码可以看到，loadRecords 也是加载本地数据用的，不过其数据必须是模型实例组成的数组。

方法 loadPage 的作用是加载指定页的数据，其代码如下：

```
loadPage: function(page) {
    var me = this;

    me.currentPage = page;

    me.read(Ext.apply({
        page: page,
        start: (page - 1) * me.pageSize,
        limit: me.pageSize,
        addRecords: !me.clearOnPageLoad
    },options);
},
```

代码设置了 currentPage 属性为指定的页数，然后执行 read 方法。方法 read 是继承自 AbstractStore 对象的，其代码如下：

```
read: function() {
```

```
        return this.load.apply(this, arguments);
    },
```

直接调用了 load 方法，在这里不要把代码所处位置搞糊涂了，虽然 read 方法的定义在 AbstractStore 对象中，但是当前执行环境是在 Store 对象内，而 Store 对象已经重写了 AbstractStore 对象的 load 方法，因而这里执行的是 Store 对象内定义的 load 方法，其代码如下：

```
    load: function(options) {
        var me = this;
        options = options || {};
        if (Ext.isFunction(options)) {
            options = {
                callback: options
            };
        }
        Ext.applyIf(options, {
            groupers: me.groupers.items,
            page: me.currentPage,
            start: (me.currentPage - 1) * me.pageSize,
            limit: me.pageSize,
            addRecords: false
        });
        return me.callParent([options]);
    },
```

在这里只是封装了加载用的配置对象，包括分组信息、当前页、开始位置、每页记录数和 addRecords 等信息，要注意 addRecords 属性的值为 false，这样加载后会清除原有的数据。最后还是执行父类的 load 方法，其代码如下：

```
    load: function(options) {
        var me = this;

        options = options || {};

        if (typeof options == 'function') {
            options = {
                callback: options
            };
        }

        options.groupers = options.groupers ||  me.groupers.items;
        options.page = options.page || me.currentPage;
        options.start = options.start || (me.currentPage - 1) * me.pageSize;
        options.limit = options.limit || me.pageSize;
        options.addRecords = options.addRecords || false;

        return me.callParent([options]);
    },
```

如果参数 options 是函数，则将其作为回调函数，重新构建一个配置对象。
接着是初始化配置对象的分组、当前页、开始位置、每页记录数及数据添加方式等配置项。
最后调用父类的 load 方法，其代码如下：

```
load: function(options) {
    var me = this,
        operation;

    options = options || {};

    options.action = options.action || 'read';
    options.filters = options.filters || me.filters.items;
    options.sorters = options.sorters || me.getSorters();

    operation = new Ext.data.Operation(options);

    if (me.fireEvent('beforeload', me, operation) !== false) {
        me.loading = true;
        me.proxy.read(operation, me.onProxyLoad, me);
    }

    return me;
},
```

在这里还要初始化操作方式（默认是 read 操作）、过滤和排序配置项。

然后创建一个 Operation 实例。

如果 beforeload 方法确认加载数据，调用代理的 read 方法加载数据，实际是调用代理的 doRequest 方法加载数据。

注意调用 read 方法的第二参数，这个参数在 doRequest 方法中是作为回调函数使用的，因而在数据加载后会执行 onProxyLoad 方法，其代码如下：

```
onProxyLoad: function(operation) {
    var me = this,
        resultSet = operation.getResultSet(),
        records = operation.getRecords(),
        successful = operation.wasSuccessful();
    if (resultSet) {
        me.totalCount = resultSet.total;
    }
    if (successful) {
        me.loadRecords(records, operation);
    }
    me.loading = false;
    me.fireEvent('load', me, records, successful);
    me.fireEvent('read', me, records, operation.wasSuccessful());
    Ext.callback(operation.callback, operation.scope || me, [records, operation,
        successful]);
},
```

如果返回的 resultSet 存在，则先提取记录总数。如果操作成功，则使用 loadRecords 方法将数据添加到 data 中。最后是设置加载标志为 false，并触发 load 和 read 事件，最后执行自定义的回调函数。

很多初学者对 load 方法感到困惑的是不知道如何在 load 方法中添加自己的参数，如果明白整个加载过程，这就很会很清晰了。整个流程中，load 方法参数 options 只是在不断的添加

属性，最后成为 Operation 对象的成员，然后在 Proxy 对象中转换为 request 对象的成员，最后作为请求的配置对象，**因而，只要按照数据请求的配置对象设置配置项就可以了**。

在习惯使用 load 方法加载远程数据的同时，别忘记还可以使用 loadData 加载本地数据，通过 loadData 可以轻松地实现数据的联动，例如联动的 Combobox。

ArrayStore 对象虽派生于 Store 对象，但它只能加载本地数据，因而只能使用 loadData 方法加载数据。

NodeStore 对象只是 Ext JS 内部使用的对象，并不公开使用，其数据是内部加载并生成的，因而不适合使用 Store 对象的加载方法。

JsonStore、JsonPStore、XmlStore 和 PropertyStore 这 4 个对象则可以直接使用 Store 对象的加载数据方法。

7.5.5 Ext.data.TreeStore 加载数据的方法

TreeStore 对象加载数据的方式只有 load 方法，原因是本地加载数据要遵循添加节点的规则，因而本地数据的添加都是以节点操作形式实现的。

方法 load 的代码如下：

```
load: function(options) {
    options = options || {};
    options.params = options.params || {};

    var me = this,
    node = options.node || me.tree.getRootNode(),
    root;

if (!node) {
    node = me.setRootNode({
        expanded: true
    }, true);
}

if (me.clearOnLoad) {
    if(me.clearRemovedOnLoad) {
        me.clearRemoved(node);
    }
    me.tree.un('remove', me.onNodeRemove, me);
    node.removeAll(false);
    me.tree.on('remove', me.onNodeRemove, me);
}

Ext.applyIf(options, {
    node: node
});
options.params[me.nodeParam] = node ? node.getId() : 'root';

if (node) {
    node.set('loading', true);
}
```

```
        return me.callParent([options]);
    },
```

与 Store 对象最大的不同就是 TreeStore 加载数据完全围绕着节点展开。

代码首先检查参数 options 是否包含了 node 属性，如果不包含，则将根节点作为当前节点。

如果根节点不存在，使用 setRootNode 方法设置根节点。

如果 clearOnLoad（默认值为 true）为 true，则使用 removeAll 方法清除当前节点下的所有节点。如果 clearRemoveOnLoad 为 true，则调用 clearRemoved 方法清除已删节点。接着先移除树的 remove 事件，避免触发用户自定义的 remove 事件，然后调用 removeAll 方法清除节点，最后把 remove 事件绑定回树。

接着是很重要的一步，将节点加入到配置对象 options 中。

接着为配置对象添加一个由 nodeParam 属性指定的值作为提交参数，其值为节点 id，如果不存在 id，则为 root。**很多初学者都会为不知如何在服务器端获取树节点提交的参数而苦恼，关键就在这里**。属性 nodeParam 默认的值为 node，而在这里为配置对象的 params 添加了一个 node 的属性，意味着提交的参数就是 node，其值为当前节点的 id 值或 root，那么在服务器端，则可通过 node 获取到该值。

接着，如果节点存在，使用 set 方法设置节点的 Loading 字段值为 true，表示节点正在加载数据。

最后还是调用 AbstractStore 对象的 load 方法加载数据。加载后会执行 onProxyLoad 方法，其代码如下：

```
onProxyLoad: function(operation) {
    var me = this,
        successful = operation.wasSuccessful(),
        records = operation.getRecords(),
        node = operation.node;

    me.loading = false;
    node.set('loading', false);
    if (successful) {
        records = me.fillNode(node, records);
    }
    me.fireEvent('read', me, operation.node, records, successful);
    me.fireEvent('load', me, operation.node, records, successful);
    Ext.callback(operation.callback, operation.scope || me, [records, operation,
        successful]);
},
```

代码先设置节点的 loading 字段值为 false，表示加载结束了。如果加载成功，调用 fillNode 方法添加节点，其代码如下：

```
fillNode: function(node, records) {
    var me = this,
        ln = records ? records.length : 0,
        i = 0, sortCollection;
    Ext.Array.sort(records, me.sortByIndex);
    if (ln && me.sortOnLoad && !me.remoteSort && me.sorters && me.sorters.items) {
```

```
        sortCollection = Ext.create('Ext.util.MixedCollection');
        sortCollection.addAll(records);
        sortCollection.sort(me.sorters.items);
        records = sortCollection.items;
    }

    node.set('loaded', true);
    for (; i < ln; i++) {
        node.appendChild(records[i], undefined, true);
    }

    return records;
},
```

代码先调用 Ext.Array 的 sort 方法将返回的记录数组根据 sortByIndex 指定的值进行排序。

然后判断数据是否要执行本地排序，如果是，则通过 MixedCollection 对象将数据排序，再执行后续操作。

设置节点的 loaded 字段值为 true。然后使用节点的 appendChild 方法（NodeInterface.js）将数据追加为当前节点的子节点，其代码如下：

```
appendChild : function(node, suppressEvents, suppressNodeUpdate) {
    var me = this,
        i, ln,
        index,
        oldParent,
        ps;

    if (Ext.isArray(node)) {
        me.callStore('suspendAutoSync');
        for (i = 0, ln = node.length; i < ln; i++) {
            me.appendChild(node[i]);
        }
    } else {
        node = me.createNode(node);
        if (suppressEvents !== true && me.fireEvent("beforeappend", me, node) ===
             false) {
            return false;
        }
        index = me.childNodes.length;
        oldParent = node.parentNode;
        if (oldParent) {
            if (suppressEvents !== true && node.fireEvent("beforemove", node,
                 oldParent, me, index) === false) {
                return false;
            }
            oldParent.removeChild(node, null, false, true);
        }
        index = me.childNodes.length;
        if (index === 0) {
            me.setFirstChild(node);
        }
        me.childNodes.push(node);
        node.parentNode = me;
        node.nextSibling = null;
```

```
            me.setLastChild(node);
            ps = me.childNodes[index - 1];
            if (ps) {
                node.previousSibling = ps;
                ps.nextSibling = node;
                ps.updateInfo(suppressNodeUpdate);
            } else {
                node.previousSibling = null;
            }
            node.updateInfo(suppressNodeUpdate);
            if (!me.isLoaded()) {
                me.set('loaded', true);
            }
            else if (me.childNodes.length === 1) {
                me.set('loaded', me.isLoaded());
            }
            if (suppressEvents !== true) {
                me.fireEvent("append", me, node, index);

                if (oldParent) {
                    node.fireEvent("move", node, oldParent, me, index);
                }
            }
            return node;
        }
    },
```

如果参数 node 是数组，则递归调用 appendChild 方法。否则，直接使用 createNode 方法处理节点数据，其代码如下：

```
    createNode: function(node) {
        if (Ext.isObject(node) && !node.isModel) {
            node = Ext.ModelManager.create(node, this.modelName);
        }
        return Ext.data.NodeInterface.decorate(node);
    },
```

代码很清楚，如果节点数据不是模型实例，则将其转换为模型实例，以确保后续操作是以模型为基准进行的。最后使用 NodeInterface 对象 decorate[⊖]方法处理节点。

回到 appendChild 方法，如果没有禁止事件，且 beforeappend 事件返回 false，则中止追加节点，返回 false。

如果要添加的节点存在父节点，说明节点要实现移动操作，所以需要从其父节点移除当前节点。

如果添加的节点没有子节点，则需要使用 setFirstChild 方法将其 firstChild 属性指向添加节点。

接着将添加节点加入 childNodes 指向的子节点数组中。

后续要做的就是修改添加节点及其父节点的 lastChild、previousSibling、nextSibling 或 previousSibling 等属性。

⊖ 相关信息阅读 7.5.11 节。

然后就是触发事件了。

最后回到 onProxyLoad 方法，触发 read、load 等方法，结束加载过程。

与 Store 对象类似，load 方法的配置对象可参考 Ajax、JsonP 或 DirectProxy 对象的配置对象。

7.5.6 Store 的配置项

根据图 7-5 所示的结构以及各类 Store 对象的功能，可以知道主要的配置项在 AbstractStore、Store 和 TreeStore 这 3 个对象里。了解类的继承关系，对查阅 API 是相当有帮助，这个技巧要掌握。

1. AbstractStore 提供的 6 个基本的配置项

- autoLoad ：可以是布尔值，也可以是 load 方法的配置对象。如果该值为 true 或配置对象，则 Store 会在创建后自动加载数据。默认值为 false。
- autoSync ：布尔值，如果为 true，则 Store 的数据被修改后会立即同步到服务器。默认值为 false。
- fields ：如果你不想单独定义一个模型，那么可以使用该属性定义一个字段数组，让 Store 自动生成一个模型。不过现在不建议这样做，因为完整的模型功能更强大。
- listeners：可选参数，绑定监听事件。
- proxy：设置 Proxy 对象，可以为字符串，也可以为配置对象。
- storeId ：可选参数，设置当前 Store 的唯一标识。如果设置了该配置项，当前 Store 会注册到 StoreManager 对象中，从而实现在任何地方都可通过 id 在 StoreManager 对象内获取到该 Store，实现重用。默认值为 undefined。

2. Store 提供的 13 个配置项

- buffered ：布尔值，为 true 时，允许 Store 对象缓冲和预取数据。该属性主要用于不分页显示数据，在移动到下一页时预加载数据。
- clearOnPageLoad ：布尔值，默认值为 ture，每当加载新一页的数据，就会清空数据。如果为 false，则会保持现有数据，而且加载后，会显示所有数据。
- data：本地要加载的数据，可为一个由模型实例组成的数组或者数据对象。
- model ：字符串，关联的模型名称，如果没有定义该配置项，则会根据 fields 定义的字段创建一个模型。
- pageSize：每页的记录数。默认值为 25。
- currentPage：当前页，默认值为 1。
- purgePageCount ：在内存中保留多少页数据，默认值为 5。如果值为 0，则表示不清除预加载的数据。该值只有在 buffered 设置为 true 时才起作用。
- remoteSort ：布尔值，默认值为 false，表示在本地进行排序。如果设置为 true，则在服务器端进行排序，也就是对数据进行重新排序时，会发送一个请求给服务器端。
- remoteFilter ：布尔值，默认值为 false，表示在本地进行过滤。如果设置为 true，则在服务器端进行过滤，每当执行过滤时，会向服务器发送一个请求。

- remoteGroup：布尔值，默认值为 false，表示在本地进行分组。如果设置为 true，则在服务器端进行分组，每当执行分组时，会向服务器发送一个请求。
- sortOnFilter：布尔值，默认值为 true，表示在执行过滤后，还要执行排序操作，不过该配置项只有在执行本地过滤时才有效。
- groupField：要执行分组的字段，默认值为 undefined。
- groupDir：分组是的排序方向，值可为 ASC 或 DESC。默认值为 ASC，表示升序排序。

3. TreeStore 提供的 5 个配置项：

- clearOnLoad：布尔值，默认值为 true，表示在加载子节点前清除已经存在的子节点。
- nodeParam：加载节点时提交给服务器端的节点变量名称，它的值是节点的唯一标识。默认值是 node。
- defaultRootId：默认的根节点的值，默认值为 root。
- defaultRootProperty：Reader 对象从返回数据读取数据的属性名称。默认值是 children。
- folderSort：布尔值，如果设置为 true，会自动对没有子节点的节点进行排序。默认值为 false。

一般情况下，TreeStore 不需要定义模型，节点就是它的数据模型，不过如果有特殊的数据要记录在节点上，也可以为它定义一个数据模型。

很多时候，需要绑定一些特殊的数据在节点上做特殊处理的，例如绑定一个地址，单击节点的时候让面板访问指定地址的内容。很多初学者不清楚的是如何去访问取这些数据，其实了解数据结构，这个问题会相当容易，在 Ext JS 4，这个变得更简单了，在 TreePanel 的 itemclick 事件中，第 2 个参数会返回当前节点的模型实例，只要使用数据模型的属性和方法就能轻松的访问到需要的数据了。还不行？ console.log 一下这个模型实例，然后在 DOM 面板中研究一下结构就清楚了。

7.5.7 Store 的分页

因为数据的加载是通过 Store 进行的，因而其分页操作也是在 Store 中进行的，而不是在分页工具栏内。分页工具栏只是提供分页操作的 UI，与具体的分页操作是无关的，这关系一定要清楚。

要在 Store 中进行分页，定义配置项 pageSize 就行了，其默认值是 25，也就是说，Store 在默认状态下是分页的，每页有 25 条记录。如果你的 Store 不需要分页，也没关系，不用理会提交的参数，直接将所有数据返回就可以了，Store 会把返回的数据都保存起来，不会理会返回的数据长度是大于还是小于 pageSize 的。

分页的重点是 Store 会提交哪些参数让服务器端知道现在要取的是哪一页数据，其主要的提交参数有以下三个：

- limit：每页的记录数，也就是 pageSize 的值。如果每页记录是动态的，也就是在程序运行中，运行用户动态修改 pageSize 的值，那么服务器端就可以根据该值决定每页需要返回多少记录。

- page：要获取的页数。
- start：获取页的记录的开始位置。这里要注意，记录位置是从 0 开始算的，因而当每页记录数为 25 的时候，开始位置是 25，而不是 26。

服务器端通过提取以上三个参数就可以知道当前需要返回哪一页的数据了。

7.5.8 Store 的排序：Ext.util.Sorter 与 Ext.util.Sortable

很多初学者以为单击 Grid 的标题进行排序，是在 Grid 中进行的，会有这个错误想法的原因是不了解 Ext JS 的 UI 组件是和数据分离的，UI 组件只负责显示，而数据的操作是在 Store 中进行的，数据操作完成后会自动更新 UI 组件，UI 组件做的只是将用户操作触发的事件转化成 Store 的操作。所以要对数据排序也是在 Store 中进行的。排序完成后，Store 会去触发事件告诉 UI 组件数据已经改变，需要刷新组件。

Sorter 对象的作用是为每一个查询建立一个 Sorter 实例，在实例中包含排序字段、排序方向、排序方法和数据转换方法等信息。Sorter 对象有以下 5 个配置项：

- property ： 要排序的字段名称，如果没有定义 sorterFn，该参数必须定义，否则会产生错误。
- direction ： 排序方向，值可以为 ASC 和 DESC。ASC 表示升序排列，DESC 表示降序排列。
- root：字符串，设置获取排序数据的位置。
- sorterFn：排序时执行的排序函数，定义了该函数就可以不定义 property 配置项。
- transform ： 数据转换函数，排序的数据可使用该函数转换后再执行排序。该配置项会将字段定义的 sortType 配置项定义的函数作为转换函数，因而只要定义好字段的 sortType 配置项就行了。

Sorter 对象中默认的排序函数的代码如下：

```
defaultSorterFn: function(o1, o2) {
    var me = this,
        transform = me.transform,
        v1 = me.getRoot(o1)[me.property],
        v2 = me.getRoot(o2)[me.property];

    if (transform) {
        v1 = transform(v1);
        v2 = transform(v2);
    }

    return v1 > v2 ? 1 : (v1 < v2 ? -1 : 0);
},
```

看出有什么问题没有？该排序函数进行一般的排序没有问题，但是处理中文排序是错误的。打开 7.4.2 节示例，单击姓名进行排序，会看到中文的排序结果是“张三”、“李四”和“王五”，对中文来说这次序是错误的。从 Ext JS 2 以来一直存在这问题，但是一直没有改变，不知道什么原因。不过问题不大，自己修改一下，当数据类型是字符串的时候，使用

localeCompare 方法进行排序就行了。现在的问题是怎么修改，定义数据类型也是一种可行办法，但是最大问题是每次定义字符串字段的时候，都必须为其绑定数据类型或排序类型，相当麻烦。所以，最好的方法还是直接修改这个默认函数，将其放到本地化文件里就可一劳永逸了，只要在本地化文件（ext-lang-zh_CN.js）尾部添加以下代码就行了：

```
if(Ext.util.Sorter){
    Ext.apply(Ext.util.Sorter.prototype, {
            defaultSorterFn: function(o1, o2) {
                var me = this,
                transform = me.transform,
                v1 = me.getRoot(o1)[me.property],
                v2 = me.getRoot(o2)[me.property];

                if (transform) {
                    v1 = transform(v1);
                    v2 = transform(v2);
                }
            if(Ext.isString(v1)){
                    return v1.localeCompare(v2);
                    }else{
                    return v1 > v2 ? 1 : (v1 < v2 ? -1 : 0);
                }
            }
    })
}
```

代码先判断 Sorter 对象是否存在，如果存在，修改其原型的 defaultSorterFn 方法。主要修改的地方在粗体代码，如果 v1 的值是字符串，使用 localeCompare 方法比较数据。刷新一下示例，再对姓名排序，已可正确排序中文了。**在本地化文件做一些 bug 修正，或者提示信息的汉化是常用的手法，需要掌握。**

一个 Store 会有很多字段，而很多字段是可以排序的，那么如何管理这些排序字段的 Sorter 对象实例呢？ Sortable 对象的用途就是用来管理 Sorter 对象实例的。在 Store 进行初始化时（AbstractStore 对象的构造函数），都会执行 Sortable 对象的 initSortable 方法，其代码如下：

```
initSortable: function() {
    var me = this,
        sorters = me.sorters;
    me.sorters = Ext.create('Ext.util.AbstractMixedCollection', false,
        function(item) {
        return item.id || item.property;
    });
        if (sorters) {
            me.sorters.addAll(me.decodeSorters(sorters));
        }
},
```

方法会创建一个 AbstractMixedCollection 对象，属性 sorters 会指向该对象，然后会将 Store 配置对象中 sorters 配置项经过 decodeSorters 方法处理后添加到 AbstractMixed-Collection 对象中。方法 decodeSorters 的代码如下：

```
decodeSorters: function(sorters) {
    if (!Ext.isArray(sorters)) {
        if (sorters === undefined) {
            sorters = [];
        } else {
            sorters = [sorters];
        }
    }
    var length = sorters.length,
        Sorter = Ext.util.Sorter,
        fields = this.model ? this.model.prototype.fields : null,
        field,
        config, i;
for (i = 0; i < length; i++) {
    config = sorters[i];

    if (!(config instanceof Sorter)) {
        if (Ext.isString(config)) {
            config = {
                property: config
            };
        }

        Ext.applyIf(config, {
            root     : this.sortRoot,
            direction: "ASC"
        });
        if (config.fn) {
            config.sorterFn = config.fn;
        }
        if (typeof config == 'function') {
            config = {
                sorterFn: config
            };
        }
        if (fields && !config.transform) {
            field = fields.get(config.property);
            config.transform = field ? field.sortType : undefined;
        }
        sorters[i] = Ext.create('Ext.util.Sorter', config);
    }
}

    return sorters;
},
```

如果参数 sorters 不为数组，也就是定义的配置项不是数组，则先将 sorters 转换为数组。

注意变量 fields，它会获取模型中的字段对象。

接着在循环中取出每个 sorter 的定义，如果不是 Sorter 对象的实例，则需要将其转换为 Sorter 对象的实例。否则，不需要进行处理。

转换为 Sorter 对象的实例过程是，先检查是否为字符串（字段名称），如果是，将其转换为对象。然后在对象中添加 root 和 direction 属性。如果定义了 fn 配置项（兼容 Ext JS 3.x），

将其转换为 Ext JS 4 的 sorterFn 配置项的值（排序函数）。如果配置项本身是一个函数，则直接将其转为对象，并将 sorterFn 属性指向函数。

接着要做的是，检查是否存在字段和转换函数，如果有，从字段定义中取出转换函数并将 transform 属性指向转换函数。

最后是创建 Sorter 实例。

从以上代码可以知道，在 Store 的定义中可以加入配置项 sorters 来定义排序对象，而 sorters 配置项的值可以是字符串、函数、对象、数组或 Sorter 对象实例。值为字符串时，必须为字段名称。为函数时是排序函数。为对象时，对象的配置项必须是 Sorter 对象的配置项。为数组时，数组项可以是字符串、函数、对象或 Sorter 对象实例。

那么，是否需要排序的字段都要进行定义，这样不是很麻烦？答案当然是不需要。在每次排序的时候，都会从 sorters 查询是否存在查询字段的 Sorter 对象实例，如果不存在，则会新建一个。具体代码可阅读 Sortable 对象的 sort 方法。

7.5.9 Store 的过滤：Ext.util.Filter

与排序一样，过滤也有自己的对象 Filter。Filter 对象的配置项有以下 5 个配置项：

- property：要过滤的字段。
- filterFn ：过滤函数。MixedCollection 对象会枚举所有数据给过滤函数，如果过滤函数返回 true，表示该数据符合要求，否则返回 false，数据将被过滤掉。
- anyMatch：布尔值，是否执行任意匹配过滤。默认值为 false，表示不执行任意匹配过滤。
- exactMatch ：布尔值，是否执行完全匹配过滤。默认值为 false，表示不执行完全匹配过滤。
- caseSensitive：布尔值，执行过滤时，是否区分大小写。默认值为 false，不区分大小写。

Filter 对象默认是使用正则表达式进行过滤的，而其默认的过滤方式是匹配开始位置，也就是在查询值前加入“^”符号。配置项 exactMatch 的作用是在匹配开始位置的前提下，再加入“$”符号，以达到完全匹配的效果。而配置项 anyMatch 则不会添加“^”和“$”符号，实现匹配任何位置的过滤。

与排序不同的是不需要特殊对象来管理 Filter 对象实例，而是直接使用 MixedCollection 对象。

与排序一样的是可以在 Store 定义中加入 filters 配置项来添加过滤设置，其值也可以是字符串、对象、函数、Filter 对象实例以及由以上类型组成的数组。

7.5.10 Store 的分组：Ext.util.Grouper

不用说，分组也会有自己的对象，这就是 Grouper 对象。Grouper 对象是从 Sorter 继承过来的，因而它具有 Sorter 对象的配置项。要注意，分组是 Store 对象特有的特性，AbstractStore 和 TreeStore 对象是不具备的。

管理 Grouper 对象实例没有特殊对象，也是使用 MixedCollection 对象管理的。

与排序一样，可以在 Store 定义中加入 groupers 配置项来添加分组设置，其值也可以是字符串、对象、函数、Grouper 对象实例以及由以上类型组成的数组。

7.5.11 树节点：Ext.data.NodeInterface 与 Ext.data.Tree

NodeInterface 对象的作用是为模型数据添加节点特性以及为节点提供操作接口。在 7.5.3 小节，已经知道 TreeStore 加载数据后，都会使用 NodeInterface 对象的 decorate 方法处理节点，其代码如下：

```
decorate: function(modelClass) {
    var idName, idType;
    if (typeof modelClass == 'string') {
        modelClass = Ext.ModelManager.getModel(modelClass);
    }
    if (modelClass.prototype.isNode) {
        return;
    }
    idName = modelClass.prototype.idProperty;
    idType = modelClass.prototype.fields.get(idName).type.type;
    modelClass.override(this.getPrototypeBody());
    this.applyFields(modelClass, [
        {name: 'parentId',   type: idType,    defaultValue: null},
        {name: 'index',      type: 'int',     defaultValue: null, persist: false},
        {name: 'depth',      type: 'int',     defaultValue: 0, persist: false},
        {name: 'expanded',   type: 'bool',    defaultValue: false, persist: false},
        {name: 'expandable', type: 'bool',    defaultValue: true, persist: false},
        {name: 'checked',    type: 'auto',    defaultValue: null, persist: false},
        {name: 'leaf',       type: 'bool',    defaultValue: false},
        {name: 'cls',        type: 'string',  defaultValue: null, persist: false},
        {name: 'iconCls',    type: 'string',  defaultValue: null, persist: false},
        {name: 'icon',       type: 'string',  defaultValue: null, persist: false},
        {name: 'root',       type: 'boolean', defaultValue: false, persist: false},
        {name: 'isLast',     type: 'boolean', defaultValue: false, persist: false},
        {name: 'isFirst',    type: 'boolean', defaultValue: false, persist: false},
        {name: 'allowDrop',  type: 'boolean', defaultValue: true, persist: false},
        {name: 'allowDrag',  type: 'boolean', defaultValue: true, persist: false},
        {name: 'loaded',     type: 'boolean', defaultValue: false, persist: false},
        {name: 'loading',    type: 'boolean', defaultValue: false, persist: false},
        {name: 'href',       type: 'string',  defaultValue: null, persist: false},
        {name: 'hrefTarget', type: 'string',  defaultValue: null, persist: false},
        {name: 'qtip',       type: 'string',  defaultValue: null, persist: false},
        {name: 'qtitle',     type: 'string',  defaultValue: null, persist: false},
        {name: 'children',   type: 'auto',    defaultValue: null, persist: false}
    ]);
},
```

如果参数 modelClass 是字符串，则从模型管理器中取出模型。

如果模型原型的 isNode 为 true，表示该模型已经是节点，直接返回。

如果模型不是节点，首先取得 id 字段及其类型。然后调用 override 方法重写模型原型，为模型添加以下几个节点操作方法：

- appendChild：追加节点。如果节点是其他节点的子节点，会先移除节点。

- bubble：从当前节点往上开始执行冒泡过程，过程中的每个节点都会执行一次指定的函数，只有在函数返回 false 或不再有节点时才中止过程。
- cascadeBy ：从当前节点往下执行瀑布过程，过程中的每个节点都会执行一次指定的函数，只有在函数返回 false 或不再有节点时才中止过程。
- collapse：折叠节点。
- collapseChildren：折叠当前节点的子节点。
- contains：如果当前节点是指定节点的祖先，返回 true。
- copy：复制当前节点。
- createNode：确保指定的节点是模型实例。
- destroy：销毁节点。
- eachChild：枚举子节点。
- expand：展开节点。
- expandChildren：展开当前节点的子节点。
- findChild：查找指定属性且其值为指定值的第一个子节点。
- findChildBy：通过指定函数查找符合要求（返回 true）的第一个子节点。
- getChildAt：返回指定位置的子节点。
- getDepth：返回当前节点的深度。
- hasChildNodes：如果当前节点有子节点，返回 true。
- indexOf：返回子节点的位置。
- insertBefore：在指定的子节点前插入一个节点。
- insertChild：在指定位置插入一个子节点。
- isAncestor：如果指定节点是当前节点的祖先，返回 true。
- isExpandable：如果当前节点是可展开的，返回 true。
- isExpanded：如果当前节点的状态是展开状态，返回 true。
- isFirst：如果当前节点是其父节点的第一个子节点，返回 true。
- isLast：如果当前节点是其父节点的最后一个子节点，返回 true。
- isLeaf：如果当前节点是叶节点，返回 true。
- isLoaded：如果当前节点加载节点过程已完成，返回 true。
- isLoding：如果当前节点正在加载节点，返回 true。
- isRoot：如果当前节点是根节点，返回 true。
- isVisible：如果当前节点是可见的，返回 true。
- remove：移除当前节点。
- removeAll：移除当前节点的所有子节点。
- removeChild：移除指定的子节点。
- replaceChild：替换节点。
- sort：使用指定函数对子节点排序。
- updateInfo：更新节点的基本信息，如 isFirst、isLast、depth 等。

最后如调用 applyFields 方法为模型添加额外的字段。

通过重写模型原型，把一个简单的模型武装成了一个强大的节点，这样每个节点都可以独立运作了，这样的好处就是只要取出节点，就能做插入节点、删除节点等操作。

节点有了，现在的问题是怎么在 TreeStore 中存放这些节点，这就要使用 TreeData 对象。TreeData 通过两种方式存放数据，第 1 种是使用 nodeHash 对象，对象中，以节点的 id 作为属性名称、以节点为值，这样通过 id 就可以获取到节点。第 2 种是将 root 属性指向根节点，然后通过根节点以层次结构访问子节点，子节点都存放在 childNodes 属性指向的数组中。这样的好处就是通过 id 可在 nodeHash 对象方便地找到节点并执行操作，显示的时候就可根据节点层次结构进行显示。能实现这样的效果得益于 JavaScript 的语言的特点，对象都是独立的，变量存储的只是对象的指针，这样，一个对象就可以根据变量在任何位置出现，但是其指向的还是同一个对象。

TreeData 对象的主要有以下几个方法：

- ❑ getRootNode：返回根节点。
- ❑ setRootNode：设置根节点。
- ❑ flatten：将所有节点转换为一个数组。
- ❑ getNodeById：通过 id 获取节点。
- ❑ registerNode：在 nodeHash 对象中注册一个节点。
- ❑ unregisterNode：在 nodeHash 对象中取消节点的注册。
- ❑ sort：对树进行排序。
- ❑ filter：对树进行过滤。

7.5.12 Store 的方法

与配置项一样，Store 的方法也是分散在 AbstractStore、Store 和 TreeStore 这 3 个对象中。

1. AbstractStore 对象提供的方法

- ❑ setProxy：设置代理类型。
- ❑ getProxy：返回代理对象。
- ❑ getNewRecords：返回新建的模型实例。
- ❑ getRemovedRecords：返回已经被删除但还没提交的记录。
- ❑ getUpdatedRecords：返回在 Store 已被更新但还没提交的记录。
- ❑ isLoading：返回 true 表示 Store 正在加载数据。
- ❑ load：加载数据。
- ❑ removeAll：删除 Store 中的所有数据。
- ❑ sort：执行排序操作。
- ❑ sync：将 Store 的数据与服务器进行同步，该方法会将新增、更新或删除的数据提交到服务器。

2. Store 对象提供的方法

- add：将 JavaScript 对象转换为模型实例添加到 Store。如果要添加模型实例，使用 insert 方法。
- aggregate：使用指定的函数对 Store 的模型实例进行合计计算。
- average：对 Store 中的数据，计算指定字段的平均值。
- clearFilter：清理过滤操作，显示原始数据。
- clearGrouping：清理分组操作，显示原始数据。
- collect：收集指定字段的唯一值。
- count：返回模型实例总数。
- each：枚举模型实例。
- filter：执行过滤操作。
- filterBy：使用指定函数执行过滤。
- find：根据指定的字段和值，查询符合条件的第一个模型实例，并返回其索引值。
- findBy：使用指定函数搜索记录，如果函数返回 true，返回该模型实例的索引。
- findExact：查找指定字段的值与指定的值完全匹配的第一个模型实例的索引。
- findRecord：与 find 方法功能一样，只是返回值为模型实例。
- first：返回 Store 中第一个模型实例，如果不存在，返回 undefined。
- getAt：返回指定位置的模型实例。
- getById：通过 id 获取模型实例。
- getCount：返回模型实例的总数。
- getGroupString：返回分组的字段名称。
- getGroups：返回分组后的数据。返回的数据格式为一个数组，每一个分组数组就是一个对象，对象中的 children 属性指向分组后的模型实例组成的数组。
- getPageFromRecordIndex：通过模型实例的索引确定其所在页，并返回该页数。
- getRange：返回指定范围内的模型实例。
- getTotalCount：返回数据库中的数据总数。
- group：进行分组操作。
- guaranteeRange：加载指定范围内的数据，范围必须小于或等于每页记录数。
- hasPendingRequests：返去预请求的数量。
- indexOf：返回指定模型实例的索引。
- indexOfId：根据指定 id 获取模型实例并返回其索引。
- indexOfTotal：返回指定模型实例在整个实体数据集中的索引。
- insert：将模型实例插入到指定位置。
- isFiltered：如果 Store 处于过滤状态，返回 true。
- isGrouped：如果 Store 处于分组状态，返回 true。
- last：返回 Store 中最后一个模型实例。

- max：返回指定字段的最大值。
- min：返回指定字段的最小值。
- nextPage：加载下一页数据。
- prefetch：预取数据。
- prefetchPage：预取指定页的数据。
- previousPage：加载上一页数据。
- purgeRecords：如果 prefetchCount 被改变，清除最近的很少使用的预取记录。
- queryBy：使用指定的过滤函数过滤数据。
- sum：合计指定字段的值。

3. TreeStore 对象提供的方法

- getRootNode：返回根节点。
- setRootNode：设置根节点。

因为节点的操作都在节点里，所以 TreeStore 本身不需要太多操作。

注意 *在 Ext JS 3 中的 commitChanges 和 rejectChanges 方法不存在了，所以不能在 Store 里确认或取消修改。代替它们的方法是模型的 commit 和 reject 方法。*

7.5.13 Store 的事件

Store 的事件也是分散在 AbstractStore、Store 和 TreeStore 这 3 个对象中。

1. AbstractStore 对象中的事件

- add：当模型实例添加到 Store 时会触发该事件。
- beforeLoad：加载数据前会触发该事件，如果返回 false，会取消加载。
- beforesync：在同步前会触发该事件，如果返回 false，会取消同步。
- clear：执行 removeAll 方法后会触发该方法。
- datachanged：模型实例被改变（添加、删除和更新）后会触发该事件。
- load：从远程服务器加载数据后会触发该事件。
- remove：模型实例被删除时会触发该事件。
- update：模型实例更新后会触发该事件。

2. Store 对象中的事件

- beforeprefetch：在预取数据前会触发该事件，如果返回 false，会取消预取数据。
- groupchange：分组改变后会触发该事件。

3. TreeStore 中的事件

- append：当新的子节点追加到一个节点后触发该事件。
- beforeappend：在追加子节点前会触发该事件，如果返回 false，会取消追加操作。

- beforecollapse：在节点折叠前会触发该事件，如果返回 false，会取消折叠操作。
- beforeexpand：在节点展开前会触发该事件，如果返回 false，会取消展开操作。
- beforeinsert：在树中插入一个节点时触发该事件，如果返回 false，会取消插入操作。
- beforemove：在移动节点前，会触发该事件，如果返回 false，会取消移动操作。
- beforeremove：在删除节点前，会触发该事件，如果返回 false，会取消删除操作。
- collapse：当节点折叠后，会触发该事件。
- expand：当节点展开后，会触发该事件。
- insert：当新节点插入完成后，会触发该事件。
- move：当节点移动后，会触发该事件。
- remove：当节点被删除后，会触发该事件。
- rootchange：当根节点被改变后，会触发该事件。
- sort：执行排序过程后，会触发该事件。
- update：当节点被更新后，会触发该事件。

7.5.14 Store 管理器：Ext.data.StoreManager

在这里明确一点，只有设置了 storeId 配置项的 Store 才会在管理器中注册。StoreManager 对象是从 MixedCollection 对象继承的，因而具有 MixedCollection 对象的操作方法。

StoreManager 对象提供了 register 和 unregister 对象来注册和撤销注册 Store。还可以使用 Ext.regStore 方法创建一个 Store 并注册它。

要从 StoreManager 对象中获取一个 Store，可使用 Ext.getStore 方法，它会调用 StoreManager 对象的 lookup 方法来获取 Store。

7.6 综合实例

7.6.1 远程读取 JSON 数据

（1）功能描述

本示例将演示如何使用 AjaxProxy 读取远程数据以及如何操作这些数据。

（2）实现代码

在实现代码前首先要做的是确定数据源。本示例将使用微软示例数据库“Northwnd”的产品表（Products）作为数据源。

使用模板页创建一个文件名为 7-5.html 的页面文件。

首先要做的是根据 Products 表结果定义一个模型：

```
Ext.define('Products', {
    extend: 'Ext.data.Model',
    fields: [
      {name:'id',type:"int"},
```

```
        'ProductName',
        {name:'SupplierID',type:"int"},
        {name:'CategoryID',type:"int"},
        'QuantityPerUnit',
        {name:'UnitPrice',type:"float"},
        {name:'UnitsInStock',type:"int"},
        {name:'UnitsOnOrder',type:"int"},
        {name:'ReorderLevel',type:"int"},
        {name:'Discontinued',type:"bool"}
    ]
});
```

除了 ProductID 字段没有使用数据库中的字段名称外，其余字段与数据库的名称是一样的，这个可以根据需要做修改，没太大关系。因为将 ProductID 字段作为了 id 字段，所以不需要修改模型的 idProperty 属性。这里也没有定义 Proxy 对象，因为会在 Store 中定义。

接着在 OnReady 函数内定义一个 Store：

```
var store=Ext.create("Ext.data.Store",{
      model:'Products',
      autoLoad:true,
      pageSize:20,
    proxy: {
        type: 'ajax',
        url : 'Products.ashx' ,
        //url:"/Restful/Products",    //java
        reader:{
                type: 'json',
                root:"data"
            }
    },
    storeId:"ProductStore"
})
```

最简单的定义只有模型、自动加载、每页记录数和 proxy 等配置项。Reader 的 root 设置为 data，所以服务器返回的数据必须在 data 属性内。

其实不需要 UI 组件也不能操作 Store 的数据，不过为了直观点，还是定义一个 Grid 比较合适：

```
var panel=Ext.create("Ext.grid.Panel",{
   renderTo:Ext.getBody(),
   title:"产品列表",
   height:600,
   width:800,
store:store,
columns:[
        {text:'id',dataIndex:'id'},
        {text:'ProductName',dataIndex:'ProductName'},
        {text:'SupplierID',dataIndex:'SupplierID'},
        {text:'CategoryID',dataIndex:'CategoryID'},
        {text:'QuantityPerUnit',dataIndex:'QuantityPerUnit'},
        {text:'UnitPrice',dataIndex:'UnitPrice'},
        {text:'UnitsInStock',dataIndex:'UnitsInStock'},
```

```
            {text:'UnitsOnOrder',dataIndex:'UnitsOnOrder'},
            {text:'ReorderLevel',dataIndex:'ReorderLevel'},
            {text:'Discontinued',dataIndex:'Discontinued'}

    ]
});
```

没什么特殊配置，只是简单地将数据显示出来。

根据 Store 的定义，服务器返回的数据格式应该是：

```
{
  total: 0,// 记录总数
  success: true, // 必需的，说明返回数据成功
  data: [
        ...
   ]
}
```

在服务器端从数据库读取数据后，根据该格式组织数据就行了。

C# 代码（products.ashx）：

```
public void ProcessRequest (HttpContext context) {
    context.Response.ContentType = "text/javascript";
    int start = 0;
    int limit = 20;
    int.TryParse(context.Request.Params["start"], out start);
    string sort = "it.ProductID";
    if (context.Request.Params["sort"] != null)
    {
        sort="";
      JArray ja = JArray.Parse(context.Request.Params["sort"]);
      foreach (JObject jo in ja)
      {
          sort += "it."+(string)jo["property"] + " " + (string)jo["direction"]  + ",";
      }
      sort = sort.Substring(0, sort.Length - 1);
    }
    using(NorthwindEntities ne = new NorthwindEntities())
    {
      int total = ne.Products.Count();
      if (total > start) start = 0;
      var q = ne.Products.OrderBy(sort).Skip(start).Take(limit).Select(m =>
          m).ToList();
      var output = new JObject
      {
      new JProperty("total",total),
      new JProperty("success",true),
      new JProperty("data",new JArray(
          from p in q
          select new JObject(
              new JProperty("id",p.ProductID),
              new JProperty("ProductName",p.ProductName),
              new JProperty("QuantityPerUnit",p.QuantityPerUnit),
              new JProperty("ReorderLevel",p.ReorderLevel),
```

```
                    new JProperty("SupplierID",p.SupplierID),
                    new JProperty("UnitPrice",p.UnitPrice),
                    new JProperty("UnitsInStock",p.UnitsInStock),
                    new JProperty("UnitsOnOrder",p.UnitsOnOrder),
                    new JProperty("CategoryID",p.CategoryID),
                    new JProperty("Discontinued",p.Discontinued)
                )
            ))
        };
        context.Response.Write(output.ToString());
    }
    ;
}
```

Java 代码（Products.java）：

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
   response.setContentType("text/javascript; charset=utf-8");
    String connectionUrl = "jdbc:sqlserver://192.168.0.254:1433;" +
    "databaseName=Northwind;;user=sa;password=abcd-1234";
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;
    ResultSet rscount=null;
    int start= 0;
    if(request.getParameter("start") != null){
          start=Integer.parseInt(request.getParameter("start")) ;
    }
    int limit=20;
    String sort="ProductID";
    if(request.getParameter("sort")!=null){
          sort="";
          JsonParser jparser = new JsonParser();
       JsonArray ja = jparser.parse(request.getParameter("sort")).
           getAsJsonArray();
       for (JsonElement je : ja) {
          JsonObject jo = je.getAsJsonObject();
          sort+=jo.get("property").getAsString()+ " "
                        + jo.get("direction").getAsString()+",";
       }
       sort=sort.substring(0,sort.length()-1);
    }
    try {

       Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
       con = DriverManager.getConnection(connectionUrl);

       int count = 0;
       stmt = con.createStatement();
       rscount=stmt.executeQuery("select count(ProductID) as count from
          products");
       rscount.next();
       count=rscount.getInt(1);
       if(start>count){
```

```
        start=0;
}
StringBuilder sb = new StringBuilder();
sb.append("select top 20 * from products where ProductID ");
sb.append(" not in (select top ");
sb.append(start);
sb.append(" ProductID from products order by ");
sb.append(sort);
sb.append(" ) order by ");
sb.append(sort);

rs = stmt.executeQuery(sb.toString());

JsonArray array=new JsonArray();
while (rs.next()) {
        JsonObject obj= new JsonObject();
        obj.addProperty("id", rs.getString("ProductID"));
        obj.addProperty("ProductName", rs.getString("ProductName"));
        obj.addProperty("SupplierID", rs.getString("SupplierID"));
        obj.addProperty("CategoryID", rs.getString("CategoryID"));
        obj.addProperty("QuantityPerUnit", rs.getString("QuantityPerUnit"));
        obj.addProperty("UnitPrice", rs.getString("UnitPrice"));
        obj.addProperty("UnitsInStock", rs.getString("UnitsInStock"));
        obj.addProperty("UnitsOnOrder", rs.getString("UnitsOnOrder"));
        obj.addProperty("ReorderLevel", rs.getString("ReorderLevel"));
        obj.addProperty("Discontinued", rs.getString("Discontinued"));
        array.add(obj);
    }
    JsonObject json=new JsonObject();
    json.addProperty("totals", count);
    json.addProperty("success",true);
    json.add("data", array);
    response.getWriter().write(json.toString());
    rs.close();

}
catch (Exception e) {
        response.getWriter().write(e.getMessage());
}
finally {
    if (rscount != null) try { rscount.close(); } catch(Exception e) {}
    if (rs != null) try { rs.close(); } catch(Exception e) {}
    if (stmt != null) try { stmt.close(); } catch(Exception e) {}
    if (con != null) try { con.close(); } catch(Exception e) {}
}
```

服务器端代码的基本流程是，先从提交的参数中取出要取数据的开始位置，变量 limit 也可以从提交的参数中获取，如果是固定的，就没必要了。如果客户端未提交排序信息，默认是使用 ProductID 字段排序。如果提交参数 sort 有数据，因为其提交的是 JOSN 数组，所以要使用 Parse 方法将其转换为 JSON 数组，然后从数组中读出 JSON 对象，再从 property 属性中读出字段名称，最后从 direction 属性中读出排序方向。因为可能会有多个字段排序，

所以要用循环遍历 JSON 数组，逐个取出字段和排序方向，然后组合成排序字符串。如果你对字符串处理比较有信心，也可以用字符串拆解方式处理这些数据。远程过滤和分组也是以这样的方法提交的，可以使用相同的方式进行处理。

接着要做的是获取表的记录总数，如果 start 值大于记录总数，则设置为 0，取第 1 页的数据，这个可根据你自己的情况做处理。

接下来要做的就是从数据库中取出分页数据，然后使用 JSON 对象构建输出了。C# 使用 LINQ to JSON 直接按照数据格式构建数据就行了。Java 的步骤也不难，先构建好 JSON 数组格式的数据，这里要注意，因为客户端定义的模型，ProductID 的字段名称是 id，id 构建数组数据时要使用 id 作为 ProductID 字段值的属性。然后再创建一个 JSON 对象，添加 total、success 和 data 等属性就可以了。这里要注意的地方是，因为客户端读取数据的位置是 data，所以这里数据的属性必须是 data。

特别要注意的一点是，如果客户端模型的字段名称与服务器端的名称不一样，就需要做一个映射表，将客户端提交过来的字段名转换为表的字段名，不然在查询时会提示找不到字段。因为这里使用的是相同的字段名，所以不需要转换。不过，如果对 ProductID 进行排序，还是要转换的，因为客户端提交的是 id。问题不大，这只是一个简单示例，在开发应用的时候注意就行。

（3）页面效果

完成以上代码后，在浏览器中打开页面，将看到如图 7-6 所示界面。

id	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder
1	Chai	1	1	10 boxes x 20 bags	18	39	0
2	Chang	1	1	24 - 12 oz bottles	19	17	40
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10	13	70
4	Chef Anton's Cajun	2	2	48 - 6 oz jars	22	53	0
5	Chef Anton's Gumb	2	2	36 boxes	21.35	0	0
6	Grandma's Boysenb	3	2	12 - 8 oz jars	25	120	0
7	Uncle Bob's Organi	3	7	12 - 1 lb pkgs.	30	15	0
8	Northwoods Cranbe	3	2	12 - 12 oz jars	40	6	0
9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	97	29	0
10	Ikura	4	8	12 - 200 ml jars	31	31	0
11	Queso Cabrales	5	4	1 kg pkg.	21	22	30
12	Queso Manchego L	5	4	10 - 500 g pkgs.	38	86	0
13	Konbu	6	8	2 kg box	6	24	0
14	Tofu	6	7	40 - 100 g pkgs.	23.25	35	0
15	Genen Shouyu	6	2	24 - 250 ml bottles	15.5	39	0
16	Pavlova	7	3	32 - 500 g boxes	17.45	29	0
17	Alice Mutton	7	6	20 - 1 kg tins	39	0	0
18	Carnarvon Tigers	7	8	16 kg pkg.	62.5	42	0
19	Teatime Chocolate	8	3	10 boxes x 12 piece	9.2	25	0
20	Sir Rodney's Marma	8	3	30 gift boxes	81	40	0

图 7-6　示例的页面效果

展开控制台面板中显示的URL地址，在参数标签中可看到提交“_dc”、“limit”、“page”和“start”4个参数。然后可看到以下的响应数据：

```
{
  "total": 77,
  "success": true,
  "data": [
    {
      "id": 1,
      "ProductName": "Chai",
      "QuantityPerUnit": "10 boxes x 20 bags",
      "ReorderLevel": 10,
      "SupplierID": 1,
      "UnitPrice": 18.0,
      "UnitsInStock": 39,
      "UnitsOnOrder": 0,
      "CategoryID": 1,
      "Discontinued": false
    },
   ...
}
```

通过这些信息，可以很好地了解提交的参数是否正确、服务器端是否运行正确以及返回的数据是否符合要求等情况，这是必须掌握的调试方法。

修改一下服务器端代码，将success的返回值改为false，刷新一下页面，会发现返回数据正常，但是Grid没显示了，这就是success的作用。当然，你也可以不设置success，显示也正常，但笔者不建议这样，因为Ext JS很多时候是靠这个标记来控制返回信息的正确与错误的，可以说是一套标准方法，例如form提交，当提交数据有错误或服务代码运行有错误时，可通过success返回false提示客户端产生了错误，让用户有一个好的体验。

为了方便使用控制台操作Store，先定义一个store变量，让其指向页面中的Store对象实例：

```
var store=Ext.StoreManager.get("ProductStore");
console.log(store)
```

“ProductStore”是创建Store时定义storeId，它会让Store对象实例在StoreManager对象中注册，通过get方法就可以指向它，然后调用它。console.log的目的是确保变量已经正确指向Store对象实例。如果显示以下信息，则表示已经正确指向了Store对象实例：

```
[Trial] Ext.data.Store <Products> [20] { storeId="ProductStore"}
```

在显示信息上单击鼠标右键，选择“在DOM标签中查看”菜单切换到DOM面板。不直接单击在“Illuminations”面板查看的目的是因为笔者感觉那里看的数据经过处理后，会有点找不到方向，不如DOM面板中直观。

在DOM面板中，查看一下data、filters、groupers和sorters这些对象，熟悉它们的结构对调试是非常有帮助的。

下面先让数据按照ProductName进行排序，在控制台输入以下命令（ProductName大小写）：

```
store.sort("ProductName");
```

运行后，会看到如图 7-7 所示，Grid 的显示已经按“ProductName”排序了。这说明，Grid 的排序与 Grid 本身无关，只与 Store 有关，Grid 只提供了一个操作界面给用户，而最后的执行是在 Store 中。

产品列表

id	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder
17	Alice Mutton	7	6	20 - 1 kg tins	39	0	0
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10	13	70
18	Carnarvon Tigers	7	8	16 kg pkg.	62.5	42	0
1	Chai	1	1	10 boxes x 20 bags	18	39	0
2	Chang	1	1	24 - 12 oz bottles	19	17	40
4	Chef Anton's Cajun	2	2	48 - 6 oz jars	22	53	0
5	Chef Anton's Gumb	2	2	36 boxes	21.35	0	0
15	Genen Shouyu	6	2	24 - 250 ml bottles	15.5	39	0
6	Grandma's Boysenb	3	2	12 - 8 oz jars	25	120	0
10	Ikura	4	8	12 - 200 ml jars	31	31	0
13	Konbu	6	8	2 kg box	6	24	0
9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	97	29	0
8	Northwoods Cranbe	3	2	12 - 12 oz jars	40	6	0
16	Pavlova	7	3	32 - 500 g boxes	17.45	29	0
11	Queso Cabrales	5	4	1 kg pkg.	21	22	30
12	Queso Manchego L	5	4	10 - 500 g pkgs.	38	86	0
20	Sir Rodney's Marm	8	3	30 gift boxes	81	40	0
19	Teatime Chocolate	8	3	10 boxes x 12 piec	9.2	25	0
14	Tofu	6	7	40 - 100 g pkgs.	23.25	35	0
7	Uncle Bob's Organi	3	7	12 - 1 lb pkgs.	30	15	0

图 7-7　按照 ProductName 重新排序后的 Grid 显示

你可能会发现，怎么没提交数据到服务器排序？因为在没创建 Store 时，没配置 remoteSort 属性为 true，也就不执行远程排序了。现在使用以下代码将其设置为 true：

```
store.remoteSort=true
```

然后单击 Grid 的 ProductName 标题栏，在控制台会看到有一个数据提交信息，展开后，在参数标签可看到提交的排序信息：

```
sort        [{"property":"ProductName","direction":"ASC"}]
```

当你不知道 Ext JS 是如何提交参数的时候，这是一个很好的了解提交参数的方法，也是必须掌握的。

再尝试一下多列排序，在控制台中输入以下代码：

```
store.sort(["CategoryID","ProductName"])
```

千万别少了中括号，不然会把 ProductName 当成排序方向了。这时候的提交参数变成了：

```
sort  {"property":"CategoryID","direction":"ASC"},{"property":
"ProductName","direction":"ASC"}]
```

现在测试一下过滤，在控制台输入以下命令：

```
store.filter("CategoryID",1)
```

运行后可到到 Grid 中只剩下 CategoryID 为 1 的数据了。在 DOM 面板中，看一下 Store 的 data 属性，会发现数据只剩下过滤后的数据了。如果要恢复，被删除的数据在哪里呢？往下找到 snapshot 属性，会看到其 items 属性下保存着全部数据，也就是说，过滤后，可在 snapshot 属性可找到全部数据。

因为没设置 remoteFilter 为 true，所以只在本地过滤，现在在控制台中将其设置为 true，然后再执行一次刚才的过滤代码，在控制台，将看到一个请求，在其参数标签可看到过滤的提交参数为：

```
filter[{"property":"CategoryID","value":1},{"property":"CategoryID","value":1}]
```

奇怪吧，为什么会有两个过滤对象？这是因为过滤不像排序那样，只有一种情况，例如你不能根据某个字段升序排序，同时做降序排序。但是，过滤允许你查询 CategoryID 值为 1 的同时又查询 CategoryID 值为 2 的数据。不过，这里有个问题，就是两者之间是与的关系还是或的关系。如果是本地排序，肯定是与的关系，有兴趣自己研究一下 AbstractMixedCollection 对象的 filter 方法就会清楚为什么是这样了。至于服务器端，这个可根据你需要自己调整。

在控制台使用 clearFilter 方法清除所有过滤，然后尝试一下分组：

```
store.group("SupplierID")
```

代码将以 SupplierID 进行分组，代码运行，会提交了一个请求，请求内包含了分组的信息：

```
group      [{"property":"SupplierID","direction":"ASC"}]
```

没有设置远程分组，为什么会提交请求？这是因为本地分组只是执行一次排序而已，而目前的状态还是远程排序状态，所以会提交请求。

使用以下命令恢复到本地排序和清除分组：

```
store.remoteSort=false;
store.clearGrouping();
```

然后在执行一次分组。语句执行后，没任何反应，原因是没对 Grid 进行分组显示设置，所以页面没有任何改变，在命令行中执行以下代码：

```
console.dir(store.getGroups())
```

代码运行后将看到如图 7-8 所示的输出信息，从图中可以看到，数据被分成了 10 组，每组的 name 属性记录的是分组字段的值，children 里则保存了在该组中的记录。

事实上，分组其实没做太多的事情，只是在 grouper 属性中添加一个 Grouper 对象实例而已，这个可以在 DOM 面板中看到。而 getGroups 方法返回的结果是根据 Grouper 对象实例临时组织的数据，为 Grid 的显示提供数据。

```
>>> console.dir(store.getGroups())
▼ 0                          Object { name=1, children=[3] }
  ▶ children                 [ [Trial] Products <Products> [20] { id="Products-1", internalId=1 }, [Trial]
                             Products <Products> [20] { id="Products-2", internalId=2 }, [Trial] Products
                             <Products> [20] { id="Products-3", internalId=3 } ]
    name                     1
▶ 1                          Object { name=18, children=[2] }
▶ 2                          Object { name=10, children=[1] }
▶ 3                          Object { name=20, children=[2] }
▶ 4                          Object { name=23, children=[1] }
▶ 5                          Object { name=16, children=[3] }
▶ 6                          Object { name=7, children=[1] }
▶ 7                          Object { name=12, children=[1] }
▶ 8                          Object { name=2, children=[4] }
▶ 9                          Object { name=6, children=[1] }
▶ 10                         Object { name=3, children=[1] }
```

图 7-8 执行“console.dir(store.getGroups())”命令后的控制台输出

7.6.2 读取 XML 数据

修改一下示例 7-5，将 reader 里 type 的值改为 xml。将读取记录的 root 配置项修改为 record。XMLReader 是使用 record 配置项定义数据节点的。

接着要做的是将服务器代码输出的 ContentType 修改为“text/xml”。

最后要做的就是将生成 JSON 格式的代码修改为生成 XML 文档的：

C#：

```
XDocument doc = new XDocument();
XElement xe= new XElement("Products",
 new XElement("total",total.ToString()),
 new XElement("success", "true"),
 from p in q
 select new XElement("data",
         new XElement("id",p.ProductID.ToString()),
         new XElement("CategoryID", p.CategoryID.ToString()),
         new XElement("ProductName", p.ProductName.ToString()),
         new XElement("QuantityPerUnit", p.QuantityPerUnit.ToString()),
         new XElement("ReorderLevel", p.ReorderLevel.ToString()),
         new XElement("SupplierID", p.SupplierID.ToString()),
         new XElement("UnitPrice", p.UnitPrice.ToString()),
         new XElement("UnitsInStock", p.UnitsInStock.ToString()),
         new XElement("UnitsOnOrder", p.UnitsOnOrder.ToString())
     )
 );
doc.Add(xe);
context.Response.Write(doc.ToString());
```

Java：

```
Document doc= DocumentHelper.createDocument();
Element root = doc.addElement("Products");
root.addElement("total").addText(String.valueOf(count));
root.addElement("success").addText("true");
while (rs.next()) {
   Element data =root.addElement("data");
```

```
    data.addElement("id").addText(rs.getString("ProductID"));
    data.addElement("ProductName").addText(rs.getString("ProductName"));
    data.addElement("SupplierID").addText(rs.getString("SupplierID"));
    data.addElement("CategoryID").addText(rs.getString("CategoryID"));
    data.addElement("QuantityPerUnit").addText(rs.getString("QuantityPerUnit"));
    data.addElement("UnitPrice").addText(rs.getString("UnitPrice"));
    data.addElement("UnitsInStock").addText(rs.getString("UnitsInStock"));
    data.addElement("UnitsOnOrder").addText(rs.getString("UnitsOnOrder"));
    data.addElement("ReorderLevel").addText(rs.getString("ReorderLevel"));
    data.addElement("Discontinued").addText(rs.getString("Discontinued"));
}
response.getWriter().write(doc.asXML());
```

代码修改完成后，在浏览器中打开页面，会看到是与读取 JSON 数据的效果是一样。因为数据已经转为 Ext JS 的格式，所以操作与原始数据格式无关。

7.6.3 Store 的数据操作

要对 Store 的数据进行操作，并将它们提交到服务器，首先要做的是区分这些数据是新建的、删除的或更新的，当然，对读取数据来说也是要的，也就是说，客户端会有 4 种不同类型的请求，而服务器端则根据不同的请求对数据进行不同的处理。

实现这个功能，Ext JS 提供了两种解决办法：在 Proxy 中定义 API 配置项、使用 Restful。API 配置项就是为每个操作配置一个路径；Restful 则是根据提交方式判断操作，具体实现可阅读 7.2.5 小节。

现在要做的是修改 7.5.2 节的示例，将 proxy 的 url 配置项屏蔽，然后添加一个 API 配置项：

C#：

```
api:{
    read:'Products.ashx',
    create:"action.ashx?act=add",
    update:"action.ashx?act=edit",
    destroy:"action.ashx?act=del"
},
```

Java：

```
api:{
    read:"/Restful/Products",
    create:"/Restful/Action?act=add",
    update:"/Restful/Action?act=edit",
    destroy:"/Restful/Action?act=del"
},
```

全部操作其实可以集中在一个文件内处理，笔者只是为了方便演示，分开了两个文件，这些都可根据你项目约定进行修改。

还要为 proxy 添加一个 writer：

```
writer:{
    type:'json'
}
```

好了，不急于写服务器代码，先执行一下，看看 create、update 和 destroy 这三种方式会提交什么的数据。

在浏览器中打开页面，然后使用 7.6.1 的代码，先在控制台作用域中定义一个 store 变量，然后添加一个记录：

```
store.add({
    ProductName:" 电脑 ",
    SupplierID:1,
    CategoryID:1,
    QuantityPerUnit:" 台 ",
    UnitPrice:3000,
    UnitsInStock:10,
    UnitsOnOrder:10,
    ReorderLevel:0,
    Discontinued:false
})
```

在 Grid 中会看到在底部添加了一条 id 为 0 的记录，说明 Store 已经保存了该记录，使用 getNewRecords() 方法可找出这条记录：

```
console.log(store.getNewRecords());
```

控制台会输出：

```
[[Trial] Products <Products> [21] { id="Products-ext-record-1", internalId="ext-
    record-1"}]
```

可以看到，记录是在数组里的。

好了，现在使用 sync 方法将数据同步到服务器：

```
store.sync()
```

在提交链接的 Post 标签内可看到如图 7-9 所示的信息。

参数 头信息 Post 响应 HTML

JSON

CategoryID	1
Discontinued	false
ProductName	"电脑"
QuantityPerUnit	"台"
ReorderLevel	0
SupplierID	1
UnitPrice	3000
UnitsInStock	10
UnitsOnOrder	10
id	0

源代码

{"id":0,"ProductName":"\u7535\u8111","SupplierID":1,"CategoryID":1,"QuantityPerUnit":"\u53f0","UnitPrice":3000,"UnitsInStock":10,"UnitsOnOrder":10,"ReorderLevel":0,"Discontinued":false}

图 7-9 Post 标签

在参数标签可看到如下信息：

```
_dc1   307712611235
act    add
```

对 POST 方式不熟悉的初学者，一定会以为使用字段名称就能获取到值，事实上是不行的，使用字段名称获取值，只适用 Content-Type 为 application/x-www-form-urlencoded 的情况。这里比较特殊，需要使用 InputStream 或 getInputStream 来读取数据。

下面来完成服务器端代码，不过在完成前，要先了解操作后数据应该如何返回？方法 sync 是使用 Proxy 对象 batch 方法提交数据的，而在 batch 方法中，创建 Batch 对象实例时，会为其绑定 operationcomplete 事件，该事件会在提交操作完成后执行。也就是操作完成后，会执行 Store 对象的 onBatchOperationComplete 方法，而 onBatchOperationComplete 方法会调用 onProxyWrite 方法。在 onProxyWrite 方法中，会根据操作类型（create、update、destroy）执行 onCreateRecords、onUpdateRecords 和 onDestroyRecords 三个方法。在 onCreateRecords 方法内，会移除添加的原始记录，然后将返回的记录插入 Store。在 onUpdateRecords 方法内，会用返回的记录替换原始记录。在 onDestroyRecords 方法内，则会根据返回记录删除原始记录。因为 MixedCollection 对象（Store 的数据结构是 MixedCollection 对象）的 insert、replace 和 remove 方法需要一个完整结构的数据进行操作，所以服务器端无论是添加、更新或删除操作，都必须将处理过的记录完整地返回客户端，不然就不能做到客户端数据与服务器端数据的同步，除非使用 load 重新加载数据。

这有点“雷人”，删除数据还要从数据库查询一次数据再删除，所以保持 Writer 对象的 writeAllFields 属性为 true，是一个好办法，它会将客户端的所有字段都提交到服务器端，如果是新增操作，将数据保存到数据库后，将字段产生或改变过的数据替换掉对应的 JSON 对象里的数据返回就可以了；如果是更新操作，和新增操作类似；如果是删除操作，则无须处理，直接将提交的 JSON 数据打包返回客户端即可。

因为数据返回后，会使用 Reader 对象将数据转换回模型实例，所以返回的数据格式与读取数据时的格式是一样的，只是不需要 total 这些属性。这里一定要 success 属性，因为代码是通过该属性决定服务器端的操作是否成功的，如果不为 true 或 undefined，都会认为处理失败，不对客户端数据进行任何处理。这时候信息字段非常重要，它可以在 Store 的事件中捕获这些信息，然后将发生的错误提示给用户。

目标明确后，可以写代码了：

C#：

```
public void ProcessRequest (HttpContext context) {
    context.Response.ContentType = "text/javascript";
    string action =context.Request.Params["act"] ?? "";
    switch (action.ToLower())
    {
  case "add":
      context.Response.Write(Add(context.Request));
      break;
  case "edit":
      break;
  case "del":
      break;
  default:
context.Response.Write("{success:false,msg:'错误的操作！ '}");
```

```
                break;
            }
        }

        private string Add(HttpRequest request){
            Stream resStream = HttpContext.Current.Request.InputStream;
            StreamReader sr = new StreamReader(resStream, System.Text.Encoding.
                Default);
            string data = sr.ReadToEnd();
            JObject datajo = JObject.Parse(data);
            datajo["id"] = 1000;
            JObject jo = new JObject(
                new JProperty("success",true),
                new JProperty("data",new JArray(
                    new JObject(new JProperty("id",1001))
                    //datajo
                    ))
                );

            return jo.ToString();
        }
```

Java：

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
   response.setContentType("text/javascript; charset=utf-8");
   String action = "";
   if(request.getParameter("act") != null){
           action=request.getParameter("act").toLowerCase();
   }
   if(action.equals("add")){
           response.getWriter().write(Add(request));
   }else if(action.equals("edit")){
   }else if (action.endsWith("del")) {
   }else {
           response.getWriter().write("{success:false,msg:'错误的操作! '}"
);
   }
}

private String Add(HttpServletRequest request) throws IOException {
   JsonStreamParser parser  = new JsonStreamParser(new InputStreamReader(request.
       getInputStream(), "UTF-8"));
   JsonObject org=new JsonObject();
   if(parser.hasNext()){
       org=parser.next().getAsJsonObject();
}
JsonObject jo = new JsonObject();
jo.addProperty("id", "1000");
jo.addProperty("CategoryID", org.get("CategoryID").toString());
jo.addProperty("ProductName", org.get("ProductName").toString());
jo.addProperty("SupplierID", org.get("SupplierID").toString());
jo.addProperty("QuantityPerUnit", org.get("QuantityPerUnit").toString());
```

```
jo.addProperty("UnitPrice", org.get("UnitPrice").toString());
jo.addProperty("UnitsInStock", org.get("UnitsInStock").toString());
jo.addProperty("UnitsOnOrder", org.get("UnitsOnOrder").toString());
jo.addProperty("ReorderLevel", org.get("ReorderLevel").toString());
jo.addProperty("Discontinued", org.get("Discontinued").toString());
 JsonArray ja=new JsonArray();
 ja.add(jo);
 JsonObject json=new JsonObject();
 json.addProperty("success",true);
json.add("data",ja);
return json.toString();
}
```

服务器端代码只是简单把 id 值修改后返回，大家都根据自己的情况修改相应的数据库操作代码再返回数据。这里 C# 代码会简单点，因为它的 JSON 对象可以直接修改值，Gson 对象则需要从旧的 JSON 对象读出再写入新的 JSON 对象。

再执行一次同步操作，这时候会看到新增的记录的 id 已经为 1000 了，是服务器返回的数据的 id 值。

现在尝试运行代码，然后执行同步操作，在提交链接的 POST 标签内可看到如图 7-10 所示的结果，从图中可看到，数据以数组形式提交了，也就是说，服务器端需要做一个判断，判断提交的数据是对象还是数组，这有点麻烦，设置 Wtire 对象的 allowSingle 为 false 是一个好的解决方案，这样提交的数据无论是一个还是多个，都是以数组形式提交了，服务器代码只要处理数组这一种情况就行了。

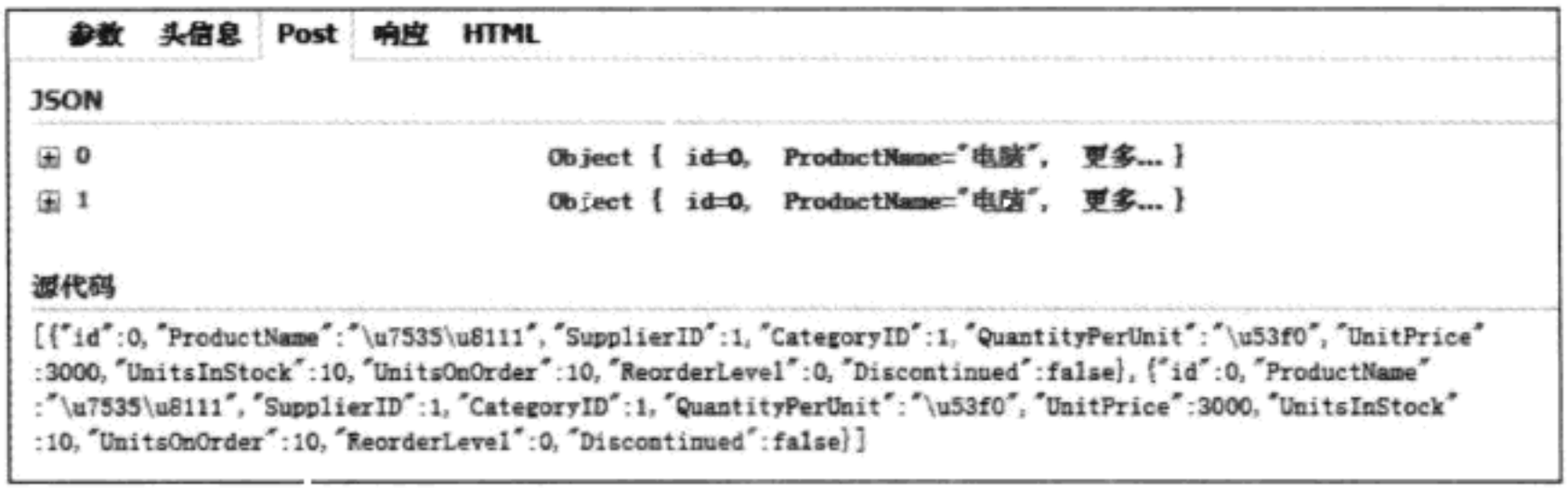


图 7-10 创建两条新记录后，同步显示提交的 POST 数据

现在试一下同时有新增数据和删除数据是怎么提交的，先刷新一下页面，新增一条记录后，使用下面的语句删除两条记录：

```
store.removeAt(4);
store.removeAt(6);
```

执行同步后，可以看到控制台出现了两个请求，第 1 个请求是执行新增记录的，第 2 个请求是执行删除的，也就是说，Store 会将这些操作根据操作类型分别提交。这里要注意的是，如果前面的请求不能获取正确的数据，表示操作不成功，例如服务器端代码出错，那么下一个请求就不会执行。你可以新增两条记录，然后删除两条记录，执行同步，来测试这个问题。

更新和删除操作的服务器端代码与新增记录差不多，就不啰嗦了，有兴趣可以自己尝试完成它。

下面修改一下页面里 writer 的定义，添加以下两个配置项：

```
root:'data',
encode:true
```

完成后，刷新页面，新增一个记录后，同步数据，将会看到如图 7-11 的 POST 数据。从数据可以看到，现在的 Content-Type 是 application/x-www-form-urlencoded，也就是说可以用习惯的方法提取数据了，而参数 data 就是 root 的值，数据还是 JSON 格式的数据。

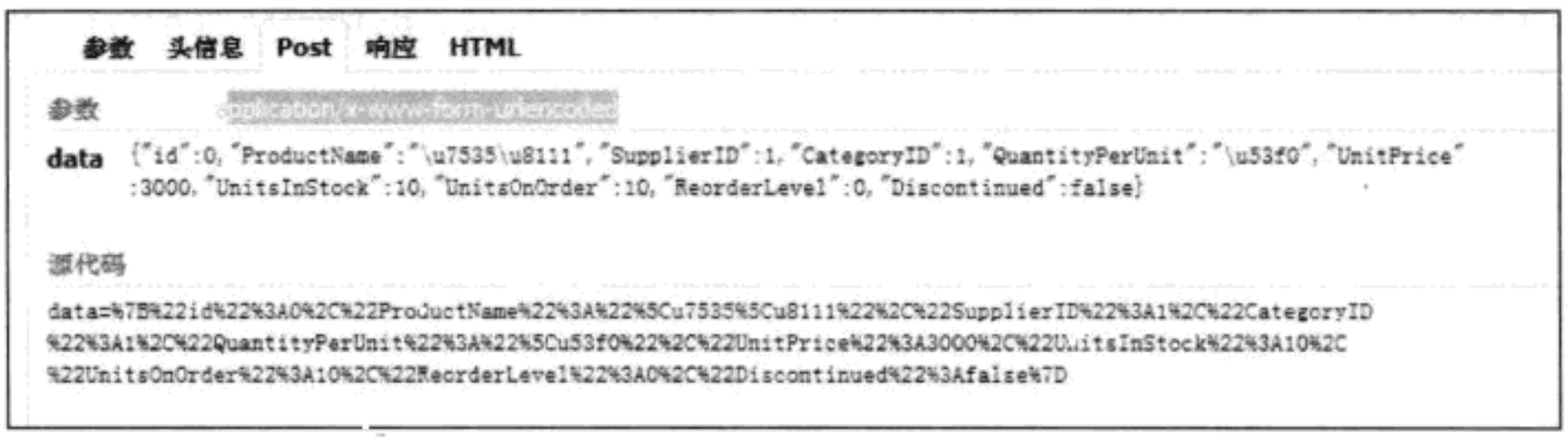


图 7-11　为 Writer 配置 root 和 encode 属性后的 POST 数据

以上配置可以根据自己习惯或者项目的约定灵活进行配置，主要是要明白如何在服务器端对这些数据进行处理。

7.7　本章小结

本章主要介绍了 Ext JS 的数据交互功能，是 Ext JS 库的一个重要组成部分，深入了解和掌握数据交换的方式以及它的处理过程，是学好用好 Ext JS 的前提之一。很多时候，初学者都会犯经验主义的错误，对 Store 等了解不够，导致出现错误或者需要处理数据时无从入手，造成开发上的举步维艰。

因而，笔者建议读者有时间，最好能自己看一次 data 目录下的源代码，深入了解数据的运作流程，这对使用 Ext JS 会有很大的帮助。

第 8 章　模板与组件基础

Ext JS 4 的一个重大改变，就是使用统一的渲染管道，让渲染过程更加条理化，更易扩展。而渲染管道的方式只有一个，即基于 Xtemplates 对象。所以，说模板是 Ext JS 组件的基石一点儿也不为过。掌握好模板，对掌握组件的使用、创建自定义组件是非常有帮助的。

本章将讲述模板类及其使用方法，并在此基础上讲述 Ext JS 的基础组件。

8.1　模板

8.1.1　模板概述

模板对开发人员来说，应该不陌生了，Ext JS 之所以强大，其中一个原因是它拥有强大的模板。试想如果没有模板，Ext JS 的组件或是通过 DOM 操作去实现，或是通过逐个拼接字符串的方法生成，其效率不用说都是相当低效的，而且维护这些代码费时费力，更别说用户自己去扩展组件了。笔者最大的感触是：jQuery 在 1.4.3 版前没有提供模板，当维护那些脚本输出的 HTML 代码时非常痛苦；找到插件也是各有特色，并不像 Ext JS 这样强大。从 1.4.3 版本开始，jQuery 终于加入了模板这个强大的工具。没有强大的模板，要构建一个好的 UI 库是相当困难的。

Ext JS 有 Template 和 XTemplate 两个模板类。其中 Template 类是基础模板，提供一些基础功能，而 XTemplate 是从 Template 扩展出来的、具有强大功能的模板类。

8.1.2　Ext.Template 的创建与编译

要创建 Template 对象，可以用 new 关键字，也可以用 Ext.Create 方法，根据自己的喜好选择一种方式即可。我们来看看实例是如何创建的，Template 对象的构造函数如下：

```
constructor: function(html) {
    var me = this,
        args = arguments,
        buffer = [],
        i = 0,
        length = args.length,
        value;

me.initialConfig = {};

if (length > 1) {
    for (; i < length; i++) {
        value = args[i];
```

```
            if (typeof value == 'object') {
                Ext.apply(me.initialConfig, value);
                Ext.apply(me, value);
            } else {
                buffer.push(value);
            }
        }
        html = buffer.join('');
    } else {
        if (Ext.isArray(html)) {
            buffer.push(html.join(''));
        } else {
            buffer.push(html);
        }
    }

    me.html = buffer.join('');

    if (me.compiled) {
        me.compile();
    }
},
```

代码中有两个关键的数据结果——buffer 数组和 initialConfig 对象，buffer 数组最后会将数组内的元素组成一个字符串赋值给 html 属性，而 initialConfig 对象则会保存参数中是对象的成员。

如果参数多于一个，则将不是对象类型的参数放到 buffer 数组中，如果是对象则将其复制到 initialConfig 对象和实例本身。

如果参数只有一个且为数组，则将参数转换为字符串再加到 buffer 数组中；如果不是数组，则直接加到 buffer 数组中。

以上代码说明，创建 Template 对象的参数可以是一个也可以是多个。如果是一个参数，可以为数组，也可以为字符串，例如：

```
var tpl = new Templated(['<div>','<a  href="{src}">{text}</a>','</div>']);
var tpl = new Templated('<div><a  href="{src}">{text}</a></div>');
```

以上代码的结果是一样的。

如果是多个参数，这时候就要有点规矩了，虽然没规定除了对象外的其他数据类型，但是最好还是用字符串，不然出了错误，还得自己去找原因。因而，上面的代码也可以这样定义：

```
var tpl = new Templated('<div>','<a  href="{src}">{text}</a>','</div>');
```

如果要加入配置项，可以这样：

```
var tpl = new Templated('<div>','<a  href="{src}">{text}</a>','</div>',
   {compiled: true, disableFormats:true}
);
```

Template 对象只有 compiled 和 disableFormats 两个配置项。其中，compiled 为布尔值，决定模板是否进行编译，默认值为 undefined，不编译，若设置为 true，则进行编译；

disableFormats 也为布尔值，决定是否使用格式化函数，默认值为 false，使用格式化函数。如果模板没有格式化函数的要求，最后设置为 true，可以提高运行速度。

编译与不编译有什么不同呢？这要从模板如何生成最终的 HTML 代码入手。模板是使用 apply 方法生成最终的 HTML 代码的，其代码如下：

```
apply: function(values) {
    var me = this,
        useFormat = me.disableFormats !== true,
        fm = Ext.util.Format,
        tpl = me;

    if (me.compiled) {
        return me.compiled(values).join('');
    }
    function fn(m, name, format, args) {
        if (format && useFormat) {
            if (args) {
                args = [values[name]].concat(Ext.functionFactory('return ['+ args
                    +'];')());
            } else {
                args = [values[name]];
            }
            if (format.substr(0, 5) == "this.") {
                return tpl[format.substr(5)].apply(tpl, args);
            }
            else {
                return fm[format].apply(fm, args);
            }
        }
        else {
            return values[name] !== undefined ? values[name] : "";
        }
    }
    return me.html.replace(me.re, fn);
},
```

上述代码有个明显的分支，如果 compiled 方法存在，会使用它生成 HTML 代码。

如果没有编译，则使用函数 fn 作为替换函数替换 html 内的文本，这不太好理解，我们通过一个例子来分析一下这个过程。例如，要处理以下字符串：

```
<div><a  href="{src}">{text:ellipsis(10)}</a><br/></div>
```

在浏览器打开模板页，然后在命令行输入以下代码：

```
var re = /\{([\w\-]+)(?:\:([\w\.]*)(?:\((.*?)?\))?)?\}/g;
var str='<div><a  href="{src}">{text:ellipsis(10)}</a><br/></div>';
str.replace(re,function(){
    console.log("replace:",arguments)
})
```

代码中变量 re 就是模板用来查找替换位置的正则表达式，变量 str 是要处理的字符串。代码运行后，会在控制台看到以下输出：

```
replace: ["{src}", "src", "", "", 15, "<div><a href="{src}">{...sis(10)}</
    a><br/></div>"]
replace: ["{text:ellipsis(10)}", "text", "ellipsis", "10", 22, "<div><a
    href="{src}">{...sis(10)}</a><br/></div>"]
"<div><a href="undefined">undefined</a><br/></div>"
```

可以看到，输出中的粗体代码就是函数里用 console.log 输出的参数，将输出参数对照一下 apply 方法中 fn 的参数，就很清楚代码的操作流程了。首先，检查 format 是否存在且 disableFormats 是否为 false，如果是，则将 apply 方法传递进来的对象或数组，根据 name（可以是关键字或数组索引）取出其值，作为数组的第 1 个元素，如果 args 参数存在，则将其作为数组的第 2 个元素。直接判断 format 是否以“this.”开头，如果是，则去掉“this.”后的字符串就是要调用的方法名称，在实例中调用该方法处理并返回替换，这时 args 会作为参数传递给方法。如果不是，format 为 Format 对象的方法，调用它处理并返回替换数据，这时 args 会作为参数传递给方法。如果不需要格式化处理数据，则直接返回替换数据。

现在再看看 compile 方法，其代码如下：

```
compile: function() {
    var me = this,
        fm = Ext.util.Format,
        useFormat = me.disableFormats !== true,
        body, bodyReturn;

    function fn(m, name, format, args) {
        if (format && useFormat) {
            args = args ? ',' + args: "";
            if (format.substr(0, 5) != "this.") {
                format = "fm." + format + '(';
            }
            else {
                format = 'this.' + format.substr(5) + '(';
            }
        }
        else {
            args = '';
            format = "(values['" + name + "'] == undefined ? '' : ";
        }
        return "'," + format + "values['" + name + "']" + args + ") ,'";
    }

    bodyReturn = me.html.replace(me.compileARe, '\\\\').replace(me.compileBRe,
        '\\n').replace(me.compileCRe, "\\'").replace(me.re, fn);
    body = "this.compiled = function(values){ return ['" + bodyReturn + "'].
        join('');};";
    eval(body);
    return me;
},
```

过程与 apply 方法的差不多，不过这里 fn 函数返回的是字符串。就这样看代码，估计很难清楚知道到底发生了什么，通过刚才的例子，编译一次，看看结果应该会清晰很多。在控制台中以刚才的字符串创建一个模板，然后编译一下，再看看生成的 compiled 方法里的内容

就清楚多了，执行代码如下：

```
var tpl= new Ext.Template('<div><a  href="{src}">{text:ellipsis(10)}</a><br/></
    div>');
tpl.compile();
console.log(tpl.compiled);
```

执行后控制台会输出“function()”，在上面单击鼠标右键，然后在菜单中选择“复制函数”，打开记事本，将刚才复制的内容粘贴到记事本中，可看到以下代码：

```
(function (values) {return ["<div><a  href=\"", values.src == undefined ? ""
    : values.src, "\">", fm.ellipsis(values.text, 10), "</a><br/></div>"].
    join("");})
```

到这里，还不明白？编译后，不需要使用正则表达式处理数据，而是直接使用值组合成字符串输出，速度比不编译要快。编译与否，可根据自己的情况处理。

8.1.3　格式化输出数据：Ext.String、Ext.Number、Ext.Date 和 Ext.util.Format

从上一节了解到，模板中，要替换数据的位置需要用大括号（“{}”）做标记，括号里可以是数字或字符串。如果是数字，表示的是数组中的索引；如果是字符串，则表示对象中的属性名称。如果需要输出格式化数据，则需要在索引或名称后加上冒号（“：”），然后是格式化函数的名称；如果需要传递参数，则用括号（“()”）括起来。例如，上一节例子中的“{text:ellipsis(10)}”就使用了 Format 对象的 ellipsis 方法格式化数据。

Ext JS 提供了丰富的格式化函数，分别在 Ext.String、Ext.Number、Ext.Date 和 Ext.util.Format 这 4 个对象中。这 4 个对象都不是使用 Ext.define 定义的类，而是 JavaScript 对象，在对象里包含了一些属性和方法，因而不需要实例化就可以通过命名空间直接调用。

1）Ext.String 对象是用来格式化字符串的，提供了 10 个方法。

- htmlEncode：对字符串进行 html 编码。
- htmlDecode：对字符串进行 html 解码。
- urlAppend：为地址添加查询字符串。
- trim：清除字符串开头和结尾的空白字符。
- capitalize：将字符串的开头字母转换为大写。
- ellipsis：根据指定的长度截断字符串，并在后面加上省略号（“…”）。
- escapeRegex：将字符串中正则表达式使用的特殊字符转义。
- toggle：可将字符串在两个值之间进行切换。例如按钮的文本要在“开始”和“停止”之间切换，可使用该方法。
- leftPad：输出固定长度的字符串，经常用于为数字添加前导字符“0”，例如数字为 123，要将其格式转换为“00123”则可使用该方法。
- format：将各类数据格式转换为字符串并输出。这个功能和 C# 与 Java 中字符串的 format 差不多，在字符串中可以使用大括号定位替换位置，大括号中的数字表示用参数中的哪个数据进行替换，例如：

```
Ext.String.format("我叫{0}，生于{1}年","张三",1990);
//输出："我叫张三，生于1990年"
```

2）Ext.Number 对象是用来格式化数字的，提供了 4 个方法。

- constrain：检查数字是否在指定范围内。如果数字小于最小值，返回最小值；如果数字大于最大值，返回最大值；否则，返回数字本身。
- snap：该方法主要用于 MultiSlider 取值。
- toFixed：根据指定的位数格式化数字的小数点后的输出位数。
- from：如果值不是数字，返回指定的默认值，否则返回值。

3）Ext.Date 对象是用来对日期进行格式化、运算等操作的，它有 16 个属性和 26 个方法。

常用的属性：

- defaultFormat：默认的日期格式，其默认值为“m/d/Y”。它会影响 Format 对象中的 date 方法和 dateRenderer 方法，因而在本地化文件中重新定义该值会简化代码的书写。

常用的方法：

- add：可进行日期运算，例如计算 20 天后是什么日期：

```
Ext.Date.add(new Date("2011/1/20"),Ext.Date.DAY,20);
//结果：Date {Wed Feb 09 2011 00:00:00 GMT+0800}
```

- between：检查指定日期是否在指定的范围内。如果在指定范围内，返回 true，否则返回 false。
- format：格式化日期输出。表 8-1 列出了常用的格式化字符。

表 8-1　常用的日期格式化字符

字　符	说　明
d	使用两位数字显示天数，前导字符为 0
j	不使用 2 位数字显示天数，因而天数不加前导字符 0
m	使用两位数字显示月份，前导字符为 0
n	不使用 2 位数字显示月份，因而月份不加前导字符 0
Y	使用 4 位数字显示年份
y	使用 2 位数组显示年份
G	使用 24 小时格式显示小时，没有前导字符 0
H	使用 24 小时格式显示小时，有前导字符 0
i	显示分钟，有前导字符 0
s	显示秒，有前导字符 0

- getDayOfYear：返回指定日期中其年份的天数。
- getWeekOfYear：返回指定日期中其年份的周数。
- isLeapYear：如果指定日期中的年份为闰年，返回 true。
- getFirstDayOfMonth：返回指定日期中其月份的第一天是星期几，返回值为 0 到 6 中的数字，0 表示星期日。

- ❑ getLastDayOfMonth：返回指定日期中其月份的最后一天是星期几，返回值为 0 到 6 中的数字，0 表示星期日。
- ❑ getFirstDateOfMonth：返回指定日期中其月份的第一天的日期值。
- ❑ getLastDateOfMonth：返回指定日期中其月份的最后一天的日期值。
- ❑ getDaysInMonth：返回指定日期中其月份的天数。
- ❑ now：返回当前时间。
- ❑ parse：根据指定格式将字符串转换为日期。

4）Ext.util.Format 除了提供自身的一些格式化函数外，还将其他 3 个格式化对象的函数转接过来以便模板使用。它提供了 5 个属性和 27 个方法。

属性：

- ❑ thousandSeparator：千位的分隔符号，默认值是逗号。本来不用修改的，在本地化文件中却被修改成了小数点，请自己修改本地化文件。
- ❑ decimalSeparator：小数点符号，默认值是小数点。本来不用修改的，在本地化文件中却被修改成了逗号，请自己修改本地化文件。
- ❑ currencyPrecision：货币精度，默认值是 2 位。
- ❑ currencySign：货币符号，默认值是美元符号。在本地化文件中已被修改为“￥”。
- ❑ currencyAtEnd：在货币数字后面附加的货币单位，默认值是 false。

方法：

- ❑ undef：如果值为 undefined，将其转换为空字符串，否则返回值。
- ❑ defaultValue：如果值为 undefined 或空字符串，返回指定的默认值，否则返回值。
- ❑ substr：将值转换为字符串后，再调用字符串的 substr 方法从指定位置开始获取指定长度的子字符串。
- ❑ lowercase：将值转换为字符串后，再调用字符串的 toLowerCase 方法将字符串中的字母转换为小写。
- ❑ uppercase：将值转换为字符串后，再调用字符串的 toLowerCase 方法将字符串中的字母转换为大写。
- ❑ usMoney：将数字格式化为美国货币格式。
- ❑ currency：将数字格式化为货币格式。
- ❑ date：将日期转换为指定格式的日期。
- ❑ dateRenderer：返回一个日期格式转换函数，这样就不需要每次使用 date 方法时都要加上日期格式定义，非常方便使用。最好的办法是在本地化文件的 Format 对象中添加一个应用程序常用的日期格式函数。
- ❑ stripTags：清除所有 html 标记。
- ❑ stripScripts：清除 script 标记。
- ❑ fileSize：格式化输出文件大小。如果文件大小小于 1024，则显示为“xxx bytes”（xxx 为具体数字，下同）；如果文件大小小于 1048576，则显示为“xxx KB”；如果文件大小大于等于 1048576，则显示为“xxx MB”。

- math：可以在模板中进行简单数学运算。该方法会返回一个函数，在方法里会构造一个匿名函数，匿名函数里根据模板定义进行简单的运算，例如定义为“{value:math("*10")}”，则会用 value 乘以 10 的值作为数据替换标记。
- round：根据指定的精度数格式化数字，也就是根据指定的位数返回小数点后的数字。
- number：根据指定的格式格式化数字。格式可包含的字符为 0、小数点和逗号。小数点后 0 的个数决定了显示的小数位数。使用逗号，正数部分从个位开始，每 3 个数字就显示一个逗号作为分隔符号。
- escapeRegex：转义字符，规避正则表达式中特殊字符。
- numberRenderer：与 dateRenderer 一样，可以固定一个数组格式函数。
- plural：将单词进行复数处理，对中文作用不大。
- nl2br：将换行符转换为“
”。
- capitalize：直接调用 Ext.String 的 capitalize 方法。
- ellipsis：直接调用 Ext.String 的 ellipsis 方法。
- format：直接调用 Ext.String 的 format 方法。
- htmlDecode：直接调用 Ext.String 的 htmlDecode 方法。
- htmlEncode：直接调用 Ext.String 的 htmlEncode 方法。
- leftPad：直接调用 Ext.String 的 leftPad 方法。
- trim：直接调用 Ext.String 的 trim 方法。
- parseBox：将数字或字符串转换为对象的 margin 尺寸。

如果以上格式化对象还不能满足需要，那么可以定义自己的格式化函数，例如，要将值中符合条件的字符以红色显示，可这样定义模板：

```
var tpl= new Ext.Template('<div><a  href="{src}">{text:this.hightlight}</a><br/></
    div>');
tpl.searchString=" 降价 ";
tpl.hightlight=function(v){
    var search=this.searchString,
        re=new RegExp(search,"g"),
        replace="<font color='red'>"+search+"</font>";
    return v.replace(re,replace);
}
var value={src:"/10000",text:" 汽车又降价了 "};
console.log(tpl.applyTemplate(value));
```

格式的关键是在冒号后加上“this.”，表示执行实例中的 hightlight 方法，从上节的源代码可以知道它是如何运行的。接着为实例绑定自定义的 hightlight 方法就行了。

代码执行后，可在控制台看到以下输出：

```
<div><a href="/10000"> 汽车又 <font color='red'> 降价 </font> 了 </a><br/></div>
```

这正是我们期望的效果。

灵活地运用这些格式化函数和自定义格式化函数，可以让模板发挥出更强大的威力。

8.1.4　超级模板：Ext.XTemplate（包括 Ext.XTemplateParser 和 Ext.XTemplateCompiler）

XTemplate 对象实在太强大了，高级已经不足以表达其强大，其强大的原因在于它能做以下事情：

- ❑ 可以自动填充数组数据。
- ❑ 可以执行判断语句。
- ❑ 与 Template 对象一样可以进行简单数学运算及执行实例内的方法。
- ❑ 模板有 values、parent、xindex 和 xcount 等 4 个内建的模板变量，用于特殊处理。
- ❑ 还可以根据需要自定义操作。

XTemplate 对象继承于 Template 对象，也就是说，其实例化过程与 Template 对象是一样的，最重要的修改是重写了 apply 方法，其代码如下：

```
apply: function(values) {
    return this.applyOut(values, []).join('');
},
```

在这里调用了 applyOut 方法，其代码如下：

```
applyOut: function(values, out) {
    var me = this,
        compiler;

    if (!me.fn) {
        compiler = new Ext.XTemplateCompiler({
            useFormat: me.disableFormats !== true
        });

        me.fn = compiler.compile(me.html);
    }

    try {
        me.fn.call(me, out, values, {}, 1, 1);
    } catch (e) {
        //<debug>
        Ext.log('Error: ' + e.message);
        //</debug>
    }

    return out;
},
```

如果 fn 属性不存在，则创建一个 XTemplateCompiler 实例。XTemplateCompiler 对象派生于 XTemplateParser 对象，其本身没有重写构造函数，因而会直接指向 XTemplateParser 对象的构造函数，其代码如下：

```
constructor: function (config) {
    Ext.apply(this, config);
},
```

只是将配置对象复制到实例中。

回到 applyOut 方法，创建 XTemplateCompiler 对象实例后，就调用其 compile 方法，代码如下：

```
compile: function (tpl) {
    var me = this,
        code = me.generate(tpl);

    return me.useEval ? me.evalTpl(code) : (new Function('Ext', code))(Ext);
},
```

先调用 generate 方法处理模板数据，其代码如下：

```
generate: function (tpl) {
    var me = this;

    me.body = [
        'var c0=values, p0=parent, n0=xcount, i0=xindex;\n'
    ];
    me.funcs = [
        'var fm=Ext.util.Format;' // note: Ext here is properly sandboxed
    ];
    me.switches = [];

    me.parse(tpl);

    me.funcs.push(
        (me.useEval ? '$=' : 'return') + ' function (' + me.fnArgs + ') {',
            me.body.join(''),
        '}'
    );

    var code = me.funcs.join('\n');

    return code;
},
```

从代码中可以了解到，generate 会生成一段用于生成 HTML 代码的脚本。其中，会调用 parse 方法对模板代码进行语法解释，这里就不深入研究具体的代码了，有兴趣的话可以自己去了解一下。我们来看看以下模板代码：

```
var html = [
    '<p>',
    '<h2>{name}</h2>',
    '<tpl if="discount <= .5">',
    '<h2 style="color:red">超级大优惠</h2>',
    '<tpl else>',
    '<h2 style="color:red">优惠</h2>',
    '</tpl>',
    '<a href="{bigimg}"><img src="{smallimg}" alt={name} /></a>',
    '市场价：{price}',
    '折扣：{discount}',
    '折扣价：{[values.price*values.discount]}',
    '<tpl for="comments">',
        '<div>{#}.{title}<br/>',
        '<p>{user}：{content}</p></div>',
```

```
    '</tpl>',
    '</p>',
].join('');
```

这是一个比较典型的模板，用来显示产品信息，其中包括了 if 语句、for 语句。打开模板页，然后在控制台输入以上模板代码后，再添加以下代码：

```
var compiler = new Ext.XTemplateCompiler({
    useFormat: true
});
console.log(compiler.compile(html));
```

运行后可看到控制台输出“function()”，单击进去，可看到以下代码：

```
var fm=Ext.util.Format;
function f1(out,values,parent,xindex,xcount) {
   try { with(values) {
           return(discount <= .5)
           }} catch(e) {}
   }
function f6(out,values,parent,xindex,xcount) {
   try { with(values) {
           return(comments)
   }} catch(e) {}
}
$= function (out,values,parent,xindex,xcount) {
   var c0=values, p0=parent, n0=xcount, i0=xindex;
   out.push('<p><h2>')
   out.push(String((values['name'] === undefined ? '' : values['name'])))
   out.push('</h2>')
   if (f1.call(this,out,values,parent,xindex,xcount)) {
       out.push('<h2 style="color:red">超级大优惠</h2>')
   } else {
       out.push('<h2 style="color:red">优惠</h2>')
   }
   out.push('<a href="')
   out.push(String((values['bigimg'] === undefined ? '' : values['bigimg'])))
   out.push('"><img src="')
   out.push(String((values['smallimg'] === undefined ? '' : values['smallimg'])))
   out.push('" alt=')
   out.push(String((values['name'] === undefined ? '' : values['name'])))
   out.push(' /></a>市场价：')
   out.push(String((values['price'] === undefined ? '' : values['price'])))
   out.push('折扣：')
   out.push(String((values['discount'] === undefined ? '' : values['discount'])))
   out.push('折扣价：')
   out.push(String(values.price*values.discount))
   var c1=f6.call(this,out,values,parent,xindex,xcount), a1=Ext.isArray(c1),p1=(p
   arent=c0),r1=values
   for (var i1=0,n1=a1?c1.length:(c1?1:0), xcount=n1;i1<n1;++i1){
       values=a1?c1[i1]:c1
       xindex=i1+1
       out.push('<div>')
       out.push(String((xindex === undefined ? '' : xindex)))
       out.push('.')
       out.push(String((values['title'] === undefined ? '' : values['title'])))
       out.push('<br/><p>')
```

```
            out.push(String((values['user'] === undefined ? '' : values['user'])))
            out.push(': ')
            out.push(String((values['content'] === undefined ? '' :
                values['content'])))
            out.push('</p></div>')
        }
        parent=p0;values=r1;xcount=n0;xindex=i0
        out.push('</p>')
    }
```

注意 在低版本的 Firebug 中可能看不到函数内的代码，所以建议更新到最新版的 Firebug。

根据函数代码可以看到其作用就是把值嵌入模板中，然后生成最终的 HTML 代码。

从函数代码可以看到，XTemplateCompiler 对象就是一个简单的编译器，可以将模板编译为执行脚本，用于输出 HTML 代码。这样做的好处就是可以在模板中实现逻辑，令模板功能更强大。

可以这样说，XTemplate 对象就是 Template 对象的基本数据显示方式加上 tpl 标记实现的高级功能组合而成的，非常强大，这也是 Ext JS 组件会以 XTemplate 对象为基准的根本原因，还是数据与显示分离的最好体现。

8.1.5 模板的方法

Template 对象提供了以下几个方法：

- ❑ append：将数据应用到模板，并追加到指定元素内。
- ❑ applyTemplate 或 apply：将数据应用到模板并返回生成的代码。
- ❑ compile：编译模板。
- ❑ from：从指定元素的值或 innerHTML 属性创建模板。
- ❑ insertAfter：将数据应用到模板，并在指定元素后插入。
- ❑ insertBefore：将数据应用到模板，并在指定元素前插入。
- ❑ insertFirst：将数据应用到模板，并作为指定元素的第一子节点插入。
- ❑ overwrite：将数据应用到模板，并覆盖元素的内容。
- ❑ set：将指定的 html 代码作为模板的 html 代码。

XTemplate 对象重写了 Template 对象的 apply 方法和 applyOut 方法，其余方法全部继承自 Template 对象。

8.2 组件的基础知识

8.2.1 概述

Ext JS 的一个最大特点就是组件丰富，而且功能强大。笔者当初选择 Ext JS 就是因为其组件多，整合性好，写法简练。实际上，框架众多，各有优点，但是缺点也不少，例如，Dojo

的主要问题是组件少、整合性差；而 Qoodoo 的组件虽多，但是代码不够简练，用它定义一个菜单，需要一个个定义好菜单项，再添加到菜单上；当时的 jQuery 没有组件，插件倒不少，要把 Ext JS 这样一套组件收集齐，难度是非常大的，笔者认为函数式的代码影响了其发展。

要充分了解 Ext JS 的组件，必须先明白以下几个概念：

- 布局（layout）：布局的作用就是控制和调整组件区域的大小和位置，因而几乎所有组件都需要相应的布局类来控制和调整组件区域的大小和位置。因而，要设计页面的布局，就要想到布局，用布局去控制各个区域的位置和大小，不要到容器里去找答案。
- 容器（container）：容器主要负责管理容器内的组件，包括增加、插入和删除等操作。
- 面板（panel）：面板是特殊的容器，除了包含容器的功能外，还固定了结构，其结构主要包括标题栏、顶部工具栏、底部工具栏、左边工具栏、右边工具栏、停靠工具栏和主体等 7 个部分，因而只要看到面板，就应该条件反射地知道它大概是什么样子，包括哪些东西。
- 视图（view）：视图父对象的别名是 DataView，其形象地表明了视图就是用来显示数据的，Grid 和 Tree 等控件都要与数据打交道，都要显示 Store 中的数据，那么，它们必然会用到视图。图表是特例，因为它们是使用图像而不是使用文本方式来组织数据。

明白以上 4 个概念很重要，例如，页面组件的位置不对，并不是预想的那样，极有可能是用错布局了。又如，要显示的内容或组件没显示出来，可能是容器或面板出了问题。

Ext JS 包括 89 个以 AbstractComponent 对象为父对象的组件，38 个以 Layout 对象为父对象的布局对象，还有 3 个比较特殊的对象——Floating 对象、Shadow 对象和 Layer 对象，总数为 130 个，这还不包括那些用户自定义的组件。数量是惊人的，工作量是巨大的，使用是非常方便的。有点回到 Delphi 时代的感觉，进入了一个良性循环：因为组件多、好用，吸引了大量用户，因大量用户为它添加组件，从而吸引更多的用户。

8.2.2　组件类的整体架构

因为只用一个图没有办法显示组件的整体架构，所以只能将其拆分成几个图来说明。如果想要完整的图，可在本书的资源包内的“组件类图”文件夹中找到图片。

组件从抽象基类 AbstractComponent 开始，从其子类 Component 开始派生出如图 8-1 所示的 19 个组件。在这 19 个组件里，派生了 AbstractContainer、AbstractView 和 BaseField 这 3 个重要分支。AbstractContainer 类是所有容器类的基类；AbstractView 是所有视图类的基类；BaseField 则是表单组件的基类。余下的 16 个都是基本的功能组件，以下是这些组件的说明：

- ColorPalette：简单的颜色选择器。
- DatePicker：日期选择器。
- HtmlEditor：HTML 编辑器。
- Label：标签。
- Tool：面板标题栏上的小按钮。
- Button：按钮。
- Editor：为页面元素提供内联编辑的文本框。

- FlashComponent：用来显示 Flash 的组件。
- Img：显示图片的组件。
- ToolbarItem：工具栏组件的基类。
- Fill：用于填充工具栏的组件，以实现工具栏组件的右对齐。
- ProgressBar：进度条。
- MenuItem：菜单上组件的基类。
- Splitter：分隔条。
- Handle：为元素或组件调整大小提供句柄。
- DrawComponent：画图组件，可以在上面使用 SVG 或 VML 画图。

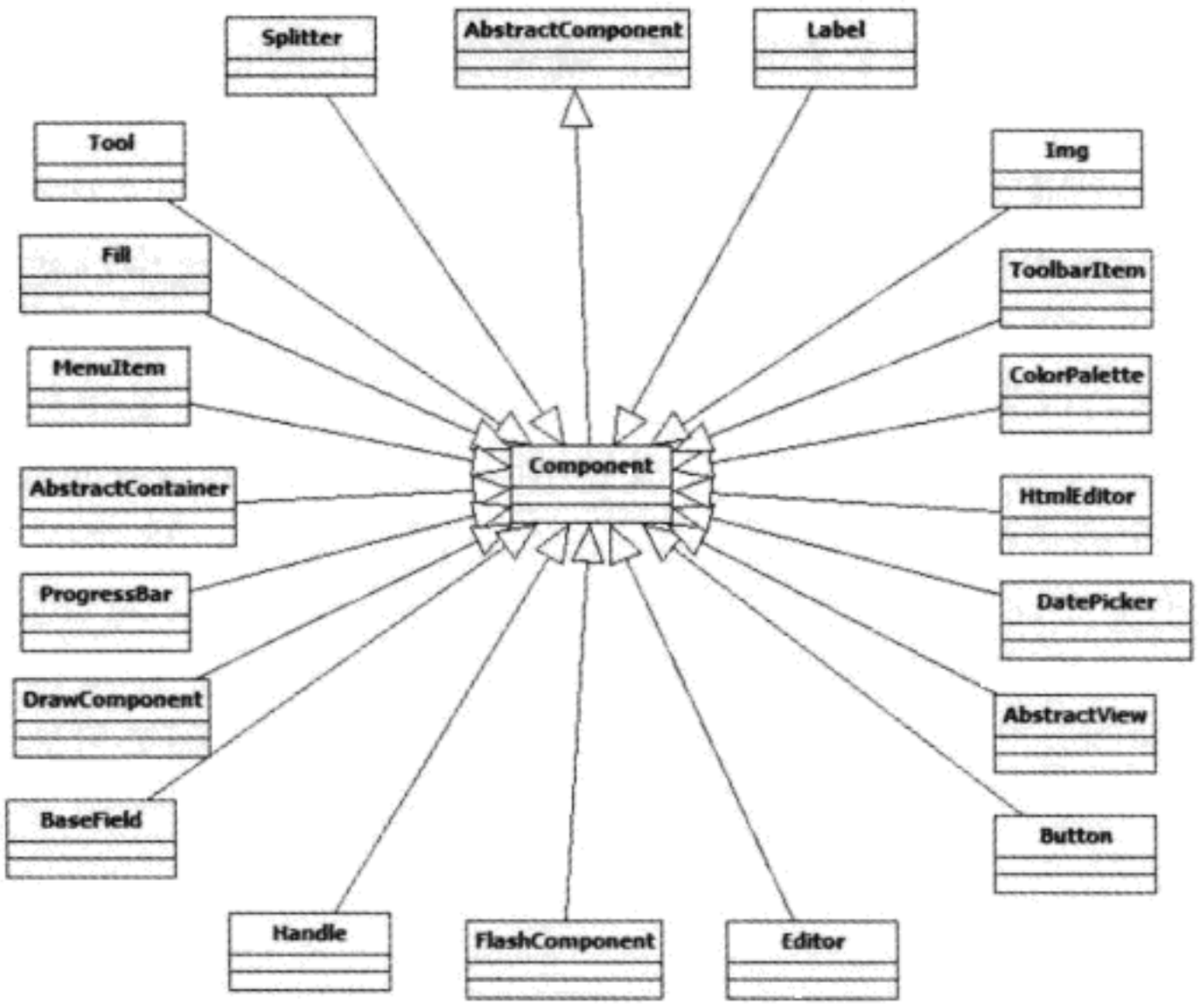


图 8-1　从 Component 类派生出的 19 个子类

以 BaseField 类为基类的表单组件一共有 16 个，其继承关系如图 8-2 所示。直接从 BaseField 类派生出了以下 5 个组件类：

- Hidden：用于显示隐藏文本框。
- Display：用于显示只读的值。
- Text ：基本文本框。从其派生出 File 类用于提交文件，TextArea 类用于输入多行文本，Trigger 类则为文本框添加单击按钮，然后派生出 Picker 类和 Spinner 类。Picker 类是下拉选择框的基类，从其派生出 TimeField 用于选择时间 ；DateField 用于选择日期 ；ComboBox 则是通用的下拉选择框。Spinner 提供了上下两个按钮，用于改变文本框内的值，从其派生出只能输入和调整数组的 NumberField。
- MultiSlider：多滑块组件，从其派生出单滑块组件（Slider）。
- Checkbox：复选框，从其派生出单选框（Radio）。

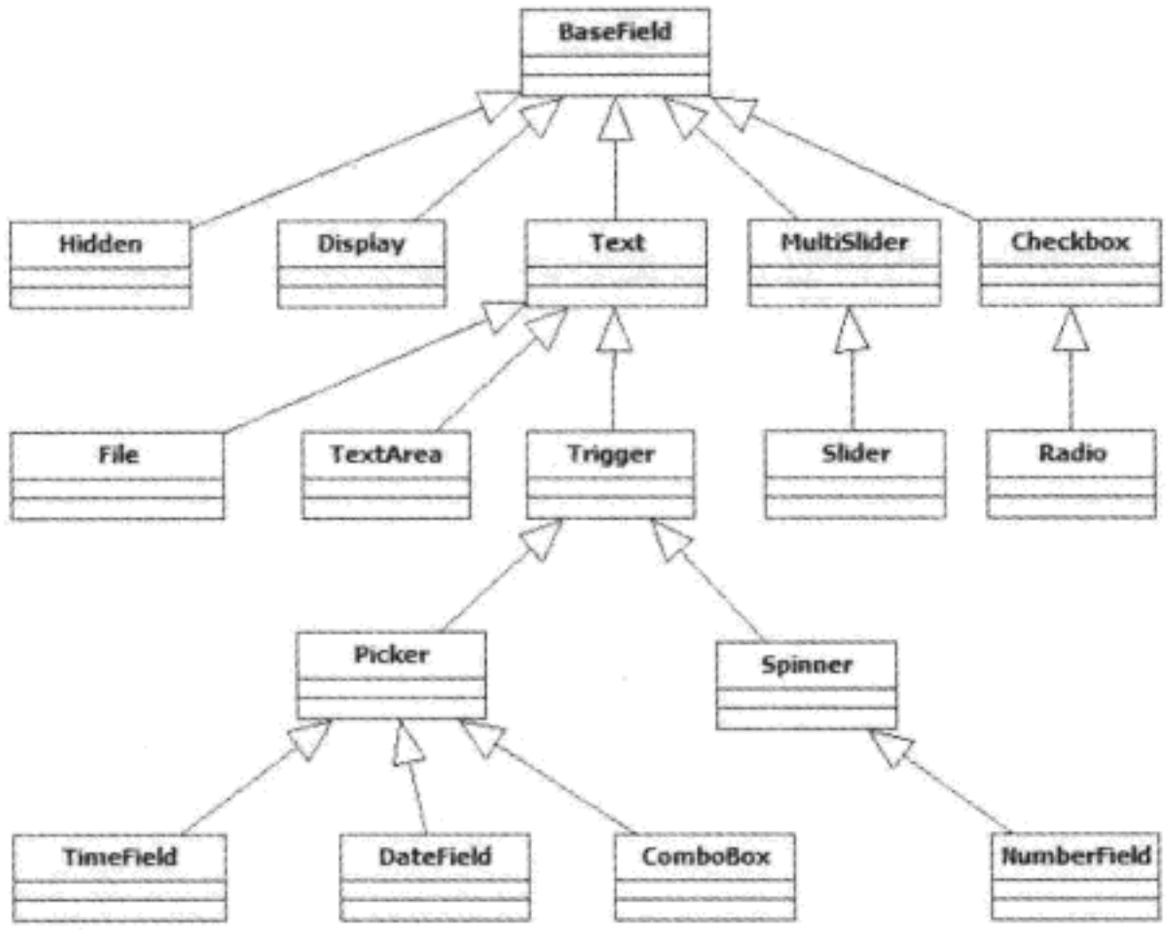


图 8-2　BaseField 类及其子类关系图

以 AbstractContainer 类为基类的容器类一共有 37 个，其继承关系如图 8-3 和图 8-4 所示，其中图 8-4 为以 AbstractPanel 类为基类的面板类。容器类是组件的第一大类，这是因为工具栏、Grid 的标题等都是容器，甚至文本框也需要一个容器为它配置标签和提供错误信息显示。从抽象基类 AbstractContainer 类派生出 Container 类后，再派生出以下 7 个类：

- FieldSet：用来分组输入组件。
- FieldContainer：输入组件容器，为输入组件提供标签和错误信息显示。从其派生出用于复选框的容器 CheckboxGroup，与单选框派生于复选框类似，单选框的容器 RadioGroup 也派生于 CheckboxGroup。
- Header：面板的标题栏。从其派生出 TabPanel 的 TabBar，即标签页的标签。
- Viewport：将浏览器的可视区域作为一个容器。
- Toolbar：工具栏，从其派生出分页工具栏 PagingToolbar。
- HeaderContainer：TablePanel 的标题，实际多用于 Grid 的列标题。从其派生出 PropertyHeaderContainer 类用于 PropertyGrid；Column 类用于定义 Grid 的列，又根据显示的数据或方式派生出 DateColumn、BooleanColumn、ActionColumn、NumberColumn、TemplateColumn 和 RowNumberer 这 6 个类。
- AbstractPanel：面板的抽象基类。

从 AbstractPanel 派生出 Panel 类后，从其派生出了以下 7 个面板类：

- Window：窗口是一个特殊的面板。从其派生出 MessageBox 类。
- FormPanel：表单面板。
- TablePanel：Ext JS 用于改善 Grid 以及树的性能和功能的重要面板，从其派生出 GridPanel 和 TreePanel，从 GridPanel 又派生出 PropertyGrid。
- ButtonGroup：按钮组，用来制作 Ribbon 界面的好组件。
- TabPanel：标签面板。

- Menu：菜单容器。从其派生出用于菜单的日期选择器（MenuDatePicker）和颜色选择器（ColorPicker）。
- Tip：提示信息。从其派生出用于滑块组件的 SliderTip 和通用信息提示组件 ToolTip，又从 ToolTip 派生出特殊的信息提示类 QuickTip。

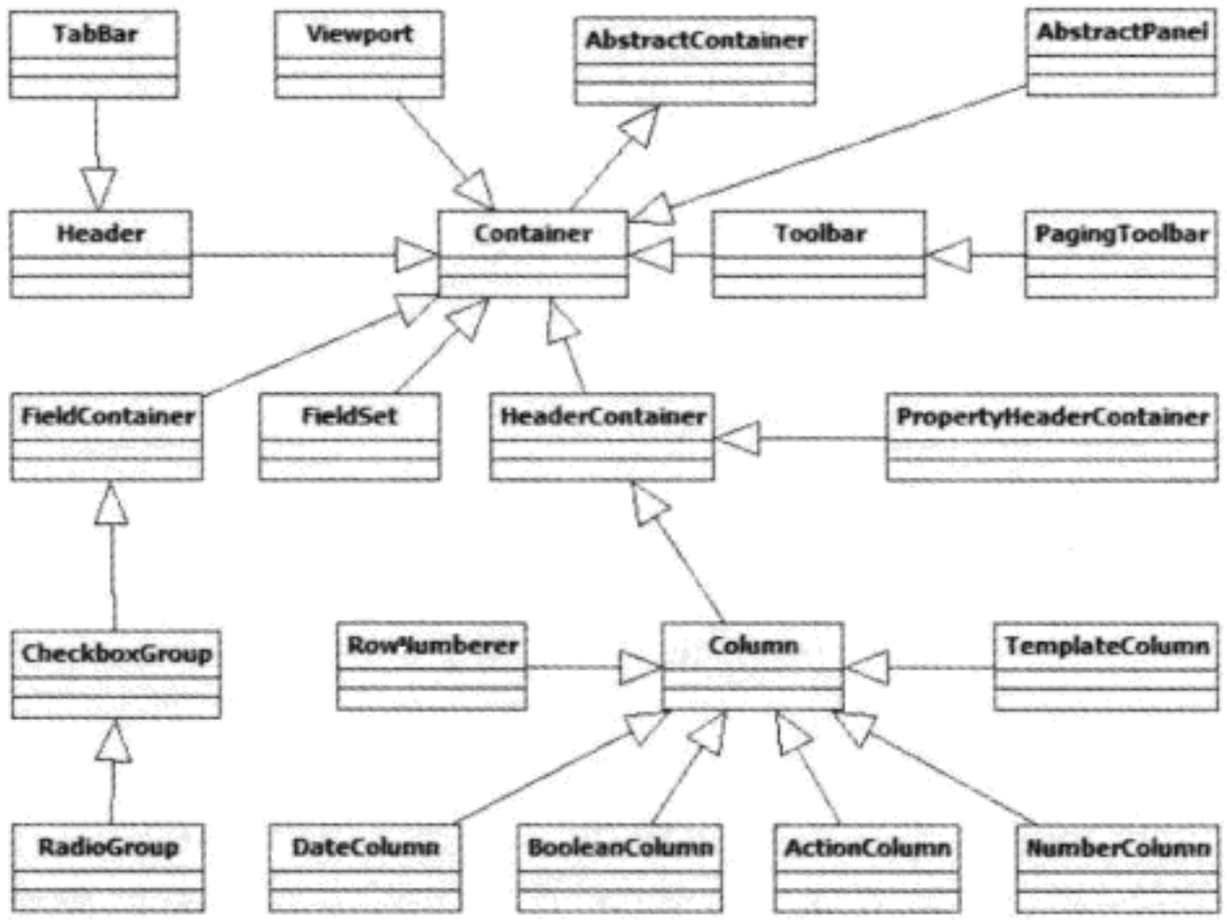


图 8-3　AbstractContainer 类及其子类关系图

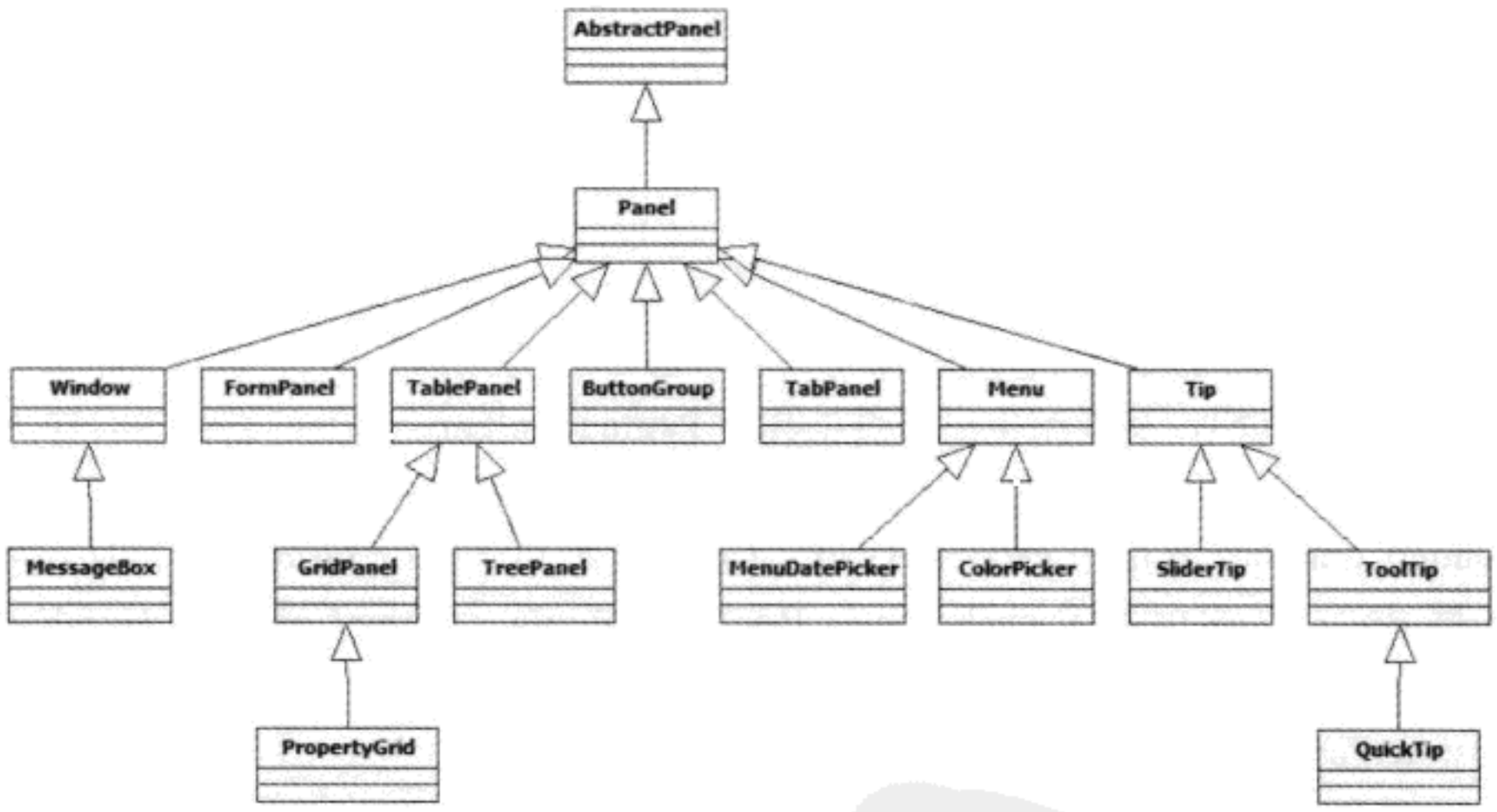


图 8-4　AbstractPanel 类及其子类关系图

在组件中，AbstractView、MenuItem、Button、ToolBarItem 和 DrawComponent 类都有子类，其结构如图 8-5 所示，以下是这些类的说明：

- AbstractView：视图的抽象基类，从其派生出 DataView。然后从 DataView 派生出 TableView 和 BoundList。又从 TableView 派生出用于 Grid 的 GridView 及用于树的 TreeView。BoundList 类主要用于 ComboBox、MultiSelect 和 ItemSelector，显示数据列

表，从其又派生出显示时间列表的 TimePicker。

- CheckItem：派生于 MenuItem，带复选框的菜单项。
- MenuSeparator：派生于 MenuItem，菜单的分隔条。
- Chart ：派生于 DrawComponent，显示图表的组件。没错，只有这一个类，在这里没分饼图、柱状图等子类，因为不需要，只要为该组件提供画图数据就可以画出你需要的图表。
- SpiltButton ：派生于 Button，为按钮提供一个下拉箭头。从其派生出 CycleButton，为按钮提供一个下拉菜单，菜单过多时可自动滚动。
- Tab：派生于 Button，标签页的标签。
- Separator：派生于 ToolbarItem，工具栏的分隔条。
- ToolBarTextItem：派生于 ToolbarItem，在工具栏显示文本。
- Spacer：派生于 ToolbarItem，在工具栏上添加空白。

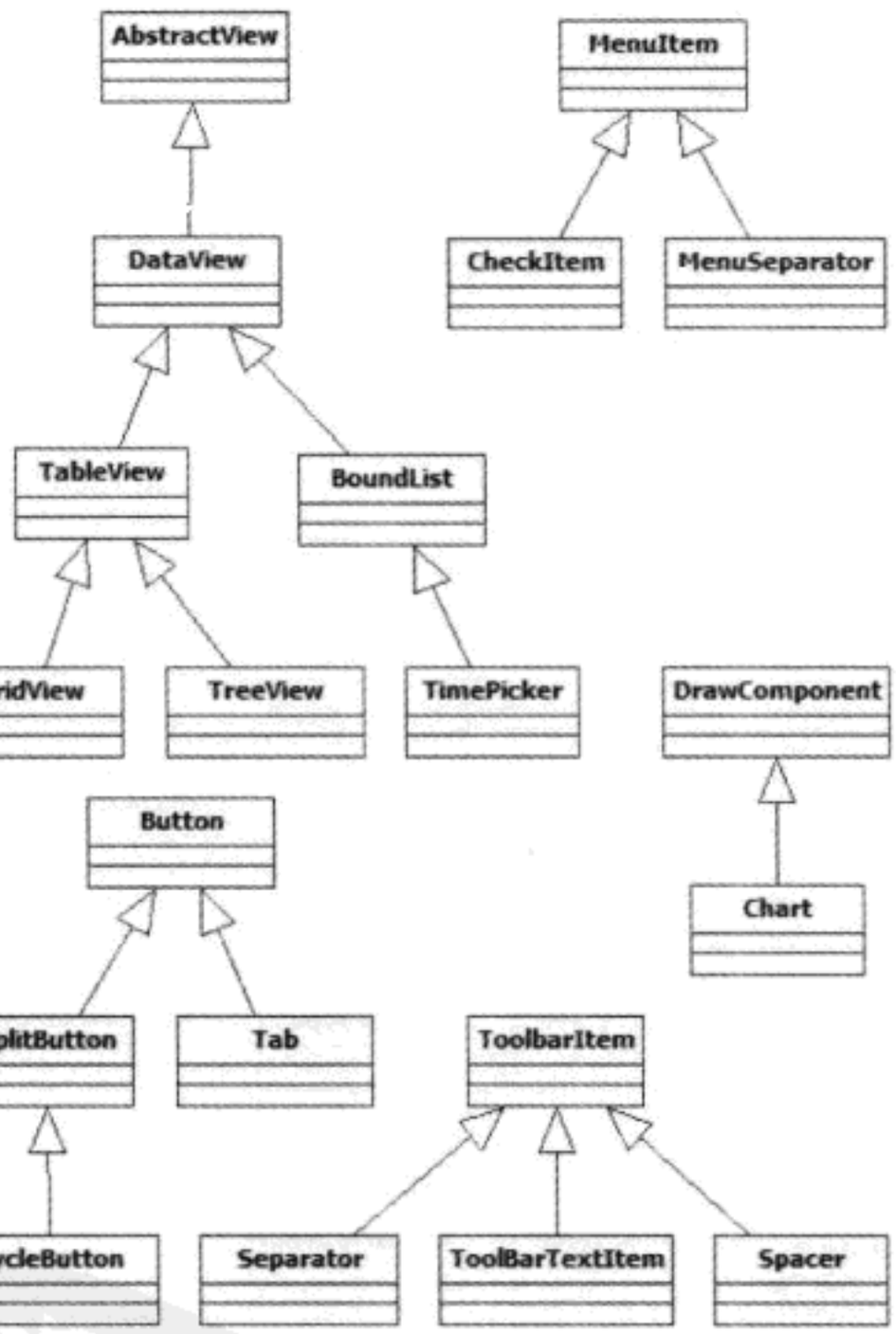


图 8-5　AbstractView、MenuItem、Button、ToolBarItem 和 DrawComponent 类的子类

以上就是 89 个以 Abstract Component 对象为父对象的组件的具体架构。了解这些，可以清楚地了解组件的构成，从而在使用或调试时，可根据组件的构成，也就是其继承关系，知道使用的组件具有什么特性。例如，用到 GridPanel，就要清楚其是继承自面板，而面板的特色是由 5 个部分组成的。又如使用图表，就要清楚图表的显示只有一个组件，其余的是提供数据的。这对于排错也是很有用的，可以根据组件特点，找出错误原因。如果要自己扩展组

件，那就更有用了，可根据组件特点，清楚自己的组件最适合从哪个类扩展。

8.2.3 布局类的整体架构

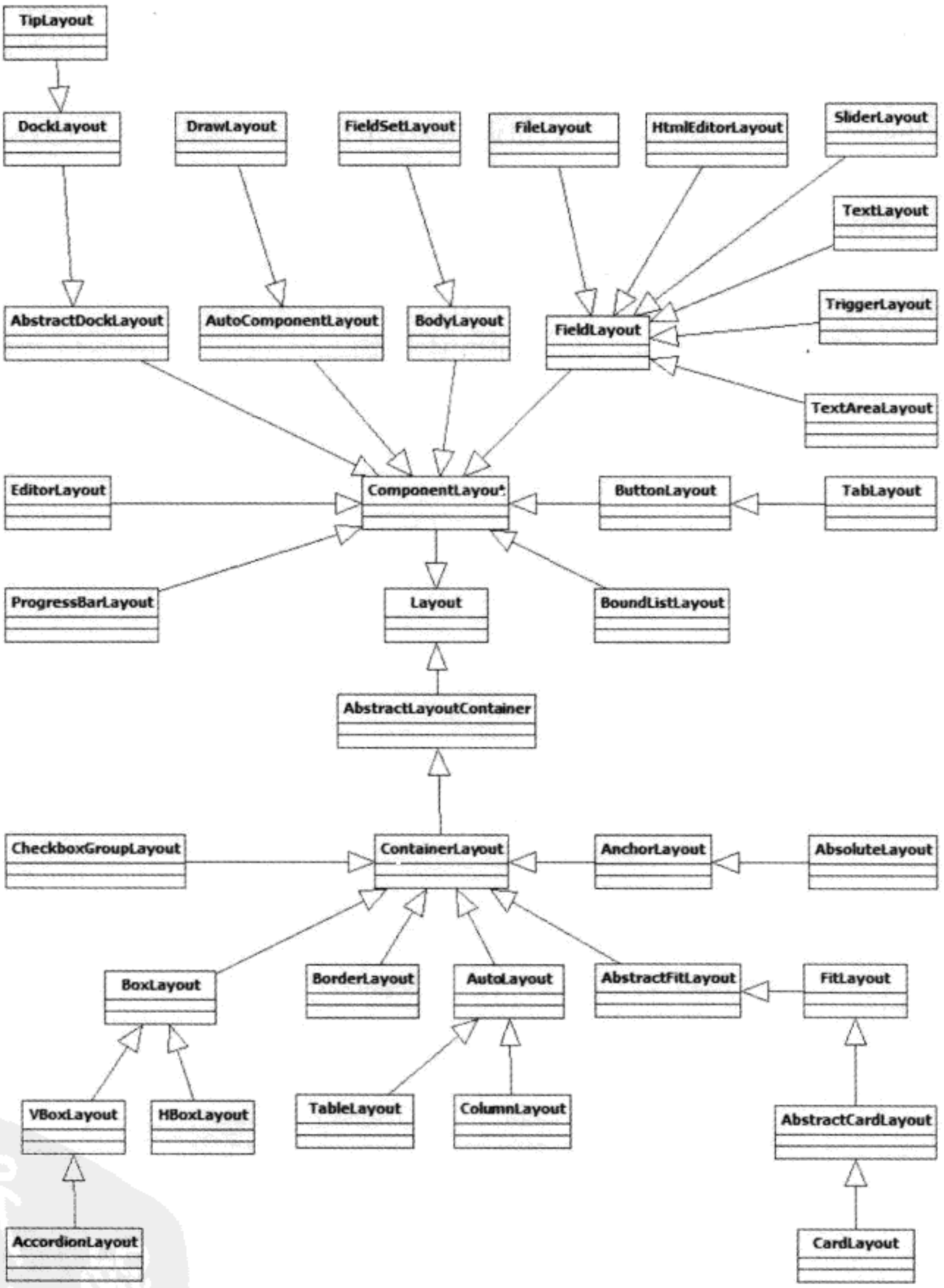


图 8-6 Layout 类的类结构图

布局类以 Layout 类为基类，派生出 AbstractLayoutContainer 和 ComponentLayout 两个分支，其结构如图 8-6 所示。

AbstractLayoutContainer 类为通用布局的抽象基类，从其派生出 ContainerLayout 类后，派生出以下的 15 个通用布局类：

- CheckboxGroupLayout：顾名思义，用于 CheckboxGroup 和 RadioGroup 的布局。
- AnchorLayout：固定布局，会根据容器大小固定其相对于容器的尺寸。
- AbsoluteLayout：派生于 AnchorLayout，绝对布局，可使用坐标进行布局。
- AbstractFitLayout：自适应布局的抽象基类。
- FitLayout：派生于 AbstractFitLayout，自适应布局，如果布局内只有一个组件，会将该组件自动填充满容器。
- AbstractCardLayout：派生于 FitLayout，CardLayout 的抽象基类。
- CardLayout ：卡片布局，该布局可维护多个子组件，每个子组件都会自适应于容器，而每次只会显示一个子组件，因而多用于实现向导功能。
- BorderLayout：边界布局，可将一个容器划分成几个区域，每个区域可以展开或折叠。
- BoxLayout：VBoxLayout 和 HBoxLayout 布局的基类。
- VBoxLayout：垂直布局，派生于 BoxLayout，用于垂直划分容器的布局。
- AccordionLayout：手风琴布局，派生于 VBoxLayout，实现 Accordion 效果的布局。
- HBoxLayout：水平布局，派生于 BoxLayout，用于水平划分容器的布局。
- AutoLayout：当容器没有定义布局时，会默认使用该布局作为其布局。
- TableLayout：表格布局，派生于 AutoLayout，使用 HTML 的 Table 作为布局。
- ColumnLayout：列布局，派生于 AutoLayout，将容器分成列的布局。

ComponentLayout 类为组件专用布局的基类，从其派生出专门用于特定组件的布局，如 EditorLayout 用于 Editor 组件，HtmlEditorLayout 用于 HtmlEditor。因为是专用布局，不建议使用，所以就不详细说明了，有兴趣可以自己进行研究。

了解布局类与了解组件类的架构同样重要，因为它是与组件息息相关的类，可以说组件离不开布局，有组件就有布局。

8.2.4　组件的创建流程

要了解组件的创建流程，只要了解 AbstractComponent 对象的创建流程就行了，因为它是所有组件的基类。所有组件，无论其有无构造函数，其继承深度有多少层，都会执行 AbstractComponent 对象的构造函数，没有构造函数的，肯定是用其父的构造函数；重写了构造函数的，必然会使用 callParent 方法一层层地调用其父的构造函数，最终都会执行 AbstractComponent 对象的构造函数。

Ext JS 4 组件最大的变化是使用统一的渲染管道，那么怎样才能实现统一的渲染管道呢？这必然要先将渲染流程统一。既然渲染流程是统一的，最好的办法当然是将流程放在组件基类，这样在避免重复的代码之余也避免了组件开发人员的不小心改动流程。因而放置渲染流程的最佳对象就是 AbstractComponent，反过来说，AbstractComponent 对象统一了组件的渲染流程。在 Ext JS 4.1 中，组件渲染又更新了技术，使用了批处理方式渲染。

综合以上两点，可确定 AbstractComponent 对象的创建流程就是组件的创建流程。至于组

件之间的渲染差异，完全可通过重写渲染方法解决，这样既可以保证渲染管道的统一，又不至于让每个组件都渲染成一样。

目标明确后，可以工作了，AbstractComponent 对象的构造函数代码如下：

```
constructor : function(config) {
    var me = this,
        i, len, xhooks;

    if (config) {
        Ext.apply(me, config);

        xhooks = me.xhooks;
        if (xhooks) {
            me.hookMethods(xhooks);
            delete me.xhooks;
        }
    } else {
        config = {};
    }

    me.initialConfig = config;

    me.mixins.elementCt.constructor.call(me);

    me.addEvents(
          // 省略事件代码
    );

    me.getId();

    me.setupProtoEl();

    me.mons = [];
    me.renderData = me.renderData || {};
    me.renderSelectors = me.renderSelectors || {};

    if (me.plugins) {
        me.plugins = [].concat(me.plugins);
        me.constructPlugins();
    }

    me.initComponent();
    Ext.ComponentManager.register(me);

    me.mixins.observable.constructor.call(me);
    me.mixins.state.constructor.call(me, config);

    this.addStateEvents('resize');

    if (me.plugins) {
        me.plugins = [].concat(me.plugins);
        for (i = 0, len = me.plugins.length; i < len; i++) {
            me.plugins[i] = me.initPlugin(me.plugins[i]);
        }
```

```
    }

    me.loader = me.getLoader();

    if (me.renderTo) {
        me.render(me.renderTo);
    }

    if (me.autoShow) {
        me.show();
    }

        // 省略调试信息
},
```

如果存在配置对象，把配置对象复制到实例；如果存在钩子（xhooks），调用 hookMethods 方法把钩子方法挂上。如果没有配置对象，创建一个空的配置对象。

接着把 initialConfig 属性指向配置对象；初始化混入的 ElementContainer 对象（用于管理子组件）；调用 addEvents 方法添加事件；调用 getId 方法设置实例的 id，如果没定义，会自动生成一个 id。id 属性对组件来说是必需的，这和组件管理有关。

接着调用 setupProtoEl 方法创建 ProtoElement 对象实例，用于管理组件主元素的样式。

接着定义了 1 个数组和 2 个对象。数组 mons 的目的不是很明确，代码中只有定义，从没使用。对象 renderData 记录了渲染的数据，包括 ui、uiCls、baseCls、componentCls 和 frame 等属性。对象 renderSelectors 会以选择符为关键字，指向通过选择符查询到的元素（Element 对象）。

接着是处理插件，会调用 constructPlugin 方法，其实就是根据配置项的类型去创建一个插件实例，也就是说数组 plugins 的每个元素都指向一个插件实例。

接着调用 initComponent 方法对组件进行初始化，通过该方法可实现每个组件的独特功能。

接着是在组件管理器（ComponentManager）里注册组件。然后初始化事件和状态，还要使用 addStateEvents 将 resize 事件绑定到状态，也就是当触发了 resize 事件时，状态要做记录。

接着是使用 initPlugin 方法初始化插件，也就是调用插件的 init 方法。

接着是使用 getLoader 方法创建 ComponentLoader 的实例，如果设置 autoLoad，会自动加载数据。

如果设置了 renderTo，执行 render 方法（由混入的 Renderable 对象提供），开始渲染组件，其代码如下：

```
render: function(container, position) {
    var me = this,
        el = me.el && (me.el = Ext.get(me.el)), // ensure me.el is wrapped
        tree,
        nextSibling;

    Ext.suspendLayouts();
```

```
    container = me.initContainer(container);

    nextSibling = me.getInsertPosition(position);

    if (!el) {
        tree = me.getRenderTree();
        if (nextSibling) {
            el = Ext.DomHelper.insertBefore(nextSibling, tree);
        } else {
            el = Ext.DomHelper.append(container, tree);
        }
        me.wrapPrimaryEl(el);
    } else {
        me.initStyles(el);
        if (me.allowDomMove !== false) {
            if (nextSibling) {
                container.dom.insertBefore(el.dom, nextSibling);
            } else {
                container.dom.appendChild(el.dom);
            }
        }
    }

    me.finishRender(position);

    Ext.resumeLayouts(!container.isDetachedBody);
},
```

代码先调用 suspendLayouts 方法暂停布局，其代码如下：

```
suspendLayouts: function () {
    var me = this;
    if (!me.rendered) {
        return;
    }
    if (++me.layoutSuspendCount == 1) {
        me.suspendLayout = true;
    }
},
```

如果 rendered 为 false，表示还未渲染，直接返回。

如果计算器 layoutSuspendCount 加 1 后其值等于 1，设置 suspendLayout 属性为 true，表示要暂停布局运算。

回到 render 方法，接着调用 initContainer 方法初始化容器，其代码如下：

```
initContainer: function(container) {
    var me = this;

    if (!container && me.el) {
        container = me.el.dom.parentNode;
        me.allowDomMove = false;
    }
    me.container = container.dom ? container : Ext.get(container);
```

```
        return me.container;
    },
```

如果参数 container 不存在，且 el 指向一个元素，那么将容器设置为 el 的父节点。

如果参数 container 的 dom 属性存在，说明已经是 Element 对象实例；否则调用 get 方法获取实例，再返回。

容器初始化后，调用 getInsertPosition 获取组件的插入位置，其代码如下：

```
    getInsertPosition: function(position) {
        if (position !== undefined) {
            if (Ext.isNumber(position)) {
                position = this.container.dom.childNodes[position];
            }
            else {
                position = Ext.getDom(position);
            }
        }

        return position;
    },
```

如果参数 position 是数字，把容器内由 position 指定位置的子节点作为插入点，否则调用 getDom 方法获取节点。

返回后，如果 el 不存在，调用 getRenderTree 方法返回渲染树，其代码如下：

```
    getRenderTree: function() {
    var me = this;

    me.beforeRender();

    if (me.fireEvent('beforerender', me) !== false) {
        me.rendering = true;

        if (me.el) {
            return {
                tag: 'div',
                id: (me.$pid = Ext.id())
            };
        }

        return me.getElConfig();
    }

    return null;
    },
```

这里先调用 beforeRender 方法做渲染前的处理，其代码如下：

```
    beforeRender: function () {
        var me = this,
            layout = me.getComponentLayout();

        if (!layout.initialized) {
```

```
            layout.initLayout();
        }

        me.setUI(me.ui);

        if (me.disabled) {
            me.disable(true);
        }
    },
```

这里先调用 getComponentLayout 方法获取组件布局，其代码如下：

```
    getComponentLayout : function() {
        var me = this;

        if (!me.componentLayout || !me.componentLayout.isLayout) {
            me.setComponentLayout(Ext.layout.Layout.create(me.componentLayout,
                'autocomponent'));
        }
        return me.componentLayout;
    },
```

如果组件布局还没创建，就调用 Layout 对象的 create 方法创建布局。其过程就是实例化一个布局对象，在此不详细说明，有兴趣可以自己阅读相关源代码。

布局创建后，如果它还没进行初始化（initialized 为 false），调用布局的 initLayout 方法。一般情况下，initLayout 方法只是设置 initialized 属性为 true，仅仅在 CheckboxGroup 对象中重写了该方法，因而在此就不展开了。

接着调用 setUI 方法设置用于自定义的组件样式。

最后设置 disabled 属性为 true，冻结所有事件，结束 beforeRender 方法，返回 getRenderTree 方法，触发 beforerender 事件。

如果自定义事件代码返回值不为 false，则设置 rendering 为 true，表示正在渲染。

如果 el 存在，返回一个包含 tag、id 成员的对象，用于 Helper 对象生成节点。否则，调用 getElConfig 方法使用渲染模板和数据来生成元素，并返回。

返回后，根据插入位置 nextSibling 决定是调用 Helper 对象的 insertBefore 还是 append 添加节点。

节点添加完成后，调用 wrapPrimaryEl 方法将 el 属性指向该节点。

如果 el 已存在，则调用 initStyles 方法初始化样式，然后根据插入位置插入节点。

接着调用 finishRender 方法完成渲染，其代码如下：

```
finishRender: function(containerIdx) {
    var me = this,
        tpl, data, contentEl, el, pre, hide, target;

    if (!me.el || me.$pid) {
        if (me.container) {
            el = me.container.getById(me.id, true);
        } else {
            el = Ext.getDom(me.id);
```

```
}

if (!me.el) {
    me.wrapPrimaryEl(el);
} else {
    delete me.$pid;

    if (!me.el.dom) {
        me.wrapPrimaryEl(me.el);
    }
    el.parentNode.insertBefore(me.el.dom, el);
        Ext.removeNode(el); // remove placeholder el
    }
} else if (!me.rendering) {
    tpl = me.initRenderTpl();
    if (tpl) {
        data = me.initRenderData();
        tpl.append(me.getTargetEl(), data);
    }
}

if (!me.container) {
    me.container = Ext.get(me.el.dom.parentNode);
}

if (me.ctCls) {
    me.container.addCls(me.ctCls);
}

me.onRender(me.container, containerIdx);

target = me.getTargetEl();
target.setStyle(me.getOverflowStyle());

me.el.setVisibilityMode(Ext.Element[me.hideMode.toUpperCase()]);

if (me.overCls) {
    me.el.hover(me.addOverCls, me.removeOverCls, me);
}

me.fireEvent('render', me);

if (me.contentEl) {
    pre = Ext.baseCSSPrefix;
    hide = pre + 'hide-';
    contentEl = Ext.get(me.contentEl);
    contentEl.removeCls([pre+'hidden', hide+'display', hide+'offsets',
        hide+'nosize']);
    target.appendChild(contentEl.dom);
}

me.afterRender();
me.fireEvent('afterrender', me);
me.initEvents();
```

```
        if (me.hidden) {
            me.el.hide();
        }
    },
```

如果 el 不存在，但“$pid”存在，那么，当 container 存在时，根据 id 获取元素。否则调用 getDom 方法获取元素。

如果 el 还不存在，则调用 wrapPrimaryEl 方法获取。如果存在，则删除成员“$pid”，并确保 el 是 Element 对象实例；然后调用 insertBefore 在父节点插入元素，并移除占位符 el。

如果 rendering 为 false，调用 initRenderTpl 方法初始化模板，根据组件 renderTpl 属性的定义创建模板实例。如果 initRenderTpl 方法成功返回模型实例，则调用 initRenderData 方法，初始化渲染数据，然后将模板生成的 HTML 代码追加到目标元素中，完成渲染。

接着是获取容器，并调用 addCls 方法为容器添加样式。

接着调用 onRender 方法，修改渲染状态，其代码如下：

```
onRender: function(parentNode, containerIdx) {
    var me = this,
        x = me.x,
        y = me.y,
        lastBox, width, height,
        el = me.el;

    if (Ext.scopeResetCSS && !me.ownerCt) {
        if (el.dom == Ext.getBody().dom) {
            el.parent().addCls(Ext.baseCSSPrefix + 'reset');
        }
        else {
            me.resetEl = el.wrap({
                cls: Ext.baseCSSPrefix + 'reset'
            });
        }
    }

    me.applyRenderSelectors();

    delete me.rendering;

    me.rendered = true;

    lastBox = null;

    if (x !== undefined) {
        lastBox = lastBox || {};
        lastBox.x = x;
    }
    if (y !== undefined) {
        lastBox = lastBox || {};
        lastBox.y = y;
    }
    if (!me.getFrameInfo()) {
        width = me.width;
        height = me.height;
```

```
        if (typeof width == 'number') {
            lastBox = lastBox || {};
            lastBox.width = width;
        }
        if (typeof height == 'number') {
            lastBox = lastBox || {};
            lastBox.height = height;
        }
    }

    me.lastBox = me.el.lastBox = lastBox;
},
```

如果 scopeResetCSS 属性存在且不存在 ownerCt，那么要为元素添加 reset 样式。

接着调用 applyRenderSelectors 方法应用渲染选择器，其代码如下：

```
    applyRenderSelectors: function() {
        var me = this,
            selectors = me.renderSelectors,
            el = me.el,
            dom = el.dom,
            selector;

        me.applyChildEls(el);

        if (selectors) {
            for (selector in selectors) {
                if (selectors.hasOwnProperty(selector) && selectors[selector]) {
                    me[selector] = Ext.get(Ext.DomQuery.selectNode(selectors
                        [selector], dom));
                }
            }
        }
    },
```

这里先调用 applyChildEls 方法取得子元素，其代码如下（ElementContainer.js 内）：

```
applyChildEls: function(el, id) {
    var me = this,
        childEls = me.getChildEls(),
        baseId, childName, i, selector, value;

    baseId = (id || me.id) + '-';
    for (i = childEls.length; i--; ) {
        childName = childEls[i];

        if (typeof childName == 'string') {
            value = el.getById(baseId + childName);
        } else {
            if ((selector = childName.select)) {
                value = Ext.select(selector, true, el.dom); // a CompositeElement
            } else if ((selector = childName.selectNode)) {
                value = Ext.get(Ext.DomQuery.selectNode(selector, el.dom));
            } else {
                value = el.getById(childName.id || (baseId + childName.itemId));
```

```
                }

                childName = childName.name;
            }

            me[childName] = value;
        }
    },
```

上述代码先调用 getChildEls 方法获取子元素，然后遍历返回的子元素数组。

如果子元素是字符串，则通过 getById 获取元素对象。

如果不是字符串，则先判断子元素是否存在 select 属性，如果存在，通过 Ext.select 方法返回 CompositeElement 对象。如果存在 selectNode 属性，则调用 Ext.get 方法返回 Element 对象。如果以上都不是，则调用 getById 返回 Element 对象。

然后在对象内加入一个以子元素 name 属性为关键字、子对象为值的成员。

返回 applyRenderSelectors 方法后，根据 renderSelectors 属性的值，在实例内加入已定义的选择符为关键字，调用 Ext.get 方法获得对象为值的成员。

返回 onRender 方法后，删除成员 rendering，设置 rendered 属性为 true，表示渲染已经结束。

最后就是为组件的边框计算宽度和高度。

完成 onRender 方法的调用后，调用 setStyle 方法设置元素样式，调用 setVisibilityMode 设置元素的隐藏模式。如果要有鼠标移动效果，则调用 hover 方法设置效果。

接着触发 render 事件。

如果 contentEl（放置内容的元素）存在，则获取元素，并调用 appendChild 方法将其追加到目标元素内。

接着调用 afterRender 方法，其代码如下：

```
afterRender : function() {
    var me = this;

    me.finishRenderChildren();

    if (me.styleHtmlContent) {
        me.getTargetEl().addCls(me.styleHtmlCls);
    }

    if (!me.ownerCt) {
        me.updateLayout();
    }
},
```

先调用 finishRenderChildren 方法，完成子组件的渲染，其代码如下：

```
finishRenderChildren: function () {
    var layout = this.getComponentLayout();

    layout.finishRender();
},
```

先调用 getComponentLayout 获取组件布局，如果不存在，则会创建布局实例，然后调用布局的 finishRender 方法。要注意的是，finishRender 方法在当前 4.1 beta 1 版本只有 Ext.layout.component.Dock 对象和 Ext.layout.container.Container 对象这两个地方有具体定义，在这里只研究 Ext.layout.container.Container 对象中的方法，因为这是容器布局类的父类，如边界布局、垂直布局等。具体代码如下：

```
finishRender: function () {
    var me = this,
        target, items;

    me.callParent();

    me.cacheElements();

    target = me.getRenderTarget();
    items = me.getLayoutItems();

    if (me.targetCls) {
        me.getTarget().addCls(me.targetCls);
    }

    me.finishRenderItems(target, items);
},
```

先调用 callParent 方法调用父类的同名方法，但其为空函数，因而这里可以忽略。接着调用 cacheElements 方法，其代码如下：

```
    cacheElements: function () {
        var owner = this.owner;

        this.applyChildEls(owner.el, owner.id);
    },
```

上述代码调用了 applyChildEls 方法获取子元素。

回到 finishRender 方法，接着调用 getRenderTarget 方法返回渲染目标节点，然后调用 getLayoutItems 方法获取子组件，其代码如下：

```
getLayoutItems: function() {
    var owner = this.owner,
        items = owner && owner.items;

    return (items && items.items) || [];
},
```

也就是返回组件代码定义中的 items 配置项。

回到 finishRender 方法，在完成给目标元素添加样式后，调用 finishRenderItems 方法开始完成子组件的渲染，其代码如下（Layout.js 中）：

```
finishRenderItems: function (target, items) {
    var length = items.length,
        i, item;
```

```
        for (i = 0; i < length; i++) {
            item = items[i];

            if (item.rendering) {
                item.finishRender(i);
                this.afterRenderItem(item);
            }
        }
    },
```

在这里会遍历子组件，然后逐个调用其 finishRender 方法，也就是重复组件的创建流程完成子组件的渲染。

最后调用 afterRenderItem 方法，目前版本只有卡片布局有该方法的具体定义，只需是设置子组件的隐藏和显示，在此就不具体讲述了。

完成子组件渲染后返回 afterRender 方法。更新样式后，如果组件不存在 ownerCt 属性，则调用 updateLayout 方法，其代码如下：

```
updateLayout: function (comp, defer) {
    var me = this,
        running = me.runningLayoutContext,
        pending;

    if (running) {
        running.queueInvalidate(comp);
    } else {
        pending = me.pendingLayouts || (me.pendingLayouts = new Ext.layout.
            Context());
        pending.queueInvalidate(comp);

        if (!defer && !me.layoutSuspendCount && !comp.isLayoutSuspended()) {
            me.flushLayouts();
        }
    }
}
```

在这里会建立一个 Context 对象实例，通过队列的方式对组件进行批处理渲染，因本书篇幅有限，具体的机制就不详细探讨了，只要知道在这里会通过批处理方法计算组件的尺寸、位置，并将结果渲染到 DOM 就行了。

afterRender 方法执行完后，返回 finishRender 方法触发 afterRender 事件。

然后调用 initEvents 方法初始化事件，其代码如下：

```
initEvents : function() {
    var me = this,
        afterRenderEvents = me.afterRenderEvents,
        el,
        property,
        fn = function(listeners){
            me.mon(el, listeners);
        };
```

```
if (afterRenderEvents) {
    for (property in afterRenderEvents) {
        if (afterRenderEvents.hasOwnProperty(property)) {
            el = me[property];
            if (el && el.on) {
                Ext.each(afterRenderEvents[property], fn);
            }
        }
    }
}
me.addFocusListener();
}
```

在代码中会用到 afterRenderEvents 属性，该属性的值为一个对象，在 AbstractComponent 对象的 addListener 方法中，也就是为组件绑定事件的方法中，会将绑定事件的元素及调用函数作为对象存储到 afterRenderEvents 属性指向的对象中。这样在组件渲染后，就可以通过调用 initEvents 方法将事件绑定到元素。

在 initEvents 事件最后，会调用 addFocusListener 方法，为元素添加 tabIndex 属性，这样元素就可以接收焦点了。

返回 finishRender 方法后，如果组件 hidden 属性为 true，则隐藏组件，完成 finishRender 方法的执行，返回到 render 方法，执行 Ext.resumeLayouts 方法恢复布局计算。

至此渲染就完成了，返回到组件的构造函数。

如果组件 autoShow 属性为 true，调用 show 方法显示组件。

至此，整个组件的创建流程就完成了。

总结以上分析，可得出如图 8-7 所示的流程图。流程中各组件一般会重写 initComponent、onRender、afterRender 和 initEvents 这些方法，这样，各组件就可实现自己的 UI、功能和事件了。

在 Component 对象的构造函数中，还添加了将组件绑定到 Action 对象的流程；在 initComponent 方法中则添加了事件绑定和开启冒泡事件的流程；在 afterRender 方法中添加了设置自动滚动、设置页面位置、初始化 Resizer 对象、初始化拖放对象等流程。

8.2.5　常用的组件配置项、属性、方法和事件

1. 常用的配置项

- autoEl：可以为 HTML 标记名称或 DomHelper 对象，它会被用来作为封装组件的元素。
- autoRender：布尔值，为 true 时会自动渲染，主要用于浮动组件，例如窗口。配置该项就可在显示之前执行渲染，而无须在 show 的时候才渲染。对于 Window 类，该配置项默认值是 true。
- autoScroll：布尔值，为 true 时组件会自动显示滚动条。
- autoShow：布尔值，为 true 时会自动显示组件。
- center：将组件居中显示。
- cls：可以为组件配置额外的 CSS 样式类。
- contentEl：将该值指定的 HTML 元素或元素的 id 作为组件内容的容器。

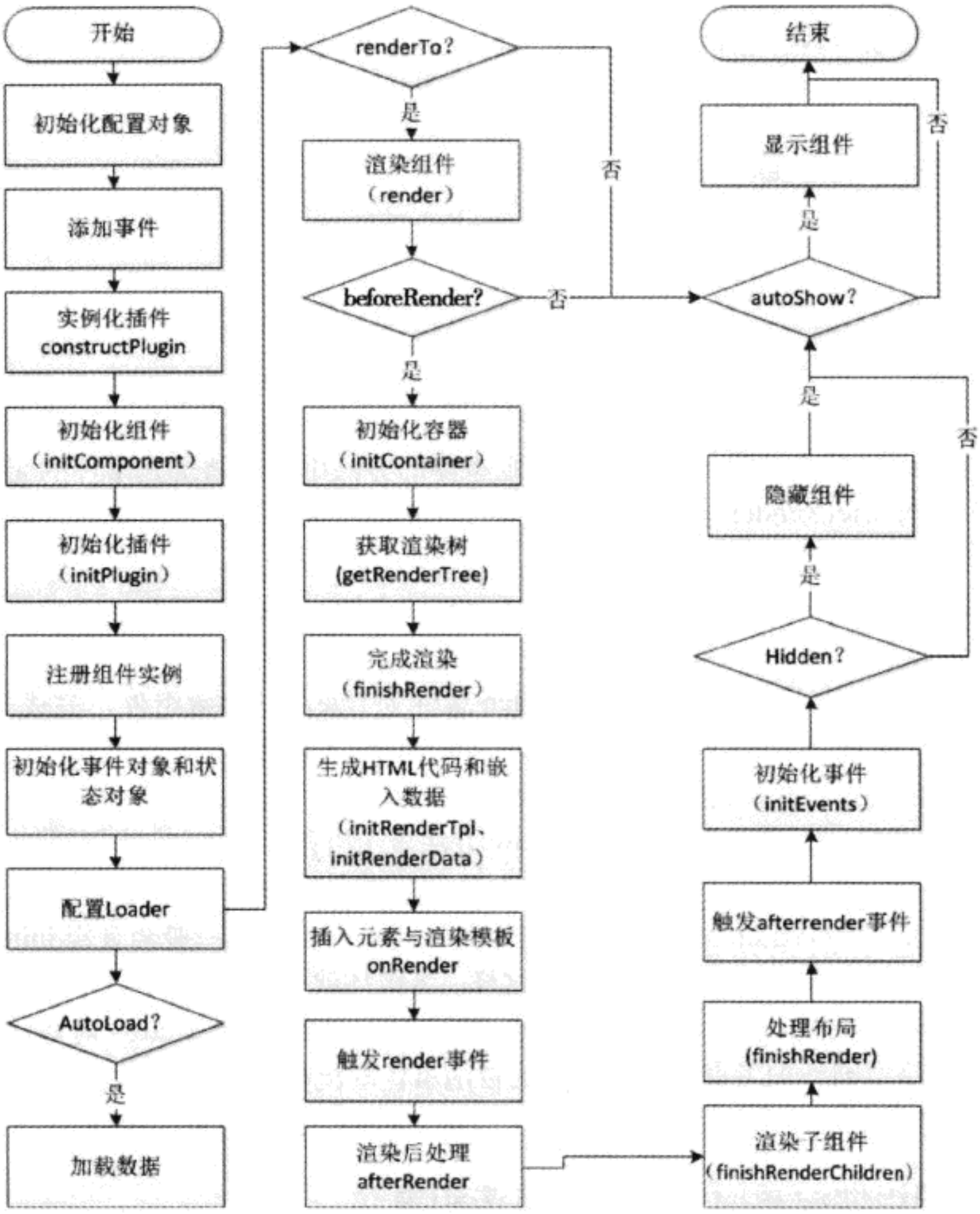


图 8-7　组件的创建流程图

- ❑ data：应用于 tpl 配置项的数据。
- ❑ disabled：布尔值，为 true 时使组件被禁用，默认值为 false。
- ❑ disabledCls：在组件被禁用时使用的样式类。
- ❑ draggable：布尔值，为 true 时允许组件被拖动。
- ❑ floating：布尔值，为 true 时组件会浮动和使用绝对定位，默认值为 false。
- ❑ frame：布尔值，为 true 时组件的边角为圆角。
- ❑ height：数组，设置组件的高度。
- ❑ hidden，布尔值：为 true 时会隐藏组件，默认值为 false。
- ❑ hideMode：组件的隐藏模式，值可以为 display、visibility 和 offsets。值为 display 时

使用 none 值控制组件的显示与隐藏；值为 visibility 时使用 hidden 控制组件的显示与隐藏；值为 offsets 时，会通过绝对坐标将组件放到可视区域之外进行隐藏。默认值为 display。

- html：设置组件的内容，可以为 HTML 代码，也可以是 DomHelper 的配置对象。
- id：设置组件的 id，如果没有配置，会自动生成。
- itemId：一个非常实用的新功能，当一个组件带有子组件时，可以为这些子组件定义该配置项，然后通过父组件的 getComponent 方法，就可返回该子组件。如果是同层的子组件要访问另外的子组件，可使用子组件的 ownerCt 属性获得子组件的容器后，再使用 getComponent 方法获取另外的子组件。
- loader：可以是 ComponentLoader 对象实例或配置对象，用于动态加载内容。可实现动态加载组件。
- margin：设置组件的外边距。
- maxHeight：组件允许的最大高度。
- maxWidth：组件允许的最大宽度。
- minHeight：组件允许的最小高度。
- minWidth：组件允许的最小宽度。
- overCls：鼠标移入组件时会应用该样式。
- padding：设置组件的内边距。
- plugins：可以为对象或数组，配置组件使用的插件。
- renderTo：可以为 DOM 元素、元素的 id，将组件渲染到该元素。
- resizable：为 true 时表示组件可调整大小，该配置项会指向一个 Resizer 对象。
- resizeHandles：当组件允许调整大小时，该配置项用来设置调整大小的句柄，默认值为 all。
- style：应用到组件元素的样式。
- styleHtmlCls：如果 styleHtmlContent 配置为 true，则会将此配置项定义的样式应用到内容，默认值是“x-html”。通过该配置项可以根据自己的需要轻松地为组件的内容设置样式。
- styleHtmlContent：布尔值，为 true 时会为组件的内容设置样式。默认值为 false。
- toBack：将组件放到其他窗口的后面。
- toFront：将组件放到其他窗口的前面。
- toFrontOnShow：布尔值，主要用于浮动组件，默认值为 true，表示当组件显示时会自动将组件显示在最前面。
- tpl：可以为模板的配置对象或模板实例，该模板会应用于组件的内容。
- tplWriteMode：字符串，应用模板到组件的内容时使用的模板的方法，默认值是 overwrite。该值的选项可参考模板的方法。
- width：组件的宽度。

2. 常用的属性

- maskOnDisable：布尔值，为 true 且组件被禁用时，会显示遮蔽。

- ovnerCt：只读属性，会返回组件的容器。
- rendered：只读属性，如果为 true，表示组件已经完成渲染。

3. 常用的方法

- destroy：销毁组件。
- disable：禁用组件。
- enable：启用组件。
- focus：将焦点移动到组件。
- getEl：返回组件的顶层元素。
- hide：隐藏组件。
- setDisabled：禁用或启用组件。
- show：显示组件。
- update：更新内容。

4. 常用的事件

- activate：组件获得焦点时触发该事件。
- added：当容器内添加了一个组件时会触发该事件，对动态加载的组件，该事件相当有用。
- afterrender：当组件渲染完成后会触发该事件。
- beforehde：当组件隐藏前会触发该事件，返回 false 可中止隐藏。
- beforeshow：当组件显示前会触发该事件，返回 false 可中止显示。
- deactivate：当组件失去焦点时会触发该事件。
- disable：当组件被禁用后会触发该事件。
- enable：当组件启用后会触发该事件。
- hide：当组件隐藏后会触发该事件。
- move：当组件移动后会触发该事件。
- remove：当组件从容器中被移除时会触发该事件。
- resize：当组件调整大小后会触发该事件。
- show：当组件显示后会触发该事件。

8.3 为组件添加功能

8.3.1 为元素添加阴影：Ext.Shadow 与 Ext.ShadowPool

Shadow 对象提供以下 3 种阴影方式：

- sides：只两边和底部显示。
- frame：在 4 个边显示。
- drop：拖动时显示。

主要是在组件的底层放置一个阴影层来产生阴影效果。如果是 IE 浏览器或支持 CSS 3 的

浏览器，则比较简单，使用滤镜就可以实现阴影，即放置一个 div 就行了。如果不支持滤镜，则只能通过设置每条边的效果来实现阴影了。

整个工作流程是，Shadow 对象根据阴影类型先计算好阴影元素的位置和大小。在显示时，先使用 ShadowPool 对象的 pull 方法将 HTML 代码插入到页面中，然后将插入的阴影元素移动到目标前面。再根据浏览器是否支持滤镜，来确定是使用滤镜还是调整阴影元素大小和位置来实现阴影效果。

ShadowPool 对象的主要作用是管理阴影，可用 push 方法在 shadows 数组中加入一个阴影元素，或使用 reset 方法清理数组。

Shadow 对象只有 mode 和 offset 两个配置项，mode 可设置阴影的模式，offset 则用于设置阴影的偏移量，默认值是 4。

要使用 Shadow 对象非常简单，打开模板页，先在页面中生成一个长度和宽度都为 100 的方块：

```
var el=Ext.core.DomHelper.append(document.body,{
    tag:"div",
    html:" 阴影测试 ",
    style:"width:100px;height:100px;display:block;"
})
```

然后创建一个 Shadow 对象，并使用 show 方法将其应用于方块：

```
var sd=new Ext.Shadow({mode:"frame"});
sd.show(el)
```

运行后将看到图 8-8 所示的效果。

图 8-8　应用阴影后的效果

8.3.2　为组件提供阴影和 shim 功能：Ext.Layer

做过 Web 开发的人估计都碰到过弹出子菜单时，因为菜单下有输入框或 iframe，导致菜单显示有问题的情况，因而就有了 shim 技术。Layer 对象的作用就是为组件提供阴影功能和 shim 功能。

Layer 对象派生于 Element 对象，因而具有 Element 对象的特性，在内部通过 Shadow 对象提供阴影功能。

这一般很少单独使用，知道功能就行，有兴趣可以自己研究一下。

8.3.3 让组件实现浮动功能：Ext.util.Floating

Floating 对象一般是作为混合对象混合到组件中的，主要是使组件能实现浮动功能。在 Component 对象中混合了该功能，也就是说，在 Component 对象中已经包含了 Floating 对象的方法，其主要方法有 center、toBack 和 toFront。

在 Floating 对象的构造函数中，会创建一个 Layer 对象来实现浮动组件的阴影和 shim 功能，余下就是通过计算调整浮动组件的显示位置，在此就不多说了，有兴趣可以自己研究一下。

8.3.4 记录组件状态：Ext.state.Stateful

Stateful 对象的作用是混合到组件，使组件具有记录其状态的功能。

其整个工作流程是，首先使用 Stateful 对象的 addStateEvents 事件将要记录状态的事件绑定 Stateful 对象的 onStateChange 方法，在该方法内会创建一个 DelayedTask 对象，以延时触发事件。因为改变大小这些操作很频繁，如果不延时执行，会造成组件过于频繁地进行计算，从而影响性能。当用户不再操作后，就执行 saveState 方法在状态管理器（StateManager）记录状态。每个组件实例都有其唯一的状态 id，状态管理器会以状态 id 作为关键字记录组件的状态，而要保存的状态数据则由 getState 方法提供，AbstractComponent 对象的 getState 方法会记录 collapsed、width、height、flex 等信息。

如果要使组件恢复之前的状态，例如窗口最小化后恢复显示，则可使用 applyState 方法恢复状态。

8.3.5 实现调整大小功能：Ext.resizer.Resizer 与 Ext.resizer.ResizeTracker

Resizer 对象的工作原理是，在可改变大小的目标元素内，根据其定义的调整方向，放置一个基本的组件（Component 对象），也就是一个绝对定位的 div 元素，然后使用 ResizeTracker 对象跟踪这些 div 的行为，再不断地调整组件的大小。我们可以通过一个简单的例子来看一下 Resizer 对象是如何工作的。

打开模板页，使用 8.3.1 节的代码生成一个 div，然后执行以下代码为 div 绑定一个 Resizer 对象：

```
Ext.create('Ext.resizer.Resizer', {
        el: el,
        handles: 'all',
        pinned: true
});
```

代码执行后会看到如图 8-9 所示的效果，从图中可以看到，在 div 内添加了 4 条边，对于每条边和 4 个角，鼠标进入时都会将鼠标指针改变成可拖放的指针，这是由 ResizeTracker 实现的。

图 8-9 绑定了 Resizer 对象的 div 效果

从例子可以看到，在 Ext JS 中实现调整大小是相当简单的。

1. 常用属性

- handles：一个由空格分隔的、指示可调整大小方向的字符串值。方向值有 n（北）、

s（南）、e（东）、w（西）、ne（东北）、nw（西北）、se（东南）、sw（西南）和 all（全部方向）等 9 个值。默认值是“s e se”。

- height：设置目标元素的高度。默认值是 null。
- heightIncrement：每次调整高度时的增量，默认值是 0。
- maxHeight：元素可以调整的最大高度，默认值是 10000。
- maxWidth：元素可以调整的最大宽度，默认值是 10000。
- minHeight：元素可以调整的最小高度，默认值是 20。
- minWidth：元素可以调整的最小宽度，默认值是 20。
- pinned ：布尔值，为 true 时会一直显示调整大小的句柄。默认值为 false，只有在鼠标移动到句柄位置时才显示句柄。
- preserveRatio ：布尔值，为 true 时可在调整大小时保持高度和宽度的比率。默认值为 false。
- target：要调整大小的目标元素。
- transparent：布尔值，为 true 时句柄会显示为透明。默认值为 false。
- width：设置目标元素宽度。默认值为 null。
- widthIncrement：每次调整宽度时的增量，默认值是 0。

2. 常用方法

- resizeTo：手动调整大小。

3. 常用事件

- beforesize：在开始调整大小前会触发该事件，如果返回 false，会中止调整大小。
- resize：调整大小后会触发该事件。
- resizedrag：在调整大小期间会触发该事件，如果返回 false，会中止调整大小。

8.3.6　为组件提供拖动功能：Ext.util.ComponentDragger

ComponentDragger 对象派生于 DragTracker 对象，主要作用是使组件实现拖动功能。

要使组件能够拖动，需要定义组件为浮动（配置项 floating 为 true），这样才可以让组件移动。然后定义 draggable 配置项，其值为 ComponentDragger 对象的配置对象，包含以下配置项：

- constrain：布尔值，为 true 时会限制组件的边界在 constrainTo 指定的区域内。
- constrainDelegate：为 true 时，会将 constrainTo 指定的区域作为拖动句柄。
- delegate：委托一个元素为组件的拖动句柄，其值可以为元素的 id 或元素。

打开模板页，在命令行输入以下命令并运行：

```
Ext.create('Ext.Component', {
    floating:true,
    html: '<img src="../images/s1.jpg" style="cursor:move" />',
    draggable: {
      constrain: true,
```

```
            constrainDelegate: true
        }
    }).show();
```

因为是浮动组件，所以需要使用 show 方法显示。运行后，在页面中会看到如图 8-10 所示的效果，按下鼠标左键就可以拖动组件了。在按下鼠标左键后才将鼠标指针显示为移动指针比较适合，有兴趣可以自己研究一下。

图 8-10　组件拖动示例的效果

8.3.7　为组件实现动画功能：Ext.util.Animate

Animate 对象是一个混合对象，可混合到 Element 对象、CompositeElement 对象、DrawCompositeSprite 对象、CompositeSprite 对象和组件等中，其主要作用是为这些对象提供简单的动画功能。

因为 Animate 对象已经混合到组件或 Element 对象中，因而，直接调用组件或 Element 对象的 animate 方法就可实现动画。animate 方法是使用配置对象来设置其效果的，其主要配置项有以下几个：

- from：一个包含 x、y 坐标的对象，表示动画目标的初始位置。
- to：为对象，表示动画的结束位置，其对象的配置一般有 x、y、left、top、width 和 height 等 6 个配置项。对于 Element 类对象，还可以定义 opacity、color、scrollLeft 和 scrollTop 等配置项。对于 Sprite 类对象，可定义 translation、path、scale、stroke 和 rotation 等配置项。对于组件，可定义 dynamic 配置项，其值为 true 时会更新组件的布局。
- duration：动画运行的持续时间，单位是微秒，默认值是 250。
- easing：动画类型，值可以为 ease、easeIn、easeOut、easeInOut、backIn、backOut、eleasticIn、elasticOut、bounceIn 和 bounceOut。
- keyframes：关键帧，在持续时间内，可以通过该配置项将其分段，然后在不同的段内实现不同的效果。
- listeners：定义事件，事件有 beforeanimate 和 afteranimate 两个，beforeanimate 在动画开始前会触发，afteranimate 在动画结束后触发。

要测试动画功能很简单，打开模板页，然后在命令行中创建一个面板：

```
var panel = new Ext.create("Ext.panel.Panel",{
    title:"动画测试",
    html:"动画测试",
    renderTo:Ext.getBody(),
    width:300,
    height:200
})
```

然后就可以使用 animate 测试各种参数了，例如输入以下代码：

```
panel.animate({
    to:{x:200,y:300}
})
```

面板将移动到坐标为（200，300）的位置。

Animate 对象还提供了以下几个方法：

- getActiveAnimation：返回当前动画，如果对象没执行过任何动画，返回 false。
- sequenceFx：确保队列中的动画是依次序进行的。
- stopAnimation：停止动画，并清除队列。
- syncFx：确保队列中的动画是同时进行的。

8.3.8　其他的组件辅助功能类

在 4.1 Beta 1 版本中，为了实现批处理渲染，需要对子组件进行管理，因而添加了以下几个对象：

- Ext.util.Renderable：为组件提供渲染功能。
- Ext.util.ElementContainer：用于处理子元素之间的关系。
- Ext.util.ProtoElement：用于管理元素的渲染前的数据，主要用于管理组件的样式。

以上这些辅助功能类是私有类，只在 Ext JS 框架内部使用。

8.4　组件的管理

8.4.1　组件管理及查询：Ext.ComponentManager 与 Ext.ComponentQuery

ComponentManager 对象是从 AbstractManager 继承的，相关信息可阅读 7.4.7 节的内容。

ComponentManager 对象将 AbstractManager 对象的 typeName 属性的 type 修改为 xtype，并重写了 create 方法和 registerType 方法。

ComponentQuery 对象提供了一种类似 DomQuery 对象的使用选择符查询组件的方法，DomQuery 对象可使用 DOM 元素的标记作为选择符，而 ComponentQuery 对象则使用组件的 xtype 值作为选择符，因而要搜索页面中的所有面板，可使用“panel”作为选择符进行搜索。当然，与 DomQuery 对象一样，也可以使用 id、在选择符内加入属性或使用伪类选择符来进行搜索。

ComponentQuery 对象的实现与 DomQuery 对象非常类似，只是它搜索的对象是 ComponentManager 对象里的实例，而不是 DOM 树。

在 Ext JS 4 之前的版本，只能利用 Ext.getCmp 从 ComponentManager 对象里通过 id 返回对象，限制比较大，而且组件必须自定义 id，才能有目的地去取组件，有了 ComponentQuery 对象就方便多了，例如，要找出 FormPanel 里的 TextField，可以使用以下语句：

```
Ext.ComponentQuery.query("form > textfield")
```

这样就可以使用循环或枚举等操作对一组控件进行操作了，而不需要使用 id 一个个地去找这些控件。

ComponentQuery 对象提供了以下两个方法：

- is：检查组件是否匹配选择符，如果匹配返回 true，否则返回 false。
- query：使用选择符查询组件。

8.4.2 焦点管理：Ext.FocusManager

FocusManager 对象可为组件提供键盘导航功能，还可根据设置选择是否使用可视化手段提示当前焦点所在组件。

要开启组件焦点管理，可执行以下代码：

```
Ext.FocusManager.enable(options);
```

如果 options 为 true，则使用可视化手段显示当前焦点所在组件，否则不显示。

如果要关闭管理，调用 disable 方法即可。

打开 7.2.4 节的示例，然后在命令中运行开启焦点管理代码，用鼠标点第一个 Grid 的标题，并按下两次 Tab 键，将看到如图 8-11 所示的效果。

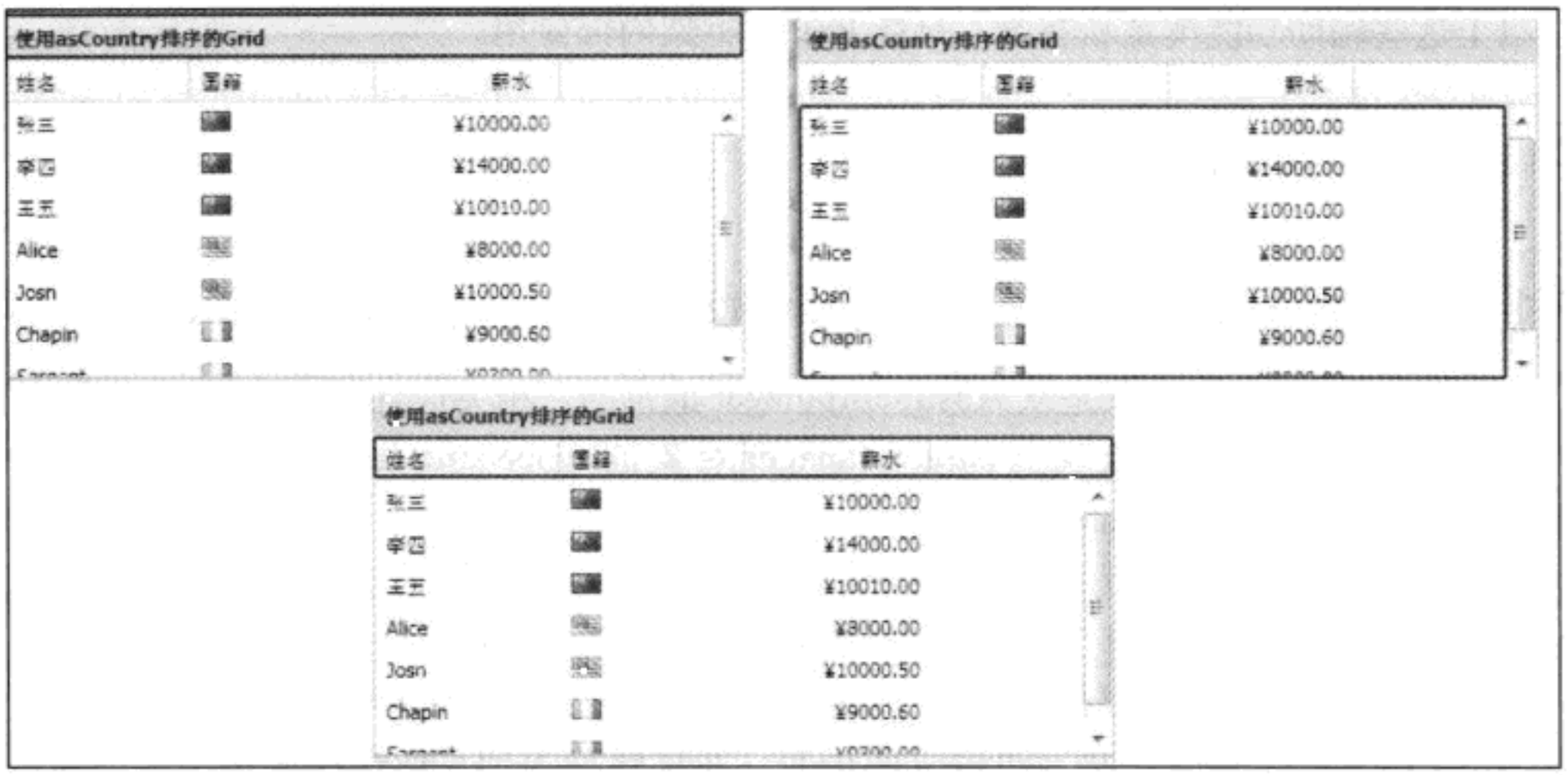


图 8-11　焦点管理的示例效果

焦点管理的最大问题是不能跨越两个不在同一容器内的组件，例如示例中，要切换到第二个 Grid，必须使用鼠标。不过，这个功能在表单中使用时已经足够强大了。

1. 属性

- enabled：为 true 时开启焦点管理，为 false 时关闭焦点管理。
- focusedCmp：当前活动焦点的组件，默认值为 undefined。
- whitelist：一个白名单，可以让名单里的组件在键盘键入 Backspace、Delete 和 4 个箭头键等 6 个键的时候不是执行导航操作，而是执行浏览器事件。白名单里默认值为“textfield”。

2. 方法

- addXTypeToWhitelist：添加 xtype 值到白名单。
- disable：关闭焦点管理。
- enable：开启焦点管理。可以传递参数 true 开启可视化焦点。
- removeXTypeFromWhitelist：将组件的 xtype 值移出白名单。
- subscribe：设置一个容器内的子组件可以使用键盘导航。
- unsubscribe：取消容器内的子组件使用键盘导航的功能。

3. 事件

- beforecomponentfocus：在组件获得焦点前触发该事件，如果返回 false，可不让组件获得焦点。
- componentfocus：组件获得焦点后触发该事件。
- disable：关闭焦点管理后会触发该事件。
- enable：开启焦点管理后会触发该事件。

8.4.3　z-order 管理：Ext.ZindexManager 与 Ext.WindowManager

ZindexManager 对象是用来管理浮动组件的 z-order、管理组件的激活行为以及在激活组件前遮蔽组件操作的。其工作原理是在容器创建时，如果容器带浮动组件，就会在容器的 ZindexManager 实例内注册，ZindexManager 会用一个对象来记录这些对象，用一个数组来记录这些对象的显示顺序并控制其 z-index 属性，当一个活动的浮动组件隐藏后，就会从数组中找到最后一个组件，并激活它。如果某个对象变成最前面显示的组件，就会从数组中先移除，然后再追加到数组，这样就保持了数组的显示顺序。

要注意的一点是，容器内的 ZindexManager 实例只能管理容器内部的浮动组件，而窗口等组件是管理不了的。要管理窗口等组件，必须使用 WindowManager，它是 ZindexManager 对象的一个实例，实际上它是把页面的 body 作为其容器，然后实行管理的。

ZindexManager 对象的方法如下：

- bringToFront：将指定对象显示在最前面。
- each：枚举 ZindexManager 对象内的浮动组件。
- eachBottomUp：由底往上枚举 ZindexManager 对象内的浮动组件。
- eachTopDown：由顶往下枚举 ZindexManager 对象内的浮动组件。
- get：返回指定 id 的浮动组件。
- getActive：返回当前活动的浮动组件。
- getBy：通过指定函数返回浮动组件。
- hideAll：隐藏所有组件。
- register：注册一个浮动组件。
- sendToBack：将指定组件放到其他活动组件后面。
- unregister：对指定的组件取消注册。

8.4.4 状态管理：Ext.state.Manager、Ext.state.Provider、Ext.state.LocalStorageProvider 和 Ext.state.CookieProvider

StateManager 对象会使用状态提供者记录组件的状态，默认是使用 StateProvider 对象，也可以使用 LocalStorageProvider 和 CookieProvider。使用 set 方法，可在状态提供者内添加一个状态信息；使用 get 方法，可返回指定的状态 id 的状态信息。使用 clear 方法可清除状态提供者内的全部状态信息。使用 setProvider 方法可改变状态管理器的状态提供者，而 getProvider 可返回当前的状态提供者。

StateProvider 对象会使用 JavaScript 对象作为存储状态信息的数据结构，使用 set 方法可在对象内添加一个状态信息，其属性名称为状态 id，使用 get 方法可返回状态信息。使用 clear 方法可清除所有状态信息。

StateProvider 对象使用的是内存对象，所以在浏览器关闭后，所有状态信息都会丢失，如果要实现与 Windows 应用一样的功能，可以在下次打开同一组件时恢复最后一次的状态，如窗口的位置和大小，这样就需要将状态信息存储在 Cookie 或 HTML 5 支持的 LocalStorage 里，因而，从 StateProvider 对象派生了 CookieProvider 对象和 LocalStorageProvider 对象。这两个状态提供者都会在初始化时从 Cookie 或 LocalStorage 中将状态恢复到内存的 JavaScript 对象，然后在 initState 方法中根据组件的状态 id 就可以将组件的状态恢复到最后一次使用的情况了。为了将状态保存到 Cookie 或 LocalStorage，需要重写 set 方法，而且因为数据格式不同，还需要 decodeValue 和 encodeValue 将状态信息处理后再保存。

如果要使用 CookieProvider 对象或 LocalStorageProvider 对象，必须在创建组件之前就设置好提供者，在 OnReady 函数内创建组件之前执行以下代码即可：

```
Ext.state.Manager.setProvider(Ext.create('Ext.state.CookieProvider));
```

或

```
Ext.state.Manager.setProvider(Ext.create('Ext.state.LocalStorageProvider));
```

一劳永逸的方法是在本地化文件内，先判断 Ext.state.Manager 是否存在，如果存在则设置状态提供者，这样，无论在任何页面，只要加载了本地化文件，就已经定义好了状态提供者了。

8.5 综合实例

8.5.1 使用子模板

（1）功能描述

本实例主要演示如何使用子模板。

（2）实现代码

例如，有以下数据：

```
var data={
    name:"Ext JS 4 权威指南 ",
    src:'1.jpg',
    price:99,
    discount:.6,
    stock:10,
    author:" 黄灯桥 ",
    comments:[
        {title:" 好书 ",user:" 张三 ",content:" 书不错 ",posttime:"2011-5-10",star:4},
        {title:" 推荐 ",user:" 李四 ",content:" 很不错的书 ",posttime:"2011-6-10",star:5}
    ]
}
```

在这里最难处理的是将 star 的值转换为由 5 个星组成的图形，通常的做法是使用成员函数在函数内组合字符串返回，不过，使用子模板的方法会更简单、更直观，首先是将数值转换为一个数组，然后由子模板根据数组的元素值输出对应的图片文件名，代码如下：

```
var subtpl=new Ext.XTemplate(
    '<tpl for=".">',
    '<img src="../images/{[values?"star-yellow.gif":"star-gray.gif"]}" />',
    '</tpl>'
);
```

上述代码中，只要值为 true，就显示“star-yellow.gif”，否则显示“star-gray.gif”。

目标明确，可以开工了，使用模板页创建一个名称为 8-1.html 的页面文件，先把以上代码加入 OnReady 函数，然后再加入以下代码：

```
var tpl = new Ext.XTemplate(
    '<h2>{name}',
    '<tpl if="discount <= .6">',
    ' (<font style="color:red"> 大优惠 </font>) ',
    '</tpl>',
    '</h2>',
    '<img src="{img}" alt="{name}" />',
    ' 单价：{price:currency}<br/>',
    ' 折扣：{discount*100} 折 <br/>',
    ' 折扣价：{[this.discountprice(values)]}<br/>',
    ' 用户评论：<br/>',
    '<tpl for="comments">',
        '<p><h3>{title}</h3>',
        ' 用户：{user}',
        ' 发表时间：{posttime}',
        '{star:this.starimg}',
        '<p>{content}</p></p>',
    '</tpl>',
    {
        starimg:function(v){
            var vv=[v>=1,v>=2,v>=3,v>=4,v==5];
            return subtpl.apply(vv);
        },
        discountprice:function(v){
            return Ext.util.Format.currency(v.price*v.discount);
        }
```

```
        }
    );

    tpl.overwrite(Ext.getBody(),data);
```

模板代码中，当折扣（discount）小于 6 折的时候会显示“大优惠”；单价则使用货币格式；折扣因为是小数，所以需要使用简单运算将其转换为整数；折扣价因为要进行运算，还要进行格式转换，所以需要使用成员函数 discountprice 来处理；粗体代码在成员函数 starimg 中会先将数值转换为一个数组，然后调用子模板的 apply 方法返回生成的代码。

（3）页面效果

在浏览器中打开页面，将看到如图 8-12 所示的效果。

图 8-12　子模板示例的效果

8.5.2　递归调用模板

（1）功能描述

本实例主要演示如何使用模板的递归功能。

（2）实现代码

在网站中，有时候需要用层次结构显示产品的分类，例如以下数据结构：

```
var data=[
{
        name:" 电脑 ",
        items:[
                {name:" 手提电脑 ",items:[
                        {name:" 联想 "},
                        {name:" 戴尔 "},
                        {name:" 惠普 "}
                ]},
                {name:" 台式机 ",items:[
                        {name:" 联想 "},
                        {name:" 戴尔 "},
                        {name:" 惠普 "}
                ]}
        ]
},
{
        name:" 手机 ",
        items:[
                {name:" 手机 ",items:[
                        {name:" 诺基亚 "},
                        {name:"HTC"},
                        {name:" 摩托罗拉 "}
                ]},
                {name:" 配件 "}
        ]
    },
    {name:" 其他 "}
]
```

因为不知道数据里的 items 会有多少层，所以在模板中使用固定层级直接输出数据的方式不大可取。最好的方式就是使用递归，每当数据带 items 时，就递归调用一次模板，代码如下：

```
var tpl=new Ext.XTemplate(
    '<tpl for=".">',
    '<div class="list">{name}</div>',
    '<tpl if="items">',
    '<div class="listitem">{[this.listItems(values)]}</div>',
    '</tpl>',
    '</tpl>',
    {
            listItems:function(v){
                    return this.apply(v.items)
            }
    }
);
```

模板中，使用 if 语句判断 items 是否存在，如果存在，则执行 listItems 方法，递归调用模板。

现在的输出还看不出层次结构，这需要使用样式做调整，定义 listitem 的样式如下：

```
.listitem{padding-left:20px;}
```

因为每一层都是在上一层的 div 里的，所以会自动偏移 20 个像素，这样层次结构就出来了。

使用模板页，创建一个名称为 8-2.html 的文件，然后将以上代码加入文件，就完成示例了。

（3）页面效果

在浏览器中打开页面，将看到如图 8-13 所示的效果。

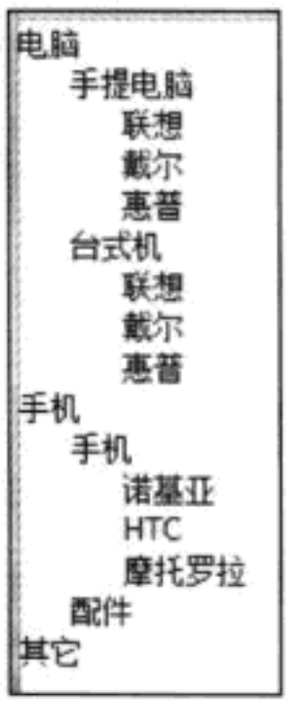


图 8-13　递归调用模板的页面显示效果

8.6　本章小结

组件是 Ext JS 的基础，模板是 Ext JS 4 的基础，熟悉模板，也就了解了组件的组织方式，也为使用组件和构建自定义组件打下了基础。Ext JS 的组件很多，但最基本的是容器、面板、布局和视图这 4 类，而且在使用中也最容易混淆其作用，所以需要特别注意其区别和作用。在下一章我们将讨论这 4 类基本组件。

通过功能类可以为组件添加特别的功能，这也是组件的重要一环，需要了解。

开发应用就要用到组件，就要对组件进行管理，Ext JS 4 新增的 ComponentQuery 对象是组件管理的好助手，需要熟悉和掌握其使用方法。状态管理可以让用户感受 Windows 应用一样的体验。在每天开始工作时，打开 Web 应用，可维持昨天的状态，感觉一定会相当亲切，所以，应该好好花点时间在状态管理上，以增加用户体验。

第9章　容器、面板、布局和视图

容器、面板、布局和视图是 Ext JS 的最基本组件，它们之间的关系较难区分，很容易造成不必要的混乱。主要原因是创建组件时，布局只是一个简单的配置项，并不像容器或其他组件一样进行配置和定义。本章将详细讲述这 4 类组件，并通过丰富的例子让大家熟悉它们的使用方法。

9.1　容器与布局的关系

容器的主要作用是管理其内部的组件，通过容器的方法，可在容器内添加、删除、插入组件。最简单的容器就是一个 HTML 元素，例如一个 div、一个 span 等，可通过组件的 autoEl 配置项进行配置。不过在开发中这并不常用，常用的是 Panel、Window 和 TabPanel 等特殊的容器。

而布局决定着容器的组件以怎样的形式渲染，以及布局之间是否可以调整大小或位置。例如，垂直布局可垂直划分容器的区域，容器内的组件会依次序从上往下排列在容器内；而边框布局则会提供 5 个固定区域（除了 center 区域，其他可选），子组件可根据配置放置在指定区域内，各区域之间可调整大小。因此，容器内无论只有一个组件还是有多个组件，都必须配置布局，以决定这些组件是如何放置的。

因而，可以说容器和布局是共生关系。在容器中都会有 layout 配置项，该配置项决定了容器将使用何种布局方式布置子组件。但这里有一个问题要解决，即子组件的大小和位置如何确定。这与布局的特性有关，有些布局的大小和位置可由组件的大小和位置配置项来决定；而有些则需要在子组件中应用布局的配置项来决定组件的大小和位置。这样会造成相当程度的混乱，尤其是在子组件也是一个容器的情况下，这时候既有子组件的布局配置项，又有其容器的布局配置项。不过，只要记住以下 3 点就不容易搞混了：

- 布局定义必然是在容器内的。
- 子组件的布局定义只应用于其内部的组件，与子组件本身及其容器的布局无关。简单地说，当前组件配置的 layout 配置项只应用于 items 里定义的组件，与当前组件及其父组件的布局无关。
- 一定要熟悉各布局的特性，如果布局不能由子组件的大小和位置决定其大小和位置，那么它必然由应用于子组件的配置项来决定子组件的大小或位置。

如果容器内部没有默认的布局，在创建时也没有为其配置 layout 配置项，那么它会使用自动布局作为其布局，也就是说，子组件只会机械地追加到容器内，其初始大小和位置就是子组件的原始大小及原始位置，而且不能改变大小及移动。因而，最好在使用容器类的时候，

搞清楚其默认布局，如果是使用自动布局，一定要配置 layout 配置项，以避免出现非预期的结果。

9.2　容器

9.2.1　容器的创建过程：Ext.container.AbstractContainer 与 Ext.container.Container

AbstractContainer 对象是容器类的抽象基类，封装了容器类的操作及其子组件的方法。Container 派生于 AbstractContainer 对象，作为容器里的基类，它只在 AbstractContainer 对象的基础上添加了 getChildByElement、afterHide 和 afterShow 方法。因而基本容器类是在 Abstract-Container 对象里创建。

在 AbstractContainer 对象中，只重写了 initComponent 方法，没有构造函数，因而其创建会使用 Component 对象的构造函数。

根据 8.2.4 节组件的创建流程，AbstractComponent 对象的构造函数在初始化配置对象、添加事件、实例化插件后，才会调用 AbstractContainer 对象的 initComponent 方法，其代码如下：

```
initComponent : function(){
    var me = this;
    me.addEvents(
            // 省略事件代码
    );
    me.callParent();
   me.getLayout();
    me.initItems();
},
```

代码很简单，先是添加与容器相关的事件。接着调用 Component 对象的 initComponent 方法。然后调用 getLayout 方法创建布局。最后执行 initItems 方法，初始化容器 items 配置项定义的子组件，其代码如下：

```
initItems : function() {
    var me = this,
        items = me.items;
    me.items = Ext.create('Ext.util.MixedCollection', false, me.getComponentId);
    if (items) {
        if (!Ext.isArray(items)) {
            items = [items];
        }

        me.add(items);
    }
},
```

变量 item 会指向 items 配置项的值，然后将配置项 items 指向一个 MixedCollection 对象实例。如果 items 不是数组，则将其转换为数组，再调用 add 方法，其代码如下：

```
add : function() {
```

```
var me = this,
    args = Ext.Array.slice(arguments),
    index = (typeof args[0] == 'number') ? args.shift() : -1,
    layout = me.getLayout(),
    addingArray, items, i, length, item, pos, ret;

if (args.length == 1 && Ext.isArray(args[0])) {
    items = args[0];
    addingArray = true;
} else {
    items = args;
}

ret = items = me.prepareItems(items, true);
length = items.length;

if (me.rendered) {
    me.suspendLayouts();
}

if (!addingArray && length == 1) {
    ret = items[0];
}

for (i = 0; i < length; i++) {
    item = items[i];
    //省略调试代码

    pos = (index < 0) ? me.items.length : (index + i);

    if (item.floating) {
        item.onAdded(me, pos);
    } else if (me.fireEvent('beforeadd', me, item, pos) !== false &&
                me.onBeforeAdd(item) !== false) {
        me.items.insert(pos, item);
        item.onAdded(me, pos);
        me.onAdd(item, pos);
        layout.onAdd(item, pos);

        me.fireEvent('add', me, item, pos);
    }
}

if (me.rendered) {
    me.resumeLayouts(true);
}

    return ret;
},
```

上述代码先调整好传递参数，将变量 items 指向子组件数组。然后调用 prepareItems 方法预处理子组件，其代码如下：

```
prepareItems : function(items, applyDefaults) {
```

```
        if (Ext.isArray(items)) {
            items = items.slice();
        } else {
            items = [items];
        }

        var i = 0,
            len = items.length,
            item;

        for (; i < len; i++) {
            item = items[i];
            if (applyDefaults) {
                item = this.applyDefaults(item);
            }
            items[i] = this.lookupComponent(item);
        }
        return items;
    },
```

上述代码先将 items 转换为数组。接着在循环中逐个对组件进行预处理。预处理包括两个处理。第 1 个是 applyDefaults 方法，用于将配置项 defaults 的成员复制到组件。通过 defaults 配置项可以为组件配置一些默认值。第 2 个是 lookupComponent 方法，如果给的参数是字符串，它根据指定的字符串会从 ComponentManager 对象中寻找组件实例。如果参数不是字符串，则使用 ComponentManager 对象的 create 方法创建一个组件实例。也就是说，prepareItems 会将组件实例化。

回到 add 方法，如果容器渲染已完成（rendered 为 true），则要暂停布局运算，以便添加子组件。如果子组件只有一个，设置返回值 ret 为该子组件。

接着使用循环通过 onAdded 方法添加组件。如果是浮动组件（floating 为 true），则不需要加入子组件集合，直接调用 onAdded 方法就行了。在 4.1 beta 1 版本中，只有 AbstractComponent 和 ComboBox 对象有具体定义，我们来看看 AbstractComponent 对象中的定义，代码如下：

```
onAdded : function(container, pos) {
    this.ownerCt = container;
    this.fireEvent('added', this, container, pos);
},
```

上述代码的主要工作是将组件的 ownerCt 属性指向容器，也就是把组件的拥有者设置为容器，接着触发 added 事件。

如果是非浮动组件，则需要调用 insert 方法在集合中插入组件。接着要调用的 onAdd 方法是给工具栏、视图等做些特殊处理的，这里就不展开了，有兴趣自己研究一下。最后触发 add 事件。如果容器已经渲染，要调用 resumeLayouts 方法恢复布局运算。

至此，容器的创建过程就完成了。

9.2.2 Ext.container.AbstractContainer 和 Ext.container.Container 的配置项、属性、方法和事件

1. 配置项

- activeItem：可以为子组件的 id 或索引，当容器渲染后会将指定的子组件设置为活动。
- autoDestroy：布尔值，设置为 true 时会自动销毁容器，默认值为 true。
- bubbleEvents：一个事件数组，当数组内的事件被触发时，会传递到其父容器，默认值为“['add','remove']”。
- defaultType：设置容器的子组件的默认类型，也就是定义子组件时如果没有设置 xtype 配置项，则使用该配置项设置的值作为子组件的 xtype 值。默认值是 panel。
- defaults：可以为对象或函数，当子组件创建时，该配置项的值会作为子组件的默认值应用到子组件中。
- items：可以为对象或数组，用于定义容器的子组件。
- layout：可以为字符串或对象，设置容器使用的布局。当值为字符串时，为布局的名称。值为对象时，为布局的配置对象。
- suspendLayout：布尔值，如果为 true，suspendLayout 会暂停 doLayout 的调用，通常用于批量添加组件。其默认值为 false。

2. 属性

- items：容器的子组件，为 MixedCollection 对象实例。

3. 方法

- add：在容器中添加组件，值可为组件的配置对象或由配置对象组成的数组。
- cascade：以层叠方式显示容器内的组件。
- child：返回匹配选择符的第一个直接子组件。该方法会使用 ComponentQuery 查找组件。
- doLayout：强制容器重新计算布局。
- down：返回匹配选择符的第一个子组件。
- getComponent：从 items 属性中返回由 id 或索引指定的组件。
- getLayout：返回容器的布局实例。
- insert：在指定位置插入组件。
- move：将组件移动到指定位置。
- query：通过选择符查找组件。
- remove：移除组件。
- removeAll：移除容器内的所有组件。

4. 事件

- add：当组件添加到容器后触发该事件。
- afterlayout：当容器计算布局后触发该事件。

- beforeadd：在组件添加到容器前触发该事件，如果返回 false，会取消添加操作。
- beforecardswitch：当容器使用 CardLayout 作为布局时，在 card 进行切换前会触发该事件，返回 false 可取消切换。
- beforeremove：当组件被移除前会触发该事件，返回 false 可取消移除操作。
- cardswitch：当容器使用 CardLayout 作为布局时，在 card 进行切换后会触发该事件。
- remove：当组件被移除后会触发该事件。

9.2.3　将 body 元素作为容器：Ext.container.Viewport

Viewport 是使用 Ext JS 开发应用时常用的对象，它会将 HTML 的 body 元素作为一个容器，然后通过布局设计出应用的整体 UI。

Viewport 重定义了 initComponent 方法，其代码如下：

```
initComponent : function() {
    var me = this,
        html = Ext.fly(document.body.parentNode),
        el;
    Ext.getScrollbarSize();
    me.callParent(arguments);
    html.addCls(Ext.baseCSSPrefix + 'viewport');
    if (me.autoScroll) {
        delete me.autoScroll;
        html.setStyle('overflow', 'auto');
    }
    me.el = el = Ext.getBody();
    el.setHeight = Ext.emptyFn;
    el.setWidth = Ext.emptyFn;
    el.setSize = Ext.emptyFn;
    el.dom.scroll = 'no';
    me.allowDomMove = false;
    me.renderTo = me.el;
},
```

上述代码先调用 getScrollbarSize 方法获取滚动条尺寸，然后调用 callParent 方法，初始化容器内的组件并计算好 body 的布局大小。

接着为页面的 html 元素添加样式类“x-viewport”，如果设置 autoScroll 为 true，则添加样式“overflow:auto”。

接下来是关键一步，使用 Ext.getBody 方法把 body 元素作为 el，也就是说，Viewport 的容器元素就是 body。很多对 Viewport 对象不熟悉的初学者在定义 Viewport 对象时，以为它和其他组件一样，可以使用 renderTo 配置项将其渲染到某个节点。这是错误的做法，因为 Viewport 对象的容器是 body，不可能渲染到其他节点下。切记！切记！当然，定义 renderTo 配置项的值为 Ext.getBody 是可以的。

接着将 setHeight、setWidth 和 setSize 这 3 个方法设置为空函数，避免调用这 3 个方法来设置 body 的大小，从而造成错误。

接着将页面的滚动条隐藏，设置 allowDomMove 为 false，表示该节点不能移动。

接着将 renderTo 属性指向自己，这样就可以进行渲染处理了。

Viewport 对象派生于 Container 对象，所以它也具有 Container 对象的配置项、属性、方法和事件。

9.3 面板

9.3.1 面板的结构

面板是有特殊结构的容器，因而它既有容器的特性，也有自己特有的特性。面板可以说是开发 Web 应用常用的组件，因而了解其结构是非常有必要的。

打开模板页，在命令行中运行以下代码：

```
Ext.create('Ext.panel.Panel', {
    renderTo:Ext.getBody(),
    title:"面板",
    tbar: [{ xtype: 'button', text: '上' }],
    bbar: [{ xtype: 'button', text: '下' }],
    lbar: [{ xtype: 'button', text: '左' }],
    rbar: [{ xtype: 'button', text: '右' }],
    buttons:[{ xtype: 'button', text: '确定' }],
    tools:[{type:"refresh"},{type:"help"},{type:"save"},{type:"search"}],
    html:"body",
    width:500,
    height:300
});
```

将会看到如图 9-1 所示的面板结构图。图中，面板整体被分成了 7 个部分。顶部是标题栏，标题栏里还包含一个工具区，这是 Header 对象的功能。在面板上有 5 个工具栏，其中分别包含上、下、左、右按钮的是通用工具栏，而包含确定按钮的是修改了样式的工具栏。工具栏包围的区域是放置内容的区域，可以在这里添加组件或者 HTML 内容。

图 9-1 面板结构图

以上构件的实现依赖于 DockLayout 布局，它会计算出构件的绝对定位的坐标，然后将样

式应用于构件的 HTML 元素。

当然，以上构件除了主体部分外都不是必需的，如果不定义这些构件的配置项，那么它就不会创建。因而你可以自由选择需要的构件，如果不需要，可以把 Panel 对象当成纯粹的容器使用。以下是这些构件对应的配置项：

- bbar：底部工具栏。
- buttonAlign：设置 fbar 的按钮对齐方式，值可以为 right、left 和 center，默认值为 right。
- buttons 或 fbar：页脚工具栏。
- closable：为 true 时，会在标题栏的右边显示关闭图标。
- collapsible：为 true 时，会在标题栏的右边显示折叠（展开）图标。
- headerPosition：设置标题栏的位置，值可以为 top、bottom、left 和 right。默认值是 top。这是新的功能，通过新的画图功能解决了文字的竖向显示的问题。
- hideCollapseTool：为 true 时隐藏折叠（展开）图标。
- lbar：左边工具栏。
- minButtonWidth：设置页脚工具栏按钮的宽度，默认值为 75。
- preventHeader：阻止标题栏的创建和显示，与不定义 title 效果一样。默认值为 false。
- rbar：右边工具栏。
- tbar：顶部工具栏。
- title：标题栏的标题。
- titleCollapse：如果为 true，单击标题栏就折叠（展开）面板，否则，需要单击右边的图标按钮才能折叠（展开）面板。
- tool：设置标题栏右边显示的工具按钮图标。

图 9-1 中，面板主体部分因没有配置 frame 为 true，背景色是白色的，在有工具栏时，不太美观。但是如果设置了 frame，又会因圆角部分占位而出现如图 9-2 所示的情况，工具栏两边有明显的空位，这是为了显示圆角效果添加的边条，效果也不是很好。因而，最好的办法是使用 bodyStyle 配置项，设置面板主体的背景颜色为“#DFE9F6”，例如在命令行中输入以下代码并执行：

```
Ext.create('Ext.panel.Panel', {
    renderTo:Ext.getBody(),
    title:"面板",
    tbar: [{ xtype: 'button', text: '上' }],
    html:"body",
    width:200,
    height:100,
    bodyStyle:"background:#DFE9F6"
});
```

运行后可看到如图 9-3 所示的效果，这样既可避免 frame 生成的多余的 HTML 代码以及对工具栏和主体的影响，又实现了背景一致的效果。不过要注意，如果使用不同的主题，则需要根据主题修改背景色，只要在主题的 CSS 文件中寻找“x-panel-body-default-framed”的

背景颜色设置，复制过来就行了。

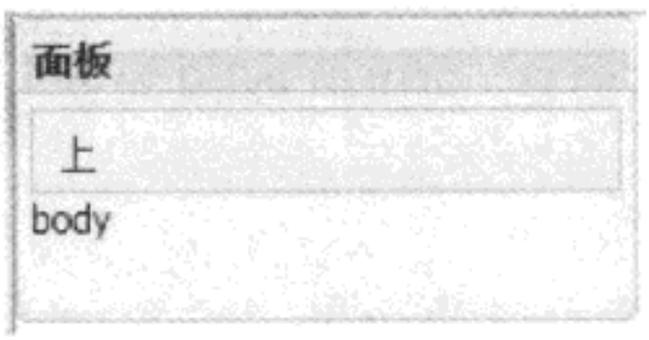


图 9-2　面板设置 frame 为 true 的效果

图 9-3　面板设置背景颜色后的效果

9.3.2　构件的放置：dockedItems

虽然面板预设了标题栏、工具栏等构件，但可根据需要添加自己的构件。例如，打开模板，在命令行添加并执行以下代码：

```
Ext.create("Ext.panel.Panel",{
    width:160,
    height:300,
    renderTo:Ext.getBody(),
    dockedItems:[
          {xtype:"image",dock:'top',src:"../images/logo-screen-ie.png"},
          {xtype:"image",dock:'bottom',src:"../images/logo-screen-ie.png"}
    ]
})
```

上述代码运行后将看到如图 9-4 所示的效果，在面板顶部和底部都添加了一幅图片。

图 9-4　使用 docked-Items 配置项添加配置的页面效果

之所以能实现该功能，主要是因为面板使用了 DockLayout 作为其构件的布局。该布局可轻松地将构件停靠在面板的 4 个边上。在面板的配置对象里，使用 dockedItems 配置项就可轻松定义停靠的组件。如果只是停靠单个组件，可直接将组件的配置对象作为配置项 dockedItems 的值；如果有多个停靠对象，则需要使用数组将停靠的配置对象组合起来，例如图 9-4 的示例。示例中定义了两个图片组件，在它们的配置对象中都有一个 dock 的配置项，用来设置组件停靠在面板的位置，其值可以为 top、left、right 或 bottom，分别表示面板的顶部、左边、右边和底部。示例中的第一个图片停靠在面板顶部，而第二个图片停靠在面板的底部，也就是图 9-4 所看到的效果。

只要是面板或从面板类基础派生的组件都可以使用 dockedItems 来定义停靠组件，灵活运用该配置项，可让应用变得更美观。

9.3.3　面板标题栏构件：Ext.panel.Header 与 Ext.panel.Tool

Header 对象派生于 Container 对象，也是一个容器，在容器内会根据 titile、iconCls 和 tools 这 3 个配置项添加对应的组件。配置项 iconCls 会为标题栏添加一个图标，并创建一个 Component 对象实例。配置项 title 会为标题栏添加标题，如果标题栏是横向的，则会创建一

个 Component 对象实例；而如果是竖向的，则创建一个 DrawComponent 对象实例，通过图形引擎将标题画出来。配置项 tools 会创建一个 Tool 对象的实例，用于显示图标按钮。

Tool 对象派生于 Component 对象，主要作用是为标题栏提供功能图标，如图 9-5 所示，它依次序提供了 close、collapse、down、expand、gear、help、left、maximize、minimize、minus、move、next、pin、plus、prev、print、refresh、resize、restore、right、save、search、toggle、unpin、up 等 25 个功能图标。其中 move 和 restore 两个 CSS 代码存在问题，而且也只有 24 个图片。这 25 个功能图标，有许多功能是固定的，一般可以自定义功能的有 gear、help、move、refresh、restore、save 和 search。

图 9-5　Tool 对象提供的功能图标

Header 对象和 Tool 对象是不能单独使用的组件，它们只是面板的构件，一般的配置项都是在面板中定义的，其属性和方法多数也可以在面板中看到。不过 Tool 对象虽然是在面板的 tools 配置项中定义的，但是其配置项比较多，因而需要说明一下，以下是其配置项的说明：

- ❑ type：显示的按钮类型，值为 25 个按钮的名称。
- ❑ handler：单击图标按钮后执行的函数。
- ❑ scope：作用域。
- ❑ stopEvent：布尔值，如果为 false，表示允许事件进行传播。默认值为 true，不允许事件传播。
- ❑ tooltip：可以是字符串，也可以说是 QuickTips 的配置对象，在图标上显示的提示信息。

9.3.4　记录和恢复面板属性：Ext.util.Memento

面板折叠后，宽度或高度（与折叠方向有关）会改变，因而需要记录其原始值，以便展开时可恢复原始值。Memento 对象的作用就是捕获这些属性值，在需要的时候恢复这些属性值。

其工作原理是在内部创建一个 JavaScript 对象，用来保存属性及其值。当调用 capture 方法时，它会将面板对应的属性及其值保存在 JavaScript 对象内，例如要保存面板的宽度（width），在 Memento 对象的 JavaScript 对象内也会有一个 width 的属性，调用 capture 方法后，其值就是面板的宽度值。

调用 restore 方法可将 JavaScript 对象属性值恢复到面板对应的属性里。而 restoreAll 则可将 Memento 对象捕获的所有属性值恢复到面板对应的属性里。Memento 对象还有一个 remove 方法用于移除 JavaScript 对象的属性。

9.3.5　面板常用的配置项、方法和事件

1. 常用配置项

- ❑ animCollapse：布尔值，为 true 时，面板折叠时会使用动画。
- ❑ bbar：底部工具栏，可为按钮的配置对象或由按钮的配置对象组成的数组。

- baseCls：面板的基本样式，默认值是“x-panel”。
- bodyBorder：布尔值，是否显示面板的主体的边框。只有在 frame 设置为 true 时才起作用。默认值为 undefined，显示面板的主体的边框。
- bodyCls：可设置面板的主体的样式类。
- bodyPadding：设置面板的主体的内边距。
- bodyStyle：设置面板的主体的样式。
- buttonAlign：页脚工具栏按钮的对齐方式，可为 right、left 和 center。默认值为 right。
- buttons：页脚工具栏，可为按钮的配置对象或由按钮的配置对象组成的数组。
- closable：布尔值，为 true 时表示面板可关闭。默认值为 false。
- closeAction ：字符串值，可为 destroy 和 hide。为 destroy 时，表示面板在关闭后会从 DOM 中移除面板节点及其子节点。为 hide 时，只是隐藏面板节点，也就是说可使用 show 方法重新显示面板。
- collapseDirection ：面板的折叠方向，值可以为 top、bottom、left 和 right。默认向标题栏方向折叠，因而该配置项一般可不定义。
- collapseFirst ：布尔值，默认值为 true，表示折叠（展开）按钮图标总是第一个渲染。为 false 时，将最后一个渲染折叠（展开）按钮图标。
- collapseMode ：字符串，表示折叠方式。该配置项只有在面板是 BorderLayout 布局的直接子组件时才起作用。值为 undefined（omitted）时，标题栏将作为占位符显示在布局内，通过工具按钮可重新展开面板；值为 header 时，标题栏仍可见，但不在 BorderLayout 内。
- collapsed ：布尔值，为 true，则面板渲染时是折叠的。默认值为 false，面板渲染时是展开的。
- collapsedCls：字符串，面板折叠时面板元素的样式，默认值为 collapsed。
- collapsible ：布尔值，为 true 时，表示面板是可折叠的，会在标题栏显示一个折叠（展开）的切换按钮。默认值为 false。
- dockedItems ：可将一个组件或多个组件作为停靠组件添加到面板。停靠位置可以是 top、right、left 和 bottom。一般用来停靠工具栏。可以根据需要，通过该配置项停靠你想要的组件。
- fbar：页脚工具栏。
- floatable：布尔值，该配置项只有在面板是 BorderLayout 布局的直接子组件时才起作用。默认值为 true，允许单击面板将面板浮动显示在布局之上。为 false 时，用户只有单击标题栏上的图标才能折叠（展开）面板。
- frameHeader：布尔值，为 true 且 frame 也为 true 时，会在标题栏显示圆角。
- headerPosition：字符串，标题栏的位置，值可以为 top、bottom、left 或 right。默认值为 top。
- hideCollapseTool：布尔值，为 true 时会隐藏折叠（展开）的按钮图标。
- lbar：左边工具栏。
- minButtonWidth：页脚工具栏按钮的最小宽度，默认值为 75。
- overlapHeader：为 true 时会将标题栏重叠在面板圆角之上。

- placeholder：该配置项只有在面板是 BorderLayout 布局的直接子组件时才起作用。将一个组件作为面板折叠时的占位符。
- preventHeader：布尔值，为 true 时会阻止标题栏的创建或显示。
- rbar：右边工具栏。
- tabr：顶部工具栏。
- title：标题栏的标题。
- titleCollapse：布尔值，为 true 时运行单击标题栏折叠（展开）面板。
- tools：配置标题栏显示的工具图标。
- unstyle：面板不使用任何样式。

2. 常用方法

- close：关闭面板。
- collapse：折叠面板。
- expand：展开面板。
- setIconCls：设置面板的图标的样式。
- setTitle：设置面板标题。
- toogleCollapse：切换面板的折叠或展开状态。

3. 事件

- beforecollapse：面板折叠前会触发该事件，返回 false 可阻止折叠。
- beforeexpand：面板展开前会触发该事件，返回 false 可阻止展开。
- collapse：面板折叠后会触发该事件。
- expand：面板展开后会触发该事件。
- iconchange：面板的图标改变后会触发该事件。
- titlechange：面板的标题改变后会触发该事件。

9.4　布局

9.4.1　布局概述

布局的基类是 Layout，从其派生出两个分支，一类是用于具体组件的布局（Component-Layout），另一类是用于容器的布局（ContainerLayout）。一般布局都不会生成具体的 HTML 元素，只负责计算组件的大小和位置，而表格等特殊的布局会生成 HTML 元素。组件类布局用于计算组件内的构件（也是组件）的大小和位置，如面板用到的 DockLayout 负责布置面板内的标题栏和工具栏。而容器类布局则负责计算容器内的子组件的大小和位置。

组件类布局是专用的，在此就不讲述了，本节主要讨论容器类布局。

9.4.2　布局的运行流程：Ext.layout.Layout

与组件一样，Layout 对象已经包含了布局类的整个运作流程，子类可通过重写方法添加

一些特殊处理，但是基本流程还是按照 Layout 对象走的，其流程在 8.2.4 节组件的创建流程中已讲述过，在此就不再赘述了。

9.4.3 容器类布局基类：Ext.layout.container.Container

ContainerLayout 对象是所有布局类的基类，它为容器类提供了 11 个共享方法。该类中一个比较重要的与渲染有关的方法是 finishRender 方法，在 8.2.4 节创建流程中已讲述过。

9.4.4 盒子布局、垂直布局与水平布局：Ext.layout.container.Box、Ext.layout.container.VBox 与 Ext.layout.container.HBox

HBoxLayout 对象与 VBoxLayout 对象的主要区别是，HBoxLayout 是以宽度划分区域，而 VBoxLayout 是以高度划分区域，它们的基本计算方式都是一样的，所以可将这些共同的东西抽出来创建 BoxLayout 对象作为它们的基类。

BoxLayout 对象作为基类，为 HBoxLayout 对象与 VBoxLayout 对象提供了基本的配置项和布局的计算方法。它为 HBoxLayout 对象与 VBoxLayout 对象提供了 defaultMargins、flex、pack、align 和 padding 这 5 个常用配置项。HBoxLayout 对象与 VBoxLayout 对象则提供了各自的 align 对象，用于设置组件的对齐方式。

下面我们通过 Firebug 来看看这些配置项的作用，练习将以 VBoxLayout 对象为主。

1. 配置项 flex

打开模板页，在命令行中输入以下代码：

```
Ext.create("Ext.panel.Panel",{
    width:300,
    height:300,
    renderTo:Ext.getBody(),
    layout:{type:"vbox"},
    items:[
        {width:50,flex:1,title:"面板1"},
        {width:100,flex:2,title:"面板2"},
        {width:150,flex:2,title:"面板3"}
    ]
})
```

这里要注意的是，**配置项 flex 是在子组件内定义的**，而不是在 layout 配置项中定义。上述代码会将一个 300×300 的面板渲染到页面，然后使用垂直布局在其内部渲染 3 个面板，子面板的宽度是固定的，而高度则由 flex 配置项决定。上述代码执行后将在页面看到如图 9-6 所示的结果，将 Firebug 面板切换到 Illuminations 面板，展开 Panel 对象后将子面板切换到 Layout 面板，将看到如图 9-7 所示的结果。图中，顶层面板的高度是 300，切换到子面

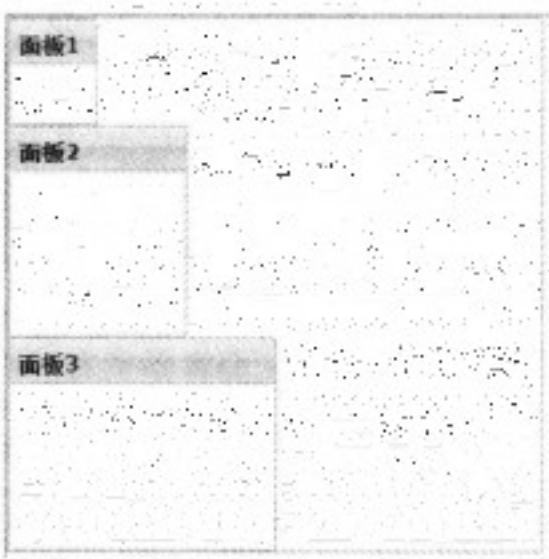


图 9-6 使用 flex 配置项的子面板显示效果

板，可看到子面板高度依次为 60、119 和 119，高度与总高度（300 减去顶部和底部边框的高度，实际是 298）的比值正好是各子面板 flex 值与总的 flex 值的比值。

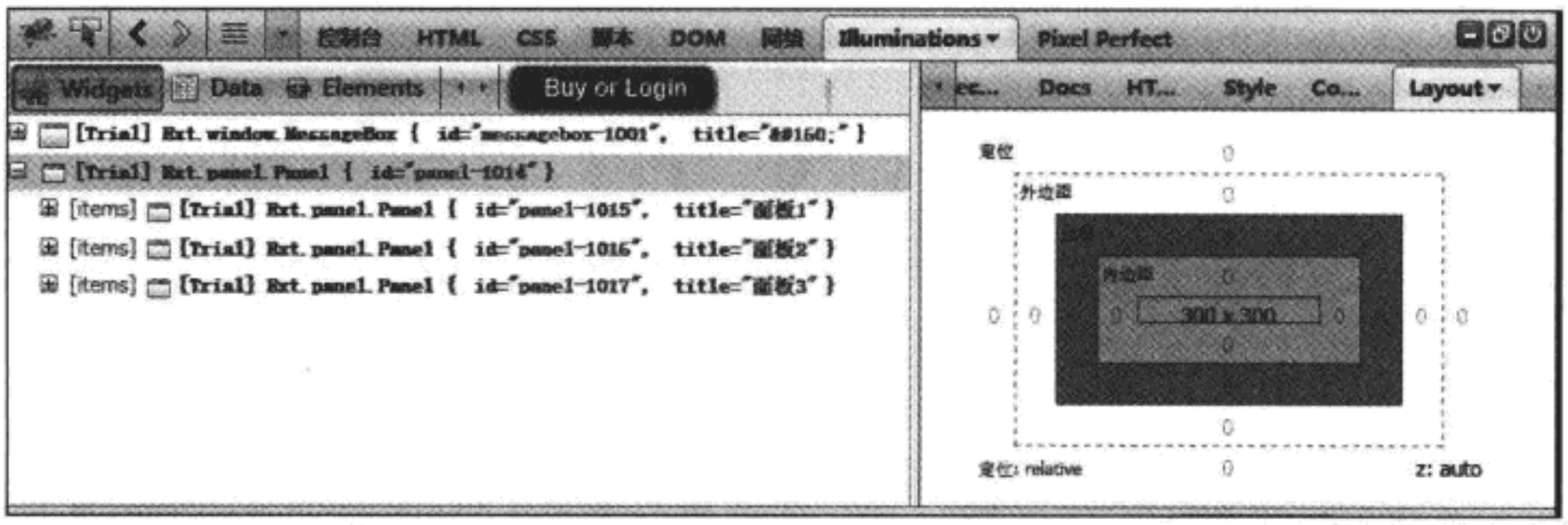


图 9-7　在 Illuminations 面板观察 Panel 的布局

现在将第 3 个子面板的 flex 值修改为“height:100”，刷新页面后再运行，会观察到面板 1 的高度是 66，而面板 2 的高度是 132，高度的和正好是总高度减去面板 3 的高度的值。这说明计算配置项 flex 的组件的高度（宽度）时，会先减去固定高度（宽度），再按比例划分。

2. 配置项 align

在配置项 layout 中添加“align:'center'”，将会看到如图 9-8 所示的效果，所有子面板都居中了，如果是水平布局面板则会水平居中。将 align 的值改为“stretchmax”，将看到子面板 1 和面板 2 忽略定义的宽度，变成与面板一样的宽度。再将 align 的值改为“stretch”，会看到所有子面板的宽度与容器的宽度一样了。

3. 配置项 defaultMargins

保持 align 的值为 stretch，再在配置项中添加 defaultMargins 属性，值为“{top:5, right:5, bottom:5, left:5}”，运行后将看到如图 9-9 所示的效果，在 Illuminations 面板中选择面板 1，在 Layout 子面板可看到如图 9-10 所示的效果，在图中并没有看到面板 1 设置了外边距，看看数值，再定位可看到面板 1 的 left 值是 5，top 值也是 5，正是 defaultMargins 属性定义的值，而宽度是 288，正好是总宽度 298 减去定义两边要偏移的数据 10。这说明 defaultMargins 属性定义的值参与了模板的计算。如果还不确定，再看看面板 2 的 top 坐标，是 71，正好是面板 1 的高度 56 加上 15 的结果。

根据以上对比，可以了解到，defaultMargins 属性会为每个子组件增加外边距，不过不是使用样式 margin 属性，而是直接使用坐标和组件大小控制的。

4. 配置项 padding

那么 padding 的情况是否一样呢？移除 defaultMargins 属性的定义，以避免影响对比。添加 padding 的定义，值为“5 5 5 5”（注意空格），运行后将看到如图 9-11 所示的效果。padding 属性并没有影响所有子组件，只是在容器内实现了 padding 的效果。检查面板的数值也是这样的。这说明 padding 是作为容器的内边距使用的，并且不是使用样式实现，而是通过计算子组件的位置和大小实现的。

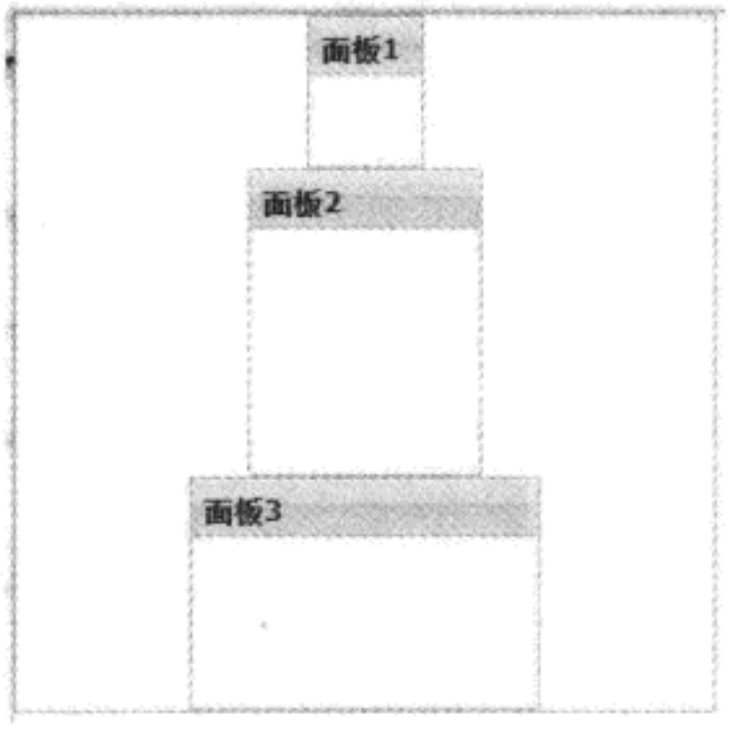


图 9-8　子面板居中的效果

图 9-9　配置了 defaultMargins 属性的效果

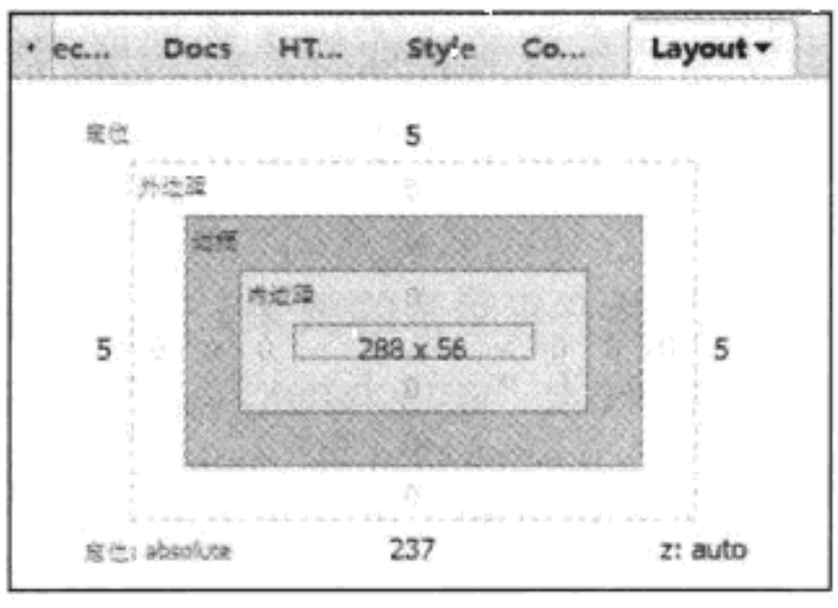


图 9-10　面板 1 的 Layout 面板

图 9-11　配置 padding 属性的效果

5. 配置项 pack

将配置项 layout 内除了 type 属性外的属性全部去掉，然后添加 pack 属性，值为“center”，接着将子面板的 flex 配置项删除，全部修改为 height，其值为 50，运行后将看到如图 9-12 所示的效果。从图中可以看到，如果子组件的大小不能填满容器，配置项 pack 可定义子组件的对齐方式。将 pack 的值改为 end，子面板将会排列在容器的底部。面板中的页脚工具栏是使用盒子布局的，因而也可以通过该配置项定义其对齐方式。

图 9-12　配置 pack 属性的效果

6. 总结

- defaultMargins：可为每个子组件定义外边距，但不是使用样式实现，而是通过计算位置和大小实现。其数据格式为包含 top、left、bottom、right 属性的对象。
- flex：可按照比例计算子组件的高度或宽度。
- pack：子组件在水平方向（垂直方向）的对齐方式。

值为 start 时，会左边（顶部）对齐；为 center 时，居中对齐；为 end 时，右边（底部）对齐。

- padding：可为容器设置内边距，但不是使用样式实现，而是通过计算子组件的位置和大小实现的。

9.4.5　为盒子模型提供调整大小的功能：Ext.resizer.Splitter

Splitter 对象可以为水平布局和垂直布局提供一个分隔条，通过拖动分隔条可调整子组件的大小，单击中间的按钮可折叠或展开子组件。

下面通过一个示例说明如何使用 Splitter 对象。

（1）功能描述

本示例主要演示如何使用 Splitter 对象。

（2）实现代码

要使用 Splitter 对象，有以下 3 个重要步骤：

- 定义容器组件的 maintainFlex 为 true。
- 创建 Splitter 对象实例，其配置项 collapseTarget 指向绑定组件，如果希望实例控制组件的折叠与展开，设置 performCollapse 配置项为 true。
- 将 Splitter 对象实例与子组件依次序放到容器中。

目标明确后，使用模板创建一个名为 9-1.html 的页面文件，然后在 OnReady 函数内先创建 3 个子面板：

```
var p1=Ext.create("Ext.panel.Panel",{
    collapsible:true,flex:1,title:" 面板 1"
    ,headerPosition:"left"
})

var p2=Ext.create("Ext.panel.Panel",{
    collapsible:true,flex:2,title:" 面板 2"
    ,headerPosition:"left"
})
var p3=Ext.create("Ext.panel.Panel",{
    collapsible:true,flex:2,title:" 面板 3",
    collapseDirection:"right"
    ,headerPosition:"left"
})
```

这次使用水平布局，因而需要设置面板的标题栏在左边。第 3 个面板设置了折叠方向为右边，目的是防止面板折叠后在右边显示一个空白区域。设置了折叠方向为右边，则可扩大子面板 2 的区域。

接着创建两个 Splitter 对象：

```
var r1=Ext.create('Ext.resizer.Splitter', {
    collapseTarget: p1,
    performCollapse: true
```

```
});

var r2=Ext.create('Ext.resizer.Splitter', {
    collapseTarget: p3,
    defaultSplitMin:50,
    defaultSplitMax:200,
    performCollapse: true
});
```

因为只有 3 个子面板，因此它们的相关关系只有两个，所以创建两个 Splitter 对象就可以了。

第一个 Splitter 对象实例负责管理子面板 1，使用了最简单的配置。第二个 Splitter 对象则负责管理子面板 3，并设置了 defaultSplitMin 和 defaultSplitMax 配置项。其中，defaultSplitMin 配置项控制面板可以调整到的最小宽度，defaultSplitMax 配置项则是控制面板可以调整到的最大宽度。

最后是定义容器：

```
var panel=Ext.create("Ext.panel.Panel",{
    width:300,
    height:300,
    maintainFlex:true,
    renderTo:Ext.getBody(),
    layout:{type:"hbox",align:'stretch'},
    items:[p1,r1,p2,r2,p3]
})
```

容器的配置项已经添加了 maintainFlex 配置项，且为 true。这里要注意 items 的组件位置，放错了会引起错误。

（3）页面效果

在浏览器中打开页面，并从左到右单击分隔条中的按钮，将看到如图 9-13 所示的效果。

将面板恢复原状后，可拖动一下最右边的分隔条，测试一下限制了最大宽度和最小宽度的效果。

在某些情况下直接使用 Splitter 对象比使用边框布局方便，可根据项目情况自己做出判断。以下是 Splitter 对象的 7 个配置项的说明：

- collapseOnDblClick：布尔值，为 true 时可使用双击折叠或展开组件。
- collapseTarget：Splitter 对象管理的组件。
- collapsedCls：字符串，组件折叠时会将该配置项定义的样式类添加到 Splitter 对象。
- collapsible ：布尔值，为 true 时会在分隔条中间显示一个按钮用来折叠或关闭组件。默认使用组件的 collapsible 配置项的设置。
- defaultSplitMax：数字，允许组件调整到的最大宽度。
- defaultSplitMin：数字，允许组件调整到的最小宽度。
- performCollapse ：布尔值，为 false 会阻止分隔条按钮管理组件的折叠与展开状态。默认值为 undefined。

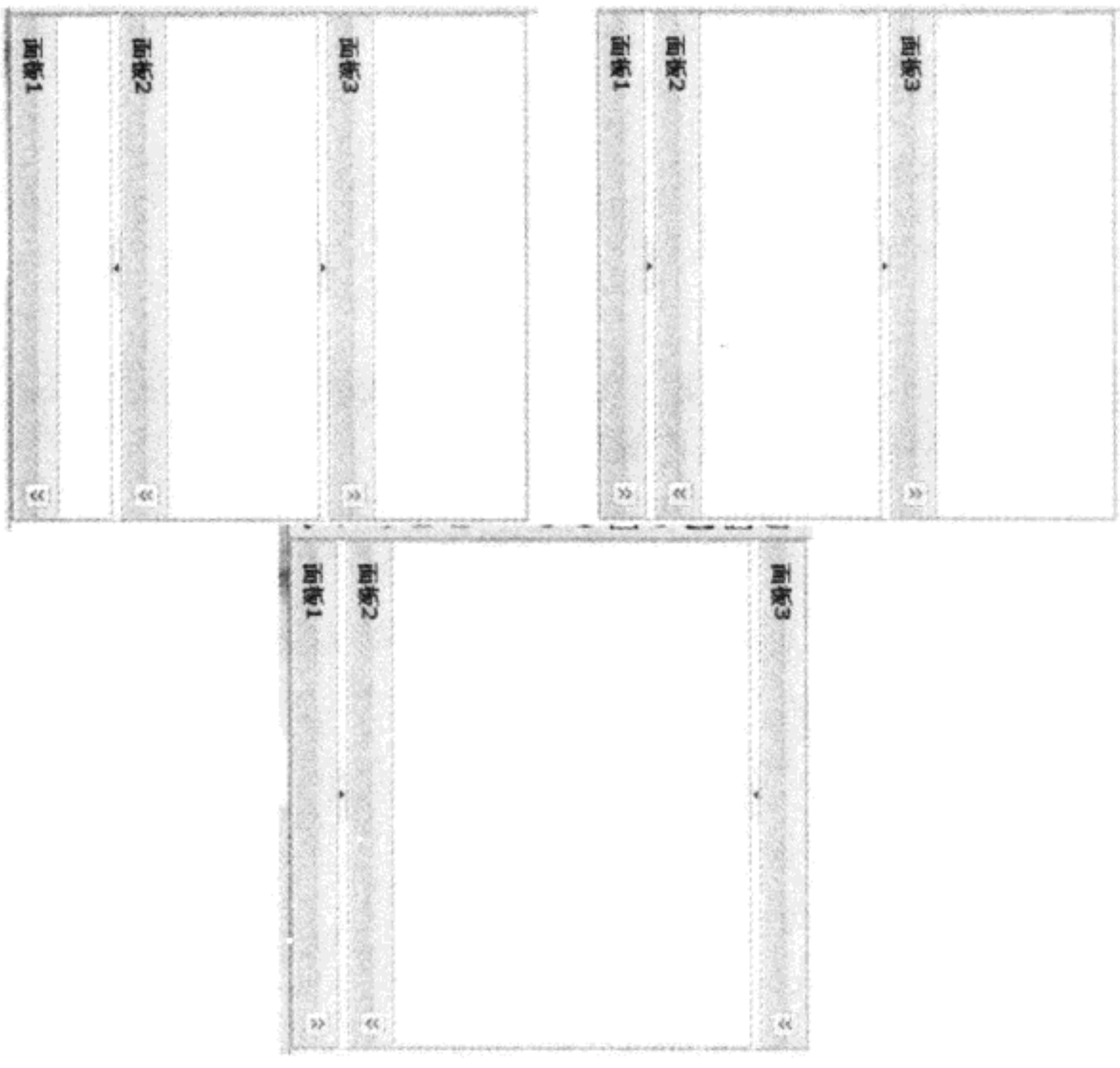


图 9-13　Splitter 对象示例的页面效果

9.4.6　手风琴布局：Ext.layout.container.Accordion

AccordionLayout 对象继承于 VboxLayout 对象，但是不用像 VboxLayout 对象那样需要在子组件内配置高度，它会填满容器的高度。以下是其独有的配置项：

- animate：布尔值，默认值为 true，布局的改变会使用动画过渡。
- collapseFirst：布尔值，为 true 时，组件在渲染 Tool 对象时会先渲染折叠（展开）按钮。默认值为 false。
- fill：布尔值，默认值为 true，活动组件的高度会填满容器空白区域的高度。
- hideCollapseTool：布尔值，为 true 时，会隐藏折叠（展开）按钮，默认值为 false。
- multi：布尔值，为 true 时运行显示多个组件的内容。默认值为 false。
- titleCollapse：布尔值，值为 true 时允许单击组件标题折叠或展开组件。默认值为 true。

以上配置项都是在容器中定义的，在子组件中没有配置项。

下面通过一些示例来练习一下如何使用 AccordionLayout 对象，先从最简单的定义开始。打开模板页，然后在命令行中输入以下代码：

```
Ext.create("Ext.panel.Panel",{
    width:300,
```

```
        height:300,
        renderTo:Ext.getBody(),
        layout:{type:"accordion"},
        items:[
            {title:" 面板 1"},
            {title:" 面板 2"},
            {title:" 面板 3"}
        ]
})
```

运行后将看到如图 9-14 所示的效果。从代码中可以看到，使用 AccordionLayout 对象非常简单，只需要简单配置容器的 layout 配置项就可以了。

现在测试一下 multi 的情况，在 layout 配置项中添加 multi 配置项，其值为 true，然后为所有子面板添加“height:50”，运行后将所有子面板展开将看到如图 9-15 所示的效果。

图 9-14　简单 Accordion 布局效果

图 9-15　配置项 multi 为 true 时的效果

其他配置项有兴趣可以自己研究一下。

9.4.7　锚固布局：Ext.layout.container.Anchor

锚固布局可将子组件锚固于容器，如果容器的尺寸发生改变，子组件会根据配置项 anchor 的设置跟随容器调整尺寸。

锚固布局在容器里没有额外的配置项，需要在子组件中通过 anchor 配置项进行定义，其值是由空格分隔的两个数值组成的字符串。这些值可以是百分数、整数或字符串。第一个值用来设置子组件的宽度与容器的宽度之间的关系，第二个值用来设置子组件的高度与容器的高度之间的关系。

如果值为百分数，则该值为子组件的宽度（高度）占容器宽度（高度）的百分比；如果值为整数，则子组件的宽度（高度）为容器宽度（高度）加上该值后的结果；如果为字符串，子组件必须先定义好宽度和高度，调整尺寸时，会根据子组件的初始宽度（高度）与容器的初始宽度（高度）的比值计算出子组件的宽度（高度）。为字符串时，对应于宽度，可设置的值为 right 或 r；对应于高度，可设置的值为 bottom 或 b。

如果只设置一个值，则调整大小时只会影响宽度。

打开模板页，测试一下锚固布局，在命令行中输入以下代码：

```
Ext.create("Ext.panel.Panel",{
    width:300,
    height:500,
    renderTo:Ext.getBody(),
    layout:{type:"anchor"},
    resizable:{handles: 's e se',pinned:true},
    items:[
          {title:" 面板 1",anchor:"90% 20%"},
          {title:" 面板 2",anchor:"-50 -300"},
          {title:" 面板 3",anchor:"-100 20%"},
          {title:" 面板 4",height:80,width:80,anchor:"r"}
    ]
})
```

容器为了实现可调整大小，添加了配置项 resizable，配置项 handles 定义了只能在右边、底部和右下角调整大小。定义了 4 个子面板，面板 1 的宽度为容器宽度的 90%，高度为容器高度的 20%；面板 2 的宽度为容器的宽度减去 50，高度为容器高度减去 300；面板 3 的宽度为容器宽度减去 100，高度为容器高度的 20%；面板 4 则固定了宽度和高度为 80，不过其宽度可随容器宽度的改变而改变。

执行后，调整一下容器的高度和宽度，将看到如图 9-16 所示的效果。

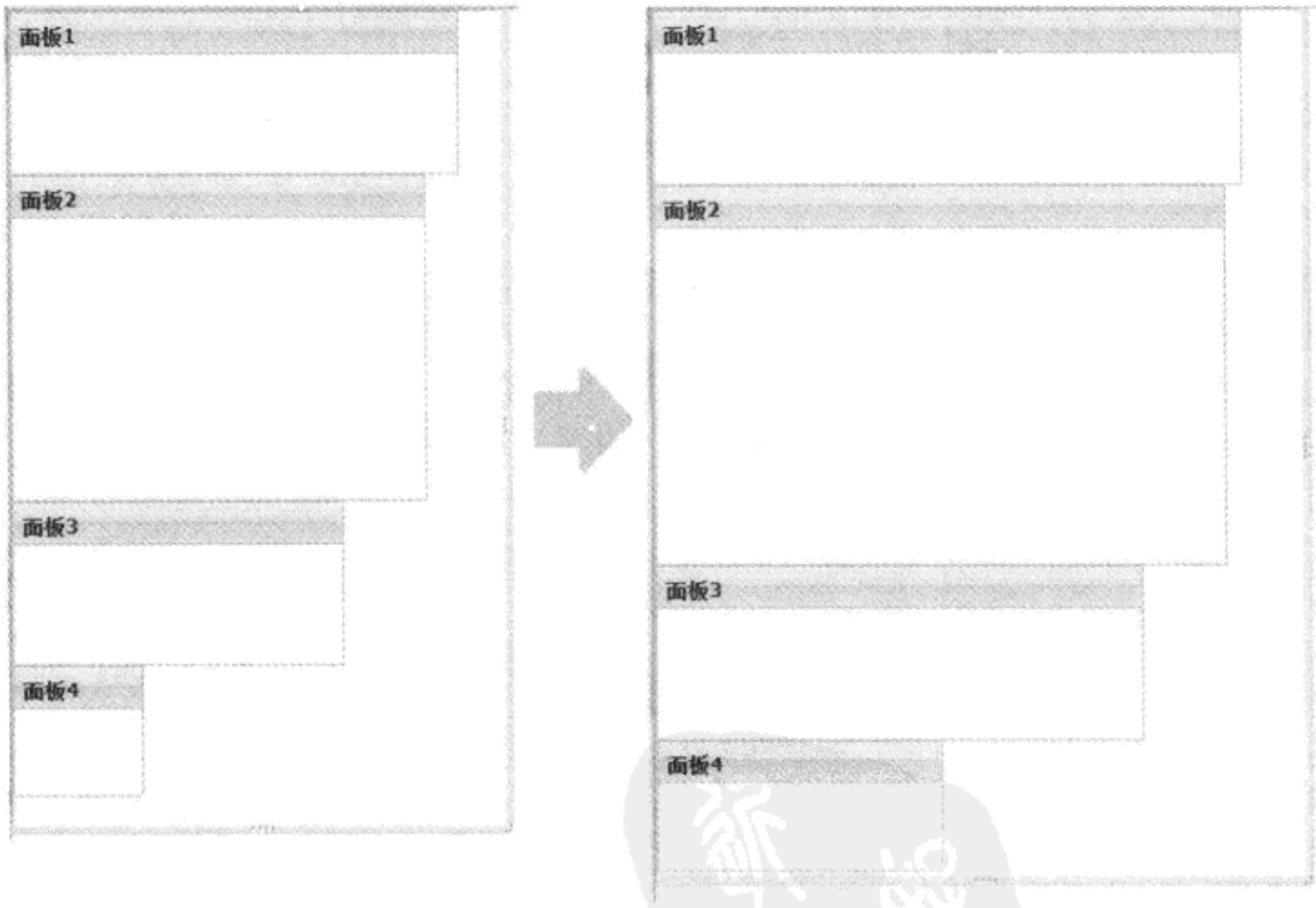


图 9-16　锚固布局的测试效果

锚固布局常用于可调整大小的表单，这样就可让输入框跟随容器的变化而变化，不至于造成很大的一片空白的现象。当然，也可以将容器设置为固定、不可调整大小的，这样也不

会出现空白的效果。

9.4.8 绝对定位布局：Ext.layout.container.Absolute

绝对定位布局继承于锚固布局，在锚固布局的基础上添加了定位功能，这样组件就可以根据坐标确定其起始位置。通过固定高度或宽度，再结合锚固，就可轻松让组件固定在某个位置，其实这就是使用样式的绝对定位，把组件固定在相对于容器的某个坐标位置。

AbsoluteLayout 对象与 AnchorLayout 对象一样，在容器里没有配置项，需要在子组件中进行定义。在 AnchorLayout 对象提供的 anchor 配置项的基础上增加了 x 和 y 这两个配置项来定义组件左上角的坐标。

AbsoluteLayout 对象不太常用，比较适合喜欢用坐标定位组件进行开发的开发人员，例如定义一个表单，可这样定义：

```
Ext.create("Ext.panel.Panel",{
    width:300,
    height:300,
    renderTo:Ext.getBody(),
    layout:{type:"absolute"},
    resizable:{handles: 's e se',pinned:true},
    items:[
          {xtype:"label",text:"标题：",x:5,y:5},
          {xtype:"textfield",name:"title",x:85,y:5,anchor:"-10"},
          {xtype:"label",text:"发布时间：",x:5,y:35},
          {xtype:"datefield",name:"posttime",x:85,y:35,anchor:"-10"},
          {xtype:"label",text:"内容：",x:5,y:65},
          {xtype:"htmleditor",name:"content",x:5,y:95,anchor:"-10 -10"}
    ]
})
```

绝对定位布局的最大好处就是组件的位置非常清晰，如上面代码，几个组件在同一行，每行组件之间的间隔是多少看看坐标就知道了，这也是有人喜欢这样布局的主要原因。

打开模板页，将以上代码放到命令行中运行，并调整一下容器的大小，可看到如图 9-17 所示的效果。因为包含了锚固布局，调整容器大小并没有对布局整体观感产生大的影响。

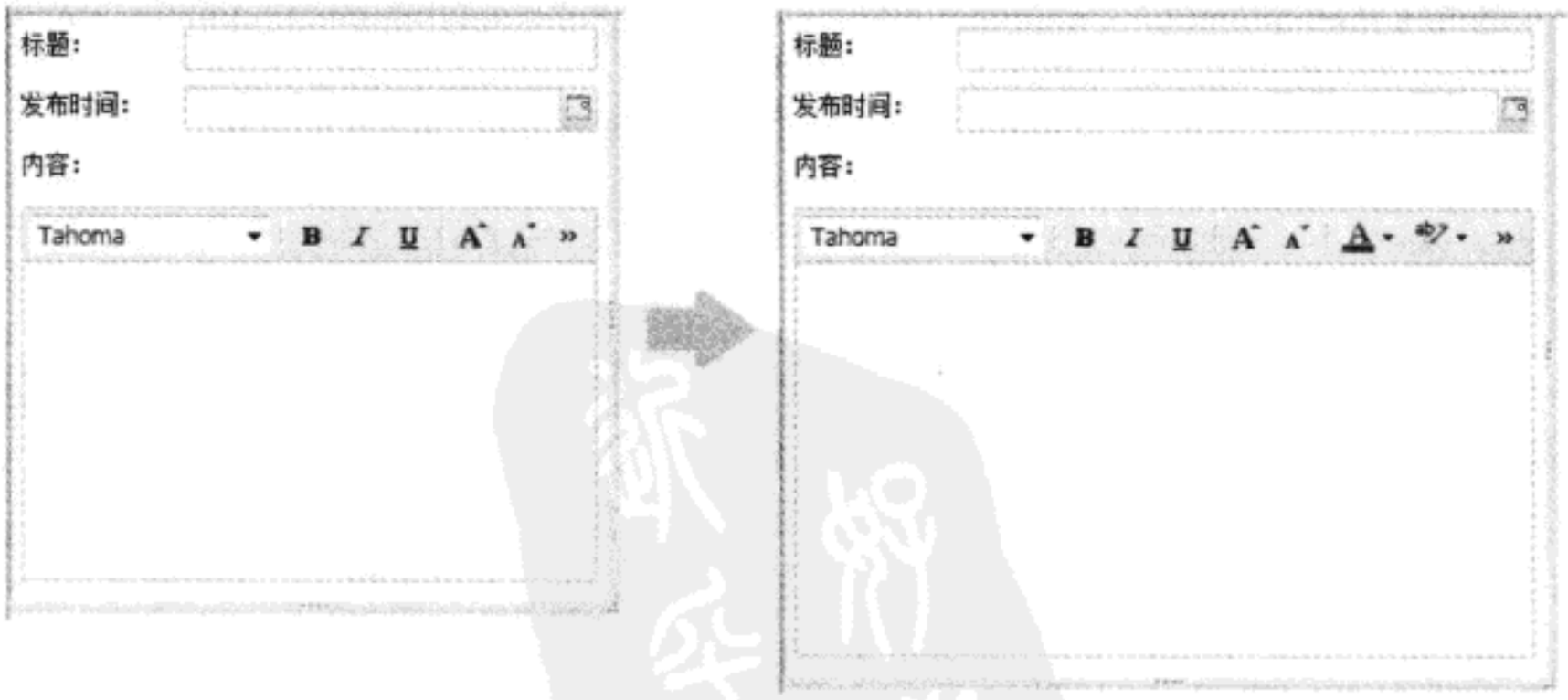


图 9-17 绝对定位布局的页面效果

9.4.9　边框布局：Ext.layout.container.Border

边框布局是应用程序常用的布局，也是在使用中会碰到比较多问题的布局，主要是因为对布局的整体不熟悉。如图 9-18 所示，它是最简单且完整的边框布局，由以下代码生成：

```
Ext.create("Ext.panel.Panel",{
    width:100,
    height:100,
    renderTo:Ext.getBody(),
    layout:"border",
    items:[
        {region:"north",height:30,html:"north"},
        {region:"west",width:30,html:"west"},
        {region:"center",html:"center"},
        {region:"east",width:30,html:"east"},
        {region:"south",height:30,html:"south"}
    ]
})
```

从图中可以看到，边框布局将容器划分成 north、west、center、east 和 south 这 5 个区域，而在代码中，这 5 个区域是由子组件的 region 配置项的值定义的，与子组件的位置没有关系，随便这 5 个子组件的位置怎么放，结果都是一样的。在这 5 个区域中，**center 区域是必须定义的，其他区域则可有可无**。如果没有定义 center 区域，在 Firebug 中会看到如图 9-19 所示的错误。

图 9-18　最简单且完整的边框布局

图 9-19　没有定义 center 区域的错误提示

从图 9-18 中可以看出，north 和 south 区域只要定义高度就行了，它们会自动在水平方向填满整个容器。当然，也可以根据 west、center、east 的高度，再加一个 north 或 south 的高度，让最后一个高度自适应容器余下的高度。而 west、center、east 这 3 个区域，需要定义宽度，一般情况是定义 west 和 east 的宽度，让 center 自适应，不过这些都可根据应用程序需求调整。

定义高度或宽度，可以使用 width 或 height 配置项，也可以使用 flex 配置项。center 区域默认的 flex 值为 1。如果使用 width 或 height 配置项，其值可以是具体数值，也可以是百分数。具体情况，可在 Firebug 中修改一下示例自己做一下测试。

如果需要使用分隔条调整子组件之间的大小，可在子组件中使用配置项 split，其值为 true 时会出现分隔条。**一般不在 center 区域定义配置项 split，因为无法判断该分隔条是使用在 east 区域还是 west 区域**。在 Firebug 命令行中，将刚才的代码修改成以下代码：

```
Ext.create("Ext.panel.Panel",{
    width:300,
    height:300,
    renderTo:Ext.getBody(),
```

```
    layout:"border",
    items:[
        {region:"north",height:"20%",split:true},
        {region:"west",flex:1,split:true},
        {region:"center"},
        {region:"east",flex:1},
        {region:"south",height:"20%"}
    ]
})
```

运行后将看到如图 9-20 所示的效果，在 north 的底部和 east 的右边出现了一个分隔条，可以通过拖动分隔条调整 north 和 east 的大小。

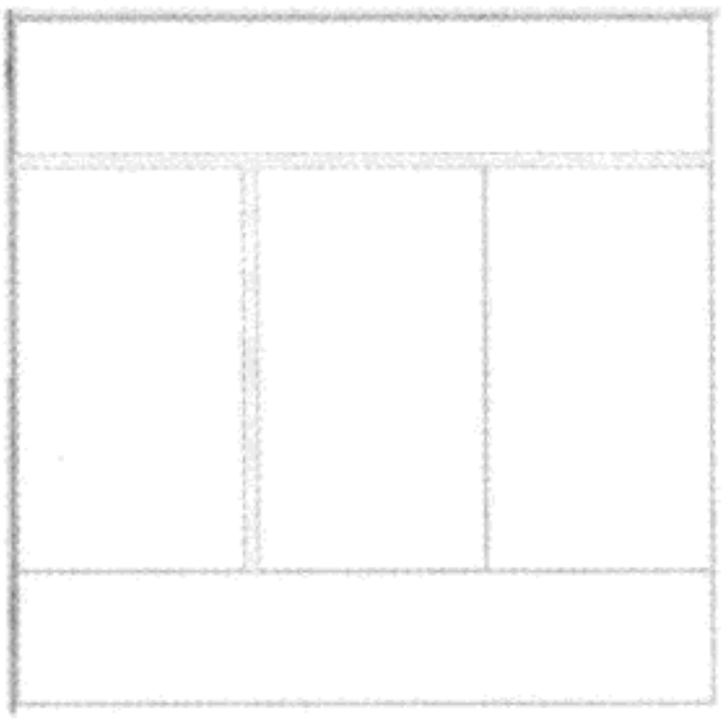


图 9-20　使用 split 配置项后的效果

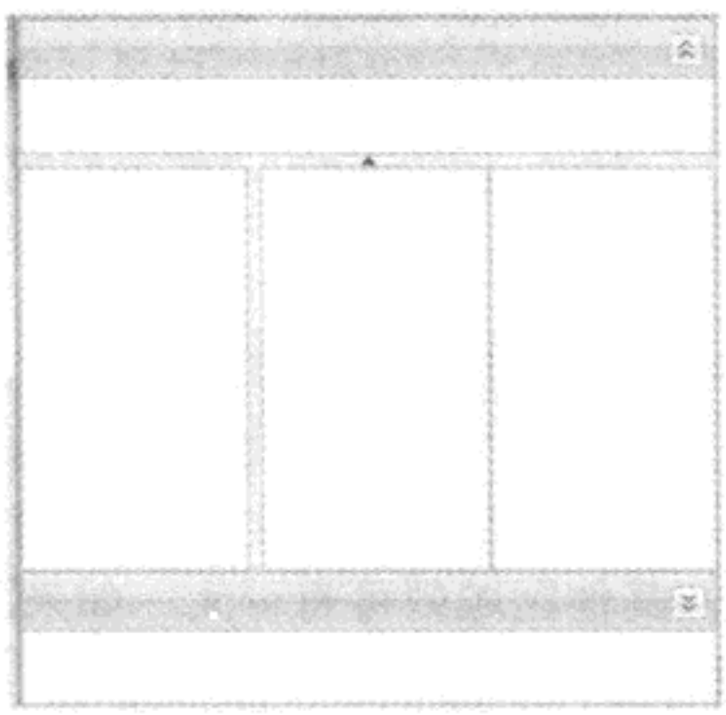


图 9-21　区域面板的折叠功能效果

如果子组件使用的是面板，可以使用面板的折叠功能折叠区域，例如修改刚才的代码，在 north 和 south 区域加入配置项 collapsible，设置其值为 true，执行后将看到如图 9-21 所示的效果。north 区域因为有分隔条，所以在分隔条中加了一个按钮用来折叠（展开）north 区域。而 south 没有分隔条，只能靠标题栏的按钮折叠（展开）面板。没有分隔条也可以折叠（展开）面板，它们是没有关联的。

在子组件中定义 minHeight、maxHeight、minWidth 和 maxWidth 这 4 个配置项可控制区域的最小高度、最大高度、最小宽度和最大宽度。当使用分隔条调整大小时，调整值如果超过最大值或小于最小值，将不能再扩大或缩小区域的高度或宽度。根据区域所处的位置可以知道，最小宽度和最大宽度对于 south 和 north 是没有意义的，因为它们左边和右边都没有可调整大小的对象。而 west、center、east 则都可以配置，这个自己可以做一下测试。

根据边框布局的行为，是否觉得这和在 VboxLayout 对象和 HboxLayout 对象中使用 Splitter 对象有点类似？是这样的，边框布局其实就是先用 VboxLayout 将容器分成 3 个区域，然后在中间区域使用 HboxLayout 将其分成 3 个区域构成的，而分隔条是使用 Splitter 对象生成的。不过控制区域的最大值和最小值不是使用 Splitter 对象的配置项来实现的，而是通过布局计算实现的。有兴趣可以自己深入研究一下。

在 4.1 Beta 1 版本中，多了一个 regionWeights 配置项，用于控制区域的权重。哪个区域的权重大，它显示的区域就优先，例如默认的权重为以下值：

```
regionWeights: {
    north: 20,
    south: 10,
    center: 0,
    west: -10,
    east: -20
},
```

因为 north 和 south 区域的权重比 west 和 east 的权重大，因而如图 9-18 所示，north 和 south 会优先显示。使用图 9-18 的示例代码，将“layout:"border",”这句修改为以下代码：

```
layout:{type:"border",
   regionWeights: {
       north: -20,
       south: -10,
       center: 0,
       west: 10,
       east: 20
   }
},
```

上述代码把 south 和 north 的权重与 west 和 east 的权重换过来了，代码执行后将看到如图 9-22 所示的结果。现在的结果是先按水平划分区域，再在中间区域垂直划分。这让边框布局变得更灵活了，再也不用像以前版本那样，要实现图 9-22 所示的布局需要嵌套使用边框布局了。

要修改权重除了通过在容器组件的 layout 配置对象中定义 regionWeights 配置项实现外，也可以单独在子组件中通过定义配置项 weight 来实现，例如要实现图 9-22 所示的效果，可在示例的 west 和 east 区域中添加配置项 weight，值为 50（只要权重值大于 north 和 south 的值就可以）。

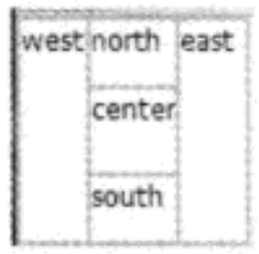


图 9-22　修改权重后的显示结果

在 4.1 版本中还有一个重大修改，就是允许相同区域有多个定义。如果示例中需要定义两个 south 区域，只需要增加一个 south 区域的定义就行了，不用再像以前一样，需要通过嵌套边框布局来实现。

9.4.10　自动布局：Ext.layout.container.Auto

自动布局是在容器没有默认布局也没有为其配置布局的情况下使用的，一般不单独使用。其基本功能就是依次序把子组件从上到下渲染到容器内，子组件的原始大小是怎样的就是怎样。

自动布局还会在最后一个子组件之后插入以下 HTML 代码：

```
<div role="presentation" class="x-clear" id="ext-gen1032"></div>
```

上述代码中，id 的值是随机的，并不一定是现在这个值。代码的关键在样式“x-clear”，其定义如下：

```
.x-clear {
clear:both;
```

```
font-size:0;
height:0;
line-height:0;
overflow:hidden;
width:0;
}
```

熟悉 CSS 的，对这个样式定义的作用应该很了解，其目的就是清除样式 float 对后续元素的影响，该 div 是为列布局配置的。

9.4.11 表格布局：Ext.layout.container.Table

顾名思义，表格布局就是使用表格来进行布局，而且它还真的会使用 Table 元素对子组件进行布局。Web 标准开发流行后，似乎谁用表格谁就落后。笔者的看法是，表格自有其用途，该用就用，尤其是显示数据的时候，表格比 div 方便多了。至于布局，就要慎重再慎重。

表格布局的配置项不少，既有应用于容器的，也有应用于子组件的，这与表格的特性有关。应用于容器的配置项有以下 4 个：

- columns：数字，表示表格的列数。
- tableAttrs：DomHelper 配置对象，可为 table 元素添加属性。
- tdAttrs：DomHelper 配置对象，可为 td 元素添加属性。
- trAttrs：DomHelper 配置对象，可为 tr 元素添加属性。

应用于子组件的配置项有以下 4 个：

- rowspan：与 HTML 元素 td 的 rowspan 作用一样，规定单元格可横跨的行数。
- colspan：与 HTML 元素 td 的 colspan 作用一样，规定单元格可横跨的列数。
- cellId：单元格的 id。
- cellCls：应用于单元格的样式类的名称。

下面尝试一下使用表格布局。打开模板页，在命令行中输入以下代码：

```
Ext.create("Ext.panel.Panel",{
    width:150,
    height:150,
    renderTo:Ext.getBody(),
    layout:{
            type:"table",
            tableAttrs:{style:"width:100%;height:100%"},
            columns:3
    },
    defaults:{bodyStyle:"background:#DFE9F6",width:50,height:50},
    items:[
            {colspan:2,html:"1-1",width:100},
            {rowspan:2,html:"1-2",height:100},
            {rowspan:2,html:"2-1",height:100},
            {html:"2-2"},
            {colspan:2,html:"3-1",width:100}
    ]
})
```

上述代码中为表格定义了 3 列，如果每个单元格的宽度和高度是固定的，就没必要为 table 元素添加高度和宽度为 100% 的样式属性了。defaults 的配置会成为子组件的默认配置项，包括面板的主体的背景色为 #DFE9F6，宽度和高度默认是 50。在 items 中定义了 5 个单元格，第 1、2 行都是两个，第 3 行只有一个。如果合并了行或列的单元格不指定宽度或高度，那么子组件就会根据默认配置的高度和宽度显示，不会填充满整个单元格的空间。先运行为合并单元格设置了高度或宽度的代码，再运行没有设置的代码，会看到如图 9-23 所示的效果。

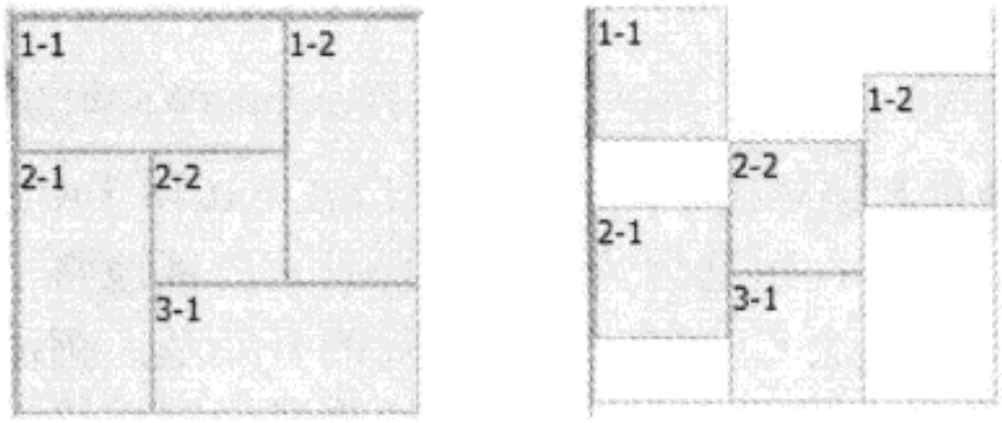


图 9-23　表格布局示例效果

表格布局可以解决一些比较复杂情况下的布局。这些复杂的布局，基本可以使用绝对定位布局实现，不过，需要耐心为每个区域定义设置好坐标和大小，如果没有这个耐心，那还是使用表格布局吧。

9.4.12　列布局：Ext.layout.container.Column

列布局的作用是将容器分列，不过，它给人的印象是难用，从 Ext JS 3 开始基本就被水平布局取代了。个中原因与列布局使用的布局样式有关。列布局使用 CSS 的 float 属性进行布局，熟悉 CSS 的，使用 float 布局不会太困难，但是对于不熟悉的，问题就会比较多，尤其是在不注意宽度的时候，布局很容易乱，这无疑是噩梦。例如，打开模板页，在命令中执行以下代码：

```
Ext.create("Ext.panel.Panel",{
    width:150,
    height:150,
    renderTo:Ext.getBody(),
    layout:{
          type:"column",
    },
    defaults:{width:50},
    items:[
          {html:"列 1"},
          {html:"列 2"},
          {html:"列 3"}
    ]
})
```

在页面中会看到如图 9-24 所示的效果，第 3 列到第 2 行去了，熟悉 CSS 的会知道这是

因为列宽小于 3 列的总宽度造成的，容器虽然定义了宽度是 150，但实际上只有 148（减了 2 个边框的宽度），而 3 个子组件的宽度总为 150，由于位置不够，第 2 列到第 2 行去了。

图 9-24　错误的列布局

图 9-25　第 3 列 columnWidth 为 1 的显示效果

Ext JS 的开发人员估计也意识到这个问题了，所以在 Ext JS 3 中加入了水平布局来代替列布局，不过，列布局也被保留了，这是因为水平布局只是机械地将子组件分列，当容器宽度不够时，超出容器宽度部分的就会看不到。而列布局的好处是，超过容器宽度的列会自动换行到下一行，这个特性通常会被用于网站中代替表格来显示固定格式的数据块，例如包含产品的缩略图、名称和价格的数据块。总而言之，与表格布局一样，好与不好，在于如何使用它，而与它本身的特性无关。

使用列布局，在容器里没有额外的配置项，在子组件中只要定义好宽度就行了，不过不建议全部子组件都使用固定宽度，不然，你必须精确计算好每列的宽度，其总宽度不能大于容器的实际宽度（不包含内边距、外边距、边框宽度）。最省事的办法是任何一个子组件都使用 columnWidth 配置项设置其宽度，并设置其值为 1，这样，该列就会使用计算处理的宽度作为其宽度，保证不会出现图 9-24 的错误了。例如，为示例中的第 3 列添加 columnWidth 配置项，并设置其值为 1，可看到如图 9-25 所示的效果，3 列都排列好了。当然，你也可以为多个列配置 columnWidth 配置项，只要它们的总和为 1 就行了。

9.4.13　自适应布局：Ext.layout.container.AbstractFit 与 Ext.layout.container.Fit

自适应布局主要用于只有一个子组件的时候，将子组件填充满容器。因而它在容器和子组件里都没有配置项，直接定义容器的配置项 layout 的值为 fit 就可以了。在此就不做演示了，有兴趣自己练习一下。

9.4.14　卡片布局：Ext.layout.container.AbstractCard 与 Ext.layout.container.Card

卡片布局是在自适应布局的基础上允许定义多个子组件，但每次只显示一个子组件，这类需要进行页面切换的布局比较适合做向导，Ext JS 4 的标签面板也是使用卡片布局。

卡片布局在容器和子组件中都没有配置项，只需要将容器的 layout 配置项的值设为 card 就可以了。不过为了操作卡片的切换，AbstractCardLayout 对象提供了以下 5 个方法：

- ❑ getActiveItem：返回布局中当前活动的子组件。
- ❑ getNext：返回当前活动的子组件的下一个子组件，如果不存在，会返回 false。
- ❑ getPrev：返回当前活动的子组件的上一个子组件，如果不存在，会返回 false。

❑ next：将当前活动的子组件的下一个子组件设置为活动（可见）状态。

❑ prev：将当前活动的子组件的上一个子组件设置为活动（可见）状态。

CardLayout 对象在 AbstractCardLayout 对象的基础上添加了 setActiveItem 方法，用来将指定的子组件、子组件的索引或子组件的 id 设置为活动状态。

下面用一个示例来练习一下如何使用卡片布局。

（1）功能描述

示例会在容器的顶部停靠一个工具栏，并定义三个按钮，其中“上一页”、“下一页”按钮用于切换使用卡片布局的子组件，“添加面板”按钮则会为容器动态添加一个面板。

（2）实现代码

例子的难点在于按钮调用的函数如何获取面板里面的布局对象，使用作用域内的变量是常用方法，不过该方法在代码太多、变量太多的时候不好使，所以在这里尝试一下为实例绑定方法的实现方式。

使用模板页创建一个名称为 9-2.html 的页面文件，然后在 OnReady 函数中添加以下代码：

```
Ext.create("Ext.panel.Panel",{
    width:200,
    height:200,
    renderTo:Ext.getBody(),
    layout:{type:"card"},
    activeItem:1,
    tbar:[
          {text:"上一页",id:'PrePage',handler:function(){
                this.up("panel").pageMove("prev")
          }},
          {text:"下一页",id:'NextPage',handler:function(){
                this.up("panel").pageMove("next")
          }},
          {text:"添加面板",id:'AddPage',handler:function(){
                var p=this.up("panel");
                var panel=Ext.create("Ext.panel.Panel",{title:"面  板"+(p.items.
                    length+1;})
                p.add(panel);
                p.getLayout().setActiveItem(panel);
                Ext.getCmp("NextPage").setDisabled(true);
          }},
    ],
    items:[
          {title:"面板1"},
          {title:"面板2"},
          {title:"面板3"},
          {title:"面板4"},
          {title:"面板5"}
    ],
    pageMove:function(dir){
          var layout=this.getLayout();
          layout[dir]();
          Ext.getCmp("PrePage").setDisabled(!layout.getPrev());
          Ext.getCmp("NextPage").setDisabled(!layout.getNext());
    }
})
```

粗体代码就是为实例绑定的方法，这样的好处是其作用域在实例内，所以其 this 指针会执行实例本身，这样直接使用 getLayout 就可获取实例的面板了。“上一页”、“下一页”按钮的句柄函数会把卡片布局的方法名（prev 或 next）传递给函数，这样，直接根据 JavaScript 对象的特性就可直接执行方法了，非常简单方便，连判断语句都不用。最后两句根据 getPrev 或 getNext 方法获取上一面板或下一面板，根据其是否为对象来判断能否执行上一页或下一页操作，如果返回 Null，则禁用按钮；否则，开启按钮。

在上一页、下一页按钮事件中，使用了 up 方法获取其父组件，这是 Ext JS 4 新增的功能，非常方便实用，然后直接调用实例内的 pageMove 方法即可。注意传递的参数是卡片布局的方法名称。

从代码中可以看到，在实例内绑定方法的方式在 OnReady 函数中不需要定义变量，因为其作用域就是在实例内部，直接通过 this 就可以访问容器本身。不过，要注意方法名称不能与对象已有的方法名称相同，不然会重写原有方法。自己做一个规定，例如在方法名称前加一个前缀，以区别这是自定义的方法。

添加面板按钮的句柄函数内，会先取得容器面板，然后创建一个新的面板，并使用 add 方法将面板添加到容器，接着就可以使用 setActiveItem 方法将其设置为活动状态了，要注意关闭“下一页”按钮，因为这时活动面板是最后一个面板。

在容器的定义中还使用了 activeItem 配置项设置容器渲染时首先显示哪个子面板，这里要注意索引是从 0 开始的，因而目前值为 1，会显示面板 2。

（3）页面效果

在浏览器中打开页面，然后单击创建面板将会看到如图 9-26 所示的效果。

图 9-26 卡片布局示例效果

注意面板标题的变化，刚开页面时显示的是面板 2，是配置项 activeItem 实现的效果。创建新面板后，新面板的标题是“面板 6”，单击“上一页”将回到面板 5。

9.5 标签面板

9.5.1 标签面板的构成及其运行流程：Ext.tab.Panel、Ext.tab.Bar 与 Ext.tab.Tab

图 9-27 说明了标签面板的结构，在 TabPanel 对象这个标签面板内，TabBar 可以像面板标题栏一样停靠在面板的顶部或底部，实际上它就是面板标题栏的子类。如果使用面板向左

右折叠的画图功能，应该还可以实现停靠左边或右边，不过估计难度比较大，目前还没有实现。在 TabBar 内，标签面板的标签（Tab）实际上就是修改了样式的按钮，按钮可以不断地加下去，而 TabPanel 的主体部分则使用卡片布局组织标签的内容页，所以每次只显示一个页面，标签的变化对主体来说只是添加或减少了一个组件而已。

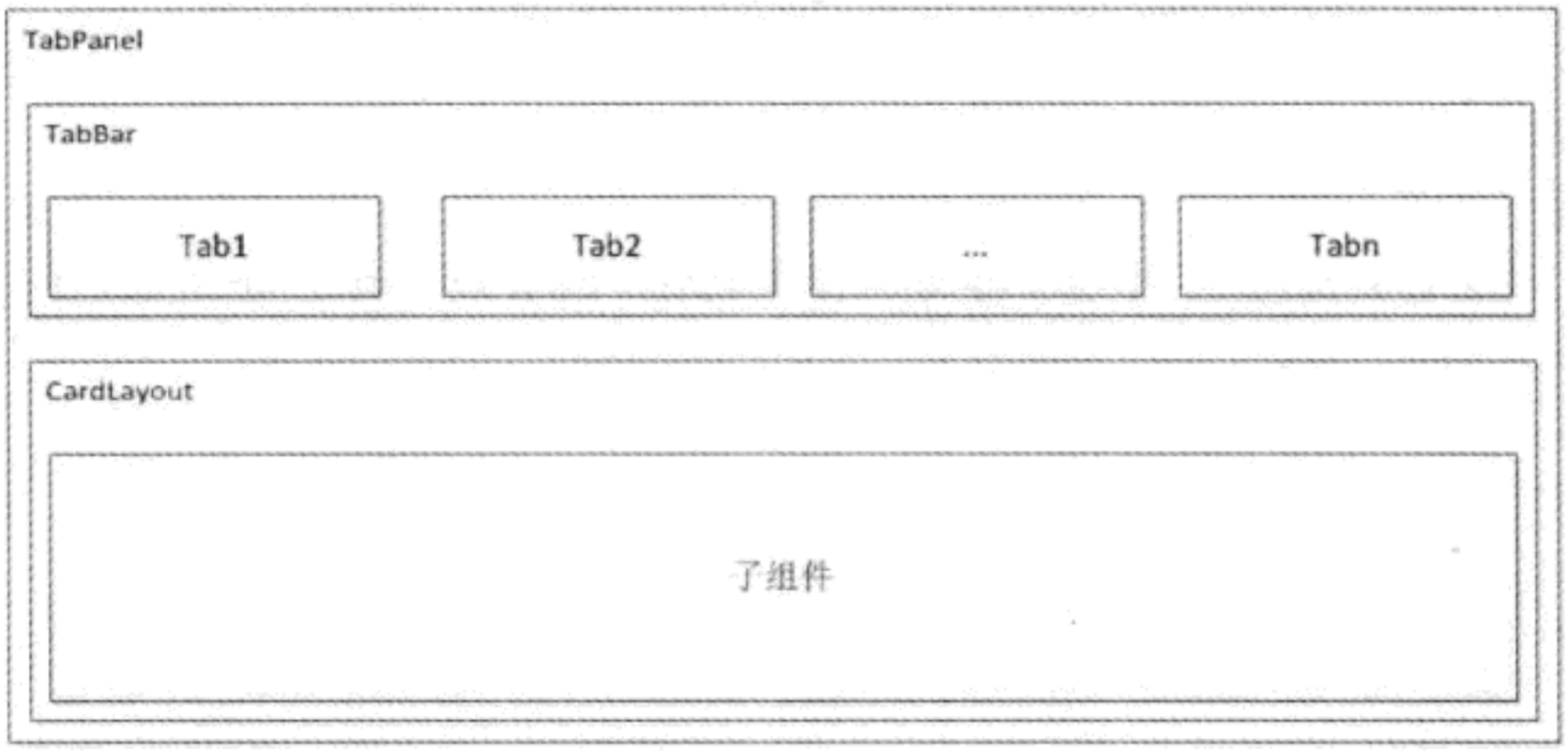


图 9-27　标签面板的结构图

下面看看标签页是如何运作的，首先从创建一个标签页开始，也就是创建 TabPanel 对象的实例，执行 initComponent 方法，其代码如下：

```
initComponent: function() {
    var me = this,
        dockedItems = me.dockedItems || [],
        activeTab = me.activeTab || 0;
    me.layout = Ext.create('Ext.layout.container.Card', Ext.apply({
        owner: me,
        deferredRender: me.deferredRender,
        itemCls: me.itemCls
    }, me.layout));
    me.tabBar = Ext.create('Ext.tab.Bar', Ext.apply({}, me.tabBar, {
        dock: me.tabPosition,
        plain: me.plain,
        border: me.border,
        cardLayout: me.layout,
        tabPanel: me
    }));
    dockedItems.push(me.tabBar);
    me.dockedItems = dockedItems;
    me.addEvents(
        'beforetabchange',
        'tabchange'
    );
    me.callParent(arguments);

    me.setActiveTab(activeTab);
    me.on('afterlayout', me.afterInitialLayout, me, {single: true});
},
```

上述代码先处理了dockedItems配置项和activeTab配置项，接着创建了一个CardLayout对象实例和TabBar对象实例。注意，TabBar对象实例的cardLayout属性会指向CardLayout对象实例，而tabPanel属性会执行标签面板自身，而标签面板的配置项tabPosition的值会作为TabBar对象实例停靠位置的配置项dock的值。

接着将tabBar放入dockedItems数组，并将dockedItems属性指向该数组。这样TabBar对象实例可以作为停靠对象停靠在标签面板的顶部或底部了，这由实例的dock配置项决定，也就是tabPosition配置项决定着停靠位置。

接着是为面板添加beforetabchange和tabchange事件，然后调用父类的initComponent方法，对子组件进行初始化等工作了。这些工作完成后，调用setActiveTab方法，设置活动标签和显示标签对应的内容页，其代码如下：

```
setActiveTab: function(card) {
    var me = this,
        previous;
    card = me.getComponent(card);
    if (card) {
        previous = me.getActiveTab();

        if (previous && previous !== card && me.fireEvent('beforetabchange', me,
            card, previous) === false) {
            return false;
        }
        me.tabBar.setActiveTab(card.tab);
        me.activeTab = card;
        if (me.rendered) {
            me.layout.setActiveItem(card);
        }
        if (previous && previous !== card) {
            me.fireEvent('tabchange', me, card, previous);
        }
    }
},
```

上述代码首先使用getComponent方法通过实例本身、索引或者id取得子组件。如果返回成功，则用getActiveTab方法返回当前活动标签，然后触发beforetabchange事件；如果返回false，则不再继续执行。否则，使用TabBar对象的setActiveTab方法设置子组件的标签为活动状态，再将activeTab属性指向当前的子组件。如果标签面板已经渲染（说明子组件也渲染了），则使用setActiveItem将子组件设置为活动状态。最后触发tabchange事件。

回到initComponent方法，将afterInitialLayout方法绑定到afterlayout事件。这样当布局计算结束后，就会调用afterInitialLayout方法，使用setActiveItem方法将activeTab属性指向的子组件设置为活动组件。

下面来看看TabBar对象实例是如何创建的，其initComponent方法代码如下：

```
initComponent: function() {
    var me = this,
        keys;
```

```
if (me.plain) {
    me.setUI(me.ui + '-plain');
}

me.addClsWithUI(me.dock);

me.addEvents(
    'change'
);

me.callParent(arguments);

me.on({
    click: me.onClick,
    element: 'el',
    delegate: '.' + Ext.baseCSSPrefix + 'tab',
    scope: me
});

me.layout.align = (me.orientation == 'vertical') ? 'left' : 'top';
me.layout.overflowHandler = new Ext.layout.container.boxOverflow.Scroller(me.
    layout);

me.remove(me.titleCmp);
delete me.titleCmp;

Ext.apply(me.renderData, {
    bodyCls: me.bodyCls
});
},
```

如果设置了 plain 配置项，为标签栏添加 plain 样式。接着添加自定义样式。调用 addEvents 方法添加 change 事件后调用父类的 initComponent 方法。

接着为元素绑定单击事件，实现标签的单击操作，这里采用了冒泡方式处理单击事件。

接着设置布局的对齐方式。然后创建 Ext.layout.container.boxOverflow.Scroller 的实例，当标签的总宽度大于容器的宽度时，在左边和右边显示一个箭头按钮用于滚动显示标签。

接着调用 remove 方法移除标题组件，因为 TabBar 是从面板的标题组件继承过来的，带有标题组件，在标签里并不需要，因而要移除。

最后将样式复制到渲染数据里。

从 TabBar 对象的 initComponent 方法可以看到，它并没有负责生成标签，那么标签是怎么添加到 TabBar 里的呢？这要从标签面板初始化子组件时入手，在 9.2.1 节容器的创建过程中，可以了解到容器有个初始化子组件的过程，而该过程会调用 Add 方法添加子组件，在 add 方法内会调用 onAdd 方法做一些处理，以下是 Tabpanel 的 onAdd 方法代码：

```
onAdd: function(item, index) {
    var me = this,
        cfg = item.tabConfig || {},
        defaultConfig = {
            xtype: 'tab',
            card: item,
```

```
            disabled: item.disabled,
            closable: item.closable,
            hidden: item.hidden && !item.hiddenByLayout, // only hide if it wasn't
                hidden by the layout itself
            tooltip: item.tooltip,
            tabBar: me.tabBar,
            closeText: item.closeText
        };

    cfg = Ext.applyIf(cfg, defaultConfig);
    item.tab = me.tabBar.insert(index, cfg);

    item.on({
        scope : me,
        enable: me.onItemEnable,
        disable: me.onItemDisable,
        beforeshow: me.onItemBeforeShow,
        iconchange: me.onItemIconChange,
        iconclschange: me.onItemIconClsChange,
        titlechange: me.onItemTitleChange
    });

    if (item.isPanel) {
        if (me.removePanelHeader) {
            item.preventHeader = true;
            if (item.rendered) {
                item.updateHeader();
            }
        }
        if (item.isPanel && me.border) {
            item.setBorder(false);
        }
    }
},
```

在开始时定义了一个默认的标签配置对象 defaultConfig，接着调用 applyIf 方法将默认配置对象与成员复制到自定义的配置对象中。

接着使用 insert 方法将 Tab 对象插入 TabBar 对象中，并用子组件的 tab 属性执行该 Tab 对象实例。然后为子组件绑定 6 个事件。

标签页是使用标签作为标题的，因而面板类组件的标题在 Tab 对象插入 TabBar 对象时，会将标题的文本复制到 Tab 对象并作为标签的显示文本。然后需要将该标题去掉，不然就重复显示标题了，一个在标签上，另一个在页面内。不过，如果你需要显示两个标题，可以将 removePanelHeader 配置项设置为 false。面板类组件还要将其边框去掉，以避免重复的边框。

最后是保证有一个能显示的活动标签页。

至此，标签面板就创建好了。

9.5.2 标签面板的配置项、属性、方法和事件

1. 标签面板的配置项

- plain：布尔值，如果为 true，则不显示背景。默认值为 false。

- deferredRender：布尔值，为 true 时，在其标签被激活时才渲染。为 false 时，所有子组件在布局渲染后会立即渲染。**默认值为 true，以便改善性能。设置该值为 false 要相当慎重，不然可能会造成性能损失。**
- itemCls：应用于子组件的样式。
- minTabWidth：标签的最小宽度，默认值为 30。
- removePanelHeader：布尔值，默认值为 true，表示不对容器的标题栏进行渲染。
- tabPosition：标签栏的放置位置，目前允许值为 top（顶部）和 bottom（底部）。

2. 子组件的 tabConfig（格式为对象）配置项内的配置项

- activeCls：子组件活动时应用的样式类。
- closable：布尔值，默认值为 true，表示标签可关闭，会显示关闭图标。**该值在 Tab 对象的定义中默认为 true，但在创建实例时，是根据子组件的 closable 值来配置的，因而，如果子组件没有配置 closable 为 true，则标签是不可关闭的。**
- closableCls：当标签可关闭时应用到标签的样式类。
- closeText：鼠标移动到关闭图标时显示的提示信息，默认值是“Close Tab”。可在本地化文件中重定义该值，让它显示中文。
- disable：布尔值，为 true 时，会禁用标签。默认值为 undefined。与 closable 配置项一样，创建时会使用子组件的 disable 配置项的值。
- disabledCls：禁用时应用到标签的样式类。

3. 标签面板的方法

- getActiveTab：返回当前标签面板内的活动子组件。
- getTabBar：返回标签栏。
- setActiveTab：将指定的组件设置为活动。

4. 标签的方法

- setCloseable：设置标签的关闭状态。

5. 标签面板的事件

- beforetabchange：当标签切换前会触发该事件，如果返回 false，会中止切换。
- tabchange：当标签切换后，会触发该事件。

6. 标签的事件（需在子组件的 tabConfig 内定义）

- active：当标签变为活动状态时触发该事件。
- beforeclose：标签关闭前会触发该事件，如果返回 false，会中止关闭。
- deactivate：当标签失去焦点时会触发该事件。

9.5.3　使用标签页

打开模板页，在命令行中输入以下代码：

```
var tp=Ext.create("Ext.tab.Panel",{
```

```
        width:400,
        height:300,
        plain:true,
        renderTo:Ext.getBody(),
        tabPosition:"bottom",
        minTabWidth:80,
        items:[
                {title:" 面板 1",closable:true},
                {title:" 面板 2",
                        tabConfig:{closable:true,
                                listeners:{
                                beforeclose:function(){
                                        return false;
                                }
                        }
                }},
                {title:" 面板 3"},
                {title:" 面板 4( 禁用 )",disabled:true}
        ],
})
```

上述代码会创建一个标签面板，运行后会看到如图 9-28 所示的结果，标签停靠在了底部，这是设置了 tabPosition 值为 bottom 的效果。标签栏的背景为白色是因为 plain 的值为 true。面板 1 和面板 2 都是可关闭的标签，不过一个是直接定义 closable 为 true，另一个是在 tabConfig 里配置的，但效果是一样的。禁用面板需要设置 disabled 为 true。

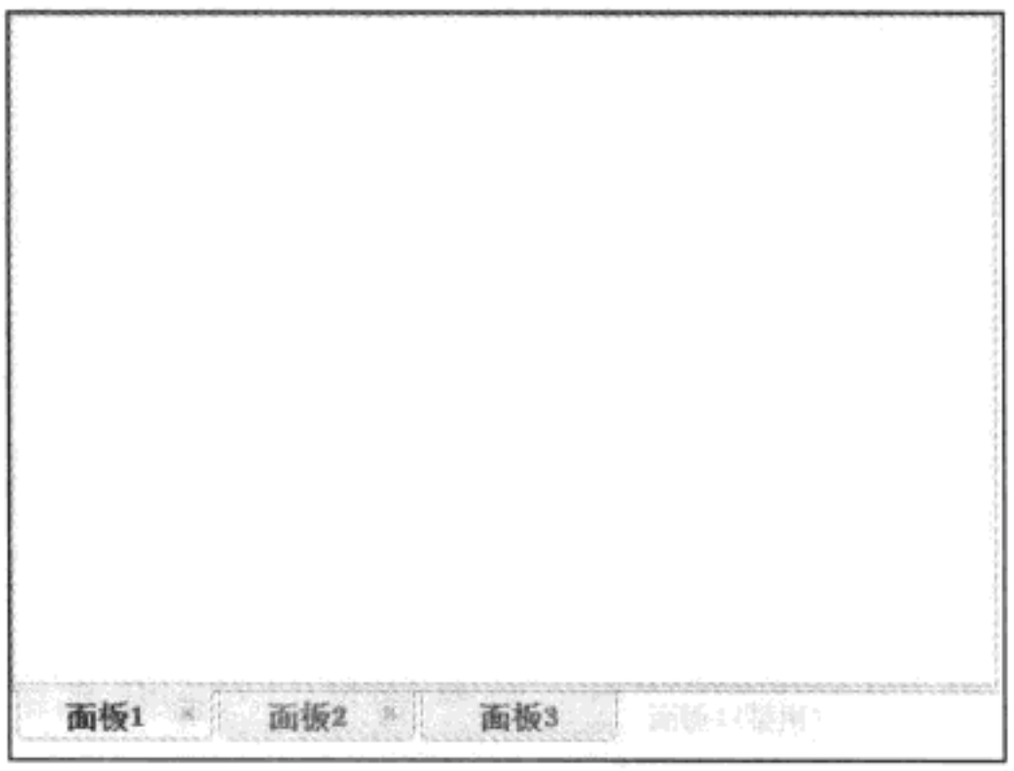


图 9-28　示例运行后的标签面板

单击面板 2 的“关闭”按钮，会发现关闭不了面板 2，因为在 tabConfig 里监听了事件 beforeclose 并返回 false。

1. 改变标签状态

先将面板 4 的状态修改为开启状态：

```
tp.items.getAt(3).setDisabled(false)
```

运行后，就可以单击选择面板 4 了。这里没有使用 Tab 对象的方法是因为从 TabPanel 到 Tab 对象需要先引用 TabBar 对象，最终还是要调用 items 取对象，而控制面板的禁用或开启状态会改变标签的状态，所以不如现在这样直接。要禁用面板的话，将代码中的 false 修改为 true 就行了。

设置标签是否可以关闭就要通过 tab 对象了，代码如下：

```
tp.items.getAt(3).tab.setClosable(true)
```

因为子组件的 tab 属性会指向它的标签对象，因而可直接引用。然后再调用其 setClosable 方法即可。

如果要把面板 3 变成活动状态，可输入：

```
tp.setActiveTab(2)
```

注意索引值是从 0 开始算的。

2. 添加标签页

添加标签页很简单，使用标签面板的 add 方法添加一个组件就可以了：

```
var p=Ext.create("Ext.panel.Panel",{title:" 面板 5"});
tp.add(p);
```

如果想在某个位置插入标签，可使用 insert 方法：

```
var p1=Ext.create("Ext.panel.Panel",{title:" 面板 6"});
tp.insert(1,p1);
```

运行代码后，面板 6 会插入到面板 1 之后。

9.5.4　可重用的标签页

标签关闭后，会自动销毁子组件，因而要重用必须重新创建并添加到标签面板里，API 里的建议是在定义标签面板时配置 autoDestroy: 的值为 false，这样就不会自动销毁子组件，但问题是，如果它不能自动销毁，就必须手动销毁了，又比较麻烦。笔者建议使用克隆的办法解决这个问题，在定义子组件时添加一个 clone 方法，用来返回实例的克隆实例，例如：

```
var p2=new Ext.tab.Panel({title:" 面板 7",closable:true,
 clone: function() {
   return new this.self(this.initialConfig);
 }
});
```

在类定义的时候，都会将其 self 属性指向类自身，而在类实例化的时候，会将 initialConfig 属性指向创建时的配置对象，因而通过 new 关键字就可重新创建一个实例并返回，也就相当于克隆了一个实例。接着要做的就是让标签面板添加这个克隆实例：

```
tp.add(p2.clone());
```

这样 p2 指向的实例就不会被销毁，自动销毁的只是它的克隆而已。

9.6　视图与选择模型

9.6.1　视图与选择模型概述

视图，也叫数据视图，其作用是将数据以模板及固定格式呈现出来，模板为显示的数据提供了一套操控机制，可以对数据进行选择、拖放、单击等操作。根据视图的功能可以知道，视图必须有数据与模板，也就是需要 Store 和 XTemplate 两个对象（Template 对象只能简单地

显示单个数据，不适合视图使用），因而，要使用视图，必须为视图定义 Store 和 XTemplate 对象。

一些与数据有关的组件，都会使用视图显示数据，如 Grid、树和下拉列表框，因而熟悉视图是了解这些组件运作的前提，基本上这些组件的行为都与视图有关。

视图类以 AbstractView 对象为抽象基类，DataView 为基类，分成两个分支，其中 TableView 对象会派生出用于 Grid 的 GridView 对象、用于树的 TreeView 对象，而另一分支 BoundLis 对象则用于下拉列表框，在其下派生出了 TimePicker 对象用于显示时间列表。

视图的数据与 UI 是分离的，那么当用户选择了某个条目时，程序怎么知道是选择了 Store 中的哪条记录呢？这就是选择模型的作用，它会跟踪用户的选择操作，然后再将对应的记录记录到选择模型内，这样，通过访问选择模型就可以获取选择的记录了。

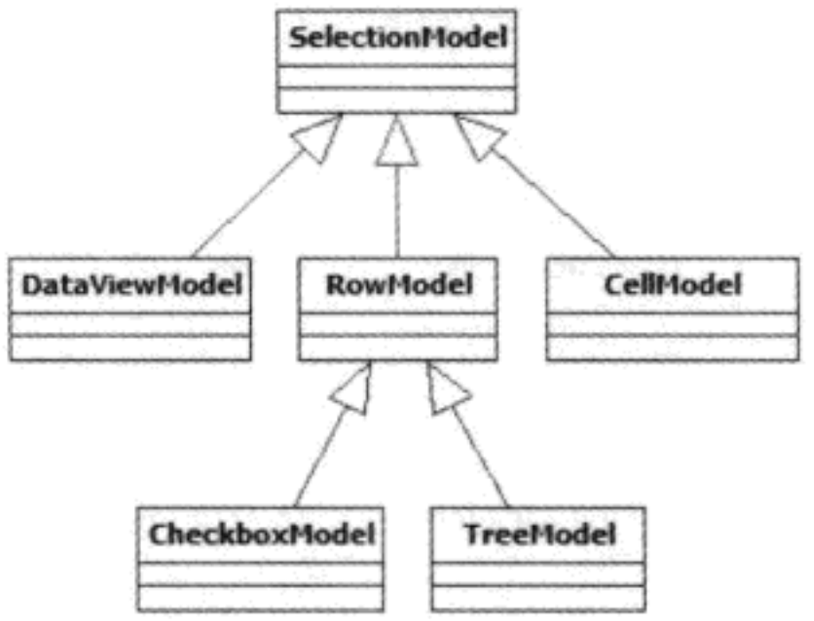


图 9-29　选择模型类的层次结构图

如图 9-29 所示，选择模型总共有 6 个类，以 SelectionModel 为基类，派生出了 DataView-Model、RowModel 和 CellModel 三个选择模型类，从 RowModel 又派生出了 Checkbox-Model 和 TreeModel 两个模型类。

9.6.2　视图的运行流程：Ext.view.AbstractView 与 Ext.view.View

AbstractView 对象派生于 Component 对象，因而其流程遵循 Component 对象的流程，但又有其自身的流程，其 initComponent 方法代码如下：

```
initComponent : function(){
    var me = this,
        isDef = Ext.isDefined,
        itemTpl = me.itemTpl,
        memberFn = {};
    if (itemTpl) {
        if (Ext.isArray(itemTpl)) {
            itemTpl = itemTpl.join('');
        } else if (Ext.isObject(itemTpl)) {
            memberFn = Ext.apply(memberFn, itemTpl.initialConfig);
            itemTpl = itemTpl.html;
        }
        if (!me.itemSelector) {
            me.itemSelector = '.' + me.itemCls;
        }
        itemTpl = Ext.String.format('<tpl for="."><div class="{0}">{1}</div></
            tpl>', me.itemCls, itemTpl);
        me.tpl =new Ext.XTemplate(itemTpl, memberFn);
}
// 省略调试代码
me.callParent();
```

```
if(Ext.isString(me.tpl) || Ext.isArray(me.tpl)){
    me.tpl = new Ext.XTemplate(me.tpl);
}
// 省略调试代码
me.addEvents(
        // 省略事件代码
);
me.addCmpEvents();
me.store = Ext.data.StoreManager.lookup(me.store || 'ext-empty-store');
me.all = new Ext.CompositeElementLite();

    me.scrollState = {
        top: 0,
        left: 0
    };
    me.on({
        scroll: me.onViewScroll,
        element: 'el',
        scope: me
    });
},
```

如果定义了配置项 itemTpl，则先处理它，其作用在粗体代码这句体现得很明显，你只需要定义一个条目的模板，视图会自动帮你加上循环代码，条目选择的样式也是由 itemCls 配置项决定的。这个配置项比较简单省事，但是最大的问题是成员函数必须先创建一个模板实例后才能应用到新创建的模板，这样就需要创建两次模板。所以，如果没有成员函数的模板，使用配置项 itemTpl 会比较简单；如果有成员函数，还是使用配置项比较好。

接着是添加事件，需要调用 addCmpEvents 方法，这个方法在 DataView 里有定义，会为对象添加 itemclick 等重要事件。

接着是处理 Store，将 store 属性指向 Store 实例。

最后将 all 属性指向一个 CompositeElementLite 对象实例，用来记录所有条目。这很重要，通过该对象可方便地控制每一个条目。

至此，initComponent 方法就完成了，接着会返回 AbstractComponent 继续后续的处理，而我们的关注点是渲染后的处理，首先是执行 DataView 对象 afterRender，其代码如下：

```
afterRender: function(){
    var me = this,
        listeners;
    me.callParent();
    listeners = {
        scope: me,
        freezeEvent: true,
        click: me.handleEvent,
        mousedown: me.handleEvent,
        mouseup: me.handleEvent,
        dblclick: me.handleEvent,
        contextmenu: me.handleEvent,
        mouseover: me.handleEvent,
        mouseout: me.handleEvent,
        keydown: me.handleEvent
```

```
        };
        me.mon(me.getTargetEl(), listeners);
        if (me.store) {
            me.bindStore(me.store, true);
        }
    },
```

调用 AbstractView 对象的 afterRender 方法，其代码如下：

```
afterRender: function() {
    this.callParent(arguments);
    this.getSelectionModel().bindComponent(this);
},
```

还是先调用其父类的 afterRender 后再执行 getSelectionModel 方法，其代码如下：

```
getSelectionModel: function(){
    var me = this,
        mode = 'SINGLE';

    if (!me.selModel) {
        me.selModel = {};
    }
    if (me.simpleSelect) {
        mode = 'SIMPLE';
    } else if (me.multiSelect) {
        mode = 'MULTI';
    }
    Ext.applyIf(me.selModel, {
        allowDeselect: me.allowDeselect,
        mode: mode
    });
    if (!me.selModel.events) {
        me.selModel = Ext.create('Ext.selection.DataViewModel', me.selModel);
    }
    if (!me.selModel.hasRelaySetup) {
        me.relayEvents(me.selModel, [
            'selectionchange', 'beforeselect', 'beforedeselect', 'select',
                'deselect']);
        me.selModel.hasRelaySetup = true;
    }
    if (me.disableSelection) {
        me.selModel.locked = true;
    }
    return me.selModel;
},
```

从上述代码可以看到，默认是单一选择（SINGLE）模式，如果定义 simpleSelect 为 true，则使用简单（SIMPLE）模式；如果定义了 multiSelect 为 true，则使用多选（MULTI）模式。然后创建 DataViewModel 对象。接着使用 relayEvents 方法将选择模型的事件传播到视图，这样相当于为视图添加了选择模型的事件。如果设置了 disableSelection 为 true，则将选择的属性 locked 设置为 true，表示不能进行选择，最后返回选择模型。

选择模型返回后，会执行选择模型的 bindComponent 方法，这个方法稍后介绍。现在回到 DataView 的 afterRender 方法。

调用父类的 afterRender 方法后，会定义一个 listeners 对象，然后将其定义的事件绑定到视图的目标元素，这将采用事件传播机制管理条目的鼠标和键盘操作。要注意的是，所有的事件都绑定了 handleEvent 方法，它会怎么处理这些不同的事件呢？其代码如下：

```
handleEvent: function(e) {
    if (this.processUIEvent(e) !== false) {
        this.processSpecialEvent(e);
    }
},
```

调用了 processUIEvent 方法，其代码如下：

```
processUIEvent: function(e) {
    var me = this,
        item = e.getTarget(me.getItemSelector(), me.getTargetEl()),
        map = this.statics().EventMap,
        index, record,
        type = e.type,
        overItem = me.mouseOverItem,
        newType;
    if (!item) {
        if (type == 'mouseover' && me.stillOverItem(e, overItem)) {
            item = overItem;
        }
        if (type == 'keydown') {
            record = me.getSelectionModel().getLastSelected();
            if (record) {
                item = me.getNode(record);
            }
        }
    }
    if (item) {
        index = me.indexOf(item);
        if (!record) {
            record = me.getRecord(item);
        }
        if (me.processItemEvent(record, item, index, e) === false) {
            return false;
        }
        newType = me.isNewItemEvent(item, e);
        if (newType === false) {
            return false;
        }
        if (
            (me['onBeforeItem' + map[newType]](record, item, index, e) === false)
                ||
            (me.fireEvent('beforeitem' + newType, me, record, item, index, e) ===
                false) ||
            (me['onItem' + map[newType]](record, item, index, e) === false)
        ) {
            return false;
```

```
            }
            me.fireEvent('item' + newType, me, record, item, index, e);
        }
        else {
            if (
                (me.processContainerEvent(e) === false) ||
                (me['onBeforeContainer' + map[type]](e) === false) ||
                (me.fireEvent('beforecontainer' + type, me, e) === false) ||
                (me['onContainer' + map[type]](e) === false)
            ) {
                return false;
            }

            me.fireEvent('container' + type, me, e);
        }
        return true;
    },
```

上述代码首先使用了事件对象的 getTarget 方法，通过配置项 itemSelector 定义的选择符获取对象。接着从静态方法中的 EventMap 对象取得事件映射表。属性 mouseOverItem 记录的是鼠标移动到其上方的条目，它的值会赋给变量 overItem。

如果取不到目标条目，则检查是否事件为 mouseover 且鼠标一直在 overItem 指向的条目内，若是，就指向该条目；如果事件是 keydown，则从选择模型中使用 getLastSelected 方法获取最后选择的记录，然后调用 getNode（AbstractView 对象）方法，其代码如下：

```
    getNode : function(nodeInfo) {
        if (!this.rendered) {
            return null;
        }
        if (Ext.isString(nodeInfo)) {
            return document.getElementById(nodeInfo);
        } else if (Ext.isNumber(nodeInfo)) {
            return this.all.elements[nodeInfo];
        } else if (nodeInfo instanceof Ext.data.Model) {
            return this.getNodeByRecord(nodeInfo);
        }
        return nodeInfo;
    },
```

如果组件还没渲染，返回 null；如果参数 nodeInfo 是字符串，则使用 document 对象的 getElementById 返回 HTMLElement 对象；如果参数 nodeInfo 是数字，则在 all 指向的 CompositeElementLite 对象实例记录的元素中，根据索引找出 HTMLElement 对象并返回；如果参数 nodeInfo 是数据模型，则调用 getNodeByRecord 返回元素。其代码如下：

```
    getNodeByRecord: function(record) {
        var ns = this.all.elements,
            ln = ns.length,
            i = 0;
        for (; i < ln; i++) {
            if (ns[i].viewRecordId === record.internalId) {
                return ns[i];
```

```
            }
        }
        return null;
    },
```

代码很简单，关键是粗体代码这一行，**它说明了元素对象与 Store 中的记录是通过记录的 internalId 属性建立关系的**，在 all 指向的 CompositeElementLite 对象实例中的元素对象，会在 viewRecordId 属性里保存记录的 internalId。明白这点，整个选择操作流程就很清楚了。用户选择了一个条目，就会从 all 中找出其元素对象，然后根据 viewRecordId 在 Store 中寻找 internalId 与之相等的记录，然后该记录会被记录到选择模型中。取消选择则简单很多，在选择模型中删除 internalId 属性值与 viewRecordId 属性值相等的记录就行了。

回到 processUIEvent 方法，如果 item 存在，则使用 indexOf 方法获取条目的索引，其代码如下：

```
indexOf: function(node) {
    node = this.getNode(node);
    if (Ext.isNumber(node.viewIndex)) {
        return node.viewIndex;
    }
    return this.all.indexOf(node);
},
```

上述代码调用 getNode 方法返回元素对象，如果其 viewIndex 是数字，则说明是索引，返回该值；否则，使用 CompositeElementLite 对象的 indexOf 方法找索引。

索引返回后，如果记录不存在，使用 getRecord 方法返回记录，其代码如下：

```
getRecord: function(node){
    return this.store.data.getByKey(Ext.getDom(node).viewRecordId);
},
```

上述代码根据 HTMLElement 对象的 viewRecordId 属性在 Store 中找出记录并返回。

记录找到后，则开始处理事件。首先是调用 processItemEvent 方法处理条目的事件，如 TabelView 对象会处理单元格单击、单元格双击等事件。

然后调用 isNewItemEvent 方法检测是否有新事件。如果有，方法 isNewItemEvent 会将 mouseover 事件转换为 mouseenter，而 mouseout 则转换为 mouseleave 事件；如果没有，返回 false。

接着是调用事件对应的方法和触发 before 类事件。

最后还要触发操作完成后的事件。

如果条目还是不存在，则会将事件传播到容器，触发容器的事件，最后触发容器操作完成后的事件。

从上述代码可以看到，processUIEvent 方法主要通过一个事件的映射表来执行对应的方法，例如事件类型是 click，对照表的值是 Click，则会调用 onBeforeItemClick、onItemClick 等方法，而触发事件的事件 beforeitemclick 或 itemclick。非常巧妙的一个处理，避免了书写大量的重复代码。

方法 processUIEvent 处理完成后，如果返回 true，则调用 processSpecialEvent 方法执行一

些特殊处理，例如 TableView 会根据加入的功能类（feature）进行功能处理。

回到 afterRender 方法，绑定事件后，如果 store 属性不存在，则调用 bindStore 方法绑定 Store，其代码如下：

```
bindStore : function(store, initial) {
    var me = this;
    me.mixins.bindable.bindStore.apply(me, arguments);
    me.getSelectionModel().bindStore(me.store);

    if (store) {
        if (initial && me.deferInitialRefresh) {
            Ext.Function.defer(function () {
                if (!me.isDestroyed) {
                    me.refresh(true);
                }
            }, 1);
        } else {
            me.refresh(true);
        }
    }
},
```

上述代码首先调用 Bindable 对象的 bindStore 方法，其代码如下：

```
bindStore: function(store, initial){
    var me = this,
        oldStore = me.store;

    if (!initial && me.store) {
        if (store !== oldStore && oldStore.autoDestroy) {
            oldStore.destroyStore();
        } else {
            me.unbindStoreListeners(oldStore);
        }
        me.onUnbindStore(oldStore, initial);
    }
    if (store) {
        store = Ext.data.StoreManager.lookup(store);
        me.bindStoreListeners(store);
        me.onBindStore(store, initial);
    }
    me.store = store || null;
    return me;
},
```

如果还没进行初始化且 Store 存在，则检查传递过来的 Store 对象与已存在的 Store 对象是不是同一对象，如果不是且 autoDestroy 属性为 true，则销毁已存在的 Store 对象；否则，只是解除事件的绑定。最后都要调用 onUnbindStore 方法，在 AbstractView 对象中，该方法的作用是将 LoadMask[⊖]对象与 Store 解除绑定。

如果参数 store 不存在，则在 StoreManager 中找出 Store 对象，调用 bindStoreListeners 方

⊖ 相关信息请阅读 16.4 节。

法为 Store 绑定事件，其代码如下：

```
bindStoreListeners: function(store) {
    var me = this,
        listeners = Ext.apply({}, me.getStoreListeners());

    if (!listeners.scope) {
        listeners.scope = me;
    }
    me.storeListeners = listeners;
    store.on(listeners);
},
```

上述代码先调用 getStoreListeners 方法返回监听对象，在 AbstractView 对象中，其代码如下：

```
getStoreListeners: function() {
    var me = this;
    return {
        refresh: me.onDataRefresh,
        add: me.onAdd,
        remove: me.onRemove,
        update: me.onUpdate,
        clear: me.refresh
    };
},
```

从返回的对象可以看到，需要为 Store 的 refresh、add、remove、update 和 clear 这 5 个事件绑定 AbstractView 对象中的方法。这样，当 Store 执行了更新、删除、新增和加载等操作时，就会调用相应的方法去处理对应的 HTML 元素，例如，当记录被删除时，会调用 onRemove 方法，它会从 all 对象中删除元素，然后更新余下元素的索引。如果还有记录，则调用 refresh 方法刷新显示，最后触发 itemremove 事件。有兴趣自己多研究研究，这对了解 Ext JS 的三层架构模式非常有帮助。

回到 bindStore 方法，为 Store 绑定事件后，就调用 onBindStore 方法为 LoadMask 对象绑定 Store，这样在 Store 发生加载操作时就会自动出现遮蔽，非常方便。

将 store 数组指向 Store 对象后，则调用选择模型的 bind 方法将 Store 绑定到选择模型。

最后，如果 store 存在，无论何种情况（延时或不延时），都会调用 refresh 方法刷新 UI，其代码如下：

```
refresh: function() {
    var me = this,
        targetEl,
        targetParent,
        records;

if (!me.rendered || me.isDestroyed) {
    return;
}

me.fireEvent('beforerefresh', me);
```

```
        targetEl = me.getTargetEl();
        records = me.store.getRange();

        if (!me.preserveScrollOnRefresh) {
            targetParent = targetEl.dom.parentNode;
            targetEl.dom.style.display = 'none';
            targetParent.removeChild(targetEl.dom);
        }

        if (me.refreshCounter) {
            me.clearViewEl();
        } else {
            me.fixedNodes = targetEl.dom.childNodes.length;
            me.refreshCounter = 1;
        }

        me.tpl.append(targetEl, me.collectData(records, 0));

        if (records.length < 1) {
            if (!me.deferEmptyText || me.hasSkippedEmptyText) {
                Ext.  core.DomHelper.insertHtml('beforeEnd',  targetEl.dom,
                    me.emptyText);
            }
            me.all.clear();
        } else {
            me.all.fill(Ext.query(me.getItemSelector(), targetEl.dom));
            me.updateIndexes(0);
        }

        me.selModel.refresh();
        me.hasSkippedEmptyText = true;

        if (!me.preserveScrollOnRefresh) {
            targetParent.appendChild(targetEl.dom);
            targetEl.dom.style.display = '';
        }

        me.fireEvent('refresh', me);

        if (!me.viewReady) {
            me.viewReady = true;
            me.fireEvent('viewready', me);
        }
    },
```

如果组件还没渲染或已被删除，则直接返回，不进行刷新。

接着触发 beforerefresh 事件，调用 getTargetEl 方法返回目标元素，调用 getRange 方法返回记录。

如果 preserveScrollOnRefresh 为 false，表示要维持当前滚动位置，因而可隐藏节点，再调用 removeChild 方法删除子节点，以加速节点操作。

如果 refreshCounter 存在，表示视图已存在数据，则调用 clearViewEl 方法清除视图元素；否则将节点的数量值赋值给 fixedNodes 属性，并设置 refreshCounter 的值为 1。

接着调用模板的 append 方法将模板和数据渲染到页面。

如果记录的长度小于 1，也就是没有记录，则调用 Helper 对象的 insertHtml 方法，在视图内显示空白文本，并调用 clear 方法清理 all 指向的数据集。否则，调用数据集的 fill 方法将 getItemSelector 方法获得的条目写入数据集，也就是说，可以在 all 属性中找到条目，并调用 updateIndexes 方法将记录的 internalId 值赋给条目元素的 viewRecordId 属性。

接着调用选择模型的 refresh，刷新选择模型的选择项。

如果 preserveScrollOnRefresh 为 false，则还要恢复节点显示。

接着触发 refresh 方法。

如果 viewReady 为 false 或不存在，则设置其为 true，并触发 viewReady 事件。

至此，bindStore 方法就执行完毕了。

从以上的代码分析，可以清楚地了解到视图是怎么将 Store 中的数据与 UI 结合到一起的，**也从另一个方面说明了数据与 UI 是分离的，Store 中数据的改变会自动刷新 UI 的显示，而 UI 的选择操作是通过选择模型实现的，这些都是开发中一定要明确的概念，不然就会发生在 UI 中找数据、认为更新了数据就需要手动去改写 UI 这样的错误。切记！切记！**

9.6.3　选择模型的工作流程

上一小节提到了条目使用选择模型来选择，那么选择模型是如何工作的呢？

我们要先研究一下数据模型是怎么创建的，数据模型的基类是 SelectionModel 对象，它扩展自 Observable 对象，其构造函数如下：

```
constructor: function(cfg) {
    var me = this;
    cfg = cfg || {};
    Ext.apply(me, cfg);
    me.addEvents(
        'selectionchange'
    );
    me.modes = {
        SINGLE: true,
        SIMPLE: true,
        MULTI: true
    };
    me.setSelectionMode(cfg.mode || me.mode);
    me.selected = Ext.create('Ext.util.MixedCollection');
    me.callParent(arguments);
},
```

上述代码只添加了 selectionchange 事件，设置了三种选择模式：SINGLE、SIMPLE、MULTI。

接着调用 setSelectionMode 方法设置选择模式，其代码如下：

```
setSelectionMode: function(selMode) {
    selMode = selMode ? selMode.toUpperCase() : 'SINGLE';
    this.selectionMode = this.modes[selMode] ? selMode : 'SINGLE';
},
```

如果在 modes 对象中存在定义的模式，则将其复制给 selectionMode 属性，否则其值为

SINGLE。

回到构造函数，创建一个 MixedCollection 对象实例，并将 selected 属性指向该实例，用来记录被选择的记录。最后是调用 Observable 对象的构造函数。

DataViewModel 和 RowModel 两个选择模型的构造函数只是添加了 beforedeselect、beforeselect、deselect 和 select 这 4 个事件，CellModel 只是添加了 deselect 和 select 两个事件。

在 AbstractView 对象的 afterRender 方法中，会调用 getSelectionModel 方法，再调用选择模型的 bindComponent 方法将视图绑定到选择模型。因为不同视图的条目的组织形式不一样，所以选择模型的 bindComponent 方法也不同，DataViewModel、RowModel、CheckboxModel 和 CheckboxModel 等都重写了 bindComponent 方法，下面将研究一下它们的 bindComponent 方法。

（1）DataViewModel 的 bindComponent

```
bindComponent: function(view) {
    var me = this,
        eventListeners = {
            refresh: me.refresh,
            scope: me
        };
    me.view = view;
    me.bindStore(view.getStore());
    eventListeners[view.triggerEvent] = me.onItemClick;
    eventListeners[view.triggerCtEvent] = me.onContainerClick;
   view.on(eventListeners);

    if (me.enableKeyNav) {
        me.initKeyNav(view);
    }
},
```

这里先要清楚 DataViewModel 对象主要用于 DataView 对象，其选择是基于 HTML 块的。

上述代码先是调用 bindStore 方法为选择模型绑定 Store，接着将视图条目的单击事件、容器单击事件及刷新事件（refresh）绑定到选择模型的方法，这样视图的操作就与选择模型关联起来了。

先来看 onItemClick 方法，其代码如下：

```
onItemClick: function(view, record, item, index, e) {
    this.selectWithEvent(record, e);
},
```

直接奔 selectWithEvent 方法，其代码如下：

```
selectWithEvent: function(record, e, keepExisting) {
    var me = this;

    switch (me.selectionMode) {
        case 'MULTI':
            if (e.ctrlKey && me.isSelected(record)) {
                me.doDeselect(record, false);
```

```
} else if (e.shiftKey && me.lastFocused) {
    me.selectRange(me.lastFocused, record, e.ctrlKey);
} else if (e.ctrlKey) {
    me.doSelect(record, true, false);
} else if (me.isSelected(record) && !e.shiftKey && !e.ctrlKey &&
     me.selected.getCount() > 1) {
    me.doSelect(record, keepExisting, false);
} else {
    me.doSelect(record, false);
}
```

```
                break;
            case 'SIMPLE':
                if (me.isSelected(record)) {
                    me.doDeselect(record);
                } else {
                    me.doSelect(record, true);
                }
                break;
            case 'SINGLE':
                if (me.allowDeselect && me.isSelected(record)) {
                    me.doDeselect(record);
                } else {
                    me.doSelect(record, false);
                }
                break;
        }
    },
```

上述代码会根据选择模式分别处理选择。

在多选模式（MULTI）时，如果键盘 Ctrl 键是按下的且记录已被选择，则调用 doDeselect 方法取消选择记录；如果按下了 Shift 键且最后选择记录存在，则调用 selectRange 方法根据范围选择记录；如果只是按下了 Ctrl 键，则调用 doSelect 方法选择记录；如果记录已选择，但没按下 Shift 键或 Ctrl 键，而且 selected 对象的总数大于 1，则调用 doSelect 方法选择记录。默认情况是调用 doSelect 方法选择记录。

在简单选择模式（SIMPLE）时比较简单，如果记录已被选择，则调用 doDeselect 方法取消选择；如果记录未选择，则调用 doSelect 方法选择记录。

在单选模式（SINGLE）时，如果允许取消选择且记录已被选择，则调用 doDeselect 方法取消选择；其余情况调用 doSelect 方法选择记录。

方法 doSelect 要做的操作就是把选择的记录写到 select 属性指向的记录集中，具体代码大家可以自己研究一下。

再来看看 onContainerClick 方法，其代码如下：

```
onContainerClick: function() {
    if (this.deselectOnContainerClick) {
        this.deselectAll();
    }
},
```

上述代码比较简单，如果运行单击容器取消选择（deselectOnContainerClick 为 true），

则调用 deselectAll 方法取消所有选择。

回到 bindComponent 方法，如果开启了键盘导航（enableKeyNav 为 true），则调用 initKeyNav 方法通过创建 KeyNav 对象实例将键盘的箭头键动作绑定到选择模型的方法，这样就可通过键盘操作视图了。

至此，DataViewModel 对象的 bindComponent 方法就执行完了，它的主要工作就是将视图的 UI 操作与选择模型捆绑在一起，实现 UI 操作与选择模型的联动。

（2）RowModel 对象的 bindComponent 方法

RowModel 对象的 bindComponent 方法代码如下：

```
bindComponent: function(view) {
    var me = this;
    me.views = me.views || [];
    me.views.push(view);
    me.bindStore(view.getStore(), true);
    view.on({
        itemmousedown: me.onRowMouseDown,
        scope: me
    });
    if (me.enableKeyNav) {
        me.initKeyNav(view);
    }
},
```

与 DataViewModel 对象的差不多，区别在绑定的事件，这里只绑定了 itemmousedown 事件，会执行 onRowMouseDown 方法，其代码如下：

```
onRowMouseDown: function(view, record, item, index, e) {
    view.focus(false, Ext.isIE ? 200 : false);
    if (!this.allowRightMouseSelection(e)) {
        return;
    }
    this.selectWithEvent(record, e);
},
```

上述代码先调用视图的 focus 方法使视图的容器获得焦点。

如果不允许单击鼠标右键进行选择，则直接返回。

最后是调用 selectWithEvent 方法选择记录，也就是说，RowModel 对象只要按下鼠标就会进行选择，比 DataView 对象简单。

回到 bindComponent 方法，最后是初始化键盘导航，RowModel 对象的键盘导航使用了 4 个箭头键、PageDown、PageUp、Home 和 End 等 8 个键。

（3）CheckboxModel 对象的 bindComponent 方法

CheckboxModel 对象的 bindComponent 方法代码如下：

```
bindComponent: function(view) {
    var me = this;

    me.sortable = false;
    me.callParent(arguments);
```

```
        if (!me.hasLockedHeader() || view.headerCt.lockedCt) {
            view.headerCt.on('headerclick', me.onHeaderClick, me);
            me.addCheckbox(true);
            me.mon(view.ownerCt, 'reconfigure', me.onReconfigure, me);
        }
    },
```

先调用 RowModel 对象的 bindComponent 方法了，也就是说，复选框选择模型的行为完全遵循行选择模型的行为。

其主要区别有两个。一是添加了单击列标题的事件，已实现了单击列标题全选或者取消选择的操作。二是重写了 onRowMouseDown 方法，代码如下：

```
    onRowMouseDown: function(view, record, item, index, e) {
        view.el.focus();
        var me = this,
            checker = e.getTarget('.' + Ext.baseCSSPrefix + 'grid-row-checker');

        if (!me.allowRightMouseSelection(e)) {
            return;
        }

        if (me.checkOnly && !checker) {
            return;
        }

        if (checker) {
            var mode = me.getSelectionMode();
            if (mode !== 'SINGLE') {
                me.setSelectionMode('SIMPLE');
            }
            me.selectWithEvent(record, e);
            me.setSelectionMode(mode);
        } else {
            me.selectWithEvent(record, e);
        }
    },
```

重写的目的是根据复选框列的样式获取选择行，即粗体这行代码。

（4）CellModel 对象的 bindComponent 方法

CellModel 对象的 bindComponent 方法如下：

```
bindComponent: function(view) {
    var me = this;
    me.primaryView = view;
    me.views = me.views || [];
    me.views.push(view);
    me.bindStore(view.getStore(), true);
    view.on({
        cellmousedown: me.onMouseDown,
        refresh: me.onViewRefresh,
        scope: me
    });
    if (me.enableKeyNav) {
```

```
            me.initKeyNav(view);
        }
    },
```

CellModel 对象与 RowModel 对象的 bindComponent 方法除了绑定事件不同，绑定的键盘按键也不同。它为 cellmousedown 事件绑定了 onMouseDown 方法，其代码如下：

```
onMouseDown: function(view, cell, cellIndex, record, row, rowIndex, e) {
    this.setCurrentPosition({
        row: rowIndex,
        column: cellIndex
    });
},
```

直接调用 setCurrentPosition 方法，其代码如下：

```
setCurrentPosition: function(pos) {
    var me = this;

    if (me.position) {
        me.onCellDeselect(me.position);
    }
    me.position = pos;
    if (pos) {
        me.onCellSelect(pos);
    }

},
```

如果 position 属性存在，说明已经选择了单元格，则调用 onCellDeselect 取消单元格的选择。如果参数 pos 存在，调用 onCellSelect 方法选择单元格。具体选择如何执行，有兴趣可以自己研究一下。

CellModel 对象除了可以使用 4 个箭头键操作，还允许 tab 键进行操作。

（5）TreeModel 对象的 bindComponent 方法

TreeModel 对象没有重写 bindComponent 方法的原因是其选择操作与 RowModel 是一致的，只是左右箭头的作用不是移动位置，而是展开与折叠节点。

通过各选择模型的 bindComponent 方法，可以清楚地了解到选择模型的工作流程，也就是如何进行选择与取消选择。

9.6.4 选择模型的配置项、属性、方法和事件

可以在定义面板或视图的 selModel 配置项中定义选择模型的配置项，其值为一个配置对象，其中配置项 selType 可定义选择模型的类型，然后可根据选择模型的类型加入对应的选择模型配置项。

1. 配置项

由 SelectionModel 对象提供的选择模型公共的配置项：

- allowDeselect：布尔值，该配置项只在单一选择模式时才有效，为 true 时允许取消选

择，默认值为 false。

- mode：设置选择模式，值可以是 SINGLE、SIMPLE 或 MULTI。默认值是 SINGLE。

一般不用设置该配置项，在视图或面板中会有对应的配置项来设置选择模式。

RowModel 对象特有的配置项：

- enableKeyNav：布尔值，默认值为 true，表示开启键盘导航。

CheckboxModel 对象特有的配置项：

- checkOnly：布尔值，为 true 时，只能通过单击 CheckboxModel 所在列进行选择。默认值为 false。
- injectCheckbox：设置选择列的位置，值可以是列的索引值、first 或 last，first 表示在第一列渲染选择列，last 表示在最后一列渲染选择列。默认值是 first，在第一列渲染选择列。

2. 属性

选择模型只有一个属性 selected，它指向一个包含选择记录的 MixedCollection 对象实例。

3. 方法

由 SelectionModel 对象提供的选择模型公共的方法：

- deselect：根据指定的记录或记录索引取消对记录的选择。
- deselectAll：取消所有选择。
- getCount：返回选择的记录总数。
- getLastSelected：返回最后选择的记录。
- getSelection：以数组形式返回选择的记录。
- getSelectionMode：返回当前的选择模式，返回值是 SINGLE、SIMPLE 或 MULTI。
- hasSelection：如果已经有记录被选择，返回 true。
- isFocused：如果指定的记录是当前焦点所在记录，返回 true。
- isLocked：如果选择已被锁定，返回 true。
- isSelected：如果指定的记录或指定索引的记录已选择，返回 true。
- select：根据指定的记录或记录索引选择记录。
- selectAll：选择所有记录。
- selectRange：选择指定范围的记录。
- setLastFocused：将指定记录设置为最后获得焦点的记录。
- setLocked：锁定当前的选择，这可避免在提交更新时用户操作改变选择而造成错误。
- setSelectionMode：设置选择模式。

CheckboxModel 对象特有的方法：

- getHeaderConfig：返回标题栏的配置。
- onHeaderClick：在全部选择与全部取消选择之间进行切换。

4. 事件

选择模型的事件一般都会通过事件传播机制传播到面板或视图，因而一般不需要在选择模型内定义事件监听。

SelectionModel 对象只提供了 selectionchange 一个事件，它在选择发生改变时会触发。DataViewModel 和 RowModel 都提供了 beforeselect、beforedeselect、select 和 deselect 这 4 个事件。而 CellModel 只有 select 和 deselect 这两个事件。其中，beforeselect 事件会在选择发生前触发；beforedeselect 会在取消选择前触发；select 会在选择完成后触发；deselect 会在取消选择完成后触发。

9.6.5 视图的配置项、属性、方法和事件

1. 配置项

- blockRefresh：布尔值，设置为 true 时，会忽略 store 的 datachange 事件。这通常用于通过插件实现自定义转换动画。默认值为 false。
- deferEmptyText：布尔值，为 true 时，Store 第一次加载后才应用空白文本。
- disableSelection：布尔值，为 true 时会禁止视图进行选择。默认值为 false。
- emptyText：没有显示数据时，会显示该配置项定义的文本，默认值为空字符串。
- itemCls：会应用于条目的样式名称，只有在定义了 itemTp! 时才起作用。
- itemSelector：对视图来说，这是必须定义的配置项，视图会根据该选择符选择节点，然后与记录做一个映射。
- itemTpl：条目的模板，视图字段为其包装模型的循环语句。
- loadMask：布尔值或 loadMask 的配置对象。如果为 false，则会禁用遮蔽。默认值为 true。
- loadingCls：应用于遮蔽元素的样式名称。
- loadingHeight：遮蔽的高度，默认值为 undefined。
- loadingText：遮蔽中显示的文本信息。默认值为 undefined。
- multiSelect：布尔值，如果为 true，则允许视图进行多选，默认值为 false。
- overItemCls：鼠标在条目上面时应用到条目的样式名称，默认值为 undefined。
- selectedItemCls：条目选择后应用到条目上的样式名称，默认值为“x-view-selected”。
- simpleSelect：布尔值，如果设置为 true，则不需要按下 Shift 或 Ctrl 键就可以在视图中进行多选。默认值为 false。
- singleSelect：布尔值，如果为 true，视图的选择模式为单一选择模式。设置了 multiSelect 为 true 时，会忽略该配置项。默认值为 false。
- store：必需的配置项，用来定义为视图提供数据的 Store，既可以是 Store 的 id，也可以是 Store 对象。
- tpl：视图用来生成 HTML 代码的模板。如果定义了 itemTpl，则会忽略该配置项。
- trackOver：布尔值，为 true 时会开启 mouserenter 和 mouserleave 事件。

2. 属性

AbstractView 对象和 DataView 对象都没有属性。

3. 方法

- bindStore：重新绑定一个 Store。
- collectData：该方法可被重写，用来为模板重新组织数据。
- deselect：取消选择指定的记录或由索引定位的记录。
- findItemByChild：如果指定节点是模板的节点，则返回节点；否则返回 null。
- findTargetByEvent：根据事件返回模板节点，如果不存在，则返回 null。
- getNode：根据指定的 HTMLElement、节点的索引、节点 id 或者对应的记录返回节点。
- getNodes：返回指定范围的节点。
- getRecord：根据指定的 Element 对象或 HTMLElement 对象返回其对应的记录。
- getRecords：根据数组中的节点返回对应的记录。
- getSelectedNodes：返回选择节点。
- getSelectedRecords：返回选择的记录。
- getSelectionCount：返回选择节点的数量。
- getSelectionModel：返回视图的选择模型。
- getStore：返回视图绑定的 Store。
- indexOf：根据指定的 HTMLElement、节点的索引、节点 id 或者对应的记录返回节点的索引。如果没有找到节点，则返回 –1。
- isSelected：如果指定的记录或指定索引的记录已选择，则返回 true。
- prepareData：可重写该方法，对模板的数据进行预处理。
- refresh：刷新视图。
- refreshNode：根据指定的索引刷新节点数据。
- select：根据指定的记录或记录索引选择记录。
- clearHighlight：取消当前高亮显示的条目。
- highlightItem：高亮显示当前条目。

4. 事件

- beforerefresh：在刷新前会触发该事件。返回 false 可取消刷新。
- beforeitemmousedown：在触发 mousedown（鼠标按下按钮）事件之前会触发该事件。返回 false 可取消触发 mousedown 事件。
- beforeitemmouseup：在触发 mouseup（鼠标按钮松开）事件之前会触发该事件。返回 false 可取消触发 mouseup 事件。
- beforeitemmouseenter：在触发 mouseenter（鼠标进入条目）事件之前会触发该事件。返回 false 可取消触发 mouseenter 事件。
- beforeitemmouseleave：在触发 mouseleave（鼠标离开条目）事件之前会触发该事件。返回 false 可取消触发 mouseleave 事件。
- beforeitemclick：在触发 itemclick（单击条目）事件之前会触发该事件。返回 false 可取消触发 itemclick 事件。

- beforeitemdblclick：在触发 itemdblclick（双击条目）事件之前会触发该事件。返回 false 可取消触发 itemdblclick 事件。
- beforeitemcontextmenu：在触发 itemcontextmenu（单击鼠标右键）事件之前会触发该事件。返回 false 可取消触发 itemcontextmenu 事件。
- beforeitemkeydown：在触发 itemkeydown（按下一个键）事件之前会触发该事件。返回 false 可取消触发 itemkeydown 事件。
- itemadd：当记录添加到 Store 后，会触发该事件。
- itemremove：当记录被移动后，会触发该事件。
- itemupdate：当记录被更新后，会触发该事件。
- itemmousedown：当鼠标按下按钮后会触发该事件。
- itemmouseup：当鼠标松开按钮后会触发该事件。
- itemmouseenter：当鼠标移入条目后会触发该事件。
- itemmouseleave：当鼠标移出条目后会触发该事件。
- itemclick：单击条目后会触发该事件。
- itemdblclick：双击条目后会触发该事件。
- itemcontextmenu：在条目单击鼠标右键后会触发该事件。
- itemkeydown：当鼠标按下按钮后会触发该事件。
- refresh：当视图刷新后，会触发该事件。
- selectionchange：当选择改变后会触发该事件。
- beforeselect：在选择之前会触发该事件，返回 false 可取消选择。

以上没列出容器的事件，其事件与条目（item）的事件一样，但目标是容器。

9.6.6 使用视图

要使用视图必须定义 store、tpl（itemTpl）和 itemSelector 这 3 个配置项。下面通过一个示例来学习一下如何使用视图，图 9-30 是示例最后的效果图。

视图使用的数据在 phone-data.js 文件中，每一条数据包括 id、name、Brands、system、WCDMA、GSM、CDMA、price 和 image 等 9 个字符，以下是其中一条数据：

```
{id:8,name:"诺基亚（NOKIA）C1-02",Brands:"诺基亚",system:"",WCDMA:false,GSM:true,CD
    MA:false,price:200,image:"c1-02.jpg"},
```

使用模板创建一个名称为 9-3.html 的页面文件。

从图 9-30 可以看到，每一个条目只使用了 name、price 和 image 这 3 个字段，而且都是一个字段占一行，因而，模板可以这样定义：

```
    tpl:[
        '<tpl for=".">',
            '<div class="phoneList">',
                '<img width="160" height="160" src="../images/phones/{image}"
/><br/>',
                '<center>{name}</center>',
```

```
            '<center style="color:red">价格: {price:usMoney}</center>',
        '</div>',
    '</tpl>'
],
```

图 9-30　视图示例的效果图

最外层 div 用来包装条目，接下来是一行图片、一行产品的名称、一行价格。现在的重点是 phoneList 的样式，为了实现分列的效果，在 CSS 中可使用 float 属性实现，这与列布局效果一样。现在要计算好 div 的宽度和高度，为了使两个条目之间有点间隔，需要为 div 定义内补丁为 10（这样条目四边就有空格了），图标的宽度为 160（这样 div 的宽度就是 180 了）。图片高度加内补丁的高度也是 180，再加 2 行文字，每行高度 20，div 的高度设置为 220 比较合适。这样就定义好 div 的样式了：

```
.phoneList{float:left;display:block;width:180px;height:220px;padding:10px;}
```

为了加强效果，还需要定义条目选择后和鼠标在其上面时的样式：

```
.selected{background-color: #D3E1F1;}
.overitem{background-color: #E7E7E7;}
```

只是改变一下条目的背景颜色。

工具栏的排序组下的两个按钮根据单击变换图标来表示当前的排序方向，这可通过切换样式来实现，因而需要定义以下两个样式：

```
.asc{background:url(../images/hmenu-asc.gif) no-repeat;}
.desc{background:url(../images/hmenu-desc.gif) no-repeat;}
```

样式 asc 用于升序排序，而 desc 则用于降序排序。

样式确定了，现在可以编写代码，需要使用 Store，所以先为其定义一个模型：

```
Ext.define("Phone",{
    extend: 'Ext.data.Model',
    fields:[
            {name:"id",type:"int"},"name","Brands","system",
            {name:"WCDMA",type:"bool"},{name:"GSM",type:"bool"},
            {name:"CDMA",type:"bool"},{name:"price",type:"float"},"image"

    ]
});
```

模型的字段是根据数据的字段定义而成的，没什么特别的地方。

接着是创建一个 Store：

```
Ext.create("Ext.data.Store",{
    id:'phoneStore',
    model:"Phone",
    autoLoad:true,
    sorters:[
            {property:"name",direction:"ASC"},
            {property:"price",direction:"ASC"}
    ],
     proxy: {
            type: 'ajax',
            url : 'Phone-data.js',
            reader: {
                type: 'json',
                root: ''
            }
        }
    });
```

Store 预设了排序方式，会根据 name 字段升序排序，再在这基础上根据 price 排序。

现在要考虑怎么定义视图。因为视图没有面板上可以放置工具栏的特性，所以需要在其上套一个面板，在面板的顶部放一个工具栏，其主体则显示视图。

```
Ext.create("Ext.panel.Panel",{
    title:"视图示例",
     width:920,
     height:600,
     autoScroll:true,
     renderTo:Ext.getBody(),
    tbar:[{
                    xtype:'buttongroup',
                    title:"网络",
              items:[
                    {text:"GSM",dataIndex:"GSM",enableToggle:true,handler:storeFilter},
                    {text:"WCDMA",dataIndex:"WCDMA",enableToggle:true,handler:storeFilter},
                    {text:"CDMA",dataIndex:"CDMA",enableToggle:true,handler:storeFi
```

```
lter}
                    ]
        },
        {
                xtype:'buttongroup',
                title:" 品牌 ",
              items:[
                {text:" 全部 ",dataIndex:"Brands",toggleGroup:"Brands",pressed:true,handler:s
                    toreFilter},
                {text:" 摩托罗拉 ",dataIndex:"Brands",toggleGroup:"Brands",handler:storeFilter},
                {text:" 诺基亚 ",dataIndex:"Brands",toggleGroup:"Brands",handler:storeFilter},
                {text:"HTC",dataIndex:"Brands",toggleGroup:"Brands",handler:storeFilter},
                {text:" 其他 ",dataIndex:"Brands",toggleGroup:"Brands",handler:storeFilter},
            ]
    },
    {
                xtype:'buttongroup',
                title:" 价格 ",
             items:[
{text:" 全部 ,dataIndex:"price",toggleGroup:"price",pressed:true,handler:storeFilter},
                {text:"1~1000",dataIndex:"price",toggleGroup:"price",handler:storeFilter},
                {text:"1000~2000",dataIndex:"price",toggleGroup:"price",handler:storeFilter},
                {text:"2000~3000",dataIndex:"price",toggleGroup:"price",handler:storeFilter},
                {text:"3000 以上 ",dataIndex:"price",toggleGroup:"price",handler:storeFilter}
            ]
    },
    {
                xtype:'buttongroup',
                title:" 排序 ",
              items:[
                {text:" 名称 ",iconCls:"asc",dataIndex:"name",handler:function(){
                        var p=this.up("buttongroup").up("panel");
                        p.storeSort(this);
                }},
                {text:" 价格 ",iconCls:"asc",dataIndex:"price",handler:function(){
                        var p=this.up("buttongroup").up("panel");
                        p.storeSort(this);
                }},
            ]
    }]
})
```

面板的宽度计算时一定要把滚动条的宽度加上，不然出现滚动条后，一行就只能显示 4 列了。要出现滚动条，必须定义 autoScroll 为 true，将模板主体样式属性 overflow 的值由 hidden 修改为 auto。

在按钮中都加入了自定义的 dataIndex 属性，用来指示其要操作哪个字段。按钮的句柄分了两种方式写，storeFilter 不是面板的成员函数，而 storeSort 是面板的成员函数。这样做的目的是想告诉大家，有时候作为成员函数的书写方式在按钮多且都是调用同一函数的时候，代码会很繁琐，写成员函数的目的是便于对象的引用，但是要具体情况具体分析，灵活运用。**而最好的方式是将面板定义为面板的扩展，这样就可以完全使用成员函数方式了，Ext JS 4 的开发架构也建议这样做，便于维护。**

视图还没定义，在面板里加一个 items 配置项，开始定义模板：

```
items:[{
    xtype:"dataview",
     store:"phoneStore",
     multiSelect:true,
     selectedItemCls:"selected",
     itemSelector: 'div.phoneList',
     overItemCls:"overitem",
     trackOver:true,
    // 省略模板定义代码，前面代码已有，
    listeners:{
            selectionchange:function(view,recs){
                    console.log(recs);
            }
    }
}],
```

视图定义了 multiSelect 为 true，因而需要与 Shift 或 Ctrl 键配合才能进行多项选择。通过模板可以知道根据选择符“div.phoneList”就能找到所有条目，所以可定义它为 itemSelector 的值。而选择和鼠标移动到条目上的样式定义，直接使用样式名称就可以了。要实现鼠标移入或移出条目的效果，必须设置 trackOver 为 true，不然 overItemCls 的设置就会被忽略。

在视图中，还监听了 selectionchange 事件，将在控制台显示选择的记录。

现在要考虑怎么实现数据的过滤与排序，从第 7 章可以知道，数据的过滤与排序是在 Store 中进行的，与 UI 没有关系，因而按钮的动作只需要处理 Store，而不需要对视图进行任何处理。

排序的思路比较简单，单击按钮后，修改按钮的样式，然后根据新的状态构建新的排序对象，调用 Store 的 sort 方法就可以，以下是其代码：

```
storeSort:function(btn){
    var store=this.down("dataview").store;
    var dir=Ext.String.toggle(btn.iconCls, "asc", "desc");
    btn.setIconCls(dir);
    var sort=[];
    sort.push({property:btn.dataIndex,direction:btn.iconCls.toUpperCase()});

    var b2=btn.previousSibling("button");
    if(!b2) b2=btn.nextSibling("button");
    sort.push({property:b2.dataIndex,direction:b2.iconCls.toUpperCase()});
    store.sort(sort);
}
```

方法 storeSort 是面板的成员函数，因而需要在面板内定义。上述代码先从视图中取得 Store，然后根据按钮当前的样式做一个字符串切换，再使用 setIconCls 方法改变其样式，就可实现按钮切换图标了。

接着要做的是构建排序数组了，首先使用传递过来的按钮生成修改后的排序对象并将其加入数组。接着利用 Ext JS 4 提供的获取相邻组件的新方法——previousSibling 和 nextSibling，

就可找到按钮的前一个按钮或后一个按钮，这样可构建余下的排序对象并加入数组了。最后是调用 sort 方法。

接下来要考虑怎么使用按钮过滤数据。首先可以确定是使用 Filter 对象本身的配置项 property 和 value 来实现过滤的，对于价格范围，无法使用大于、等于、小于这些条件，而只能通过生成比较函数的方式。要实现网络、品牌、价格的复合查询，只要把这些查询函数组合成数组传递给 filter 方法，它就会逐个调用数组里的比较函数，全部符合时才会返回 true，这样就不用把不同字段的比较复合到一个函数里，可简化处理。

方法明确后，现在最大的问题是如何获取网络、品牌、价格中起作用的按钮，有了 ComponentQuery 对象，这也变得轻松多了，查询面板中的按钮，其 pressed 属性为 true 就是了，因为对于切换按钮，本示例约定了只有在按下的时候才过滤数据。

以下是 storeFilter 的代码：

```
var storeFilter=function(){
   var store=Ext.StoreManager.lookup("phoneStore"),
   cq=Ext.ComponentQuery,
   btns=cq.query("button[pressed=true]"),
   net=["GSM","CDMA","WCDMA"],
   filter=[];
    for(var i=btns.length-1;i>=0;i--){
          btn=btns[i];
          if(btn.text!=" 全部 "){
           if(net.indexOf(btn.dataIndex)!=-1){
             var p=btn.dataIndex;
            filter.push({filterFn:function(record){return record.get(p)}});
           }else if(btn.dataIndex=="Brands"){
   if(btn.text==" 其他 "){
          filter.push(function(record){
                 var other=[" 摩托罗拉 "," 诺基亚 ","HTC"];
          return other.indexOf(record.get("Brands"))==-1}
   );
   }else{
   var text=btn.text;
filter.push({filterFn:function(record){return record.get("Brands")==text}});
}
}else if(btn.dataIndex=="price"){
   var range=btn.text.split("~");
   if(!range[1]){
          range[0]=3000;
          range[1]=Number.MAX_VALUE;
   };
          filter.push({filterFn:function(record){
                 var price=record.get("price")
                 return price>=range[0] && price<range[1];
          }});
      }
    }
}
if(filter.length>0){
       store.suspendEvents();
       store.clearFilter();
```

```
                store.resumeEvents();
                store.filter(filter);
        }else{
                store.clearFilter();
        }
}
```

使用 ComponentQuery 对象的 query 方法获取按钮后，就可以在循环中根据按钮组织过滤函数了。

如果按钮的文本是“全部”，则不需要过滤，不用做任何处理，直接跳过就行。如果不是全部，则先检查是否是网络组的按钮，如果是，因为这些字段都是布尔值，所以直接根据其值就可进行过滤；如果是品牌组且“其他”按钮按下了，则需要检查 Brands 字段是否包含在已固定品牌的数组中即可进行过滤，已固定品牌的，将其 Brands 字段值与按钮文本比较即可；如果是价格组，先将按钮文本拆分一下，如果文本能拆分成两个元素的数组，说明数组第 1 个数字是范围的开始值，第 2 个数字是范围的结束值，如果不能拆分，说明文本为“3000 以上”，需要修改开始值为 3000，结束值为数组的最大值，然后将比较函数加入数组。

过滤数组完成后，如果长度大于 0，表示存在过滤，为了不在清理查询时刷新一次显示，在执行查询后又刷新一次显示，要在清理查询时先中断事件的执行，等清理完成后再允许执行事件。如果过滤数组为空数组，则说明没有过滤条件，直接清理过滤就行了。

打开页面 9-3 后将看到如图 9-30 所示的效果，单击 GSM、摩托罗拉、3000 以上和名称等按钮后，将看到如图 9-31 所示的结果。

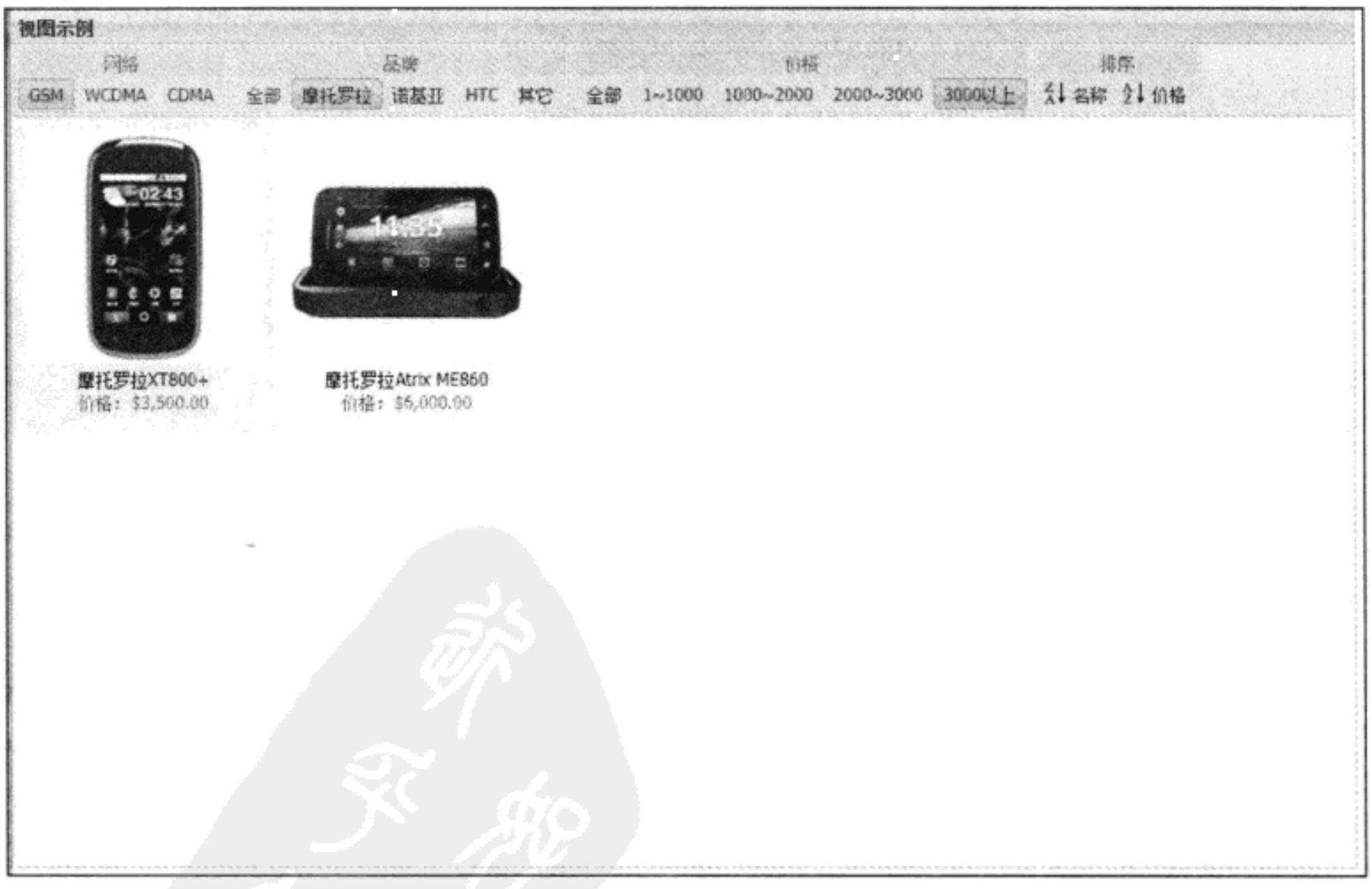


图 9-31　单击 GSM、摩托罗拉、3000 以上和名称等按钮后的示例效果

9.7　页面布局设计

熟悉 Web 开发的人对页面布局设计应该不陌生，通过搜索引擎可以找到许多例子，但是如果要使用 Ext JS 的组件对页面进行布局，有些初学者就无从下手了。其实，Ext JS 的布局与 CSS 的标准布局方式没有太大区别，只不过其布局的元素是由定义的脚本生成而已，譬如图 9-18 由边框布局生成的页面，就是我们常用的三行三列布局，如果把 east 或 west 区域去掉，就是三行两列的布局，所以，从边框布局可以演化出很多我们常用的布局。因此，本节的讨论重点不是这些简单的布局，而是比较复杂的布局。

要设计比较复杂的布局，可以按照以下原则进行设计：

1）不要在潜意识里认为必须在一层内实现布局。布局是可以嵌套的，如果一层布局解决不了问题，那么就要嵌套一层，如果还不行，那么就继续嵌套，直到完全划分好所有区域。

2）如果布局需要嵌套，在 4.1 之前版本的原则是把等宽或等高的布局先合并成一个大的布局，尽量合并到可套用边框布局。这样的好处就是使用边框布局划分完第一层区域后，可对这些区域使用 HBox 或 Vbox 布局进行布局。例如图 9-32，顶部两个布局就可以合并为一个大的布局作为边框布局的 north 部分，底部的也可以这样处理，而中间的 4 个布局，可以根据情况将任意两个合并为一个大布局。这样就可以直接套用边框布局先整体实现整个布局，接着在 north 和 south 两个区域再使用 VBox 布局将其划分为两个布局，而中间合并了两个布局的大布局可使用 HBox 布局将其划分为两个布局。利用该原则可大大简化布局。版本 4.1 之后图 9-32 的布局，不需要嵌套就可以一次成形了，这是相当不错的改进，简化了布局的设计。

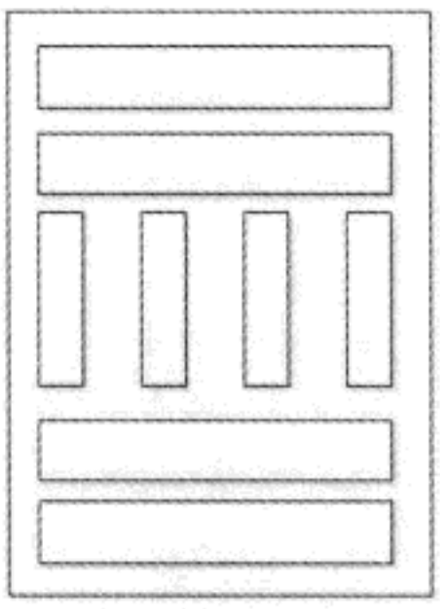


图 9-32　合并布局

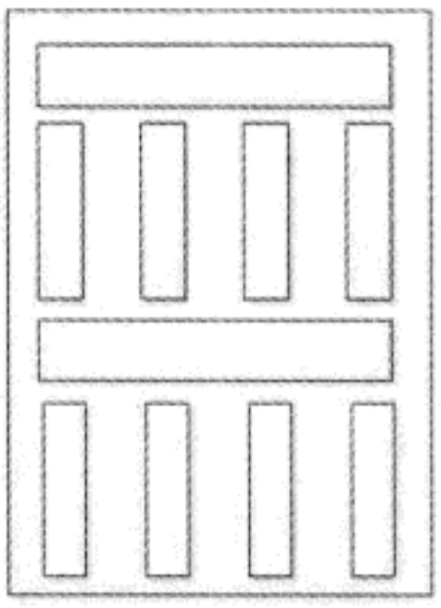


图 9-33　不能组合出边框布局的布局

3）如果第 2 个原则还不能组合出边框布局，如图 9-33 所示的布局，那就先将布局通过合并构建一个可以充分利用边框布局的布局，再在某个区域内使用边框布局。图 9-33 的布局，可以先将 3、4 行合并，然后在 south 区域再使用边框布局或使用 VBox 布局后再用 HBox 布局进行布局。如果将 2、3 行合并，那么就不能重复利用边框布局的区域了，因为在 south 区域，不能直接划分出 east 或 west 区域，这样 center 和 south 区域都要再次使用边框布局。不过，这要根据区域之间是否可调整大小来灵活划分。使用边框布局的目的就是可为各区域灵活配置分隔条，以调整区域大小。

4）如果是不规则布局，先将不规则的布局通过横向或竖向合并变成规则布局，再重复第1到3步。

5）如果布局经过第4步的处理后还是不规则的，那么可考虑使用表格布局或者绝对定位布局划分布局。布局中要调整大小的区域，基本是等宽或等高的，所以可按第2步处理。

根据以上原则基本可以划分出所需要的布局，但需要灵活使用这些原则。布局千变万化，但又万变不离其宗，因而，笔者认为布局都是试出来的，一个布局可以有多种实现方法，只有经过尝试，才知道哪种比较好。

9.8 综合实例

9.8.1 布局设计实例：仿 Eclipse 界面

图 9-34 是 Eclipse 的界面，从界面可以知道，这是一个不太规则的布局，只使用一次边框布局是不能构建出界面的，因而需要进行合并。

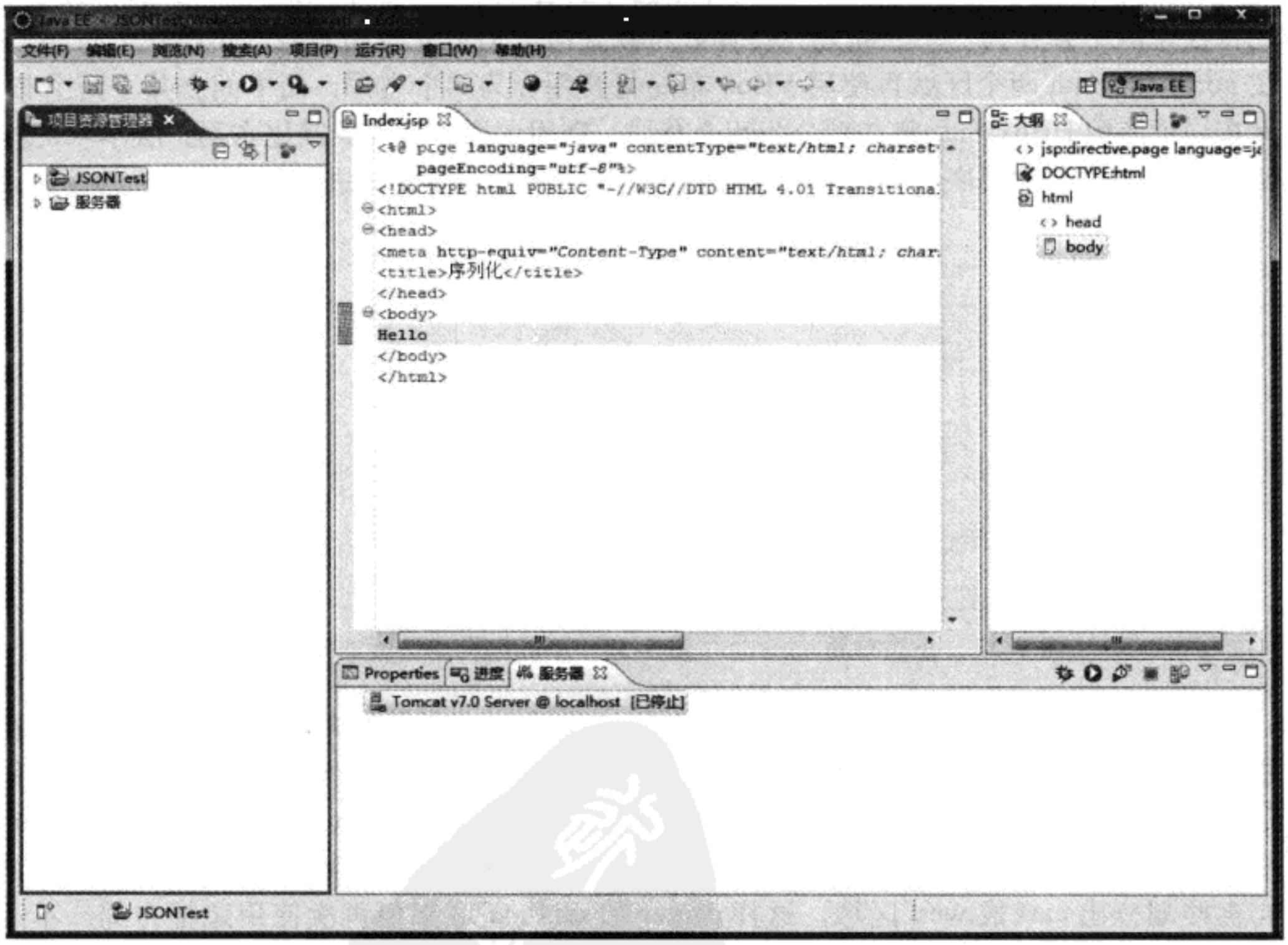


图 9-34 Eclipse 的界面

在界面中，最能充分利用边框布局的情形是将顶部的菜单和工具栏作为 north 区域，底部

工具栏作为 south 区域，中间部分可将资源管理器作为 west 区域，余下部分则作为 center 区域，在 center 区域再使用一次边框布局就可完全构建出整个界面了。

目标明确了，现在可以动手创建布局。使用模板页创建一个名称为 9-4.html 的页面文件，在 OnReady 函数内首先创建一个 Viewport 对象，使用 BorderLayout 将其划分出相应的区域：

```
Ext.create("Ext.Viewport",{
    layout:"border",
    items:[
            // 区域定义
    ]
})
```

这里不要尝试在 Viewport 里定义应用的标题，从 9.2.3 节可以知道，Viewport 只是将 body 作为容器，不会生成任何 HTML 元素，因而在这里定义标题是没有任何作用的，标题需下放到 north 区域定义。

下面来定义 north 区域，它包含 3 部分：一个标题栏、一个菜单栏和一个工具栏，标题栏可以用 title 配置项定义，菜单栏和工具栏可使用 tbar 与 bbar 结合，也可以使用 dockedItems 定义。north 区域最重要的是设置好高度，能刚好显示标题栏、菜单栏和工具栏就可以了，主体部分不需要分配，菜单栏与工具栏的高度大约是 28，而标题栏的高度大约是 25，因而整个 north 区域的高度可以先设置为 81，然后进行微调。以下是 north 区域的定义代码：

```
{region:"north",height:81,title:" 布局设计示例一：仿 Eclipse 界面 ",
    dockedItems:[
            {xtype:"toolbar",dock:"top",items:[
                    {xtype:"splitbutton",text:" 文件 ",menu:[
                            {text:" 打开 "},{text:" 关闭 "}
                    ]}
            ]},
            {xtype:"toolbar",dock:"top",items:[
                    {text:" 工具栏 "}
            ]}
    ]
},
```

区域 south 与 north 一样，重点在高度，基本上没什么其他配置，其代码如下：

```
{region:"south",height:28,
    tbar:[
            {text:" 底部工具栏 "}
    ]
}
```

区域 west 的定义就简单多了，它要使用标签面板，而且与 center 区域是可调整大小的，因而其代码定义如下：

```
{region:"west",
    xtype:"tabpanel",
    split:true,
    width:200,
    minWidth:100,
```

```
        items:[
                {title:"项目资源管理器",closable:true}
        ]
},
```

区域 center 还需要使用一次 BorderLayout 将其划分成 3 个部分，而且每个部分都是使用标签面板，各面板之间允许调整大小，因而可定义如下：

```
{region:"center",layout:'border',
        items:[
                {region:"center",xtype:"tabpanel",items:[
                        {title:"Index.jsp",closable:true}
                ]},
                {region:"east",split:true,width:200,minWidth:100,xtype:"tabpanel",items:[
                        {title:"大纲",closable:true}
                ]},
                {region:"south",xtype:"tabpanel",split:true,height:200,minHeight:100,items:[
                        {title:"属性",closable:true},
                        {title:"进度",closable:true},
                        {title:"服务器",closable:true}
                ]}
        ]
},
```

注意加分隔条的位置，不能加到 center 位置，因为这里 center 不清楚分隔条是使用在哪个方向的。

这样，一个仿 Eclipse 界面的布局就构建完成了，在浏览器打开页面，将看到如图 9-35 所示的效果。在 north 区域与下面区域之间有比较深色的线条，可通过微调 north 区域的高度来去掉。在 center 区域也有一条比较深的线条，这是 center 的边框造成的，在 center 区域中定义 border 配置项为 false 就可以取消。

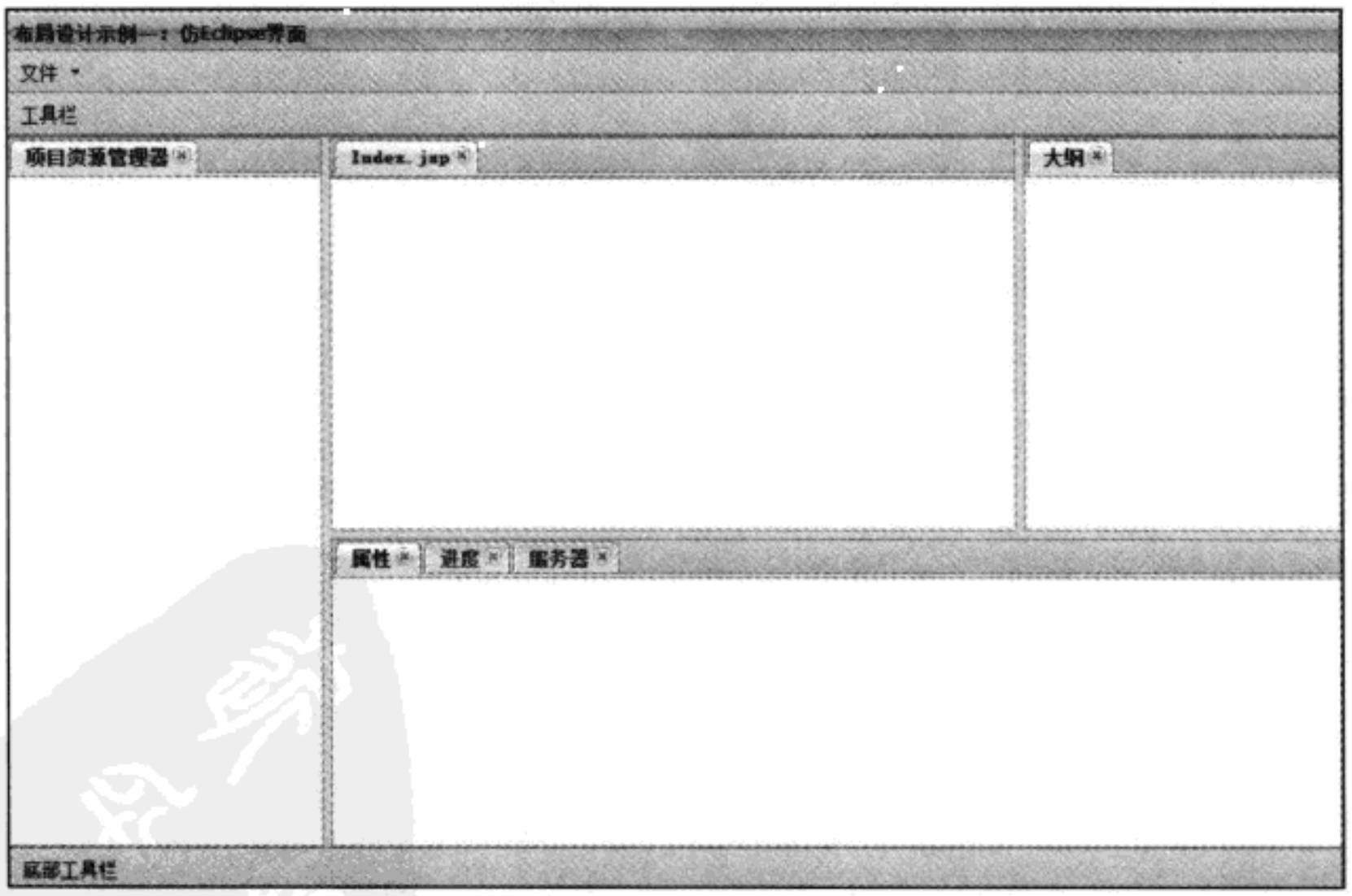


图 9-35　仿 Eclipse 界面的布局效果

以上是 4.1 之前版本需要通过嵌套实现布局，使用 4.1 版本不需要嵌套一次就可成形了，关键是控制好每个区域的权重，以下是 4.1 版本的主要实现代码（9-4-4.1.html）：

```
Ext.create("Ext.Viewport",{
    layout:"border",
    items:[
            // 区域定义
            {region:"north",height:79,title:" 布局设计示例一：仿 Eclipse 界面 ",
                    dockedItems:[
                            {xtype:"toolbar",dock:"top",items:[
                                    {xtype:"splitbutton",text:" 文件 ",menu:[
                                            {text:" 打开 "},{text:" 关闭 "}
                                    ]}
                            ]},
                            {xtype:"toolbar",dock:"top",items:[
                                    {text:" 工具栏 "}
                            ]}
                    ]
            },
            {region:"west",
                    xtype:"tabpanel",
                    split:true,
                    width:200,
                    minWidth:100,
                    items:[
                            {title:" 项目资源管理器 ",closable:true}
                    ]
            },
            {region:"center",border:false,xtype:"tabpanel",
                    items:[
                            {title:"Index.jsp",closable:true}
                    ]
            },
            {region:"east",split:true,width:200,minWidth:100,xtype:"tabpanel",weigh
                t:-60,items:[
                    {title:" 大纲 ",closable:true}
            ]},
            {region:"south",xtype:"tabpanel",split:true,height:200,minHeight:100,we
                ight:-50,items:[
                    {title:" 属性 ",closable:true},
                    {title:" 进度 ",closable:true},
                    {title:" 服务器 ",closable:true}
            ]},
            {region:"south",height:28,
                    tbar:[
                            {text:" 底部工具栏 "}
                    ]
            }
    ]
})
```

完全不需要嵌套，只是在“大纲”区域中加入了权重（weight）的定义，其值为 –60，比其他区域的权重都小，所以最后确定其位置。而“属性”区域的权重为 –50，比“项目管理器”的权重小，所以它会显示在“项目管理器”区域右边。

9.8.2 在单页面应用中使用卡片布局实现“页面”切换

（1）功能描述

通过卡片布局实现“内容页”的切换，如果“内容页”还没加载，则进行远程加载。

（2）实现代码

使用模板页创建一个名称为 9-5.html 的页面文件，先定义好页面的整体布局，首先是将布局使用边框布局分成 north、west 和 center 这 3 个区域。其中，north 区域用来显示标题。west 区域使用手风琴布局，包括产品管理和系统管理两个面板，产品管理内使用树面板显示产品管理和统计管理两个节点；系统管理内使用树面板显示用户管理和系统设置两个节点，单击这些节点将切换到 center 区域内容面板。center 区域默认显示产品管理内容面板。

现在的问题是，如何将节点与其对应的脚本文件以及内容面板对应，通常的做法是在节点上定义一个附加字段，然后根据附加字段的值和约定好的规则来关联脚本文件和内容面板。不过，Ext JS 4 的树不能简单地附加字段，要附加字段必须先修改模型，对于一个简单的示例，这有点复杂。因而这里将利用节点的 id 作为附加字段，id 的值加上“.js”就是脚本文件名称，加上“Content”就是面板的 id 值，这样可大大简化工作。

以下是定义好的布局代码：

```
Ext.create("Ext.Viewport",{
    layout:{type:"border",padding:5},
    items:[
    // 区域定义
    {xtype:"container",region:"north",height:30,
        html:"<h1>示例 9-5 单页面应用中使用 Card 实现“页面”切换 </h1>"
},
{region:"west",split:true,width:200,minWidth:100,
        layout:"accordion",
        items:[
                {title:" 产品管理 ",xtype:"treepanel",
                rootVisible: false,
                    root: {
                        text: 'root',id: 'rootProduct',
                        expanded: true,children:[
                        {text:" 产品管理 ",id:"Product",leaf:true},
                        {text:" 统计管理 ",id:"Stat",leaf:true}
                        ]
                        },
                        listeners:{itemclick:itemclick}
                },
                {title:" 系统管理 ",xtype:"treepanel",
rootVisible: false,
    root: {
        text: 'root',id: 'rootSyetem',
expanded: true,children:[
        {text:" 用户管理 ",id:"User",leaf:true},
        {text:" 系统设置 ",id:"System",leaf:true}
        ]
        },
        listeners:{itemclick:itemclick}
```

```
            }
            ]
},
{region:"center",layout:'card',border:false,
            id:"ContentPage",loader:true,
            items:[
            {title:"产品管理",id:"ProductContent",tbar:[
                        {text:"增加"},{text:"编辑"},{text:"删除"}
            ]}
            ],
            listeners:{
                        add:function(cmp,con,pos){
                                    if(this.items.length>1){
                                    this.getLayout().setActiveItem(pos);
                                    }
                        }
                        }
                  }
      ]
})
```

上述代码中，树节点都定义了 id 字段。而初始产品管理面板也根据规则定义了 id 配置项。因为树节点的单击操作相同，因而都调用了 itemclick 函数。在 center 区域绑定了 add 事件的作用是使用远程加载内容面板，在其添加到容器后激活它。因为在初始化显示产品管理面板的时候也会触发该方法，所以需要加判断，只有当面板数多于一个时才激活面板。add 事件会将新添加的面板的索引作为参数传递到触发函数，因而可使用该索引激活面板。

另外，还要为 center 区域定义配置项 loader 为 true，它才会去创建 ComponentLoader 对象，从而动态加载组件。

现在要完成 itemclick 函数，函数首先要做的就是根据节点的 id 去组件中找对应的内容面板，如果没有找到，说明内容面板还没有加载，需要进行加载。如果内容面板已经存在，就可以直接调用布局的 setActiveItem 方法激活它。以下是其代码：

```
var itemclick=function(view,node){
   var cmp=Ext.getCmp(node.data.id+"Content");
   var contentPage=Ext.getCmp("ContentPage");
   if(cmp){
            contentPage.getLayout().setActiveItem(cmp);
   }else{
            contentPage.getLoader().load({
                     url:node.data.id+".js",
                     renderer:"component"
            });
   }
}
```

主页面已经完成了，余下就是统计管理、用户管理和系统设置这 3 个面板的脚本文件了。先创建统计管理的脚本文件 stat.js，其代码如下：

```
{
   title:"统计",
```

```
    id:"StatContent",
    tbar:[
            "统计日期：从",
            {xtype:"datefield"},
            "到",
            {xtype:"datefield"}
    ]
}
```

正规的做法是使用 Ext.define 定义一个单件模式的类，不过问题不大，上面的做法也行。

接着创建 user.js，代码如下：

```
{
    title:"用户管理",
    id:"UserContent",
    layout:{type:"vbox",align:"stretch"},
    items:[
            {title:"用户列表",flex:1,width:"100%"},
            {title:"编辑用户",flex:1,width:"100%"}
    ]
}
```

最后是创建 syste.js，其代码如下：

```
{
    title:"系统设置",
    id:"SystemContent",
}
```

（3）页面效果

在浏览器中打开页面，并依次单击统计管理、用户管理和系统设置，将看到如图 9-36 所示的结果。注意控制台，会有 3 个加载请求。

如果是使用标签面板的，实现原理也一样。不过要注意，标签面板如果是允许关闭的，就需要使用 9.5.4 节介绍的方法创建一个可重用的对象，而且不能直接使用面板加载组件，必须自己使用 Ajax 加载后，再将克隆对象添加到面板内。

如果不是使用单页面应用，则可以在内容面板中使用的配置项 html 内加入 iframe 的 HTML 代码，例如：

```
html:'<iframe src="目标文件" width="100%" height="100%" frameBorder="0"></iframe>'
```

这时候，就不需要动态加载脚本了，只需要创建一个包含 iframe 的面板，根据树节点的 id 或者附加字段提供的值，为 iframe 的 src 属性提供地址即可。内容面板切换依然可以使用本示例中的切换方法。

9.9 本章小结

容器、面板、布局和视图是使用 Ext JS 开发应用必然要用到的组件，而且是构建界面的主要元素，它们的区别、使用方法、应用场合都必须熟悉。俗语说，熟能生巧，熟悉了才能

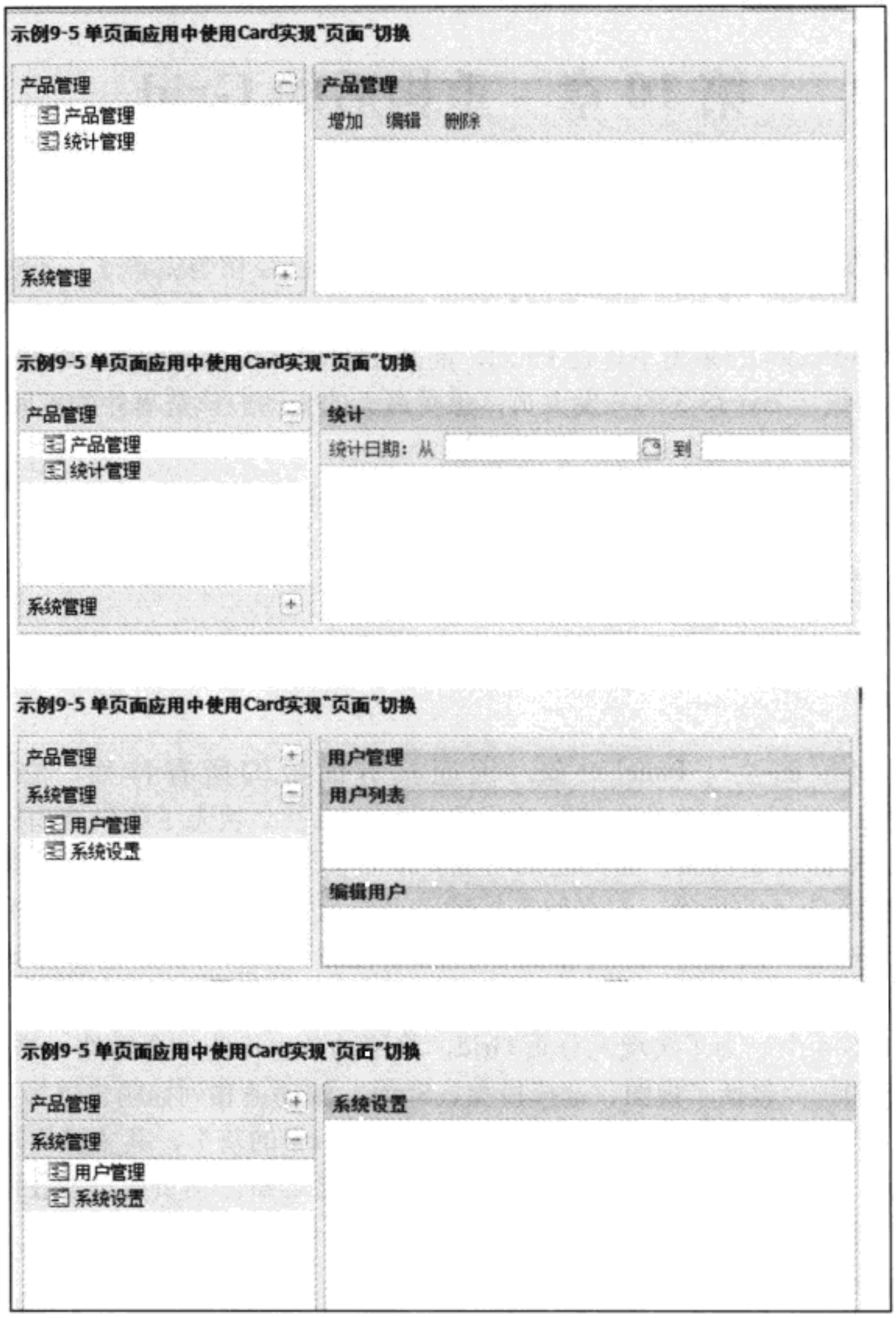


图 9-36　单页面应用中使用卡片布局实现“页面”切换示例的效果

灵活运用这些组件构建出用户体验好的应用界面。

要熟练掌握和使用容器、面板、布局和视图，就要多看 API 文档，多研习 Ext JS 包中的示例，看通看透那些示例，才能快速进步。

第 10 章　重构后的 Grid

Grid 一直是 Ext JS 的焦点所在，在 Ext JS 3 中功能已经相当完善了，不过其性能问题比较突出。为了改善性能、增加更多的功能和让它更灵活，在 Ext JS 4 中对其进行了重构。重构后的 Grid 中，GridPanel 的基类不再是 Panel，而是 TablePanel，Gridview 的基类也相应的修改为 TableView。习惯了 Ext JS 3 的开发人员，感受最大的变动应该是事件的不同，之前的 Click 事件现在是 itemclick 了，这个还需要慢慢适应。

本章将从 TablePanel 对象和 TableView 对象开始，逐步介绍 Grid 的构成及其使用方法。

10.1　Grid 的基类及其构成

10.1.1　概述

TablePanel 对象派生于 Panel 对象，因而具有面板的所有特性。它是 GridPanel 和 TreePanel 的基类，因而，要熟悉和掌握 Grid 和树的构成就必须先了解表格面板的构成。首先要明确的是，表格面板是面板，因而它的功能就是负责管理其内部的组件，那么它内部会有什么组件呢？根据 Grid 的用途，它必然有视图用来显示数据，而有视图就必然有选择模型和 Store。Grid 有列标题，因而需要一个放置列标题的容器。在 Ext JS 3 中，Grid 的滚动条是一种自然状态的滚动条，就是当视图高度或宽度超出容器范围时，会自动显示，可以说是不可控的。而在 Ext JS 4 中，为了实现无分页 Grid，将滚动条也作为一个组件，这样就可管理了。所以，表格面板的构件包括了视图、选择模型、Store、滚动条和列标题容器等 5 个部件。

TableView 派生于 DataView，是 GridView 和 TreeView 的基类，其主要作用是以表格形式显示数据。这个一定要和 TablePanel 区分开来，面板只是容器，不负责显示数据。因而，如果数据显示错误，很大部分是视图出了问题，而视图的数据是从 Store 中来的。清楚这些关系，对排错是非常有帮助的。

10.1.2　表格面板的运行流程：Ext.panel.Table

TablePanel 对象是面板类对象，因而它不仅会遵循面板的工作流程，也会有自己特有的流程，其 initComponent 方法代码如下：

```
initComponent: function() {
    // 省略调试代码

    var me          = this,
        scroll      = me.scroll,
```

```
        vertical     = false,
        horizontal   = false,
        headerCtCfg = me.columns || me.colModel,
        view,
        border = me.border;

    if (me.hideHeaders) {
        border = false;
    }

    if (me.columnLines) {
        me.addCls(Ext.baseCSSPrefix + 'grid-with-col-lines');
    }

    if (me.rowLines) {
        me.addCls(Ext.baseCSSPrefix + 'grid-with-row-lines');
    }

    me.store = Ext.data.StoreManager.lookup(me.store || 'ext-empty-store');

    if (headerCtCfg instanceof Ext.grid.header.Container) {
        me.headerCt = headerCtCfg;
        me.headerCt.border = border;
        me.columns = me.headerCt.items.items;
    } else {
        if (Ext.isArray(headerCtCfg)) {
            headerCtCfg = {
                items: headerCtCfg,
                border: border
            };
        }
        Ext.apply(headerCtCfg, {
            forceFit: me.forceFit,
            sortable: me.sortableColumns,
            enableColumnMove: me.enableColumnMove,
            enableColumnResize: me.enableColumnResize,
            enableColumnHide: me.enableColumnHide,
            border:  border,
            restrictReorder: me.restrictColumnReorder
        });
        me.columns = headerCtCfg.items;

         if (me.enableLocking || Ext.ComponentQuery.query('{locked !== undefined}
             {processed != true}', me.columns).length) {
             me.self.mixin('lockable', Ext.grid.Lockable);
             me.injectLockable();
         }
    }

    me.addEvents(
            // 省略事件代码
    );

    me.bodyCls = me.bodyCls || '';
    me.bodyCls += (' ' + me.extraBodyCls);
```

```
me.cls = me.cls || '';
me.cls += (' ' + me.extraBaseCls);

delete me.autoScroll;

if (!me.hasView) {
    if (!me.headerCt) {
        me.headerCt = new Ext.grid.header.Container(headerCtCfg);
    }
    me.columns = me.headerCt.items.items;
    if (me.store.buffered && !me.store.remoteSort) {
        for (var i = 0, len = me.columns.length; i < len; i++) {
            me.columns[i].sortable = false;
        }
    }

    if (me.hideHeaders) {
        me.headerCt.height = 0;
        me.headerCt.border = false;
        me.headerCt.addCls(Ext.baseCSSPrefix + 'grid-header-ct-hidden');
        me.addCls(Ext.baseCSSPrefix + 'grid-header-hidden');
        if (Ext.isIEQuirks) {
            me.headerCt.style = {
                display: 'none'
            };
        }
    }

    if (scroll === true || scroll === 'both') {
        vertical = horizontal = true;
    } else if (scroll === 'horizontal') {
        horizontal = true;
    } else if (scroll === 'vertical') {
        vertical = true;
    }

    me.relayHeaderCtEvents(me.headerCt);
    me.features = me.features || [];
    if (!Ext.isArray(me.features)) {
        me.features = [me.features];
    }
    me.dockedItems = me.dockedItems || [];
    me.dockedItems.unshift(me.headerCt);
    me.viewConfig = me.viewConfig || {};

    if (me.store && me.store.buffered) {
        me.viewConfig.preserveScrollOnRefresh = true;
    } else if (me.invalidateScrollerOnRefresh !== undefined) {
        me.viewConfig.preserveScrollOnRefresh = !me.invalidateScro —
            llerOnRefresh;
    }

    view = me.getView();
```

```
me.items = [view];
me.hasView = true;

if (vertical) {
    if (me.store.buffered) {
        me.verticalScroller = new Ext.grid.PagingScroller(Ext.apply({
            panel: me,
            store: me.store,
            view: me.view
        }, me.verticalScroller));
    }
}

if (horizontal) {
    if (!me.hideHeaders) {
        view.on({
            scroll: {
                fn: me.onHorizontalScroll,
                element: 'el',
                scope: me
            }
        });
    }
}

me.mon(view.store, {
    load: me.onStoreLoad,
    scope: me
});
me.mon(view, {
    viewready: me.onViewReady,
    refresh: me.onViewRefresh,
    scope: me
});

this.relayEvents(me.view, [
      // 省略事件代码
]);

me.callParent(arguments);
},
```

如果 hideheaders 为 true（隐藏标题栏），则设置 border 为 false；如果 columnLines 为 true（显示列的框线），则调用 addCls 方法增加样式；如果 rowLines 为 true（显示行的框线），则调用 addCls 方法增加样式。

接着将 store 属性指向从 StoreManager 获取的 Store。

如果 headerCtCfg 是标题栏容器的实例，那么不需要再做处理，将 headerCt 属性指向实例；将标题栏边框值设置为配置项定义的值；将属性 columns 指向标题栏容器的 items 属性的值。

如果不是标题栏容器的实例，且定义为数组，则将其转换为包含 items 和 border 两个成员的对象；调用 apply 方法将默认配置项复制到对象，这里，主要是将标题栏的一些操作与表格

面板内的方法关联起来，实现交互操作；将 headerCtCfg 对象中的 items 属性值赋值给 columns 属性；如果开启了锁定功能，**则要混入 Lockable 对象，并调用 injectLockable 方法创建两个表格，一个用于显示锁定数据，一个用于显示活动数据**。

为对象添加事件和一些样式后，如果视图还没创建（hasView 为 false），就开始创建视图。首先检查标题栏是否已经创建，若还没有，则创建标题栏；设置 columns 属性；如果 Store 对象不分页且不进行远程排序，则设置每列的 sortable 属性都为 false；如果标题栏是隐藏的，则设置标题栏的高度为 0，不显示边框，并添加隐藏样式；根据配置项设置滚动条属性；调用 relayHeaderCtEvents 方法将标题栏的事件传播到面板；设置好附加功能（features）、工具栏（dockedItems）、视图配置项（viewConfig）和 Store 后，调用 getView 方法创建视图，其代码如下：

```
getView: function() {
    var me = this,
        sm;

    if (!me.view) {
        sm = me.getSelectionModel();
        me.view = Ext.widget(Ext.apply({}, me.viewConfig, {
            deferInitialRefresh: me.deferRowRender !== false,
            scroll: me.scroll,
            xtype: me.viewType,
            store: me.store,
            headerCt: me.headerCt,
            selModel: sm,
            features: me.features,
            panel: me
        }));
        me.mon(me.view, {
            uievent: me.processEvent,
            scope: me
        });
        sm.view = me.view;
        me.headerCt.view = me.view;
        me.relayEvents(me.view, ['cellclick', 'celldblclick']);
    }
    return me.view;
},
```

代码会调用 Ext.widget 方法创建视图，然后为视图绑定 uievent 事件，设置选择模型的 view 属性及标题栏容器对象的 view 属性为视图实例，最后将视图的单元格单击和双击事件传播到面板。

视图创建后，如果要显示垂直滚动条，且是无分页滚动条，则要创建 PagingScroller 对象实例；如果要显示水平滚动条，则为视图绑定 scroll 事件，以同步水平滚动。

最后为视图的 Store 绑定 load 事件，目前 onStoreLoad 是空函数，暂时未明确要做什么操作；为视图绑定 viewready 事件，触发 viewready 事件；为视图绑定 refresh 事件，主要操作是更新标题栏容器的布局。

视图创建后，调用 relayEvents 方法将视图的事件传播到面板，接着调用 callParent 方法调

用父类的 initComponent 方法。

从以上的代码分析可以看到，表格面板的工作就是将视图、列标题和滚动条这 3 者串联起来实现互动，至于数据与 UI 的互动是视图要做的事情。

10.1.3　表格视图的运行流程：Ext.view.Table 与 Ext.view.TableChunker

TableView 的 initComponent 方法代码如下：

```
initComponent: function() {
    var me = this,
        scroll = me.scroll;

    me.autoScroll = undefined;

    if (scroll === true || scroll === 'both') {
        me.style = Ext.apply(me.style||{}, {
            overflow: 'auto'
        });
    } else if (scroll === 'horizontal') {
        me.style = Ext.apply(me.style||{}, {
            "overflow-x": 'auto',
            "overflow-y": 'hidden'
        });
    } else if (scroll === 'vertical') {
        me.style = Ext.apply(me.style||{}, {
            "overflow-x": 'hidden',
            "overflow-y": 'auto'
        });
    } else {
        me.style = Ext.apply(me.style||{}, {
            overflow: 'hidden'
        });
    }
    me.selModel.view = me;
    me.headerCt.view = me;
    me.headerCt.markDirty = me.markDirty;
    me.initFeatures();
    me.tpl = '<div></div>';
    me.callParent();
},
```

代码先根据滚动条的设置调整好样式。然后设置选择模型和标题栏容器的 view 属性指向视图，还设置了标题栏容器的脏数据标记（markDirty）。

接着调用 initFeatures 方法初始化附加功能，其代码如下：

```
initFeatures: function() {
    var me = this,
        i = 0,
        features,
        len;

    me.features = me.features || [];
    features = me.features;
```

```
    len = features.length;

    me.featuresMC = new Ext.util.MixedCollection();
    for (; i < len; i++) {
        if (!features[i].isFeature) {
            features[i] = Ext.create('feature.' + features[i].ftype, features[i]);
        }
        features[i].view = me;
        me.featuresMC.add(features[i]);
        features[i].init();
    }
},
```

这里用了两个属性来记录这些附加功能实例，其中属性 features 为数组，而 featuresMC 为 MixedCollection 对象实例。每个附加功能都会调用其 init 方法进行初始化。

回到 initComponent 方法，定义完 tpl 配置项，接着就调用父类的 initComponent 方法。

根据 9.6.2 节，我们知道视图的主要是在 afterRender 方法中完成运作的，TableView 的 afterRender 方法代码如下：

```
afterRender: function() {
    var me = this;
    me.callParent();

    if (!me.enableTextSelection) {
        me.el.unselectable();
    }
    me.attachEventsForFeatures();
},
```

代码先调用父类的 afterRender 方法。如果没有开启文本选择，就调用 unselectable 方法禁止视图根元素进行文本选择。最后调用 attachEventsForFeatures 方法将附加功能的事件注入到视图。

下面来研究一下 TableView 的刷新操作，也就是 refresh 方法，代码如下：

```
refresh: function() {
    this.setNewTemplate();
    this.callParent(arguments);
   this.doStripeRows(0);
},
```

代码直接调用 setNewTemplate 方法，其代码如下：

```
setNewTemplate: function() {
    var me = this,
        columns = me.headerCt.getColumnsForTpl(true);

    me.tpl = me.getTableChunker().getTableTpl({
        rowCount: me.store.getCount(),
        columns: columns,
        features: me.features,
   enableTextSelection: me.enableTextSelection
    });
},
```

代码先调用 HeaderContainer 对象的 getColumnsForTpl 方法返回所有列以及其相关信息，包括 dataIndex、align、width、id、columnId 和 cls 等。这些都是在渲染数据到模板时要用到的。

接着是使用 getTableChunker 方法返回 TableChunker 对象，其代码如下：

```
getTableChunker: function() {
    return this.chunker || Ext.view.TableChunker;
},
```

因为 TableChunker 对象是单件模式的对象，是实例，所以可直接返回。返回后调用其 getTableTpl 方法生成模板，其代码如下：

```
getTableTpl: function(cfg, textOnly) {
    var tpl,
        tableTplMemberFns = {
            openRows: this.openRows,
            closeRows: this.closeRows,
            embedFeature: this.embedFeature,
            embedFullWidth: this.embedFullWidth,
            openTableWrap: this.openTableWrap,
            closeTableWrap: this.closeTableWrap
        },
        tplMemberFns = {},
        features = cfg.features || [],
        ln = features.length,
        i  = 0,
        memberFns = {
            embedRowCls: this.embedRowCls,
            embedRowAttr: this.embedRowAttr,
            firstOrLastCls: this.firstOrLastCls
        },
        metaRowTpl = Array.prototype.slice.call(this.metaRowTpl, 0),
        metaTableTpl;

    for (; i < ln; i++) {
        if (!features[i].disabled) {
            features[i].mutateMetaRowTpl(metaRowTpl);
            Ext.apply(memberFns, features[i].getMetaRowTplFragments());
            Ext.apply(tplMemberFns, features[i].getFragmentTpl());
            Ext.apply(tableTplMemberFns, features[i].getTableFragments());
        }
    }
    metaRowTpl = Ext.create('Ext.XTemplate', metaRowTpl.join(''), memberFns);
    cfg.row = metaRowTpl.applyTemplate(cfg);
    metaTableTpl  = Ext.create('Ext.XTemplate', this.metaTableTpl.join(''),
        tableTplMemberFns);
    tpl = metaTableTpl.applyTemplate(cfg);
    if (!textOnly) {
        tpl = Ext.create('Ext.XTemplate', tpl, tplMemberFns);
    }
    return tpl;
}
```

生成模板分 3 步骤，首先是生成数据行的模板（metaRowTpl），然后将它套进表格（metaTableTpl）内，最后是将表格生成最终的模板（tpl）。在每一个过程中都会将附加功能以成员函数的方式将 HTML 代码附加到模板中。

现在模板已经生成，通过调用父类的 refresh 方法就可以将模板渲染到视图的根元素了。

最后调用 doStripeRows 方法为视图添加条纹。

至此，一个视图的刷新操作就完成了。

10.1.4 列标题容器的运行流程：Ext.grid.header.Container

列标题容器是派生于 Container 对象的，因而，它没有面板的特性，只是一个容器，一个 div 而已，其 initComponent 方法代码如下：

```
initComponent: function() {
    var me = this;

    me.plugins = me.plugins || [];

    if (!me.isHeader) {
        me.resizer   = new Ext.grid.plugin.HeaderResizer();
        me.reorderer = new Ext.grid.plugin.HeaderReorderer();
        if (!me.enableColumnResize) {
            me.resizer.disable();
        }
        if (!me.enableColumnMove) {
            me.reorderer.disable();
        }
        me.plugins.push(me.reorderer, me.resizer);
    }

    if (me.isHeader && !me.items) {
        me.layout = me.layout || 'auto';
    }
    else {
        me.layout = Ext.apply({
            type: 'gridcolumn',
            align: 'stretchmax'
        }, me.initialConfig.layout);
    }
    me.defaults = me.defaults || {};
    Ext.applyIf(me.defaults, {
        triStateSort: me.triStateSort,
        sortable: me.sortable
    });

    me.menuTask = new Ext.util.DelayedTask(me.updateMenuDisabledState, me);
    me.callParent();
    me.addEvents(
        // 省略事件代码
    );
},
```

如果标题栏允许调整大小或移动列，（isHeader 为 false），那么创建插件 HeaderResizer 和 HeaderReorderer 的实例，HeaderResizer 的作用是实现拖动列标题改变列宽的功能，HeaderReorderer 的作用是实现通过拖动列标题改变列的位置。

如果标题栏不允许调整大小或移动列，且 items 不存在，则使用自动布局；否则使用 GridColumn 布局。

接着处理默认配置项；创建一个延时任务来处理列菜单的状态。

接着调用父类的 initComponent 方法。在 9.2.1 节中我们已经知道，容器会调用 initItems 方法初始化子条目，而在表格面板的 initComponent 方法中，已经将 HeaderContainer 对象的 items 属性指向配置项 columns 或 colModel 的值，因而列标题会在 initItems 方法中实例化。

现在还有个问题，就是带数据格式的列标题是怎么控制数据的显示的？在视图应用模板时，会调用 collectData 方法组织数据，而 collectData 方法会调用 prepareData 方法预处理数据，在 HeaderContainer 的 prepareData 方法中，有以下代码：

```
prepareData: function(data, rowIdx, record, view, panel) {
    var me         = this,
        obj        = {},
        headers    = me.gridDataColumns || me.getGridColumns(),
        headersLn = headers.length,
        colIdx     = 0,
        header,
        headerId,
        renderer,
        value,
        metaData,
        store = panel.store;

    for (; colIdx < headersLn; colIdx++) {
        metaData = {
            tdCls: '',
            style: ''
        };
        header = headers[colIdx];
        headerId = header.id;
        renderer = header.renderer;
        value = data[header.dataIndex];

        if (typeof renderer == "string") {
            header.renderer = renderer = Ext.util.Format[renderer];
        }

        if (typeof renderer == "function") {
            value = renderer.call(
                header.scope || me.ownerCt,
                value,
                metaData,
                record,
                rowIdx,
                colIdx,
                store,
                view
```

```
            );
        }

        // 省略调试代码
        if (me.markDirty) {
            obj[headerId + '-modified'] = record.isModified(header.dataIndex) ? Ext.
                baseCSSPrefix + 'grid-dirty-cell' : '';
        }
        obj[headerId+'-tdCls'] = metaData.tdCls;
        obj[headerId+'-tdAttr'] = metaData.tdAttr;
        obj[headerId+'-style'] = metaData.style;
        if (typeof value === 'undefined' || value === null || value === '') {
            value = header.emptyCellText;
        }
        obj[headerId] = value;
    }
    return obj;
},
```

在粗体代码中，如果自定义了配置项 renderer 且其为函数，那么就会调用它，返回值就是渲染数据时的显示值。

从整个列标题容器的运行流程可以看到，它只是提供一个渲染列标题用的容器，而列标题是由一个个列标题实例构成的，调整大小或移动列等功能则是由插件完成的。

10.1.5 列标题的运行流程：Ext.grid.column.Column

Column 对象派生于 HeaderContainer 对象，按理说它应该是一个容器，但是它又要显示列标题。这种双重身份的特征允许它在分组的时候是一个容器，不分组的时候当组件来用，但其运行流程是根据容器流程走的，其 initComponent 方法如下：

```
initComponent: function() {
    var me = this;

    if (Ext.isDefined(me.header)) {
        me.text = me.header;
        delete me.header;
    }

    if (me.flex) {
        me.minWidth = me.minWidth || Ext.grid.plugin.HeaderResizer.prototype.
            minColWidth;
    }
    else {
    }

    if (!me.triStateSort) {
        me.possibleSortStates.length = 2;
    }

    if (Ext.isDefined(me.columns)) {
        me.isGroupHeader = true;
```

```
        // 省略调试代码

        me.items = me.columns;
        delete me.columns;
        delete me.flex;
        delete me.width;
        me.cls = (me.cls||'') + ' ' + Ext.baseCSSPrefix + 'group-header';
        me.sortable = false;
        me.resizable = false;
        me.align = 'center';
    } else {
        me.isContainer = false;
    }
    me.addCls(Ext.baseCSSPrefix + 'column-header-align-' + me.align);

    me.callParent(arguments);

    me.on({
        element:  'el',
        click:    me.onElClick,
        dblclick: me.onElDblClick,
        scope:    me
    });
    me.on({
        element:    'titleEl',
        mouseenter: me.onTitleMouseOver,
        mouseleave: me.onTitleMouseOut,
        scope:      me
        });
    },
```

如果 header 属性值不为 undefined，那么将其值赋值给 text 属性，并删除该属性。

如果是使用 flex 定义列的宽度，那么要设置其最小宽度（minWidth）。

接着根据排序情况设置排序状态。

如果定义了 columns 配置项，也就是说该列是一个分组列，这时候要把列标题作为容器处理，即把列定义赋值到 items 属性里。这样，分组包含的列就成为列标题容器的子组件了。

在添加必要的样式和调用父类方法后，就要为元素绑定事件了。首先是为容器的元素绑定单击和双击事件，处理列标题的单击和双击操作；接着是为标题元素绑定鼠标移入或移出事件，实现高亮显示的效果。

10.1.6　虚拟滚动条的工作原理：Ext.grid.PagingScroller

虚拟滚动条，说是虚拟，主要是指其通过 div 元素的高度（height 属性）虚拟出表格无限滚动的效果。这是怎么实现的呢？我们研究一下 Ext JS 4 的示例“Buffered Scrolling”的滚动条就知道了，打开示例，在 HTML 面板中通过“获取元素”按钮选择 Grid 的垂直滚动条，会看到如下的代码：

```
<div style="overflow: auto; position: relative; -moz-user-select: -moz-none;
    margin: 0pt; width: 698px; height: 450px;" class="x-grid-view x-fit-item
```

```
        x-grid-view-default x-unselectable" id="gridview-1020" tabindex="-1">
    <div style="position: absolute; width: 1px; height: 105000px; top: 0pt; left:
        0pt;" id="ext-gen1045"></div>
    <table cellspacing="0" cellpadding="0" border="0" style="width: 681px;
        position: absolute; top: 0px;" class="x-grid-table x-grid-table-resizer">
            // 省略代码
    </table>
</div>
```

这是由视图生成的代码。注意粗体代码中的高度，值是 105000，其作用是虚拟出所有记录所占的高度（行高 × 记录总数），这样滚动条就会根据这个高度进行滚动，也就虚拟出了无分页滚动的效果了。那么这是如何实现的呢？

首先，在构造函数中，会绑定视图的 refresh 事件到 onViewRefresh 方法，也就是当视图刷新后，会执行该方法。而在该方法中，会调用 getScrollHeight 方法获取滚动高度，其代码如下：

```
getScrollHeight: function() {
    var me = this,
        view    = me.view,
        table,
        firstRow,
        store   = me.store,
        rowCount,
        deltaHeight = 0;

    if (me.variableRowHeight) {
        table = me.view.el.down('table', true);
        if (me.rowHeight) {
            deltaHeight = table.offsetHeight - me.initialTableHeight;
        } else {
            me.initialTableHeight = table.offsetHeight;
            me.rowHeight = me.initialTableHeight / me.store.pageSize;
        }
    } else if (!me.rowHeight) {
        firstRow = view.el.down(view.getItemSelector());
        me.rowHeight = firstRow ? firstRow.getHeight(false, true) : 0;
    }

    rowCount = store[(!store.remoteFilter && store.isFiltered()) ? 'getCount' :
        'getTotalCount']() || 0;
    return Math.floor(rowCount * me.rowHeight) + deltaHeight;
},
```

代码的重点在粗体字体的返回语句，返回的值是记录总数乘以行高再加上修正值得到的。而行高的获取有两种方式，一是根据表格元素的高度除以每页记录数，一是直接通过条目行的高度获取。示例中，一行的高度是 21，记录总数为 5000 行，也就是总高度是 105000。

接着，就是怎样将滚动条的滚动与记录的显示关联起来。在 PagingScroller 对象的构造函数中，会为视图绑定 scroll 事件到 onViewScroll 方法，其代码如下：

```
onViewScroll: function(e, t) {
    var me = this,
```

```
            view = me.view,
            lastPosition = me.position;

        me.position = view.el.dom.scrollTop;
        me.lastScrollDirection = me.position > lastPosition ? 1 : -1;
        me.handleViewScroll(e, me.lastScrollDirection);
    },
```

在这里一定要先弄清楚 scrollTop 属性的作用，它的值表示一个元素滚动后，其顶部到可视区域顶部的高度，也就是元素在可视区域之上的高度。滚动条滚动的时候，会修改该值。与上一次滚动后的值进行比较，就可知道移动的方向是向上还是向下，然后调用 handleViewScroll 方法进行处理，其代码如下：

```
handleViewScroll: function(e, direction) {
    var me = this,
        store = me.store,
        view = me.view,
        guaranteedStart = me.previousStart = store.guaranteedStart,
        guaranteedEnd = me.previousEnd = store.guaranteedEnd,
        renderedSize = store.getCount(),
        totalCount = store.getTotalCount(),
        visibleStart = me.getFirstVisibleRowIndex(),
        visibleEnd = me.getLastVisibleRowIndex(),
        requestStart,
        requestEnd;

    if (totalCount >= renderedSize) {
        me.scrollProportion = undefined;
        if (direction == -1) {
            if (visibleStart !== undefined) {
                if (visibleStart < (guaranteedStart + me.numFromEdge)) {
                    requestStart = Math.max(0, visibleEnd + me.numFromEdge +
                        me.trailingBufferZone - renderedSize);
                }
            }
            else {
                me.scrollProportion = view.el.dom.scrollTop / (view.el.dom.
                    scrollHeight - view.el.dom.clientHeight);
                requestStart = Math.max(0, totalCount * me.scrollProportion -
                    (renderedSize / 2) - me.numFromEdge - ((me.leadingBufferZone
                    + me.trailingBufferZone) / 2));
            }
        }
        else {
            if (visibleStart !== undefined) {
                if (visibleEnd > (guaranteedEnd - me.numFromEdge)) {
                    requestStart = Math.min(visibleStart - me.numFromEdge -
                        me.trailingBufferZone, totalCount - renderedSize);
                }
            }
            else {
                me.scrollProportion = view.el.dom.scrollTop / (view.el.dom.
                    scrollHeight - view.el.dom.clientHeight);
```

```
                requestStart = Math.min(totalCount - renderedSize, totalCount *
                    me.scrollProportion - (renderedSize / 2) - me.numFromEdge -
                    ((me.leadingBufferZone + me.trailingBufferZone) / 2));
            }
        }
        if (requestStart !== undefined) {
            requestStart = requestStart &~1;
            requestEnd = requestStart + renderedSize - 1;

            if (requestEnd > totalCount - 1) {
                me.cancelLoad();
                if (!store.rangeSatisfied(totalCount - renderedSize + 1, totalCount
                    - 1)) {
                    store.guaranteeRange(totalCount - renderedSize + 1, totalCount
                        - 1);
                }
            }

            else {

                if (store.rangeSatisfied(requestStart, requestEnd)) {
                    me.cancelLoad();
                    store.guaranteeRange(requestStart, requestEnd);
                } else {
                    me.attemptLoad(requestStart, requestEnd);
                }
            }
        }
    },
```

在分析代码前，要先清楚 guaranteedStart、guaranteedEnd、visibleStart 和 visibleEnd 这个 4 个变量的作用。其中，guaranteedStart 的作用是保证滚动条在往上滚动有可显示的记录，以避免忽然要加载数据而造成页面停顿；guaranteedEnd 是保证往下移动时有可显示记录；visibleStart 则是保证在视图可视区域内能看到的第一条记录；visibleEnd 是保证能看到的最后一条记录。

代码中，numFromEdge 属性的作用是一个页与页之间的边界值，当滚动显示的记录行距离下一页（或上一页）的第一条记录（最后一条记录）少于该值时，就要预加载下一页（或上一页）。例如，每页记录数是 100，那么当滚动显示到顶部 3 条记录之前的记录时，或底部 98 条记录之后的记录时，就要预加载前一页或后一页的数据了。

整个运算的目的是求 requestStart 和 requestEnd，requestStart 是刷新视图时渲染的第一条记录的索引，requestEnd 则是最后一条记录的索引。而这些索引都是有可能跨页面的，因而需要通过计算获取这些索引。

开始索引和结束索引计算好以后，会调用 Store 的 guaranteeRange 方法获取这些记录所在的页数，并调用 onGuaranteedRange 方法将该范围的记录保存到数组，最后触发 guaranteedrange 事件。在此过程，如果需要加载数据，则调用 prefetchPage 加载数据。

在虚拟滚动条的构造函数中，将 Store 的 guaranteedrange 事件绑定到了 onGuaranteed-

Range 方法，其代码如下：

```
onGuaranteedRange: function(range, start, end) {
    var me = this,
        ds = me.store;

    if (range.length && me.visibleStart < range[0].index) {
        return;
    }

    ds.loadRecords(range);
},
```

在这里会调用 Store 的 loadRecords 加载记录，而这将会触发视图的 refresh 事件刷新视图，这样又会执行虚拟滚动条的 onViewRefresh 方法，其代码如下：

```
onViewRefresh: function() {
    var me = this,
        newScrollHeight = me.getScrollHeight(),
        view = me.view,
        viewEl = view.el.dom,
        store = me.store,
        rows,
        newScrollOffset,
        scrollDelta,
        table,
        tableTop;

    me.stretcher.setHeight(newScrollHeight);

    if (me.scrollProportion !== undefined) {
        table = me.view.el.child('table', true);
        me.scrollProportion = view.el.dom.scrollTop / (newScrollHeight - table.
            offsetHeight);
        table = me.view.el.child('table', true);
        table.style.position = 'absolute';
        table.style.top = (me.scrollProportion ? (newScrollHeight *
            me.scrollProportion) - (table.offsetHeight * me.scrollProportion) : 0)
            + 'px';
    }
    else {
        table = me.view.el.child('table', true);
        table.style.position = 'absolute';
        table.style.top = (tableTop = (store.guaranteedStart||0) * me.rowHeight)
            + 'px';

        if (me.scrollOffset) {
            rows = view.getNodes();
            newScrollOffset = -view.el.getOffsetsTo(rows[me.commonRecordIndex])[1];
            scrollDelta = newScrollOffset - me.scrollOffset;
            me.position = (view.el.dom.scrollTop += scrollDelta);
        }

        else if ((tableTop > viewEl.scrollTop) || ((tableTop + table.offsetHeight)
            < viewEl.scrollTop + viewEl.clientHeight)) {
```

```
                me.position = viewEl.scrollTop = tableTop;
            }
        }
    },
```

在代码中，会根据当前滚动条位置调整表格元素的 top 属性，从而达到虚拟显示出当前滚动位置记录的效果。

以上就是虚拟滚动条实现无分页效果的过程，笔者对这技术相当的佩服，在滚动条滚动的时候，除非改变幅度很大，不然你根本就感觉不到视图已经被替换，像变魔术一样在不知不觉中就完成了。如果不相信，可打开示例“Infinite Scrolling”，在 Firebug 的网络面板中观察一下，当滚动条滚动时，在请求了数据之后的 Grid 的变化。

10.1.7 锁定列的运行流程：Ext.grid.Lockable 与 Ext.grid.LockingView

在 TablePanel 的 initComponent 方法中，如果在列的配置对象中定义了配置项 locked 为 true，则会将 Lockable 对象混入 Grid 中，并调用其 injectLockable 方法，其代码如下：

```
injectLockable: function() {
    this.lockable = true;
    this.hasView = true;
    var me = this,
       store = me.store = Ext.StoreManager.lookup(me.store),
        xtype = me.determineXTypeToCreate(),
        selModel = me.getSelectionModel(),
        lockedGrid = {
        xtype: xtype,
        store: store,
        scrollerOwner: false,
        enableAnimations: false,
        scroll: false,
        selModel: selModel,
        border: false,
        cls: Ext.baseCSSPrefix + 'grid-inner-locked',
        isLayoutRoot: function() {
            return false;
        }            },
    normalGrid = {
        xtype: xtype,
        store: store,
        scrollerOwner: false,
        enableAnimations: false,
        selModel: selModel,
        border: false,
        isLayoutRoot: function() {
            return false;
        }
    },
    i = 0,
    columns,
    lockedHeaderCt,
    normalHeaderCt,
```

```
        lockedView,
        normalView;
    me.addCls(Ext.baseCSSPrefix + 'grid-locked');
    Ext.copyTo(normalGrid, me, me.bothCfgCopy);
    Ext.copyTo(lockedGrid, me, me.bothCfgCopy);
    Ext.copyTo(normalGrid, me, me.normalCfgCopy);
    Ext.copyTo(lockedGrid, me, me.lockedCfgCopy);
    for (; i < me.normalCfgCopy.length; i++) {
        delete me[me.normalCfgCopy[i]];
    }
    for (i = 0; i < me.lockedCfgCopy.length; i++) {
        delete me[me.lockedCfgCopy[i]];
    }
    me.addEvents(
        'lockcolumn',
        'unlockcolumn'
    );

    me.addStateEvents(['lockcolumn', 'unlockcolumn']);
    me.lockedHeights = [];
    me.normalHeights = [];
    columns = me.processColumns(me.columns);
    lockedGrid.width = columns.lockedWidth;
    lockedGrid.columns = columns.locked;
    normalGrid.columns = columns.normal;

    normalGrid.flex = 1;
    lockedGrid.viewConfig = me.lockedViewConfig || {};
    lockedGrid.viewConfig.loadingUseMsg = false;
    normalGrid.viewConfig = me.normalViewConfig || {};
    Ext.applyIf(lockedGrid.viewConfig, me.viewConfig);
    Ext.applyIf(normalGrid.viewConfig, me.viewConfig);
    me.normalGrid = Ext.ComponentManager.create(normalGrid);
    me.lockedGrid = Ext.ComponentManager.create(lockedGrid);
    me.view = Ext.create('Ext.grid.LockingView', {
        locked: me.lockedGrid,
        normal: me.normalGrid,
        panel: me
    });
    lockedView.on({
        scroll: {
            fn: me.onLockedViewScroll,
            element: 'el',
            scope: me
        },
        mousewheel: {
            fn: me.onLockedViewMouseWheel,
            element: 'el',
            scope: me
        }
    });
    normalView.on({
        scroll: {
            fn: me.onNormalViewScroll,
            element: 'el',
```

```
                scope: me
            },
            refresh: me.createSpacer,
            beforerefresh: me.destroySpacer,
            scope: me
        });

        if (me.syncRowHeight) {
            lockedView.on({
                refresh: me.onLockedViewRefresh,
                itemupdate: me.onLockedViewItemUpdate,
                scope: me
            });

            normalView.on({
                refresh: me.onNormalViewRefresh,
                itemupdate: me.onNormalViewItemUpdate,
                scope: me
            });
        }
        lockedHeaderCt = me.lockedGrid.headerCt;
        normalHeaderCt = me.normalGrid.headerCt;
        lockedHeaderCt.lockedCt = true;
        lockedHeaderCt.lockableInjected = true;
        normalHeaderCt.lockableInjected = true;

        lockedHeaderCt.on({
            columnshow: me.onLockedHeaderShow,
            columnhide: me.onLockedHeaderHide,
            columnmove: me.onLockedHeaderMove,
            sortchange: me.onLockedHeaderSortChange,
            columnresize: me.onLockedHeaderResize,
            scope: me
        });

        normalHeaderCt.on({
            columnmove: me.onNormalHeaderMove,
            sortchange: me.onNormalHeaderSortChange,
            scope: me
        });

        me.modifyHeaderCt();
        me.items = [me.lockedGrid, me.normalGrid];

        me.relayHeaderCtEvents(lockedHeaderCt);
        me.relayHeaderCtEvents(normalHeaderCt);

        me.layout = {
            type: 'hbox',
            align: 'stretch'
        };
    },
```

代码一开始就定义了 lockedGrid 与 normalGrid 这两个 GridPanel 的配置对象，它们使用的是同一选择模型，也就是说，无论在哪个面板进行选择，结果都是一样的。

接着会使用 copyTo 方法将 GridPanel 中 normalCfgCopy 与 lockedCfgCopy 数组内的成员复制到对应的面板配置对象中。

接着调用 addEvents 方法添加事件，并且调用 addStateEvents 方法添加状态事件。

接着要调用 processColumns 方法将锁定的列（locked 为 true）与未锁定的列拆分开来，再将拆分好的列作为列定义加入到对应的面板。

要在同一行将两个面板合并在一起，最好的办法是使用 HBox 布局将原面板的区域划分成两个区域，而锁定的列是不允许调整列宽度的，因而其宽度基本是固定的，所以设置 normalGrid 的 flex 值为 1 时可让其自适应余下的空间。

接着是将配置项 lockedViewConfig 内的配置项复制到 lockedGrid 的 viewConfig 中，将 normalViewConfig 内的配置项复制到 normalGrid 的 viewConfig 中，也就是说，在定义 Grid 时可使用 lockedViewConfig 或 normalViewConfig 分别为锁定列的面板及正常面板的视图进行配置。锁定面板内不会有任何操作，不需要遮蔽，因而设置 loadingUseMsg 为 false。

接着将配置项 viewConfig 内的配置项复制到两个面板的 viewConfig 中，然后创建两个面板。

接着是创建 LockingView 对象，其构造函数如下：

```
constructor: function(config){
    var me = this,
        eventNames = [],
        eventRe = me.eventRelayRe,
        locked = config.locked.getView(),
        normal = config.normal.getView(),
        events,
        event;

    Ext.apply(me, {
        lockedView: locked,
        normalView: normal,
        lockedGrid: config.locked,
        normalGrid: config.normal,
        panel: config.panel
    });
    me.mixins.observable.constructor.call(me, config);

    events = locked.events;
    for (event in events) {
        if (events.hasOwnProperty(event) && eventRe.test(event)) {
            eventNames.push(event);
        }
    }
    me.relayEvents(locked, eventNames);
    me.relayEvents(normal, eventNames);

    normal.on({
        scope: me,
        itemmouseleave: me.onItemMouseLeave,
        itemmouseenter: me.onItemMouseEnter
    });
```

```
        locked.on({
            scope: me,
            itemmouseleave: me.onItemMouseLeave,
            itemmouseenter: me.onItemMouseEnter
        });
    },
```

首先要做的是将 LockingView 的属性 lockedView、normalView、lockedGrid 和 normalGrid 指向对应的对象。

接着要做的是将两个视图的事件传播到 LockingView 对象，而这也正好体现了 LockingView 对象的作用，无论是锁定视图的事件，还是正常视图的事件，都会传播到 LockingView 对象中，也就是，LockingView 对象将两个视图的事件集合到一起了。这样，两个视图的操作就好像在一个视图里操作一样，不会让用户感觉是在两个不同的视图里操作。如果没有 LockingView 对象，简单如 itemclick 事件，都必须分别定义，这就太麻烦了。

最后将两个视图的 itemmouseleave 事件都绑定到 onItemMouseLeave 方法，将 itemmouseenter 事件绑定到 onItemMouseEnter 方法，这样鼠标移入或移出视图的效果就可同步进行，而不会出现一边显示了效果，而另一边没有显示的情况。

回到 injectLockable 方法，接着为锁定的视图绑定滚动和鼠标滚轮事件，为正常视图绑定滚动、刷新和 beforerefresh 事件。

如果定义了 syncRowHeight 为 true，要同步行高，就要将两个面板视图的 refresh 事件和 itemupdate 事件绑定方法，处理高度的同步问题。

接着要做的是处理列标题，允许通过拖动和通过列标题的菜单对列锁定和对列解锁。

最后是定义 Grid 的 items 属性和设置布局为 HBox 布局。

至此，锁定列的工作就完成了。

10.2 使用 Grid

10.2.1 最简单的 Grid

GridPanel 继承自 TablePanel，因而最简单的 Grid 必须有视图、选择模型、Store、列标题容器和滚动条。在这 5 个部件中，视图、选择模型、列标题容器和滚动条都有默认的配置，而 Store 和列标题容器中的列标题组件是没有定义的，因而一个最简单的 Grid 需要定义 Store 和列标题。

现在我们就来创建一个最简单的 Grid，打开模板页，在命令行先创建一个 Store：

```
var store=Ext.create("Ext.data.ArrayStore",{
    fields:["id","name"],
    data:[["1","张三"],["2","李四"]]
});
```

这是一个最简单的 Store，它自动生成模型和加载两条数据。

然后创建一个 Grid：

```
Ext.create("Ext.grid.Panel",{
    renderTo:Ext.getBody(),
    store:store,
    columns:[
            {text:"编号",dataIndex:"id",flex:1},
            {text:"姓名",dataIndex:"name",flex:1}
    ]
})
```

因为没有父容器，所以必须要有 renderTo，不然就不会渲染了，余下两个配置项 Store 和 columns 也必须要有，不然就不知道 Grid 要显示什么数据及如何显示数据。最简单的 Grid 也就是定义了这两个配置项的 Grid。当然了，如果没有渲染容器自动渲染它，就必须定义配置项 renderTo 渲染 Grid。运行后将看到如图 10-1 所示的效果，因为列的宽度定义了配置项 flex 为 1，因而会把整个浏览器窗口的宽度平均分成两部分，它们的高度就是两条记录的高度。

编号	姓名
1	张三
2	李四

图 10-1　最简单 Grid 的效果

10.2.2　列的配置项

列定义对 Grid 来说是必须。配置项 columns 内的列定义，是 Column 对象的配置对象，它负责列标题的显示及相关列的操作，如排序、列的隐藏以及列的单元格的显示格式。

1. 字段：dataIndex

配置项 dataIndex 的作用是指明该列将显示 Store 中的那个字段，是必须配置的项。字段值就是 Store 中定义的字段名称。

2. 标题：text

配置项 text 定义的值会作为列的标题，如果不想显示标题，则可不设置。

3. 数据对齐方式：align

通过配置项 align 可控制列内数据的对齐方式，其值可以是 left、center 和 right，分别对应左对齐、居中对齐和右对齐。它是使用样式 text-align 属性控制对齐的，默认值是 left。Ext JS 2.0 到目前的 Ext JS 4.0，列标题的对齐与数据的对齐使用的是相同的形式，例如定义了 align 为 right，那么列标题也是右对齐的，这不太符合中国人的习惯。可通过重写 Column 对象的 afterRender 方法并将其放到本地化文件中来解决，具体做法如下：

在本地化文件中添加以下代码：

```
if(Ext.grid.Column){
    Ext.grid.Column.override({
```

```
                //要重写的方法
        });
    }
```

这里不能使用 Ext.apply 方法来重写，因为在 afterRender 方法中，要通过 callParent 方法调用父类的 afterRender 方法，这需要用到 afterRender 方法内的“$owner”和“$name”属性，而使用 apply 方法会丢失这两个属性，导致调用 callParent 方法失败，所以必须使用 override 方法重写，才可以保留这两个属性。

接着将 afterRender 方法的定义代码全部复制到注释中的位置（注意将最后的逗号去掉），然后修改下面代码：

```
    el.addCls(Ext.baseCSSPrefix + 'column-header-align-' + me.align).addClsOnOver(me.
overCls);
```

这句代码是为列标题添加对齐样式的，默认使用配置项 align 的值，因而我们只要添加一个配置项并使用该配置项就行了，修改代码如下：

```
me.titleAlign = me.titleAlign || me.align;
el.addCls(Ext.baseCSSPrefix + 'column-header-align-' + me.titleAlign).
    addClsOnOver(me.overCls);
```

代码中添加了配置项 titleAlign，如果定义了该配置项，则使用该配置项的值；否则使用 align 的值，注意一定要修改粗体代码，不然添加的 titleAlign 就是多余的。

这样，就可以在列的配置对象中添加 titleAlign 配置项来设置列标题的对齐方式了。

修改一下 10.2.1 节的例子的代码，在 id 列添加配置项 titleAlign 并设置其值为 center，将看到如图 10-2 所示的结果。

编号	姓名
1	张三
2	李四

图 10-2　自定义列标题的显示效果

4. 列的宽度：flex 或 width

列标题容器是使用 HBox 布局的，因而可使用 flex 或 width 这两个配置项设置列的宽度。如果没有设置，则默认宽度为 100 像素。

5. 固定列：fixed

Grid 在默认情况下，列是可以调整宽度的。如果要固定该列又不希望其调整宽度，则可设置配置项 fixed 为 true。补充一点，设置配置项 resizable 为 false 固定不了列的宽度。

6. 排序功能：sortable

Grid 默认允许单击列标题及其下拉菜单的排序子菜单对 Store 进行排序，如果不希望对该列进行排序，则可设置其配置项 sortable 为 false。

7. 隐藏列：hidden 与 hideable

如果要在 Grid 初始化时隐藏列，则可定义配置项 hidden 为 true。

在默认情况下，Grid 允许通过列标题的菜单隐藏列。如果禁止通过菜单隐藏列，则可设置配置项 hideable 为 false。

要注意，如果配置了 hidden 为 true，又配置了 hideable 为 false，那么就要通过代码或者其他操作才可以显示隐藏列。

8. 禁用列菜单：menuDisabled

Grid 在默认情况下，在列标题里会有一个下拉菜单用来执行排序、隐藏列或显示列，如果要禁用菜单，可设置任意一列的配置项 menuDisable 为 true。

9. 改变列的位置：draggable

Grid 在默认情况下，允许通过拖动列标题改变列的位置。如果要禁止改变列的位置，则可设置配置项 draggable 为 false。

10. 设置单元格的样式：tdCls

如果希望该列的单元格有特别的样式，可设置配置项 tdCls，其值为样式名称。

10.2.3　自定义单元格的显示格式

控制单元格的显示格式有两种方法，一是使用配置项 renderer，一是使用列对象。本节将主要讲述如何使用 renderer 配置项自定义单元格的显示格式，下节将讲述如何使用列对象自定义单元格的显示格式。

配置项 renderer 的值，可以是一个函数或对象的方法。如果是函数，在函数内可依次接收以下参数：

- value：当前单元格对应的字段值。
- metadata：当前单元格的元数据集合，集合中包括 tdCls、tdAttr 和 style 三个属性，可用来设置单元格（TD）的样式类、属性和样式。
- record：当前行的记录。
- rowIndex：当前行索引。
- colIndex：当前列索引。
- store：当前 Grid 的 Store 对象。
- view：当前视图。

下面通过一个例子演示如何使用 renderer 配置项自定义显示格式。

（1）功能描述

将下面代码中 Store 的数据以指定格式显示：

```
var store=Ext.create("Ext.data.ArrayStore",{
    fields:[{name:"id",type:"int"},
            "name","nationality",
            {name:"birthday",type:"date",format:"Y-m-d"},
```

```
            "photo"
    ],
    data:[
            [1,"张三","cn.gif","1948-03-02","head-1.jpg"],
            [2,"李四","us.gif","1960-04-03","head-2.jpg"],
            [3,"王五","fr.gif","1999-08-13","head-3.jpg"]
    ]
});
```

格式要求：

- id 字段固定以 6 位的数字长度输出，空位补 0。
- name 字段会显示提示信息，其内容为头像图片（photo 字段）。
- nationality 字段以图片形式显示。
- birthday 字段如果显示生日，格式为：“1990 年 12 月 25 日”。
- birthday 字段如果显示年龄，当年龄大于 60 岁时，字体为红色；年龄小于 16 岁，字体为绿色。

（2）实现代码

使用模板页创建一个名称为 10-1.html 的页面文件，首先要加入的是初始化 QuickTips[⊖]对象以显示提示信息，加入以下代码：

```
Ext.QuickTips.init();
```

然后将功能描述中的 store 定义加入到 OnReady 函数内。接着定义一个 Grid，字段先不加入：

```
Ext.create("Ext.grid.Panel",{
    renderTo:Ext.getBody(),
    width:300,
    height:200,
    store:store,
    columns:[
            // 字段定义
    ]
});
```

现在开始加入字段，首先是加入 id，要显示固定长度的数字，使用 Ext.String.leftPad 方法就可以，定义如下：

```
{text:"编号",dataIndex:"id",flex:1,align:"center",
    renderer:function(v){
            return Ext.String.leftPad(v,6,"0");
    }
},
```

字段 name 要显示提示信息，且其为图像，所以不能使用标准的 title 属性，必须使用 QuickTips 对象，只要为 td 添加“data-qtip”属性就可以了，其定义代码如下：

```
{text:"姓名",dataIndex:"name",flex:1,
```

⊖ 相关信息请阅读 16.3 节。

```
    renderer:function(v,meta,rec){
meta.tdAttr="data-qtip=\"<img  src='../images/"+rec.data.photo+"'>\"
     title='"+v+"'";
            return v;
    }
},
```

代码中，通过第二个参数 metaData 对象的属性 tdAttr 可为 td 添加属性，有多个属性时需要用空格将其隔开，例子中添加了“data-qtip”和“title”两个属性。

字段 nationality 要显示图片，只要构建一个图片的 HTML 返回即可，其代码如下：

```
{text:"国籍",dataIndex:"nationality",flex:1,align:"center",
    renderer:function(v){
            return "<img src='../images/flag/"+ v+ "'/>"
    }
},
```

字段 birthday 显示为生日时，需要转换日期格式，使用 Ext.util.Format.dateRenderer 方法即可，其代码如下：

```
{text:"生日",dataIndex:"birthday",flex:2,renderer: Ext.util.Format.dateRenderer("Y
    年m月d日")},
```

最后要将 birthday 字段显示为年龄，且根据年龄不同用不同颜色字体，可使用第二参数 metaData 对象的 style 为字体设置颜色，其定义如下：

```
{text:"年龄",dataIndex:"birthday",flex:1,align:"center",
    renderer:function(v,meta){
            var n=new Date(),
                age=n.getFullYear()-v.getFullYear();
            if(age>=60)
                meta.style="color:red";
            else if(age<=16){
                meta.style="color:green";
            }
            return age;
    }
}
```

至此，本示例就完成了。

（3）页面效果

在浏览器中打开示例页，并将鼠标移动到“王五”上，将看到如图 10-3 所示的效果。图中，图片的提示是 QuickTips 对象显示的，“王五”则是 title 属性显示的。

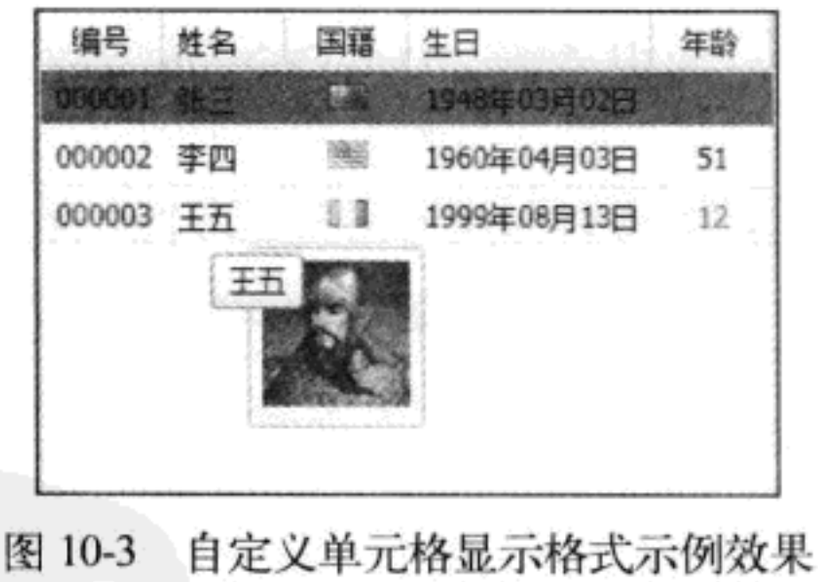


图 10-3　自定义单元格显示格式示例效果

10.2.4　通过列对象定义单元格的显示格式

从 Column 对象中（包括示例中的），一共派生了 7 个对象，这 7 个对象的主要作用是控制单元格的显示格式（RowNumberer 对象例外）。这 7 个对象的共同点就是通过重写 Column 对象的 renderer 方法实现格式化显示。

1. 用来显示行号的 RowNumberer 对象

RowNumberer 对象重写的 renderer 方法代码如下：

```
renderer: function(value, metaData, record, rowIdx, colIdx, store) {
    if (this.rowspan){
        metaData.cellAttr = 'rowspan="'+this.rowspan+'"';
    }
    metaData.tdCls = Ext.baseCSSPrefix + 'grid-cell-special';
    return store.indexOfTotal(record) + 1;
}
```

测试过代码中的 cellAttr，是不起作用的。如果起作用，则会合并单元格，估计是为更复杂的行合并显示预备的吧，观望中。最后两句是重点，第一句设置单元格的样式，第二句返回记录在数据中的索引。

使用 RowNumberer 对象很简单，其配置对象只需要设置标题和宽度就可以，所以配置项有 text、width 和 flex，不过一般 text 都不定义，width 则根据显示需要定义，当记录总数比较大时，为了完整显示行号，就需要对宽度进行定义。

2. 渲染数字字段：NumberColumn 对象

NumberColumn 对象的 renderer 方法定义如下：

```
this.renderer = Ext.util.Format.numberRenderer(this.format);
```

这相当简单，就是使用 Ext JS 格式化函数来显示数据，其默认格式是“0,000.00”。

使用 NumberColumn 对象很简单，在定义列的配置对象时，设置配置项 xtype 的值为 numbercolumn，然后根据数字格式定义配置项 format 即可，如果没有定义 format，则使用默认格式。

3. 渲染日期字段：DateColumn 对象

DateColumn 对象的 renderer 方法定义如下：

```
this.renderer = Ext.util.Format.dateRenderer(this.format);
```

可以看到，DateColumn 对象只是把 dateRenderer 方法封装起来，然后通过定义配置项 format 来实现日期的格式化输出，当然，还需要在配置对象中加入 xtype 配置项的定义。至于是使用 renderer 配置项方便还是使用 DateColumn 对象方便，还是根据自己喜好去选吧。

4. 渲染布尔值：BooleanColumn

BooleanColumn 对象的 renderer 方法定义如下：

```
this.renderer = function(value){
    if(value === undefined){
        return undefinedText;
    }
    if(!value || value === 'false'){
        return falseText;
    }
    return trueText;
};
```

当值不存在时，会显示 undefinedText 的值，默认值为空格；当值取反为 true 或值等于字符串“false”时，会显示 falseText 的值；如果以上条件都不成立，则显示 trueText 的值，默认值为 true。

undefinedText、falseText 和 trueText 是 BooleanColumn 对象的配置项，因而可在列的配置对象中改变配置显示值。

BooleanColumn 对象的好处是有 3 种状态可显示，不过以文字显示出来不太直观。

5. 不是默认组件的 Ext.ux.CheckColumn

想不明白，为什么不把 CheckColumn 对象作为默认组件添加到 Ext JS 4 中，而只是把它作为用户扩展。其定义文件在 Ext JS 包的“\examples\ux\”目录下，文件名为 CheckColumn.js，因而要使用该组件必须引用该脚本，也可以用动态加载的方法加载。其 renderer 方法定义如下：

```
renderer : function(value){
    var cssPrefix = Ext.baseCSSPrefix,
        cls = [cssPrefix + 'grid-checkheader'];
    if (value) {
        cls.push(cssPrefix + 'grid-checkheader-checked');
    }
    return '<div class="' + cls.join(' ') + '"> </div>';
```

从代码可以看到，它不是真的使用复选框作为显示，而是使用背景图片虚拟复选框效果的。CheckColumn 对象的不足是它只有两种状态，而 BooleanColumn 对象有 3 种状态，不过根据 CheckColumn 对象写一个有 3 种状态的组件也不是难事，难的是要找美工去做 3 个图片。CheckColumn 对象的优点是可通过鼠标和键盘改变状态，并触发 checkchange 事件，因而非常适合用于在非编辑的 Grid 中修改状态。它没有作为默认组件加入实在是遗憾。

6. 使用模板渲染数据：TemplateColumn

TemplateColumn 对象 renderer 方法定义如下：

```
tpl = me.tpl = (!Ext.isPrimitive(me.tpl) && me.tpl.compile) ? me.tpl : new Ext.
    XTemplate(me.tpl);
me.renderer = function(value, p, record) {
    var data = Ext.apply({}, record.data, record.getAssociatedData());
    return tpl.apply(data);
};
```

从代码可以看到，使用 TemplateColumn 对象时必须定义配置项 tpl，其值可以是 XTemplate 对象的配置对象，也可以是 XTemplate 实例。

7. 允许单击操作的图标：ActionColumn

这是为传统 Web 操作定制的列对象，可在列中放置一些小图标来实现内联的单击操作。如果使用自定义链接的方式，开发人员要在定义链接时把记录的关键字加入链接，不过单击链接后，还要去查找对象，这样比较麻烦。ActionColumn 对象解决了这个问题，不但实现了单击操作在列的作用域内，而且封装了单击句柄，通过句柄的参数可获得当前单元格的位置、

视图等信息。以下是 ActionColumn 对象的构造函数：

```
constructor: function(config) {
    var me = this,
        cfg = Ext.apply({}, config),
        items = cfg.items || [me],
        l = items.length,
        i,
        item;
    delete cfg.items;
    me.callParent([cfg]);
    me.items = items;
    me.renderer = function(v, meta) {
       v = Ext.isFunction(cfg.renderer) ? cfg.renderer.apply(this, arguments)||'' : '';
        meta.tdCls += ' ' + Ext.baseCSSPrefix + 'action-col-cell';
        for (i = 0; i < l; i++) {
            item = items[i];
            item.disable = Ext.Function.bind(me.disableAction, me, [i], 0);
            item.enable = Ext.Function.bind(me.enableAction, me, [i], 0);
            v += '<img alt="' + (item.altText || me.altText) + '" src="' + (item.
                icon || Ext.BLANK_IMAGE_URL) +
                '" class="' + Ext.baseCSSPrefix + 'action-col-icon ' + Ext.
                    baseCSSPrefix + 'action-col-' + String(i) + ' ' + (item.
                    disabled ? Ext.baseCSSPrefix + 'item-disabled' : ' ') +
                ' ' + (Ext.isFunction(item.getClass) ? item.getClass.apply(item.
                        scope||me.scope||me, arguments) : (me.iconCls || '')) + '"' +
                ((item.tooltip) ? ' data-qtip="' + item.tooltip + '"' : '') + ' />';
        }
        return v;
    };
},
```

从粗体代码可以知道，只有一个图标时，不需要使用 items 配置项，而有多个图标时，就要使用 items 配置项，也就是说，AccordionColumn 对象的部分配置项既可在自身使用，也可以在 items 内的配置对象使用。

在 renderer 方法内，居然可以再用 renderer 配置项渲染数据。在循环里，创建 HTML 代码居然连简单的模板都没有用。可能看着有点晕，不过主要操作就是把配置项组合在一起。

AccordionColumn 对象有以下配置项：

- altText：应用于 img 来标记 alt 属性的值。可用于 items 内的配置对象。
- disabled：开启或关闭操作。
- getClass：返回应用于图标 IMG 标记的样式的函数，函数的接收参数与由 renderer 配置项定义的函数的接收参数一样，这个从源代码中可以看到，renderer 配置项的参数（arguments）会直接传递给函数。该函数的作用是可根据当前记录的状态配置图标的显示状态，例如当前记录不运行删除，可将删除图标显示为禁止单击。该配置项可用于 items 内的配置对象。
- handler：图标的单击句柄，函数依次可接收 view（视图）、rowIndex（行索引）、colIndex（列索引）、item（触发句柄的元素对象）、e（事件）等参数。通过行索引可以从 Store 中找到记录。可用于 items 内的配置对象。

- icon：图标显示的图片地址。可用于 items 内的配置对象。
- iconCls：应用到图标的样式。可用于 items 内的配置对象。
- items：由图标定义的配置对象组成的数组。
- scope：作用域。可用于 items 内的配置对象。
- stopSelection：默认值为 true，会阻止行选择操作。
- tooltip：在图标上显示的提示信息。

下面通过一个示例演示如何使用列对象定义单元格的显示格式。

（1）功能描述

将下面代码中 Store 的数据以指定格式显示：

```
var store=Ext.create("Ext.data.ArrayStore",{
   fields:[{name:"id",type:"int"},
           "name","nationality",
           {name:"birthday",type:"date",format:"Y-m-d"},
           "photo",{name:"performance",type:"float"},
           {name:"editable",type:"bool"},
           {name:"delete",type:"bool"},
   ],
   data:[
           [1,"张三","cn.gif","1948-03-02","head-1.jpg","10000.12",true,false],
           [2,"李四","us.gif","1960-04-03","head-2.jpg","869.7",true,true],
           [3,"王五","fr.gif","1999-08-13","head-3.jpg","3443.3",false,false]
   ]
});
```

格式要求：

- 显示行号。
- 在显示姓名的同时显示国籍。
- birthday 字段显示格式如：“1999 年 9 月 9 日”。
- performance 字段以货币形式显示。
- editable 字段为 true 时显示允许，为 false 时显示不允许。
- delete 字段使用复选框显示。
- 显示查看、编辑和删除 3 个图标，editable 为 false，显示灰色图标，不执行操作；delete 为 false 时，显示灰色图标，不执行操作。

（2）实现代码

使用模板页创建一个名称为 10-2.html 的页面文件，因要使用 CheckColumn 对象，所以要先加入对象使用的样式文件：

```
<link rel="stylesheet" type="text/css" href="../Ext4/examples/ux/css/CheckHeader.
    css" />
```

接着定义动态加载，加载 CheckColumn.js 文件：

```
Ext.Loader.setConfig({enabled: true});
Ext.Loader.setPath({
   "Ext.ux":"../ext4/examples/ux/"
```

```
});
Ext.require([
    'Ext.ux.CheckColumn'
]);
```

第一句开启动态加载，然后使用 setPath 方法设置目录位置，最后请求文件。

在 ActionColumn 中显示图标，有直接定义 icon 和定义图标样式两种方式，在本示例中将使用图标样式的方式，因而需要定义两个样式，代码如下：

```
.edit{background:url("../images/edit.png")no-repeat;height:16px;width:16px;}
.delete{background:url("../images/del.png")no-repeat;height:16px;width:16px;}
```

图标禁用状态可通过 Ext JS 的样式“x-item-disable”来实现，不需要自己做处理。

接着要做的和 10.2.3 节示例一样，初始化 QuickTips 对象，定义 Store，再定义一个 Grid，要将 Grid 的宽度改为 500，否则宽度不够。下面开始定义列，先定义行号，这个简单，代码如下：

```
{xtype:"rownumberer",width:30},
```

接着定义姓名列，这里要显示两个字段，可使用 renderer 方法组织 HTML 代码，不过本示例会使用模板列实现，其代码如下：

```
{xtype:"templatecolumn",text:'姓名',dataIndex:"name",flex:1.5,
   tpl:["<span data-qtip='<img src=\"../images/{photo}\">'>{name}</span>"," (<img
       src='../images/flag/{nationality}'/>) "]
},
```

只要定义配置项 tpl 就行了，比 renderer 简单，不过不能在 td 上添加提示信息了，只能在模板内通过添加 HTML 标记解决。

接着使用 DateColumn 对象显示生日，只要定义好配置项 format 就行：

```
{xtype:"datecolumn",text:"生日",dataIndex:"birthday",flex:2.5,format:"Y年m月d日"},
```

使用 NumberColumn 对象显示业绩也很简单，定义好配置项 format 就行：

```
{xtype:"numbercolumn",text:"业绩",dataIndex:"performance",flex:1.5,format:"¥0,0.0
    0",titleAlign:"center",align:"right"},
```

显示 editable 字段，要多下点工夫，需要定义配置项 trueText 和 falseText：

```
{xtype:"booleancolumn",text:"编辑？",dataIndex:"editable",flex:1,
   trueText:"允许",falseText:"不允许"
},
```

显示 delete 字段就简单多了，基本没什么额外配置（除了加载步骤比较多）：

```
{xtype:"checkcolumn",text:"删除？",dataIndex:"delete",flex:1,
   listeners:{
          checkchange:function(obj,row,v){
                 var ac=obj.nextNode("actioncolumn"),
                        rowNode=obj.up("gridpanel").view.getNode(row),
                        el=Ext.fly(rowNode).select('.x-action-col-2');
```

```
                if(v){
                        el.removeCls(ac.disabledCls);
                }else{
                        el.addCls(ac.disabledCls);
                }
        }
    }
},
```

因 Ext JS 4.1 Beta 1 的渲染模式发生了改变，直接单击 checkcolumn 列更改值后不会刷新操作列，会造成操作列的图标不能更新，因而必须在该列绑定 checkchange 事件，当其值改变的时候，通过返回的行号（row）找到该行的元素，然后通过图标的样式找到图标，再调用 removeCls 方法移除禁用图标样式或调用 addCls 方法添加禁用图标样式。在 ActionColumn 对象的 renderer 方法中，每一个操作生成的 HTML 代码都会添加“x-action-col-i”（i 为操作索引）的样式，因而可以根据该样式获取这些图标。

如果不需要根据每一行数据设置操作，而是统一控制，则可通过 ActionColumn 对象的 enableAction 方法和 disableAction 方法来开启或禁用操作。

最后完成的是显示操作图标了，因为要显示 3 个图标，所以要在 items 配置项里进行定义。查看图标没特殊要求，因而直接定义配置项 icon 显示图标就行。编辑和删除图片都要根据字段值进行图标切换，因而要定义 getClass 配置项。代码如下：

```
{xtype:"actioncolumn",text:"操作",flex:2,align:"center",items:[
    {altText:"查看",tooltip:"查看",icon:"../images/view.png",
            handler:function(view, row, col, item, e){
                console.log("查看: [",row,",",col,"]");
        }
},
{altText:"编辑",tooltip:"编辑",getClass:function(v,meta,r){
                var css="edit";
                if(!r.data.editable){
                        css="x-item-disabled edit";
                }
                return css;
        },
        handler:function(view, row, col, item, e){
                var rec=view.getStore().getAt(row);
                if(!rec.data.editable){

                        console.log("不允许编辑")
                        return;
                }
                console.log("允许编辑");
        }
},
{altText:"删除",tooltip:"删除",getClass:function(v,meta,r){
                var css="delete";
                if(!r.data.delete){
                        css="x-item-disabled delete";
                }
                return css;
```

```
            },
            handler:function(view, row, col, item, e){
                var rec=view.getStore().getAt(row);
                if(!rec.data.delete){
                    console.log("不允许删除")
                    return;
                }
                console.log("允许删除");
            }
        }
    ]}
```

先看看图标编辑和删除的 getClass 方法，两个函数大同小异，都是先默认设置正常显示，然后根据值的情况修改其样式。如果要禁用，则添加禁用样式。

因为没有设置操作的 disable 状态，不能在对象内部的处理中屏蔽单击事件，所以只能在单击句柄中忽略字段值为 false 的情况，直接使用 return 语句返回即可。

至此示例就完成了，可以调试了。

（3）页面效果

在浏览器中打开页面，然后尝试单击第一行的编辑列或删除列，删除列是允许直接修改字段值，而且它的改变会导致删除图标的显示效果改变，从图 10-4 可以看到这些变化。单击一下查看、编辑和删除图标，在控制台面板会依次输出“查看：[0,6]”、“允许编辑”和“允许删除”。

图 10-4　通过列对象定义单元格的显示格式示例的效果

10.2.5　设置行的背景颜色

很多时候，需要根据列中的某个数据修改行的背景颜色，这可通过视图的 addRowCls 方法实现，它会给行添加一个样式类。设置行的背景颜色的关键是它要在什么时候设置，在列的 renderer 函数内并不合适，因为列还在生成。最好是在视图的数据渲染完毕，也就是在 refresh 事件中。例如，在 10.2.3 节的示例中，要将年龄大于 60 岁的背景色设置为绿色，可先在页面定义一个样式：

```
.green{background:green;}
```

然后在 Grid 的 viewConfig 中监听 refresh 事件：

```
viewConfig:{
    listeners:{
            refresh:function(){
                    var me=this,
                            data=me.store.data.items;
                    for(var i=0;ln=data.length,i<ln;i++){
                            var n=new Date(),
                    age=n.getFullYear()-data[i].data.birthday.getFullYear();
                            if(age>=60){
                                    me.addRowCls(data[i],"green")
                            };
                    }
            }
    }
}
```

计算出年龄大于 60 后，使用 addRowCls 方法添加一个样式就行了。

代码运行后，可看到 Grid 第一行的 tr 标记代码为：

```
<tr class="x-grid-row green">
```

可以看到，使用 addRowCls 方法添加的样式已经添加到 tr 标记了。

以上方法只适用于 4.0.7 版本，在 4.1 Beta 1 中，视图的样式已经修改成为每一个单元格设置背景颜色，会覆盖 tr 标记的背景，所以，必须通过修改每个 td 的背景颜色来实现。如果使用脚本去实现，则需要获取该行所有 td 元素，然后修改其背景，这就比较麻烦了，最简单的方法是添加以下样式：

```
.green td{background:green !important;}
```

粗体代码是关键代码，如果没有该定义，样式会被 td 内的背景颜色替换掉，添加了该定义后，它就会获得优先权，替代 td 内设置的背景颜色，从而实现修改行的背景颜色。

10.2.6　列标题的分组

要将列标题分组，只要在定义列的配置对象时，加入 columns 配置项，然后在其内部定义列就可以了。例如以下数据：

```
var store=Ext.create("Ext.data.ArrayStore",{
    fields:["name","Q1","Q2","Q3","Q4"],
    data:[
            ["广东",10000,20000,30000,20000],
            ["广西",15000,12000,40000,10000]
    ]
});
```

要将 Q1、Q2、Q3 和 Q4 字段分在一组，可这样定义 Grid：

```
Ext.create("Ext.grid.Panel",{
    renderTo:Ext.getBody(),
```

```
        width:400,
        height:200,
        store:store,
        columns:[
                {text:"地区",dataIndex:"name",width:40,},
                {text:"销售业绩",columns:[
                        {text:"第一季度",dataIndex:"Q1",width:80},
                        {text:"第二季度",dataIndex:"Q2",width:80},
                        {text:"第三季度",dataIndex:"Q3",width:80},
                        {text:"第四季度",dataIndex:"Q4",width:80}
                ]}
        ]
})
```

在定义 Grid 的第二列时，使用 columns 配置项来定义其子列，就可实现列的分组了。打开模板页，在命令行中运行以上代码，可看到如图 10-5 所示的结果。

地区	销售业绩			
	第一季度	第二季度	第三季度	第四季度
广东	10000	20000	30000	20000
广西	15000	12000	40000	10000

图 10-5　列的分组示例效果

10.2.7　使用锁定列

在 Lockable.js 的注释中，要实现列锁定的功能，可在 GridPanel 中定义配置项 lockable 为 true，不过在代码中却发现不了它的身影，估计其完整的功能还在开发中，这个从 API 中就可以看到。因而目前只能在列中添加配置项 loecked 来显示锁定功能，在列标题菜单中会出现锁定（Lock）与解锁（Unlock）的子菜单，而且还没有本地化，要本地化，可在本地化文件中加入以下代码：

```
if(Ext.grid.Lockable){
    Ext.apply(Ext.grid.Lockable.prototype,{
        unlockText: '解锁',
            lockText: '锁定'
    })
}
```

例如在 10.2.5 节的例子的姓名列定义加入 locked 配置项，其值为 false，可看到如图 10-6 所示的结果，在列标题菜单中加入了锁定与解锁菜单。在 Firebug 的 Illuminations 面板，展开 GridPanel，可看到如图 10-7 所示的结果，虽然现在没有锁定列，但还是创建了锁定面板，锁定面板占位就是为什么在图 10-6 的左边会有缝隙的原因。

图 10-6　带锁定功能的 Grid

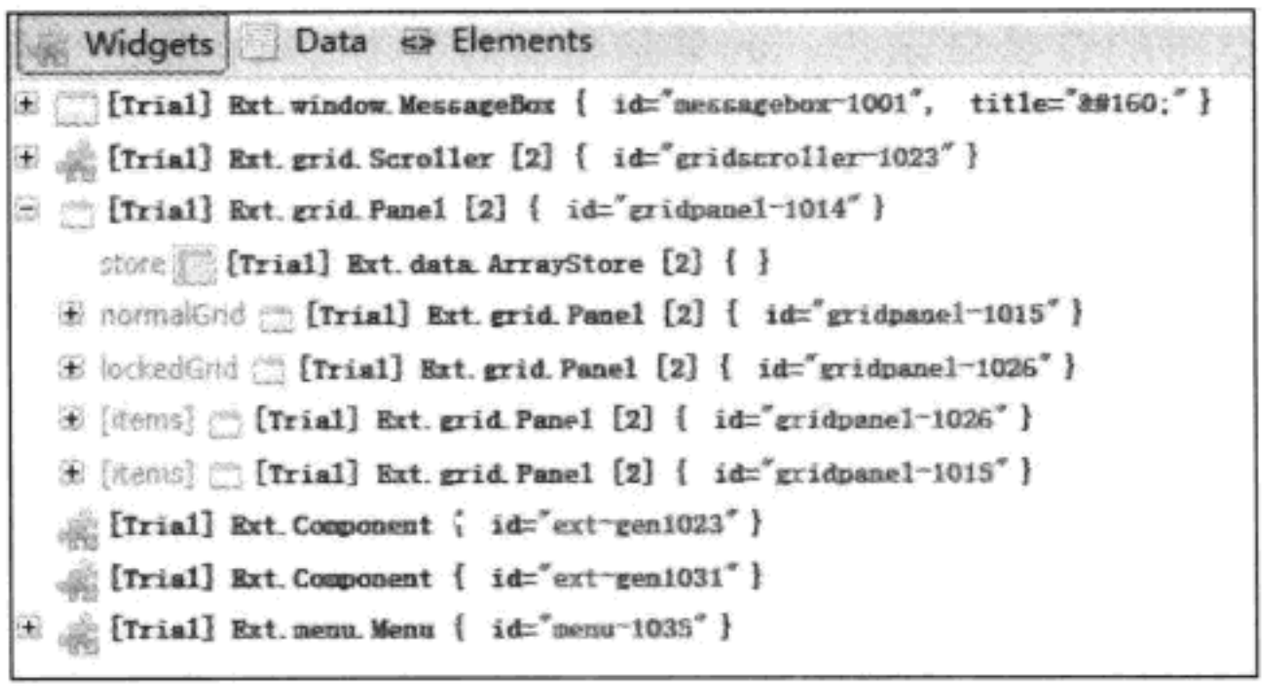


图 10-7　Illuminations 面板的显示结果

锁定列还在开发完善中，会存在问题，在使用时要考虑清楚是否有这个必要再使用，例如示例中，把季度列锁定后，要想将它解锁后回到分组里面就不可能了。

10.2.8　Grid 的配置项、属性、方法和事件

Grid 的配置项主要由其构成部分的配置项组合而成，而大部分配置项是来源于 TabPanel 和 TableView 的。

1. 配置项

- columns：必需的配置项，用来定义 Grid 中的列。
- deferRowRender：布尔值，默认值为 true，会延迟行的渲染。这样会让 GridView 执行 refresh 时更快速，行的结构会延迟到 GridPanel 执行布局时才渲染，这样就使 Grid 的显示更快速。因而，在没有特殊需求的情况下，不要修改该值为 false。
- enableColumnHide：布尔值，默认值为 true，允许列隐藏。
- enableColumnMove：布尔值，默认值为 true，允许列移动。
- enableColumnResize：布尔值，默认值为 true，允许调整列宽度。
- featrues：数组，数组的元素为附加功能的配置对象。
- forceFit：布尔值，默认值为 undefined。如果设置为 true，规定列的总宽度完全填满 Grid 的宽度。

- hideHeaders：布尔值，默认值为 undefined。如果设置为 true，会隐藏列标题。
- scroll ：默认值为 true，会创建垂直和横向两个滚动条（不一定显示）。值可以为 false、horizontal、vertical 或 both。值为 both 与 true 的作用一样 ；值为 false 时，不会创建滚动条；值为 horizontal 时，只创建水平滚动条；值为 vertical 时，只创建垂直滚动条。
- scrollDelta：数组，表示鼠标滚轮滚动时会滚动多少像素，默认值为 40。
- sortableColumns：布尔值：默认值为 true，允许单击列标题或通过列标题菜单进行排序。
- columnLines ：这个是 GridPanel 提供的配置项，布尔值，默认值为 undefined，不显示列之间的分隔线。设置为 true，可显示列之间的分隔线。
- viewConfig ：视图的配置对象。主要配置项是 stripeRows，如果其值为 true，则会在视图中使用交错的背景显示行。如果是带有锁定功能的 Grid，可使用配置项 lockedViewConfig 配置锁定视图，使用配置项 normalViewConfig 配置正常视图。

2. 属性

- hasView：如果面板已经注入了视图，返回 true；否则返回 false。

3. 方法

面板的方法：

- determineScrollbars：重新计算滚动条的大小与位置，并放置它们。
- getSelectionModel：返回选择模型。
- getStore：返回 Store。
- getView：返回视图。
- hideHorizontalScroller：隐藏水平滚动条。
- hideVerticalScroller：隐藏垂直滚动条。
- initHorizontalScroller：返回水平滚动条的配置对象。
- initVerticalScroller：返回垂直滚动条的配置对象。
- invalidateScroller ：当前滚动条失效的时候，规定其重新进行计算，这与滚动条的显示或隐藏无关。
- reconfigure ：可通过重新配置 Store 或列模型，重新生成 Grid，也就是说，Grid 可以不用更换 GridPanel 对象就更新显示了。
- scrollByDeltaX：让水平滚动条滚动指定的距离。
- scrollByDeltaY：让垂直滚动条滚动指定的距离。
- setScrollTop：设置垂直滚动条的 scrollTop 值。
- showHorizontalScroller：显示水平滚动条。
- showVerticalScroller：显示垂直滚动条。

视图的方法：

- addRowCls：为指定行添加一个样式类。
- focusRow：使指定行获得焦点。
- getFeatrue：返回指定的附加功能对象。

- getRowClass：可重写该方法返回应用于行的样式。
- getTableChunker：返回 TableChunker 对象的配置对象。
- refresh：刷新视图。
- removeRowCls：移除指定行的指定的样式。
- saveScrollState：保存滚动条的状态。

4. 事件

Grid 的主要事件是从视图和选择模型传播过来的，因而有关事件就不列出来了，具体可参考 9.6 节中选择模型和视图的事件。

Grid 本身定义的事件有 3 个：

- reconfigure：Grid 使用 reconfigure 重新构建后会触发该事件。
- scrollerhide：滚动条隐藏时会触发该事件。
- scrollershow：滚动条显示时会触发该事件。

10.3　Grid 的附加功能

10.3.1　概述

为了减少 Grid 的 HTML 代码，Ext JS 4 不再像 Ext JS 3.0 那样，把所有功能的 HTML 代码都加上，而是把这些功能作为附加功能，由开发人员决定在 Grid 上附加什么功能，然后再在 Grid 中渲染这些附加功能的 HTML 代码。

由图 10-8 可以看到，附加功能类的基类派生于 Observable 类，然后从其派生出 3 个子类，总共由 6 个类组成。

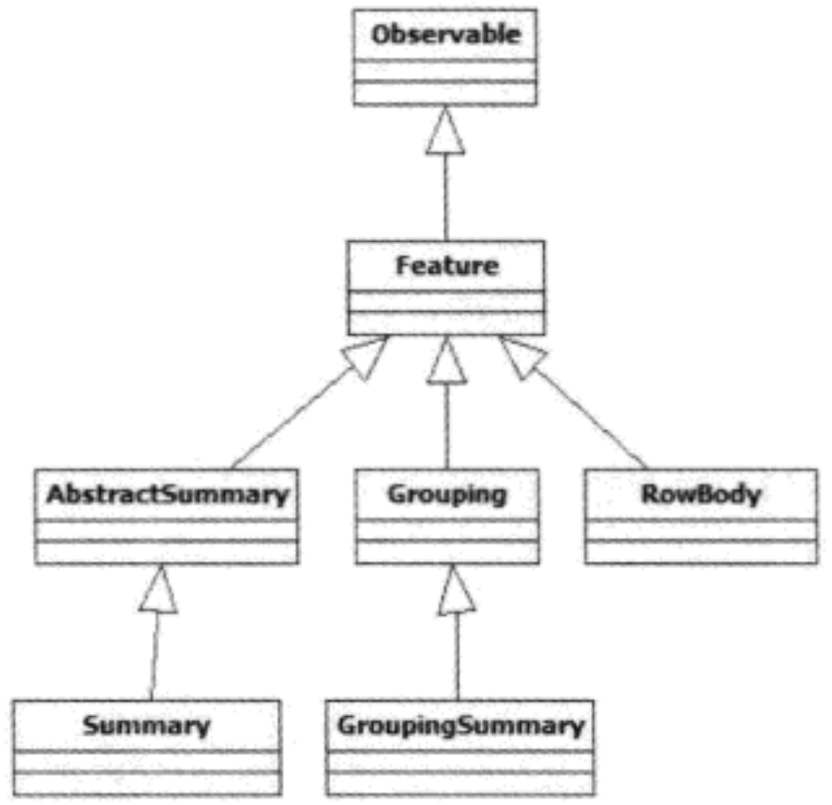


图 10-8　附加功能类结构图

注意　Ext.grid.featrue.Chunking 类虽然也是附加功能类，但没有说明，也没有具体使用的例子，估计还在开发中，所以在本书暂时未把它列在附加功能类里。

10.3.2　附加功能基类：Ext.grid.featrue.Featrue

Featrue 对象没有构造函数，其作用是为开发人员提供几个挂钩，用于将附加功能注入到 Grid 中。这有点类似于模板，不过它不是提供 HTML 代码，而是提供基本的属性和方法。

1. 提供的属性

- collectData：在视图调用 collectData 方法收集数据的时候，如果该值为 false，则说明附加功能不会返回数据给视图。如果需要给视图返回数据，则需要将其重写为方法，

在视图的 collectData 方法中调用附加功能的 collectData 方法。

- eventPrefix：在视图触发事件时的前缀。例如前缀为 group，则事件为 groupclick、groupdbclick。
- eventSelector：当有事件前缀的事件被触发时，用来确定元素的选择符。
- grid：指向面板。
- hasFeatrueEvent：如果附加功能有附加事件要传播到面板或视图，则需要在这里定义这些事件；如果没有，则设置该值为 false。
- view：指向视图。

2. 提供的方法

- disable：禁用附加功能。
- enable：启用附加功能。
- getAdditionalData：在视图调用 prepareData 方法时，为其提供附加数据。
- getFireEventArgs：附加特性需要重写该方法，为事件添加附加的参数。
- getMetaRowTplFragments：该方法可将附加功能 HTML 代码注入到行模板中（metaRowTpl）。
- mutateMetaRowTpl：该方法允许附加功能修改行模板（metaRowTpl）。

10.3.3 为行添加附加信息：Ext.grid.featrue.RowBody

RowBody 对象的作用是在 Grid 原有的行下插入一行，用来显示附加的信息，加入行的 HTML 代码到 getRowBody 方法内：

```
getRowBody: function(values) {
    return [
        '<tr class="' + this.rowBodyTrCls + ' {rowBodyCls}">',
            '<td class="' + this.rowBodyTdCls + '" colspan="{rowBodyColspan}">',
                '<div class="' + this.rowBodyDivCls + '">{rowBody}</div>',
            '</td>',
        '</tr>'
    ].join('');
},
```

可以看到，行内只有一个单元格，可以根据 rowBodyColspan 的值合并多个单元格。而要显示的内容由 rowBody 的值提供，还有一个用于行的样式的值由 rowBodyCls 提供，因而，要使用 RowBody 对象就必须提供这 3 个字段值。而为视图提供附加数据的方法是 getAdditionalData 方法，因此，要使用 RowBody 对象，重写 getAdditionalData 方法就行了。在原有基础上重写方法就行，原有的代码如下：

```
getAdditionalData: function(data, idx, record, orig) {
    var headerCt = this.view.headerCt,
        colspan  = headerCt.getColumnCount();

    return {
        rowBody: "",
```

```
            rowBodyCls: this.rowBodyCls,
            rowBodyColspan: colspan
        };
    }
```

可以看到，方法会一次接收 4 个参数，第 1 个参数是记录 data 属性，也就是可以通过 data 直接访问记录的字段；第 2 个参数是行的索引值；第 3 个参数是当前行的记录；第 4 个参数是未经 prepareData 方法处理的原始记录。

代码开始先获取了视图的列标题容器，然后获取列数，这将是 RowBody 对象插入的行要合并的单元格数量，这个重写的时候可以不修改。

最后是返回包含 3 个数据字段的对象，我们要做的就是修改 rowBody 的值；需要修改行样式时就修改 rowBodyCls 的值。但一般情况下不修改 rowBodyColspan 的值。

下面实践一下如何使用 RowBody 对象，在 10.2.3 节的示例中，添加一行用来显示头像，在 Grid 的定义中添加以下代码就可以了：

```
featrues:[
   {ftype:"rowbody",
       getAdditionalData: function(data, idx, record, orig) {
           var headerCt = this.view.headerCt,
               colspan  = headerCt.getColumnCount();

           return {
               rowBody: "<img src='../images/"+data.photo+"'>",
               rowBodyCls: this.rowBodyCls,
               rowBodyColspan: colspan
           };
       }
   }
],
```

要注意，附加功能的配置对象中，对象类型使用的是 ftype 配置项。代码只是修改了原有代码 rowBody 的值，不过这已经可以实现需要的功能了。在浏览器中打开示例 10-1，可看到如图 10-9 所示的结果。

图 10-9　使用 RowBody 对象后的页面效果

10.3.4　数据汇总功能：Ext.grid.featrue.AbstractSummary 与 Ext.grid.featrue.Summary

数据汇总功能包括一个抽象基类和实体类。抽象基类 AbstractSummary 提供汇总的计算方法、如何使用模板渲染汇总数据以及汇总行的显示与隐藏等基本功能。Summary 对象则在抽象基类的基础上通过重写接口方法将数据渲染到视图。

具体过程是 TableChunker 对象在 getTableTpl 方法中调用 Summary 对象的 getFragmentTpl 方法，getFragmentTpl 方法会调用 generateSummaryData 方法获取汇总数据，其代码如下：

```
generateSummaryData: function(){
    var me = this,
        data = {},
        store = me.view.store,
        columns = me.view.headerCt.getColumnsForTpl(),
        i = 0,
        length = columns.length,
        fieldData,
        key,
        comp;
    for (i = 0, length = columns.length; i < length; ++i) {
        comp = Ext.getCmp(columns[i].id);
        data[comp.id] = me.getSummary(store, comp.summaryType, comp.dataIndex,
            false);
    }
    return data;
}
```

代码会遍历所有列，调用 getSummary 方法获取每列的汇总数据。注意粗体代码，getSummary 方法会根据列的配置项 summaryType 调用 Store 中对应的方法进行数据汇总，例如 summaryType 的值为 count，则会调用 Store 的 count 方法统计数据的记录数。

数据准备好以后，就会调用 getSummaryFragments 方法，其代码如下：

```
getSummaryFragments: function(){
    var fragments = {};
    if (this.showSummaryRow) {
        Ext.apply(fragments, {
            printSummaryRow: Ext.bind(this.printSummaryRow, this)
        });
    }
    return fragments;
},
```

在这里会构建一个对象，成员只有 printSummaryRow 一个，它会调用 printSummaryRow 方法，而这个对象也会作为 getFragmentTpl 方法的返回值返回。返回后，会将该对象的成员复制到 tplMemberFns 对象中。对象 tplMemberFns 是模板的配置对象，也就是说，printSummaryRow 方法会成为模板的一个成员函数，在模板渲染的时候会调用该方法输出汇总信息，其代码如下：

```
printSummaryRow: function(index){
    var inner = this.view.getTableChunker().metaRowTpl.join('');
inner = inner.replace('x-grid-row', 'x-grid-row-summary');
inner = inner.replace('{{id}}', '{gridSummaryValue}');
inner = inner.replace(this.nestedIdRe, '{id$1}');
inner = inner.replace('{[this.embedRowCls()]}', '{rowCls}');
inner = inner.replace('{[this.embedRowAttr()]}', '{rowAttr}');
inner = Ext.create('Ext.XTemplate', inner, {
    firstOrLastCls: Ext.view.TableChunker.firstOrLastCls
});
```

```
        return inner.applyTemplate({
            columns: this.getPrintData(index)
        });
    },
```

它会替换掉原行模板中一些属性，然后将数据应用到模板后再返回。方法 getPrintData 可将计算好的数据组织成数组返回。

这样在就会在表格闭合前添加一个汇总列。

要使用汇总功能，首先要在 Grid 的配置项 featrues 中添加 Summary 对象的配置项对象，也就是定义一下 ftype 配置项而已。最后要做的是在要汇总的列上添加配置项 summaryType 指明统计类型，其值可以是 count（计数）、sum（总和）、min（最小值）、max（最大值）或 average（平均值）。汇总列也是可以格式化显示的，只要定义配置项 summaryRenderer 就可以了。格式化函数依次可接收 value、summaryData 和 field 这 3 个参数，参数 value 表示当前列的汇总值；参数 summaryData 包含各列的汇总值；field 是当前列的字段名称。

下面实践一下如何使用汇总功能，利用 10.2.5 节的示例代码，先为 Grid 添加汇总功能：

```
featrues:[{ftype:"summary"}],
```

然后修改列定义如下：

```
columns:[
    {text:"地区",dataIndex:"name",width:80,
            summaryType:"count",
            summaryRenderer:function(v,d,f){
                return "合计: "+v;
            }
    },
    {text:"销售业绩",columns:[
            {text:"第一季度",dataIndex:"Q1",width:80,summaryType:"sum"},
            {text:"第二季度",dataIndex:"Q2",width:80,summaryType:"sum"},
            {text:"第三季度",dataIndex:"Q3",width:80,summaryType:"sum"},
            {text:"第四季度",dataIndex:"Q4",width:80,summaryType:"sum"}
    ]},
    {text:"合计",width:80,
            renderer:function(v,meta,rec){
                return (rec.data.Q1+rec.data.Q2+rec.data.Q3+rec.data.Q4)
            },
            summaryRenderer:function(v,sd,f){
                var list=["Q1","Q2","Q3","Q4"],
                    sum=0;
                for(var id in sd){
                    if(sd.hasOwnProperty(id)){
                    var cmp = Ext.getCmp(id);
                    if(cmp && list.indexOf(cmp.dataIndex)>=0){
                        console.log(sd[id]);
                        sum+=parseFloat(sd[id]);
                    }
                    }
                }
                return sum;
    }
```

```
        },
        {text:"平均",width:80,summaryType:"sum",
                renderer:function(v,meta,rec){
                        return (rec.data.Q1+rec.data.Q2+rec.data.Q3+rec.data.Q4)/4
                },
                summaryRenderer:function(v,sd,f){
                        var list=["Q1","Q2","Q3","Q4"],
                                sum=0;
                        for(var id in sd){
                                if(sd.hasOwnProperty(id)){
                                 var cmp = Ext.getCmp(id);
                                if(cmp && list.indexOf(cmp.dataIndex)>=0){
                                        console.log(sd[id]);
                                        sum+=parseFloat(sd[id]);
                                }
                                }
                        }
                        return sum/4;
                }
        }
    ]
```

代码中，“地区”这列使用了计数汇总，并且显示为“合计：”加上总数；4 个季度列则直接使用求总和方式汇总；接着虚拟了两列，第一列会显示 4 个季度的总和，第二列会显示平均值。因为是虚拟列，没有合适的字段，所以显示的值要在 renderer 配置项中计算出来。两个虚拟列的汇总也存在没有具体数据字段、不能汇总的情况，因而也要在 summaryRenderer 配置项中进行计算处理。在 summaryRenderer 配置项中进行计算比较麻烦的是，summaryData 返回的值格式是以列标题的 id 为属性名称的，因而需要先使用 getCmp 方法返回列对象，然后检查其字段名称（dataIndex）是否是 4 个季度的字段名称，如果是，就将汇总值求和。最后返回计算好的值。

为了完整显示全部列，还要修改 Grid 的 width 值为 600，代码运行后会显示如图 10-10 的结果。

地区	销售业绩				合计	平均
	第一季度	第二季度	第三季度	第四季度		
广东	10000	20000	30000	20000	80000	20000
广西	15000	12000	40000	10000	77000	19250
合计：2	25000	32000	70000	30000	157000	39250

图 10-10　汇总功能的示例显示结果

AbstractSummary 对象提供了配置项 showSummaryRow 用来确定是否显示汇总行，其默认值为 true，显示汇总行。可通过方法 toggleSummaryRow 来切换汇总行的显示状态。

AbstractSummary 对象还提供了 getColumnValue 方法返回列的汇总数据。

10.3.5　分组功能：Ext.grid.featrue.Grouping

Grouping 对象可分组显示 Grid 中的数据，实现的方法是在 TableChunker 对象的 metaTableTpl 模板定义中，通过 embedFeatrue 成员函数，调用 Grouping 对象的 getFeatrueTpl 方法，将分组的 HTML 代码嵌入模板。方法 getFeatrueTpl 的代码如下：

```
getFeatrueTpl: function(values, parent, x, xcount) {
    var me = this,
        tpl = me.groupHeaderTpl;

if(Ext.isString(tpl)) {
    tpl = new Ext.XTemplate(tpl);
} else if(Ext.isArray(tpl)) {
    tpl = Ext.Array.clone(tpl);
    Ext.Array.splice(tpl, 0, 0, 'Ext.XTemplate');
    tpl = Ext.create.apply(Ext, tpl);
}
me.groupHeaderTpl = tpl;

return [
    '<tpl if="typeof rows !== \'undefined\'">',
        '<tr class="' + Ext.baseCSSPrefix + 'grid-group-hd ' + (me.
            startCollapsed ? me.hdCollapsedCls : '') + ' {hdCollapsedCls}"><td
            class="' + Ext.baseCSSPrefix + 'grid-cell" colspan="' + parent.
            columns.length + '" {[this.indentByDepth(values)]}><div
            class="' + Ext.baseCSSPrefix + 'grid-cell-inner"><div class="'
            + Ext.baseCSSPrefix + 'grid-group-title">{collapsed}{[this.
            renderGroupHeaderTpl(values)]}</div></div></td></tr>',
        '<tr id="{viewId}-gp-{name}" class="' + Ext.baseCSSPrefix + 'grid-
            group-body ' + (me.startCollapsed ? me.collapsedCls : '') +
            ' {collapsedCls}"><td colspan="' + parent.columns.length +
            '">{[this.recurse(values)]}</td></tr>',
    '</tpl>'
].join('');
},
```

从代码可以看到，getFeatrueTpl 方法在 metaTableTpl 中加入了两行，一行用来显示分组标题，而第二行粗体代码调用了模板的成员函数 recurse 方法，其定义如下：

```
Ext.XTemplate.prototype.recurse = function(values, reference) {
    return this.apply(reference ? values[reference] : values);
};
```

代码为 XTemplate 的原型添加了一个 recurse 方法，方法的目的是使用指定的值递归调用模板，也就是说，在第二行内会使用指定的值重新生成一个表格用来显示数据，这就是第二行只有一个单元格的原因。

这样就很清楚是怎么分组的了，从 Store 中取得分组的字段，然后根据字段获取组内的数据，渲染的时候，组标题在第一行显示，而分组数据通过递归方式调用模板，将分组数据渲染到第二行。

这样的好处就是分组数据可轻松实现折叠和展开，如果在同一表格的话，要折叠或展开

分组数据，必须一行行的设置其显示或隐藏，比较麻烦，现在只要将第二行隐藏或显示就可实现折叠与展开功能了。

要实现分组，首先要做的是在 Store 里定义分组字段，然后在 Grid 中设置分组功能，例如要将以下数据根据省份分组显示：

```
var store=Ext.create("Ext.data.ArrayStore",{
    fields:["area","province","Q1","Q2","Q3","Q4"],
    groupField:"province",
    data:[
            ["广州","广东",20000,50000,30000,10000],
            ["佛山","广东",10000,20000,30000,20000],
            ["南宁","广西",15000,12000,40000,10000],
            ["桂林","广西",25000,22000,30000,12000]
    ]
});
```

要分组显示数据，在定义 Store 时，就要定义 groupField 配置项，这里要根据省份分组，因而字段为 province。

最简单的分组 Grid 就是将分组功能加入 Grid 就行了：

```
Ext.create("Ext.grid.Panel",{
    renderTo:Ext.getBody(),
    width:600,
    height:300,
    store:store,
    featrues:[{ftype:"grouping"}],
    columns:[
            {text:"地区",dataIndex:"area",width:80},
            {text:"销售业绩",columns:[
                    {text:"第一季度",dataIndex:"Q1",width:80},
                    {text:"第二季度",dataIndex:"Q2",width:80},
                    {text:"第三季度",dataIndex:"Q3",width:80},
                    {text:"第四季度",dataIndex:"Q4",width:80}
            ]}
    ]
})
```

打开模板页，在命令行中运行以上代码可看到如图 10-11 所示的结果。

地区	销售业绩			
	第一季度	第二季度	第三季度	第四季度
Group: 广东				
广州	20000	50000	30000	10000
佛山	10000	20000	30000	20000
Group: 广西				
南宁	15000	12000	40000	10000
桂林	25000	22000	30000	12000

正序
倒序
列
Group By This Field
Show in Groups

图 10-11　分组 Grid 的显示结果

从图 10-12 中可以看到，Grouping 对象的默认文本还没有本地化，只要在本地化文件中

加入以下代码即可实现：

```
if(Ext.grid.featrue.Grouping){
    Ext.apply(Ext.grid.featrue.Grouping.prototype,{
        groupHeaderTpl: '分组：{name}',
            groupByText: '使用本字段进行分组',
            showGroupsText:'分组显示'
    })
}
```

在 Grouping 对象中有 8 个配置项用来配置分组显示：

- depthToIndent：数组，表示分组层级的缩进量。
- enableGroupingMenu：布尔值，默认值为 true，会在列标题开启分组菜单。
- enableNoGroups：布尔值，默认值为 true，表示可以关闭分组。
- groupByText：列标题分组菜单的文本，本地化后文本为“使用本字段进行分组”。
- groupHeaderTpl ：分组标题显示内容，会使用模板渲染，本地化后显示内容为“分组 ：{name}”，name 会使用字段名称替换。
- hideGroupedHeader：布尔值，为 true 时会隐藏分组标题。默认值为 false。
- showGroupsText ：显示在列标题菜单中用于启用或禁用分组的菜单文本，本地化后，默认为“分组显示”。
- startCollapsed ：布尔值，默认值为 false，在开始时分组是展开的。如果设置为 false，则在开始时分组是折叠的。

Grouping 对象还有自己的事件：

- groupclick：单击分组时触发该事件。
- groupcollapse：分组折叠后触发该事件。
- groupcontextmenu：在分组按下右键时会触发该事件。
- groupdbclick：双击分组时触发该事件。
- groupexpand：分组展开后会触发该事件。

10.3.6　分组汇总功能：Ext.grid.featrue.GroupingSummary

GroupingSummary 对象派生于 Grouping 对象，又混入了 AbstractSummary 对象，因而是分组与汇总的结合体，也就是说，GroupingSummary 对象既可使用 Grouping 对象的配置项、属性和方法，又可以使用 AbstractSummary 对象的配置项、属性和方法。

GroupingSummary 对象是在 Grouping 对象插入了两行之后，插入一行来渲染汇总数据，具体代码如下：

```
getFeatrueTpl: function() {
    var tpl = this.callParent(arguments);

    if (this.showSummaryRow) {
        tpl = tpl.replace('</tpl>', '');
        tpl += '{[this.printSummaryRow(xindex)]}</tpl>';
    }
```

```
        return tpl;
    },
```

在调用 Grouping 对象的 getFeatrueTpl 方法获得模板代码后，替换掉“</tpl>”，再调用成员方法 printSummaryRow 生成汇总行插入到 Grouping 对象生成的分组行之后，之后基本就是按 AbstractSummary 对象的流程进行的。

把上一节分组功能示例中的 ftype 修改为 groupingsummary，然后在地区列加入配置项 summaryType，其值为 count；在 4 个季度列上加入配置项 summaryType，其值为 sum，在命令行中运行后可看到如图 10-12 所示的结果。

地区	销售业绩			
	第一季度	第二季度	第三季度	第四季度
分组：广东				
广州	20000	50000	30000	10000
佛山	10000	20000	30000	20000
2	30000	70000	60000	30000
分组：广西				
南宁	15000	12000	40000	10000
桂林	25000	22000	30000	12000
2	40000	34000	70000	22000

图 10-12　GroupingSummary 的示例效果

10.4　可编辑的 Grid

10.4.1　概述

在 Ext JS 4 中，取消了 Ext.grid.EditorGridPanel 对象，取而代之的是在 Grid 内部通过插件的方式实现编辑功能。这样的好处是减少 Grid 与可编辑功能之间的耦合度，在 Ext JS 3 中，EditorGridPanel 是继承自 GridPanel 的，所以可以说是全耦合的，也就是 Grid 的变动，会直接影响 EditorGridPanel，可能要修改的代码就很多了。而插件的方式，就没有这个麻烦了，只要根据 Grid 接口修改就行了，甚至有可能根本就不需要修改。

Grid 要实现编辑功能有两种方式，一种是使用 CellEditing 对象直接编辑单元格，一种是使用 RowEditing 对象对整行进行编辑，这两个对象都派生于 Editing 对象。Editing 对象只是负责管理编辑状态和编辑组件的对象，而不是具体的编辑组件。因而还需要编辑组件 CellEditor，它派生于 Editor [⊖]组件。

⊖　相关信息请阅读 10.4.3 节。

10.4.2　Grid 实现可编辑功能的运行流程：Ext.grid.plugin.Editing

Editing 对象派生于 Observable 对象，其构造函数如下：

```
constructor: function(config) {
    var me = this;
    Ext.apply(me, config);
    me.addEvents(
        'beforeedit',
        'edit',
        'validateedit'
    );
    me.mixins.observable.constructor.call(me);
    me.relayEvents(me, ['afteredit'], 'after');
},
```

代码先复制了配置项到对象，然后添加了 beforeedit、edit 和 validatedit 这 3 个事件。接着初始化 Observable 对象，然后传播 afteredit 事件。这只是做了一些初始化工作。关键不在 AbstractComponent 对象的构造函数里，创建插件后，往往会调用插件的 init 方法，其代码如下：

```
init: function(grid) {
    var me = this;
    me.grid = grid;
    me.view = grid.view;
    me.initEvents();
    me.mon(grid, 'reconfigure', me.onReconfigure, me);
    me.onReconfigure();
    grid.relayEvents(me, ['beforeedit', 'edit', 'validateedit']);
    grid.isEditable = true;
    grid.editingPlugin = grid.view.editingPlugin = me;
},
```

可以看到，Editing 对象的 grid 和 view 属性都指向了对应的面板和视图。接着调用 initEvents 方法初始化事件，其代码如下：

```
initEvents: function() {
    var me = this;
    me.initEditTriggers();
    me.initCancelTriggers();
},
```

先调用 initEditTriggers 方法初始化进入编辑状态的事件，其代码如下：

```
initEditTriggers: function() {
    var me = this,
        view = me.view,
        clickEvent = me.clicksToEdit === 1 ? 'click' : 'dblclick';
    me.mon(view, 'cell' + clickEvent, me.startEditByClick, me);
    view.on('render', me.addHeaderEvents, me, {single: true});
},
```

如果配置项 clicksToEdit 的值为 1，表示单击进入编辑状态；否则为双击进入编辑状态。

接着将绑定单元格的单击事件或双击事件到 startEditByClick 方法，它会直接调用 startEdit 方法，代码如下：

```
startEdit: function(record, columnHeader) {
    var me = this,
        context = me.getEditingContext(record, columnHeader);
    if (me.beforeEdit(context) === false || me.fireEvent('beforeedit',me, context)
        === false || context.cancel) {
        return false;
    }
    me.context = context;
    me.editing = true;
},
```

代码开始就调用了 getEditingContext 方法，其代码如下：

```
getEditingContext: function(record, columnHeader) {
    var me = this,
        grid = me.grid,
        store = grid.store,
        view = grid.getView(),
        node = view.getNode(record),
        rowIdx, colIdx, tableView, node, value;

    if (Ext.isNumber(columnHeader)) {
        // look up column header if numeric column index was passed
        colIdx = columnHeader;
        columnHeader = grid.headerCt.getHeaderAtIndex(colIdx);
    } else {
        colIdx = columnHeader.getIndex();
    }

    view = columnHeader.ownerCt.lockableInjected ? (columnHeader.locked ? view.
        lockedView : view.normalView) : view;

    if (Ext.isNumber(record)) {
        // look up record if numeric row index was passed
        rowIdx = record;
        record = view.getRecord(node);
    } else {
        rowIdx = view.indexOf(node);
    }

    value = record.get(columnHeader.dataIndex);
    return {
        grid: grid,
        record: record,
        field: columnHeader.dataIndex,
        value: value,
        row: view.getNode(rowIdx),
        column: columnHeader,
        rowIdx: rowIdx,
        colIdx: colIdx
    };
},
```

从代码中可以看到，getEditingContext 方法会从 Store 中获取当前记录和行索引，从列标题中获取列索引，从当前记录中根据列的字段定义获取值。最后返回一个由面板、当前记录、字段名称、当前单元格的值、行号、列对象、行索引和列索引组成的对象。

也就是说，在执行完 startEdit 方法后，context 属性将指向返回的对象。

回到 initEditTriggers 方法，绑定事件后，会为视图的 render 事件绑定 addHeaderEvents 方法，其代码如下：

```
addHeaderEvents: function(){
    var me = this;
    me.mon(me.grid.headerCt, {
        scope: me,
        add: me.onColumnAdd,
        remove: me.onColumnRemove
    });

    me.keyNav = Ext.create('Ext.util.KeyNav', me.view.el, {
        enter: me.onEnterKey,
        esc: me.onEscKey,
        scope: me
    });
},
```

代码先为列标题容器绑定了 add 和 remove 事件，当添加了一列的时候，需要添加一个编辑组件；同样当删除了一列的时候，则要删除该列的编辑组件。

接着为视图的元素绑定键盘操作，包括回车和 ESC 两个键，回车键用于进入编辑状态（会调用 startEdit 方法），Esc 键用于取消编辑（会调用 cancelEdit 方法）。

回到 initEvents 方法，接着调用 initCancelTriggers 方法。要注意在 Editing 对象内，initCancelTriggers 方法是一个空函数。

初始化事件完成后回到 init 方法，为 Gird 的 reconfigure 事件绑定 onReconfigure 方法，并调用一次 onReconfigure 方法，该方法会直接调用 initFieldAccessors 方法，其代码如下：

```
initFieldAccessors: function(column) {
    var me = this;
    if (Ext.isArray(column)) {
        Ext.Array.forEach(column, me.initFieldAccessors, me);
        return;
    }
    Ext.applyIf(column, {
        getEditor: function(record, defaultField) {
            return me.getColumnField(this, defaultField);
        },
        setEditor: function(field) {
            me.setColumnField(this, field);
        }
    });
},
```

如果参数 column 是数组，会递归调用 initFieldAccessors 方法。如果不是数组，则为列对象添加 getEditor 和 setEditor 两个方法。在 getEditor 方法内会调用 getColumnField 方法，

其代码如下：

```
getColumnField: function(columnHeader, defaultField) {
    var field = columnHeader.field;
    if (!field && columnHeader.editor) {
        field = columnHeader.editor;
        delete columnHeader.editor;
    }
    if (!field && defaultField) {
        field = defaultField;
    }
    if (field) {
        if (Ext.isString(field)) {
            field = { xtype: field };
        }
        if (Ext.isObject(field) && !field.isFormField) {
            field = Ext.ComponentManager.create(field, this.defaultFieldXType);
            columnHeader.field = field;
        }
        Ext.apply(field, {
            name: columnHeader.dataIndex
        });

        return field;
    }
},
```

从代码可以看到，其作用就是创建一个编辑组件，然后将列对象的 field 属性指向该对象，且将字段名称赋值给编辑组件的 name 属性。

在创建编辑组件之前，变量 field 会从列对象的配置项 field 和 editor 中取值，也就是说，在列定义编辑组件时，可使用 field 配置项或 editor 配置项，不过标准的做法是使用 field 配置项，因为 editor 是为了兼容 3.x 版本而保留的。

方法 setColumnField 的作用与 getColumnField 方法一样。

也就是说 initFieldAccessors 方法为每列提供了访问编辑组件的方法。而且，每当 Grid 重新构建时，由于绑定 reconfigure 方法，会重新构建一次编辑组件。这是必须的，因为重新构建后，列可能已经不是原来的列了。

完成 onReconfigure 的调用，会将 beforeedit、edit 和 validateedit 这 3 个事件传播到 Grid，这样就可以在 Grid 内监听这些事件。

最后要做的是设置 isEditable 属性为 true，表示 Grid 是可编辑的，并设置 Grid 和视图的 editingPlugin 属性指向 Editing 对象实例。

从以上代码分析可以看到，Editing 对象初始化时，主要的工作是初始化进入和退出编辑状态的事件、在列对象加入创建编辑组件的方法并初始化编辑事件。

10.4.3 单元格编辑的运行流程：Ext.grid.plugin.CellEditing、Ext.grid.CellEditor 与 Ext.Editor

CellEditing 对象虽然派生于 Editing 对象，但有自己的构造函数，代码如下：

```
constructor: function() {
    this.callParent(arguments);
    this.editors = Ext.create('Ext.util.MixedCollection', false, function(editor){
        return editor.editorId;
    });
    this.editTask = new Ext.util.DelayedTask();
},
```

它首先调用 Editing 对象的构造函数，然后创建了一个 MixedCollection 对象实例用来保存编辑组件。接着创建了一个 DelayedTask 对象实例。

它没有 init 方法，也就是会使用 Editing 对象的 init 方法进行初始化工作。在 initEditTriggers 调用的过程中，它重写了 startEdit 方法，其代码如下：

```
startEdit: function(record, columnHeader) {
    var me = this,
        context = me.getEditingContext(record, columnHeader),
        value, ed;

    me.completeEdit();

    record = context.record;
    columnHeader = context.column;
    value = record.get(columnHeader.dataIndex);

    context.originalValue = context.value = value;
    if (me.beforeEdit(context)=== false || me.fireEvent('beforeedit', me, context)
        === false || context.cancel) {
        return false;
    }

    if (columnHeader && !columnHeader.getEditor(record)) {
        return false;
    }

    ed = me.getEditor(record, columnHeader);
    if (ed) {
        me.context = context;
        me.setActiveEditor(ed);
        me.setActiveRecord(record);
        me.setActiveColumn(columnHeader);

        me.editTask.delay(15, ed.startEdit, ed, [me.getCell(record, columnHeader),
            value]);
        me.editing = true;
    } else {
        me.grid.getView().getEl(columnHeader).focus((Ext.isWebKit || Ext.isIE) ?
            10 : false);
    }
},
```

代码先调用 getEditingContext 返回必需的信息。

接着调用一次 completeEdit 方法，其目的是防止用户在没有明确完成上一次的单元格编辑，而进入当前单元格进行编辑时，结束上一次单元格的编辑状态。

在处理好原始值并触发 beforeedit 事件后，调用 getEditor 获取编辑组件，其代码如下：

```
getEditor: function(record, column) {
    var me = this,
        editors = me.editors,
        editorId = column.getItemId(),
        editor = editors.getByKey(editorId);
    if (editor) {
        return editor;
    } else {
        editor = column.getEditor(record);
        if (!editor) {
            return false;
        }
        if (!(editor instanceof Ext.grid.CellEditor)) {
            editor =new Ext.grid.CellEditor({
                editorId: editorId,
                field: editor,
          editingPlugin: me,
          ownerCt: me.grid
            });
        }
        editor.on({
            scope: me,
            specialkey: me.onSpecialKey,
            complete: me.onEditComplete,
            canceledit: me.cancelEdit
        });
        editors.add(editor);
            return editor;
        }
    },
```

属性 editors 会指向一个对象，其成员会以编辑组件的 id 为关键字、编辑组件为值，作用就是为编辑组件做一个缓存，然后通过 editorId 找编辑组件。如果找到，可以直接返回；如果找不到，则调用列对象的 getEditor 方法，也就是在 initFieldAccessors 方法中注入列对象的方法，它会调用 getColumnField 方法创建编辑组件。如果创建成功，但不是 CellEditor 实例，则创建 CellEditor 对象实例，将编辑组件统一为 CellEditor 对象实例，这样的目的是让编辑组件按照单元格编辑组件的操作方式进行操作，简单来说，就是为编辑组件附加一些单元格编辑组件需要的属性、方法、事件及行为。

为编辑组件绑定完 specialkey、complete 和 canceledit 事件后，将其加入到 editors 中并返回。

回到 startEdit 方法，如果编辑组件存在，将 CellEditing 实例的 context 属性指向 context 对象，调用 setActiveEditor 方法将 activeEditor 属性指向编辑组件，调用 setActiveRecord 将 activeRecord 属性指向当前编辑的记录，调用 setActiveColumn 将属性 activeColumn 指向当前编辑的列，然后使用延时任务，在 15 微秒后，调用编辑组件（CellEditor 对象）的 startEdit 方法，其代码如下（在 Editor 对象中）：

```
startEdit : function(el, value) {
    var me = this,
```

```
        field = me.field;
    me.completeEdit();
    me.boundEl = Ext.get(el);
    value = Ext.isDefined(value) ? value : me.boundEl.dom.innerHTML;
    if (!me.rendered) {
        me.render(me.parentEl || document.body);
    }
    if (me.fireEvent('beforestartedit', me, me.boundEl, value) !== false) {
        me.startValue = value;
        me.show();
        field.suspendEvents();
        field.reset();
        field.setValue(value);
        me.realign(true);
        field.focus(false, 10);
        if (field.autoSize) {
            field.autoSize();
        }
        me.editing = true;
    }
},
```

粗体代码是 startEdit 方法的关键代码，它设置了开始值（startValue），然后显示编辑组件，使用 reset 方法清空编辑组件后再使用 setValue 方法将当前单元格的值设置为编辑值。最关键的一步来了，通过 realign 方法，将编辑组件调整到单元格的上面，这实际就是用了 div 层的概念，将编辑组件放到最顶层遮盖住后面的元素。最后是将焦点移到编辑组件内，并调整组件大小，设置 editing 为 true，表示当前处于编辑状态。

下面通过形象一点的过程来看看它是如何实现的。在浏览器中，打开 Ext JS 4 包中的示例“Cell Editing Grid Example”，单击第 3 行、第 2 列，进入编辑状态。然后使用元素选择按钮，在页面中选择编辑组件，在 HTML 代码中找到组件的 HTML 代码。然后如图 10-13 那样，选择其根元素。在选择代码上单击鼠标右键并在菜单中选择“在属性改变时中断”。然后单击第 3 行、第 5 列，这时候代码运行中断，面板会切换到脚本面板，单击“继续”按钮或按 F8 键让代码继续执行，代码执行后，又会如图 10-14 所示的位置中断代码的执行。

图 10-13　选择编辑组件的根元素

图 10-14　第二次中断时脚本面板的显示

这时候，编辑组件应该显示在第 3 行，第 5 列。而脚本面板中黄色背景部分也说明 div 的样式已经改变。这时候切换到脚本面板右边的堆栈面板会看到如图 10-15 所示的结果，从堆栈中可以看到脚本的调用过程。从 startEdit 中，调用 realign，一直调用到 alignTo 方法，坐标（248，313）已经计算出来了，然后调用 setPagePostition 方法后，坐标调整为（222，96），而这正是图 10-15 中 style 值中的 left 值和 top 值，也就是说，在调用 setLeftTop 方法之后，编辑组件已经移动到单元格上方了。单击堆栈中的方法名称可切换到方法的调用位置，在监控子面板中就可看到调用前各变量的值。

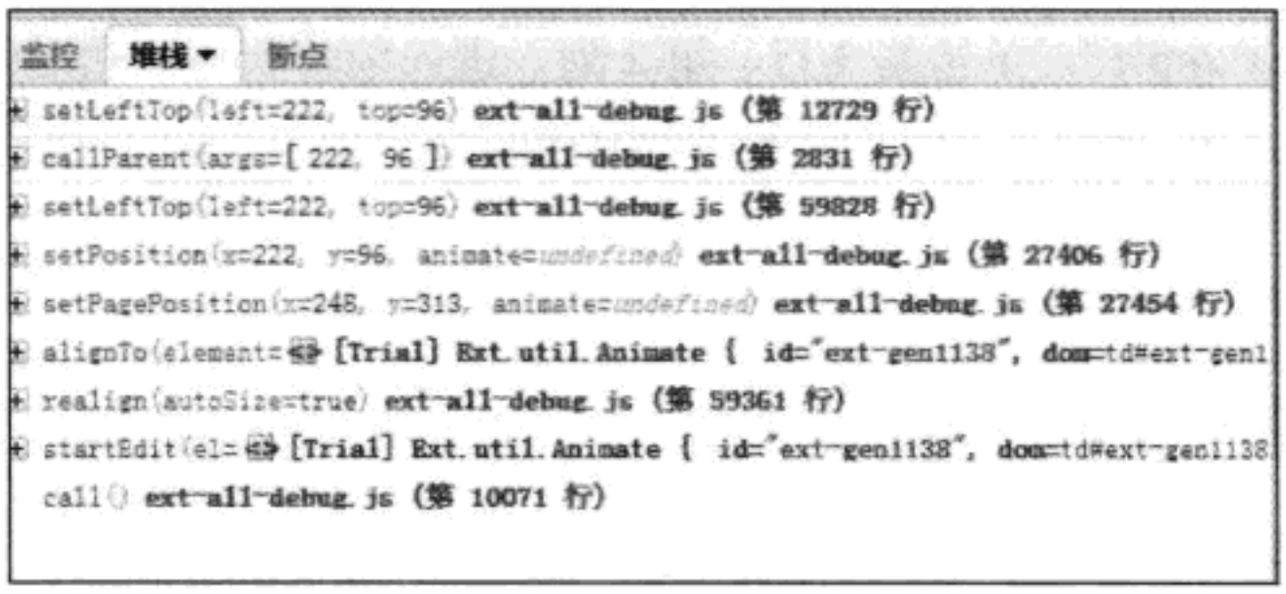


图 10-15　堆栈子面板的显示

以上代码是进入编辑状态的，下面看看编辑完成后的操作。编辑完成动作是从编辑组件开始的，会运行 Editor 对象内的 completeEdit 方法，代码如下：

```
completeEdit : function(remainVisible) {
    var me = this,
        field = me.field,
        value;

    if (!me.editing) {
        return;
```

```
        }
        if (field.assertValue) {
            field.assertValue();
        }
        value = me.getValue();
        if (!field.isValid()) {
            if (me.revertInvalid !== false) {
                me.cancelEdit(remainVisible);
            }
            return;
        }
        if (String(value) === String(me.startValue) && me.ignoreNoChange) {
            me.hideEdit(remainVisible);
            return;
        }
        if (me.fireEvent('beforecomplete', me, value, me.startValue) !== false) {
            value = me.getValue();
            if (me.updateEl && me.boundEl) {
                me.boundEl.update(value);
            }
            me.hideEdit(remainVisible);
            me.fireEvent('complete', me, value, me.startValue);
        }
    },
```

代码首页通过 getValue 方法返回值，然后使用 isValid 方法验证值是否有效，如果设置了配置项 revertInvalid 为 true，则取消编辑；否则直接返回，继续编辑。

如果新值与原值相同，则隐藏编辑组件，直接返回。

如果确认是新值与原值不同，则再次使用 getValue 获取值，这是为了避免触发 beforecomplete 事件后，值会被修改。如果要更新表格的显示，则调用 update 方法更新。最后隐藏编辑组件，触发 complete 事件。

在 getEditor 方法中，编辑组件的 complete 事件绑定了 onEditComplete 方法，这时候会执行该方法，其代码如下：

```
onEditComplete : function(ed, value, startValue) {
    var me = this,
        grid = me.grid,
        activeColumn = me.getActiveColumn(),
        record;

    if (activeColumn) {
    record = me.context.record;
        me.setActiveEditor(null);
        me.setActiveColumn(null);
        me.setActiveRecord(null);
        delete grid.getSelectionModel().wasEditing;
        if (!me.validateEdit()) {
            return;
        }
        if (!record.isEqual(value, startValue)) {
            record.set(activeColumn.dataIndex, value);
        } else {
```

```
                grid.getView().getEl(activeColumn).focus();
            }
            me.context.value = value;
            me.fireEvent('edit', me, me.context);

        }
    },
```

代码先使用 getActiveColumn 获取当前列对象。

如果列对象存在，将 activeEditor、activeColumn 和 activeRecord 这 3 个属性设置为 null，并将选择模型的 wasEditing 属性删除，表示编辑状态已结束。接着触发 validateedit 事件，如果返回 false，则直接返回。下面是检查值是否已改变，如果已改变，调用 set 方法设置值，**set 方法设置值后会自动更新 UI 的显示，所以不需要在 set 方法后加入刷新 UI 的代码**。如果值没有改变，则将焦点移动到单元格。

最后修改 context 的 value 属性，并触发 edit 事件。

取消编辑的过程与完成编辑差不多，只是把修改值的代码删除就可以了。

10.4.4 行编辑的运行流程：Ext.grid.plugin.RowEditing 与 Ext.grid.RowEditor

因为行编辑是在编辑时把所有列的编辑组件都显示出来的，因而列的变化会引起编辑组件的变化，所以在 initEditTriggers 方法中，把列标题容器的 add、remove 等与列操作有关的事件都绑定对应的方法来改变行编辑内的编辑组件。

RowEditor 对象是派生于 FormPanel 的，是一个面板，这是它与 CellEditor 对象的最大区别，因而 RowEditing 不需要使用 MixedCollection 对象去管理每列的编辑组件，只需要创建一个 RowEditor 对象实例就行了，它内部已包含所有的编辑组件。

接着还是从 startEdit 方法启动，代码如下：

```
startEdit: function(record, columnHeader) {
    var me = this,
        editor = me.getEditor();
    if (me.callParent(arguments) === false) {
        return false;
    }
    if (editor.beforeEdit() !== false) {
        editor.startEdit(me.context.record, me.context.column);
    }
},
```

还是先调用 getEditor 方法初始化编辑组件，其代码如下：

```
getEditor: function() {
    var me = this;
    if (!me.editor) {
        me.editor = me.initEditor();
    }
    return me.editor;
},
```

直接调用 initEditor 方法了，其代码如下：

```
initEditor: function() {
    var me = this,
        grid = me.grid,
        view = me.view,
        headerCt = grid.headerCt,
        cfg = {
            autoCancel: me.autoCancel,
            errorSummary: me.errorSummary,
            fields: headerCt.getGridColumns(),
            hidden: true,

            editingPlugin: me,
            renderTo: view.el
        };

    Ext.Array.forEach(['saveBtnText', 'cancelBtnText', 'errorsText', 'dirtyText'],
        function(item){
        if (Ext.isDefined(me[item])) {
            cfg[item] = me[item];
        }
    });

    return Ext.create('Ext.grid.RowEditor', cfg);
},
```

在代码里创建了一个 RowEditor 对象实例，配置项 fields 包含了所有列。下面看看 RowEditor 对象是如何初始化的，其 initComponent 方法如下：

```
initComponent: function() {
    var me = this,
        form;
    me.cls = Ext.baseCSSPrefix + 'grid-row-editor';
    me.layout = {
        type: 'hbox',
        align: 'middle'
    };
    me.columns = Ext.create('Ext.util.HashMap');
    me.columns.getKey = function(columnHeader) {
        var f;
        if (columnHeader.getEditor) {
            f = columnHeader.getEditor();
            if (f) {
                return f.id;
            }
        }
        return columnHeader.id;
};
me.mon(me.columns, {
    add: me.onFieldAdd,
    remove: me.onFieldRemove,
    replace: me.onFieldReplace,
    scope: me
```

```
        });
        me.callParent(arguments);
        if (me.fields) {
            me.setField(me.fields);
            delete me.fields;
        }
        form = me.getForm();
        form.trackResetOnLoad = true;
    },
```

控件都在一行，所以用 HBox 布局很合适。接着创建一个 HashMap 实例，这样就可通过编辑组件的 id 方便地找到对应的列对象。同时，使用 getKey 方法也可以通过列对象找到编辑组件的 id。

接着为 HashMap 实例绑定 3 个事件，当其发生添加、删除或替换操作的时候，创建、删除或替换对应的编辑组件。

调用父类的 initComponent 方法后，调用 setField 方法创建编辑组件，其代码如下：

```
setField: function(column) {
    var me = this,
        field;
    if (Ext.isArray(column)) {
        Ext.Array.forEach(column, me.setField, me);
        return;
    }
    field = column.getEditor(null, {
        xtype: 'displayfield',
        getModelData: function() {
            return null;
        }
    });
    field.margins = '0 0 0 2';
    me.mon(field, 'change', me.onFieldChange, me);
    if (me.isVisible() && me.context) {
        if (field.is('displayfield')) {
            me.renderColumnData(field, me.context.record, column);
        } else {
            field.suspendEvents();
            field.setValue(me.context.record.get(column.dataIndex));
            field.resumeEvents();
        }
    }

    me.columns.add(field.id, column);
    if (column.hidden) {
        me.onColumnHide(column);
    } else if (me.hasFields) {
        me.onColumnShow(column);
    }},
```

如果参数 column 是数组，递归调用 setField 方法。接着调用绑定到列对象的 getEditor 方法创建编辑组件，默认创建 DisplayField 对象。

编辑组件创建后，调整一下其位置，然后绑定 change 事件到 onFieldChange 方法做校验

以决定是否显示错误提示。然后将编辑组件加入 HashMap 实例中。

如果 RowEditor 对象可见，则调用 renderColumnData 渲染数据，其代码如下：

```
renderColumnData: function(field, record) {
    var me = this,
        grid = me.editingPlugin.grid,
        headerCt = grid.headerCt,
        view = grid.view,
        store = view.store,
        column = activeColumn || me.columns.get(field.id),
        value = record.get(column.dataIndex),
renderer = column.editRenderer || column.renderer,
        metaData,
        rowIdx,
        colIdx;
 if (renderer) {
        metaData = { tdCls: '', style: '' },
        rowIdx = store.indexOf(record),
        colIdx = headerCt.getHeaderIndex(column);

        value = column.renderer.call(
            column.scope || headerCt.ownerCt,
            value,
            metaData,
            record,
            rowIdx,
            colIdx,
            store,
            view
        );
    }
    field.setRawValue(value);
    field.resetOriginalValue();
},
```

代码先凑齐了配置项 renderer 定义的函数的参数，然后调用它处理好显示值，再调用 setRawValue 方法设置编辑组件的值，接着调用 resetOriginalValue 重新设置一下原值。

回到 startEdit 方法，当组件编辑器等都准备好了以后，就调用 RowEditor 对象的 startEdit 方法进入编辑状态，其代码如下：

```
startEdit: function(record, columnHeader) {
    var me = this,
        grid = me.editingPlugin.grid,
        view = grid.getView(),
        store = grid.store,
        context = me.context = Ext.apply(me.editingPlugin.context, {
            view: grid.getView(),
            store: store
        });
context.grid.getSelectionModel().select(record);
me.loadRecord(record);
if (!me.isVisible()) {
    me.show();
```

```
            me.focusContextCell();
        } else {
            me.reposition({
                callback: this.focusContextCell
            });
        }
    },
```

以上代码通过选择模型获取到数据后，调用 loadRecord 方法加载数据，其代码如下：

```
    loadRecord: function(record) {
        var me = this,
            form = me.getForm(),
           fields = form.getFields();;
        form.loadRecord(record);
fields.each(function(field) {
        field.resumeEvents();
});
if (me.errorSummary) {
        if (form.isValid()) {
            me.hideToolTip();
        } else {
            me.showToolTip();
        }
}

Ext.Array.forEach(me.query('>displayfield'), function(field) {
        me.renderColumnData(field, record);
}, me);     },
```

在调用 FormPanel 的 loadRecord 方法加载记录时，要调用 suspendEvents 方法先中断编辑组件的事件，以避免触发 change 事件。记录加载后，调用 resumeEvents 方法恢复事件。如果数据有错误，显示错误提示，没有则隐藏错误提示。

最后是把所有 DisplayField 找出来，调用 renderColumnData 方法处理完数据再赋值给 DisplayField 显示。

加载数据后，如果当前状态是不可见的，则调用 show 方法显示行编辑组件，再调用 focusContextCell 设置焦点。否则，调用 reposition 方法重新计算位置。

至此，就可以进入行编辑状态了。

行编辑是靠两个按钮控制完成编辑和取消编辑的，更新按钮绑定了 RowEditing 的 completeEdit 方法，其代码如下：

```
    completeEdit: function() {
        var me = this;
        if (me.editing && me.validateEdit()) {
            me.editing = false;
            me.fireEvent('edit', me.context);
        }
},
```

代码先调用 validateEdit 方法校验输入是否有效，其代码如下：

```
validateEdit: function() {
    var me          = this,
        editor      = me.editor,
        context     = me.context,
        record      = context.record,
        newValues   = {},
        originalValues = {},
        name;

    editor.items.each(function(item) {
        name = item.name;
        newValues[name]       = item.getValue();
        originalValues[name] = record.get(name);
    });

    Ext.apply(context, {
        newValues       : newValues,
        originalValues : originalValues
    });

    return me.callParent(arguments) && me.getEditor().completeEdit();
},
```

代码首先枚举每个编辑组件，将它们的原值和修改值分别写到 originalValues 和 newValues 对象里。然后再将这两个对象复制到 context 中。

最后调用父类的 validateEdit 方法去触发 validateEdit 事件。还要调用 RowEditor 对象的 completeEdit 方法，其代码如下：

```
completeEdit: function() {
    var me = this,
        form = me.getForm();
    if (!form.isValid()) {
        return;
    }
    form.updateRecord(me.context.record);
    me.hide();
    return true;
},
```

这里使用 isValid 方法验证输入是否有效，如果无效，则直接返回，不更新记录。如果有效，调用 updateRecord 方法更新记录。最后隐藏行编辑组件。

10.4.5　在 Grid 中使用单元格编辑模式

要使用单元格编辑模式，首先要做的是在 Grid 的 plugins 配置项内加入 CellEditor 对象的配置对象。然后在列的配置对象中加入配置项 editor 或 field，其值为编辑组件的配置对象。例如使用 10.2.5 的例子，首先在 Grid 定义内加入 plugins 配置项（注意类型要用 ptype）：

```
plugins:[{ptype:"cellediting"}],
```

接着为列的配置对象添加编辑组件：

```
{text:"地区",dataIndex:"name",width:80,
   field:{
        xtype: 'combobox',
        typeAhead: true,
        triggerAction: 'all',
        selectOnTab: true,
        store: ["广东","广西","北京","上海"],
        lazyRender: true
    }
},
{text:"销售业绩",columns:[
    {text:"第一季度",dataIndex:"Q1",width:80,editor:{xtype:"numberfield"}},
    {text:"第二季度",dataIndex:"Q2",width:80,editor:{xtype:"numberfield"}},
    {text:"第三季度",dataIndex:"Q3",width:80,editor:{xtype:"numberfield"}},
    {text:"第四季度",dataIndex:"Q4",width:80,editor:{xtype:"numberfield"}}
]}
```

地区所在列使用了一个下拉选择框，4 个季度则使用了数组输入框，打开模板页，然后在命令行运行后，随便修改一下数据，注意要双击才能进入编辑状态，可看到如图 10-16 所示的效果。修改过的单元格左上角红色小三角表示该列数据已修改过，要确认修改，需要使用模型的 commit 方法。

图 10-16　使用单元格编辑方式的 Grid

10.4.6　在 Grid 中使用行编辑模式

使用行编辑模式与单元格编辑模式一样简单，只要将上面示例中“cellediting”修改为“rowediting”就可以了。不过，做之前还要在本地化文件中将行编辑组件中两个按钮实现本地化，代码如下：

```
if(Ext.grid.RowEditor){
    Ext.apply(Ext.grid.RowEditor.prototype,{
        saveBtnText: '保存',
            cancelBtnText: '取消',
            errorsText:'错误',
            dirtyText:"你要确认或取消更改"
    })
}
```

修改完成后，在命令行中运行代码并做些修改将看到如图 10-17 所示的效果。

图 10-17　使用行编辑模式的 Grid

10.4.7　Grid 编辑插件的配置项、属性、方法和事件

编辑插件的主要配置项、属性、方法和事件主要集中在 Editing 对象上，而 CellEditing 对象和 RowEditing 对象因操作方式不同，会有自己的配置项、属性、方法和事件。

1. 配置项

Editing 对象提供的配置项：

- clicksToEdit：代表鼠标按钮的数字，用于显示编辑组件的鼠标动作，默认值为 2，双击单元格或行才显示编辑组件。设置为 1 则表示使用单击显示编辑组件。

CellEditing 对象没有提供自己的配置项。

RowEditing 对象提供的配置项：

- autoCancel：布尔值，默认值为 true，从当前编辑行移动到新的编辑行时，会自动取消之前的数据更改。如果设置为 false，则会要求用户确认是否取消当前修改。
- clicksToMoveEditor：与 clicksToEdit 配置项作用一样，设置进入编辑状态是使用单击还是双击操作。
- errorSummary：布尔值，默认值为 true，会通过提示信息显示校验的错误信息。设置为 false，则不显示错误信息。

2. 属性

Editing 对象、CellEditing 对象和 RowEditing 对象都没有属性。

3. 方法

Editing 对象提供的方法：

- cancelEdit：取消当前的编辑。
- completeEdit：完成当前编辑。
- startEdit：从指定的记录（行）和列，进入编辑状态。

CellEditing 对象提供的方法：

- startEditByPosition：从指定的位置进入编辑状态，位置参数是带有 row 和 column 两个属性的对象。

RowEditing 对象没有提供属于自己的方法。

4. 事件

Editing 对象提供的事件：

- beforeedit：在开始编辑前会触发该事件。
- edit：在编辑完成后会触发该事件。
- validateedit：在修改内容被写入记录前会触发该事件，返回 false 可取消改变。

以上事件都可以在 GridPanel 中进行定义，Editing 会将事件传播过去。CellEditing 和 RowEditing 对象没有其他额外的事件。

10.5 关于列表视图：ListView

在 Ext JS 3 版本中的列表视图（ListView）组件，已经被 Ext JS 4 中的 Grid 代替，在 GridPanel 的定义中，可以看到其别名包括“Ext.list.ListView”和“Ext.ListView”。主要原因是列表视图的功能可以完全由 Grid 来实现。

10.6 属性 Grid

10.6.1 概述

属性 Grid 是较特殊的 Grid，它的特殊之处在于，Grid 只有两列，一列是固定的属性名称，另一列是允许编辑的属性值。因为属性 Grid 是一个 Grid，所以它的组件构成与 Grid 是一样，只不过数据模型、Store、列标题容器等是固定了的。

因为属性 Grid 只有两列，因而其使用的数据模型 PropGridProperty 也只有 name 和 value 这两个字段。

为了方便数据处理，属性 Grid 直接使用了 JavaScript 对象作为数据源，而这对于 JsonReder 等对象都处理不了，所以从 Store 派生了用于属性 Grid 的 PropertyStore 对象，自定义了 Reader 对象，可直接读取 JavaScript 对象作为数据源。

显示的时候，Grid 必然只有两列数据，而且属性名称所在列是不可编辑的，只有属性值所在列是允许编辑的，所以专门定义 PropertyColumnModel 对象用来产生列标题。

视图没什么特殊要求，就是显示两列数据，所以没有专门的视图类。

因为属性值是可编辑的，所以需要使用 CellEditing 对象，但不可能为每个属性值都生成一个编辑组件，因而默认定义了日期、字符串、数字和布尔值 4 种编辑组件，然后根据属性值的类型使用相应的编辑组件。当然，4 种编辑组件是满足不了需求的，可自行定义编辑组件。

10.6.2 使用属性 Grid

要使用属性 Grid，只要定义配置项 source 就行了，其值是一个 JavaScript 对象，对象属性就是属性 Grid 的属性名称，其值就是要编辑的值。

打开模板页，在命令行中输入以下代码：

```
var pg=Ext.create("Ext.grid.property.Grid",{
    renderTo:Ext.getBody(),
    width:200,
    height:300,
    source:{
            "名称":"",
            GSM:false,
            CDMA:false,
            WCDMA:false,
            "系统":"",
            "发布时间":Ext.Date.format(new Date(),"Y-m-d"),
            "价格":0
    }
});
```

运行后，将看到如图 10-18 所示的效果。

名称	值
CDMA	false
GSM	false
WCDMA	false
发布时间	07/13/2011
价格	0
名称	
系统	

图 10-18　属性 Grid 的示例效果

修改 CDMA、GSM 的值为 true，修改价格的值为 3000，然后在命令行中输入：

```
console.dir(pg.getSource());
```

在控制台将看到如下输出：

```
CDMA        true
GSM         true
WCDMA       false
价格        3000
发布时间    "2011-07-13"
名称        ""
系统        ""
```

使用 getSource 方法可返回对象，且值已被修改。

使用 setProperty 方法可修改属性值，并可以立即在 Grid 中反映出来，例如输入以下命令：

```
pg.setProperty("价格",3200)
```

这时在 Grid 会看到价格的值已被修改为 3200。

如果想删除一个属性，可使用 removeProperty 方法，在命令行中输入：

```
pg.removeProperty("价格")
```

会看到 Grid 中的“价格”这行已经被删除了。

10.6.3　自定义编辑组件

默认的 4 种编辑方式肯定满足不了需求，例如，上一节示例中的“系统”属性，一般是使用下拉列表框选择的。这时，可使用自定义编辑组件来实现该功能。

实现“系统”属性使用下拉选择框首先使用 customEditors 配置项，其值是一个对象，对象的属性名称就是 source 对象中的属性名称，例如，现在要为“系统”定义一个自定义编辑组件，则“系统”就是 customEditors 配置项中的一个属性，其值是编辑组件的配置对象，要

使“系统”属性使用下拉选择框，可这样定义：

```
customEditors:{
    "系统":{
        xtype: 'combobox',
         typeAhead: true,
         triggerAction: 'all',
         selectOnTab: true,
         store: ["WM","iOS","安卓","塞班"],
         lazyRender: true
    }
},
```

将定义加到上一节的 PropertyGrid 定义中，运行后，可看到如图 10-19 所示的效果。

图 10-19 自定义编辑组件的示例效果

10.6.4 PropertyGrid 的配置项、属性、方法和事件

1. 配置项

- customEditors：用于自定义编辑组件。
- customRenderers：用于自定义显示格式，使用方法可参考自定义编辑组件。格式函数定义可参考列的 renderer 配置项。
- nameColumnWidth：属性列的宽度，默认值为 115。
- nameField：重定义属性名称的字段，默认值是 name。
- propertyNames：可用来定义属性的显示名称，例如在 source 中，属性名称为 system，但如果希望它显示“系统”，就可在该配置项中进行定义，代码如下：

```
propertyNames:{system:"系统"}
```

- source：数据对象，设置属性 Grid 的数据。
- valueField：重定义属性值的字段，默认值是 value。

2. 属性

PropertyGrid 没有属性。

3. 方法

- getSource：返回属性 Grid 的数据。
- removeProperty：删除指定的属性。
- setProperty：设置指定属性的值。
- setSource：设置数据。

4. 事件

- beforepropertychange：属性改变前会触发该事件，返回 false 可取消改变。
- propertychange：属性改变后会触发该事件。

10.7　综合实例

10.7.1　使用不同选择模型的 Grid 以及设置默认选择行

（1）功能描述

在 Grid 中，可使用的选择模型有 RowModel、CellModel 和 CheckboxModel 三种，而如果 CheckboxModel 的配置项 checkOnly 使用默认值 false，其功能和 RowModel 是一样的，单击行就可进行选择，只是视觉上通过复选框可知道哪个行被选择，如果其值为 true，则只能通过复选框对行进行选择。本示例将使用一个数据源，同时显示 4 个 Grid，分别使用 3 种选择模型，其中 CheckboxModel 会根据配置项 checkOnly 的配置分成两个 Grid。

还要实现一个功能，Grid 在显示后要选择第一行，并将焦点移动到 Grid 内，可通过键盘在 Grid 行中导航。

（2）实现代码

先使用模板页新建一个名称为 10-3.html 的页面文件，然后将 10.7.5 节的 Store 定义加入模板页中。

为了便于演示，使用表格布局分两行两列显示 4 个 Grid，先定义好示例面板和布局：

```
Ext.create("Ext.panel.Panel",{
    layout:{type:"table",columns:2},
    width:800,
    height:500,
    renderTo:Ext.getBody(),
    items:[
            //Grid 定义
    ]
})
```

下面定义使用 RowModel 的 Grid：

```
{xtype:"gridpanel",store:store,width:400,height:200,
    title:" 使用 RowModel 的 Grid",
    selModel:{mode:"MULTI"},
    columns:[
    {text:" 地区 ",dataIndex:"name",width:40,},
    {text:" 销售业绩 ",columns:[
            {text:" 第一季度 ",dataIndex:"Q1",width:80},
            {text:" 第二季度 ",dataIndex:"Q2",width:80},
            {text:" 第三季度 ",dataIndex:"Q3",width:80},
            {text:" 第四季度 ",dataIndex:"Q4",width:80}
        ]}
],},
```

因为 Grid 默认选择模型为 RowModel，这里不需要定义 selType，选择模型的配置项需要使用配置项 selModel 进行配置，也可以把把 selType 配置项添加到 selModel 内一起定义。这里设置其选择模式为多选（MULTI）。

下面定义使用 CellModel 的 Grid，可复制一份 RowModel 的代码，然后修改 title 配置

项的值为“使用 CellModel 的 Grid”，再添加配置项 selType 并将其值设置为“cellmodel”，这样就表示要使用单元格选择模型。因为单元格选择模型不允许多选，所以可删除配置项 selModel。

接着是定义使用 CheckboxModel 的 Grid，照样复制一份 RowModel 的代码，修改 title 配置项为“使用 CheckboxModel 的 Grid(checkOnly:false)”；添加配置项 selType，其值为 checkboxmodel；在 selModel 的配置对象内，添加配置项 checkOnly，其值为 false。

最后定义 checkOnly 为 true 的 Grid，复制一份使用 CheckboxModel 的 Grid 的定义代码，修改标题为“使用 CheckboxModel 的 Grid(checkOnly:true)”，然后修改 checkOnly 的值为 true。

现在要考虑怎么默认选择第一行，这个不难，使用视图的 select 方法就可以了，最大的问题是在什么时间选择。例如，在创建 Grid 后执行或在 Grid 渲染后执行，可能视图还没渲染，这时候根本就没元素可以选择，因而最佳的执行时机是在视图的数据渲染后，也就是在视图的 refresh 事件完成后。目标明确，可动工了，先在第一个 Grid 的定义内添加以下代码：

```
viewConfig:{
    listeners:{
            refresh:function(){
                    this.select(0);
            }
    }
},
```

如果希望在显示时将焦点移到 Grid 内，可立即使用键盘进行导航，可加入以下语句：

```
this.focus(false);
```

使用 CheckboxModel 的两个 Grid 可以使用与 RowModel 一样的方式定义。使用 CellModel 的 Grid 要麻烦一点，因为要通过行和列去定位，所以需要使用 selectByPosition 方法去选择单元格。不过，如果在 refresh 事件内直接调用，在后续处理 onCellFocus 方法中，会把选择取消，由于不知道是 bug 还是必须这样，因此需要做一个延时调用，具体代码定义如下：

```
afterrender:function(){
    Ext.defer(function(){
            this.getSelectionModel().selectByPosition({row:0,column:0});
    },30,this);
}
```

代码使用 defer 方法做了一个延时调用。要注意，一定要设置方法的第 3 个参数，调用函数的作用域为 this，不然在函数内就不能使用 this 访问视图对象了。

至此，示例就完成了。

（3）示例效果

在浏览器中打开示例，将看到如图 10-20 所示的结果。如果打开页面时，没使用鼠标切换焦点，那么在页面中，使用键盘中的箭头键可以在“使用 CellModel 的 Grid”的行列中进行移动，如果把其 afterrender 方法去掉，则可在第 1 个 Grid 中进行导航。

图 10-20　示例使用不同选择模型的 Grid 的运行效果

尝试一下单击两个 CheckboxModel 的 Grid，看看它们之间的不同。配置项 checkOnly 为 false，只要你单击行就可进行选择；而为 true 的，只有单击复选框才能选择行，单击行不起作用。

10.7.2　Grid 的本地排序和过滤

Ext JS 的数据与 UI 是分离的，因而 Grid 的排序和过滤也就是 Store 的排序和过滤。例如 9.6.6 节的修改示例的面板及视图 GridPanel 和 GridView，排序和过滤的代码不需要修改就可以使用了，只要修改显示方式。

将文件 9-3.html 复制一份并修改其文件名为 10-4.html。然后根据 10.2.4 节把 CheckColumn 对象的相关文件和代码复制过来，再添加 QuickTips 的初始化语句。

面板部分主要有以下修改：

- 将原来创建面板的类名“Ext.panel.Panel”修改为 Grid 的类名“Ext.grid.Panel”。
- 将 title 修改为“示例 10-4 Grid 的本地排序和过滤”。
- 删除配置项 autoScroll 和 items。
- 添加配置项 store，并设置其值为 phoneStore。
- 添加列定义，代码如下：

```
columns:[
    {text:"编号",dataIndex:"id",width:40},
    {text:"产品名称",dataIndex:"name",width:200,renderer:function(v,meta,rec){
            meta.tdAttr="data-qtip=\"<img  src='../images/phones/"+rec.data.
              image+"'>\" title='"+v+"'";
          return v;
}},
{text:"品牌",dataIndex:"Brands",width:80},
{text:"系统",dataIndex:"system",width:140},
{xtype:"numbercolumn",text:"价　格",dataIndex:"price",width:80,format:"
    ￥0,0.00",align:"right"},
{xtype:"checkcolumn",text:"GSM",dataIndex:"GSM",width:60},
{xtype:"checkcolumn",text:"WCDMA",dataIndex:"WCDMA",width:60},
```

```
        {xtype:"checkcolumn",text:"CDMA",dataIndex:"CDMA",width:60},
    ],
```

至此，修改就完成了，在浏览器中打开页面将看到如图 10-21 所示的结果。单击一下工具栏的按钮，会看到排序和过滤功能不仅一点没因为显示是 Grid 而受影响，而且还可使用 Grid 的标题栏进行排序。这就是数据与 UI 分离的好处，相同的数据，可使用不同的表达方式，操作不受表达方式的影响。

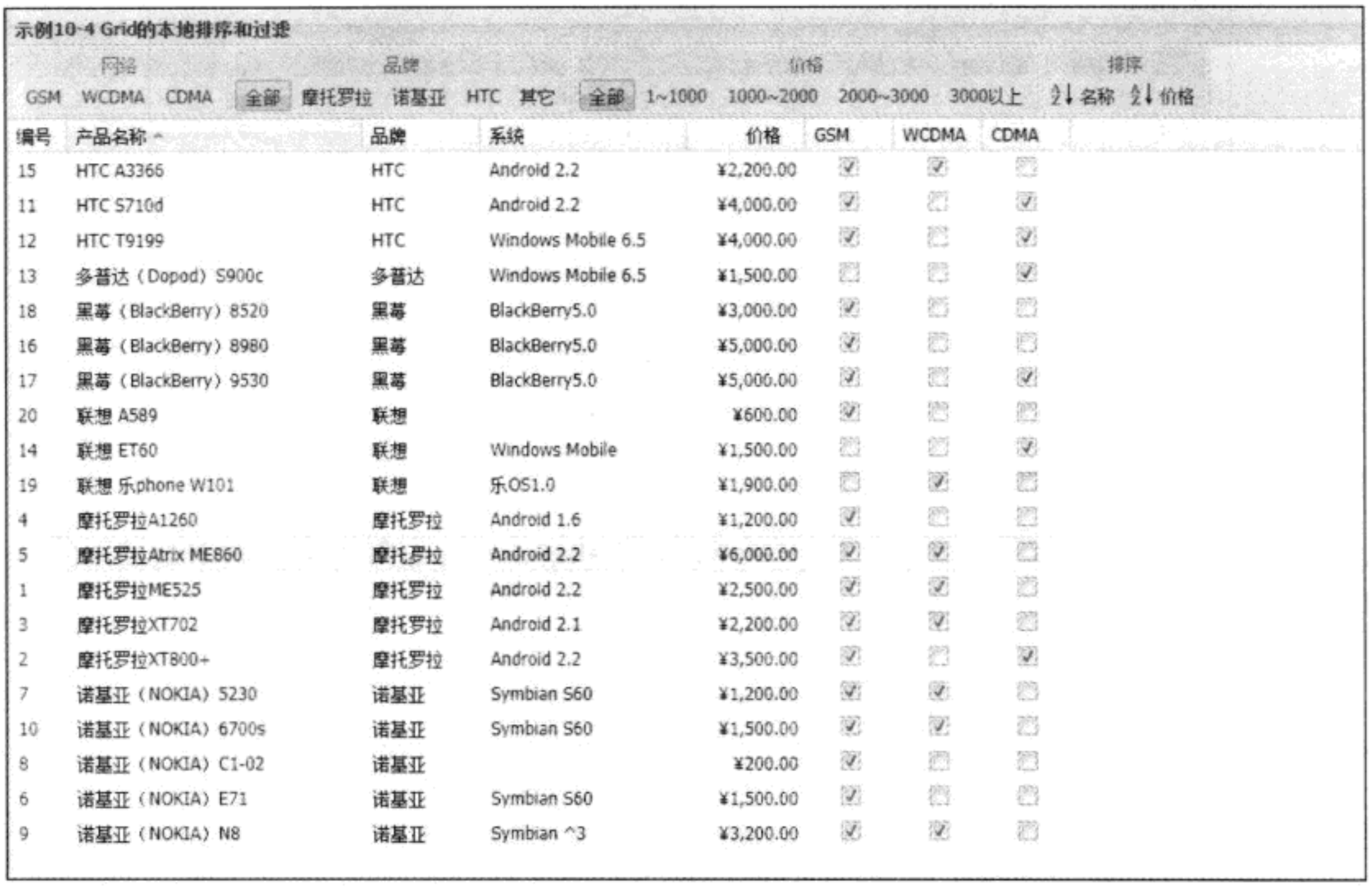


图 10-21　Grid 的本地排序和顾虑

10.7.3　使用分页工具条（PagingToolbar）实现远程分页、排序和过滤

（1）功能描述

使用分页工具条对微软示例数据库“Northwnd”的产品表（Products）进行分页显示，并实现远程排序和过滤功能。

（2）实现代码

在 7.6.1 节的示例中，已经实现基本功能，只是没有使用分页工具条，所以可在此基础上作修改。复制文件 7-5.html，然后将复制后的文件名修改为 10-5.html。首先为 Store 加上 remoteFilter 和 remoteSort 两个配置项，其值都设置为 true，表示使用远程过滤和排序。然后，为 Grid 顶部添加一个分页工具条，并在工具条上添加一个文本框和按钮，用于搜索产品名称，其代码如下：

```
tbar:{xtype:"pagingtoolbar",store:store,items:[
    "|",
    "查找: ",
    {xtype:"textfield",width:160,id:"searchText"},
    {text:"Go",handler:function(){
            var store=this.up("pagingtoolbar").store;
            var search=Ext.getCmp("searchText").getValue();
            if(search && search.length>0){
                    store.currentPage=1;
                    store.filters.clear();
                    store.filter("ProductName",search);
                    }
    }},
    {text:"显示全部",handler:function(){
            store.currentPage=1;
            this.up("pagingtoolbar").store.clearFilter();
    }}
]},
```

定义分页工具栏，关键点是为配置项 store 定义 Store 对象。如果要在工具条上添加组件，可配置 items 配置项，代码中，依次添加了分隔线、文本“查找：”、一个文本输入框、“Go”按钮和“显示全部”按钮。单击“Go”按钮会执行查询。设置属性 currentPage 为 1 的目的是保持每次查询的时候都是从第 1 页开始。在这里不能使用 Store 的 clearFilter 方法来清除之前的查询，不然清除后会提交一个没有查询条件的请求，因而只能通过 MixedCollection 对象的 clear 方法来进行清除，然后添加一个过滤条件，让 Store 自动发送请求。如果不想使用 filter 方法，也可以使用 load 方法，使用自定义的提交参数方法，例如：

```
store.load({params:{key:seach}})
```

代码中使用了 key 作为提交参数。不过该方法有个弊端，每当发送请求的时候，都必须再传递一次参数，例如取下一页的时候，也必须把参数加上，不然不能提交这个额外的参数，也就是说，不提供参数的话，获取不了下一页的查询结果。解决这个问题的办法是修改 Proxy 对象 extraParams 配置项，代码如下：

```
store.proxy.extraParams.key=search;
store.load();
```

这样就可以保证查询结果也能分页。该方法比较适合固定查询字段的情形，因为提交的参数只有一个，当然，也可以添加一个参数来指定字段，不过这与使用 filter 方法没什么区别。具体使用哪种方法，可根据需求灵活掌握。本示例使用 filter 方法，更标准化一些。

“显示全部”要提交一次请求，因而使用 clearFilter 方法。如果要使用提交 key 参数的方法，只要将 key 的值改为空字符串，再加载一次就行了。

为了在查询后能高亮显示查询结果，需要为 ProductName 添加配置项 renderer，用来高亮显示查询字符，代码如下：

```
renderer:function(v,meta,rec,row,col,store){
    var filter = store.filters.items;
    if(filter.length>0){
```

```
            var value=filter[0].value;
            return v.replace(new RegExp(value,'gi'),function(m){
                    return "<font color='red'>"+m+"</font>";
            })
    }else{
            return v;
    }
}
```

代码先从参数 store 中的 Filter 对象取得由过滤对象组成的数组，如果不存在，直接返回值。否则，获取查询值，然后使用替换函数逐个替换搜索到的值，在 replace 中使用一个枚举函数的目的是保持字符原来的大小写，当然，中文字符没这问题。

前台的页面现在做好了，要修改后台代码了。可以将 7.6.1 节示例的后台代码文件复制一份，然后做修改。

分页和排序其实已经做好了，现在要考虑的是怎么进行查询。首先要做的是添加处理提交过来的查询信息：

C#

```
string filter = "true";
if (context.Request.Params["filter"] != null)
{
    filter = "";
    JArray jaf = JArray.Parse(context.Request.Params["filter"]);
    foreach (JObject jo in jaf)
    {
        filter += "it." + (string)jo["property"] + " LIKE '%" + (string)jo["value"]
            + "%',";
    }
    filter = filter.Substring(0, filter.Length - 1);
}
```

Java：

```
String filter="";
if(request.getParameter("filter")!=null){
   JsonParser jparser = new JsonParser();
    JsonArray ja = jparser.parse(request.getParameter("filter")).getAsJsonArray();
    for (JsonElement je : ja) {
          JsonObject jo = je.getAsJsonObject();
          filter+=jo.get("property").getAsString()+ " like '%"
                + jo.get("value").getAsString()+"%',";
    }
    filter=filter.substring(0,filter.length()-1);
}
```

因为只有一个查询字段，所以直接组合成 like 查询就行了。最后是修改查询语句：代码如下：

C#

```
var q = ne.Products.Where(filter).OrderBy(sort).Skip(start).Take(limit).Select(m
    => m).ToList();
```

Java

```
sb.append("select top 20 * from products where ProductID ");
sb.append(" not in (select top ");
sb.append(start);
sb.append(" ProductID from products  ");
if(filter.length()>0){
   sb.append(" where ");
   sb.append(filter);
}
sb.append(" order by ");
sb.append(sort);
sb.append(")");
if(filter.length()>0){
   sb.append(" and ");
   sb.append(filter);
}
sb.append("  order by ");
sb.append(sort);
```

至此，全部修改就完成了。

（3）页面效果

在浏览器打开示例页面将看到如图 10-22 所示的效果。在搜索中输入“a”，将看到“ProductName”列中的单元格所有“a”字符都会以红色显示。

产品列表

第 1 页,共 4 页　查找:　Go　显示全部

id	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder
1	Chai	1	1	10 boxes x 20 ba	18	39	0
2	Chang	1	1	24 - 12 oz bottles	19	17	40
3	Aniseed Syrup	1	2	12 - 550 ml bottle	10	13	70
4	Chef Anton's Caju	2	2	48 - 6 oz jars	22	53	0
5	Chef Anton's Gum	2	2	36 boxes	21.35	0	0
6	Grandma's Boysei	3	2	12 - 8 oz jars	25	120	0
7	Uncle Bob's Orgar	3	7	12 - 1 lb pkgs.	30	15	0
8	Northwoods Cranl	3	2	12 - 12 oz jars	40	6	0
9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	97	29	0
10	Ikura	4	8	12 - 200 ml jars	31	31	0
11	Queso Cabrales	5	4	1 kg pkg.	21	22	30
12	Queso Manchego	5	4	10 - 500 g pkgs.	38	86	0
13	Konbu	6	8	2 kg box	6	24	0
14	Tofu	6	7	40 - 100 g pkgs.	23.25	35	0
15	Genen Shouyu	6	2	24 - 250 ml bottle	15.5	39	0
16	Pavlova	7	3	32 - 500 g boxes	17.45	29	0
17	Alice Mutton	7	6	20 - 1 kg tins	39	0	0
18	Carnarvon Tigers	7	8	16 kg pkg.	62.5	42	0
19	Teatime Chocolat	8	3	10 boxes x 12 pie	9.2	25	0
20	Sir Rodney's Marn	8	3	30 gift boxes	81	40	0

图 10-22　示例初始时的效果

10.7.4 使用分页滚动条（PagingScroller）实现远程分页、排序和过滤

（1）功能描述

PagingScroller 对象在 10.1.6 节已经介绍过，它使用缓存的方式，模拟出不需要分页的效果。实际上，分页还是有的，不过是在用户几乎感觉不到的情况下加载了分页，否则，真的有 100 万条记录，一次性全加载完成，不大可能。

要使用 PagingScroller 对象，有以下 5 个主要配置：

- 定义 Store 的配置项 buffered 为 true，启用缓存功能。
- 定义 Grid 的配置项 scroll 为 true、both 或 vertical，启用滚动条。可通过配置项 verticalScrollerType 为 PagingScroller 对象定义配置项。
- 定义 Grid 的配置项 invalidateScrollerOnRefresh 为 false，避免在视图刷新时重置滚动条。
- 定义 Grid 的配置项 disableSelection 为 true，禁止选择。这是分页滚动条的一大遗憾。主要原因是视图在滚动过程中会不断地变化，而选择模型是根据视图的行进行选择的，视图刷新后，已选择的行就不是原来的行了，这会造成错误，所以使用分页滚动条就不能进行选择。在使用中这是要权衡的。
- 加载数据要使用 Store 的 guaranteeRange 方法。

本示例将演示使用分页滚动条显示 Northwind 库的 Orders 表，该表的数据比较多，适合做演示。若每页数据在 50 条以下，使用分页滚动条的效果不佳。显示的字段包括 OrderID、CustomerID、OrderDate 和 CustomerName（Customers 表中的 CompanyName 字段）。

Grid 还要根据 OrderDate 过滤记录。

（2）实现代码

使用模板页创建一个名称为 10-6.html 的页面文件。首先，定义一个数据模型：

```
Ext.define('Orders', {
    extend: 'Ext.data.Model',
    fields: [
           {name:'OrderID',type:"int"},
           'CustomerID','CustomerName',
           {name:'OrderDate',type:"date",format:"Y-m-d"}
    ],
    idProperty:"OrderID"
});
```

为了避免排序时，后台代码转换字段名称，这里定义了 idProperty 为 OrderID，其余的名称都为实际的字段名称，CustomerName 的排序与 CustomerID 的排序效果是一样的，因而在定义列的时候，要禁止对 CustomerName 进行排序。

接着要定义一个 Store：

```
var store=Ext.create("Ext.data.Store",{
   model:'Orders',
   pageSize:100,
   remoteFilter:true,
   remoteSort:true,
   buffered:true,
```

```
proxy: {
    type: 'ajax',
    api:{
       read:'Orders.ashx'
       //read:"/Chapter10/Orders" //java
       },
    reader:{
       type: 'json',
       root:"data"
    }
},
    storeId:"OrdersStore"
})
```

Store 每页将会有 100 条数据，开启了远程排序、过滤和缓冲功能。

接着定义 Grid：

```
Ext.create("Ext.grid.Panel",{
    id:"myGrid",
    renderTo:Ext.getBody(),
    title:"订单列表",
    height:500,
    width:800,
    renderTo:Ext.getBody(),
    store:"OrdersStore",
    scroll:"both",
    invalidateScrollerOnRefresh: false,
    disableSelection:true,
    tbar:[
            "范围：",
    {xtype:"datefield",width:120,id:"StartDate",format:"Y-m-d",value:new Date()},
            "至",
    {xtype:"datefield",width:120,id:"EndDate",format:"Y-m-d",value:new Date()},
            {text:"Go",handler:function(){
            }},
            {text:"显示全部",handler:function(){
            }}
    ],
    columns:[
            {xtype:"rownumberer",sortable:false,width:60},
            {text:'编号',dataIndex:'OrderID'},
            {text:'客户编号',dataIndex:'CustomerID'},
            {text:'客户名称',dataIndex:'CustomerName',sortable:false,flex:1},
{xtype:"datecolumn",text:'订 购 日 期',dataIndex:'OrderDate',format:"Y-m-
    d",width:100}
    ]
    });
```

这与一般 Grid 主要区别就在 3 个分页工具条的配置项。为了能根据 OrderDate 过滤数据，在顶部工具栏添加了两个日期选择框，还定义了“Go”和“显示全部”两个按钮，分别用于指向过滤和取消过滤及显示全部数据，具体的操作代码还没写。使用分页工具条时建议显示行号，不然用户滚动过后都不知道数据滚动到什么位置了。行号列和客户名称所在列都定义了禁止排序。

现在可以定义“Go”按钮的单击事件了。首先要考虑的是怎么提交这两个日期值，使用Filter对象肯定不行，因为它只能提交字段和一个值，而且不能表示大于、小于或等于这些关系。重构Filter对象工程量太大，所以还是利用extraParams配置项比较方便。

其实，提交是小问题，最大的问题在于过滤后记录总数和记录都会发生改变，这就需要调整滚动条，因为滚动条是根据记录总数和行高计算出来的，所以如果不修改，滚动条和记录就对应不上了。过滤后返回的肯定是过滤后第一页的数据，不然就得自己计算滚动条位置，然后自己去取滚动条位置的记录，而这需要知道一个前提，就是过滤后的记录总数是多少，这样就得和服务器交互两次才能计算出获取数据的范围，把问题复杂化了。所以，返回第一页是最佳的选择。返回第一页的最大问题是Grid并不知道返回的数据是第几页的，它只会根据当前的滚动条状态做出判断。如果当前显示数据在第一页，调用guaranteeRange方法还是会从当前缓存取数据，因而必须使用load方法强制加载数据；如果不在第一页，调用guaranteeRange方法返回第一页的数据，就需要从服务器取数据，会自动更新滚动条，不需要再做其他处理。

在Store开启缓冲后，Store会从缓存中获取对应位置的数据，而数据过滤后，缓冲的数据不一定符合过滤条件，因而，还要清理缓冲中的数据。

综合以上各点，“Go”按钮的句柄该如何编写就很清晰了，其代码如下：

```
var grid=this.up("gridpanel"),
    stroe=grid.store,
    start=Ext.getCmp("StartDate").getValue(),
    end=Ext.getCmp("EndDate").getValue(),
params=store.getProxy().extraParams;
if(start>end){
        Ext.Msg.alert("提示信息","开始日期不能大于结束日期！");
        return;
}else if(start.getTime()==end.getTime()){
        params.op="=";
}else{
        params.op="between";
        params.EndDate=Ext.Date.format(end,"Y-m-d");
}
params.StartDate=Ext.Date.format(start,"Y-m-d");
store.prefetchData.clear();
store.MyFilter=true;
if(store.currentPage==1){
        store.load();
}else{
        store.guaranteeRange(0,99);
}
```

如果开始日期大于结束日期，就将此提示用户；如果两个值是相等的，意味着只查询一天的数据，因而要提交操作符op为等号。否则，就使用between进行查询，而且要记录结束日期。

代码中为extraParams配置项添加了op、StartDate和EndDate三个提交参数，分别表示过滤的操作符、开始日期和结束日期，在服务器可通过这3个参数获取过滤数据。

接着使用 clear 方法清理 Store 中的缓冲数据（prefetchData）；设置标志 MyFilter 为 true，表示要重新计算滚动条。

如果 Store 的当前页为第一页，调用 load 方法强制加载数据，否则调用 guaranteeRange 取数据。

现在要在 Store 中监听 datachanged 事件，当数据加载后且 MyFilter 为 true 时，要重新计算滚动条，代码如下：

```
listeners:{
    datachanged:function(store){
            if(store.MyFilter){
                    var grid=Ext.getCmp("myGrid");
                    store.MyFilter=false;
                    grid.invalidateScroller();
            }
    }
}
```

“Go”按钮完成后，显示全部按钮的操作就容易办了，把提交参数的值清空，重新加载数据就行了，其代码如下：

```
var grid=this.up("gridpanel");
    stroe=grid.store,
    params=store.getProxy().extraParams;
    params.op="";
    params.EndDate="";
    params.StartDate="";
    grid.store.prefetchData.clear();
    grid.store.MyFilter=true;
    if(store.currentPage==1){
            store.load();
    }else{
            store.guaranteeRange(0,99);
    }
```

最后要加入的代码是使用 guaranteeRange 方法为 Store 加载数据：

```
store.guaranteeRange(0,99);
```

不能通过设置 Store 的配置项 autoLoad 为 true 来自动加载数据，因为它不能让滚动条重新进行计算，这样会造成滚动条与视图对应不上。在“Go”按钮中可使用 load 方法是因而它会强制重新计算滚动条。当然，在自动加载数据后触发的 datachanged 事件中，强制重新计算滚动条也是可以。

服务器端代码与 10.7.3 节的代码区别不大，因而在此就不列出了，具体请看本书的源代码包中的相关代码。

（3）页面效果

在浏览器中打开页面，打开 Firebug 的控制台面板，然后往下滚动滚动条，当滚动到行号为 50 条左右的时候，会看到控制台多了一条提交请求，说明 Grid 预取了下一页的数据。然后修改开始日期为“1996-11-17”，结束日期为“1997-07-17”，单击“Go”按钮，再单击显示全

部按钮，将看到如图 10-23 所示的效果。

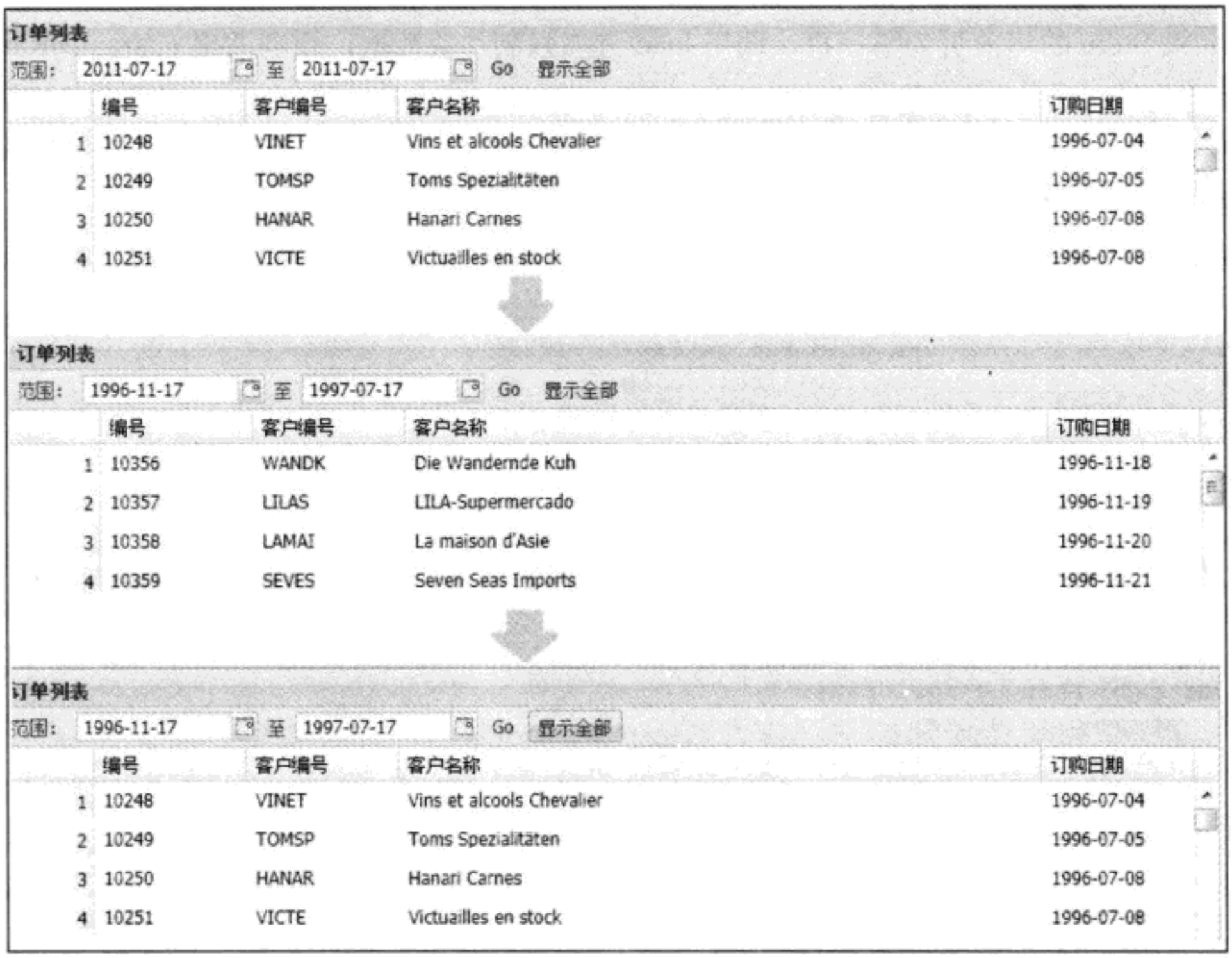


图 10-23　使用分页滚动条示例的页面效果

（4）bug 说明

4.1 Beta 1 版的 guaranteeRange 方法有 bug，不能正确计算请求页，bug 代码如下：

```
start = Math.min(Math.max(start - me.numFromEdge - ((me.leadingBufferZone -
    me.trailingBufferZone) / 2), 0), me.totalCount - me.pageSize);
end = start + (me.pageSize - 1);
startPage = me.getPageFromRecordIndex(start);
endPage = me.getPageFromRecordIndex(end);
```

本来正确的 start 和 end 值重新计算后，变得不正确，从而导致 startPage 和 endPage 的错误，进而不能正确地去服务器加载数据。解决办法是在本地化文件中重写该方法，把重新计算 start 和 end 这两句屏蔽掉就好了。

10.7.5　使用 CellEditing 实现数据的增删改

（1）功能描述

本示例将使用 CellEditing 对象在 Grid 中实现编辑 Northwind 库中 Territories 表的数据，这需要使用到相关表 Region。

（2）实现代码

无论是使用 CellEditing 对象还是 RowEditing 对象对 Grid 中的数据进行编辑，首先要考虑的都是使用什么样的提交方式。主要配置项包括 Store 对象的 autoSync，Proxy 对象的 batchActions，以及 Writer 对象的 writeAllFields、encode、root 和 allowSingle。无论前后台代码是否由你一个人开发，在开始编码前一定要有一个共同的约定来提交数据，这样代码就比较统一，也便于以后修改。

配置项 autoSync 的默认值为 false，如果设置他为 true，那么，数据在编辑后会立即提交，也就是一行记录的修改，修改了多少字段，就会提交多少次（前提是验证能通过），一般情况下这不是好的选择，因为在没有严格信息提示和确认的情况，数据修改和删除立即提交后台处理可能会造成错误操作，尤其是误删的情况，所以不提倡设置 autoSync 为 true。

配置项 batchActions 的默认值为 true，会将相同操作类型的数据合并在一起提交，这样可减少提交请求，也为数据处理提供了方便，例如删除多个数据的时候，将数据合并提交是好的选择。而如果设置其为 false，则会将数据按一条记录一个请求方式提交数据。具体可根据实际需要设置。

在 7.6.3 节已经讨论过，配置项 writeAllFields 使用默认值 true 是好的选择，这和数据的提交与确认方式有关。配置项 encode 和 root 是配对使用的，只有设置了 root 的时候，encode 才起作用，而设置 root 的好处就是可通过习惯的方式在后台提取数据，是好的选择。为了统一在后台使用数组处理提交的 JSON 数据，所以设置配置项 allowSingle 为 false 也是好的选择，不然还得去判断提交数据是数组或对象。

根据以上的所说，本示例将采用 autoSync 使用默认值，batchActions 使用默认值，writeAllFields 使用默认值，encode 设置为 true，root 设置为“data”，allowSingle 设置为 false 的数据提交约定。

现在可以开始编码了，使用模板页创建一个文件名为 10-7.html 的页面文件，首先要做的是定义 Territories 和 Region 两个数据模型，代码如下：

```
Ext.define('Territories', {
    extend: 'Ext.data.Model',
    fields: [
           {name:'TerritoryID',type:"int"},
           'TerritoryDescription','RegionDescription',
           {name:'RegionID',type:"int"}
    ],
    idProperty:"TerritoryID",
    validations: [{
        type: 'presence',
        field: 'TerritoryDescription'
    }]
});

Ext.define('Region', {
    extend: 'Ext.data.Model',
    fields: [
           {name:'RegionID',type:"int"},
           'RegionDescription'
```

```
        ],
        idProperty:"RegionID"
    });
```

为了让字段与数据库保持一致，两个模型都修改了 idProperty 配置项，值为对应的唯一字段。模型 Territories 多定义了一个字段 RegionDescription 用来选择区域名称，还定义了一个验证确保字段 TerritoryDescription 不能为空，这样，不符合验证的数据就不会被提交。RegionID 字段作为外键，也需要一个验证，不过当前没有 Region 的数据，不能构建列表，因而，只能等 RegionStore 加载数据后，根据数据创建一个列表后再加入模型。

下面创建 TerritoriesStore，代码如下：

```
Ext.create("Ext.data.Store",{
    model:'Territories',
    autoLoad:true,
     proxy: {
         type: 'ajax',
         api:{
            read:'Territories.ashx?act=read',
            create:'Territories.ashx?act=add',
            destroy:'Territories.ashx?act=del',
            update:'Territories.ashx?act=edit'
            //java 的定义
            //read:"/Chapter10/Territories?act=read",
            //create:"/Chapter10/Territories?act=add",
            //destroy:"/Chapter10/Territories?act=del",
            //update:"/Chapter10/Territories?act=edit",
            },
         reader:{
            type: 'json',
            root:"data"
         },
         writer:{
            type:"json",
            encode:true,
            root:"data",
            allowSingle:false
         }
     },
     storeId:"TerritoriesStore"
});
```

配置项都根据约定进行了定义。配置项 API 配置了 4 种操作的提交地址和操作参数，在服务器端可根据 act 的提交值判断当前是什么操作。喜欢 Restful 方式的，可删除 API 配置项，将 Proxy 的配置项 types 修改为 rest，再定义配置项 url 来设置地址就行了。

继续，定义 RegionStore。这里要注意，在第一次加载数据的时候，要为模型 Territories 添加一个验证。RegionStore 的定义代码如下：

```
Ext.create("Ext.data.Store",{
    firstLoad:true,
    model:'Region',
    autoLoad:true,
     proxy: {
         type: 'ajax',
```

```
    api:{
       read:'Region.ashx',
       //read:"/Chapter10/Region" //java
       },
    reader:{
       type: 'json',
       root:"data",
       messageProperty:"Msg"
    }
},
storeId:"RegionStore",
    listeners:{
          datachanged:function(store,opts){
                if(this.firstLoad){
                      delete this.firstLoad;
                      var list=[],
                      data=store.data.items;
                      for(var i=0;ln=data.length,i<ln;i++){
                            list.push(data[i].data.RegionID);
                      }
    Territories.prototype.validations.push({type:"inclusion",field:"RegionID",list
        :list});
                      Ext.getCmp("add").enable();
                }
          }
    }
})
```

RegionStore 配置项只需要读取数据，因而只要定义 reader 就行了。为了确认是第一次加载数据，所以添加了一个配置项 firstLoad。在 datachanged 事件中，如果其为 true，则说明是第一次加载，将其删除后，在模型 Territories 的原型的 validations 数组加入了一个验证对象，这里一定要在原型加入，因为 Territories 是一个类，不是实例。验证对象的 list 本来可直接使用数据的里的 keys 数组，不过 keys 都是字符串，在比较时，RegionID 是整型，所以验证会出问题，所以只能通过循环提取 RegionID 构建一个 list。最后启用增加按钮，这样可确保验证已经配置完后才允许添加数据。

现在定义 Grid：

```
Ext.create("Ext.grid.Panel",{
   height:500,
   width:600,
   renderTo:Ext.getBody(),
   store:"TerritoriesStore",
   selModel:{selType:"checkboxmodel",checkOnly:true,mode:"MULTI"},
   plugins:[{ptype:"celledit ing"}],
   tbar:[
         {text:"增加",id:"add",disabled:true,handler:function(){
         }},
         {text:"删除",id:"delete",disabled:true,handler:function(){
         }},
         "|",
         {text:"保存",handler:function(){
         }},
         "|",
         {text:"刷新",handler:function(){
```

```
            }}
        ],
        columns:[
            {text:'编号',dataIndex:'TerritoryID',width:40},
            {text:'地区',dataIndex:'TerritoryDescription',flex:1,
                field:{xtype:"textfield",allowBlank:false}
            },
            {text:'所属区域',dataIndex:'RegionID',flex:1,
                field:{xtype:"combobox",allowBlank:false,autoRender:true,
                    store:"RegionStore",valueField:"RegionID",
                    displayField:"RegionDescription",forceSelection:true,
                    triggerAction:'all'
                },
                renderer:function(v,meta,rec){
                    return rec.data.RegionDescription;
                }
            }
        ]
    });
```

为了便于选择删除数据，示例使用了复选框选择模型，而且只允许使用复选框进行选择。顶部工具栏有增加、删除、保存和刷新 4 个按钮，按钮的句柄还没写。需要编辑地区和所属区域的列，因为要为它们配置编辑框。

所属区域的数据必须从 RegionStore 中获取，因而使用了一个下拉列表框。这里有个问题，就是列的 dataIndex 是使用 RegionID 还是 RegionDescription，如果使用 RegionDescription，那么对下拉列表框设置值的时候就不是真实值，而是显示值，这时候下拉列表框对原值的显示就会有问题，因而这里必须使用 RegionID 作为列字段，为了显示显示值，所以定义了 renderer 函数，它会将显示值返回给列，这样列中看到的就不是 RegionID 了。这里还有个问题没解决，使用下拉选择框只会改变 RegionID 的值，并不改变 RegionDescription 的值，所以编辑 RegionID 后，还是会显示原来的 RegionDescription 的值，而不是 RegionID 对应的显示值，因而，在修改 RegionID 的值后，需要把 RegionDescription 的值也修改了。解决的办法是在 CellEditing 对象的 edit 事件中，监测到 RegionID 字段被修改后，就修改 RegionDescription 的值，因而，可在 plugins 数组内的 CellEditing 配置对象中添加以下代码：

```
listeners:{
    edit:function(edit,e){
        if(e.field=="RegionID"){
            var regionStore=Ext.StoreManager.lookup("RegionStore");
            var rec=regionStore.getById(e.value);                    e.record.
                set("RegionDescription",rec.data.RegionDescription);
        }
    }
}
```

代码判断到编辑修改的字段是 RegionID 后，就从 RegionStore 中找到其对应的显示值，然后使用 set 方法修改 RegionDescription 的值，这样，就可改变显示值了。

注意 该方法在 4.0.7 版本是没有问题的，但在 4.1 Beta 1 版本下就还是显示原来的值，问

题的关键在 set 方法设置值后，没有再一次调用 renderer 方法返回新的数据，而继续使用了原来的数据。使用 set 方法修改数据并需要配合 renderer 进行显示的要小心，在错误没有修正之前，要想另外的办法解决。

“保存”按钮的作用就是执行同步，这个容易，其句柄函数内代码如下：

```
this.up("gridpanel").store.sync();
```

刷新按钮句柄函数代码也简单，使用 load 方法重新加载数据就行，代码如下：

```
this.up("gridpanel").store.load();
```

单击“增加”按钮后，要做的事情是创建一个 Territories 模型的实例，然后在 Territory-Description 列内显示编辑框，其代码如下：

```
var grid=this.up("gridpanel");
var edit=grid.plugins[0];
var rec=new Territories({
   TerritoryDescription:"",
   RegionDescription:""
});
edit.cancelEdit();
grid.store.insert(0,rec);
edit.startEditByPosition({row:0,column:2});
```

创建 Territories 实例，TerritoryID 和 RegionID 两个字段，可以加，也可以不加，这个问题不大，因为提交时验证会把关。调用 cancelEdit 的目的是避免用户在编辑状态时单击了“增加”按钮，从而影响当前的编辑。调用 insert 方法在第一行插入一个记录，然后就可以调用 startEditByPosition 方法显示编辑框了。插入新记录的位置可根据自己情况设置，记得修改 startEditByPosition 方法中的 row 定义就行。

“删除”按钮默认是禁用的，需要在有选择的时候启用它，这个不难，在 Grid 监听 selectionchange 事件就可知道是否有选择了，代码如下；

```
listeners:{
   selectionchange:function(view,rs){
          Ext.getCmp("delete").setDisabled(rs.length==0);
   }
}
```

删除记录为了保险起见，一般都需要确认一下是否删除，然后再使用 remove 方法将选择的记录从 Store 中删除，代码如下：

```
var grid=this.up("gridpanel");
var rs=grid.getSelectionModel().getSelection();
if(rs.length > 0){
   var content=" 确定删除以下地区？ </br>";
   for(var i=0;ln=rs.length,i<ln;i++){
          content+=rs[i].data.TerritoryDescription+"<br/>";
   }
```

```
        Ext.Msg.confirm("删除记录",content,function(btn){
                if(btn=="yes"){
                        this.store.remove(this.getSelectionModel().getSelection());
                }
        },grid)
}
```

页面代码已经完成了，现在要考虑服务器端代码了。首先要做的是根据 act 的提交值分别调用 Add、Edit、Del 和 Read 等方法，这个不难，具体请阅读本书资源包内的源代码。现在要考虑的是如何处理提交上来的增加、编辑和删除的数据，数据是以 JSON 格式（数组）、通过 data 变量提交的，在服务器端获取到 data 后，直接利用 JSON 对象的功能将其转换为对象，这样处理起来就方便多了。现在，先定义 Territory 对象：

C#

```
public class Territory {
    public int TerritoryID { set; get; }
    public string TerritoryDescription { set; get; }
    public int RegionID { set; get; }
    public string RegionDescription { set; get; }
}
```

Java

```
public class Territory {
   int TerritoryID;
   int RegionID;
   String TerritoryDescription,RegionDescription;
}
```

接着定义一个方法用来将提交的 JSON 字符串转换为对象：

C#

```
private List<Territory> ProcessData(string data)
{
    List<Territory> territories = new List<Territory>();
    territories = JsonConvert.DeserializeObject<List<Territory>>(data);
    return territories;
}
```

Java

```
protected Territory[] ProcessData(String data){
   Gson gson = new Gson();
   Territory[] territories =gson.fromJson(data, Territory[].class);
   return territories;
}
```

这个一定要根据服务器端使用的 JSON 库的特性去写，充分利用其功能。

现在添加 Add 方法用于添加记录，代码如下：

C#

```
private string Add(HttpRequest request)
```

```
{
    string data=request.Params["data"]?? "";
    List<Territory> territories = new List<Territory>();
    if (data.Length > 0)
    {
        territories = ProcessData(data);
        string output = "";
        try
        {
            using (NorthwindEntities ne = new NorthwindEntities())
            {
string id = ne.Territories.Max(m => m.TerritoryID);
int sid = 0;
int.TryParse(id, out sid);
for (int i = 0; i < territories.Count; i++)
{
    Territory s = territories[i];
    s.TerritoryID = sid + i + 1;
    NorthwindModel.Territories d = new NorthwindModel.Territories
    {
        TerritoryID = s.TerritoryID.ToString(),
        TerritoryDescription = s.TerritoryDescription,
        RegionID = s.RegionID
    };
    ne.AddToTerritories(d);
    ne.SaveChanges();
}
output = new JObject{
    new JProperty("success",true),
                    new JProperty("data",JArray.Parse(JsonConvert.Serialize-
                        Object(territories)))
                }.ToString();
            }
        }
        catch (Exception e)
        {
            return "{'success':false,'Msg':'"+e.Message+"'}";
        }
        return output;
    }
    else
    {
        return "{'success':false,'Msg':'错误的提交数据！ '}";
    }
}
```

Java

```
protected String Add(HttpServletRequest request){
   if(request.getParameter("data")!=null){
        String connectionUrl = "jdbc:sqlserver://192.168.0.254:1433;" +
        "databaseName=Northwind;;user=sa;password=abcd-1234";
       Connection con = null;
       Statement stmt = null;
       ResultSet rs = null;
```

```
ResultSet rscount=null;
try {
            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
            con = DriverManager.getConnection(connectionUrl);

            int maxId = 0;
            stmt = con.createStatement();
            rscount=stmt.executeQuery("select max(cast(TerritoryID as int))
                as id from Territories");
            rscount.next();
            maxId=rscount.getInt(1);

            StringBuilder sb =new StringBuilder();
            Territory[]  territories=ProcessData(request.getParameter
                ("data"));
            for (int i = 0; i < territories.length; i++) {
                Territory t= territories[i];
                t.TerritoryID=maxId+i+1;
                sb.append("insert into Territories values (");
                sb.append(t.TerritoryID);
                sb.append(",");
                sb.append(t.TerritoryDescription);
                sb.append(",");
                sb.append(t.RegionID);
                sb.append(");");
            }
            stmt.execute(sb.toString());
            Gson gson=new Gson();
                        return "{'success':true,'data':"+gson.toJson(territories)+"}";
            }
            catch (Exception e) {
                        return "{'success':false,'Msg':'"+e.getMessage()+"'}";
            }
            finally {
                if (rscount != null) try { rscount.close(); } catch(Exception e) {}
                if (rs != null) try { rs.close(); } catch(Exception e) {}
                if (stmt != null) try { stmt.close(); } catch(Exception e) {}
                if (con != null) try { con.close(); } catch(Exception e) {}
            }
        }else{
                return "{'success':false,'Msg':'错误的提交数据！ '}";
        }
}
```

因为 TerritoryID 在表中不是字段生成的，为了便于演示，这里会取最大值，然后累加上去做 TerritoryID 的值。在编写应用的时候要根据表的情况做处理。

在取得提交的对象数组或列表后，可使用循环创建新的实体对象或构建 SQL 代码，然后将其添加到数据库就行了。关键是在循环中，记得修改 Territory 对象中的 TerritoryID 属性的值，然后再直接使用 JSON 库的功能将其转换为字符返回。在返回的 JSON 对象中，一定要加入 success 属性，这样可根据操作结果返回 true 或 false，表示操作是成功还是发生了错误。如果是发生了错误，可根据 Render 对象中 messageProperty 配置项的值返回错误信息，在本示例在，其值为“Msg”，可以看到服务器代码返回错误信息的属性都是“Msg”。

完成 Add 方法，Edit 和 Del 方法也等于完成了一半，流程是基本一样的，如果使用实体框架，先根据 TerritoryID 查询出记录，然后再修改或删除字段值即可。如果是组合 SQL 字符串，将 Territory 对象的属性值组合成 update 或 delete 的查询语句即可。Read 方法不需要使用提交数据，只需要把数据查询处理转换为 JSON 格式返回即可。具体代码可阅读本书资源包中的 Chapter10 目录下的相关文件。

（3）页面效果

在浏览器中打开页面，然后添加一条记录，在地区输入北京，然后在所属区域选择 Northern，完成编辑后，单击“保存”按钮将将看到如图 10-24 所示过程。

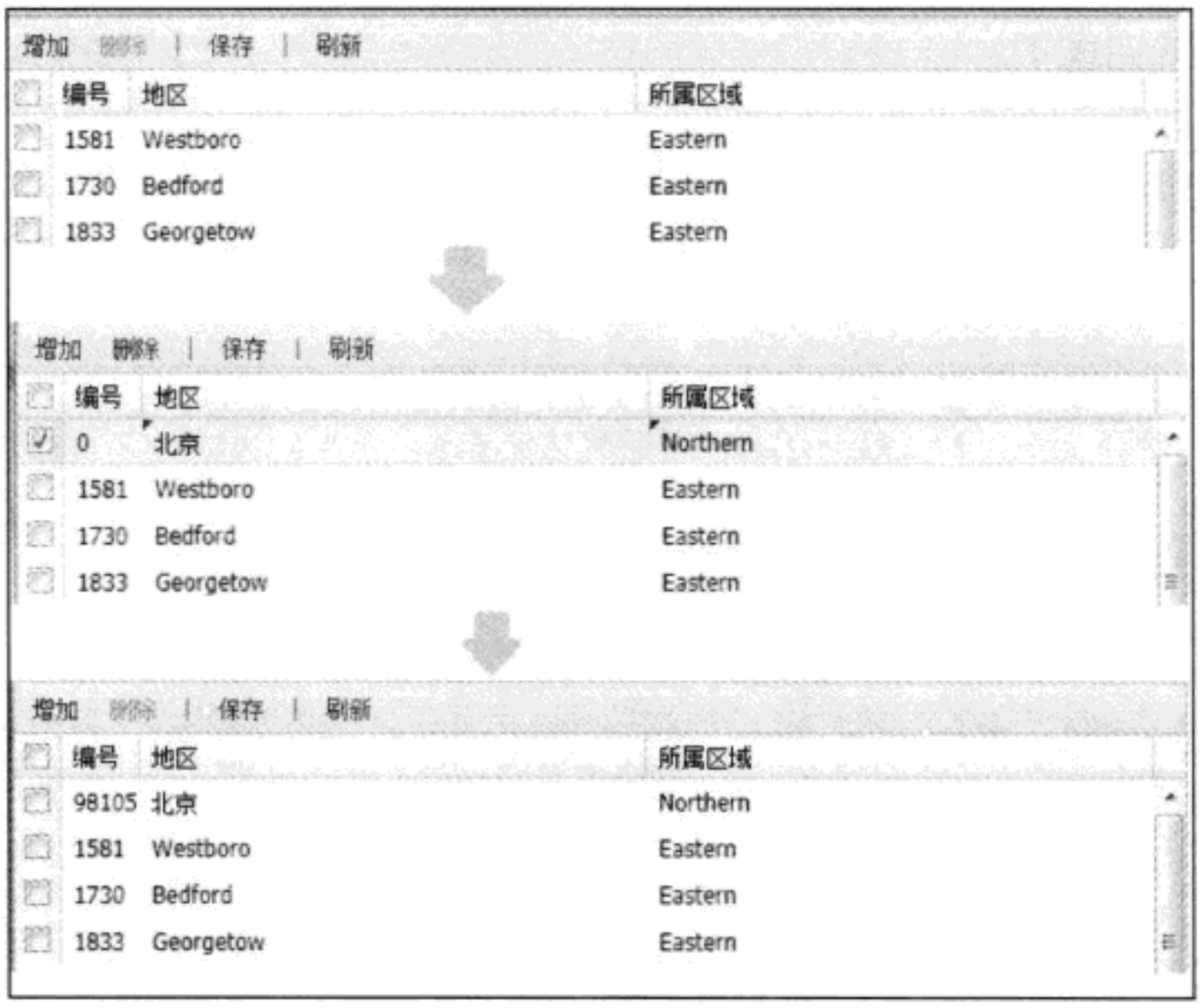


图 10-24　示例的页面操作效果

10.7.6　使用 RowEditing 实现数据的增删改

（1）功能描述

本示例将使用 RowEditing 对象在 Grid 中实现编辑 Northwind 库中 Territories 表的数据，这需要使用到相关表 Region。

（2）实现代码

RowEditing 与 Celleditng 只是编辑方式的不同，而数据的提交与服务器端处理是一样的，因而可很容易的从 Celleditng 中转换过来。

复制实例 10-7.html 文件，然后将文件名修改为 10-8.html，修改 plugins 配置项的定义，将配置项 ptype 配置项修改为 rowediting，并加入配置项 autoCancel，值为 false，这意味不会自动取消当前编辑，必须需要单击 RowEditing 上的“保存”或“取消”按钮才可结束当前编辑。RowEditing 对象的 edit 事件，是行编辑，没有具体的字段，因而要将 edit 事件做如下修改：

```
edit:function(edit,e){
    var regionStore=Ext.StoreManager.lookup("RegionStore");
    var rec=regionStore.getById(e.record.data.RegionID);
    e.record.set("RegionDescription",rec.data.RegionDescription);
}
```

没有具体的字段值，因而要从记录中获取 RegionID 值。

“增加”按钮中，进入编辑状态是基于单元格的，而行编辑是基于行的，因而 startEditByPosition 方法必须修改为 startEdit，具体修改如下：

```
edit.startEdit(0,0);
```

至此，修改就完成了。

（3）页面效果

在浏览器中打开页面，然后单击“增加”按钮，在地区中输入广州，并选择所属区域为 Southern，单击“保存”后确认修改，再单击工具栏上的“保存”按钮，将看到如图 10-25 所示的过程。

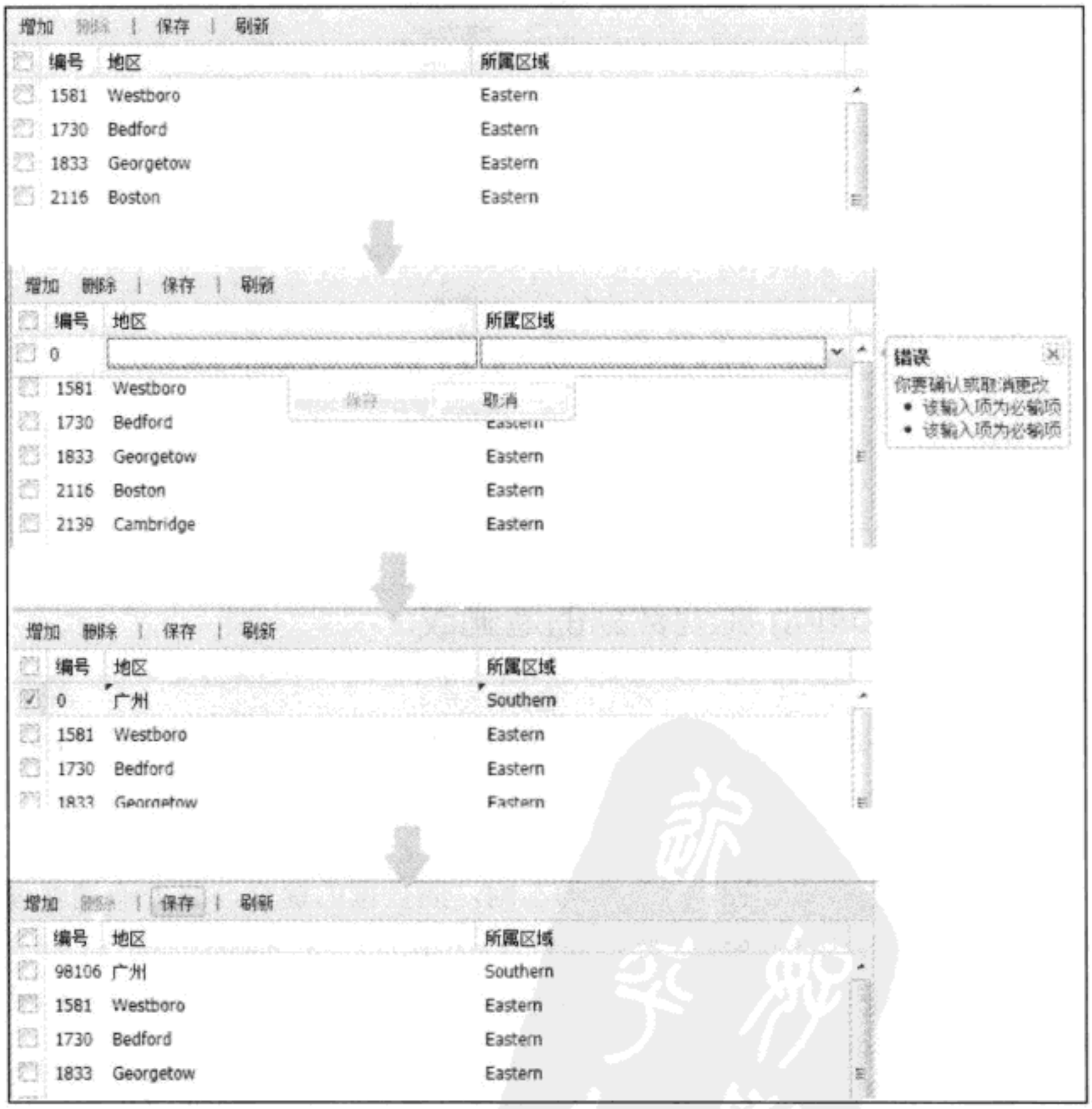


图 10-25　示例的页面操作效果

10.7.7　主从表的显示

（1）功能描述

本示例将演示如何使用 Grid 显示主从表。从表的数据提取方式有两种，一种是使用模型的 hasMany 配置项，将从表的数据附加到主表的记录中；一种是主从表分开查询，当主表的选择行发生变化后，再远程加载从表的数据。两种方法中，第一种方法因为数据已经在本地，从表切换会比较迅速，第二种方法还要从远程加载，因而有延时的感觉，用户体验不太好，所以本示例将使用第一种方法加载数据。

（2）实现代码

本示例难度不大，注意一下配置项就行。使用模板页创建一个名称为 10-9.html 的页面文件，然后创建 Order 模型：

```
Ext.define('Order', {
    extend: 'Ext.data.Model',
    fields: [
           {name:'OrderID',type:"int"},
           'CustomerID','CustomerName',
           {name:'OrderDate',type:"date",format:"Y-m-d"}
    ],
    idProperty:"OrderID",
    hasMany:{model:"OrderDetail",name:"OrderDetails",foreignKey:"OrderID"}
});
```

这里定义了 hasMany 配置项，使用的模型是 OrderDetail，在返回的数据中读取从表数据的属性名称是 OrderDetails，因而在后台返回数据时要注意将从表数据放到该属性下。还定义了 OrderDetail 模型的外键是 OrderID。

下面要定义模型 OrderDetail：

```
Ext.define('OrderDetail',{
    extend: 'Ext.data.Model',
    fields: [
           {name:'OrderID',type:"int"},
           {name:'ProductID',type:"int"},
           {name:'UnitPrice',type:"float"},
           {name:'Quantity',type:"int"},
           {name:'Discount',type:"float"},
           'ProductName'
    ]
});
```

模型定义好以后，要为它们各创建一个 Store：

```
Ext.create("Ext.data.Store",{
    model:'Order',
    autoLoad:true,
    pageSize:50,
     proxy: {
         type: 'ajax',
         api:{
            read:'Orders1.ashx'
```

```
            //read:"/Chapter10/Orders1" //java
            },
        reader:{
            type: 'json',
            root:"data"
        }
    },
    storeId:"OrderStore"
})

Ext.create("Ext.data.Store",{
    model:'OrderDetail',
     storeId:"OrderDetailStore"
})
```

OrderDetailStore 是在本地加载数据的，因而不需要定义 autoLoad、Proxy 这些配置项。接着，完成两个 Grid 的定义：

```
Ext.create("Ext.panel.Panel",{
    layout:"vbox",
    height:600,
    renderTo:Ext.getBody(),
    items:[
            {xtype:"gridpanel",
                    title:"主表",
                    flex:1,
                    width:600,
                    tbar:{xtype:"pagingtoolbar",store:"OrderStore"},
                    selMode:{mode:"SINGLE"},
                    store:"OrderStore",
                    columns:[
                            {xtype:"rownumberer",sortable:false,width:60},
                            {text:'订单号',dataIndex:'OrderID'},
                            {text:'客户编号',dataIndex:'CustomerID'},
                            {text:'客户名称',dataIndex:'CustomerName',sortable:false,flex:1},
                            {xtype:"datecolumn",text:'订购日期',dataIndex:'OrderDate'
                                ,format:"Y-m-d",width:100}
                    ]
            },
            {xtype:"gridpanel",
                    id:"DetailsGrid",
                    title:"从表",
                    flex:1,
                    width:600,
                    store:"OrderDetailStore",
                    columns:[
                    {xtype:"rownumberer",sortable:false,width:60},
                    {text:'产品名称',dataIndex:'ProductName',sortable:false,f
                        lex:1},
                    {xtype:"numbercolumn",text:'单价',dataIndex:'UnitPrice',
                        align:"right",format:"0,0.00"},
                    {xtype:"numbercolumn",text:'数量',dataIndex:'Quantity',f
                        ormat:"0,0"},
                    {xtype:"numbercolumn",text:'折扣',dataIndex:'Discount',f
                        ormat:"0,0.00"}
```

```
                ]
            }
        ]
});
```

这里主表设置了单选模式，目的是让从表的显示清晰点，只显示选择行的从表数据。如果是多选的，就要考虑是默认显示第一条还是最后一条选择记录的从表数据。

好了，现在要考虑怎么加载从表数据了，基本思路是，当用户选择了主表记录后，就立即调用 OrderDetailStore 的 loadRecords 方法加载表数据。每当用户选择主表记录时，都会触发主表的 selectionchange 事件，因而在该事件内调用 loader 方法是最佳选择，代码如下：

```
selectionchange:function(model,rs){
    if(rs.length>0){
            var g=Ext.getCmp("DetailsGrid"),
                store=model.view.store;
            g.store.loadRecords(rs[0].OrderDetailsStore.data.items);
    }
}
```

代码先判断是否有选择的记录，如果有，将记录的 OrderDetailsStore 属性指向的从表数据加载到 OrderDetailStore 就行了。

还有个小问题，就是在页面刚开始显示时，必须选择一条记录，不然，从表的显示会是空的，因而在主表渲染了数据后，必须默认选择第一条记录，在主表的 Grid 定义中加入 viewConfig 配置项，监听 refresh 事件，代码如下：

```
viewConfig:{
    listeners:{
            refresh:function(){
                    this.select(0)
            }
    }
},
```

这样，页面就已经完成了，很简单。服务器端代码会有点复杂。如果是使用实体模型，则简单很多，可直接一次查询获取从表数据。而使用组合 SQL 语句的方式，就要做两次查询，第一次查询主表数据。第二次查询与主表关联的所有从表的数据，然后将从表数据以 OrderID 为属性名称构建一个 JSON 对象，数值是一个 JSON 数组，包含对应 OrderID 的所有从表数据组成的 JSON 对象。这样，在将主表数据转换为 JSON 数据的时候，就可根据 OrderID 从构建的 JSON 对象中将从表数据组成的数组直接添加到 OrderDetails 属性中。这个难度不算太大，就是代码多点。以下是服务器端的代码：

C#

```
public void ProcessRequest (HttpContext context) {
    context.Response.ContentType = "text/javascript";
    int start = 0;
    int limit = 50;
    int.TryParse(context.Request.Params["start"], out start);
    string sort = "it.OrderID";
```

```
if (context.Request.Params["sort"] != null)
{
    sort = "";
    JArray ja = JArray.Parse(context.Request.Params["sort"]);
    foreach (JObject jo in ja)
    {
        sort += "it." + (string)jo["property"] + " " + (string)jo["direction"] + ",";
    }
    sort = sort.Substring(0, sort.Length - 1);
}
using (NorthwindEntities ne = new NorthwindEntities())
{
    int total = ne.Orders.Count();
    int a = total;
    if (total < start) start = 0;
    var q = ne.Orders.OrderBy(sort).Skip(start).Take(limit).Select(m => new
    {
        m.OrderID,
        m.CustomerID,
        m.OrderDate,
        CustomerName = m.Customers.CompanyName,
        m.Order_Details
    }).ToList();

    //返回JSON格式数据
    var output = new JObject
    {
        new JProperty("total",total),
        new JProperty("success",true),
        new JProperty("data",new JArray(
    from p in q
    select new JObject(
        new JProperty("OrderID",p.OrderID),
        new JProperty("CustomerID",p.CustomerID),
        new JProperty("OrderDate",Convert.ToDateTime(p.OrderDate).
            ToString("yyyy-MM-dd")),
        new JProperty("CustomerName",p.CustomerName),
        new JProperty("OrderDetails",new JArray(
            from m in p.Order_Details
                select new JObject(
                    new JProperty("OrderID",m.OrderID),
                    new JProperty("ProductID",m.ProductID),
                    new JProperty("ProductName",m.Products.
                        ProductName),
                    new JProperty("UnitPrice",m.UnitPrice),
                    new JProperty("Quantity",m.Quantity),
                    new JProperty("Discount",m.Discount)
                )
            ))
        )
    )
    );
    context.Response.Write(output.ToString());
}
}
```

Java

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
   // TODO Auto-generated method stub
   response.setContentType("text/javascript; charset=utf-8");
    String connectionUrl = "jdbc:sqlserver://192.168.0.254:1433;" +
    "databaseName=Northwind;;user=sa;password=abcd-1234";
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;
ResultSet rscount=null;
ResultSet rs1 = null;
int start= 0;
if(request.getParameter("start") != null){
      start=Integer.parseInt(request.getParameter("start")) ;
}
int limit=50;
String sort="OrderID";
if(request.getParameter("sort")!=null){
      sort="";
      JsonParser jparser = new JsonParser();
    JsonArray ja = jparser.parse(request.getParameter("sort")).
       getAsJsonArray();
    for (JsonElement je : ja) {
      JsonObject jo = je.getAsJsonObject();
      sort+=jo.get("property").getAsString()+ " "
                  + jo.get("direction").getAsString()+",";
    }
    sort=sort.substring(0,sort.length()-1);
}
try {

   Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
   con = DriverManager.getConnection(connectionUrl);

   int count = 0;
   stmt = con.createStatement();
   rscount=stmt.executeQuery("select count(OrderID) as count from Orders");
   rscount.next();
   count=rscount.getInt(1);
   if(start>count){
         start=0;
   }
   StringBuilder sb = new StringBuilder();
   sb.append("select top 50 OrderID,CustomerID,OrderDate,");
   sb.append("CustomerName=(select CompanyName from Customers");
   sb.append(" where Customers.CustomerID=Orders.CustomerID)");
   sb.append(" from Orders where OrderID ");
   sb.append(" not in (select top ");
   sb.append(start);
   sb.append(" OrderID from Orders  ");
   sb.append(" order by ");
   sb.append(sort);
   sb.append(")");
```

```
sb.append("  order by ");
sb.append(sort);

StringBuilder subSQL=new StringBuilder();
subSQL.append("select *,ProductName=");
subSQL.append("(select ProductName from Products ");
subSQL.append("where Products.ProductID=[Order Details].ProductID) ");
subSQL.append("from [Order Details] where OrderID in (");
subSQL.append("select top 50 OrderID from Orders where OrderID ");
subSQL.append(" not in (select top ");
subSQL.append(start);
subSQL.append(" OrderID from Orders  ");
subSQL.append(" order by ");
subSQL.append(sort);
subSQL.append("))");
subSQL.append("  order by OrderID");

rs1 = stmt.executeQuery(subSQL.toString());
JsonObject subs=new JsonObject();
String id = "";
while (rs1.next()) {
      id=rs1.getString("OrderID");
      if(!subs.has(id)){
            subs.add(id, new JsonArray());
      }
      JsonObject jd=new JsonObject();
      jd.addProperty("OrderID", rs1.getInt("OrderID"));
      jd.addProperty("ProductID", rs1.getInt("ProductID"));
      jd.addProperty("Quantity", rs1.getInt("Quantity"));
      jd.addProperty("UnitPrice", rs1.getFloat("UnitPrice"));
      jd.addProperty("Discount", rs1.getFloat("Discount"));
      jd.addProperty("ProductName", rs1.getString("ProductName"));
      subs.getAsJsonArray(id).add(jd);
}
rs1.close();
rs = stmt.executeQuery(sb.toString());
//返回JOSN格式数据
java.text.DateFormat df = new java.text.SimpleDateFormat("yyyy-MM-dd");
JsonArray array=new JsonArray();
while (rs.next()) {
      id=rs.getString("OrderID");
      JsonObject obj= new JsonObject();
      obj.addProperty("OrderID", rs.getString("OrderID"));
      obj.addProperty("CustomerID", rs.getString("CustomerID"));
      obj.addProperty("OrderDate", df.format(rs.getDate("OrderDate")));
      obj.addProperty("CustomerName", rs.getString("CustomerName"));
      obj.add("OrderDetails", subs.getAsJsonArray(id));
      array.add(obj);
}
JsonObject json=new JsonObject();
json.addProperty("total", count);
json.addProperty("success",true);
json.add("data", array);
response.getWriter().write(json.toString());
rscount.close();
```

```
    rs.close();
}
catch (Exception e) {
        response.getWriter().write(e.getMessage());
}
finally {
    if (rscount != null) try { rscount.close(); } catch(Exception e) {}
    if (rs != null) try { rs.close(); } catch(Exception e) {}
    if (rs1 != null) try { rs1.close(); } catch(Exception e) {}
    if (stmt != null) try { stmt.close(); } catch(Exception e) {}
    if (con != null) try { con.close(); } catch(Exception e) {}
        }

    }
```

实体模型在主表的查询中就可以取得从表的数据 Order_Details，因而在组件 JSON 时，通过嵌套 LINQ 查询就可直接输出了。

使用 SQL 查询，构建查询语句有点费劲，后面的就简单了。粗体代码的目的是在 JSON 对象中，如果发现当前 OrderID 对应的成员不存在，则需要先创建这个成员，其值是 JSON 数组，然后就可通过 getAsJsonArray 方法访问该数据，并将从表的 JSON 数据对象加入到数组中。在主表的数据循环中，也是通过 getAsJsonArray 方法将从表的数据直接添加到 OrderDetails 属性中。

至此，示例就完成了。

（3）页面效果

在浏览器打开页面，将看到图 10-26 所示的效果。

如果希望主表和从表的加载是分开的，可修改 selectionchange 事件，在事件中为 OrderDetailStore 的 Proxy 对象的 extraParams 配置项添加一个提交参数，例如 orderid，然后调用 load 方法加载数据，服务器端根据 orderid 查询从表数据返回就可以了，具体代码可以这样写：

```
selectionchange:function(model,rs){
    if(rs.length>0){
            var ods=Ext.Ext.StoreManager.loolup("OrderDetailStore"),
                rec=rs[0];
            ods.getProxy().extraParams.orderid=rec.data.OrderID;
            ods.load();
    }
}
```

当然，这还得为 OrderDetailStore 定义 Proxy、Reader 等配置项。

10.8　本章小结

本章主要讲述了 Ext JS 4 重构后的 Grid，笔者的感觉是，重构后的 Grid 使用起来比之前的版本简单。至于性能，笔者的感受比 Ext JS 3.x 版本的好，使用 IE 6 的读者可能会不赞成这点，对于这个古董级别的浏览器，笔者也没什么好说的，希望它早点消失吧。

主表

第 1 页,共 17 页

	订单号	客户编号	客户名称	订购日期
1	10248	VINET	Vins et alcools Chevalier	1996-07-04
2	10249	TOMSP	Toms Spezialitäten	1996-07-05
3	10250	HANAR	Hanari Carnes	1996-07-08
4	10251	VICTE	Victuailles en stock	1996-07-08
5	10252	SUPRD	Suprêmes délices	1996-07-09
6	10253	HANAR	Hanari Carnes	1996-07-10
7	10254	CHOPS	Chop-suey Chinese	1996-07-11
8	10255	RICSU	Richter Supermarkt	1996-07-12
9	10256	WELLI	Wellington Importadora	1996-07-15

从表

	产品名称	单价	数量	折扣
1	Queso Cabrales	14.00	12	0.00
2	Singaporean Hokkien Fried Mee	9.80	10	0.00
3	Mozzarella di Giovanni	34.80	5	0.00

图 10-26 页面效果

要使用好 Grid，笔者还是那个观点，先分清楚，如果数据与 UI 是分离的，那么数据的操作如排序、过滤等都是在 Store 这层进行的，UI 只负责显示及与用户进行互动。

对于分页滚动条，笔者的意见是，目前还不太实用，缺点也比较明显。当然，如果习惯早期 Web 应用操作习惯的，这个问题不大，可以使用 ActionColumn 对行进行操作。

第 11 章　与 Grid 同源的树

Ext JS 4 中的树与 Grid 一样，都派生于 TablePanel，因而，它们是同源的。因为新的树派生于 TablePanel，所以在 Grid 中可用的附加功能和插件基本都可以在树中使用。而 TreeGrid 的实现也简单很多，无非就是多了几列数据。基本上可以说，树就是一个只有一列且没有标题栏的 Grid。

本章主要讲述树的构成及如何使用树。

11.1　树的构成

11.1.1　概述

树派生于 TablePanel，因而与 TablePanel 一样，它也有视图、Store、列和选择模型等组件。在默认情况下，树会将 Treeview 对象作为树的视图，将 TreeStore 对象作为树的 Store，将 Treecolumn 对象作为列，将 TreeModel 对象作为选择模型，在 Treeview 对象中，会将 NodeStore 作为 Store。

11.1.2　树面板的运行流程：Ext.tree.Panel

TreePanel 派生于 TablePanel，因此它遵循 TablePanel 的运作流程，不过 TreePanel 也有自己的运作流程，其构造函数如下：

```
constructor: function(config) {
    config = config || {};
    if (config.animate === undefined) {
        config.animate = Ext.enableFx;
    }
    this.enableAnimations = config.animate;
    delete config.animate;

    this.callParent([config]);
},
```

在这里主要处理了一些与动画有关的配置项。

TreePanel 的 initComponent 方法的代码如下：

```
initComponent: function() {
    var me = this,
        cls = [me.treeCls],
   view
```

```
if (me.useArrows) {
    cls.push(Ext.baseCSSPrefix + 'tree-arrows');
    me.lines = false;
}
if (me.lines) {
    cls.push(Ext.baseCSSPrefix + 'tree-lines');
} else if (!me.useArrows) {
    cls.push(Ext.baseCSSPrefix + 'tree-no-lines');
}
if (Ext.isString(me.store)) {
    me.store = Ext.StoreMgr.lookup(me.store);
} else if (!me.store || Ext.isObject(me.store) && !me.store.isStore) {
    me.store = new Ext.data.TreeStore(Ext.apply({}, me.store || {}, {
        root: me.root,
        fields: me.fields,
        model: me.model,
        folderSort: me.folderSort
    }));
} else if (me.root) {
    me.store = Ext.data.StoreManager.lookup(me.store);
    me.store.setRootNode(me.root);
    if (me.folderSort !== undefined) {
        me.store.folderSort = me.folderSort;
        me.store.sort();
    }
}
me.viewConfig = Ext.apply({}, me.viewConfig);
me.viewConfig = Ext.applyIf(me.viewConfig, {
    rootVisible: me.rootVisible,
    animate: me.enableAnimations,
    singleExpand: me.singleExpand,
    node: me.store.getRootNode(),
    hideHeaders: me.hideHeaders
});
me.mon(me.store, {
    scope: me,
    rootchange: me.onRootChange,
    clear: me.onClear
});
me.relayEvents(me.store, [
//省略事件代码
]);
me.store.on({
    append: me.createRelayer('itemappend'),
    remove: me.createRelayer('itemremove'),
    move: me.createRelayer('itemmove'),
    insert: me.createRelayer('iteminsert'),
    beforeappend: me.createRelayer('beforeitemappend'),
    beforeremove: me.createRelayer('beforeitemremove'),
    beforemove: me.createRelayer('beforeitemmove'),
    beforeinsert: me.createRelayer('beforeiteminsert'),
    expand: me.createRelayer('itemexpand'),
    collapse: me.createRelayer('itemcollapse'),
    beforeexpand: me.createRelayer('beforeitemexpand'),
    beforecollapse: me.createRelayer('beforeitemcollapse')
```

```
});
if (!me.columns) {
    if (me.initialConfig.hideHeaders === undefined) {
        me.hideHeaders = true;
    }
    me.columns = [{
        xtype    : 'treecolumn',
        text     : 'Name',
        width    : Ext.isIE6 ? null : 10000,,
        dataIndex: me.displayField
    }];
    }
    if (me.cls) {
        cls.push(me.cls);
    }
    me.cls = cls.join(' ');
    me.callParent();
    view = me.getView();
    me.relayEvents(view, [
        //省略事件代码
    ]);
    if (!me.getView().rootVisible && !me.getRootNode()) {
        me.setRootNode({
            expanded: true
        });
    }
},
```

在代码开始处，根据配置项把样式类的名称收集到 cls 数组中。

接着创建 Store。如果 Store 配置项是字符串，从 StoreMgr 中找 Store；如果 Store 配置项不存在，或者是对象但不是 Store 对象，则创建 TreeStore 对象（注意创建 TreeStore 对象的配置项包括 root、fields、model 和 folderSort，这个在 TreeStore 的运行流程中会讲述，暂时跳过）；如果是定义好的 Store 对象，则从 StoreManager 中确认一次，然后调用 setRootNode 方法设置 Store 的根节点。如果设置了排序，则对数据执行一次排序。

接下来处理视图的配置项。

接着将 Store 的 rootchange 事件绑定到 onRootChange 方法，一旦根节点发生改变，调用 setRootNode 方法重新设置根节点；将 clear 事件绑定到 onClear 方法，以便清理视图。

接着为 Store 定义传播事件，还要为 Store 绑定节点操作方法。

接着是对列进行初始化，如果没有定义配置项 columns，则准备创建一个 TreeColumn 对象，并且不显示标题栏。

组合好样式后，调用父对象的 initComponent 方法开始初始化组件。完成组件初始化后，将视图的事件传播到面板，如果根节点是隐藏的，或者没有定义根节点，则展开根节点，开始加载数据。

从以上代码可以看出，TreePanel 的流程比 GridPanel 的流程复杂点，主要是 TreePanel 使用了几个专用的类，如 TreeStore、TreeColumn 等对象。

11.1.3 TreeStore 的运行流程：Ext.data.TreeStore

TreeStore 对象直接派生于 AbstractStore 对象，因而其在使用上与 Store 对象会有区别，这从其构造函数就能看出来，其构造函数的代码如下：

```
constructor: function(config) {
    var me = this,
        root,
        fields;
    config = Ext.apply({}, config);
    fields = config.fields || me.fields;
    if (!fields) {
        config.fields = [{name: 'text', type: 'string'}];
    }
    me.callParent([config]);
    me.tree = new Ext.data.Tree();
    me.relayEvents(me.tree, [
          //省略事件定义代码
    ]);
    me.tree.on({
        scope: me,
        remove: me.onNodeRemove,
        beforeexpand: me.onBeforeNodeExpand,
        beforecollapse: me.onBeforeNodeCollapse,
        append: me.onNodeAdded,
        insert: me.onNodeAdded,
sort: me.onNodeSort
    });
    me.onBeforeSort();
    root = me.root;
    if (root) {
        delete me.root;
        me.setRootNode(root);
    }
    //省略过时代码
},
```

如果没有定义字段（配置项 fields），则默认构建一个包含 text 字段，也就是树节点的显示文本的字段，然后调用 AbstractStore 对象的构造函数创建模型。

接着创建 TreeData 对象，从 7.5 节可了解到，和 MixedCollection 对象的作用类似，TreeData 对象的主要功能是为节点提供容器。**在这里，可以知道，要访问树的节点，使用的是 TreeStore 的 tree 属性，而不是 Store 对象的 data 属性。**

接着是定义要传播事件，然后将由面板的用户操作所触发的事件绑定到 TreeStore 对象的方法，实现节点操作。

在调用 onBeforeSort 方法初始化排序对象后，如果定义了根节点（root 配置项），那么调用 setRootNode 方法处理根节点，其代码如下：

```
setRootNode: function(root) {
    var me = this;
    root = root || {};
```

```
 if (!root.isNode) {
     Ext.applyIf(root, {
         id: me.defaultRootId,
         text: 'Root',
         allowDrag: false
     });
Ext.data.NodeInterface.decorate(me.model);
     root = Ext.ModelManager.create(root, me.model);
 } else if (root.isModel && !root.isNode) {
     Ext.data.NodeInterface.decorate(me.model);
 }

     me.getProxy().getReader().buildExtractors(true);
     me.tree.setRootNode(root);
     if (preventLoad !== true && !root.isLoaded() && (me.autoLoad === true || root.
         isExpanded())) {
         me.load({
             node: root
         });
     }
     return root;
},
```

如果参数 root 不存在，则为 root 构建一个空对象。

如果 root 不是模型（isModel 不存在或为 false），则将最基本的根节点属性复制到 root 中，并调用 decorate 方法修饰节点，最后创建一个根节点的模型实例。

如果 root 是模型，而不是节点，则调用 decorate 方法将其修饰为节点。

接着重新配置 Reader 对象，这是因为使用 decorate 方法修饰节点后多了字段，需要 Reader 去处理这些字段。

接着调用 TreeData 的 setRootNode 方法，在 TreeData 中注册根节点，代码如下：

```
setRootNode : function(node) {
    var me = this;
    me.root = node;
    if (me.fireEvent('beforeappend', null, node) !== false) {
        node.set('root', true);
        node.updateInfo();
   node.commit();
        node.on({
            scope: me,
            insert: me.onNodeInsert,
            append: me.onNodeAppend,
            remove: me.onNodeRemove
        });
        me.relayEvents(node, [
           // 省略事件定义代码
        ]);
    me.nodeHash = {};
        me.registerNode(node);
        me.fireEvent('append', null, node);
        me.fireEvent('rootchange', node);
    }
```

```
        return node;
    },
```

先将 root 属性指向根节点。如果 beforeappend 事件没有中止代码执行，则设置节点的 root 属性为 true，表示这为根节点。

接着调用 updateInfo 方法更新节点的基本数据，如 isFirst、isLast 或 depth 等信息，并调用 commit 方法确认这些修改。

接着为节点的 insert、append 和 remove 事件绑定方法，当发生这些操作时，就要在 TreeData 对象中注册或取消注册根节点，也就是在 nodeHash 添加或删除这些节点。

定义传播事件后，创建 nodeHash 对象来保存节点。最后注册根节点，并触发 append 和 rootchange 事件。

接下来返回 TreeStore 的 setRootNode 方法，如果根节点还没加载数据且是展开的，调用 load 方法加载数据。

至此，TreeStore 的初始化就完成了，主要工作包括创建模型、为节点创建容器、配置根节点、根据需要为服务器加载数据等。

11.1.4 TreeColumn 的运行流程：Ext.tree.Column

TreeColumn 对象派生于 Column 对象，与 NumberColumn 这些对象一样，是通过定义配置项 renderer 来渲染数据的。

TreeColumn 对象在渲染数据时，会先根据当前节点的显示字段的配置，将生成的 HTML 代码片段先保存到数据 buf 中，然后逐层向上遍历节点的父节点，每找到一层父节点就在 buf 数组前插入一个空白图片，图片的高度和宽度是固定的，这样就让节点在显示时有了层次感。最后将 buf 数组生成的字符串插入到单元格中，就完成了节点的显示。

TreeColumn 对象的源代码在此就不赘述了，有兴趣可以自己研究一下。

11.1.5 视图的运行流程：Ext.tree.View 与 Ext.data.NodeStore

TreeView 对象派生于 TableView 对象，其最特别的地方是它会使用 NodeStore 管理视图内的节点，原因是视图要对处理节点进行选择，而显示在视图中的节点不像 Grid 那样，会把 Store 中的数据都渲染出来，显示的只是可见节点，因此为了能对节点进行选择，视图中的元素必须与 Store 中的数据对应，这就需要 NodeStore 对象了。

NodeStore 对象派生于 Store 对象，因此它也是使用 MixedCollection 对象管理节点数据的。**因而，要获取当前树中的可见节点，可通过视图的 Store 中 data 属性访问。**在 NodeStore 对象中，属性 node 会指向根节点，也就是说，通过它可遍历树中的所有节点。

视图的运作流程主要就是创建 NodeStore 对象实例，并设置根节点，在此就不赘述了，有兴趣自己可以看看源代码。

11.1.6　树的选择模型：Ext.selection.TreeModel

TreeModel 对象派生于 RowModel 对象，因此它也是通过行进行选择的，只是在行选择的基础上添加了使用左右箭头展开或折叠节点的方法，以及使用空格或回车键选择节点的方法。

11.2　树的使用

11.2.1　一个最简单的树

对于最简单树，只需要为其定义配置项 root 就行了，而对于其余的部件，树都会自动生成，例如以下代码：

```
Ext.create("Ext.tree.Panel",{
    width:100,
    height:300,
    renderTo:Ext.getBody(),
    root:{
            text:"图片",
            children:[{text:"风景"},{text:"人物"},{text:"建筑"}]
    }
})
```

图 11-1　最简单的树的示例效果

打开模板页，在命令行中运行以上代码，并展开图片，将看到如图 11-1 所示的变化。

11.2.2　树节点的默认字段

在没有添加任何附加字段的情况下，树节点默认有 21 个字段，有些字段会提交到服务器，有些则不会。要想使用好树，必须了解这些字段的作用，它们的详细说明如表 11-1 所示。

表 11-1　树节点默认字段的详细说明

字段名称	树节点	默认值	提交	说明
id	string	null	是	该字段名称会根据模型的 idProperty 变化，默认值是 id
parentId	string	null	是	父节点的 id
index	int	null	是	节点在同层子节点中的索引值，例如在上一节示例中，图片节点的索引是 0，风景的索引是 0，人物的索引是 1，建筑的索引是 2。如果与图片同层的还有节点视频，那么视频的索引会是 1，如果视频下依次有电视剧与电影两个子节点，那么电视剧的索引是 0，电影的索引是 1。简单来说，就是在同一父节点下，子节点在 childNodes 数组中的索引
depth	int	0	是	节点的深度
expanded	bool	false	否	值为 true，表示节点是展开的；值为 false，则节点是折叠的
expandable	bool	true	否	值为 true，表示节点可展开或折叠；值为 false，则没有折叠与展开功能
checked	auto	null	是	设置该值为 true 或 false 可显示复选框。使用 set 方法设置该值为 true 时，复选框会被选中，设置该值为 false 时，则会取消选择。反过来，复选框被选中，该值会是 true，否则，其值为 false

（续）

字段名称	树节点	默认值	提交	说明
leaf	bool	false	否	值为 true，表示该节点是叶子节点；值为 false，则不是。该字段会影响节点的图标显示
cls	string	null	否	应用于节点的样式类
iconCls	string	null	否	应用于节点图片的样式类
root	bool	false	否	值为 true，表示该节点是根节点；值为 false，则不是根节点
isLast	bool	false	否	值为 true，表示该节点是其父节点的最后一个子节点
isFirst	bool	false	否	值为 true，表示该节点是其父节点的第一个子节点
allowDrop	bool	true	否	值为 true，表示该节点允许接收拖动的节点
allowDrag	bool	true	否	值为 true，表示该节点允许拖动
loaded	bool	false	否	值为 true，表示该节点已加载过数据
loading	bool	false	否	值为 true，表示该节点正在加载数据
herf	string	null	否	该值会为节点文本添加一个链接，链接的地址为其值
hrefTarget	string	null	否	该值会应用到链接的 target 属性
qtip	string	null	否	节点的提示信息
qtitle	string	null	否	提示信息的标题

11.2.3 为树节点添加附加字段

如果默认的字段不能满足需求，则可以添加自定义的字段，添加方法有两种：第一种是在定义树的配置对象时，在配置项 fields 中定义；第二种是定义一个模型，然后在树的配置对象中定义 model 配置项，值为该模型。**无论使用那种方法，字段 text 都必须添加到配置中，不然节点就没法显示文本了，除非定义了配置项 displayField 指定显示文本在哪一个字段。**

下面演示一下如何使用这两种方法添加附加字段，假如要在 11.2.1 节的示例中加入“url”这个附加字段，使用第一种方法，需要在树的配置对象中添加 fields 配置项。

```
fields:["text","url"],
```

在代码中，text 字段是必需的，url 是要附加的字段。

接着修改一下 root 配置对象，代码如下：

```
root:{
    text:"图片",
    url:"/photo",
    expanded:true,
    children:[
            {text:"风景",url:"/Landscape"},
            {text:"人物",url:"/people"},
            {text:"建筑",url:"/building"}
    ]
},
```

为了检验记录中是否已添加附加字段，监听 itemclick 事件，在单击节点后显示 url 的

值，代码如下：

```
listeners:{
   itemclick:function(view,rec){
           console.log(rec.data.url);
   }
}
```

运行后单击“风景”节点，将看到控制台输出的内容如下：

```
/Landscape
```

这说明记录中已经有 url 字段了。

如果使用模型添加附加字段，就会复杂点，需要先定义模型。

```
Ext.define("TreeTest",{
    extend:"Ext.data.Model",
   fields:["text","url"]
})
```

然后在树中添加 model 配置项。

```
model:"TreeTest"
```

运行后，单击“风景”节点，控制台中的输出也会是“/Landscape”。

使用配置项 fields 与定义模型这两个方法的主要区别是：使用 fields 配置项，树会自己创建一个模型；而使用定义模型的方法，树只是使用了已定义好的模型而已。

11.2.4　显示多列数据（TreeGrid 效果）

树现在就是以 Grid 形式显示的，因此要实现 TreeGrid 的效果太容易了，只要为树定义 columns 配置项就行了，前提是第一列要用 TreeColumn 对象。例如要将 11.2.5 节示例中的 url 显示出来，可定义 columns 配置项如下：

```
columns:[
   {xtype:"treecolumn",dataIndex:"text",text:"Text"},
   {dataIndex:"url",text:"URL"}
],
```

运行后，会看到如图 11-2 所示的效果。

图 11-2　TreeGrid 的显示效果

11.2.5　在树中使用复选框

如果要在树中使用复选框，只要将树节点的字段 checked 设置为 true 或 false 就可以了，下面通过一个示例来演示一下。

（1）示例功能

实现通过复选框选择或取消选择根节点后，选择或取消选择全部可选的子节点。

（2）实现代码

要实现全部选择或取消选择全部子节点，前提是要知道当前哪个节点改变了复选框的状

态，树的 checkchange 事件会在节点的复选框状态改变后触发，因此它是合适的监听事件。在事件中，通过节点的 childNodes 就可访问其子节点，而通过递归就可遍历所有的子节点。

目标明确后，开始工作，使用模板页创建一个名称为 11-1.html 的页面文件，然后加入以下代码：

```
Ext.create("Ext.tree.Panel",{
    width:200,
    height:300,
    renderTo:Ext.getBody(),
root:{
        text:"根节点",
        url:"/photo",
        expanded:true,
        children:[
            {text:"节点 1",checked:true,expanded:true,
                children:[
                {text:"节点 1-1",checked:true,leaf:true},
                {text:"节点 1-2",leaf:true},
                {text:"节点 1-3",checked:false,leaf:true}
                ]
            },
            {text:"节点 2",expanded:true,
                children:[
                {text:"节点 2-1",checked:true,leaf:true},
                {text:"节点 2-2",leaf:true},
                {text:"节点 2-3",checked:false,leaf:true}
                ]
            },
            {text:"节点 3",checked:false,expanded:false,
                children:[
                {text:"节点 3-1",checked:true,leaf:true},
                {text:"节点 3-2",leaf:true},
                {text:"节点 3-3",checked:false,leaf:true}
                ]
            }
        ]
},
listeners:{
        checkchange:function(node,checked){
            if(node.childNodes.length>0)
                this.changeChecked(node.childNodes,checked)
        }
},
changeChecked:function(node,checked){
        if(Ext.isArray(node)){
        for(var i=node.length-1;i>=0;i--){
            this.changeChecked(node[i],checked);
        }
    }else{
        if(node.data.checked!=null){
            node.set("checked",checked);
        }
        if(node.childNodes.length>0)
            this.changeChecked(node.childNodes,checked);
```

```
            }
        }
    })
```

在示例中，定义了 3 组子节点。注意，节点中没有定义 checked 属性的节点，这些节点在显示时将不会显示复选框。在 checkchange 事件中，如果改变了 checked 值的节点有子节点，则调用 changeChecked 方法改变子节点的 checked 值。在 changeChecked 方法中，如果传递过来的参数 node 是数组，则递归调用 changeChecked 方法。如果不是，则要判断节点的 checked 值是否为 null，为 null 说明该节点不允许通过复选框设置 checked 值，因而不应该修改其 checked 值，若 checked 值不是 null，则使用 set 方法根据父节点的 checked 修改其 checked 值。这里一定要用 set 方法设置值，不然可见的节点会更换复选框的状态。如果当前节点有子节点，则递归调用 changeChecked 方法。

（3）页面效果

在浏览器中打开页面，并取消选择“节点 1”，选择“节点 3”，然后展开此节点，将看到如图 11-3 所示的变化。

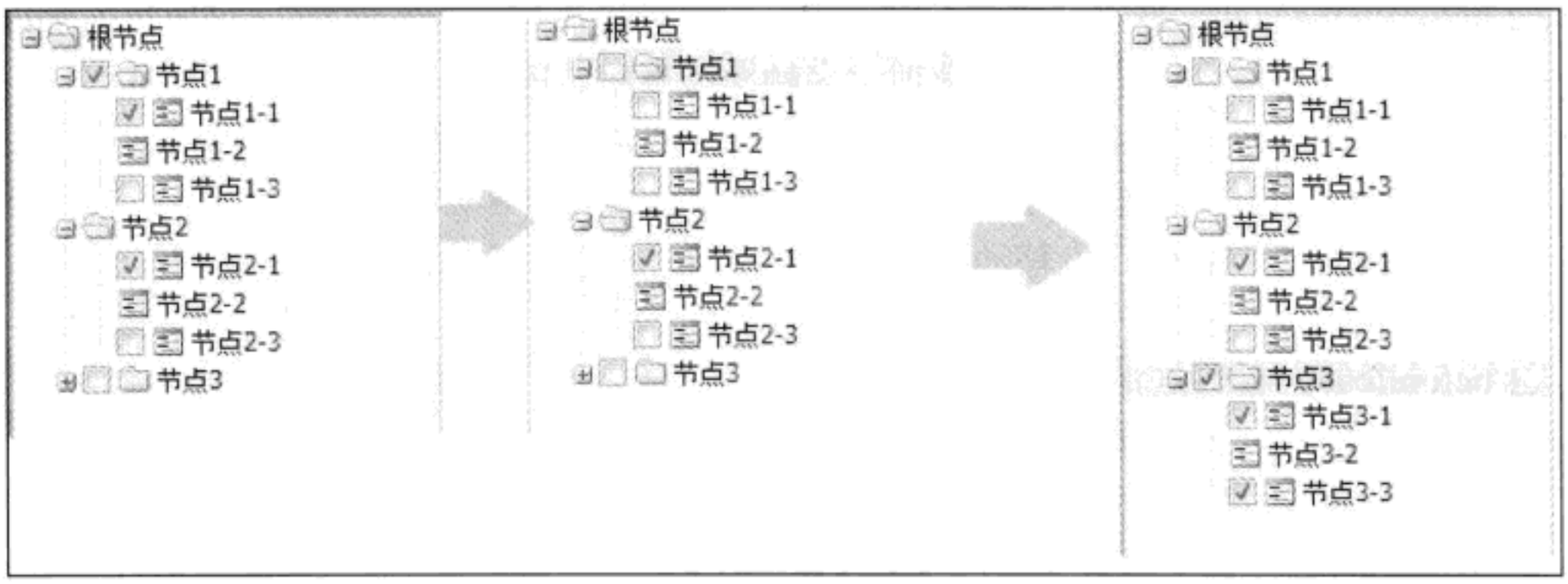


图 11-3　在树中使用复选框的显示效果

11.2.6　树的配置项、属性、方法和事件

1. 配置项

- animate：布尔值，值为 true，在折叠或展开节点时会使用动画过渡。
- displayField：指定节点的显示文本所在字段，默认值为 text。
- floderSort：布尔值，值为 true 时，会自动对叶节点进行排序。默认值为 undefined。
- hideHeaders：布尔值，值为 true 时，会隐藏列标题，默认值为 undefined。
- lines：布尔值，默认值为 true，显示树的层次线。
- root：配置根节点。
- rootVisible：布尔值，默认值为 true，显示根节点。
- singleExpand ：布尔值，值为 true，在每个分支下，只允许一个节点是展开的，类似

Accordion 布局的显示，每次只显示一个面板，而这里是每次只展开一个节点。默认值为 false，可展开多个节点。

- useArrows：布尔值，值为 true，使用 Vista 风格的箭头作为树的展开或折叠图标。默认值为 false。

2. 属性

树没有属性。

3. 方法

- coll　apseAll：折叠所有节点。
- expandAll：展开所有节点。
- expandPath：根据指定路径展开节点。
- getChecked：以数组形式返回所有 checked 值为 true 的节点。
- selectPath：展开指定路径的节点并选择它们。

4. 事件

- beforeitemappend：在新节点追加到节点前，会触发该事件，返回 false 可取消追加操作。
- beforeitemcollapse：在节点折叠前，会触发该事件，返回 false 可取消折叠操作。
- beforeitemexpand：在节点展开前，会触发该事件，返回 false 可取消展开操作。
- beforeiteminsert：在新节点插入到节点前，会触发该事件，返回 false 可取消插入操作。
- beforeitemmove：在节点移动前，会触发该事件，返回 false 可取消移动操作。
- beforeitemremove：在节点被删除前，会触发该事件，返回 false 可取消删除操作。
- beforeload：在节点加载子节点前，会触发该事件，返回 false 可取消加载操作。
- checkchange：当节点复选框的状态改变时，会触发该事件。
- itemappend：在追加节点操作完成后，会触发该事件。
- itemcollapse：在节点折叠后，会触发该事件。
- itemexpand：在节点展开后，会触发该事件。
- iteminsert：在节点插入操作完成后，会触发该事件。
- itemmove：在节点移动后，会触发该事件。
- itemremove：在节点被删除后，会触发该事件。
- load：在节点加载子节点后，会触发该事件。

11.3 综合实例

11.3.1 树的远程加载

（1）示例功能

本示例主要实现远程加载 JSON 格式的数据，并默认选择第一个节点。数据如下（Tree.js）：

```
[
    {id:"1",text:" 节点 1",expanded:true,"children":[
            {id:"4",text:" 节点 1-1",children:[
                    {id:"13",text:" 节点 1-1-1",leaf:true},
                    {id:"14",text:" 节点 1-1-2",leaf:true},
                    {id:"15",text:" 节点 1-1-3",leaf:true}
            ]},
            {id:"5",text:" 节点 1-2",children:[
                    {id:"7",text:" 节点 2-1",leaf:true},
                    {id:"8",text:" 节点 2-2",leaf:true},
                    {id:"9",text:" 节点 2-3",leaf:true}
                ]},
                {id:"6",text:" 节点 1-3",children:[
                        {id:"10",text:" 节点 3-1",leaf:true},
                        {id:"11",text:" 节点 3-2",leaf:true},
                        {id:"12",text:" 节点 3-3",leaf:true}
                ]}
    ]},
    {id:"16",text:" 节点 2"}
]
```

（2）实现代码

要实现远程加载，为 TreeStore 对象配置好 Proxy 即可，只要是 url 配置项，不需要像 Grid 那样包含 success 和记录总数这些属性，并且只要以 JSON 数组形式返回数据就可以了，同时注意子节点要放在 children 属性下。

使用模板页创建一个名称为 11-2.html 的页面文件，先定义 TreeStore：

```
Ext.create("Ext.data.TreeStore",{
    id:'treeStore',
     proxy: {
         type: 'ajax',
         url : 'Tree.js',
     }
});
```

然后定义树：

```
Ext.create("Ext.tree.Panel",{
    width:200,
    height:300,
    renderTo:Ext.getBody(),
    store:"treeStore",
    rootVisible:false,
    viewConfig:{
            listeners:{
                refresh:function(){
                        this.select(0);
                }
            }
    }
})
```

默认选择第一行，这与 Grid 是一样的，因为它们是同源的，所以选择操作一样。

就这么简单，页面就完成了。

（3）页面效果

在浏览器中打开页面将看到如图 11-4 所示的效果，说明数据已经加载了。单击“节点 2”，会在控制台发送一个请求，然后出现错误信息。这是因为设置了 Proxy 的树，大家会认为其是动态加载的树，因此当节点不指明 leaf 为 true 时，会认为其有子节点，就会发送加载数据的请求，但是 tree.js 并不会处理请求并返回原来的数据，这就造成了因 id 冲突而出现错误。因此，对于静态远程一次性加载的树，如果是叶节点的，一定要设置 leaf 为 true，或者修改为动态页面文件，然后返回空数组。

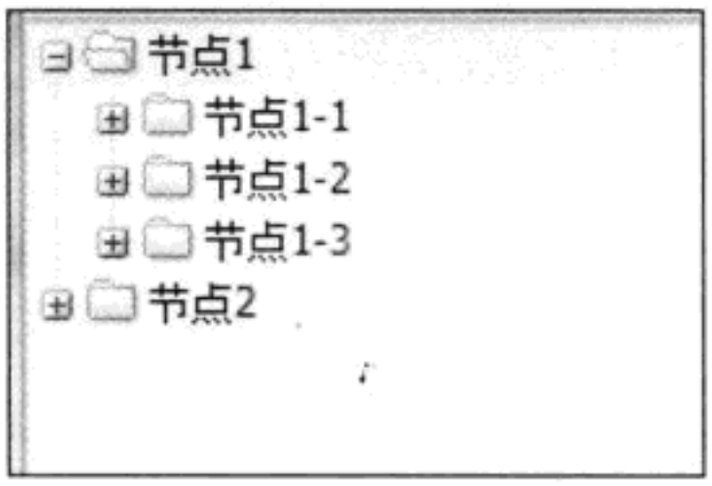


图 11-4　树的远程加载的显示效果

11.3.2　树的动态加载及节点维护

（1）示例功能

虽然上一节的示例已经实现了动态加载，但是并没有在服务器端实现动态加载，因此本节将演示如何在服务器端通过树实现动态加载，添加节点、删除节点、修改节点，以及通过拖动移动节点等功能。

因为 Northwind 库没有合适的表，所以不得不在实现代码前，在 Northwind 库中建立一个 TreeTest 表，表结构的详细说明如表 11-2 所示。

表 11-2　TreeTest 表的结构说明

字段名称	类型	默认值	运行空	说明
ID	int		否	自增主键
Text	nvarchar(100)		否	显示文本
ParentID	int	0	是	父节点的 ID

在 TreeTest 表中插入如表 11-3 所示的数据。

表 11-3　在 TreeTest 表中插入的数据

ID	Text	ParentNode
1	图片	0
2	文档	0
3	视频	0
4	电视剧	3
5	报告	2
7	电影	3
8	活动	1
9	报表	2
10	旅游	1

（2）实现代码

树节点的编辑与 10.7.5 节使用 Cellediting 编辑 Grid 类似，不过在有些地方需要小心，不然会很费劲。先用模板页创建名称为 11-3.html 的页面文件。

在定义模型前，要清楚不能像 Grid 那样添加节点了，并且也不能将数据全部集中起来才保存，原因是用户在一个没有 id 的节点下添加了 n 层节点，然后将数据提交到服务器，这样提交的节点的 id 都是 0，服务器根据 parentId 根本就无法分清楚哪个节点是哪个节点的子节点，因此每新增一个节点，都必须第一时间到服务器获取实际 id，返回后再将节点添加到树。新增节点后，可使用模型的 save 方法保存数据，这样的好处是可以监控节点的提交操作是否成功，如果成功，则添加到树，不成功则显示错误信息。而使用 Store 的 sync 方法保存数据，需要先将记录添加到 Store 才能使用，这不是好的选择。要调用模型的 save 方法，需要为模型配置 Proxy，这样就不需要在 Store 中配置 Proxy 了。目标明确，现在可以定义模型了。

```
Ext.define("TreeTest",{
    extend:"Ext.data.Model",
    fields:["text",
            {name:"id",type:"int"},
            {name:"parentId",type:"int"}
    ],
     validations: [{
         type: 'presence',
         field: 'text'
     }],
     proxy: {
            type: 'ajax',
            api:{
               read:'Tree.ashx?act=read',
               create:'Tree.ashx?act=add',
               destroy:'Tree.ashx?act=del',
               update:'Tree.ashx?act=edit'
            },
            reader:{
               messageProperty:"Msg"
            },
            writer:{
               type:"json",
               encode:true,
               root:"data",
               allowSingle:false
            }
        }
})
```

定义的字段最后要与模型的默认字段保持一致，尤其是 parentId，这样会省事很多。节点的移动会自动修改 parentId 值，只需要保存节点就行了，不需要自己处理。配置对象 reader 中的配置项 messageProperty 的作用是当服务器端返回的对象中的 success 为 false 时，提取错误信息的属性名称。配置对象 writer 还是习惯做法，使用 data 提取数据。

下面定义 TreeStore：

```
Ext.create("Ext.data.TreeStore",{
    id:'treeStore',
    model:"TreeTest",
    root:{text:" 目录 ",id:-1,expanded:true}
});
```

Proxy 都在模型中定义了，因此在 TreeStore 中没多少配置项。这里要注意，根节点的 id 不能设置为 0，因为如果是 0，Store 会认为此节点是新节点，调用 sync 方法时候会将此节点提交，由于 id 是整型数据，id 也不能定义为“root”这样的字符串值，最好的办法是将 id 定义为 –1，这样不可能在数据库中出现的值。不过在写服务器端代码的时候要注意，当 parentId 为 –1 时，要将其修改为 0。在这里没把根节点隐藏是考虑到要在根节点下添加节点，除非多添加一个增加按钮来添加根节点下的节点，不然很难判断到底是在哪个位置添加节点，也可以通过弹出对话框来解决这个问题，不过笔者觉得前一种方式更简便，只要禁止用户编辑根节点就行了。

下面定义 TreePanel：

```
Ext.create("Ext.tree.Panel",{
    title:" 树的动态加载及节点维护 ",
    width:200,
    height:300,
    hideHeaders:true,
    plugins:[{ptype:"cellediting",
            listeners:{
                    beforeedit:function(e){
                            if(e.record.isRoot()) return false;
                    }
            }
    }],
    renderTo:Ext.getBody(),
    store:"treeStore",
    tbar:[
            {text:" 增加 ",id:"add",handler:function(){

            }},
            {text:" 删除 ",id:"delete",disabled:true,handler:function(){
            }},
            "|",
            {text:" 刷新 ",handler:function(){
                    this.up("treepanel").store.load();
            }}
    ],
    columns:[
            {xtype:"treecolumn",dataIndex:"text",flex:1,
                    field:{allowBlank:false}
            }
    ],
    viewConfig:{
            toggleOnDblClick:false,
        plugins: {
            ptype: 'treeviewdragdrop'
        },
```

```
        listeners:{
                refresh:function(){
                        this.select(0);
                        this.focus(0);
                }
        }
},
listeners:{
        selectionchange:function(view,rs){
                Ext.getCmp("delete").setDisabled(rs.length==0);
        }
})
```

树与 Grid 同源的好处是可以使用 CellEditing 插件编辑节点，而且它们的定义也基本一样。不过要注意的问题也很多。首先，因为定义了配置项 columns，所以会显示列标题，需要定义 hideHeaders 配置项将其隐藏。配置项 columns 的定义是必须的，不然无法加入编辑框，而且要定义配置项 flex 为 1，使整个面板的宽度作为列宽度。为了禁止编辑根节点，需要在 Cellediting 的配置对象中监听 beforeedit 事件，如果要编辑的节点是根节点，那么直接返回 false 可中止进入编辑状态。在视图的配置对象中，toggleOnDblClick 为 false 这个配置项很关键，该配置项默认为 true，因此在默认情况下，双击节点会展开或折叠节点，这就要求必须使用单击进入编辑状态，但这样做会很麻烦，例如要选择某个节点，单击使其处于编辑状态后，只能退出后再选择好位置单击才能选中节点。所以，最好是将该值设置为 false，留出双击操作给编辑操作，这样就方便多了。要允许通过拖动改变节点位置，只要加入 TreeViewDragDrop 插件就可以了，通过该插件可以在视图渲染时设置视图的 dragZone 和 dropZone 配置项，从而在视图中实现拖放操作。事件 refresh 的代码的作用应该清楚了，它会设置默认选择第一个节点，并将焦点移动到视图，然后通过键盘导航。

现在考虑增加按钮的操作，这也不难，先通过选择模型获取父节点，然后获取其 id 作为新节点的 parentId 值，关键是调用 save 方法后的处理，如果保存成功，则要看父节点的状态是不是展开状态，若不是，则要展开父节点，让它去下载数据，这样自然会把新节点也加载进来，如果父节点已经展开，则直接调用 appendChild 方法将新节点追加到父节点下。如果发生错误，则显示错误信息。思路想好后，编写代码也就容易了。

```
var tree=this.up("treepanel"),
    parent=tree.getSelectionModel().getSelection()[0];
if(! parent){
    parent=tree.store.tree.root;
}
var rec=new TreeTest({
    text:"新节点",
    id:"",
    parentId:parent.data.id
});
rec.save({
    parentNode:parent,
    success:function(rec,opt){
            if(opt.parentNode.isExpanded())
```

```
                    opt.parentNode.appendChild(rec);
                else
                    opt.parentNode.expand();
        },
        failure:function(e,op){
                Ext.Msg.alert("发生错误",op.error);
        },
        scope:tree
    });
```

因为要先向服务器添加数据后才能追加节点，所以新节点文本只能先固定为“新节点”，然后再由用户通过编辑的方法修改，如果使用弹出对话框，让用户先输入节点文本，再保存，那么最简单的办法就是使用 MessageBox 对象的 prompt 方法，这个可根据项目需求确定使用哪个办法。这里没有像 Grid 那样，插入后记录后在记录位置显示编辑框，主要原因是，查找记录的行位置的工作量太大，有兴趣的读者可自己研究一下。当服务器端返回错误信息的时候，会将 Msg 属性中的信息记录到 Operation 对象的 error 属性中，因此直接将其显示出来就行了。

下面考虑编辑的问题，这里没有按钮操作，主要考虑如何保存修改过的数据，简单的办法是和 Grid 一样，通过一个保存按钮来保存数据，不过在树中的单元格左上角会显示一堆的红色小三角，这不太好，还是编辑一个保存一个比较好。最大的问题来了，TreeStore 的 autoSync 配置项居然不起作用，所以只能通过代码来实现保存了。Celleditting 对象在编辑完成后，都会触发 edit 事件，而这正是最好的保存时机，因而，现在要为 Celleditting 对象的配置对象添加一个 edit 事件。

```
edit:function(edit,e){
    e.record.save({
            success:function(rec,opt){
                opt.records[0].commit();
            },
            failure:function(e,op){
                op.records[0].reject();
                Ext.Msg.alert("发生错误",op.error);
            }
    });
}
```

如果保存成功，那就使用 commit 方法确认修改，不然会使用 reject 方法恢复原来的值，无论使用哪种方法都可把编辑的小图标去掉了。

删除操作也是一个比较考验人的操作，例如，要删除一个节点，并且节点的层次很深，服务器端代码就要做很多次递归操作，效率会很低。要想不进行递归删除，办法有 3 个：第一个办法是用户自己先确保该节点没有子节点，才允许将其删除，这办法比较笨，但笔者认为比较实用；第二个办法是确保节点的层数在有限的范围内；第三个办法是添加附加字段，不过这也要确保节点在有限层内才行。最好的办法还是不要建立无限层的树，用户当然是希望能建立无限层的树，但是就像文件目录一样，超过 10 层就会让人头痛一样，无限层的树会很麻烦。本示例采用的是第一种方法，代码如下：

```
var tree=this.up("treepanel");
var rs=tree.getSelectionModel().getSelection();
if(rs.length>0){
    rs=rs[0];
    if(rs.data.root){
            Ext.Msg.alert(" 删除节点 "," 根节点不允许删除！ ");
            return;
    }
    if(rs.isExpandable() || rs.hasChildNodes()){
            Ext.Msg.alert(" 删除节点 "," 请先删除所有子节点，再删除该节点！ ");
            return;
    }else{
            var content=" 确定删除节点：  "+rs.data.text+" ？ ";
            Ext.Msg.confirm(" 删除节点 ",content,function(btn){
                    if(btn=="yes"){
                            var rs=this.getSelectionModel().getSelection();
                            if(rs.length>0){
                                    rs=rs[0];
                                    rs.remove();
                                    this.store.sync();
                                    this.view.select(0);
                                    this.view.focus(false);
                            }
                    }
            },tree)
    }
}
```

根节点是不能删除的，这个不用说。节点是折叠状态的，或者有子节点的，都不允许删除，直接返回。如果节点可以删除，会提示用户是否真的删除，如果选择是，就会调用 remove 方法删除节点，并立刻进行同步。

对于移动节点这个操作，比较难的是如何处理数据的保存，如果在 TreeStore 对象的 beforemove 事件中先保存数据，则要先挂起代码，然后调用同步方法，同步后的事件再根据成功与否返回 true 或者 false，确认是否移动，这个过程比较复杂。简单的办法是先允许移动，如果保存不成功，就刷新显示，如果成功，则什么也不用做，这样就很简单了，代码如下：

```
move:function(tree,node){
    var me=this;
    node.save({
            failure:function(e,op){
            Ext.Msg.alert(" 发生错误 "," 保存移动时发生错误，现在要刷新树！ <br/>"
                                    +" 错误原因：  "+op.error,function(){
                            this.load();
                    },me);
            },
            scope:me
    });
}
```

在这里回调的 success 方法也省了，只需要 failure 方法，在发生错误时，等用户关闭提示信息窗口后，调用 load 方法刷新根节点就好了。

现在开始编写服务器端代码，按照之前的方法，先根据参数 act 调用不同的方法。先处理 Read 方法，因为动态树是根据节点加载数据的，所以会将节点 id 提交到服务器端，默认提交参数是 node，这样在服务器端可通过 node 获取父节点的 id。如果不想将 node 作为提交参数，可在 TreeStore 的配置对象中定义 nodeParam 配置项来改变提交参数，例如定义了 nodeParam 为 id，服务器端要通过 id 获取父节点的 id。Read 方法的具体代码不难，读者可自己阅读相关文件，在此就不列了。

在完成 Add、Edit 和 Delete 方法前，与 Grid 一样，需要先定义一个对象，然后将 JSON 数据转换为对象进行操作，代码如下：

C#

```
public class Node
{
    public int id { set; get; }
    public string text { set; get; }
    public int parentId { set; get; }
    public int index { set; get; }
    public int depth { set; get; }
}
```

Java

```
public class Node {
    int id,parentId,index,depth;
    String text;
    Boolean checked;
}
```

因为模型的运作要求提交的数据字段与返回的数据字段一样才会确认数据更新，所以对象的定义最好与提交的数据字段保持一致，但是对于树的字段，因为有附加字段，所以定义其不能与数据库的字段一致。不同的编程语言会有不同的麻烦，在 C# 中，checked 是关键字，不能作为类的属性，返回的数据还得重新组合一次。还好，这里的 checked 值都是 null，容易处理，如果提交上来的 checked 值是 true、false，或者不确定的值，那麻烦就大了，只能在客户端监测到 checked 值变化后，修改一个自定义字段的值，然后在服务器端，根据该提交值在返回数据时再造 checked 的值了。在 Java 中，boolean 不能为 null，而 Boolean 可以，因此问题不大。

接下来定义一个数据转换方法 ProcessData，代码如下：

C#

```
private List<Node> ProcessData(string data)
{
    List<Node> nodes = new List<Node>();
    nodes = JsonConvert.DeserializeObject<List<Node>>(data);
    return nodes;
}
```

Java

```
protected Node[] ProcessData(String data){
    Gson gson = new Gson();
    Node[] nodes =gson.fromJson(data, Node[].class);
    return nodes;
}
```

这个问题不大，在第 10 章已经实现过类似代码。如果是在项目中，最好先写一个通用接口，然后调用，这样只要定义好类就可以调用该接口了，不用为每个处理都写一个 ProcessData 方法。

Add、Edit 和 Del 方法，与 10.7.5 节的内容差别不大，在此就不详细说明了。Add 方法的重点是获取将数据插入数据库后自增的 id 值，然后再返回，这个数据库开发人员应该很熟悉了。在这 3 个处理方法中，要共同注意的是，如果 parentId 值是 –1，在将数据保存到数据库时，要将 ParentId 值转换为 0，对于 Read 方法，则需要将 ParentId 值从 0 转换为 –1。这是笔者的疏忽，直接在 ParentId 字段以值 -1 表示根节点下的节点就不用这么麻烦了。如果你项目是使用相同的数据结构，可考虑这样使用。

还要注意，对于 C# 的 Add 和 Edit 方法，要重新构造 JSON 对象，把 checked 属性加入到返回的数据中。而 Java 中，因为 GSon 对象会忽略掉值为 null 的数据，所以需要使用以下语句定义 Gson 对象：

```
Gson gson = new GsonBuilder().serializeNulls().create();
```

至此，示例就完成了。

（3）页面效果

在浏览器中打开页面，单击“增加”，然后将“新节点”修改为“音乐”，最后将其拖动到文档下，将看到如图 11-5 所示的效果。

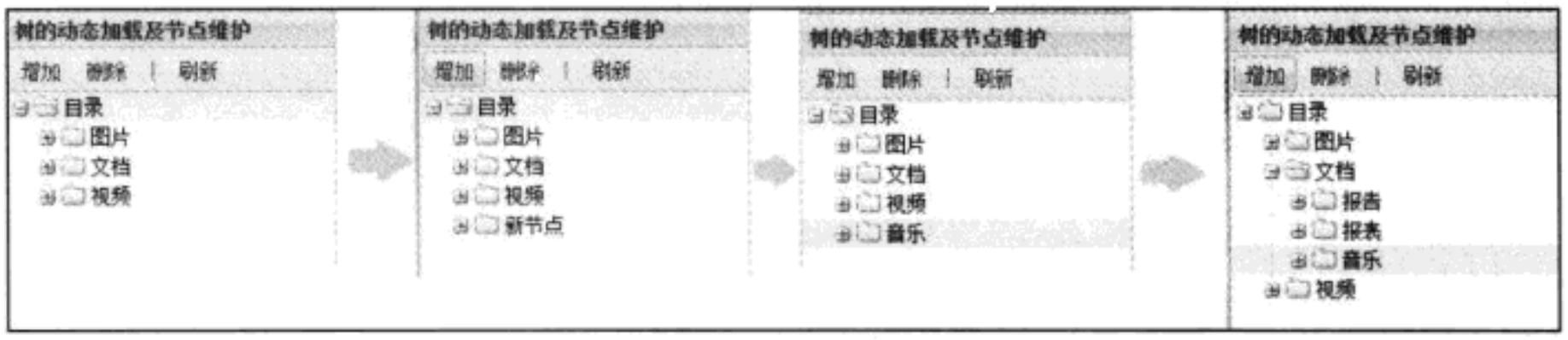


图 11-5　示例的页面效果

11.3.3　XML 树及节点维护

（1）示例功能

本示例的功能同 11.3.2 节。

（2）代码实现

将 XML 文档作为树的数据源不难，目前最大问题是不能像 11.3.1 节的示例那样一次性把树全部加载进来，只能像 11.3.2 节的示例那样根据父节点加载数据。页面文件要修改的只有 Reader 和 Writer 的配置项对象，非常简单。

首先将 11.3.2 节的示例复制一份，然后将文件名修改为 11-4.html，先修改模型的 Reader 配置项对象，让它以 XML 格式加载数据。

```
type: 'xml',
root:"data",
record:"node"
```

在代码中定义了读取的数据类型是 XML，配置项 root 定义了读取数据的节点是从 data 开始的，也就是说，无论是错误信息，还是数据，都必须包含在 data 节点内。而 record 配置项则定义了读取每一条记录的节点。

最后修改 Writer 的配置对象，让它以 XML 格式提交数据。

```
type:"xml"
```

这里只配置了提交数据的格式是 XML 格式，没有做其他修改，因而会使用默认的数据格式提交数据，默认格式如下：

```
<xmlData> //Root 节点，可通过配置项 documentRoot 更改
<record>  // 记录节点，可通过配置项 record 更改
// 每条记录的字段及值
</record>
</xmlData>
```

这样页面的修改就完成，这充分展示了 Ext JS 的魅力，数据与显示无关，要转换数据格式非常的简单，当然，服务器端代码并不是那么简单。

与 JSON 格式一样，在服务器端先把 XML 转换为对象会便于操作，不过将 XML 转换为对象麻烦点，需要定义两个类，一个是节点类，另一个是节点类的集合类，下面是它们的定义代码：

C#

```
[Serializable()]
public class Node
{
    [System.Xml.Serialization.XmlElement("id")]
    public int id { set; get; }
    [System.Xml.Serialization.XmlElement("text")]
    public string text { set; get; }
    [System.Xml.Serialization.XmlElement("parentId")]
    public int parentId { set; get; }
    [System.Xml.Serialization.XmlElement("index")]
    public int index { set; get; }
    [System.Xml.Serialization.XmlElement("depth")]
    public int depth { set; get; }
    [System.Xml.Serialization.XmlElement("checked")]
    public string mychecked { set; get; }
}

[XmlRootAttribute("xmlData")]
public class NodeList
{
    [XmlElement("record")]
```

```
    public Node[] nodes { get; set; }
}
```

Java

```
public class Node {
    int id,parentId,index,depth;
    String text;
    String checked;
}
```

```
public class NodeList {

   private ArrayList nodeList = new ArrayList();

    public ArrayList getNodeList() {
        return this.nodeList;
    }
    public void setNodeList(ArrayList persons) {
        this.nodeList = nodeList;
    }

}
```

C# 可通过属性来设置读取哪个节点，因而 checked 字段改变了 mychecked 属性，其实 JSON 对象也可以这样做。Java 则一如既往，直接把字段和提交的节点名称对应起来。NodeList 类在 C# 中已经与通过属性与 XML 文档对接上了，而在 Java 中还没有，因而还需要在实现代码中指定。

下面要做的是定义 ProcessData 方法，代码如下：

C#

```
private NodeList ProcessData(HttpRequest request)
{
    Stream stream = HttpContext.Current.Request.InputStream;
    StreamReader sr = new StreamReader(stream);
    string data = sr.ReadToEnd();
    StringReader reader = new StringReader(data);
    XmlSerializer serializer = new XmlSerializer(typeof(NodeList));
    return (NodeList)serializer.Deserialize(reader);
}
```

Java

```
protected NodeList ProcessData(HttpServletRequest request){
try {
    BufferedReader br = new BufferedReader(new InputStreamReader(request.
        getInputStream(), "UTF-8"));
     String line = null;
     StringBuilder sb = new StringBuilder();
     while((line = br.readLine())!=null){
         sb.append(line);
     }
      XStream xstream = new XStream();
```

```
        xstream.alias("record", Node.class);
        xstream.alias("xmlData", NodeList.class);
        xstream.addImplicitCollection(NodeList.class, "nodeList");
        return (NodeList) xstream.fromXML(sb.toString());
    } catch (Exception e) {
        // TODO: handle exception
        return null;
    }
}
```

因为数据是以 POST 方式提交的，所以必须使用数据流方式读取数据，然后通过 XML 序列化对象将数据反序列化为对象就行了。在 Java 中，要通过 alias 方法将对象与节点对应起来，然后通过 addImplicitCollection 方法声明集合类的标记，也就是在 NodeList 类中定义的集合类变量名称。

完成类和数据处理，代码编写就已完成大半了，余下的工作除了生成 XML 格式的输出格式外，其余数据库操作基本就与 11.3.2 节的示例一样了。

下面看看 Read 方法是如何组织返回数据的，代码如下：

C#

```
private string Read(HttpRequest request)
{
    string node = request.Params["node"] ?? "";
    if (node == "-1" || node == "")
    {
        node = "it.ParentID=0";
    }
    else
    {
        node = "it.ParentID=" + node + "";
    }
    using (NorthwindEntities ne = new NorthwindEntities())
    {
        var q = ne.TreeTest.Where(node).ToList() ;
        XDocument doc = new XDocument();
        XElement xe = new XElement("data",
            (from p in q
            select new XElement("node",
                    new XElement("id", p.ID.ToString()),
                    new XElement("text", p.Text.ToString()),
                    new XElement("parentId", p.ParentID.ToString())
                ))
            );
        doc.Add(xe);
        return doc.ToString() ;
    }
}
```

Java

```
protected String Read(HttpServletRequest request){
    String connectionUrl = "jdbc:sqlserver://192.168.0.254:1433;" +
    "databaseName=Northwind;;user=sa;password=abcd-1234";
```

```
Connection con = null;
Statement stmt = null;
ResultSet rs = null;
ResultSet rscount=null;

try {

      Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
      con = DriverManager.getConnection(connectionUrl);
      String id="0";
      if(request.getParameter("node")!=null){
          id=request.getParameter("node");
      }
      if(id.equalsIgnoreCase("-1")) id="0";
      stmt = con.createStatement();
      rs = stmt.executeQuery("select * from TreeTest where ParentID=" + id);

      Document document = DocumentHelper.createDocument();
      Element root = document.addElement("data");
      while (rs.next()) {
          Element node = root.addElement("node");
          node.addElement("id").addText(rs.getString("ID"));
          node.addElement("text").addText(rs.getString("Text"));
          node.addElement("parentId").addText(rs.getString("ParentID"));
      }

      rs.close();
      return document.asXML();
      }
      catch (Exception e) {
          return "<data><success>false</success><Msg>"+e.getMessage()+"</Msg></
              data>";
      }
      finally {
          if (rscount != null) try { rscount.close(); } catch(Exception e) {}
          if (rs != null) try { rs.close(); } catch(Exception e) {}
          if (stmt != null) try { stmt.close(); } catch(Exception e) {}
          if (con != null) try { con.close(); } catch(Exception e) {}
      }

}
```

由粗体代码可以看出，要组织 XML 文档，先根据 Reader 对象的配置项 root 创建根节点 data，然后在循环中，根据配置项 record 创建一个记录节点 node，再在其下添加字段和值。处理好格式，这工作就不会太难了。

Add、Edit 和 Del 方法在这里就不赘述了，请读者自己阅读相关文件。

这一节的示例只是在服务器处理的数据格式上不同而已，页面的操作与 11.3.2 节的示例是相同的，这里就不赘述了。

11.3.4 使用树动态控制 Grid 的显示

（1）示例功能

在应用中，常见的布局是在左边显示一个用于分类的树，而在右边显示数据的 Grid，这时单击树节点，根据树节点刷新 Grid 的显示。本示例将在 10.7.7 节的示例的基础上添加一个树来显示客户，然后单击客户可看到该客户的订单及订单明细。

（2）实现代码

先复制 10-9.html 的一个副本，将名字修改为 11-5.html。在开始编码前，要考虑一下使用什么方式来关联数据。如果使用主从表的方式，就会存在不能分页的问题，要一次性把订单全部列出来，这不太适合。另外要考虑的是使用 Filter 对象还是额外参数的形式进行过滤，使用 Filter 对象的缺点是，每当订单表进行过滤的时候，都要进行处理，保留 CustomerID 的过滤对象，这不太方便。所以，最简单的还是利用额外参数的方式，只有单击了客户后才替换该参数的值，在其他情况不修改，这样就可以保证树和 Grid 的数据能对应起来。

只需要树显示客户名称，因而不需要额外定义模型了，直接定义 Store 就好了。

```
Ext.create("Ext.data.TreeStore",{
    proxy: {
        type: 'ajax',
          url:"Customer.ashx?act=CustomerList"
          //url:"/Chapter11/Customer?act=CustomerList" //java
    },
    storeId:"CustomerStore"
})
```

接下来修改 OrderStore 的 read 配置项，先删除 autLoad 配置项，在选择发生时再加载数据，接着修改 read 配置项，令其从“Customer.ashx”（在 Java 中为 /Chapter 11/Customer）读取数据，参数为 Orderlist，代码如下：

```
read:'Customer.ashx?act=OrderList'
//read:"/Chapter11/Customer?act=CustomerList" //java
```

使用 Viewport 重新布局一下页面，代码如下：

```
Ext.create("Ext.Viewport",{
   layout:"border",
   padding: 5,
   items:[
          {xtype:"treepanel",
                title:"客户",
   region:"west",
   collapsible: true,
   rootVisible:false,
   store:"CustomerStore",
   width:200,
   minWidth:100,
   weight:50,
   split:true,
   viewConfig:{
```

```
            listeners:{
                    refresh:function(){
                            //this.select(0);
                    }
            }
    },
    listeners:{
            selectionchange:function(model,sels){
                    if(sels.length>0){
                      var rs=sels[0],
                      store=Ext.StoreMgr.lookup("OrderStore");
                      store.proxy.extraParams.CustomerID=rs.data.id;
                            store.load();
                    }
            }
    },
    // 省略原有的两个Grid 定义
  ]
});
```

Ext JS 4.1 版本的好处就是边框布局可以通过权重优先显示某个区域，因而不用嵌套使用边框布局了。在 West 区域定义一个使用 CustomerStore 的树，将其 weight 配置项设置为 50。

由于采用了批处理渲染，在 viewConfig 中监听 refresh 事件时默认选择第一个节点，会发生布局错误，所以示例目前只能把选择代码注释掉。在这里使用 selectionchange 事件修改 OrderStore 的 Proxy 参数的原因是，如果使用 itemclick 事件，用户使用键盘操作就没有反应了，而对于 selectionchange 事件，无论是单击操作还是键盘导航，只要选择发生改变就要相应改变 Grid 的显示。

至此，页面文件的修改就完成，下面开始编写服务器端代码。首先编写根据 act 的参数分别调用 OrderList 和 CustomerList 方法，这个不难。

接着定义 CustomerList 方法，代码如下：

C#

```
private string CustomerList(HttpRequest request)
{
    using (NorthwindEntities ne = new NorthwindEntities())
    {
        var q = ne.Customers.OrderBy(m => m.CompanyName).Select(m => new {
            m.CompanyName, m.CustomerID }).ToList();

        return new JArray(
new JObject(
    new JProperty("id", "-1"),
    new JProperty("text", "全部"),
    new JProperty("leaf", true)
),
from c in q
select new JObject(
    new JProperty("id",c.CustomerID),
    new JProperty("text",c.CompanyName),
```

```
                        new JProperty("leaf",true)
                    )
                ).ToString();
            }
        }
```

Java

```
private String CustomerList(HttpServletRequest request) {
    String connectionUrl = "jdbc:sqlserver://192.168.0.254:1433;" +
    "databaseName=Northwind;;user=sa;password=abcd-1234";
Connection con = null;
Statement stmt = null;
ResultSet rs = null;
ResultSet rscount=null;
try {

      Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
      con = DriverManager.getConnection(connectionUrl);
      stmt = con.createStatement();
      rs = stmt.executeQuery("select CustomerID,CompanyName from Customers
          order by CompanyName");

      //返回JOSN格式数据
      JsonArray array=new JsonArray();
         JsonObject objAll= new JsonObject();
         objAll.addProperty("id", "-1");
         objAll.addProperty("text", "全部");
         objAll.addProperty("leaf", true);
         array.add(objAll);
      while (rs.next()) {
         JsonObject obj= new JsonObject();
         obj.addProperty("id", rs.getString("CustomerID"));
         obj.addProperty("text", rs.getString("CompanyName"));
         obj.addProperty("leaf", true);
         array.add(obj);
      }

      rs.close();
      return array.toString();

   }
   catch (Exception e) {
       return  e.getMessage();
   }
   finally {
         if (rscount != null) try { rscount.close(); } catch(Exception e) {}
         if (rs != null) try { rs.close(); } catch(Exception e) {}
         if (stmt != null) try { stmt.close(); } catch(Exception e) {}
         if (con != null) try { con.close(); } catch(Exception e) {}
      }
}
```

在代码中添加了一个“全部的选项”，目的是可以在这里看到全部记录，为用户多提供一项选择，“全部”节点的 id 可根据数据类型自行设置，在服务器端根据提交值进行区分就

行了，例如，对于自增的整数字段，可以用负数代替。余下的代码只是将数据组合成 JSON 格式。

对于 OrderList 方法，在 10.7.7 节示例的基础上向其添加查询字段就可以了，这里就不再赘述了。

（3）页面效果

运行示例后，将看到如图 11-6（图为 4.0.7 版的截图）所示的效果。

图 11-6　示例的页面效果

11.4　本章小结

本章讲述了树的构成及使用方法，由于它与 Grid 是同源的，因此基本可以像 Grid 一样去使用它。不过，树的数据结构是比较特殊的。要用好树，还得先从数据结构开始，深入了解其附加字段的作用，最好要亲自测试一下每个附加字段的作用，体验一下，这样对树的理解就比较深刻了。

第12章 表　　单

表单是 Web 应用的重要做成部分，在使用时遇到的问题也是最多的，尤其是布局以及存储值的时候。Ext JS 2 和 Ext JS 3 中的 Formlayout 在 Ext JS 4 中退役了，现在，不会再因不使用 Formlayout 而没有标签了。目前作为一个功能类，通过 Mixin 功能混合到字段中。

本章将讲述表单及其组件，涉及表单及其组件的使用方法与技巧等，当然，必不可少地包括表单的布局、验证与提交。

12.1 表单的构成及操作

12.1.1 表单面板的运行流程：Ext.form.Panel 与 Ext.form.FieldAncestor

FormPanel 派生于 Panel 对象，因此它只是一个面板，一个容器，具有面板的所有功能。简单来说，FormPanel 是放置表单的容器。下面来看看 FormPanel 的运作过程，其 initComponent 方法的代码如下：

```
initComponent: function() {
    var me = this;
    if (me.frame) {
        me.border = false;
    }
    me.initFieldAncestor();
    me.callParent();
    me.relayEvents(me.form, [
            //省略事件代码
    ]);
    if (me.pollForChanges) {
        me.startPolling(me.pollInterval || 500);
    }
},
```

在代码中根据配置项 frame 设置了 border 属性后，就调用 initFieldAncestor 方法初始化 FieldAncestor 对象，其代码如下：

```
initFieldAncestor: function() {
    var me = this,
        onSubtreeChange = me.onFieldAncestorSubtreeChange;
    me.addEvents(
        'fieldvaliditychange',
        'fielderrorchange'
    );
    me.on('add', onSubtreeChange, me);
```

```
        me.on('remove', onSubtreeChange, me);
        me.initFieldDefaults();
    },
```

在代码中先添加了 fieldvaliditychange 和 fielderrorchange 两个事件，然后将 add 和 remove 事件绑定到 onFieldAncestorSubtreeChange 方法，其代码如下：

```
    onFieldAncestorSubtreeChange: function(parent, child) {
        var me = this,
            isAdding = !!child.ownerCt;
        function handleCmp(cmp) {
            var isLabelable = cmp.isFieldLabelable,
                isField = cmp.isFormField;
            if (isLabelable || isField) {
                if (isLabelable) {
                    me['onLabelable' + (isAdding ? 'Added' : 'Removed')](cmp);
                }
                if (isField) {
                    me['onField' + (isAdding ? 'Added' : 'Removed')](cmp);
                }
            }
            else if (cmp.isContainer) {
                Ext.Array.forEach(cmp.getRefItems(), handleCmp);
            }
        }
        handleCmp(child);
    },
```

代码将根据子组件的 ownerCt 属性来判断其状态是增加还是移除。属性 ownerCt 会指向组件的容器，如果该属性值不是空值或不存在，说明此子组件是添加组件，否则是移除组件。

在代码中定义了 handleCmp 函数，其作用是通过递归的方式，遍历组件内所有组件。

如果组件是带标签功能的，则在添加组件的时候将该组件的 errorchange 事件绑定到 handleFieldErrorChange 方法，并将 fieldDefaults 配置项定义的默认配置项通过 setFieldDefaults 方法应用到组件。这样，每当 errorchange 事件被触发时，都会触发 fielderrorchange 事件，并调用 onFieldErrorChange 方法。而在移除组件时，会移除 errorchange 事件的绑定。

如果组件是表单字段，那么在添加时，将该组件的 validitychange 事件绑定到 handleFieldValidityChange 方法，这样，每当 validitychange 事件被触发时，就会触发 fieldvaliditychange 事件，并调用 onFieldValidityChange 方法。而在移除组件组件时，会移除 validitychange 事件的绑定。

以上这些就是 FieldAncestor 对象的作用，在面板添加字段时绑定事件，并将默认配置应用到字段；在移除字段时解除事件绑定，从而将表单字段的错误及验证信息传递到面板。

回到 initFieldAncestor 方法，在绑定事件后，在该方法中调用 initFieldDefaults 方法初始化默认配置项。

方法 initFieldAncestor 在初始化 FieldAncestor 对象后，就调用父对象的 initComponent 方法，在此过程中会调用 FormPanel 的 initItems 方法，代码如下：

```
initItems: function() {
    var me = this;
    me.form = me.createForm();
    me.callParent();
},
```

以上代码调用了 createForm 方法，此方法的代码如下：

```
createForm: function() {
    return new Ext.form.Basic(this, Ext.applyIf({listeners: {}}, this.initial-
        Config));
},
```

以上代码创建了一个 BaseForm 对象实例，并且将 initialConfig 属性内的配置项应用到 BasicForm 对象上。在 AbstractComponent 对象的构造函数中，initialConfig 属性会指向创建对象的配置对象，因此，在创建 BaseForm 对象实例时，表单面板会将其配置项应用到 BaseForm 对象实例，也就是说，**在定义 FormPanel 的配置对象时，可包含 BaseForm 对象的配置项**。

在父对象的 initComponent 方法执行完毕后，就开始定义传播事件。如果 pollForChanges 配置项为 true，则调用 startPolling 方法，代码如下：

```
startPolling: function(interval) {
    this.stopPolling();
    var task = Ext.create('Ext.util.TaskRunner', interval);
    task.start({
        interval: 0,
        run: this.checkChange,
        scope: this
    });
    this.pollTask = task;
},
```

以上代码创建了一个定时任务，定时调用 checkChange 方法，此方法的代码如下：

```
checkChange: function() {
    this.form.getFields().each(function(field) {
        field.checkChange();
    });
}
```

以上代码会调用字段的 checkChange 方法检查字符的变化，也就是说，pollForChanges 配置项会在 pollInterval 配置项指定的间隔内定时检查字段的改变，这不是好的方式，比较耗费系统资源，因而不要随便设置 pollForChanges 为 true。

至此，面板的初始化工作就完成了。

12.1.2 表单面板的配置项、属性、方法和事件

1. 配置项

- fieldDefaults：字段的默认配置对象。

- layout：设置表单面板使用的布局，默认使用的是 Anchor 布局。
- pollForChanges ：布尔值，如果设置为 true，则根据 pollInterval 配置项设置的时间定时检查字段的值的改变，默认值是 false。
- pollInterval ：数字，指定定时检查字段的值改变的时间，单位为微秒。只有在配置项 pollForChanges 为 true 的时候才起作用。

2. 属性

表单面板没有自己属性。

3. 方法

- checkChange：强迫面板内的字段去检查值是否已经改变。
- getForm：返回面板的 BasicForm 对象。
- getRecord：如果模型实例是通过 LoadRecord 方法加载的，那么返回模型实例。
- getValues：返回面板内所有字段的值，其实是调用 BasicForm 的 getValues 方法。
- load：为表单加载数据。
- loadRecord：调用 BasicForm 的 loadRecord 方法加载一个模型。
- startPolling：开始定时任务以检查字段的改变。
- stopPolling：停止定时任务。
- submit：调用 BasicForm 的 submit 方法提交表单。

4. 事件

fielderrorchange：当面板内字段的错误信息发生改变的时候，会触发该事件。

12.1.3　表单的管理：Ext.form.Basic

从上一节可知，FormPanel 只是一个容器，并没有管理表单的功能。实现表单管理功能的是 BasisForm 对象，这个一定要分清楚。在 FormPanel 的初始化过程中会创建 BasisForm 对象并调用其 initItems 方法。下面看看 FormPanel 是怎么初始化的，其构造函数如下：

```
constructor: function(owner, config) {
    var me = this,
        onItemAddOrRemove = me.onItemAddOrRemove,
        api,
        fn;
    me.owner = owner;
    me.mon(owner, {
        add: onItemAddOrRemove,
        remove: onItemAddOrRemove,
        scope: me
});
Ext.apply(me, config);
if (Ext.isString(me.paramOrder)) {
    me.paramOrder = me.paramOrder.split(/[\s,|]/);
}
if (me.api) {
    api = me.api = Ext.apply({}, me.api);
```

```
        for (fn in api) {
            if (api.hasOwnProperty(fn)) {
                api[fn] = Ext.direct.Manager.parseMethod(api[fn]);
            }
        }
    }
    me.checkValidityTask = new Ext.util.DelayedTask(me.checkValidity, me);
    me.addEvents(
            // 省略事件代码
    );
    me.callParent();
},
```

以上代码先为 BasicForm 拥有者（owner）的 add 和 remove 事件绑定 onItemAddOr-Remove 方法，其代码如下：

```
onItemAddOrRemove: function(parent, child) {
    var me = this,
        isAdding = !!child.ownerCt,
        isContainer = child.isContainer;
    function handleField(field) {
        me[isAdding ? 'mon' : 'mun'](field, {
            validitychange: me.checkValidity,
            dirtychange: me.checkDirty,
            scope: me,
            buffer: 100
        });
        delete me._fields;
    }

    if (child.isFormField) {
        handleField(child);
    }
    } else if (isContainer) {
        if (child.isDestroyed) {
            delete me._fields;
        } else {
            Ext.Array.forEach(child.query('[isFormField]'), handleField);
        }
    }    delete this._boundItems;
    if (me.initialized) {
        me.checkValidityTask.delay(10);
    }
},
```

OnItemAddOrRemove 方法与 FieldAncestor 对象的 onFieldAncestorSubtreeChange 很类似，通过子组件的 ownerCt 判断该组件目前需要进行添加还是移除，如果添加，则绑定 validitychange 事件到 checkValidity 方法，绑定 dirtychange 事件到 checkDirty 方法，也就是组件的验证状态发生改变和数据发生改变都会通知 BasicFrom 对象进行相应处理，也会触发对应的 validitychange 或 dirtychange 事件。

完成事件绑定后，将配置对象复制到 BasicForm，如果定义配置项 paramOrder 是字符串，则将其转换为数组，这是 Ext.Direct 才需要使用的配置项。

接着创建 DelayedTask 对象的实例来实现缓冲任务，在添加事件后，调用 Observable 对象的构造函数。

接着介绍 BasicForm 的 initialize 方法，其代码如下：

```
initialize: function(){
    var me = this;
    me.initialized = true;
    me.onValidityChange(!me.hasInvalidField());
},
```

修改属性 initialized 为 true，表示初始化已经完成，然后调用 onValidityChange 方法，此方法的代码如下：

```
onValidityChange: function(valid) {
    var boundItems = this.getBoundItems();
    if (boundItems) {
        boundItems.each(function(cmp) {
            if (cmp.disabled === valid) {
                cmp.setDisabled(!valid);
            }
        });
    }
},
```

以上代码先调用 getBoundItems 方法获取组件，其代码如下：

```
getBoundItems: function() {
    var boundItems = this._boundItems;
    if (!boundItems || boundItems.getCount() === 0) {
        boundItems = this._boundItems = new Ext.util.MixedCollection();
        boundItems.addAll(this.owner.query('[formBind]'));
    }
    return boundItems;
},
```

从代码中可以看到，getBoundItems 方法会将带有 formBind 属性的组件放到 MixedCollection 对象实例中并返回组件。

返回组件后，会根据参数 valid 的值来判断是禁用还是启用这些组件。

Valid 是 hasInvalidField 方法的返回值。现在看看 valid 的值是怎么来的，其代码如下：

```
hasInvalidField: function() {
    return !!this.getFields().findBy(function(field) {
        var preventMark = field.preventMark,
            isValid;
        field.preventMark = true;
        isValid = field.isValid();
        field.preventMark = preventMark;
        return !isValid;
    });
},
```

先调用了 getFields 方法，此方法的代码如下：

```
getFields: function() {
    var fields = this._fields;
    if (!fields) {
        fields = this._fields = new Ext.util.MixedCollection();
        fields.addAll(this.owner.query('[isFormField]'));
    }
    return fields;
},
```

执行以上代码会将所有带 isFormField 属性的组件保存到 MixedCollection 对象实例中并返回。

返回后，会枚举每个获取的组件，然后调用其 isValid 方法验证其是否是有效值，如果有效，isValid 方法会返回 false，而 findBy 方法会在 isValid 方法返回 true 时停止，也就是在存在无效值的时候，findBy 方法就结束了，并将该组件返回，经过两次非操作后就是 true 值，调用 onValidityChange 方法再执行一次非操作，就是 false。也就是当字段的值无效的时候，会设置带 formBind 属性的组件的 disabled 属性为 true，也就是禁用这些组件。反过来，当没有无效值的时候，会设置这些组件的 disabled 属性为 false，启用这些组件。对这种情况是否很熟悉？这可以用来控制提交按钮，不过只在全部字段值都有效时才允许提交按钮。

至此 BasicForm 对象的初始化过程就结束了，从初始化过程来看，它只是做了错误显示和验证处理，实际上它的工作远不止这些。下面来看看 BasicForm 的一些重要方法。

1. findField：查找字段

此方法的源代码如下：

```
findField: function(id) {
    return this.getFields().findBy(function(f) {
        return f.id === id || f.getName() === id;
    });
},
```

先通过 getFields 返回所有字段，然后通过 findBy 方法枚举所有字段，当字段的 id 或者 name 属性等于参数 id 时，返回该字段。

2. getValues：返回所有字段的值

此方法的源代码如下：

```
getValues: function(asString, dirtyOnly, includeEmptyText, useDataValues) {
    var values = {};

    this.getFields().each(function(field) {
        if (!dirtyOnly || field.isDirty()) {
            var data = field[useDataValues ? 'getModelData' : 'getSubmitData']
                (includeEmptyText);
            if (Ext.isObject(data)) {
                Ext.iterate(data, function(name, val) {
                    if (includeEmptyText && val === '') {
                        val = field.emptyText || '';
                    }
                    if (name in values) {
```

```
                var bucket = values[name],
                    isArray = Ext.isArray;
                if (!isArray(bucket)) {
                    bucket = values[name] = [bucket];
                }
                if (isArray(val)) {
                    values[name] = bucket.concat(val);
                } else {
                    bucket.push(val);
                }
            } else {
                                values[name] = val;
                            }
                        });
                    }
                }
            });

            if (asString) {
                values = Ext.Object.toQueryString(values);
            }
            return values;
        },
```

以上代码先创建了 values 对象来记录结果，然后使用 getFields 获取所有字段，接着进行枚举操作。获取的值会根据参数 useDataValues 从模型或准备提交的数据中获取值，其默认值为 undefined，是从提交值中获取的。

值返回之后，如果 data 是一个对象，则对数据进行迭代操作，如果字段名称在 values 对象中已存在，则要将值改为数组，这主要是针对复选框的。否则，直接将字段及其值加入到对象中。

如果设置参数 asString 为 true，则将字段值转换为字符串后再返回。

3. setValues：设置字段的值

此方法的源代码如下：

```
setValues: function(values) {
    var me = this;
    function setVal(fieldId, val) {
        var field = me.findField(fieldId);
        if (field) {
            field.setValue(val);
            if (me.trackResetOnLoad) {
                field.resetOriginalValue();
            }
        }
    }
    if (Ext.isArray(values)) {
        Ext.each(values, function(val) {
            setVal(val.id, val.value);
        });
    } else {
        Ext.iterate(values, setVal);
```

```
        }
        return this;
    },
```

从 setVal 函数的代码中可以看到，最终还是使用字段的 setValue 方法来设置字段值。

如果设置了 trackResetOnLoad 属性，则会调用 resetOriginalValue 方法重置原始值，其作用就是编辑一条记录时，把记录的值作为原始值，这样单击重置按钮时就不会将数据重置为空值了。

4. doAction：处理表单的操作

submit 方法和 load 方法都是调用 doAction 方法进行提交或加载操作的，只是简化了参数，因此只要清楚 doAction 的运作就行了，其代码如下：

```
doAction: function(action, options) {
    if (Ext.isString(action)) {
        action = Ext.ClassManager.instantiateByAlias('formaction.' + action, Ext.
            apply({}, options, {form: this}));
    }
    if (this.fireEvent('beforeaction', this, action) !== false) {
        this.beforeAction(action);
        Ext.defer(action.run, 100, action);
    }
    return this;
},
```

如果对应的操作对象还没创建，则先创建对象，如果 beforeaction 事件没有返回 false，则调用 beforeAction 方法，其代码如下：

```
    beforeAction: function(action) {
        var waitMsg = action.waitMsg,
            maskCls = Ext.baseCSSPrefix + 'mask-loading',
            waitMsgTarget;
        this.getFields().each(function(f) {
            if (f.isFormField && f.syncValue) {
                f.syncValue();
            }
        });

        if (waitMsg) {
            waitMsgTarget = this.waitMsgTarget;
            if (waitMsgTarget === true) {
                this.owner.el.mask(waitMsg, maskCls);
            } else if (waitMsgTarget) {
                waitMsgTarget = this.waitMsgTarget = Ext.get(waitMsgTarget);
                waitMsgTarget.mask(waitMsg, maskCls);
            } else {
                Ext.MessageBox.wait(waitMsg, action.waitTitle || this.waitTitle);
            }
        }
    },
```

在以上代码中先找到 HtmlEditor 对象，将其值同步到字段，然后设置遮蔽的等待信息，

并显示遮蔽。

调用完 beforeAction 方法后，设置在 100 微秒后调用 run 方法执行操作。

12.1.4　BasicForm 的配置项、属性、方法和事件

从 12.1.1 节可以知道，BaseForm 的配置项可在 FormPanel 中定义，这个要记住。

1. 配置项

- api：配置对象，包括 load 和 submit 两个配置项，用来配置表单加载数据和提交数据时的地址或 Direct 方法。
- baseParams：配置对象，为加载或提交数据定义基本的提交参数。
- errorReader：用来读取服务器端返回的错误信息的 Reader 配置对象。
- method：发送请求的方式，值可以为 GET 或 POST（字母必须全部为大写）。
- paramOrder：提交参数的顺序，这主要用于 Direct 方式发送请求。
- paramsAsHash ：布尔值，该配置项只用于 api 中的 load 配置项。当其值为 true 时，所有参数会作为一个 JSON 对象发送到服务器端。默认值为 false。
- reader ：Reader 对象的配置对象，用以定义执行 load 操作时读取返回数据的 Reader 对象。
- standardSubmit ：布尔值，如果为 true，会使用标准的表单提交方式提交数据。默认值为 false，表示以 Ajax 方法提交数据。
- timeout：发送请求判断是否超时的数值，默认值为 30 秒。
- trackResetOnLoad ：布尔值，如果为 true，会调用 reset 方法，将表单内字段值重新设置为最后一次加载的数据，或用 setValues 方法设置的数据代替表单第一次创建时的数据。默认值为 false。
- url：表单操作的提交地址。
- waitMsgTarget ：显示等待信息模板元素，默认值为 undefined，使用 MessageBox 对象显示等待信息。
- waitTitle：提交表单时显示在等待信息提示框中的标题，默认值为“Please Wait...”。

2. 属性

owener：BasicForm 所在的容器组件。

3. 方法

- applyIfToFields：调用 applyIf 方法将指定的配置对象应用到所有字段。
- applyToFields：调用 apply 方法将指定的配置对象应用到所有字段。
- checkDirty ：检查是否有脏数据。该方法会触发 dirtychange 事件，并修改 wasDirty 属性的值。
- clearInvalid：清理所有字段的无效信息。
- destroy：摧毁当前 BasicForm 对象实例。

- doAction：执行操作。
- findField：根据指定的 id 或 name 查找字段。
- getFieldValues：返回所有字段的值。如果参数为 true，可以只返回被修改过的数据。
- getFields：通过 MixedCollection 对象实例返回所有字段。
- getRecord：返回最后一个使用 loadRecord 方法加载的模型实例。
- getValues：返回表单内所有字段的值。
- hasInvalidField：如果存在无效的值，那么返回 true，否则返回 false。
- hasUpload：如果表单包含文件上传字段，那么返回 true。
- isDirty：如果表单内的字段被修改过，那么返回 true。
- isValid：如果表单内所有字段的值都是有效的，那么返回 true。
- load：执行加载操作。
- loadRecord：加载一个数据模型实例，然后调用 setValues 方法将模型的数据应用到字段。
- markInvalid：根据指定的错误信息数组或对象，将错误应用到字段。
- reset：重置表单内字段的值。
- submit：执行提交操作。
- updateRecord:：将字段的值应用到指定的模型实例。

4. 事件

- actioncomplete：当操作完成后会触发该事件。
- actionfailed：当操作执行失败时触发该事件。
- beforeaction：在操作执行前会触发该事件，返回 false 可中止操作。
- dirtychange：当脏状态被改变时，会触发该事件。
- validitychange：当表单内字段的值被修改为有效值时，会触发该事件。

12.1.5 表单的操作：Ext.form.action.Action

表单的操作主要有提交和加载两个，这些操作如图 12-1 所示，都以 FormAction 为基类，然后派生出 LoadAction 和 SubmitAction 两个基类，这两个类默认都以 Ajax 作为数据交换方式。为了使用 Direct 方法提交和加载数据，扩展出了 DirectLoadAction 和 DirectSubmitAction 两个类。为了实现标准的表单提交方式，又扩展出了 StandardSubmitAction 类。

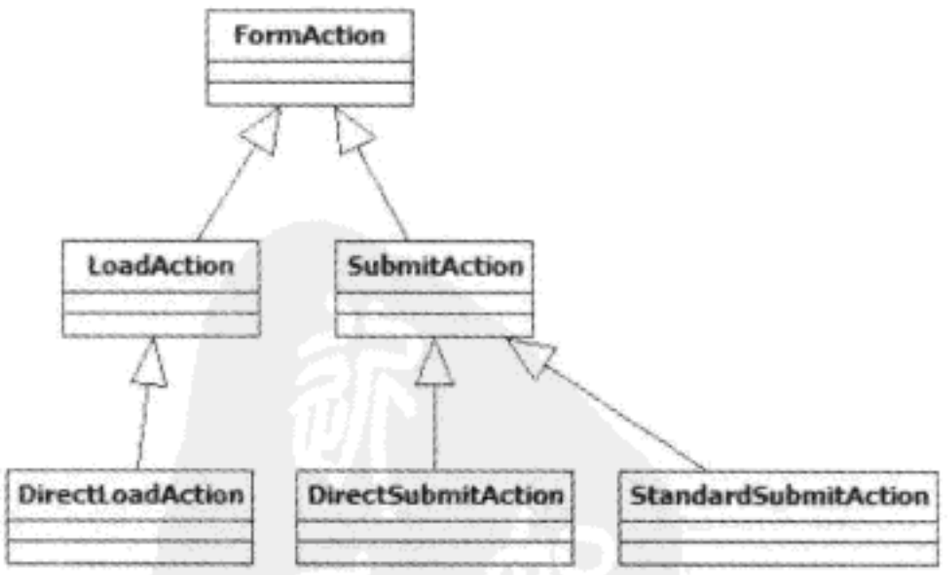


图 12-1 表单的操作类图

FormAction 对象作为基类，为子类提供了基本的配置项、属性和方法。

1. 配置项

- failure：请求发生错误后的回调函数。

- form：发送请求的 BasicForm 对象。
- headers：Ajax 请求的头信息。
- method：请求的发送方式，其值可以为 POST 或 GET，全部为大写字母。默认值为 POST。
- params：要提交的额外参数。
- reset：布尔值，如果设置此值为 true，操作成功后会重置表单。
- scope：作用域。
- submitEmptyText：布尔值，如果是默认值 true，会将 emptyText 配置项设置的值提交。
- success：提交成功后的回调函数。
- timeout：发送请求判断超时的数值，默认值为 30 秒。
- url：发送请求的地址。
- waitMsg：提交表单时显示在等待信息提示框中的信息。
- waitTitle：提交表单时显示在等待信息提示框中的标题。

2. 属性

- failureType：请求发生错误后的错误类型，此属性值可为 CLIENT_INVALID、SERVER_INVALID、CONNECT_FAILURE 或 LOAD_FAILURE。
- response：执行操作使用的原生 XMLHttpRequest 对象。
- result：解码后的响应对象。
- type：操作实例的类型，值可以为 submit 或 load。
- CLIENT_INVALID：如果客户端验证失败，会中止提交操作，返回该错误类型。
- SERVER_INVALID：当服务器端返回结果中的 success 为 false 时，会返回该错误类型。服务器端返回结果中的 errors 属性可包含字段的错误信息。
- CONNECT_FAILURE：发送请求时发生错误时，会返回该错误类型。
- LOAD_FAILURE：当请求的结果中 success 为 false 且 data 属性中没有字段值返回时，会返回该错误类型。

3. 方法

run：开始执行当前操作。

12.1.6　加载操作的运行流程：Ext.form.action.Load 与 Ext.form.action.DirectLoad

所有操作都是从 run 方法起步的，LoadAction 对象的 run 方法代码如下：

```
run: function() {
    Ext.Ajax.request(Ext.apply(
        this.createCallback(),
        {
            method: this.getMethod(),
            url: this.getUrl(),
            headers: this.headers,
```

```
                params: this.getParams()
            }
        ));
    },
```

这里直接调用了 Ajax 对象的 request 方法，而 Ajax 对象的配置对象是 createCallback 方法返回的对象与配置项结合的对象。方法 createCallback（Action.js）的代码如下：

```
createCallback: function() {
    var me = this,
        undef,
        form = me.form;
    return {
        success: me.onSuccess,
        failure: me.onFailure,
        scope: me,
        timeout: (this.timeout * 1000) || (form.timeout * 1000),
        upload: form.fileUpload ? me.onSuccess : undef
    };
},
```

这里主要是绑定了提交成功或失败后的回调函数。成功后的回调函数是 onSuccess，其代码如下：

```
onSuccess: function(response){
    var result = this.processResponse(response),
        form = this.form;
    if (result === true || !result.success || !result.data) {
        this.failureType = Ext.form.action.Action.LOAD_FAILURE;
        form.afterAction(this, false);
        return;
    }
    form.clearInvalid();
    form.setValues(result.data);
    form.afterAction(this, true);
},
```

以上代码先调用 processResponse 方法（Action.js）处理响应，其代码如下：

```
processResponse : function(response){
    this.response = response;
    if (!response.responseText && !response.responseXML) {
        return true;
    }
    return (this.result = this.handleResponse(response));
},
```

如果在返回的数据中，responseText 或 responseXML 属性都不存在，则返回 true，否则，调用 handleResponse 方法处理返回数据并返回其返回结果，其代码如下：

```
handleResponse: function(response) {
    var reader = this.form.reader,
        rs, data;
    if (reader) {
```

```
            rs = reader.read(response);
            data = rs.records && rs.records[0] ? rs.records[0].data : null;
            return {
                success : rs.success,
                data : data
            };
        }
        return Ext.decode(response.responseText);
    }
```

以上代码先检查是否定义了 Reader 对象，如果定义了，则使用 Reader 对象读取数据，然后将数据组合成 FormAction 规定的数据格式返回。如果没有定义 Reader 对象，则直接调用 decode 方法将文本转换为对象返回。这里，一般只有在使用 XML 作为数据返回格式的时候才需要定义 Reader。

在返回数据并调用 onSuccess 方法后，如果 result 为 true，那么在调用 processResponse 方法时，返回的数据不存在 responseText 或 responseXML 属性、如果 success 为 false，或者没有数据，那么说明加载数据失败，failureType 属性被设置为 LOAD_FAILURE。接着调用 BasicForm 对象的 afterAction 方法，其代码如下：

```
afterAction: function(action, success) {
    if (action.waitMsg) {
        var MessageBox = Ext.MessageBox,
            waitMsgTarget = this.waitMsgTarget;
        if (waitMsgTarget === true) {
            this.owner.el.unmask();
        } else if (waitMsgTarget) {
            waitMsgTarget.unmask();
        } else {
            MessageBox.updateProgress(1);
            MessageBox.hide();
        }
    }
    if (success) {
        if (action.reset) {
            this.reset();
        }
        Ext.callback(action.success, action.scope || action, [this, action]);
        this.fireEvent('actioncomplete', this, action);
    } else {
        Ext.callback(action.failure, action.scope || action, [this, action]);
        this.fireEvent('actionfailed', this, action);
    }
},
```

以上代码先取消等待信息和遮蔽的显示，如果成功，调用由 success 配置项定义的回调函数，并触发 actioncomplete 事件；如果不成功，调用由 failure 配置项定义的回调函数，并触发 actionfailed 事件。

继续介绍 onSuccess 方法，如果数据加载请求，则先调用 clearInvalid 方法，清理无效的提示信息，然后调用 setValues 方法将结果设置到字段中。最后调用 afterAction 方法。

请求失败时调用的 onFailure 方法，其代码如下：

```
onFailure : function(response){
    this.response = response;
    this.failureType = Ext.form.action.Action.CONNECT_FAILURE;
    this.form.afterAction(this, false);
},
```

直接设置 failureType 的值为 CONNECT_FAILURE，最后调用 afterAction 方法。

至此，LoadAction 对象的加载过程就结束了。下面看看 DirectLoadAction 的数据加载过程，其 run 方法代码如下：

```
run: function() {
    this.form.api.load.apply(window, this.getArgs());
},
```

直接调用 api 属性指向对象的 load 方法，参数是 getArgs 方法返回的参数，其代码如下：

```
getArgs: function() {
    var me = this,
        args = [],
        form = me.form,
        paramOrder = form.paramOrder,
        params = me.getParams(),
        i, len;
    if (paramOrder) {
        for (i = 0, len = paramOrder.length; i < len; i++) {
            args.push(params[paramOrder[i]]);
        }
    }
    else if (form.paramsAsHash) {
        args.push(params);
    }
    args.push(me.onSuccess, me);
    return args;
},
```

因为提交参数的顺序要与服务器端方法内的参数对应起来，所以这里要根据配置项 paramOrder 的定义顺序绑定参数。如果将 JSON 格式的提交参数作为一个对象提交（定义了 paramsAsHash 配置项），则将 params 对象作为一个参数提交。

在代码的最后，也会将回调函数 onSuccess 方法作为一个参数提交，其代码如下：

```
onSuccess: function(result, trans) {
    if (trans.type == Ext.direct.Manager.self.exceptions.SERVER) {
        result = {};
    }
    this.callParent([result]);
}
```

如果返回数据的 type 属性为 Ext.direct.Manager.self.exceptions.SERVER，则将返回结果设置为空对象，然后调用 LoadAction 的 onSuccess 方法。

DirectLoadAction 重写了 FormAction 对象的 processResponse 方法，其代码如下：

```
processResponse: function(result) {
    return (this.result = result);
},
```

这里不需要进行任何处理就直接返回结果。

12.1.7　提交操作的运行流程：Ext.form.action.Submit、Ext.form.action.DirectSubmit 与 Ext.form.action.StandardSubmit

SubmitActi 的 run 方法如下：

```
run : function(){
    var form = this.form;
    if (this.clientValidation === false || form.isValid()) {
        this.doSubmit();
    } else {
        this.failureType = Ext.form.action.Action.CLIENT_INVALID;
        form.afterAction(this, false);
    }
},
```

如果有效值验证通过，则调用 doSubmit 方法提交数据，否则，设置 failureType 为 CLIENT_INVALID 并调用 afterAction 方法。

doSubmit 方法的代码如下：

```
doSubmit: function() {
    var formEl,
        ajaxOptions = Ext.apply(this.createCallback(), {
            url: this.getUrl(),
            method: this.getMethod(),
            headers: this.headers
        });
    if (this.form.hasUpload()) {
        formEl = ajaxOptions.form = this.buildForm();
        ajaxOptions.isUpload = true;
    } else {
        ajaxOptions.params = this.getParams();
    }
    Ext.Ajax.request(ajaxOptions);
    if (formEl) {
        Ext.removeNode(formEl);
    }
},
```

从对 createCallback 方法的调用可以知道，回调函数与 LoadAction 方法的设置是一样的。

如果表单内有文件上传字段，则使用 buildForm 方法创建 form 标记，然后将非文件上传字段作为隐藏的 input 标记。文件上传字段因具备特殊性，只能转移到 form 标记下。做好以上这些后就将 form 标记的元素返回。

处理完 form 标记或提交参数后，就调用 Ajax 对象的 request 方法提交数据。提交操作完成后，如果 formEl 存在，则将其移除。

SubmitAction 对象重写了 onSuccess 方法，其代码如下：

```
onSuccess: function(response) {
    var form = this.form,
        success = true,
        result = this.processResponse(response);
    if (result !== true && !result.success) {
        if (result.errors) {
            form.markInvalid(result.errors);
        }
        this.failureType = Ext.form.action.Action.SERVER_INVALID;
        success = false;
    }
    form.afterAction(this, success);
},
```

还是先调用 processResponse 方法处理返回的数据，其中会调用 handleResponse 方法。在 SubmitAction 对象中，handleResponse 被重写了，代码如下：

```
handleResponse: function(response) {
    var form = this.form,
        errorReader = form.errorReader,
        rs, errors, i, len, records;
    if (errorReader) {
        rs = errorReader.read(response);
        records = rs.records;
        errors = [];
        if (records) {
            for(i = 0, len = records.length; i < len; i++) {
                errors[i] = records[i].data;
            }
        }
        if (errors.length < 1) {
            errors = null;
        }
        return {
            success : rs.success,
            errors : errors
        };
    }
    return Ext.decode(response.responseText);
}
```

如果配置项 errorReader 存在，即 handleResponse 方法返回的是 XML 数据，那么需要将 XML 数据转换为 JOSN 数据，然后以标准的 JSON 格式返回数据。如果配置项 errorReader 不存在，则直接使用 decode 方法将字符串转换为 JSON 对象返回。

回到 onSuccess 方法中，如果提交后有数据返回，且 Success 属性是 false，则检查结果中是否有字段的错误信息（errors 属性是否存在），如果有，则调用 markInvalid 方法显示字段的错误信息，然后设置 failureType 为 Action.SERVER_INVALID，success 为 false。

最后调用 afterAction 方法，完成操作。

DirectSubmitAction 没有重写 run 方法，不过，它重写了 doSubmit 方法，代码如下：

```
doSubmit: function() {
    var me = this,
        callback = Ext.Function.bind(me.onSuccess, me),
        formEl = me.buildForm();
    me.form.api.submit(formEl, callback, me);
    Ext.removeNode(formEl);
},
```

从代码中可以看到，DirectSubmitAction 对象一直以标准的表单方式提交，提交成功后回调的函数是 onSuccess 方法，DirectSubmitAction 对象也重写了该方法，与 DirectLoadAction 的处理过程是一样的。

StandardSubmitAction 对象只重写了 doSubmit 方法，其代码如下：

```
doSubmit: function() {
    var form = this.buildForm();
    form.submit();
    Ext.removeNode(form);
}
```

以上代码相当简单，创建 form 元素后，立即将其提交，然后将其删除。

12.1.8　字段的构成

从图 8-3 的字段的类结构图可以看到，BaseField 是所有字段类的基类，它派生于 Component 对象，混入了 FormField 对象和 Labelable 对象，因此字段主要由 BaseField 对象、FormField 对象和 Labelable 对象这 3 个对象构成。其中，FormField 对象为字段提供了获取和设置值的方法，跟踪值的变化及有效性检查的事件和方法，触发验证的方法；Labelable 对象则为字段提供了标签及显示错误信息的功能。

在 4.1 Beta 1 版，为了提高性能，减少样式运算，放弃了 4.0.7 版使用的如图 12-2 所示字段的 HTML 代码构成，改为使用表格作为字段显示的容器，标签、字段和错误提示分别在一个单元格内。

图 12-2　字段的 HTML 代码构成

12.1.9　BaseField 的配置项、属性、方法和事件

1. 配置项

- activeError：默认值为 undefined。如果设置了该值，那么当字段第一次渲染后，会把该值作为活动的错误信息显示。创建组件后，可使用 setActiveError 或 unsetActiveError

来改变该值。

- activeErrorsTpl：功能与 activeError 相同，不过它使用模板来显示错误信息。
- autoFitErrors：布尔值，默认值为 true，当错误信息显示被设置为 side 或 under 的时候，会自动调整字段的主体区域去适应错误信息的显示。
- baseBodyCls：应用于字段主体的样式类，默认值为 x-form-item-body。
- checkChangeBuffer：指定在 checkChangeEvents 配置项中定义的事件的缓冲时间，以避免频繁触发这些事件。默认值是 50 微秒。
- checkChangeEvents：由 input 元素监听值改变事件组成的数组。在 IE 浏览器中，默认值是“['change','propertychang']”，在其他浏览器中是“['change','input','textInput','keyup','dragdrop']”。
- clearCls：清除字段中 float 样式的样式类名称，默认值为“x-clear”。
- dirtyCls：当字段值为脏数据的时候应用的样式类名称，默认值为“form-dirty”。
- disabled：布尔值，如果为 true，会禁用字段，默认值为 false。
- errorMsgCls：错误信息的样式类名称，默认值为“x-form-error-msg”。
- fieldBodyCls：字段主体的样式类名称，默认值为空。
- fieldCls：字段的样式类名称，默认值为“x-form-field”。
- fieldLabel：字段的标签。
- fieldStyle：应用到字段的样式。
- focusCls：字段获得焦点后的样式名称。
- formItemCls：应用到字段最外层元素的样式类名称，默认值为“x-form-item”。
- hideEmptyLabel：布尔值，默认值为 true，字段会隐藏空的标签。如果设置为 false，则当标签为空时，也会占位置，非常适合不需要标签但要对齐字段的情形。
- inputId：input 标记的 id。默认会自动为其产生 id。
- inputType：input 标记的 type 属性的值，默认值为 text。
- invalidCls：当值无效时应用的样式类名称，默认值为“x-form-invalid”。
- invalidText：当值无效时显示的错误信息。
- labelAlign：标签的位置。值可以是 top（顶部）、left（左边）或 right（右边），默认值为 left。
- labelCls：应用到标签的样式类名称，默认值为“x-form-item-label”。
- labelPad：设置标签的内补丁，默认值为 5。
- labelSeparator：在标签文本后的符号，默认值为英文的冒号（:）。
- labelStyle：应用到标签的样式。
- labelWidth：标签的宽度，默认值为 100。
- labelableRenderTpl：标签渲染时使用的模板。
- msgTarget：错误信息的显示位置，值可以为 qtip（默认方式，使用 Quicktips 显示）、title（在 title 属性中显示）、under（在字段下面显示）、side（在字段旁边显示一个错误图标，鼠标移动到该位置时弹出错误信息）、none（不显示错误信息）和元素 id（在指

定元素内显示）。

- name：字段 name 属性的值，默认值为 undefined。
- preventMark：布尔值，设置为 true 会屏蔽所有错误信息。默认值为 false。
- readOnly：布尔值，设置为 true 会设置字段的 DOM 属性 readOnly 为 true。不过它不能禁用 Combox 或 DateField 的下拉按钮。默认值为 false。
- readOnlyCls：只读状态下应用于元素的样式，默认值为“x-form-readonly”。
- submitValue ：布尔值，如果该值为 false，即使字段没有被设置为禁用，也不会提交该值。默认值为 true。
- tabindex：使用 Tab 键切换字段时的索引，默认值为 undefined。
- validateOnBlur：布尔值，默认值为 true，只要字段失去焦点，就会进行验证。
- validateOnChange：布尔值，默认值为 true，当值被改变时就进行验证。
- value：字段的值。

2. 属性

- bodyEl：指向字段主体的 div 元素。
- errorEl：指向字段显示错误信息的 div 元素。
- inputEl：指向字段的 input 元素。
- isFieldLabelable：表示字段带有标签功能，默认值为 true。
- isFormField：表示字段是表单字段，默认值为 true。
- labelEl：指向字段的 label 元素。
- originalValue：字段的原始值。

3. 方法

- batchChanges ：通过指定的函数去修改值，避免触发 change 事件，主要用于包含子字段的情形。
- checkChange：检查值的变化，并验证值。
- checkDirty：检查字段的 isDirty 状态，如果最后一次修改发生了改变，触发 dirtychange 事件。
- clearInvalid：清理字段的无效样式和信息。
- extractFileInput ：只有字段的 isFileUpload 为 true 的时候才起作用，返回文件选择框的 DOM 元素。
- getActiveError：不执行验证，以字符串形式返回当前的错误信息。
- getActiveErrors：不允许执行验证，以数组形式返回当前的错误信息。
- getErrors：运行验证并以数组形式返回错误信息。
- getFieldLabel：返回字段的标签。
- getInputId：返回 input 元素的 id。
- getModelData：返回字段将要保存到模型实例的值。
- getName：返回字段 name 属性的值。

- ❑ getRawValue：返回字段的实际值。
- ❑ getSubmitData：以对象形式返回字段，以标准表单形式提交参数名称及其值。
- ❑ getSubmitValue：返回字段以标准表单形式提交的值。
- ❑ getValue：返回字段的当前值。
- ❑ hasActiveError：如果字段当前存在错误，返回 true，否则返回 false。
- ❑ isDirty：如果数据被修改过，返回 true，否则返回 false。
- ❑ isEqual：检查两个字段值是否是逻辑相等的。该方法可根据字段的数据类型重写该方法来实现自定义的比较。默认将值转换为字符串比较。
- ❑ isFileUpload：如果字段是文件选择框，返回 true，否则返回 false。
- ❑ isValid：如果字段值是有效的，返回 true，否则返回 false。
- ❑ markInvalid：显示错误信息。
- ❑ processRawValue：将值转换为真实值。
- ❑ rawToValue：将真实值转换为显示值。
- ❑ reset：重置字段值。
- ❑ resetOriginalValue：重置字段原始值。
- ❑ setActiveError：将指定的信息设置为当前的错误信息。
- ❑ setActiveErrors：将指定的数组内的信息设置为当前的错误信息。
- ❑ setFieldDefaults：设置字段的默认配置。
- ❑ setFieldStyle：设置字段的样式。
- ❑ setRawValue：设置字段的实际值。
- ❑ setReadOnly：设置字段的只读状态。
- ❑ setValue：设置字段的值。
- ❑ unsetActiveError：清除当前的错误信息。
- ❑ validate：验证字段，如果是有效值，那么返回 true，否则返回 false。
- ❑ validateValue：使用 getErrors 创建一个验证错误的数组，如果有错误，调用 markInvalid 显示错误并返回 false，否则返回 true。
- ❑ valueToRaw：将显示值转换为实际值。

4. 事件

- ❑ blur：当字段失去焦点的时候触发该事件。
- ❑ change：当字段值被修改的时候触发该事件。
- ❑ dirtychange：当字段 isDirty 状态被修改的时候触发该事件。
- ❑ errorchange：当前错误信息被改变的时候触发该事件。
- ❑ focus：当字段获得焦点的时候触发该事件。
- ❑ specialkey：当按下箭头、Tab、回车或 Esc 等键的时候，触发该事件。
- ❑ validitychange：当字段的有效性被改变时会触发该事件。

12.1.10　常用的验证函数：Ext.form.field.VTypes

VTypes 对象是单件模式的对象，因而可直接使用其定义的方法，不过，它一般不单独使用，主要用于字段的配置项 vtype，作用是验证字段的输入值是否有效。

VTypes 对象定义了以下 4 个常用的验证函数。

- alpha：值只能是字母。
- alphanum：值只能是字母和数字。
- email：值要符合电子邮件格式。
- url：值要符合互联网地址格式。

只有 4 个验证函数是不能满足需求的，因此需要扩展验证函数。要扩展不难，只要在本地化文件中找到 VTypes 对象的本地化定义，然后在 Ext.apply 里加入就行了。实现扩展只需要定义一个属性并指向一个函数，例如，规定值只能是汉字，可定义如下：

```
chinese:function(v){
    return /^[\u4e00-\u9fa5]+$/i.test(v)
}
```

当验证到无效值时，要显示错误信息，方法是在扩展函数的属性名称加上“Text”，错误信息作为值，例如只输入汉字的错误信息可定义如下：

```
chineseText:'必须输入中文。'
```

如果希望禁止非法字符的输入，可定义掩码，掩码的属性名称为扩展函数的属性名称加上“Mask”，例如示例中的掩码可定义如下：

```
chineseMask:/[\W]/i
```

这样就禁止了字母和数字的输入，如果还想扩展到符号，可在正则表达式中加入对应的定义。

12.2　使用字段

1. 使用 Label

Label 组件的功能是创建一个 Label 标记，用于显示标签，标签中的内容可以是 HTML 格式文本或纯文本。要显示 HTML 格式文本，可使用 html 配置项，要显示纯文本，使用 text 配置项。可通过 setText 方法改变标签的内容。

2. 使用 Text 字段

Text 字段是表单最基本的输入字段，它派生于 BasicField 对象，是 ComboBox、TextArea 和 File 等类的基类。它通过 24 个配置项来实现不同的功能，下面，我们通过一个示例来演示如何使用 Text 字段。

首先使用模板页创建一个名称为 12-1.html 的页面文件，然后在文件中创建一个表单面板。

```
Ext.create("Ext.form.Panel",{
```

```
title:" 使用 Text 字段 ",
width:300,
height:300,
renderTo:Ext.getBody(),
bodyPadding:5,
bodyStyle:"background:#DFE9F6",
fieldDefaults:{labelWidth:120,msgTarget:"side",
        labelSeparator:": "
},
items:[
]
})
```

表单面板与一般面板除了在 fieldDefaults 配置项定义上不同外，区别不是太大。在配置项 fieldDefaults 定义中，定义了字段的标签宽度（labelWidth）为 120 像素，错误信息显示方式（msgTarget）是在字段旁边显示一个图标，将标签与字段的分隔符号（labelSeparator）修改为全角的冒号。

下面在 Items 配置项中加入 Text 字段。第一个字段用来测试不能为空的字段，这涉及 allowBlank 和 blankText 两个配置项。allowBlank 配置项的值为 false，表示字段不允许为空，而错误信息就是配置项 blankText 定义的值，该值一般已在本地化文件中进行本地化定义，不需要重复定义。当字段值为空时，可使用配置项 emptyText 显示一些提示信息；当字段获得焦点时，该文本就会自动移除等待输入。以下是第一个字段的定义代码：

```
{xtype:"textfield",fieldLabel:" 字段 1",name:"field1",
   allowBlank:false,emptyText:" 请输入 "
},
```

第二个字段用来测试自动增长的字段，这涉及 grow、growMax、growMin 和 growAppend 这 4 个配置项。只有将 grow 配置为 true 的时候，才可以实现字段增长的功能。配置项 growMin 定义了字段能缩小的最小宽度，而 growMax 则定义了最大宽度。配置项 grow-Append 的作用是定义一个字符作为占位符，默认值为“W”，用来计算字段的宽度。以下是第二个字段的定义代码：

```
{xtype:"textfield",fieldLabel:" 字段 2",name:"field2",
   grow:true,growMax:60,growMin:20
},
```

第三个字段用来测试控制了输入长度的字段，这涉及 enforceMaxLength、maxLength。enforceMaxLength 的作用是强迫在 input 标记中加入 maxLength 属性，是否需要，则看需求。主要的配置项就是 maxLength 和 minLength，maxLength 定义了允许输入的字符的最大长度，而 minLength 定义了允许输入的字符的最小长度。以下是第三个字段的定义代码：

```
{xtype:"textfield",fieldLabel:" 字段 3",name:"field3",
   enforceMaxLength:10,maxLength:10,minLength:5,
},
```

字段的验证方式有三种，第一种是使用 vtype 配置项，这个很简单；第二种是使用 regex、regexText 和 maskRe 这 3 个配置项；第三种是重写 validator 方法。这里只测试前两种方法。下面定义两个字段，分别使用这两种方法实现“必须输入中文”的验证，代码如下：

```
{xtype:"textfield",fieldLabel:"字段 4",name:"field4",
   vtype:"chinese"
},
{xtype:"textfield",fieldLabel:"字段 5",name:"field5",
   regex:/^[\u4e00-\u9fa5]+$/i,
   regexText:'必须输入中文。',maskRe:/[\W]/i
},
```

从代码中可以看到，regex 就是用来验证值的正则表达式，与 VTypes 对象的验证函数作用一样，regexText 对应的就是错误信息文本，而 maskRe 对应的就是掩码。

最后三个字段分别用来测试只读的字段（readOnly 为 true）、禁用的字段（disabled 为 true）和字段获取焦点后，选中字段中的文本（selectOnFocus 为 true），代码如下：

```
{xtype:"textfield",fieldLabel:"字段 6",name:"field6",
   readOnly:true,value:"只读的字段"
},
{xtype:"textfield",fieldLabel:"字段 7",name:"field7",
   disabled:true,value:"禁用的字段"
},
{xtype:"textfield",fieldLabel:"字段 8",name:"field8",
   selectOnFocus:true,value:"selectOnFocus"
}
```

完成示例后，在浏览器中打开页面后，单击进入字段 1；直接按 Tab 切换到字段 2，并输入 123456；切换到字段 3，输入 1234；再切换到字段 4，输入小数点；切换到字段 5，输入小数点；最后切换到字段 8，将看到如图 12-3 所示的变化。

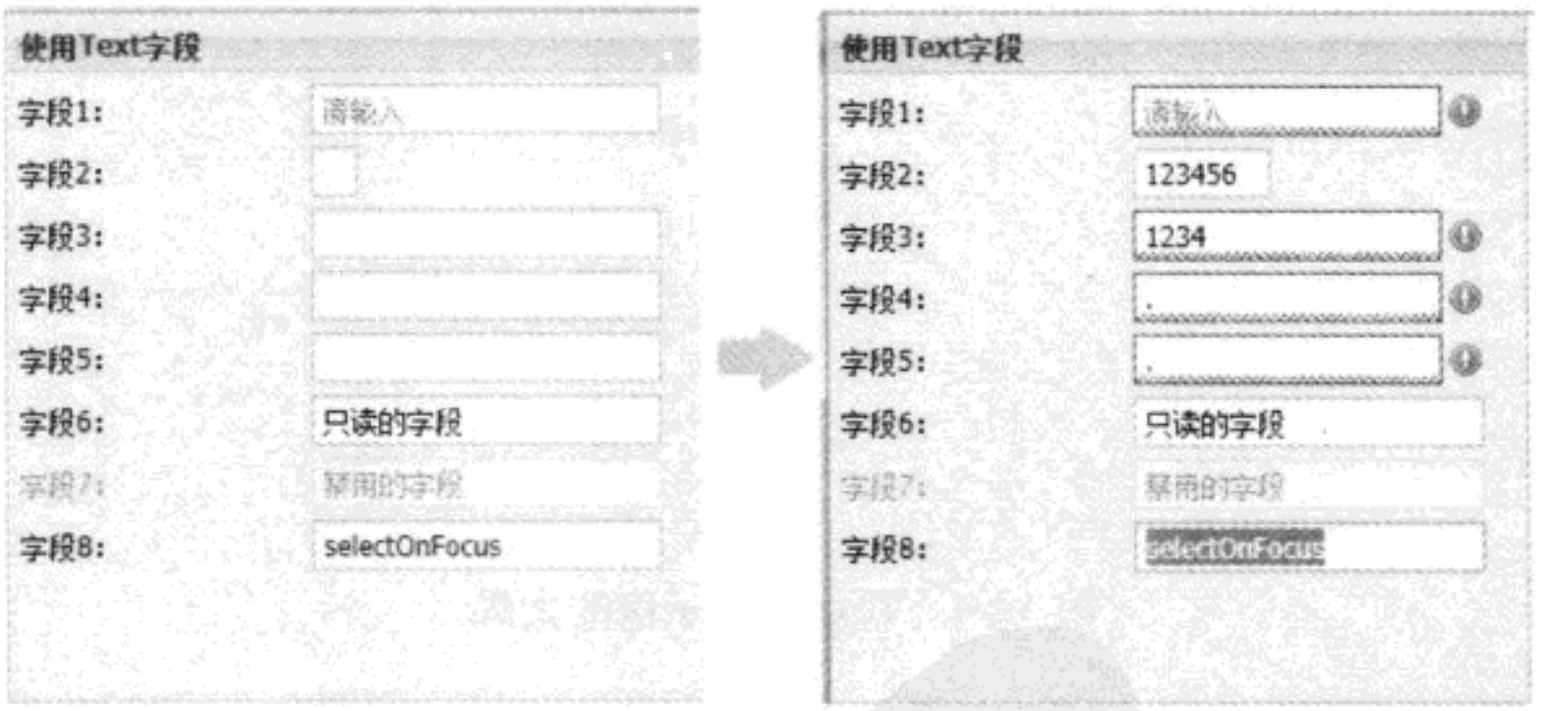


图 12-3　Text 字段的示例效果

从图 12-3 中可以看到，只读字段与普通输入字段没什么区别，容易引起误解，因而最好是定义 readOnlyCls 配置项，让字段在只读状态下如 disabled 状态那样，显示不同的背景颜色。

3. 使用 TextArea 字段

TextArea 字段派生于 Text 字段，具有 Text 字段的特点，不过，它最终的 HTML 标记是 TextArea，因而也具有 TextArea 的特点。标记 TextArea 的特点是可以通过 cols 和 rows 两个属性来控制元素的高度和宽度，因而 TextArea 字段也可以使用配置项 cols 和 rows 来定义高度和宽度。不过不建议这样，因为这两个配置项的宽度和高度很难精确，最好的方式还是使用表单面板默认的锚固布局的配置方式进行配置，最直接的当然是设置 width 和 height，因为这样可以适应布局。

下面通过一个示例来演示一下如何使用两种方式来设置 TextArea 的宽度和高度。复制文件 12-1.html 并将复制后的文件名修改为 12-2.html，然后将 items 中的代码全部删除，加入以下代码：

```
{xtype:"textareafield",fieldLabel:"字段 1",name:"field1",
   rows:3,cols:20,allowBlank:false
},
{xtype:"textareafield",fieldLabel:"字段 2",name:"field2",
   labelAlign:"top",anchor:"0 40%",
   msgTarget:"under",allowBlank:false
}
```

字段 1 使用了 rows 和 cols 来设置 TextArea 字段的高度和宽度，而字段 2 则使用 anchor 配置项来设置宽度和高度，并且将标签位置改在了顶部，而将错误信息显示在底部。

在页面中打开示例，将焦点移入字段 1，再移入字段 2，然后再移出，可看到如图 12-4 所示的效果。从图 12-4 中可以看到，字段 2 会缩小字段高度以显示错误信息，从而避免了因面板主体高度的改变而造成显示滚动条或隐藏部分组件的问题。

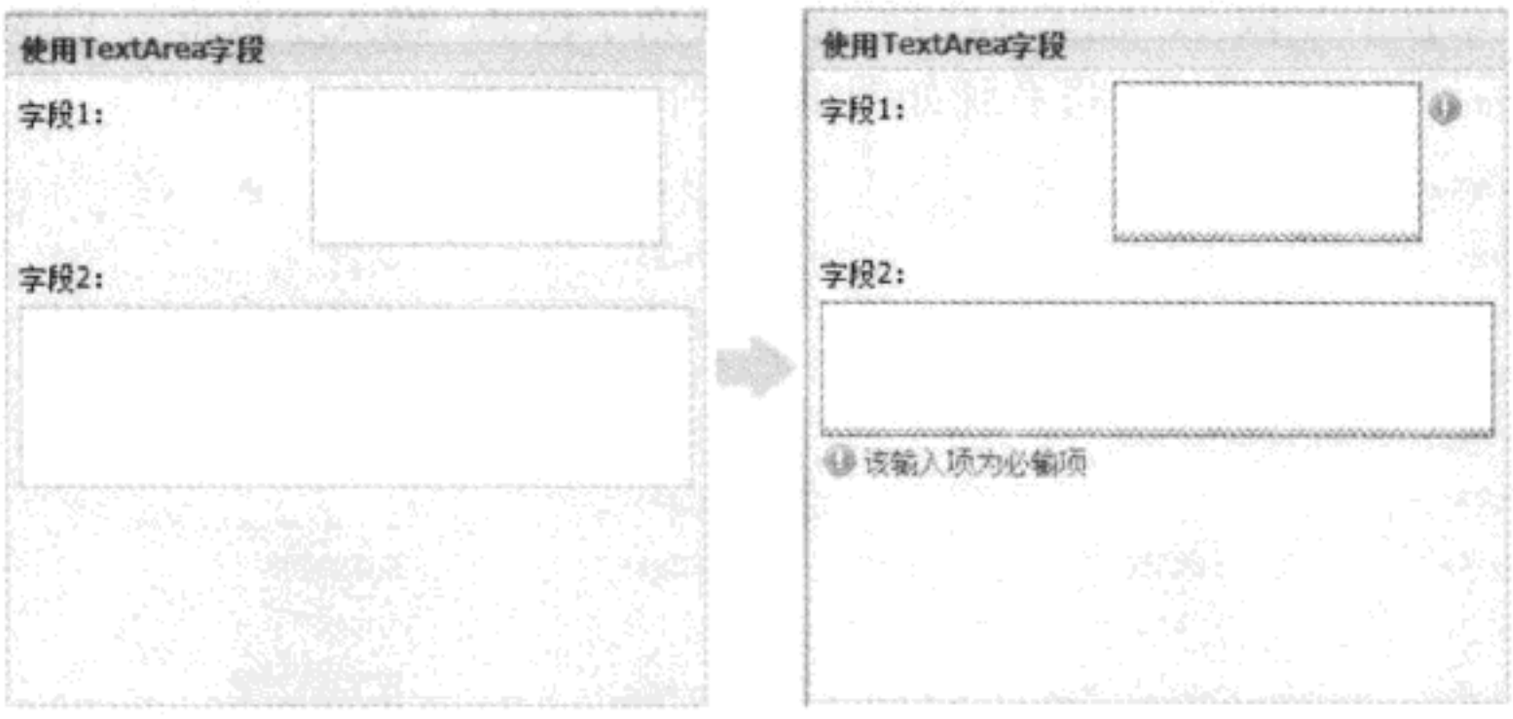


图 12-4　TextArea 示例的效果

4. 使用 Hidden 字段

Hidden 字段实际与 HTML 标记 input 中 type 值为 hidden 的元素功能一样，事实上 Hidden 字段最终生成的元素也包括 type 值为 hidden 的标记。Hidden 字段与在提交时定义 params 配置项的效果是一样，不过它的唯一好处是可以像其他字段一样使用 setValue 方法设置值，这在某些情况下会比使用 params 配置项来得方便。

使用 Hidden 字段很简单，在此就不做演示了。

5. 使用 Display 字段

Display 字段的作用是显示一个值，该值不可以编辑和提交，仅用于显示。使用 Display 字段的一大好处是可以使用 setValue 方法设置值，而不需要像使用其他显示方式那样，手动配置其 innerHTML 属性。

Display 字段默认是不对值进行 HTML 编码的，因而可显示 HTML 的格式，通过设置 htmlEncode 配置项为 true，就可以对值进行 HTML 编码，而不显示 HTML 的格式。下面通过一个示例演示一下这两种情况。

复制文件 12-1.html 并将复制后的文件名修改为 12-3.html，删除 items 中的代码，并加入以下代码：

```
{xtype:"displayfield",fieldLabel:" 字段 1",name:"field1",
   value:"<b> 带 HTML 格式 <b>"
},
{xtype:"displayfield",fieldLabel:" 字段 2",name:"field2",
   htmlEncode:true,value:"<b> 不带 HTML 格式 <b>"
}
```

运行以上代码后，将看到如图 12-5 所示的效果，因为没有对字段编码，所以会以粗体文字格式显示，而对字段 2 进行了编码，所以 HTML 标记 b 也作为显示的一部分了。

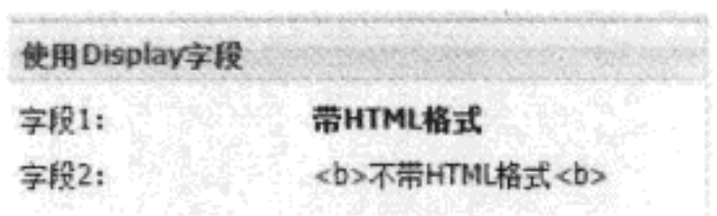


图 12-5　Display 字段的示例效果

6. 使用 FieldContainer

FieldContainer 组件派生于 Container 对象，混合了 Labelable 功能，因而可为其设置标签。该组件可以将字段组合在一个标签下显示。下面通过一个例子演示如何使用 FieldContainer 组件。

复制文件 12-1.html 并将复制后的文件名修改为 12-4.htmi，删除 items 中的代码，并加入以下代码：

```
{xtype:"fieldcontainer",fieldLabel:" 电话 ",
   defaultType: 'textfield',
   items:[
          {name:"field1",width:40,
                 emptyText:" 区号 ",maskRe:/[\d]/,reegx:/[\d{3,4}]/,
                 minLength:3,maxLength:4,regexText:" 请输入正确的区号 "
          },
          {name:"field2",flex:1,
                 emptyText:" 请输入电话号码 ",maskRe:/[\d]/,reegx:/[\d{7, 8}]/,
                 minLength:7,maxLength:8,regexText:" 请输入正确的电话号码 "
          }
   ]
},
{xtype:"fieldcontainer",fieldLabel:" 电话 ",
   defaultType: 'textfield',layout: 'hbox',
   combineErrors:true,
   items:[
          {name:"field1",width:40,
```

```
                emptyText:" 区号 ",maskRe:/[\d]/,reegx:/[\d{3,4}]/,
                minLength:3,maxLength:4,regexText:" 请输入正确的区号 "
            },
            {xtype:"label",text:"-"},
            {name:"field2",flex:1,
                emptyText:" 请输入电话号码 ",maskRe:/[\d]/,reegx:/[\d{7, 8}]/,
                minLength:7,maxLength:8,regexText:" 请输入正确的电话号码 "
            }
        ]
    }
```

以上代码定义了两个 FieldContainer 组件，第一个使用默认的锚固布局，因此其内部字段会从上到下依次显示。第二个则使用了水平布局，这样就可以把字段都放在一行。这里使用了标签插入了一个分隔字符“-”。因为定义了配置项 combineErrors 为 true，所以其内部字段错误信息就会合并在一起显示。配置项 defaultType 的作用是，在没有定义 xtype 时，在 items 中定义的字段默认使用 textfield。

在浏览器中打开页面，并在每个字段内输入 1，将看到如图 12-6 所示的效果。因为没有合并错误，所以不会显示错误的小图标，只能通过提示信息查看错误。

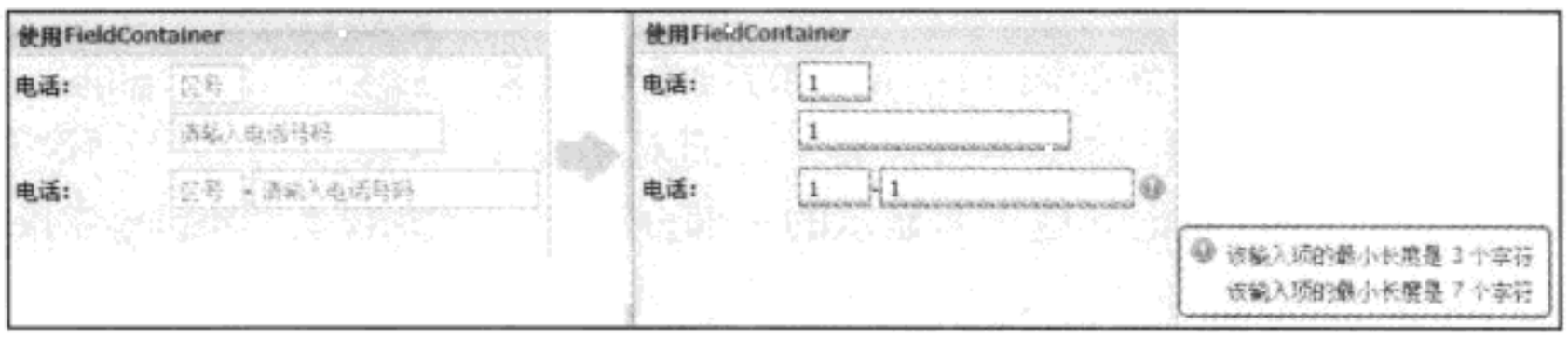


图 12-6　FieldContainer 组件示例的效果

7. 使用 FieldSet

FieldSet 与 FieldContainer 一样，可分组字段。FieldSet 的特点是使用 HTML 元素 fieldset 来进行分组，通过 collapsible 或 checkboxToggle 配置项可实现 FieldSet 的折叠。collapsible 配置项与 checkboxToggle 配置项的区别是，checkboxToggle 可提交，提交的参数名称可通过 checkboxName 配置项进行定义，这样的好处是，当服务器端检查到该组被隐藏的时候，可以不处理该组的数据，当然，这是根据设计进行的。下面通过一个示例来演示如何使用 FieldSet。

复制文件 12-1.html 并将复制后的文件名修改为 12-5.html，删除 items 中的代码，并加入以下代码：

```
{xtype:"fieldset",title: ' 可折叠的 Fieldset ',
    collapsible: true,
    defaultType:"textfield",
    items:[
            {fieldLabel:" 字段 1",name:"field1"},
            {fieldLabel:" 字段 2",name:"field2"}
    ]
},
{xtype:"fieldset",title: ' 带复选框的 Fieldset ',
```

```
        checkboxToggle:true,
        defaultType:"textfield",
        items:[
                {fieldLabel:"字段 3",name:"field3"},
                {fieldLabel:"字段 4",name:"field4"}
        ]
}
```

代码只是定义了两个 FieldSet，第一个使用了 collapsible 配置项进行折叠，第二个使用 checkboxToggle 配置项进行折叠。当然，折叠不是必须的，不定义这两个配置项，也可固定 FieldSet。

在浏览器中打开页面，折叠两个 FieldSet 后将看到图 12-7 所示的效果。

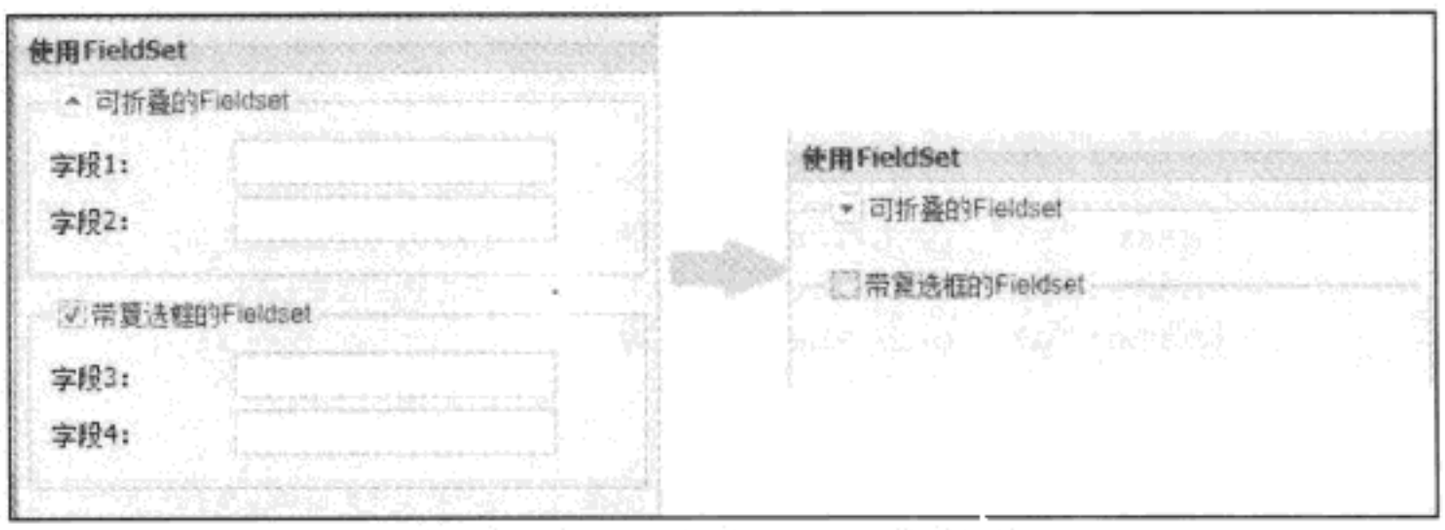


图 12-7　FieldSet 示例的效果

8. 使用 Checkbox 字段与 CheckboxGroup

Checkbox 字段并不是标准的 HTML 复选框，而是使用按钮通过背景图片模拟出的复选框效果，因而不要将标准的 HTML 复选框的属性套用在 Checkbox 字段上。

CheckboxGroup 派生于 FieldContainer，因而具有 FieldContainer 的特性，其主要作用是分组 Checkbox 字段，可以通过定义 allowBlank 配置项为 false 来要求必须选择一个选项。通过其重写过的 getChecked、getValue 和 setValue 方法可快捷地操作组内的 Checkbox 字段。

下面通过一个示例演示一下如何使用 Checkbox 字段、FieldContainer 与 CheckboxGroup 来分组 Checkbox 字段的差别。

复制文件 12-1.html 并将复制后的文件名修改为 12-6.html，删除 items 中的代码。

首先加入一个名称为 c1 的 Checkbox 字段和一个名称为 field1 的 Text 字段，通过 c1 可启用和禁用 filed1，代码如下：

```
{xtype:"checkbox",boxLabel:'禁用或启用字段 1',name:"c1",
    checked:true,handler:function(f,checked){
            this.up("form").getForm().findField("field1").setDisabled(!checked);
    }
},
{xtype:"textfield",fieldLabel:"字段 1",name:"field1"},
```

在以上代码中，配置项 checked 为 true，可初始化 Checkbox 字段的初始状态为选择状态，而由配置项 handler 定义的函数会在 Checkbox 字段状态改变时触发，这时候就可根据传递过来的 checked 值禁用或启用 field1 了。

接着加入一个分隔条：

```
{xtype:"panel",anchor:"0",heihgt:0},
```

使用 0 高度的面板做分隔条比使用 Display 字段做分隔条方便，因为使用 Display 很难调宽度，而面板比较简单，还有面板的边是天然的分隔条，不需要做其他设置。

接着要定义一个使用 FieldContainer 分组的 Checkbox 字段组。要使用 FieldContainer 做分组，必须利用布局将 FieldContainer 分成需要的列。布局可使用水平布局或列布局，使用水平布局会简单很多，本示例使用的就是水平布局。具体代码如下：

```
{xtype:"fieldcontainer",fieldLabel:"权限",
   layout:"hbox",heigh:90,
   defaults:{xtype:"panel",bodyStyle:"background:#DFE9F6",
          border:false,flex:1,defaultType:'checkbox'
   },
   items:[
          {items:[
                 {name:"p1",boxLabel:"管理员",inputValue:1},
                 {name:"p1",boxLabel:"员工",inputValue:2},
                 {name:"p1",boxLabel:"采购员",inputValue:3}
          ]},
          {items:[
                 {name:"p1",boxLabel:"仓管员",inputValue:4},
                 {name:"p1",boxLabel:"经理",inputValue:5},
                 {name:"p1",boxLabel:"总经理",inputValue:6}
          ]}
   ]
},
```

因为使用水平布局，所以必须先用面板划分好布局，再在面板中放置 Checkbox 字段。两个面板的定义是一样的，因而可使用配置项 defaults 对面板进行定义，这些配置会自动应用于 FieldContainer 中 items 配置项定义的子组件，这样在子组件中只需要定义 items 配置项就可以了。

为了方便全部选择或取消选择 FieldContainer 中的 Checkbox 字段，在 FieldContainer 的配置对象前添加一个 Checkbox 字段，其代码如下：

```
{xtype:"checkbox",boxLabel:'选择全部',
   handler:function(f,checked){
          var els=this.up("form").query("[name=p1]");
          for(var i=els.length-1;i>=0;i--){
                 els[i].setValue(checked)
          }
   },
```

因为 FieldContainer 中没有统一操作子组件的方法，所以这里只能通过 query 方法查找出全部 name 属性为 p1 的组件，然后再根据返回的 checked 值设置其状态。

再加入一条分隔线，然后加入使用 CheckboxGroup 分组的 CheckBox 字段，其代码如下：

```
{xtype:"checkboxgroup",columns:2,fieldLabel:"权限",
```

```
vertical:true,allowBlank:false,
blankText:"必须选择一个选择",
items:[
        {name:"p2",boxLabel:"管理员",inputValue:1},
        {name:"p2",boxLabel:"员工",inputValue:2},
        {name:"p2",boxLabel:"采购员",inputValue:3},
        {name:"p2",boxLabel:"仓管员",inputValue:4},
        {name:"p2",boxLabel:"经理",inputValue:5},
        {name:"p2",boxLabel:"总经理",inputValue:6}

]
}
```

CheckboxGroup 的定义比 FieldContainer 的定义简单多了，定义配置项 columns 就可设置将 CheckBox 字段分成几列，这里分成了两列。配置项 vertical 可以决定如何将 CheckBox 字段加入到每一列中，vertical 为 true 时，会先填满一列，再去填充下一列；而为 false 时，则是从左到右依次填充完一行，再填充下一行。配置项 allowBlank 为 false 说明必须选择一个 CheckBox 字段。

使用 CheckboxGroup 进行全部选择或取消选择也很简单，具体代码如下：

```
{xtype:"checkbox",boxLabel:'选择全部',
   handler:function(f,checked){
           var cg=this.up("form").query("checkboxgroup")[0];
           cg.setValue({p2:checked});
   }
},
```

只要使用 setValue 方法设置值就可以了。值是一个对象，在对象内，关键字为 CheckBox 字段 name 配置项的值，这里是 p2，然后值是函数接收的参数 checked。

最后在表单面板的底部工具栏内定义两个按钮，第一个按钮用来对两个 CheckBox 组进行管理员、采购员和总经理这 3 项选择。第二个按钮则用来提交表单。熟悉一下 CheckBox 是如何提交值的，代码如下：

```
bbar:[
   {text:"选择管理员、采购员和总经理",handler:function(){
           var selVal=[1,3,6],
                   fp=this.up("form"),
                   p1=fp.query("[name=p1]"),
                   cg=fp.query("checkboxgroup")[0];
                   for(var i=p1.length-1;i>=0;i--){
                           var el=p1[i];
                           if(selVal.indexOf(el.inputValue)>=0)
                                   el.setValue(true);
                   }
                   cg.setValue({p2:selVal});
   }},
   {text:"保存",handler:function(){
           this.up("form").getForm().submit();
   }}
]
```

在第一个按钮的句柄中，FieldContainer 这组选择具体值，和全部选择一样，需要通过循环一个一个地判断 CheckBox 字段的 inputValue 是否是要选择的选项，如果是，设置其值为 true。而 CheckboxGroup 就简单多了，将要选择的值组合成一个数组，然后就可以使用 setValue 方法进行选择了。这里要注意了，不能像全部选择那样，使用 true 作为值，而是需要使用具体且能区分各选项的提交值（inputValue）。

提交按钮的代码很简单，调用 BasicForm 的 sumit 方法就行了。

在浏览器中打开页面，单击“选择管理员、采购员和总经理”按钮，再单击“保存”按钮，会看到 Firebug 控制台发送了一个请求，请求是错误的，这个不用管，展开请求，在 POST 标签页中可看到提交的参数为：

```
c1 on
field1
p1 1
p1 3
p1 6
p2 1
p2 3
p2 6
```

因为字段 c1 没设置 inputValue，所以它会提交默认值 on。而两个分组的 CheckBox 字段则以标准的 HTML 复选框格式提交了参数，也就是说，可在服务器端使用标准的表单提交方式获取 CheckBox 字段值。例如，在服务器利用参数 p1 就可获取以逗号分隔的提交值，示例中的值会是“1,3,6”。

最后单击字段 c1、两个“选择全部”字段，再执行以上操作将看到如图 12-8 所示的效果变化。

9. 使用 Radio 字段与 RadioGroup

Radio 字段派生于 CheckBox 字段，只是改变了显示的样式，将图片更换为单选按钮，并且将设置值、取值等方法根据 Radio 字段的特性做了修改。总体上，主要接口并没有改变。

RadioGroup 派生于 CheckboxGroup，因而它的使用与 CheckboxGroup 类似。

下面通过一个示例演示如何使用 Radio 字段与 RadioGroup。

复制文件 12-5.html 并将复制后的文件名修改为 12-7.html，删除 items 中的代码。

首先加入一个使用 FieldContainer 分组的 Radio 字段组，代码如下：

```
{xtype:"fieldcontainer",fieldLabel:"性别",
   layout:"hbox",defaults:{xtype:"radio",flex:1},
   items:[
          {name:"r1",boxLabel:"男",inputValue:1},
          {name:"r1",boxLabel:"女",inputValue:2},
   ]
},
```

这里只有两个选项，所以不需要使用面板先划分区域，只要直接创建 Radio 字段就行了。

然后加入一个使用 RadioGroup 分组的 Radio 字段组，代码如下：

```
{xtype:"radiogroup",columns:2,fieldLabel:"性别",
```

```
    allowBlank:false,blankText:"必须选择一个选择",
    items:[
            {name:"r2",boxLabel:"男",inputValue:1},
            {name:"r2",boxLabel:"女",inputValue:2}
    ]
}
```

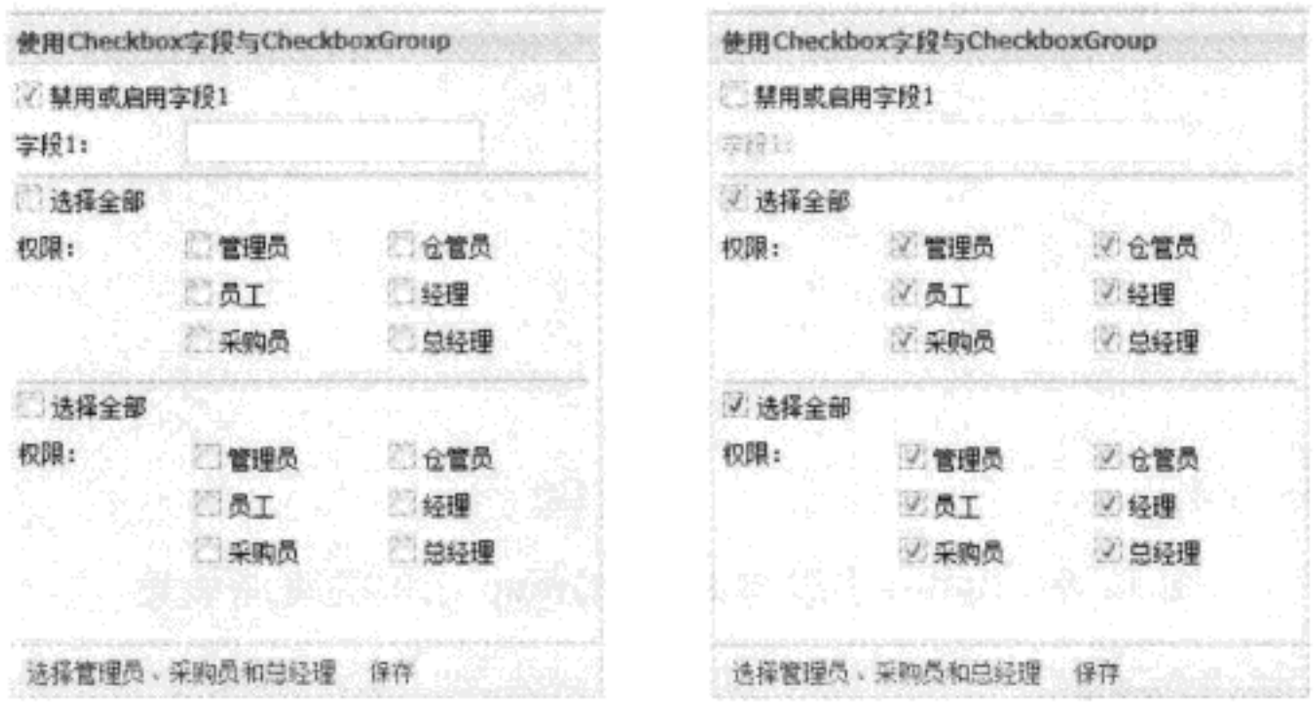


图 12-8　Checkbox 字段与 CheckboxGroup 示例的页面效果

可以看到，RadioGroup 与 CheckboxGroup 的定义区别不大。

在底部工具栏，重新定义选择按钮，提交按钮不需要改变。修改的代码如下：

```
{text:"选择女",handler:function(){
    var selVal=2,
            fp=this.up("form"),
            r1=fp.query("[name=r1]"),
            cg=fp.query("radiogroup")[0];
            for(var i=r1.length-1;i>=0;i--){
                var el=r1[i];
                if(selVal==el.inputValue)
                        el.setValue(true);
            }
            cg.setValue({r2:selVal});
}},
```

与示例 12-6 的选择按钮比较，代码改变不多，Radio 字段是单选的，所以可选择的值只能是 1 个。设置值的过程是一样的。

在浏览器中打开示例，并单击选择按钮，将看到如图 12-9 所示的效果。

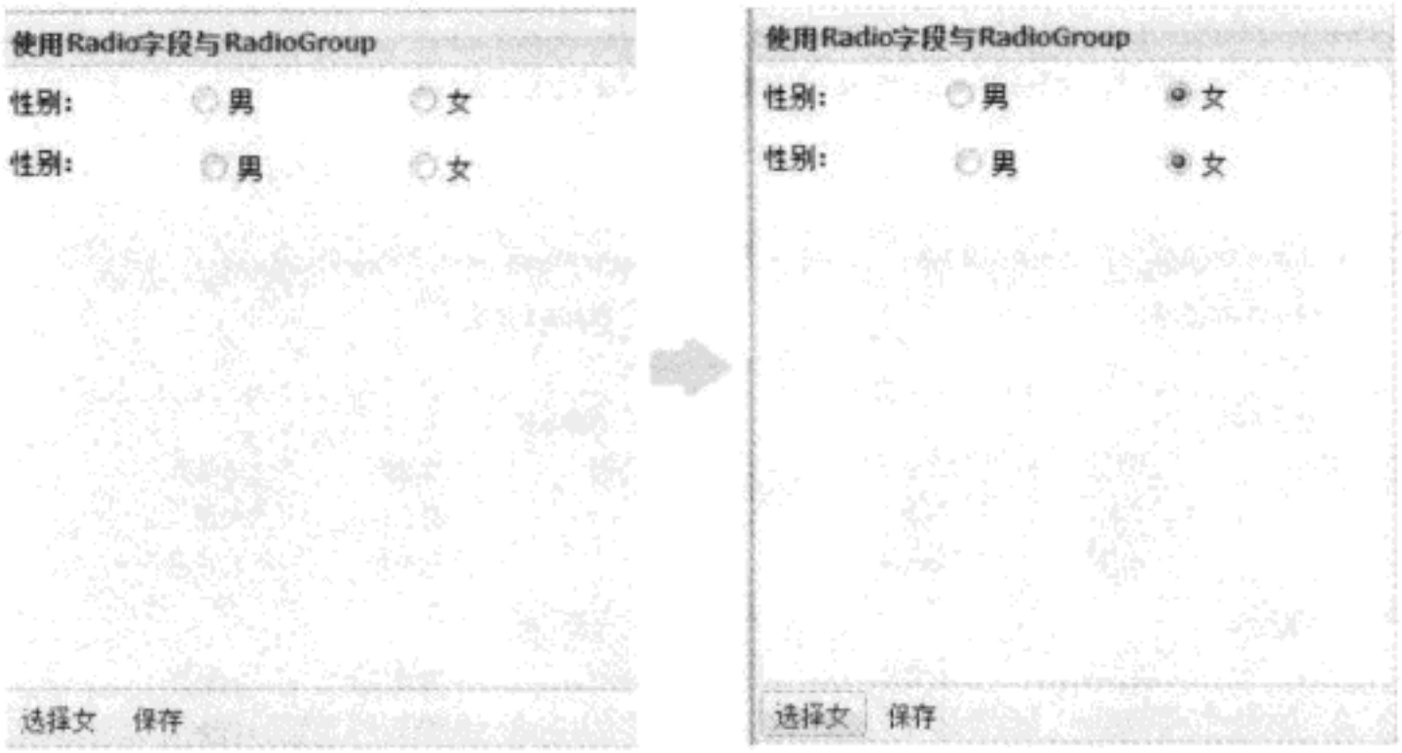


图 12-9　Radio 字段与 RadioGroup 示例的页面效果

10. 使用 HtmlEditor 组件

HtmlEditor 组件派生于 Component 组件，是一个轻型的 HTML 编辑组件，在组件内混合了 Labelable 和 Field 功能，所以该组件具有标签和表单字段的功能。下面通过一个示例演示如何使用 HtmlEditor 组件。

复制文件 12-5.html 并将复制后的文件名修改为 12-8.html，删除 items 中的代码，删除底部工具栏的选择按钮，保留保存按钮，然后在 items 中加入以下代码：

```
{xtype:"textfield",fieldLabel:"标题",anchor:"0",name:"title"},
{xtype:"htmleditor",fieldLabel:"内容",name:"content",
   labelAlign:"top",anchor:"0 -30",msgTarget:"under"
}
```

以上代码定义了一个 Text 字段，一个 HtmlEditor。可以看到编辑器和 Text 字段一样，可以在其中定义标签和 name 配置项。配置项 labelAlign 为 top，可将标签放置到编辑器顶部。而 msgTarget 则将错误信息放在了编辑器底部。

现在来完成保存按钮的代码，在完成前要注意，HtmlEditor 还不完善，因此字段验证功能基本没有，需要在提交前再做验证。一个更重要的问题是，为了防止在浏览器中出现编辑器没有焦点的问题，在对编辑器初始化的时候，默认加入一个没有宽度的空白字符作为值，因此不能使用验证空白值的方式验证值是否为空，而需要通过判断值的长度来决定值是否空值，如果长度大于 1，说明有其他字符，不为空，如果值为 1，说明是默认的空白字符，值实际为空。在 HtmlEditor 中，没有重写 markInvalid 和 clearInvalid 方法，因而如果要显示错误信息，必须先在页面 OnReady 函数先加入以下代码：

```
if(Ext.form.field.HtmlEditor){
   Ext.apply(Ext.form.field.HtmlEditor.prototype,{
         // 在此粘贴 markInvalid 和 clearInvalid 方法
   })
}
```

然后在 BaseField 对象的类文件中复制 markInvalid 和 clearInvalid 方法，然后粘贴在代码的注释下。

现在来完成保存按钮的代码：

```
{text:"保存",handler:function(){
    var f=this.up("form").getForm(),
            content=f.findField("content"),
            v=content.getValue();
    if(v.length>1){
            content.clearInvalid();
            if(f.isValid())
                    f.submit();
    }else{
            content.markInvalid("请输入内容!")
    }
}}
```

首先要找到编辑器，然后使用 getValue 方法的返回值，如果其值大于 1，说明不是空值，清理一下编辑器的错误显示，然后调用 isValid 验证其他字段是否有效，如果有效，则提交数据。

如果 getValue 的返回值是空值，则调用 markInvalid 方法显示错误。

在浏览器中打开页面，然后单击“保存”，将看到如图 12-10 所示的效果。

图 12-10　HtmlEditor 示例的页面效果

11. 使用 File 字段

File 字段用于上传文件，它会将一个按钮遮盖在用于上传文件的 input 标记的按钮之上，从而利用按钮的特性，实现标准方式、只有按钮和使用图标按钮 3 种显示形式。

从 SubmitAction 的运作流程可以知，上传文件是以标准的表单方式提交的，会被提交到 iframe 中，返回数据是从 iframe 的 body 中读取的，因而返回结果虽然是 JSON 对象，但是必须以文本形式显示在页面的 body 中，也就是说返回的页面的 contentType 必须是“text/html”或“text/xml”，而不能是“text/javascript”，这点要谨记，不然脚本运行会出错。

下面用一个示例来演示如何使用 3 种显示形式上传文件。先使用模板创建一个名称为 12-9.html 的页面文件。因为图标按钮需要使用样式实现，所有先定义一个应用于按钮的样式，代码如下：

```
    .upload{background: url("../../images/upload.png") no-repeat scroll 0 0
transparent }
```

下面先定义一个面板，其中包含 3 个表单面板，分别用于显示不同形式的 File 字段，还有一个 Img 对象用于显示上传的图片，代码如下：

```
Ext.create("Ext.panel.Panel",{
   title:"使用 File 字段",
   width:300,
   height:600,
   renderTo:Ext.getBody(),
   bodyStyle:"background:#DFE9F6",
layout:{type:"vbox",align:"stretch"},,
defaults:{xtype:"form",bodyPadding:5,
bodyStyle:"background:#DFE9F6",height:90,
url:"Upload.ashx",
defaultType:"filefield",
fieldDefaults:{labelWidth:80,msgTarget:"side",
       labelSeparator:": ",allowBlank:false,anchor:"0"
},
bbar:[
       {text:"保存",handler:function(){
       var f=this.up("form").getForm();
       if(f.isValid())
              f.submit({
              success:function(f,s){
                     var img=Ext.getCmp("img");
                     img.setSrc(s.result.msg);
              }
              });
       }}
       ]
},
items:[
{title:"标准",items:[
       {fieldLabel:"上传文件",buttonText:"选择",name:"Filedata"}
]},
{title:"只有按钮",height:120,items:[
       {buttonText:"选择",name:"Filedata",buttonOnly:true,
```

```
            listeners:{
                    change:function(f,v){
                            Ext.getCmp("selectFile").setText(v);
                    }
            }
    },
    {xtype:"label",id:"selectFile"}
    ]},
    {title:"图片按钮",items:[
            {fieldLabel:"上传文件",name:"Filedata",buttonText:"",
                    buttonConfig:{iconCls:"upload"}
                    }
            ]},
            {xtype:"image",id:"img",height:250}   ]
})
```

在这里要注意布局的定义，配置项 align 必须定义，不然显示会有问题。

因为 3 个表单面板是一样的，所以可以在配置项 defaults 中统一定义，底部工具栏的“保存”按钮也是可以一起定义的。基本上，所有配置项都可以在配置项 defaults 中进行定义，不同的部分可在 items 中重写，例如第二个表单面板，因为是只有按钮的 File 字段，所以加了个标签用来显示上传的文件，高度自然要调高点，所以需要在面板表单的定义中重定义 height 配置项。

对于第一个 File 字段，因为是标准样式的，所以只定义 buttonText 配置项作为按钮的文本就可以了。对于第二个 File 字段，只需要按钮，因此要定义配置项 buttonOnly 为 true，并设置显示文本。当然，也可以通过 buttonConfig 配置样式，只显示图标。对于第三个 File 字段，使用 buttonConfig 配置项，按钮的配置对象，配置按钮只显示图标，这里必须定义 buttonText 为空字符串，不然就显示默认的文本了。

写好客户端代码了，现在开始写服务器端代码，这对有开发经验的人来说不会太难，代码如下：

C#

```
public void ProcessRequest (HttpContext context) {
    string output = "";
    System.Drawing.Image original_image = null;
    System.Drawing.Bitmap final_image = null;
    System.Drawing.Graphics graphic = null;
    try
    {
        HttpPostedFile jpeg_image_upload = context.Request.Files["Filedata"];
        string original_fielname = jpeg_image_upload.FileName.ToLower();
string extname = original_fielname.Substring(original_fielname.
    LastIndexOf(".") + 1).ToLower();
if (",jpg,gif,bmp,png,".IndexOf(extname) >= 0)
{
    Guid guid = Guid.NewGuid();
    string filename = guid.ToString() + "." + extname;
    string path = context.Server.MapPath(".")+"\\";
    jpeg_image_upload.SaveAs(path+filename);
    output = new JObject(
```

```
                    new JProperty("success", true),
                    new JProperty("msg", filename)
                ).ToString();

        }
        else
        {
            output = new JObject(
                    new JProperty("success", false),
                    new JProperty("errors", new JObject(
                        new JProperty("Filedata", "错误的图片类型。")
                    ))
                ).ToString();
        }
    }
    catch (Exception ee)
    {
        output = new JObject(
                new JProperty("success", false),
                new JProperty("errors", new JObject(
                    new JProperty("Filedata", ee.Message)
                ))
            ).ToString();
    }
        finally
        {
            if (final_image != null) final_image.Dispose();
            if (graphic != null) graphic.Dispose();
            if (original_image != null) original_image.Dispose();
        }
    context.Response.ContentType = "text/html; charset=utf-8";
     context.Response.Write(output);
}
```

Java

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
   UUID uuid =UUID.randomUUID();
   JsonObject jo =new JsonObject();
   DiskFileItemFactory fa = new DiskFileItemFactory();
   ServletFileUpload sfu=new ServletFileUpload(fa);

   try {
           List<FileItem> list =  sfu.parseRequest(request);
           FileItem fileitem =null;
           for(FileItem item : list){
       if (!item.isFormField()){
               fileitem=item;
               break;
       }
}
String name = fileitem.getName();
String extname = name.substring(name.lastIndexOf(".")+1).toLowerCase();
if(",jpg,gif,png,bmp,".indexOf(","+extname+",")>=0)
```

```
{
        String filename = uuid+"."+extname;
        String  loadpath=request.getSession().getServletContext().
            getRealPath("/");
        File fNew= new File(loadpath,filename);
        fileitem.write(fNew);
        jo.addProperty("success", new JsonPrimitive(true));
        jo.addProperty("msg", filename);
}else{
        jo.addProperty("success", new JsonPrimitive(false));
        JsonObject j1=new JsonObject();
        j1.addProperty("Filedata", "错误的文件类型。");
        jo.add("errors",j1);
    }

}catch (FileUploadException e) {
        jo.addProperty("success", new JsonPrimitive(false));
        JsonObject j1=new JsonObject();
        j1.addProperty("Filedata", e.getMessage());
        jo.add("errors",j1);
} catch (Exception e) {
        jo.addProperty("success", new JsonPrimitive(false));
        JsonObject j1=new JsonObject();
        j1.addProperty("Filedata", e.getMessage());
        jo.add("errors",j1);
}
response.setContentType("text/html; charset=utf-8");
response.getWriter().write(jo.toString());
}
```

执行以上代码会以 Guid 作为文件名保存图片，因此需要提取到文件的扩展名，如果文件的扩展名不是 jpg、gid、png 或 bmp，则会返回一个错误信息。下面是错误信息的格式：

```
{
    success:false,
    errors:{
            field1:error_message1,
            field2:error_message2,
            ...
    }
}
```

在以上格式中，success 用于判断这次的操作是否成功，true 表示操作成功，false 表示操作不成功。errors 属性指向的对象可包含具体字段的错误信息，例如本示例的提交字段的 name 配置项是 Filedata，因而在返回其错误信息的时候，要把 Filedata 作为属性名称，错误信息作为其值，这样，当错误信息返回到客户端的时候，就可直接显示出来。

若文件保存成功，则可以将 success 的值设置为 true，而额外的信息则根据自己喜好进行定义并返回。例如示例中使用 msg 属性返回文件名，因而在客户端保存按钮调用 Submit 方法的 success 函数中，可通过返回对象的 result 属性的 msg 属性获取返回的文件名。

最后，要注意返回的 ContentType，示例返回的是“text/html”。

至此，示例就完成了。在浏览器中打开页面，然后使用第一个 File 字段上传一个脚本文

件，再使用第二个 File 字段上传一个图片文件，将看到如图 12-11 所示的效果。

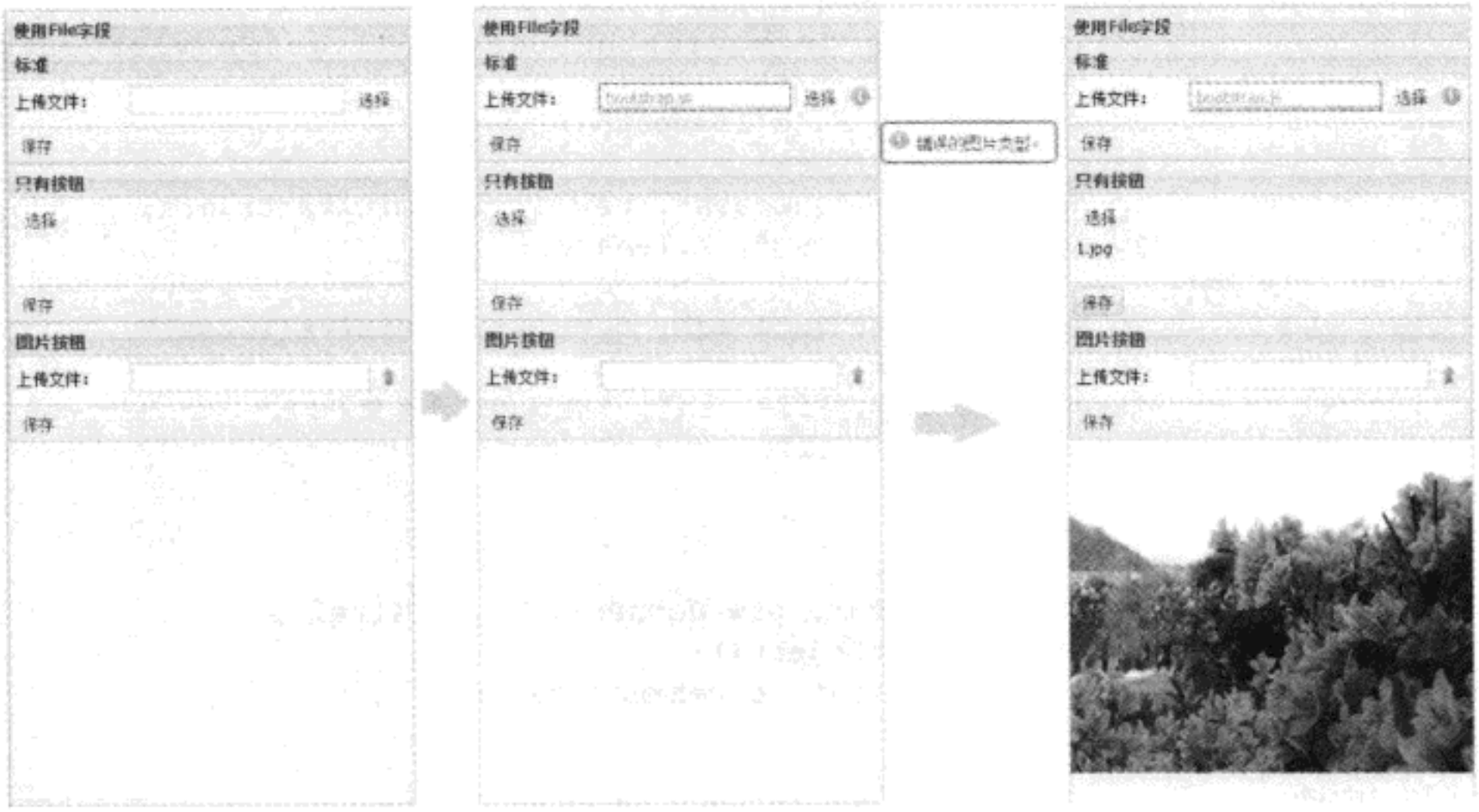


图 12-11　File 字段的示例的页面效果

12.3　使用 Trigger 类字段

12.3.1　具有单击功能的字段：Ext.form.field.Trigger

Trigger 字段派生于 Text 字段，因而具有 Text 字段的全部特性，它通过在 input 标记右边放置一个有背景图片的 div 实现单击操作，从而模拟出下拉或其他效果。它主要提供单击功能，进而派生出 Picker、Spinner 这两类字段，为这两类字段提供接口。以下是 Trigger 字段提供的配置项、属性和方法。

1. 配置项

- editable：布尔值，默认为 true，表示字段允许编辑，如果设置其为 false，则不会设置字段内的 input 标记的 readonly 属性为 true，禁止编辑 input 内的值。
- hideTrigger：布尔值，当值为 true 时，会隐藏带单击功能的元素，只显示 input 标记。默认值为 false。
- readOnly：布尔值，如果值为 true，会禁止修改字段，并隐藏带单击功能的元素，取代 editable 和 hideTrigger 配置项。默认值为 false。
- repeatTriggerClick：布尔值，如果设置为 true，会为单击元素绑定 ClickRepeater 对象。默认值为 false。
- triggerBaseCls：应用到单击元素的基本样式类名。默认值为“x-form-trigger”。
- triggerCls：追加到单击元素的样式类名。默认值为 undefined。
- triggerWrapCls：应用到封装单击元素的 div 的样式类名。默认值为“x-form-trigger-

wrap”。

2. 属性

- triggerEl：指向单击元素。
- triggerWrap：指向封装单击元素的 div。

3. 方法

- onTriggerClick：可以通过重写该方法实现单击功能，如弹出下拉列表。
- setEditable：开启或禁用字段的编辑功能。
- setReadOnly：设置字段的状态为只读或取消只读状态。

12.3.2　实现微调功能的 Spinner 字段

Spinner 字段派生于 Trigger 字段，它把 Trigger 字段的下拉按钮修改为上、下两个箭头按钮，从而实现微调功能。

Spinner 字段一般不直接使用，因为它只提供了微调的功能，没有具体实现两个按钮的功能，所以要使用 Spinner 字段，需要像 NumberField 字段那样进行扩展，重写 onSpinUp 和 onSpinDown 方法。

以下是 Spinner 的字段提供的配置项、属性、方法和事件。

1. 配置项

- keyNavEnabled ：布尔值，默认值为 true，开启键盘导航功能，可以使用上下箭头键进行微调。
- mouseWheelEnabled：布尔值，默认值为 true，可以使用键盘滚动进行微调。
- spinDownEnabled ：布尔值，默认值为 true，启用微调的下按钮，如果设置为 false，禁用微调的下按钮。
- spinUpEnabled ：布尔值，默认值为 true，启用微调的上按钮，如果设置为 false，禁用微调的上按钮。

2. 属性

- onSpinDown：需要将该方法在子类重写以实现微调的下按钮功能。
- onSpinUp：需要将该方法在子类重写以实现微调的上按钮功能。
- spinDownEl：指向微调下按钮的元素。
- spinUpEl：指向微调上按钮的元素。

3. 方法

- setSpinDownEnabled：启用或禁用下按钮。
- setSpinUpEnabled：启用或禁用上按钮。
- spinDown：触发 spin 和 spindown 事件，调用 onSpinDown 方法。
- spinUp：触发 spin 和 spinup 事件，调用 onSpinUp 方法。

4. 事件

- spin：当进行微调操作时会触发该事件。
- spindown：当往下进行微调操作时会触发该事件。
- spinup：当往上进行微调操作时会触发该事件。

12.3.3 使用 NumberField 字段

NumberField 字段派生于 Spinner 字段，主要用于编辑数字。下面通过一个示例演示如何使用 NumberField 字段。

复制文件 12-1.html 并将复制后的文件名修改为 12-10.html，删除 items 中的代码，然后在表单面板中加入以下定义：

```
defaultType:"numberfield",
```

最后向 items 中加入以下代码：

```
{fieldLabel:"没有微调按钮",name:"field1",
   allowBlank:false,hideTrigger:true
},
{fieldLabel:"不允许小数",name:"field2",
   allowDecimals:false,autoStripChars:true
},
{fieldLabel:"带步长",name:"field2",
   allowDecimals:false,autoStripChars:true,
   step:100,value:100
},
{xtype:"fieldcontainer",fieldLabel:"时间",layout:"hbox",
   defaultType:"numberfield",fieldDefaults:{width:50},
   items:[
          {maxValue:23,minValue:0,value:(new Date()).getHours()},
          {xtype:"label",text:":"},
          {maxValue:59,minValue:0,value:(new Date()).getMinutes()},
          {xtype:"label",text:":"},
          {maxValue:59,minValue:0,value:(new Date()).getSeconds()}
   ]
}
```

以上代码定义了 6 个 NumberField 字段。第一个字段定义了 hideTrigger 为 true，隐藏微调按钮。第二个字段定义了 allowDecimals 为 false，只能输入整数，autoStripChars 为 true 表示会自动清除非法字符。第三个字段设置了 step 为 100，表示每微调一次，数值会减 100 或加 100。最后使用 FieldContainer 将 3 个字段组合成一个时间输入字段，在该字段中，使用了 minValue 和 maxValue 来控制数字的输入范围。

在浏览器中打开页面，然后在第一个字段处输入 7.56，在第二个字段处也输入 7.56，往上微调第三个字段，效果如图 12-12 所示。从图 12-12 中可以看到，在第二个字段处输入的小数点被字段去掉了，因为该字段只能是整数，小数点是非法字符。

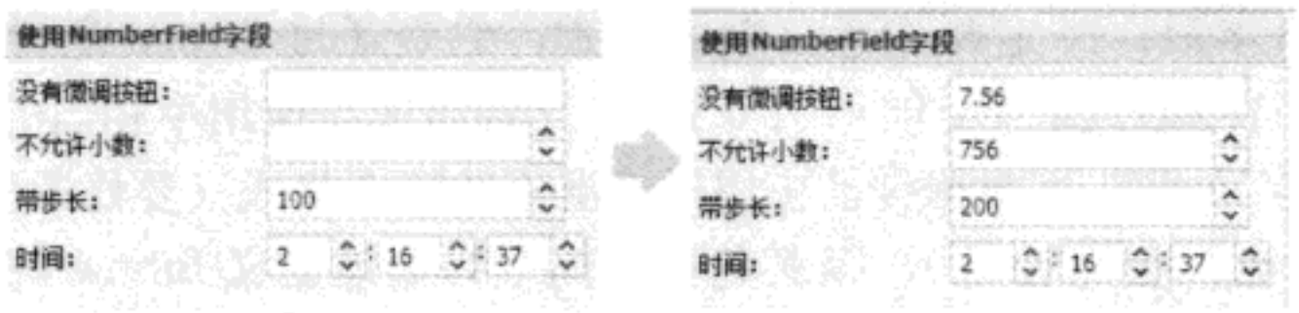


图 12-12　NumberField 示例的页面效果

12.3.4　下拉选择类字段的基类：Ext.form.field.Picker

Picker 字段在 Trigger 字段的基础上添加了单击下拉按钮可关闭与显示选择器的功能，这样其子类就可通过定义不同的选择器实现不同的功能，例如选择日期选择器，就可对日期进行选择。

以下是 Picker 字段的配置项、属性、方法和事件。

1. 配置项

- matchFieldWidth：布尔值，默认值为 true，选择器的宽度会与 input 标记的宽度一样。
- openCls：当显示选择器时，应用于列表 bodyEl 元素的样式，默认值为“x-pickerfield-open”。
- pickerAlign：选择器的对齐方式，默认值为“tl-bl?”。
- pickerOffset：选择器的偏移量，值为包括 x、y 值的数组。该值会附加到 pickerAlign 配置项中。默认值为 undefined。

2. 属性

- createPicker：该方法在子类需要重写时创建选择器。
- isExpanded：如果选择显示状态，那么返回 true，否则返回 false。

3. 方法

- collapse：关闭选择器。
- expand：显示选择器。
- getPicker：返回选择器，如果选择还没创建，调用 createPicker 方法创建选择器。
- onTriggerClick：该方法已被重写，用于关闭或显示选择器。

4. 事件

- collapse：当选择关闭时会触发该事件。
- expamd：当选择器显示时会触发该事件。
- select：当从选择器选择值的时候，会触发该事件。

12.3.5　使用 DateField 字段

DateField 字段派生于 Picker 字段，它的选择器是 DatePicker 对象实例，因而很多配置项是 DatePicker 对象的配置项。

使用 DateField 字段，一定要注意日期的格式，它是通过 format 配置项定义的。在本地化文件中，默认的日期格式是“y 年 m 月 d 日”，这种格式在使用中相当的不方便，笔者的习惯是将其修改为“Y-m-d”。可根据自己应用的情况修改这个格式（要修改的地方有 Ext.picker.Date 和 Ext.form.field.Date 这两处），以避免在应用中频繁地定义 format 配置项。设置好日期格式后，其他配置项如果是使用字符串定义的日期，必须严格按照格式定义，不然在将其转换为日期的时候，会发生错误，例如，在使用配置项 value 设置默认值时，如果日期格式不对，那么是不会显示默认值的。

下面通过一个示例来演示如何使用 DateField。

复制文件 12-10.html 并将复制后的文件名修改为 12-11.html，删除 items 中的代码，然后修改 defaultType 的值为 datefield，接着在 items 中加入以下代码：

```
{fieldLabel:"错误的默认值",name:"field1",
   value:"2011-1-1",allowBlank:false
},
{fieldLabel:"工作日",name:"field2",
   disabledDates:['01-01','05-01',"10-01"],
   disabledDays:[0,6],minValue:"2011-01-01",
   maxValue:'2011-12-31',submitFormat:"m-d",
   value:"2011-01-01"
},
{xtype:"fieldcontainer",fieldLabel:"日期范围",layout:"hbox",
   defaultType:"datefield",fieldDefaults:{width:100},
   items:[
          {name:"start",value:new Date()},
          {xtype:"label",text:"至"},
          {name:"end",value:new Date()}
   ]
}
```

以上代码定义了 4 个 DateField 字段。第一个字段用于设置错误格式的默认值，在打开页面后，将不会显示默认值。第二个字段的配置项很多，主要用于禁用一些日期。配置项用于禁用 disabledDates 设置指定的日期，这里一定要严格按照格式进行定义，例如禁用 10 月 1 日，必须定义为“10-01”，如果定义为“10-1”，禁用的将是 10 月以 1 开头的日子，即 10 号到 19 号。配置项 disabledDays 则用于禁用一周内的日期，示例中的 0 表示周日，6 表示周六，即不能选择周六、周日这些日子。配置项 minValue 和 maxValue 并不会减少日期选择器可显示的日期，只是会禁用不在日期范围内的值。配置项 submitFormat 可定义日期的提交格式，被选择的日期将会以该配置项定义的格式提交，在示例中，只会提交月份和日的数据。示例最后演示了如何使用 FieldContainer 定义放置两个 DateField 字段。

最后在表单面板内的底部工具栏定义一个保存按钮来测试提交值，代码如下：

```
{text:"保存",handler:function(){
   var f=this.up("form").getForm();
   if(f.isValid())
          f.submit();
}}
```

在浏览器中打开页面，并打开第二个字段，将看到如图 12-13 所示的效果。从图 12-13 中可以看到，2010 年的日期被禁用了，周六、日也被禁用了。修改一下字段 1 和字段 2 的值，然后单击“保存”，在控制台可看到 field2 的提交值为“01-14”，是按 submitFormat 定义的格式提交的值。

图 12-13　DateField 示例的页面效果

12.3.6　使用 TimeField 字段

TimeField 字段派生于 Picker 字段，它的选择器是 TimePicker 对象实例，因此其中的很多配置项是 TimePicker 对象的配置项。TimeField 字段只能选择以分钟为单位的时间，如果要以秒为单位，需要自己进行扩展。

TimeField 字段与 DateField 字段一样，需要严格依据配置项 format 定义的格式进行配置。其默认格式是“g:i A”，在使用时基本都要重新定义。下面通过一个示例来演示如何使用 TimeField 字段。

复制文件 12-10.html 并将复制后的文件名修改为 12-12.html，删除 items 中的代码，然后修改 defaultType 的值为 timefield，接着在 items 中加入以下代码：

```
{fieldLabel:"字段 1",name:"field1",
   format:"H:i",allowBlank:false
},
{fieldLabel:"字段 2",name:"field2",
   format:"H:i",allowBlank:false,
   increment:5,minValue:"08:00",
   maxValue:"18:00"
}
```

以上代码定义了两个 TimeField 字段：第一个只是简单地定义了时间格式；第二个则设置时间的增量（increment）为 5，这样时间就会以 5 分钟为增量增加，默认值是 15 分钟。以上代码还为时间列表定义了 minValue 和 maxValue 配置项，时间列表的时间将会是从 8 点到 18 点，而且会以 5 分钟间隔为一个选择项。注意，minValue 和 maxValue 的值必须按照时间格式定义。

在浏览器中打开页面，并打开字段 2 的列表，将看到如图 12-14 所示的效果。

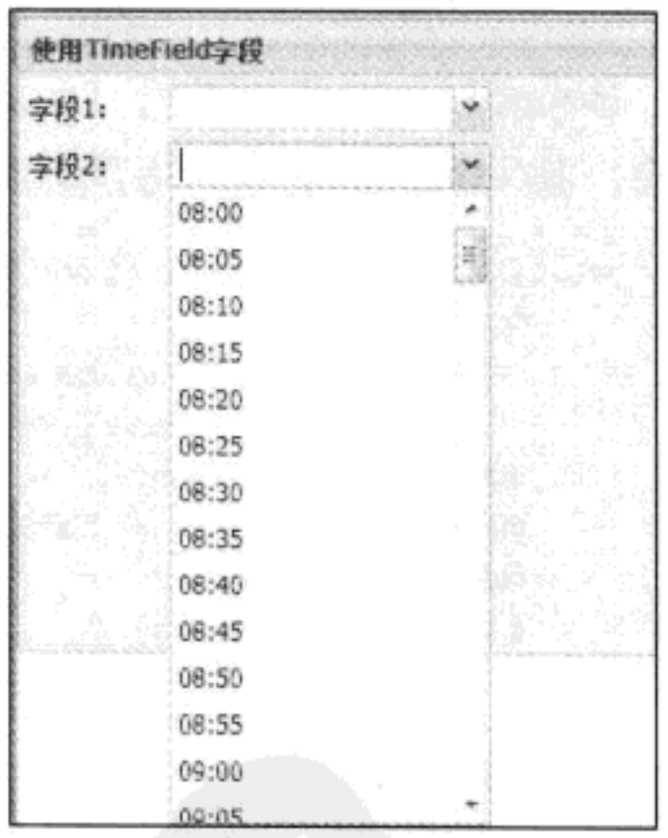


图 12-14　TimeField 示例的页面效果

12.4　使用 ComboBox 字段

12.4.1　概述

ComboBox 字段派生于 Picker 字段，它使用 BoundList 对象作为选择器，而 BoundList

对象派生于 DataView 对象，因此可以说 ComboBox 字段是 Text 字段与视图的结合体，这就决定了 ComboBox 字段必须有 Store，也会有选择模型。ComboBox 字段选择值的功能就是通过选择模型实现的，在 createPicker 方法中，会将选择模型的 SelectionChange 事件绑定到 onListSelectionChange 方法，其代码如下：

```
onListSelectionChange: function(list, selectedRecords) {
    var me = this,
        isMulti = me.multiSelect,
        hasRecords = selectedRecords.length > 0;
    if (!me.ignoreSelection && me.isExpanded) {
        if (!isMulti) {
            Ext.defer(me.collapse, 1, me);
        }
        if (isMulti || hasRecords){
            me.setValue(selectedRecords, false);
        }
        if (hasRecords) {
            me.fireEvent('select', me, selectedRecords);
        }
        me.inputEl.focus();
    }
},
```

从粗体代码可以看到，只要有选择记录，就会调用 setValue 方法从选择记录中取值。

12.4.2 BoundList 对象的运行流程

BoundList 对象派生于 DataView，这个要清楚，因为它的很大部分流程是依据 DataView 走的，其 initComponent 方法代码如下：

```
initComponent: function() {
    var me = this,
        baseCls = me.baseCls,
        itemCls = me.itemCls;
    me.selectedItemCls = baseCls + '-selected';
    me.overItemCls = baseCls + '-item-over';
    me.itemSelector = "." + itemCls;
    if (me.floating) {
        me.addCls(baseCls + '-floating');
    }
    if (!me.tpl){
        me.tpl = new Ext.XTemplate(
            '<ul><tpl for=".">',
                '<li role="option" class="' + itemCls + '">' + me.getInnerTpl(me.
                displayField)+ '</li>',
            '</tpl></ul>'
        );
    } else if (Ext.isString(me.tpl)){
        me.tpl = new Ext.XTemplate(me.tpl);
    }
    if (me.pageSize) {
        me.pagingToolbar = me.createPagingToolbar();
    }
```

```
        me.callParent();
    },
```

以上代码的重点在粗体代码的模板生成方式上。如果定义了配置项 tpl，那么使用 tpl 创建模板。如果不使用 tpl，就得重写 getInnerTpl 方法，在方法内返回模板。若不重写 getInnerTpl 方法，将返回由大括号括起来的 displayField 定义的显示字段作为模板，也就是简单的列表。

对于粗体代码下面这句，如果定义了配置项 pageSize，会创建分页工具条，并在 BoundList 对象的底部显示。

12.4.3　ComboBox 字段的配置项、属性、方法和事件

1. 配置项

- ❑ allQuery：发送到服务器的查询字符，默认值为空字符串。
- ❑ autoSelect ：布尔值，默认值为 true，当下拉列表显示时，会默认选择第一个值。如果设置为 false，则需要自己选择一个值。
- ❑ delimiter：多选时用来分隔选择值的符号，默认值为逗号（,）。
- ❑ displayField：用来显示字段名称，默认值为 text。
- ❑ forceSelection ：布尔值，设置为 true，必须从列表中选择一个值作为值。默认值为 false，表示选择的值不需要在列表中。
- ❑ listConfig ：BoundList 对象的配置对象，BoundList 只多定义了 pageSize 配置项，其余的都是从 DataView 继承过来的配置项。
- ❑ minChars ：当 typeAhead 配置项为 true 时，自动完成前用户需要输入的最小字符。在启用远程模式时默认值为 4 个字符，在启用本地模式时默认值为 0 个字符。该功能在 editable 为 false 时不起作用。
- ❑ multiSelect：布尔值，默认值为 false，只能选择一个值。如果设置为 true，则允许选择多个值。
- ❑ pageSize：每页列表的数据数量。只有在远程模式下才起作用。
- ❑ queryDelay：设置用户从开始输入到发送查询的时间间隔，单位是微秒。在启用远程模式时默认值为 500，在启用本地模式时默认值为 10。
- ❑ queryMode：值为 remote 或 local。默认值为 remote，表示将会从服务器加载列表数据。
- ❑ queryParam：定义提交到服务器的查询值的参数名称，默认值为 query。
- ❑ selectOnTab：布尔值，默认值为 true，表示可以使用 Tab 键选择当前高亮显示的值。
- ❑ store：定义 Store。
- ❑ transform：将 select 标记转换为 ComboBox。
- ❑ triggerAction ：值为 all 或 query。默认值为 all，当单击下拉按钮时，会将 allQuery 定义的值发送到服务器，否则使用真实值指向查询。
- ❑ typeAhead ：布尔值，如果设置为 true，当用户的输入停顿时间超过了 typeAheadDelay 定义的值时，将根据输入找到匹配的值并自动完成输入。

- ❑ typeAheadDelay：设置从用户输入停顿到自动完成输入值选择的时间间隔，单位为微秒。默认值为 250。
- ❑ valueField：设置实际值的字段名称。
- ❑ valueNotFoundText：当在 Store 中根据显示值找不到实际值的时候，会显示该配置项定义的文本。

2. 属性

lastQuery：最后的查询值，如果删除该值，会强迫进行新的查询。

3. 方法

- ❑ clearValue：清理 ComboBox 的值。
- ❑ doQuery：对下拉列表执行查询。
- ❑ select：选择指定的值。

4. 事件

beforeqery：在执行查询前会触发该事件。返回 false 表示可终止查询。

12.4.4 最简单的 ComboBox

对于最简单的 ComboBox，只需要配置好 Store 就行了，甚至不需要配置显示字段和实际值字段，不过，这只适用于显示值与实际值相同的情况。如果显示值和实际值不同，则需要定义 valueField 和 displayField 配置项。下面通过一个示例演示在以上两种情况下最简单的 ComboBox。

复制文件 12-11.html 并将复制后的文件名修改为 12-13.html，删除 items 中的代码，然后修改 defaultType 的值为 Combobox。

首先要做的是定义两个 Store，代码如下：

```
var store1=Ext.create("Ext.data.ArrayStore",{
    fields:["text"],
    data:[["小学"],["初中"],["高中"],["本科"],["研究生"],["博士"],["博士后"]]
});

var store2=Ext.create("Ext.data.ArrayStore",{
    fields:["id","text"],
    data:[[1,"小学"],[2,"初中"],[3,"高中"],[4,"本科"],[5,"研究生"],[6,"博士"],[7,"
        博士后"]]
});
```

最简单的 Store 用于选择学历。第一个 Store 用来演示显示值与实际值相同的情况，由于 displayField 字段默认使用 text 作为显示字段，因此将 Store 的字段名称定义为 text，就不需要做额外的定义了。第二个 Store 则用来演示显示值与实际值不同的情况。

下面在表单面板的 items 中定义两个 ComboBox，分别使用 store1、store 作为它们的数据源，代码如下：

```
{fieldLabel:"学历",name:"field1",
```

```
    store:store1,anchor:"90%",allowBlank:false
},
{fieldLabel:" 学历 ",name:"field2",valueField:"id",
    store:store2,anchor:"90%",allowBlank:false
}
```

对于第一个 ComboBox，除了 Store 是 ComboBox 增加的配置项，其余都是 BaseField 的配置项，非常简单。第二个 ComboBox 则复杂点，除了要定义 Store 外，还要指定实际值的字段（valueField）为 id。

使用 ComboBox 就是这么简单，关键在 Store。

在浏览器中打开页面，在第一个 ComboBox 中选择高中，在第二个 ComboBox 中选择研究生，将看到如图 12-15 所示的效果。单击“保存”按钮，将看到 field1 的提交值为“高中”，而 field2 的提交值为 5，正是研究生的 id 值。

图 12-15　最简单的 ComboBox 的页面效果

12.4.5　自定义列表显示格式的 ComboBox

要自定义列表的显示格式，有两种方法：一种是重写 getInnerTpl 方法；一种是使用视图的定义方式定义 tpl 配置项，不过，BoundList 在 initComponent 方法中固定了条目的选择样式，而且不能通过配置项对其进行修改，因而必须在渲染前（执行 beforerender 事件前）重新定义样式。下面通过一个示例演示如何使用这两种方式自定义列表的显示格式。

复制文件 12-11.html 并将复制后的文件名修改为 12-14.html，删除 items 中的代码，然后修改 defaultType 的值为 combobox。

首先定义一个 ComboBox，通过该 ComboBox 可以选择国家，其字段包括国家名称字母简写和中文名称。在列表中，通过简写显示国家国旗和中文名称。先定义一个带两个字段的 store1，代码如下：

```
var store1=Ext.create("Ext.data.ArrayStore",{
    fields:["id","text"],
    data:[["au"," 澳大利亚 "],["br"," 巴西 "],["ca"," 加拿大 "],
          ["cn"," 中国 "],["es"," 西班牙 "],["fr"," 法国 "],
          ["ru"," 俄罗斯 "],["us"," 美国 "]]
});
```

接着定义 ComboBox，代码如下：

```
{fieldLabel:" 国家 ",name:"field1",valueField:"id",
    store:store1,anchor:"90%",allowBlank:false,
    listConfig:{
        getInnerTpl:function(){
            return "<img src='../images/flag/{id}.gif'/>{text}";
        }
    }
},
```

这里重定义了 getInnerTpl 方法，它将返回带 img 标记的模板代码，它必须定义在

listConfig 中，这样才能将配置项应用于 BoundList。

第二个 ComboBox 将使用 9.6.6 节示例的数据和模板，通过 ComboBox 选择手机。首先要做的是复制 phoneList、selected 和 overitem 这 3 个样式，接着复制模型和 Store 定义，最后定义 ComboBox，代码如下：

```
{fieldLabel:"手机",name:"field2",valueField:"id",
   displayField:"name",store:"phoneStore",anchor:"90%",allowBlank:false,
    multiSelect:true,
    matchFieldWidth:false,
   listConfig:{
       trackOver:true,
       width:560,
       height:400,
           // 省略模板代码定义
           listeners:{
                  beforerender:function(){
                      this.selectedItemCls="selected"
                      this.itemSelector='div.phoneList'
                      this.overItemCls="overitem"
                  }
           }
   }
}
```

代码中的模板代码可以直接从 9.6.6 节的示例中复制过来。显示字段要显示的是选择手机名称，所以要定义其为 name 字段。定义了 multiSelect 为 true，说明可进行多选。必须将配置项 matchFieldWidth 设置为 false，不然列表的显示将会与 ComboBox 等宽，不便于显示和选择。在 BoundList 的配置对象中，定义了列表的宽度和高度，还有模板。在这里要监听 beforerender 事件，将条目选择的样式、选择器和鼠标划过条目的样式进行重定义。

至此，示例就完成了，在浏览器中打开页面，逐个打开两个 ComboBox 列表，将看到如图 12-16 所示的效果。

12.4.6 动态调整 ComboBox 的列表数据

经过前面的学习，如果对 Ext JS 的三层架构框架已熟悉，要实现动态调整 ComboBox 的列表数据应该不难。列表数据都在 Store 中，只要在 Store 中添加或删除数据，就会自动刷新列表的显示，也就实现了对 ComboBox 的列表数据的动态调整。下面通过一个示例来演示如何实现动态调整 ComboBox 的列表数据。

复制文件 12-13.html 并将复制后的文件名修改为 12-15.html，删除 items 中的代码，然后在 fieldDefaults 配置项中添加配置项 queryMode，值为 local，如果这样添加，默认进行远程查询，会在渲染列表的时候使用 Reader 从原有的数据中读取一次数据，这样，在渲染前就已经加入的数据就会消失了。如果是本地，则不会有这个过程。

最后在底部工具栏中添加以下 3 个按钮：

图 12-16　自定义列表显示格式的 ComboBox 的页面效果

```
{text:"+1",handler:function(){
    var id=parseInt(Ext.id(null,"0"));
    store1.add({text:"列表"+id});
    store2.add({id:id,text:"列表"+id});
}},
{text:"+5",handler:function(){
    var datas1=[],datas2=[];
    for(var i=0;i<5;i++){
            var id=parseInt(Ext.id(null,"0"));
            datas1.push({text:"列表"+id});
            datas2.push({id:id,text:"列表"+id});
    }
    store1.loadData(datas1,true);
    store2.loadData(datas2,true);
}},
{text:"-1",handler:function(){
    if(store1.count()>1){
            store1.removeAt(0);
            store2.removeAt(0);
    }
}}
```

第一个按钮使用 Store 的 add 方法为 Store 添加一个记录；第二个按钮使用 loadData 方法一次追加 5 个记录，注意第二参数必须为 true，不然 Store 会先删除原来的数据再添加新的；第三个按钮则使用 removeAt 方法删除 Store 中的第一个数据。

除了示例中的三种方法，还可以使用 insert、loadRecords、remove 或 removAll 等方法来添加或删除数据，这里就不一一举例了，有兴趣可以自己研究一下。

在浏览器中打开页面，并一次单击“+1”、“+5”和“–1”按钮，最后将看到如图 12-17 所示的效果。

图 12-17 动态调整 ComboBox 的列表数据的页面效果

12.4.7 实现 ComboBox 的联动

实现 ComboBox 的联动的原理就是在用户为第一个 ComboBox 选择值的时候，通过 change 事件监测到其值发生改变，然后使用上一节的方法为第二个 ComboBox 动态加载数据，如果还有第三个 ComboBox，则使用相同手法监测第二个 ComboBox 的值并加载第三个 ComboBox 的数据，依此类推，就可实现许多 ComboBox 的联动。

实现 ComboBox 的联动有本地模式和远程模式两种，下面通过一个示例演示如何实现省、市、县的 ComboBox 联动。在实现之前，先使用本书资源包中资源目录下的 AllCityData.sql 在 Northwind 数据库中创建 T_Province、T_City 和 T_District 这 3 张表，并插入省、市、县的数据。

复制文件 12-15.html 并将复制后的文件名修改为 12-16.html，删除 store1、store2 的定义，删除 items 中的代码，删除 fieldDefaults 配置项，删除底部工具栏中除保存外其他按钮的定义。

首先实现本地联动的 ComboBox 加载数据的方式也有两种：一种是将后台数据生成一个脚本文本放到一个全局变量中，通过 script 标记加载；另一种是通过第一个 ComboBox 的 Store 自动加载全部数据。两种方式的数据结构可以一样，但返回方式可以不同，以下是数据结构：

```
[
   {id:1,text:"广东",citys:[
          {id:1,text:"广州",districts:[
                 {id:1,text:"天河区"},
                 ...
          ]},
          ...
   ]}
   ...
]
```

从最外层数组开始，省份数组内的每个对象就是每一个省的数据，包括 id、名称（text）和由该省的城市组成的数组（citys）；在城市数组内，每个对象就是一个城市的数据，包括 id、名称（text）和由该城市的县区组成的（districts）；在县区数组内，每个对象就是一个县区的数据，因为县区是最后一层，所有只包括 id 和名称（text）两个字段。

这种结构的好处是，在 change 事件中，可根据参数返回的新 id 值调用 Store 的 getById 方法找到该条记录，然后直接使用 loadData 方法将记录中的 citys 或 districts 字段的值加载到

下一个 ComboBox 的 Store 中，操作起来简单方便，不需要二次查找。

结构定好后，可根据结构定义 3 个 Store，首先定义省份使用的 Store，省份有 id、text 和 citys 三个字段，数据是从脚本中的全局变量 data 获取的，因此可定义如下：

```
Ext.create("Ext.data.Store",{
    id:"pStore1",
    fields:[
            {name:"id",type:"int"},
            "text","citys"
    ],
    data:data
});
```

城市的数据结构包括 id、text 和 districts 三个字段，数据是动态添加的，因此可定义如下：

```
Ext.create("Ext.data.Store",{
    id:"cStore1",
    fields:[
            {name:"id",type:"int"},
            "text","districts"
    ]
});
```

县区的数据结构只有 id 和 text 两个字段，数据也是动态的，因此可定义如下：

```
Ext.create("Ext.data.Store",{
    id:"dStore1",
    fields:[
            {name:"id",type:"int"},
            "text"
    ]
});
```

注意这 3 个 Store 都定义了 id，目的是方便操作，本示例会有 9 个 Store（9 个 ComboBox），如果定义 9 个局部变量，会比较混乱，使用 id 会简单很多。

数据 data 是使用 script 标记方式加载的，因此需要在页面的 head 标记内添加一个 script 标记，代码如下：

```
<script type="text/javascript" src="province.ashx?act=js"></script>
```

下面开始定义 ComboBox。因为要使用 FieldContainer，且要定义三个，所以定义默认配置（defaults）是一个好的方法。这 3 个 FieldContainer 的共同配置是使用水平布局，标签的宽度固定为 80，字段要填满表单余下的宽度。容器内的字段都是 ComboBox，因此可定义默认的字段类型（defaultType）为 combobox，还可加上字段的默认配置（fieldDefaults）。字段的默认配置包括不允许为空（allowBlank 为 false）；平均分配宽度（flex 为 1）；强迫输入值必须在列表中（forceSelection 为 true）；打开列表时，字段选择第一个值（autoSelect 为 true）；选择查询模式为本地模式（queryMode 为 local），虽然第三个 FieldContainer 使用的是远程动态加载数据，但是和列表的数据差不多，不需要再查询，可设置查询模式为本地模式。因此，在表单面板中加入以下定义：

```
defaultType:"fieldcontainer",
defaults:{
    labelWidth:80,layout:"hbox",anchor:"0",
    defaultType:"combobox",
    fieldDefaults:{
            forceSelection:true,allowBlank:false,flex:1,
            queryMode:"local",autoSelect:true
    }
},
```

现在要在 items 中定义 FieldContainer 和第一个 ComboBox。定义 FieldContainer 非常简单，只需要定义标签就可以了，在其 items 中定义第一个 ComboBox，代码如下：

```
{fieldLabel:"地区",items:[
    {name:"p1",valueField:"id",displayField:"text",store:"pStore1",

    listeners:{
            change:function(f,n,o){
            var p=this.store,
                    c=Ext.StoreManager.lookup("cStore1"),
            cb=this.up("form").getForm().findField("c1"),
            cb1=this.up("form").getForm().findField("d1");
            if(n && n!=o){
                    var rec=p.getById(n);
                    cb.setValue();
                    cb1.setValue();
                    if(rec){
                    c.loadData(rec.data.citys);
                    }
            }
            }
    }
    },
]},
```

在 change 事件的函数中，如果新值（n）不是空值，并且和旧值（o）不同，才会调用 getById 方法返回修改后的记录。接着清理后面两个 ComboBox 的值，以避免旧的值影响最后的值。最后判断 getById 方法是否有记录返回，有返回记录才会有 citys 字段，才能调用 loadData 方法将 Citys 字段内的值加载到第二个 ComboBox 中。

接着定义第二个 ComboBox，与第一个的主要区别是 name、store 的定义不同，在 chang 事件中，只需要清除第三个 ComboBox 的值，并为第三个 ComboBox 加载数据，代码如下：

```
{name:"c1",valueField:"id",displayField:"text",store:"cStore1",
    listeners:{
change:function(f,n,o){
        var p=this.store,
        c=Ext.StoreManager.lookup("dStore1"),
        cb=this.up("form").getForm().findField("d1");
        if(n && n!=o){
        var rec=p.getById(n);
                cb.setValue();
```

```
                if(rec){
                        c.loadData(rec.data.districts);
                }
                }
                }
    }
},
```

最后一个 ComboBox 比较简单，没有 change 事件，只需要改变一下 name 和 store 的定义就可以，代码如下：

```
{name:"d1",valueField:"id",displayField:"text",store:"dStore1"}
```

至此，第一组 ComboBox 就完成了。现在完成第二组 ComboBox。第二组与第一组的区别只是省份 Store 加载数据方式不同，其余都是一样的，因而可以将第一组的 Store 定义，FieldContainer 及其内部的 ComboBox 定义复制一份，然后将 Store 中的 id 配置项、ComboBox 的 name 配置项中值的序数 1 修改为 2，例如将 pStore1 修改为 pStore2，将 p1 修改为 p2，还要相应修改 change 事件中 lookup 和 findField 方法中参数。完成修改后，将 pStore2 中的配置项 data 删除，添加 autoLoad 配置项，值为 true，然后添加 proxy 定义，代码如下：

```
proxy:{
    type: 'ajax',
    url:"province.ashx?act=json",
    reader:{
            type:"json",
            root:"data"
    }
}
```

如果使用的是 XML 格式数据，参考之前章节中的示例进行修改就可以了，不会太难。

至此，第二组 ComboBox 也完成了。

最后一组 ComboBox 将全部使用远程加载数据的方式，因此需要为 3 个 Store 都加上 Proxy 定义，定义的代码与 pStore2 中除了 url 配置项中的提交参数 act 的值不同外，其余都是一样的，因而可直接复制过来，修改 id、name 等配置项的值中的序数，并加入 proxy 定义就行了。

最后一组 ComboBox 的定义与前面两组也基本一样，先修改 id 和 name 中的序数，然后修改一下 change 事件，主要修改是：不需要使用 getById 方法获取记录了，只需要在 Proxy 对象的 extraParams 配置项下加一个属性，其值为新值（n），然后调用 load 方法就行了。

对字段 p3 的 change 事件的修改如下：

```
c.getProxy().extraParams.proid=n;
c.load();
```

这里将会提交参数 proid，其值为当前选择省份的 id。

字段 c3 的 change 事件代码修改如下：

```
c.getProxy().extraParams.cityid=n;
c.load();
```

与字段 p3 的主要区别是使用了 cityid 作为提交参数。

至此，页面就已经完成了，现在要完成服务器端代码。第一组与第二组 ComboBox 使用的基础数据是一样的，只是返回时格式有点不同，因此可使用同一个 GetAll 方法返回一个 JSON 数组，然后再根据 act 的值组合结果。如果参数值是 js，则将字符串“var data=”加上 JSON 数组生成的字符串返回；如果参数是 json，则创建一个 JSON 对象，添加 success 属性，只为 true，添加 data 属性，只为 GetAll 方法返回 JSON 数组就可以了。

而第三组 ComboBox 的数据加载是分开的，要根据其 act 的值来操作，如果是 pro，则调用 Provinces 将全部省份数据组织成标准的 Store 返回格式返回即可；如果是 city，则调用 Citys 方法提取 proid 参数的值，如果提取的值为大于 0 的整数，则从 T_City 表中查询 ProID 与其相等的数据，并组织成标准格式返回即可；如果是 dis，则调用 Districts 方法提取 cityid 参数的值，如果提取的值为大于 0 的整数，则从 T_District 表中查询 CityID 与其相等的数据，并组织成标准格式返回即可。

我们先看看 GetAll 方法，代码如下：

C#

```
private JArray GetAll()
{
    JArray ja = new JArray();
    using (NorthwindEntities ne = new NorthwindEntities())
    {
var q = ne.T_Province.OrderBy(m => m.ProName);
foreach (var c in q)
{
    JObject jo = new JObject(
        new JProperty("id",c.ProID),
        new JProperty("text",c.ProName),
        new JProperty("citys",new JArray(
            from d in c.T_City
            select new JObject(
                new JProperty("id",d.CityID),
                new JProperty("text",d.CityName),
                new JProperty("districts",new JArray(
                    from e in d.T_District
                    select new JObject(
                        new JProperty("id",e.Id),
                        new JProperty("text",e.DisName)
                    )
                ))
            )
        ))
    );
    ja.Add(jo);
}
    }
    return ja;
}
```

Java

```
private JsonArray GetAll(){
    String connectionUrl = "jdbc:sqlserver://192.168.0.254:1433;" +
    "databaseName=Northwind;;user=sa;password=abcd-1234";
    Connection con = null;
    Statement stmt = null;
    ResultSet pro = null;
    ResultSet city = null;
    ResultSet dis = null;
    JsonArray resultArray=new JsonArray();
    try {

        Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
        con = DriverManager.getConnection(connectionUrl);
        stmt = con.createStatement();
        // 处理县数据
        dis = stmt.executeQuery("select ID,DisName,CityID from T_District order by
            CityID,DisName");
        JsonObject disJsonObject=new JsonObject();
        String id = "";
        while (dis.next()) {
              id=dis.getString("CityID");
              if(!disJsonObject.has(id)){
                    disJsonObject.add(id, new JsonArray());
              }
              JsonObject jd=new JsonObject();
              jd.addProperty("id", dis.getInt("ID"));
              jd.addProperty("text", dis.getString("DisName"));
              disJsonObject.getAsJsonArray(id).add(jd);
        }
        dis.close();

        // 处理市数据
        city = stmt.executeQuery("select CityID,CityName,ProID from T_City order
            by ProID,CityName");
        JsonObject cityJsonObject=new JsonObject();
        while (city.next()) {
              id=city.getString("ProID");
              if(!cityJsonObject.has(id)){
                    cityJsonObject.add(id, new JsonArray());
              }
              JsonObject jd=new JsonObject();
              jd.addProperty("id", city.getInt("CityID"));
              jd.addProperty("text", city.getString("CityName"));
              jd.add("districts", disJsonObject.getAsJsonArray(city.
                    getString("CityID")));
              cityJsonObject.getAsJsonArray(id).add(jd);
        }
        city.close();

        // 处理省数据
        pro = stmt.executeQuery("select ProID,ProName from T_Province order by
            ProName");
        while (pro.next()) {
```

```
                JsonObject jd=new JsonObject();
                jd.addProperty("id", pro.getInt("ProID"));
                jd.addProperty("text", pro.getString("ProName"));
                jd.add("citys", cityJsonObject.getAsJsonArray(pro.
                    getString("ProID")));
                resultArray.add(jd);
            }
            pro.close();

        }
        catch (Exception e) {
            JsonObject jd=new JsonObject();
            jd.addProperty("msg", e.getMessage());
            resultArray.add(jd);
        }
        finally {
            if (dis != null) try { dis.close(); } catch(Exception e) {}
            if (city != null) try { city.close(); } catch(Exception e) {}
            if (pro != null) try { pro.close(); } catch(Exception e) {}
            if (stmt != null) try { stmt.close(); } catch(Exception e) {}
            if (con != null) try { con.close(); } catch(Exception e) {}
        }
        return resultArray;

    }
```

C# 使用实体模型，可以嵌套查询子表，因此使用 LINQ 就可一次性生成 JSON 对象了，将该对象加入 JSON 数组返回就可以了。

Java 要做 3 次查询，所以需要使用 JSON 对象过渡，先将同一城市下的县区归纳到以城市 CityID 为属性名称的 JSON 数组中；接着查询出城市数据，再从之前的县区 JSON 对象中找出对应的县区数组加入到 districts 属性中，而城市也会根据 ProID 加入到对应的数值中；最后，在查询出省份数据后，将 ProID 对应的城市数组加入到 citys 属性就完成了整个数据结构。

如果 C# 也是使用查询方法获取数据，就可使用 Java 代码中的方法组合数据。

最后看看 Citys 方法，代码如下：

C#

```
private string Citys(HttpRequest request)
{
    JArray ja = new JArray();
    int id = 0;
    int.TryParse(request.Params["proid"], out id);
    if (id > 0)
    {
        using (NorthwindEntities ne = new NorthwindEntities())
        {
            var q = ne.T_City.Where(m=>m.ProID==id);
            foreach (var c in q)
            {
                ja.Add(new JObject(
                    new JProperty("id", c.CityID),
                    new JProperty("text", c.CityName)
```

```
                ));
            }
        }
    }
    return new JObject(
        new JProperty("success", true),
        new JProperty("data", ja)
    ).ToString();
}
```

Java

```
private String Citys(HttpServletRequest request) {
    String connectionUrl = "jdbc:sqlserver://192.168.0.254:1433;" +
    "databaseName=Northwind;;user=sa;password=abcd-1234";
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;
    JsonObject jo=new JsonObject();
    JsonArray ja=new JsonArray();
    int id=0;
    if(request.getParameter("proid")!=null){
          id=Integer.parseInt(request.getParameter("proid"));
    }
    try {
          if(id>0){
         Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
         con = DriverManager.getConnection(connectionUrl);
         stmt = con.createStatement();

         rs = stmt.executeQuery("select CityID,CityName from T_City where ProID="
                   + id  +" order by CityName");

         while (rs.next()) {
             JsonObject jd=new JsonObject();
             jd.addProperty("id", rs.getInt("CityID"));
             jd.addProperty("text", rs.getString("CityName"));
             ja.add(jd);
         }
         rs.close();
          }
          jo.add("success", new JsonPrimitive(true));
          jo.add("data", ja);
}
catch (Exception e) {
          jo.add("success", new JsonPrimitive(false));
          jo.addProperty("msg", e.getMessage());
    }
    finally {
        if (rs != null) try { rs.close(); } catch(Exception e) {}
        if (stmt != null) try { stmt.close(); } catch(Exception e) {}
        if (con != null) try { con.close(); } catch(Exception e) {}
    }
    return jo.toString();

}
```

C# 和 Java 的代码差不多，都是先获取提交的 proid 值，如果此值大于 0，则执行查询，并组合 JSON 对象；如果 proid 值不大于 0，则 data 属性返回一个空数组，表示没有找到数据。

余下的代码和 Citys 方法差不多，读者可以在资源包中找到对应文件进行查看和测试。

在浏览器中打开页面，可以看到，初始时城市和县区打开的列表都是空的，是没有数据的，依次选择 3 组 ComboBox 后，会看到如图 12-18 所示的效果。

注意 因为 Northwind 数据库默认的排序规则是 SQL_Latin1_General_CP1_CI_AS，因此示例中的中文排序存在问题。

图 12-18 联动的 ComboBox 的效果

12.4.8 使用 ComboBox 的查询功能

在 ComboBox 中，配置项 queryMode 的默认值是 remote，也就是用户在输入由 minChars 配置项定义的字符串数后，在停顿超过 queryDeiay 配置项定义的时间后，就会发送一个请求到服务器端，这时服务器可根据提交的值查询出数据并返回，这样可减少选择项，功能类似搜索引擎的搜索框，当用户输入某些字符后，字段可列出类似的查询关键字供用户选择。

在应用中，常用的地方就是根据关键字搜索产品，选择产品后，将产品的单价取出，然后通过数量计算出合计金额等。下面使用 Northwind 的 Products 演示如何实现该功能。

复制文件 12-1.html 并将复制后的文件名修改为 12-17.html，删除 items 中的代码，先定义模型，在示例只需要 ProductID、ProductName 和 UnitPrice 这 3 个字段，定义如下：

```
Ext.define('Product', {
    extend: 'Ext.data.Model',
    fields: [
           {name:'ProductID',type:"int"},
           'ProductName',
           {name:"UnitPrice",type:"float"}
    ],
```

```
    idProperty:"ProductID"
});
```

这里**一定要正确定义 idProperty 配置项，不然使用 getById 方法将找不到记录。**

接着定义 Store，代码如下：

```
Ext.create("Ext.data.Store",{
    id:"productStore",
    model:Product,
    proxy:{
            type:"ajax",
            url:"Product.ashx",
            reader:{
                    type:"json",
                    root:"data"
            }
    }
});
```

现在开始定义字段，先定义 ComboBox，代码如下：

```
{xtype:"combobox",id:"pid",name:"pid",fieldLabel:"选择产品",
    valueField:"ProductID",displayField:"ProductName",
    store:"productStore",minChars:"2",forceSelection:true,
    allowBlank:false,listConfig:{
            loadingText: '搜索中...',
            emptyText: '没有匹配的产品',
        tpl:Ext.create('Ext.XTemplate','<ul><tpl for=".">',
            '<li role="option" class="x-boundlist-item">{ProductName:this.
                highlight()}</li>',
        '</tpl></ul>',
        {
            highlight:function(v){
                var cmp=Ext.getCmp("pid"),
                        query=cmp.lastQuery;
                if(Ext.isEmpty(query)){
                        return v;
                }else{
                return v.replace(new RegExp(query,'gi'),function(m){
                        return "<font color='red'>"+m+"</font>";
                });
        }
        }
    })
},
 listeners:{
        change:function(f,n,o){
        var     me=this,
                form=me.up("form").getForm(),
                rec=me.store.getById(n);
        if(rec){
                form.findField("price").setValue(Ext.Number.toFixed(
                rec.data.UnitPrice,2));
                form.findField("total").setValue(
                        Ext.Number.toFixed(form.findField("quantity").
                            getValue()
```

```
                                        *rec.data.UnitPrice,2));
                        }
                    }
                }
            },
```

为了便于搜索产品，设置 minChars 为 2，输入两个字符后就可执行查询了。因为产品是必选的项，且必须在列表中，所以设置 forceSelection 为 true。配置项 loadingText 的文本会在执行搜索时作为提示信息显示。当没有找到匹配记录时，会显示 emptyText 配置项的信息。为了实现在搜索时高亮显示搜索的文本，在示例中重定义了 tpl 配置项，而且直接创建 XTemplate 的实例，如果不这样，很难为模板定义自定义方法。要使用自定义的模板实例，一定要根据 BoundList 对象固定的条目样式定义模板代码，除非如 12.4.5 节那样，连样式都使用自定义的样式。本示例不需要那么复杂，直接使用 x-boundlist-item 作为样式就可以了。通过 ComboBox 的 lastQuery 属性，可以知道其最后一次搜索的字符串，如果不是空字符串，则高亮显示在显示值中搜索的字符串就可以了。

在 change 事件中，当值发生改变的时候，根据新的 ProductID 值找出记录，然后使用 setValue 方法设置单价和合计的值。为了固定显示格式，使用了 Ext.Number 对象的 toFixed 方法，以保证显示的是两位小数。

最后定义余下的 3 个字段，代码如下：

```
{xtype:"displayfield",name:"price",fieldLabel:"单价",value:0},
{xtype:"numberfield",fieldLabel:"数量",name:"quantity",value:0,minValue:0,
   step:10,allowDecimals:false,autoStripChars:true,listeners:{
              change:function(f,n,o){
                    var form=this.up("form").getForm()
                    form.findField("total").setValue(
              Ext.Number.toFixed(n*form.findField("price").getValue(),2));
              }
   }
},
{xtype:"displayfield",name:"total",fieldLabel:"合计",value:0}
```

这里使用了 Display 字段作为单击和合计的显示字段，如果希望界面更整齐，一般情况是使用只读或已禁用的 Text 字段作为显示。当“数量”字段发生改变的时候，需要调整合计的值。

至此客户端代码已经完成了，现在来编写服务器端代码，主要考虑提取 query 参数的值，然后将其加入查询语句中，这个不太难，代码如下：

C#

```
public void ProcessRequest (HttpContext context) {
    JArray ja = new JArray();
    string query = context.Request.Params["query"] ?? "";
    if (query.Length > 0)
    {
        query = string.Format("it.ProductName like '%{0}%'", query);
    }
    else
```

```
{
    query = "true";
}
using (NorthwindEntities ne = new NorthwindEntities())
{
    var q = ne.Products.Where(query).OrderBy(m => m.ProductName);
    foreach (var c in q)
    {
        ja.Add(new JObject(
            new JProperty("ProductID",c.ProductID),
            new JProperty("ProductName", c.ProductName),
            new JProperty("UnitPrice", c.UnitPrice)
        ));
    }
}
context.Response.ContentType = "text/javascript";
context.Response.Write(new JObject(
    new JProperty ("success",true),
    new JProperty("data",ja)
).ToString());
}
```

Java

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    String connectionUrl = "jdbc:sqlserver://192.168.0.254:1433;" +
    "databaseName=Northwind;;user=sa;password=abcd-1234";
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;
    JsonObject jo=new JsonObject();
    JsonArray ja=new JsonArray();
    String query="";
    if(request.getParameter("query")!=null){
          query=request.getParameter("query");
    }
    try {
       Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
       con = DriverManager.getConnection(connectionUrl);
       stmt = con.createStatement();

       rs = stmt.executeQuery("select  ProductID,ProductName,UnitPrice  from
            Products" +
           (query.length()>0 ? " where ProductName like '%" + query +"%'" : "")
                +" order by ProductName");
    while (rs.next()) {
          JsonObject jd=new JsonObject();
          jd.addProperty("ProductID", rs.getInt("ProductID"));
          jd.addProperty("ProductName", rs.getString("ProductName"));
          jd.addProperty("UnitPrice", rs.getFloat("UnitPrice"));
          ja.add(jd);
    }
    rs.close();
        jo.add("success", new JsonPrimitive(true));
```

```
            jo.add("data", ja);
        }
        catch (Exception e) {
            jo.add("success", new JsonPrimitive(false));
            jo.addProperty("msg", e.getMessage());
        }
        finally {
            if (rs != null) try { rs.close(); } catch(Exception e) {}
            if (stmt != null) try { stmt.close(); } catch(Exception e) {}
            if (con != null) try { con.close(); } catch(Exception e) {}
        }
        response.setContentType("text/javascript; charset=utf-8");
        response.getWriter().write(jo.toString());
    }
```

以上代码先获取 query 参数提交的查询值，如果值为空，则从数据库搜索所有数据，如果不为空，则加入查询条件。余下的工作就是根据返回的数据构建 JSON 的数据返回格式了。

在浏览器中打开页面，在 ComboBox 中输入 ch，并停顿一会儿，然后选择一个数据，并在数量中输入 100，效果如图 12-19 所示。

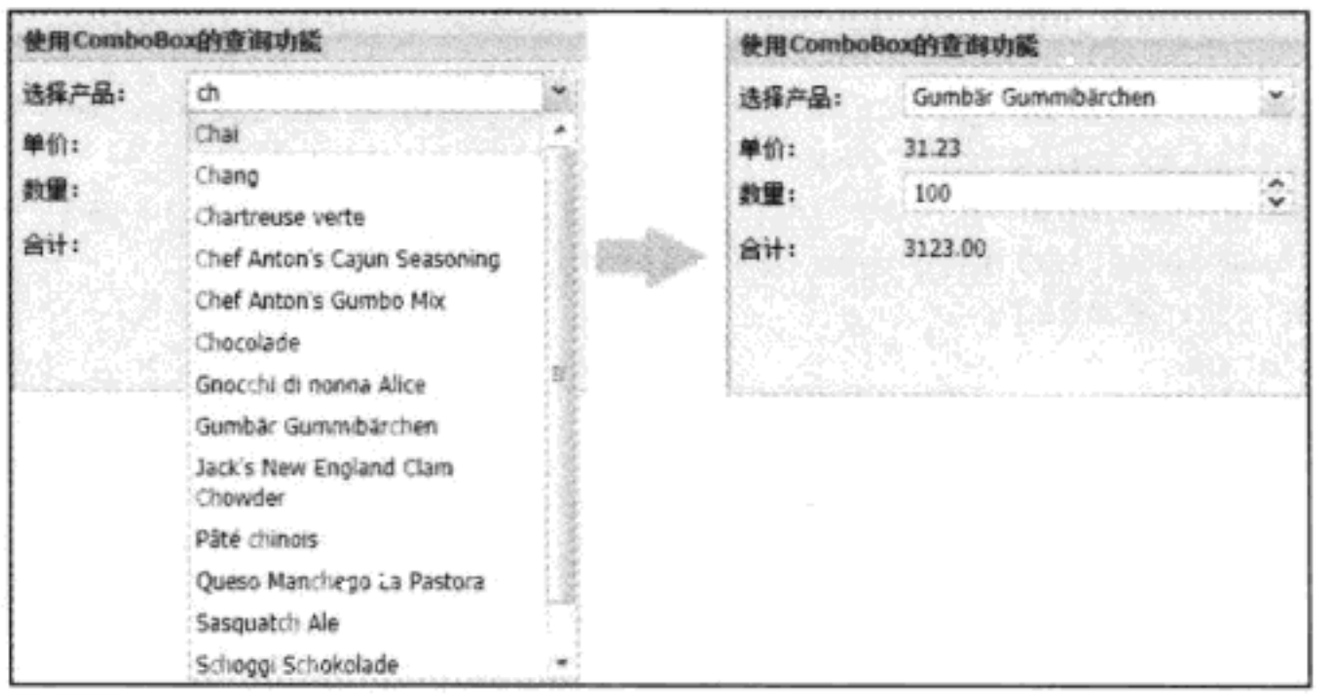


图 12-19　使用 ComboBox 查询功能的效果

12.4.9　设置 ComboBox 的默认值

设置 ComboBox 的默认值，无论采用何种模式，都可通过设置 value 配置项或者调用 setValues 方法设置默认值。如果采用单选模式，无论显示值与实际值是否相同，都要将默认值设置为实际值。如果采用多选模式，默认值为由实际值组合成的数组。如何设置分析一下 ComboBox 的 setValue 方法就很清楚了，其代码如下：

```
setValue: function(value, doSelect) {
    var me = this,
        valueNotFoundText = me.valueNotFoundText,
        inputEl = me.inputEl,
        i, len, record,
        models = [],
        displayTplData = [],
```

```
        processedValue = [];

    if (me.store.loading) {
        // Called while the Store is loading. Ensure it is processed by the onLoad method.
        me.value = value;
        me.setHiddenValue(me.value);
        return me;
    }

    value = Ext.Array.from(value);

    for (i = 0, len = value.length; i < len; i++) {
        record = value[i];
        if (!record || !record.isModel) {
            record = me.findRecordByValue(record);
        }
        if (record) {
            models.push(record);
            displayTplData.push(record.data);
            processedValue.push(record.get(me.valueField));
        }
        else {
            if (!me.forceSelection) {
                displayTplData.push(value[i]);
                processedValue.push(value[i]);
            }
            else if (Ext.isDefined(valueNotFoundText)) {
                displayTplData.push(valueNotFoundText);
            }
        }
    }

    me.setHiddenValue(processedValue);
    me.value = me.multiSelect ? processedValue : processedValue[0];
    if (!Ext.isDefined(me.value)) {
        me.value = null;
    }
    me.displayTplData = displayTplData; //store for getDisplayValue method
    me.lastSelection = me.valueModels = models;

    if (inputEl && me.emptyText && !Ext.isEmpty(value)) {
        inputEl.removeCls(me.emptyCls);
    }

    me.setRawValue(me.getDisplayValue());
    me.checkChange();

        if (doSelect !== false) {
            me.syncSelection();
        }
        me.applyEmptyText();

        return me;
    },
```

注意代码中粗体代码，无论 value 是任何类型，都将其转换为数组，然后在循环中调用 findRecordByValue 方法把值对应的记录找出来。如果记录存在，则将记录中的数据保存到 displayTplData 数组，实际值保存到 processedValue 数组；如果记录不存在，并且配置项 forceSelection 为 false，则把值直接保存到这两个数组中，否则，只保存一个在 valueNotFoundText 中定义的文本。

注意，调用 setHiddenValue 方法，会根据是单选模式还是多选模式将不同的值赋值给 value 属性，而使用的数组是 processedValue，也就是说，如果把显示值作为值调用 setValue 方法设置值，那么 value 属性指向的是显示值，而不是实际值了，可是在字段中，value 属性是用来记录实际值的，这样就会产生错误。

以上代码中的 setRawValue 方法会将显示值渲染到元素上。

12.5 表单的验证和加载数据

12.5.1 表单的验证及错误显示方式

表单的验证分客户端验证和服务器端验证两种，客户端验证在 12.2 节已经讲述过，本节就不再赘述了。本节重点是服务器端验证，因为笔者坚信“永远不要相信客户端验证后的数据必然是正确的”这个观点，所以无论是否有客户端验证，服务器端验证都是必须的。本节的重点是如何做服务器端验证并将验证结果返回客户端。

Ext JS 在显示服务器端验证错误方面提供了一个很好的机制，就是当返回的 JSON 对象的 success 属性为 false 时，会根据属性对应的字段将 errors 对象中的错误信息应用到字段，而字段会根据定义的错误信息显示方式将错误显示出来。字段的错误信息是根据配置项 msgTarget 的定义来显示的。

服务端验证主要存在的问题是如何做错误信息的收集。笔者习惯的方式是根据 Ext JS 的特点，先创建一个 JSON 对象，然后每验证一个字段，发现错误就在对象中加入以字段名称为属性，以错误信息为值的成员。

如果使用的是 XML 格式，则必须遵循以下格式：

```
<message>
    <success>true/false</success>
    <record>
            <id>fieldname</id><msg>message</msg>
    </record>
</message>
```

在以上格式中，message、id 和 msg 这些标记是固定的，而 success 标记通过 Reader 对象的 successProperty 配置项定义，record 标记则通过 record 配置项定义。还要注意一个要点，要为 id 和 msg 两个标记创建模型，这样才能读取数据，其中 id 字段返回的数据为表单中字段的名称，而 msg 字段则为其错误信息。

下面通过一个示例演示如何进行服务器端验证。示例会在面板中使用水平布局划分两个

区域，第一个区域内放置一个以 JSON 作为返回格式的验证表单，第二个区域则放置一个以 XML 作为返回格式的验证表单。每个表单都会有 5 个字段依次使用 qtip、side、title、under 和元素 id 作为错误信息的显示位置，而第 5 个字段的错误信息显示元素，会添加一个标签。

使用模板页创建一个名称为 12-18.html 的页面文件。因为 XML 格式需要一个模型，所以先定义该模型，代码如下：

```
Ext.define('FieldError', {
    extend: 'Ext.data.Model',
    fields: ['id', 'msg']
});
```

然后完成面板的定义，代码如下：

```
Ext.create("Ext.panel.Panel",{
   title:" 表单的验证及错误显示方式 ",
   width:400,
   height:400,
   renderTo:Ext.getBody(),
   bodyStyle:"background:#DFE9F6",
   layout:"hbox",
   defaults:{xtype:"form",bodyPadding:5,
           bodyStyle:"background:#DFE9F6",flex:1,height:350,
           defaultType:"textfield",
           fieldDefaults:{labelWidth:80,
                   labelSeparator:": ",anchor:"0"
           },
           bbar:[
                   {text:" 保存 ",handler:function(){
                           var f=this.up("form").getForm();
                           if(f.isValid())
                                   f.submit();
                   }}
           ]
   },
   items:[
   {title:"JSON",
           url:"Validation.ashx?fm=json",
           items:[
           {fieldLabel:"field1",name:"field1"},
           {fieldLabel:"field2",name:"field2",msgTarget:"side"},
           {fieldLabel:"field3",name:"field3",msgTarget:"title"},
           {fieldLabel:"field4",name:"field4",msgTarget:"under"},
           {fieldLabel:"field5",name:"field5",msgTarget:"label1"},
           {xtype:"label",id:"label1"}
   ]},
   {title:"XML",
   url:"Validation.ashx?fm=xml",
   errorReader:Ext.create("Ext.data.reader.Xml",{
           model:FieldError,
           record:"error",
           successProperty:"success"
   }),
   items:[
```

```
                    {fieldLabel:"field1",name:"field1"},
                    {fieldLabel:"field2",name:"field2",msgTarget:"side"},
                    {fieldLabel:"field3",name:"field3",msgTarget:"title"},
                    {fieldLabel:"field4",name:"field4",msgTarget:"under"},
                    {fieldLabel:"field5",name:"field5",msgTarget:"label2"},
                    {xtype:"label",id:"label2"}
            ]}
    ]
})
```

要使用 XML 格式作为返回格式，必须定义 errorReader，并且将它指向一个 Reader 实例，在 Action 对象内不会自动创建它。接下来需要定义 model，以让其正确读取数据，而 record 的定义为 error，因而需要将 XML 格式代码中的 record 修改为 error。

下面来完成服务器端代码，因两个表单都使用同一个文件，使用 fm 参数来区分使用何种格式，所以需要在服务器端根据 fm 参数进行处理，以下是服务器端代码：

C#

```
public void ProcessRequest (HttpContext context) {
    string fm=context.Request.Params["fm"]??"";
    string[] field = new string[]{"field1","field2","field3","field4","field5"};
    if (fm.ToLower() == "json")
    {
        JObject errors = new JObject();
        for (int i = 0; i < field.Length; i++)
        {
            string v = context.Request.Params[field[i]] ?? "";
            if (v.Length <= 0)
            {
                errors.Add(field[i], "该项为必填项。");
            }
        }
        context.Response.ContentType = "text/javscript";
        if (errors.Count > 0)
        {
            context.Response.Write(new JObject(
                new JProperty("success",false),
                new JProperty("errors", errors)
            ));
        }
        else
        {
            context.Response.Write(new JObject(
                new JProperty("success", true),
                new JProperty("msg", "保存成功")
            ));
        }
    }
    else
    {
        XDocument doc = new XDocument();
        XElement errors = new XElement("errors");
        for (int i = 0; i < field.Length; i++)
        {
```

```
        string v = context.Request.Params[field[i]] ?? "";
        if (v.Length <= 0)
        {
            XElement xe = new XElement("error",
                new XElement("id", field[i]),
                new XElement("msg", new XCData("该项为必填项。"))
            );
            errors.Add(xe);
        }
    }
    if (errors.IsEmpty)
    {
        doc.Add(new XElement("message",
            new XElement("success",true),
            new XElement("msg", "保存成功")
        ));
    }
    else
    {
        doc.Add(new XElement("message",
            new XElement("success", false),
            errors
        ));
    }
    context.Response.ContentType = "text/xml";
    context.Response.Write(doc.ToString());
  }
}
```

Java

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
  String fm="";
  String[] field=new String[]{"field1","field2","field3","field4","field5"};
  if(request.getParameter("fm")!=null) fm=request.getParameter("fm");
  if(fm.equals("json")){
        int eCount=0;
     JsonObject jo=new JsonObject();
        JsonObject errors=new JsonObject();
        for(int i=0;i<field.length;i++){
              String v="";
              if(request.getParameter(field[i])==null){
                    v =request.getParameter(field[i]);
              }
              if(v.length()<=0){
                    eCount++;
                    errors.addProperty(field[i],"该项为必填项" );
        }
  }
  if(eCount>0){
        jo.add("success", new JsonPrimitive(false));
        jo.add("errors", errors);
  }else{
        jo.add("success", new JsonPrimitive(true));
```

```
                jo.addProperty("msg", "保存成功");
            }
            response.setContentType("text/javascript; charset=utf-8");
            response.getWriter().write(jo.toString());

        }else{
            Document document = DocumentHelper.createDocument();
            Element root = document.addElement("message");
            Element errors = DocumentHelper.createElement("errors");
            int c=0;
            for(int i=0;i<field.length;i++){
                String v="";
                if(request.getParameter(field[i])==null){
                    v =request.getParameter(field[i]);
                }
                if(v.length()<=0){
                    Element error= errors.addElement("error");
                    error.addElement("id").addText(field[i]);
                    error.addElement("msg").addCDATA("该项为必填项" );
                }
                //Element error= DocumentHelper.createElement("error");
                //errors..add(error);
        }
        if(errors.hasContent()){
            root.addElement("success").addText("false");
            root.addElement("total").addText(String.valueOf(c));
            root.add(errors);
        }else{
            root.addElement("success").addText("true");
            root.addElement("msg").addText("保存成功");
        }
        response.setContentType("text/xml; charset=utf-8");
        response.getWriter().write(document.asXML());
    }
}
```

在代码中，为了便于演示，通过循环来提取字段值，在实际中这并不是好的办法，所以要注意一下。

先看 JSON 处理部分，先创建一个 JSON 对象，当验证有错误的时候，就在对象中添加一个成员。当提取完数据后，C# 可根据 JSON 对象的 Count 属性判断是否有错误，而 Java 没有这个属性，只能通过一个统计变量来执行统计。如果有错误，则返回错误信息，否则返回成功信息。

对 XML 格式的处理也差不多，只是将 JSON 对象换成了 XML 的 Element 对象。

在浏览器中打开页面，在两个表单中单击“保存”后，将看到如图 12-20 所示的效果。注意图 12-20 中使用 title 作为错误信息显示的效果并不好，因此不建议这样使用。还有使用 under 方式时要注意，它会增加表单面板的高度，如果表单面板的高度是固定的，那么就会隐藏一些字段，所以使用 under 方式时一定要注意高度，预留空位。

注意 示例在4.1 Beta 1版本中执行时存在错误，原因是Reader对象的readRecords方法在success为false时不能读取数据。

12.5.2 为表单加载数据

图 12-20 服务器端验证的页面效果

为表单加载数据有两种方式，一种是先通过模型加载数据，然后调用FormPanel的loadRecord方法为表单加载数据；一种是直接使用表单的load方法加载数据。两种方式的主要区别是加载对象不同，模型使用Reader对象加载数据，而Load方法直接使用Ajax对象加载，因此，使用load方法时不需要定义模型。两者最终都会调用setValues方法将值加载到表单中。使用哪种方式视环境而定，例如，在使用Grid浏览数据时，如果需要使用模型加载全部数据进行预览，则会使用模型加载数据，这时只需要使用loadRecord方法为表单加载数据就行了，如果模型没有加载数据，则可使用load方法加载数据。还要注意的是，如果使用模型加载数据，需要为模型定义proxy配置项。

使用模型加载数据与使用load方法加载数据在使用JSON作为数据返回格式时是有些区别的，使用模型加载，只需要把所有数据的字段名称和值作为对象的成员返回即可；而使用load方法需要根据配置项root的设置（默认值为data），将数据对象作为data属性的值，并添加success属性，该属性值为true。

对XML格式的数据进行加载比较简单，无论是模型加载还是load方法加载，其数据格式是固定的，因为两者都需要通过Reader对象加载数据。此时数据格式是以message为根节点，然后根据successProperty配置项的定义，决定success属性是从message节点的属性读取（定义为“@success”），还是从success节点读取。而记录则根据record配置项的定义来决定如何读取，例如定义为data，则从data节点读取。记录中的字段则以字段名称作为data节点的子节点，值则为节点的值。

下面通过一个示例来演示如何使用两种方式将Northwind库中Products表的记录读取到表单中。

使用模板页创建一个名称为12-19.html的页面文件，然后先为Products表定义数据模型，代码如下：

```
Ext.define('Product', {
    extend: 'Ext.data.Model',
    fields: [
            {name:"ProductID",type:"int"},
            {name:"SupplierID",type:"int"},
            {name:"CategoryID",type:"int"},
            {name:"UnitPrice",type:"float"},
            'ProductName','QuantityPerUnit',
```

```
            {name:"UnitsInStock",type:"int"},
            {name:"UnitsOnOrder",type:"int"},
            {name:"ReorderLevel",type:"int"},
            {name:"Discontinued",type:"bool"},
            "CategoryName","CompanyName"

    ],
    idProperty:"ProductID",
    proxy:{
            type:"ajax",
        url:"Product_json.ashx?act=model",
            reader:{
                type:"json"
            }
        }
});
```

在模型中定义了 proxy 配置项，这样模型就可以通过 proxy 读取数据了。

复制一份模型代码，将模型名称修改为 ProductXML，然后修改 reader 的配置项，将 type 修改为 xml，添加 record 配置项，其值为 data。这模型是用来读取 XML 格式数据的。

接着分别为供应商和产品类别的 ComboBox 创建一个 Store，代码如下：

```
Ext.create("Ext.data.Store",{
    id:"SupplierStore",
    autoLoad:true,
    fields:[
            {name:"SupplierID",type:"int"},
            "CompanyName"
    ],
    idProperty:"SupplierID",
    proxy:{
            type:"ajax",
            url:"Supplier.ashx",
            reader:{
                    type:"json",
                    root:"data"
            }
    }
})

Ext.create("Ext.data.Store",{
    id:"CategoryStore",
    autoLoad:true,
    fields:[
            {name:"CategoryID",type:"int"},
            "CategoryName"
],
idProperty:"CategoryID",
proxy:{
            type:"ajax",
            url:"Category.ashx",
            reader:{
                    type:"json",
```

```
                root:"data"
            }
        }
    })
```

Store 只需要两个字段，autoLoad 配置项一定要加上，不然，在没有数据的情况下，为表单加载数据后，在 ComboBox 的列表中找不到对应的显示值，这样 ComboBox 会显示为空，所以必须在页面初始化的时候就要预加载数据。不过，如果 Store 是分页的，也可能找不到数据，这时的解决办法是为 Store 添加一个记录，然后再设置一次 ComboBox 的值，例如，如果不设置 SupplierStore 的 autLoad 为 true，则可在表单面板中监听 actioncomplete 事件，为 SupplierStore 添加一个记录，然后再次设置 ComboBox 的值，代码如下：

```
listeners:{
        actioncomplete:function(f,action){
                var store=Ext.StoreManager.lookup("SupplierStore");
                store.add({SupplierID:action.result.data.SupplierID,
                        CompanyName:action.result.data.CompanyName
                })
                f.findField("SupplierID").setValue(action.result.data.
                   SupplierID);
        }
}
```

这时候，在 Product 模型中的冗余字段 CompanyName 和 CategoryName 就发挥作用了。当然，这个办法也可以用在（不设置 autoLoad 配置项，或设置为 false 不自动加载）的情形，视具体的设计而定。

接着，定义一个使用 HBoxLayout 布局的面板，它将包含两个表单面板，分别用于加载 JSON 格式数据和 XML 格式数据，代码就不列了。

接着定义加载 JSON 数据的表单面板，代码如下：

```
{title:"JSON",
   items:[
          {xtype:"displayfield",fieldLabel:"产品编号",name:"ProductID"},
          {xtype:"textfield",fieldLabel:"产品名称",name:"ProductName",
                allowBlank:false,maxLength:40
          },
          {xtype:"combobox",name:"SupplierID",fieldLabel:"供应商",
                valueField:"SupplierID",displayField:"CompanyName",
                store:"SupplierStore",minChars:"2",forceSelection:false,

                allowBlank:true,listConfig:{
                loadingText: '搜索中 ...',emptyText: '没有匹配的供应商'
                }
},
{xtype:"combobox",name:"CategoryID",fieldLabel:"产品类别",
        valueField:"CategoryID",displayField:"CategoryName",
        store:"CategoryStore",forceSelection:true,queryMode:"local",
        allowBlank:false,listConfig:{
        loadingText: '搜索中 ...',emptyText: '没有匹配的产品类别'
        }
```

```
        },
        {xtype:"textfield",fieldLabel:"单位",name:"QuantityPerUnit",
                maxLength:20
        },
        {xtype:"numberfield",fieldLabel:"单价",name:"UnitPrice",hideTrigger:true,
                minValue:0,autoStripChars:true
        },
        {xtype:"displayfield",fieldLabel:"库存",name:"UnitsInStock"},
        {xtype:"displayfield",fieldLabel:"订购数量",name:"UnitsOnOrder"},
        {xtype:"displayfield",fieldLabel:"再订购量",name:"ReorderLevel"},
        {xtype:"checkbox",fieldLabel:"",boxLabel:'停产',
                name:'Discontinued',inputValue:true,hideEmptyLabel:false
        }
    ],
    bbar:[
        {text:"使用模型加载",handler:function(){
                var p=Ext.ModelManager.getModel("Product"),
                        f=this.up("form");

                p.load(27,{
                        success:function(product){
                                this.loadRecord(product);
                        },
                        scope:f
                })
        }},
        {text:"使用 load 加载",handler:function(){
                var f=this.up("form").getForm();
                f.load({
                url:'Product_json.ashx?act=load',
                params: {
                    id: 29
                },
                failure: function(form, action) {
                    Ext.Msg.alert("加载失败", action.result.msg);
                }

            })
        }}
    ]
},
```

要注意，代码中的字段名称（name）一定要与模型中的字段对应起来，否则，模型加载的数据不能正确地填充到字段里，除非通过代码将数据转换一次再加载到表单中。如果使用 load 方法加载数据，则可在服务器端把字段名称对应起来。

如果停产字段使用 Radio 字段，值就不可以为 true 或 false，不然利用它只会设置 RadioGroup 内第一个字段开关，就成为 CheckBox 字段了。停产字段的值也不可以用 0 或 1，会导致一样的结果。必须使用不能作为开关的值，例如“是”或“否”，不过，这样提交到服务器又要再处理，所以字段是布尔值的，还是使用 CheckBox 比较方法。

面板内定义了两个按钮，第一个按钮使用模型加载数据，采用的方法同样是 laod 方法，这里定义了作用域为 BasicForm 对象，这样，在 success 方法内，就可直接使用 loadRecord 方

法加载记录了。

第二个按钮直接调用 load 方法加载数据，因为直接使用 Ajax 对象，所以可以直接使用 params 来定义提交参数。

最后来完成使用 XML 格式加载的表单面板，直接复制 JSON 的代码就可以了，区别不大，修改的地方包括：将 title 修改为“XML”，添加 reader 配置项，其值为一个 Reader 对象实例，用于加载 XML 数据，代码如下：

```
reader:Ext.create("Ext.data.reader.Xml",{
    model:"ProductXML",
    record:"data",
    successProperty:"success"
}),
```

代码中的 model 配置项一定要定义，不然不知道要读取什么数据；record 也是必需的，默认是没有定义的，不然不知道从哪个节点开始读取记录；successProperty 的默认值就是 success，如果想从 message 节点的属性中读取 success 属性，可将该值设置为“@success”。

服务器端代码涉及 4 个文件，其重点在输出格式上，但是实现起来不难，与之前的示例雷同，在此就不列了，文件都在资源包中，可以自行研究。

在浏览器中打开页面，分别加载 4 次数据，可看到如图 12-21 所示的效果。

图 12-21　为表单加载数据的页面效果

通过 Firebug，可看到数据的加载的结果，注意格式。

以下是通过模型加载的 JSON 格式数据：

```
{
  "ProductID": 27,
  "ProductName": "Schoggi Schokolade",
  "CategoryID": 3,
  "SupplierID": 11,
  "UnitPrice": 43.9,
  "UnitsInStock": 49,
  "UnitsOnOrder": 0,
  "QuantityPerUnit": "100 - 100 g pieces",
  "ReorderLevel": 30,
  "Discontinued": false,
  "CategoryName": "Confections",
  "CompanyName": "Heli Süβwaren GmbH & Co. KG"
}
```

以下是通过 load 方法加载的 JSON 格式数据：

```
{
  "success": true,
  "data": {
    "ProductID": 29,
    "ProductName": "Thüringer Rostbratwurst",
    "CategoryID": 6,
    "SupplierID": 12,
    "UnitPrice": 123.79,
    "UnitsInStock": 0,
    "UnitsOnOrder": 0,
    "QuantityPerUnit": "50 bags x 30 sausgs.",
    "ReorderLevel": 0,
    "Discontinued": true,
    "CategoryName": "Meat/Poultry",
    "CompanyName": "Plutzer Lebensmittelgroβm rkte AG"
  }
}
```

这两种格式的主要区别就是后一种的数据在 data 属性中，并且需要设置 success 属性为 true。

以下是以 XML 格式返回的数据：

```
<message>
  <success>true</success>
  <data>
    <ProductID>27</ProductID>
    <ProductName>Schoggi Schokolade</ProductName>
    <CategoryID>3</CategoryID>
    <SupplierID>11</SupplierID>
    <UnitPrice>43.9000</UnitPrice>
    <UnitsInStock>49</UnitsInStock>
    <UnitsOnOrder>0</UnitsOnOrder>
    <QuantityPerUnit>100 - 100 g pieces</QuantityPerUnit>
    <ReorderLevel>30</ReorderLevel>
    <Discontinued>false</Discontinued>
```

```
      <CategoryName>Confections</CategoryName>
      <CompanyName>Heli Süβwaren GmbH &amp;amp; Co. KG</CompanyName>
    </data>
</message>
```

XML 格式的数据比较固定，关键是让 Reader 知道如何读取 success、记录这些数据，这些都需要通过配置项来设置。如果必须以 XML 作为数据格式，一定要掌握好 XmlReader 对象的配置项。

12.6　在表单中使用布局

12.6.1　分列显示表单的字段

为了充分利用屏幕的宽度，避免表单因高度过高而出现滚动条，一般都会将字段分列显示，比较常用的是两列或三列布局。要将字段分列显示，可使用水平布局。使用分列布局要注意的问题主要是使用 Tab 键进行切换时字段的顺序。对于分列后的布局，其最终的 DOM 节点结构是字段节点，它是作为列容器的子节点存在的，因而，使用 Tab 键进行导航，会先遍历一列内的子节点，再切换到下一列，其切换顺序是从上到下，再从左到右，这不太符合输入习惯，所以需要通过配置项改变这种顺序，根据输入习惯修改为从左到右的顺序，先完成一行的输入再切换到下一行。

下面通过一个示例演示如何实现分列显示表单的字段，使用模板页创建一个名称为 12-20.html 的页面文件，然后定义一个使用 VBoxLayout 布局的面板，该面板内包含两个表单面板，上面的面板将显示分两列布局的表单，下面的面板将显示分三列布局的表单，定义代码如下：

```
Ext.create("Ext.panel.Panel",{
    title:" 分列显示表单的字段 ",
    width:500,
    height:600,
    renderTo:Ext.getBody(),
    layout:{type:"vbox",align:"stretch"},
    defaults:{xtype:"form",flex:1},
    items:[
            {title:" 两列布局 "},
            {title:" 三列布局 "}
    ]
})
```

在布局定义中设置 align 为 stretch 会使表单面板填满面板的宽度。如果布局内的组件定义有很多相同项，要重复利用 defaults 配置项，尽量保持代码的简洁。在浏览器中打开页面，将看到如图 12-22 所示效果。

因为两个表单面板都将使用 HBoxLayout 布局进行分列，所以可把 layout 配置项加到 defaults 配置项中，加入代码如下：

```
layout:{type:"hbox",align:"stretch"}
```

要进行分列，需要使用到面板，并且它们的配置基本相同，所以可在 defaults 配置项中加入 defaults 配置项，用来定义分列的面板，代码如下：

```
defaults:{xtype:"panel",flex:1}
```

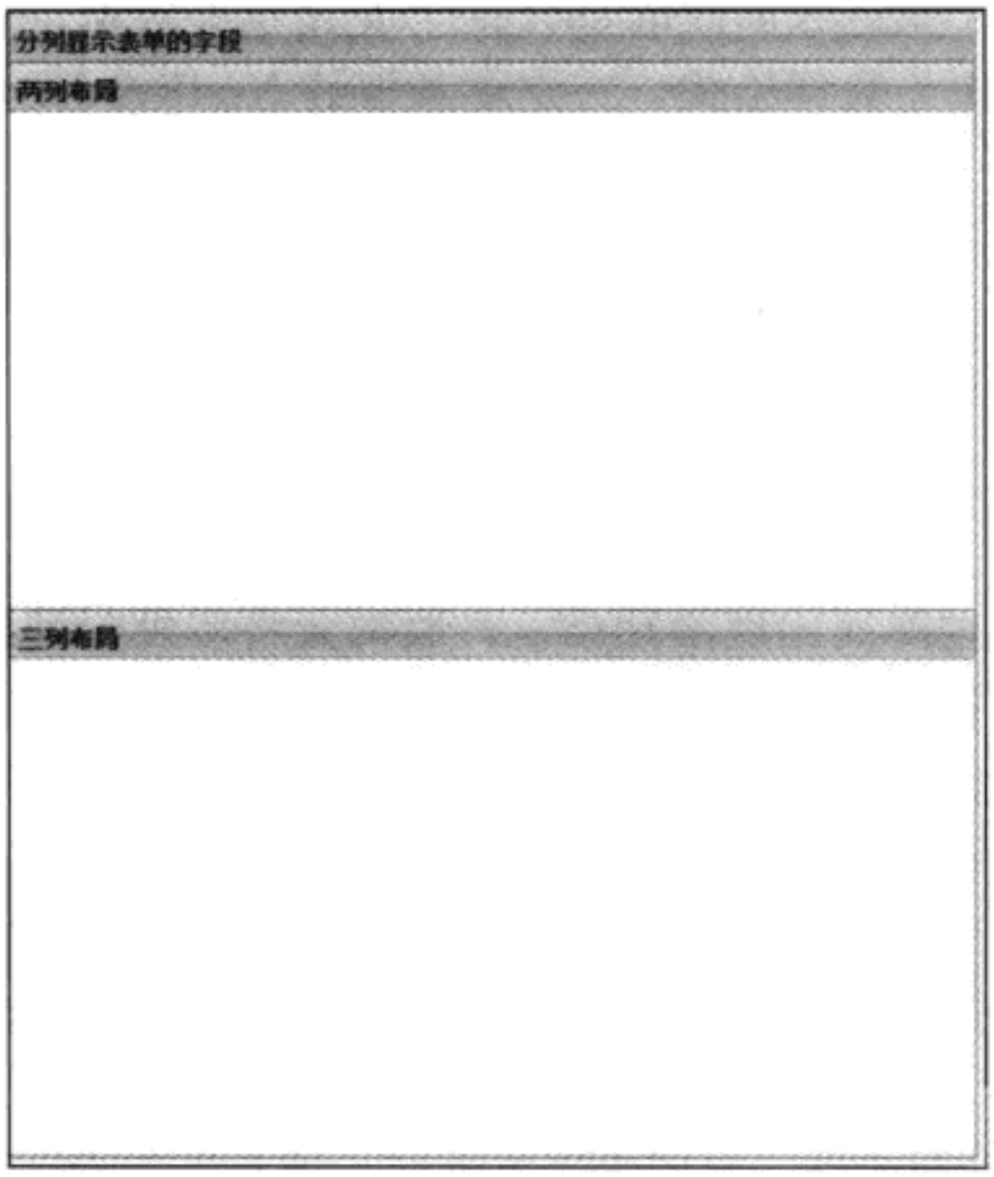


图 12-22　定义好的两个表单面板

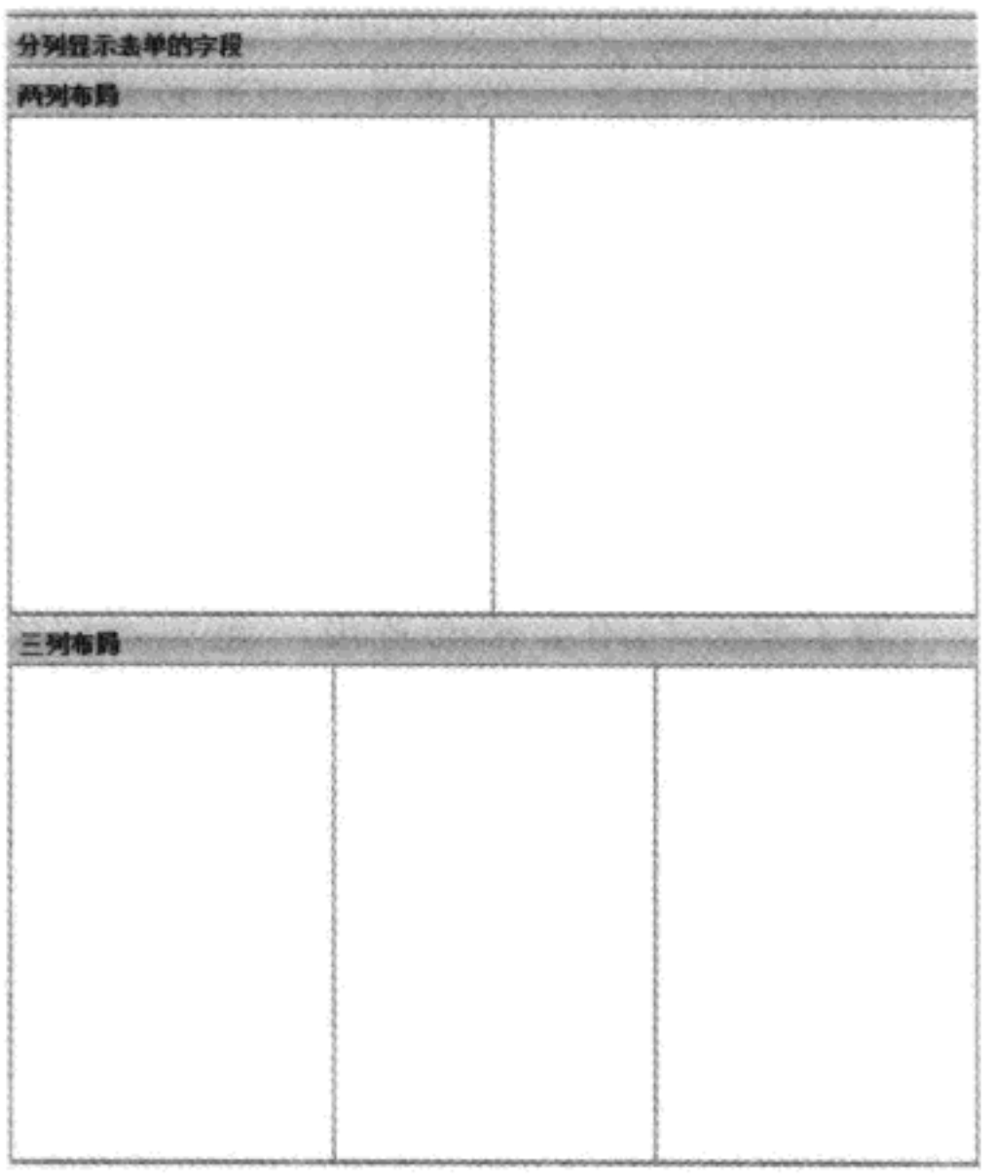


图 12-23　加入分列后的表单面板

现在可以在第一个表单面板内加入 items 配置项了，因而要分两列，可进行如下定义：

```
items:[{},{}]
```

第二个表单面板要分 3 列，在加入的 items 中加入多一个空对象就行了。完成加入后，刷新一下浏览器将看到如图 12-23 所示的效果，可以看到，表单面板的基本结构已经出现了。

现在加入字段，不过，在加入之前要清楚配置项 fieldDefaults 的定义位置。从 API 可以知道，fieldDefaults 是在 FieldAncestor 对象中定义的，只有表单面板才有 fieldDefaults 配置项，因此，fieldDefaults 必须在表单面板中定义。现在可以把常用的 fieldDefaults 配置项代码加入了，代码如下：

```
fieldDefaults:{anchor:"0",labelWidth:80,labelSeparator:": "}
```

如果分列的面板没定义布局类型，会使用自动布局，所以，为了使用方法的锚固布局，需要在分列的面板的 defaults 配置项中加入 layout 配置项，其值为 anchor，再加上 defaultType 配置项，其值为 textfield。现在可以在刚才加入的空对象中加入 items 配置项并加入字段定义了。完成后的代码如下：

```
{title:" 两列布局 ",
    items:[
            {items:[
                    {fieldLabel:" 字段 1",name:"field1"},
```

```
                {fieldLabel:"字段 3",name:"field3"}
            ]},
            {items:[
                {fieldLabel:"字段 2",name:"field2"},
                {fieldLabel:"字段 4",name:"field4"}
            ]}
    ]
},
{title:"三列布局",
    items:[
            {items:[
                {fieldLabel:"字段 1",name:"field1"},
                {fieldLabel:"字段 4",name:"field4"}
            ]},
            {items:[
                {fieldLabel:"字段 2",name:"field2"},
                {fieldLabel:"字段 5",name:"field5"}
            ]},
            {items:[
                {fieldLabel:"字段 3",name:"field3"},
                {fieldLabel:"字段 6",name:"field6"}
            ]}
    ]
}
```

在浏览器中打开页面将看到如图 12-24 所示的效果。将焦点移动到第一个面板表单的字段 1，然后使用 Tab 键进行切换，会看到光标的移动顺序是字段 1、字段 3、字段 2、字段 4，这不符合输入习惯。当然，使用 Fieldset 分组是允许这样的。这时候就要在字段中加入配置项 tanIndex 来设置切换顺序了，不过在值的安排上也要注意，如果一个页面中有多个这样的表单，例如本示例，则不能设置两个表单的字段 1 的 tanIndex 值都为 1。如果这样设置，那么光标就会在两个表单中移动，这是因为在页面中只认数字，不认位置。因此设置 tanIndex 值时一定要仔细，例如本示例，可考虑以 10 为基数进行设置，这样第一个表单的值从字段 1 到字段 4 依次是 11、12、13、14，而第二个表单值依次是 21 到 26，这样就不会出问题了。在单页面应用中，这个问题会比较突出，一定要小心。如果字段多，可将基数设置大点，一般从 100 开始就足够了，表单的字段超过 100 个，那真的让人感到无语，如果基数从 1000 开始，那简直太疯狂了。不过对于大型项目，当开发人员很多的时候会乱，所以数字要在设计的初始阶段就要分配好，使用一个类似 Ext.id 的计数器进行设置是个好办法。

从图 12-24 中还可看到，列之间太亲密了，还有边框。在面板的 defaults 配置项中加 border 配置项，其值为 false，再加入 bodyPadding 配置项，将其值设为 5，这样就行了。

刷新一下页面，将看到如图 12-25 所示的最终效果。

对表单进行布局其实不难，难的是要有耐心，一步步进行调试和调整。

12.6.2　使用 Fieldset 作为列容器

要将 FieldSet 作为列容器很简单，只要将 12.6.1 节示例的面板替换为 Fieldset 就可以了，不过修改后的效果会如图 12-26 所示，FieldSet 的边界会粘连在一起了。这时候将 bodyPadding

配置项修改为 margin 就行了，值不用变，最终效果会如图 12-27 所示。

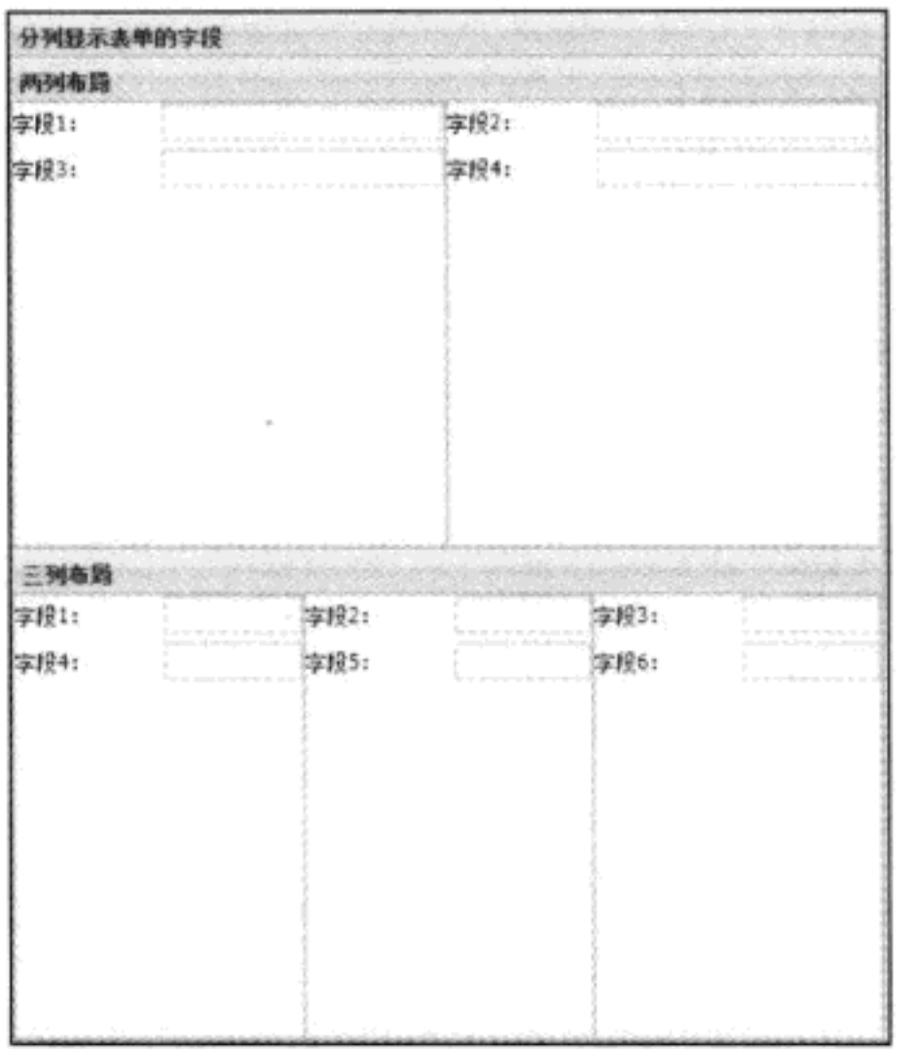


图 12-24　定义好字段的效果

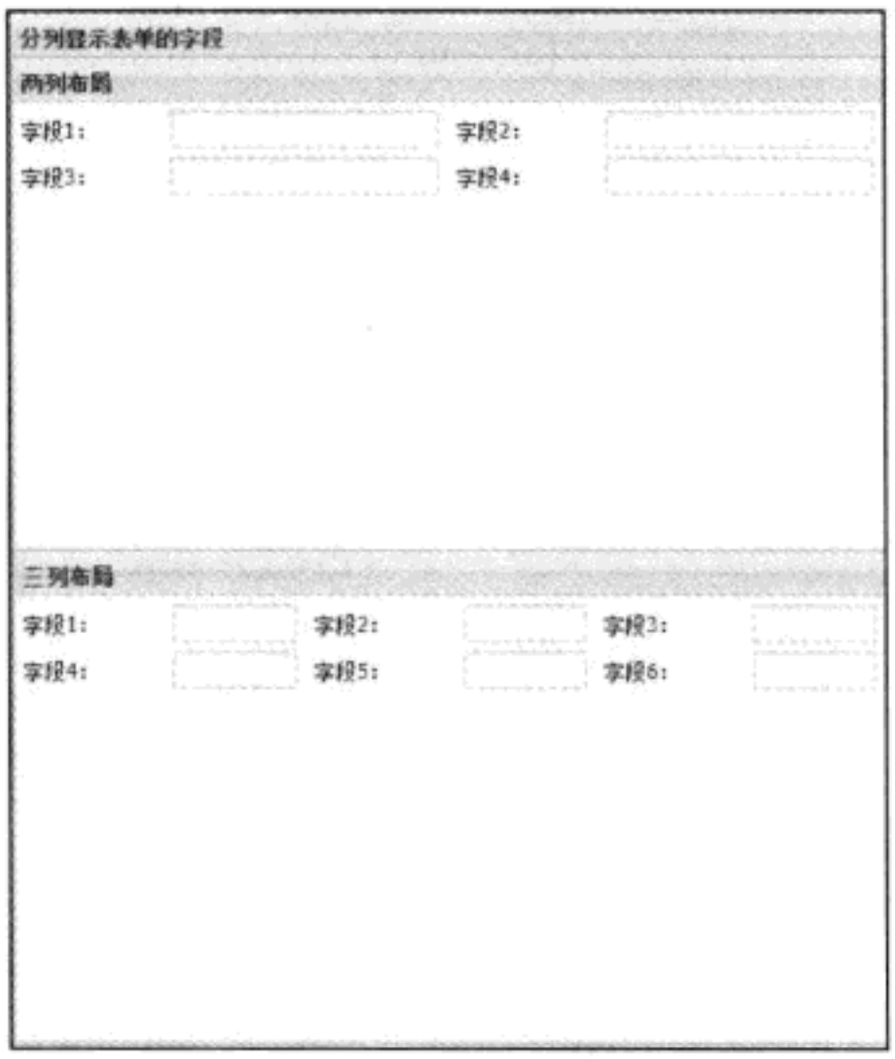


图 12-25　分列布局的表单的最终效果

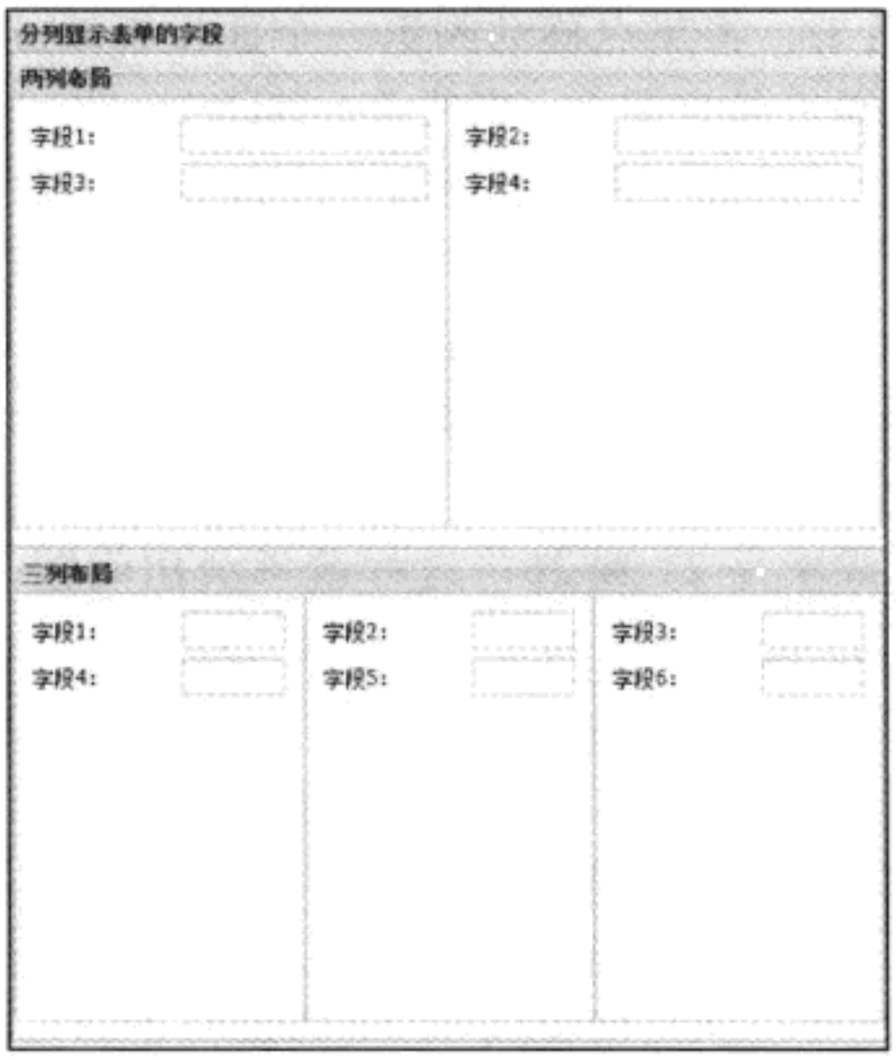


图 12-26　使用 Fieldset 作为列容器的效果

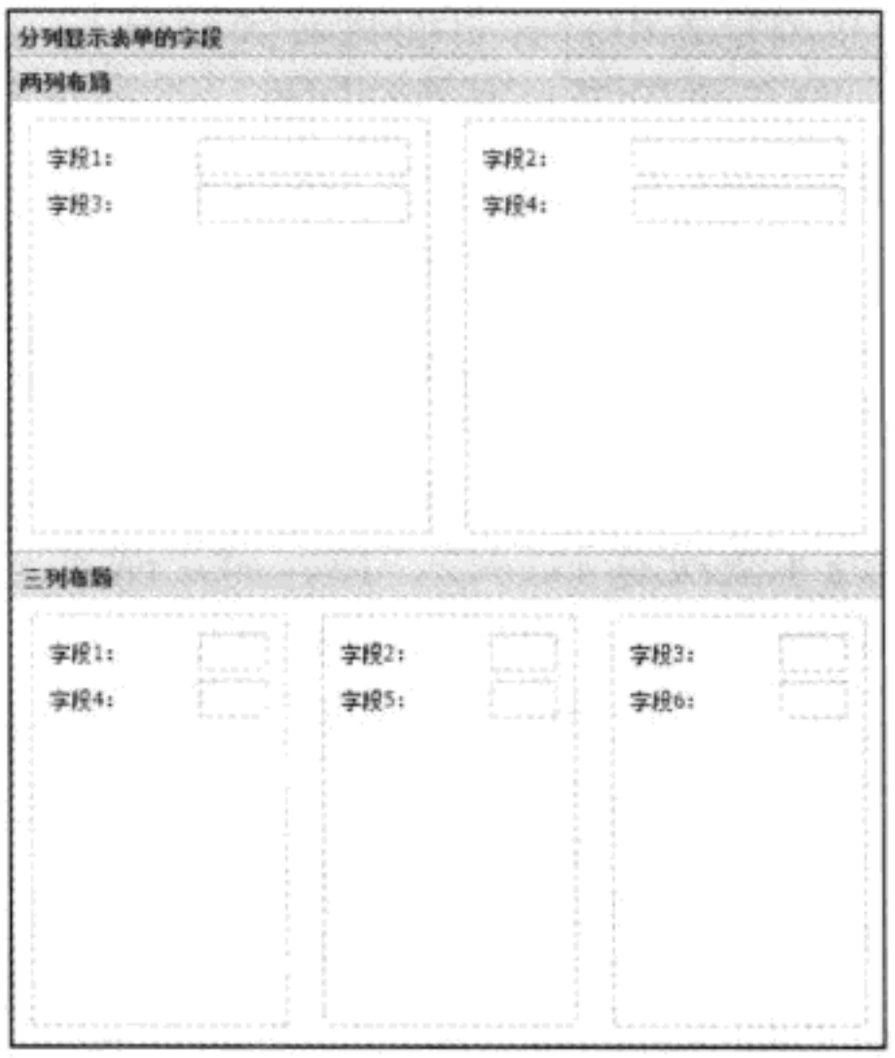


图 12-27　使用 FieldSet 作为列容器的最终效果

12.6.3　使用两列布局加 HtmlEditor 的表单

很多时候，为了给 HtmlEditor 字段足够的可视区域，会尽量将其余字段在顶部用两列显示，而余下区域就供 HtmlEditor 字段使用。下面通过一个示例演示如何实现这样的布局。

使用模版创建一个名称为 12-21.html 的页面文件，然后定义一个基本的表单面板，代码

如下：

```
Ext.create("Ext.form.Panel",{
    title:"使用两列布局加 HtmlEditor 的表单",
    width:500,
    height:600,
    renderTo:Ext.getBody(),
    layout:{type:"vbox",align:"stretch"},
    fieldDefaults:{anchor:"0",labelWidth:80,labelSeparator:": "},
    items:[
    ]
})
```

先将表单面板分成两行，顶部用来放置一般字段，底部用来放置 HtmlEditor 字段。现在要做的是估计一下顶部的高度，方法是，每行字段大约的高度为 30 像素，然后将行数乘以 30 就是顶部的高度了。示例中将有两行字段，所以顶部的高度为 60，余下为 HtmlEditor 字段的高度，使用配置项 flex，将其设置为 1，就可以填满余下的区域了。这些配置项的定义清楚后就可以先定义顶部和底部区域了，代码如下：

```
    {xtype:"panel",height:60,border:false},
    {xtype:"htmleditor",fieldLabel:"内容",name:"content",flex:1,labelAlign:"top"}
```

字段标签的默认位置在左边，因此在定义 HtmlEditor 时，注意将其 labelAlign 设置为 top，将标签放置在顶部，不过默认位置是期望的位置，那不用改。

最后完成字段的定义，将 12.6.1 节示例中面板的 defaults 定义和 items 定义复制过来就行了。

在浏览器中打开页面会看到 HtmlEditor 的标签没有与字段对齐，因而可在 HtmlEditor 加入配置项 margin，设置其值为“0 2 2 2”，就可以了。这值是调试出来的，只是需要耐心。完成对 HtmlEditor 的设置后，页面的最终效果将如图 12-28 所示。

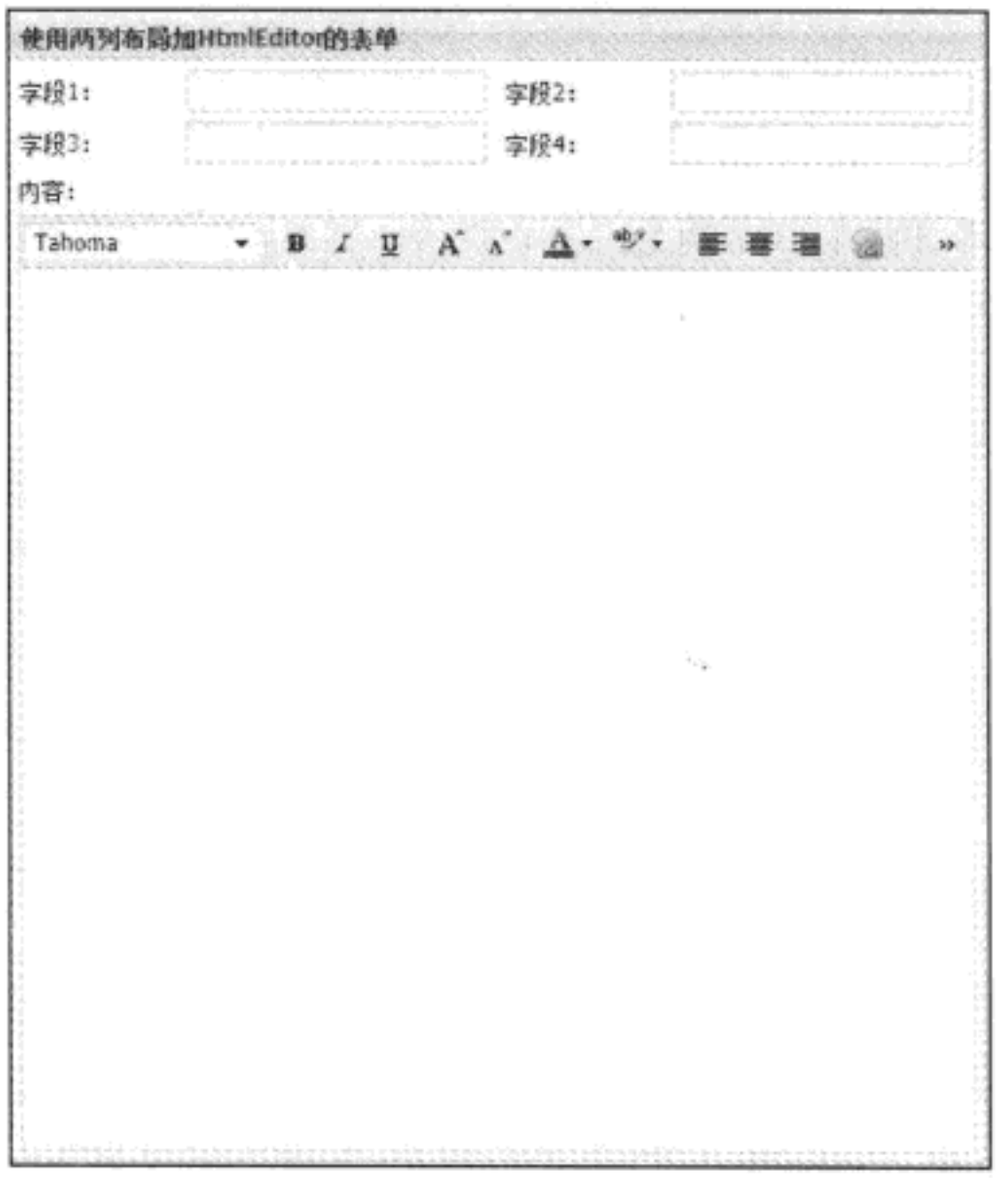


图 12-28　使用两列布局和 HtmlEditor 的表单的页面效果

12.6.4　在表单中使用标签页

当字段很多的时候，使用标签页将字段分类输入是好办法。使用标签页的关键是要设置标签页容器的布局为 fit，让每个标签页内的面板都能自适应容器的高度和宽度。

下面通过示例演示如何在表单中使用标签页，使用模板创建一个名称为 12-22. 页面文件，然后创建一个使用 FitLayout 布局的表单面板，代码如下：

```
Ext.create("Ext.form.Panel",{
```

```
    title:"在表单中使用标签页",
    width:500,
    height:400,
    renderTo:Ext.getBody(),
    layout:"fit",
    fieldDefaults:{anchor:"0",labelWidth:80,labelSeparator:": "},
    items:[
    ]
})
```

下面在 items 中定义一个标签页面板，由于各标签页内的面板配置项基本相同，因此加入 defaults 配置项，具体代码如下：

```
{xtype:"tabpanel",border:false,
   defaults:{xtype:"panel",border:false,layout:"anchor",
            defaultType:"textfield",bodyPadding:5
   },
   items:[
   ]
}
```

最后加入基本、附加和内容 3 个标签，基本和附加标签包含两个 Text 字段，而内容标签则包括一个 HtmlEditor 字段，代码如下：

```
{title:"基本",items:[
   {fieldLabel:"字段 1",name:"field1"},
   {fieldLabel:"字段 3",name:"field3"}
]},
{title:"附加",items:[
   {fieldLabel:"字段 2",name:"field2"},
   {fieldLabel:"字段 4",name:"field4"}
]},
{title:"内容",layout:"fit",items:[
   {xtype:"htmleditor",name:"content"}
]}
```

这样，使用标签页的表单面板就定义完成了，在浏览器中打开页面，将看到如图 12-29 所示的效果。

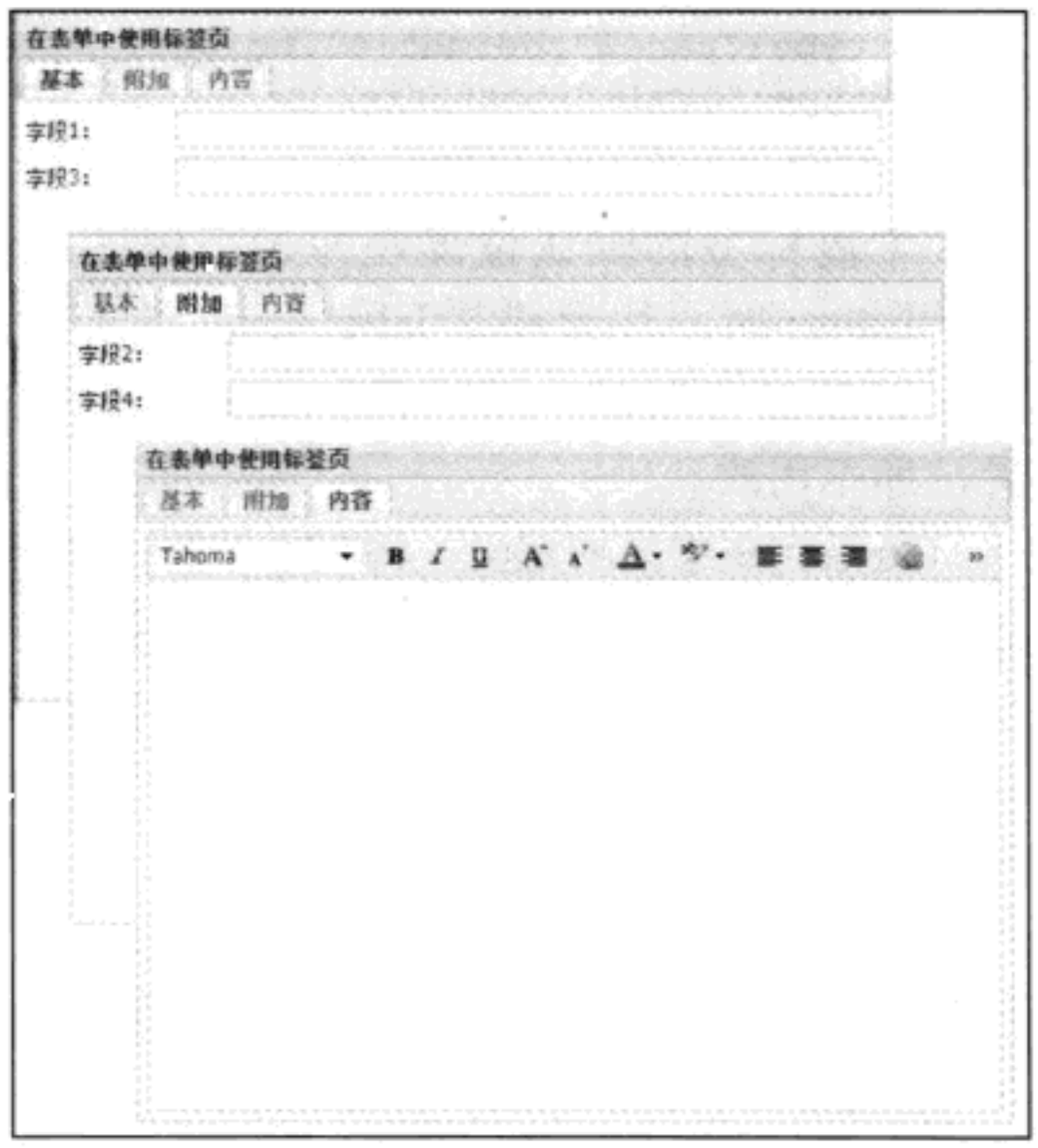


图 12-29　在表单中使用标签页的页面效果

在 Ext JS 3 中存在的 deferredRender 为 true 时，没有激活的标签页不会渲染字段，会造成设置值或提交值出现缺失的情况，在 Ext JS 4 这个问题已经解决了，因而不用为这问题操心了。

12.7　综合实例：实现 Products 表的管理功能

（1）功能描述

使用 Grid、表单等组件对 Northwind 数据库中的 Products 表进行管理。

（2）实现代码

在实现代码前，首先要考虑以下两个问题：

- 新增数据和编辑数据后如何更新 Grid 的数据？
- 使用何种方式删除数据？

对于第一个问题，编辑时产生的更新好解决，最好的办法是直接将更新后的数据写到 Store 中，或者直接刷新页面。对于新增数据，问题复杂点，主要解决办法有两个，一是刷新 Grid，把数据页转到有新记录的那页；二是新增数据时直接在 Store 中加入新增的记录。刷新 Grid 存在的问题是，可能需要改变用户当前的排序状态，才能看到最新记录；否则需要在后台计算好当前排序状态下新记录所在页，找到后不能直接将该页的数据返回，因为返回后，当前页数未必就是新记录所在页的页数，所以就要修改当前页的值了，这一系列问题肯定是可以解决的，但是成本太高了，因此笔者不建议使用这种办法。笔者认为刷新的方式，修改 Grid 的排序方式让新记录可以在第一页出现是比较适合的。不过不刷新，直接将新记录写在当前页，笔者认为这种方式更适合，用户只要保持在当前页操作，就能看到新的数据和当前页的数据，而要切换分页或进行排序时，已经想查看别的数据了，新数据已不是重点了。而这种方式又有两种处理方式，一种是由服务器端直接将保存的记录返回，然后将返回结果直接添加到 Store 中；一种是服务器端只返回新记录的 ID，然后组合表单的数据，再添加到 Store 中。返回全部记录的方式处理起来更简单直接，还可以保证数据没有隐性错误（有时候取值或保存时会造成数据丢失，在提交表单时，提交的数据并不都是完整的）。第二种方式其实没什么优势，只不过少了组织整个数据的返回过程。本示例为了多练习使用 Ext JS 处理数据，将采用第二种方式。

如果 Grid 是不允许排序的或不分页的，第一个问题可以忽略。

第二个问题的焦点是使用 Store 的 remove 方法删除记录，然后用同步的方式删除记录；还是自己组织数据，使用 Ajax 提交进行删除。使用 Store 的删除和同步功能的优点是客户端代码简单，基本两个语句就解决了，但是在服务器端要花的工夫就比较多，首先要为提交数据建立类，然后将提交的 JSON 数据转换为类，再从类中提取出记录的唯一值，然后根据唯一值在数据库中删除数据，最后，还要把这些数据完整返回到客户端来确认删除，不然，客户端会认为删除失败，又要做特殊处理。使用 Ajax 提交客户端代码要先把记录的唯一值收集起来，然后再使用 Ajax 提交，而服务器端代码就简单多了，根据提交的唯一值删除数据后，返回成功与否就行了。之后 Ajax 根据返回的数据调用 success 和 failure 定义的回调函数进行后续处理即可。Store 的删除和同步功能在之前的示例已经演示过了，因而本示例将演示使用 Ajax 的方法。

这两问题的解决方法在项目开发之初的设计阶段就应该制定好，以避免各开发人员都使用自己喜好的方式进行处理，从而造成用户使用上的混乱，而且开发之初设计好解决方法也便于以后的代码维护，因此项目设计人员应切记。

现在开始进入编码阶段，使用模板页创建一个名称为 12-23.html 的页面文件。第一步，当然是定义后模型，模型与 12.5.2 节示例的模型是一样的，把 proxy 配置项删除，然后为 UnitPrice、UnitsInStock、UnitsOnOrder、ReorderLevel 和 Discontinued 加上配置项 defaultValue，设置新建一

个模型实例使用的默认值就行了。

SupplierStore 和 CategoryStore 这两个 Store 可以从 12.5.2 示例复制过来，在表单的 ComboBox 中会用到。

下面定义 Grid 使用到的 ProductStore，代码如下：

```
Ext.create("Ext.data.Store",{
    storeId:"ProductStore",autoLoad:true,model:"Product",remoteSort:true,
     sorters:[{property :'ProductID',direction: 'DESC'}],
    proxy:{
            type:"ajax",
            url:"ProductManager.ashx?act=list",
            reader:{
                    type:"json",
                    root:"data"
            }
    }
})
```

以上定义中预设了使用 ProductID 降序排序，其余没什么特别。

接着把整个 Grid 的轮廓勾勒出来，代码如下：

```
Ext.create("Ext.Viewport",{
    layout:"fit",
    items:[
    {xtype:"grid",title:"产品管理",store:"ProductStore",id:"ProductGrid",
            selType:"checkboxmodel",selModel:{mode:"MULTI"},
            tbar:{xtype:"pagingtoolbar",store:"ProductStore",items:[
            "|",
            {text:"新增",handler:function(){
            }},
            {text:"编辑",disabled:true,id:"editButton",
                    handler:function(){
            }},
            {text:"删除",disabled:true,id:"delButton",
                    handler:function(){
            }}
            ]},
            columns:[
{text:'编号',dataIndex:'ProductID',titleAlign:"center"},
{text:'名称',dataIndex:'ProductName',titleAlign:"center",flex:1},
{text:'供应商',dataIndex:'CompanyName',titleAlign:"center",
            sortable:false,flex:1
},
{text:'类别',dataIndex:'CategoryName',titleAlign:"center",
            sortable:false,flex:1
},
{text:'单位',dataIndex:'QuantityPerUnit',titleAlign:"center",flex:1},
{xtype:"numbercolumn",text:'单价',dataIndex:'UnitPrice',titleAlign:"center",
            align:"right",format:"$0.00",width:60
},
{xtype:"numbercolumn",text:'库存',dataIndex:'UnitsInStock',align:"center",
            format:"0",width:60
},
```

```
{xtype:"numbercolumn",text:'订购量',dataIndex:'UnitsOnOrder',align:"center",
        format:"0",width:60
},
{xtype:"numbercolumn",text:'再订购量',dataIndex:'ReorderLevel',align:"center",
        format:"0",width:60
},
{xtype:"booleancolumn",text:'停产',dataIndex:'Discontinued',
        align:"center",trueText:"是",falseText:"否",width:60
}
],
listeners:{
                selectionchange:function(seltype,rs){
                        var length=!(rs.length>0);

                        Ext.getCmp("editButton").setDisabled(length);
                        Ext.getCmp("delButton").setDisabled(length);
                }
    }
    }
    ]
});
```

Grid 使用了 CheckedModel 作为选择模型，这是常用的选择模型了，允许多选。分页工具栏添加的“新增”、“编辑”和“删除”3 个按钮用来对记录进行操作。列定义在处理停产一列时除了将显示文字修改为是或否外，没什么特别的地方。不过，一般建议使用 CheckboxColumn，这样用户可以直接修改产品是否停产了。在 selectionchange 事件中，如果用户进行了选择，则启用“编辑”和“删除”按钮，否则禁用这两个按钮。这里有个问题，Grid 是允许多选的，所以在编辑的时候要考虑是编辑选择的第一个记录还是最后一个记录，本示例默认编辑第一个记录。

接着要完成弹出的编辑窗口的框架，代码如下：

```
Ext.create("Ext.window.Window",{id:"productWin",
    width:400,height:350,closeAction:"hide",modal:true,resizable:false,
    items:[
            {xtype:"form",bodyPadding:5,bodyStyle:"background:#DFE9F6",
            trackResetOnLoad:true,waitTitle:"请等待...",
            fieldDefaults:{labelWidth:80,labelSeparator:": ",anchor:"0"},
            items:[
            {xtype:"textfield",fieldLabel:"产品编号",name:"ProductID",
                    readOnly:true,readOnlyCls:"x-item-disabled"
            },
            {xtype:"textfield",fieldLabel:"产品名称",name:"ProductName",
                    allowBlank:false,maxLength:40
            },
            {xtype:"combobox",name:"SupplierID",fieldLabel:"供应商",
                    valueField:"SupplierID",displayField:"CompanyName",
            store:"SupplierStore",forceSelection:false,queryMode:"local",
                    allowBlank:false,minChars:1
                    },
            {xtype:"combobox",name:"CategoryID",fieldLabel:"产品类别",
                    valueField:"CategoryID",displayField:"CategoryName",
  store:"CategoryStore",forceSelection:true,queryMode:"local",
```

```
            allowBlank:false,minChars:1
            },
        {xtype:"textfield",fieldLabel:"单位",name:"QuantityPerUnit",
            maxLength:20
        },
        {xtype:"numberfield",fieldLabel:"单价",name:"UnitPrice",
            hideTrigger:true,minValue:0,autoStripChars:true
        },
        {xtype:"displayfield",fieldLabel:"库存",name:"UnitsInStock"},
        {xtype:"displayfield",fieldLabel:"订购数量",name:"UnitsOnOrder"},
        {xtype:"displayfield",fieldLabel:"再订购量",name:"ReorderLevel"},
            {xtype:"checkbox",fieldLabel:"",boxLabel:'停产',
                name:'Discontinued',inputValue:true,hideEmptyLabel:false
            }
    ],
    dockedItems: [
            {xtype: 'toolbar',dock:'bottom',ui:'footer',layout:{pack:"center"},
                items: [
                    {text:"保存",disabled:true,formBind: true,
                        handler:function(){
                        }
                    },
                    {text:"重置",handler:function(){
                    }}
                ]
            }]
            }
    ],
    listeners:{
                show:function(){
                    var f=this.down("form").getForm();
                    f.findField("ProductName").focus();
                }
        }
    })
```

为了窗口能重复使用，将其 closeAction 设置为 hide，这样窗口的 HTMl 代码只是隐藏起来，而不是删除了。配置项 modal 为 true 的作用是让窗口成为模态窗口，这样就只能操作窗口的内容了。配置项 resizable 为 false 则不允许用户修改窗口大小。

本示例的表单面板与 12.5.2 节示例的差不多，**不过要加入 trackResetOnLoad 和 waitTitle 两个重要的配置项**。trackResetOnLoad 的作用是当调用 reset 方法时，会将最后一次使用 setValues 方式的值作为重置的基础，其中最重要的原因是，在编辑状态下，单击 reset，如果不是最后一次设置的值，那么字段就全部是原始值，是空值，这绝对不是用户所期望的重置效果；waitTitle 的作用是在调用 submit 方法时显示的等待对话框的标题信息，如果不修改，将是“Please wait…”，所以必须改，除非设计的是英文界面的软件。

因为不会提交 Display 字段的数据，所以需要将 ProductID 的字段修改为 Text 字段，设置其为只读就行了，为了方便，将禁用时的样式“x-item-disabled”作为只读时使用的样式，将两个 ComboBox 的查询模式都修改为本地模式。

在窗口页脚工具栏中定义了两个按钮来保存和重置表单。保存按钮定义了配置项

formBind，其作用是当表单验证到没有无效值时会自动启用该按钮。不过，该方式还不够完善，在编辑数据的时候，如果没有修改字段，该按钮也会启用，因为值都是有效的，但是没有修改却提交了数据。这只能在数据提交的时候做一次验证了。

在 show 事件中，当窗口显示的时候，将光标移入 ProductName 字段。

最后要完成的是 5 个按钮的操作了，先完成新增按钮的操作。新增按钮要做的事情是创建一个新的记录，然后将其通过 loadRecord 方法加载到表单，修改窗口的标题为“添加新产品”，最后显示窗口。这里要注意，加载记录时不能使用 setValues 方法，原因是记录的数据结构不是 setValues 方法所使用的数据结构，setValues 方法使用的是一个以字段名称为属性，其值为字段的值的对象结构。“新增”按钮的代码如下：

```
var win=Ext.getCmp("productWin"),
    f=win.down("form"),
    m=Ext.ModelManager.getModel('Product');
f.loadRecord(new m());
win.setTitle("添加新产品")
win.show();
```

从 ModelManager 取到模型后，新建一个记录，然后调用 loadRecord 方法就行了。利用窗口的 setTitle 方法可重新设置窗口标题。最后调用 show 方法显示窗口。

接着完成“编辑”按钮，此时要做的事情是先判断是否有选择记录（保险起见，还是要做检查的），如果有，取第一个记录，然后通过 loadRecord 方法将其加载到表单，最后显示窗口，代码如下：

```
var grid=this.up("gridpanel"),

    rs=grid.getSelectionModel().getSelection(),
    win=Ext.getCmp("productWin"),
    f=win.down("form");
if(rs.length>0){
    win.setTitle("编辑产品："+rs[0].data.ProductName);
    f.loadRecord(rs[0]);
    win.show();
}else{
    Ext.Msg.alert("提示信息","请选择一条记录进行编辑。");
}
```

删除按钮就有点啰嗦了，先要使用提示框，让用户确认是否删除选择的记录，如果是，则获取选择记录的 ProductID，然后将这些组合起来通过 Ajax 将其提交到服务器端进行处理，处理完成后，通过回调函数 success 和 failure 进行后续处理，代码如下：

```
var grid=this.up("gridpanel"),
    rs=grid.getSelectionModel().getSelection();
if(rs.length > 0){
    var content=["确定删除以下产品？ "];
    for(var i=0;ln=rs.length,i<ln;i++){
            content.push(rs[i].data.ProductName);
    }
    Ext.Msg.confirm("删除记录",content.join("<br/>"),function(btn){
```

```
if(btn=="yes"){
        var me=this,store=me.store,ids=[];
        rs=me.getSelectionModel().getSelection();
        for(var i=0;ln=rs.length,i<ln;i++){
        ids.push(rs[i].data.ProductID);
        }
        Ext.Ajax.request({
        url:"ProductManager.ashx?act=del",
        params:{id:ids},
        success:function(response,opts){
                var obj = Ext.decode(response.responseText);
    if(obj.success){
          var msg=[].concat(["以成功删除以下产品: "],obj.msg);
          msg.push("----------");
          msg.push("列表将刷新。");
          Ext.Msg.alert("删除成功",msg.join("<br/>"),function(){
          Ext.StoreManager.lookup("ProductStore").load();
          });
          }else{
                  Ext.Msg.alert("错误",obj.msg);
          }
},
      failure: function(response,opts){
          Ext.Msg.alert('错误',
              '状态:'+response.status+': '+
              response.statusText);
      }
        })
}
    },grid)
}
```

在使用 MessageBox 时，记得设置作用域，见代码中的粗体部分，这样在其响应函数内的 this 就可以直接指向设置的对象了。

在第 7 章，我们知道 Ajax 对象只有在页面返回请求的状态码为 200 到 300（不包含 300）或 304 时才会执行 success 函数，因此需要自己判断操作是否成功。先将服务器端返回的数据（responseText）转换为 JSON 对象，如果其 success 为 true，说明删除成功，在提示用户后，重新加载 ProductStore 刷新数据；否则，显示属性 msg 内的错误信息。而在 failure 函数内显示的错误信息只有状态码代码的错误信息。

在完成保存前，先完成简单的重置按钮代码，很简单，就是调用 reset 方法，代码如下：

```
var f=this.up("form").getForm();
f.reset();
```

最后完成“保存”按钮，此时首先要做的是验证值是否有效，如果有效，还要验证是否有修改数据，如果有才提交新增的记录，确定有修改才会启用保存按钮，所以没有在修改数据，肯定是在编辑数据。如果提交成功后可使用 BasicFrom 的 getRecord 方法返回记录，不过这方法返回的是只有原始值的记录，因此还需要调用 updateRecord 将数据回写到记录后返回的才是修改后的值。这样处理后的数据其实还不完整，因为字段中没有 CompanyName

和 CategoryName 字段，所以记录中这两个字段是没有值的，需要使用 ComboBox 的 getRawValue 方法获取显示值，然后使用 set 方法将其设置到记录。对于新增的记录，还需要将服务器端返回的 ProductID 值设置到记录中，然后调用 insert 方法将其插入 Grid 的第一行。而编辑数据时的记录本身就是 Store 中的那个记录，因此不需要对其做任何处理，更新值就可以了。无论是新增的还是编辑的记录，最后都需要调用 commit 方法确认修改，这样在 Grid 中才不会显示该记录还没保存。

最后，还有一个问题要在设计阶段制订好处理策略，新增或编辑记录后，是立刻关闭窗口，还是保持窗口，让用户继续新增记录？笔者的习惯做法是，在新增记录的时候保持窗口，让用户继续新增记录，而在编辑的时候，直接关闭窗口，将操作交还用户来选择。好的策略是增加一个保存后新增的按钮，这样用户就多一个选择。

错误的处理主要是显示错误信息。

“保存”按钮的代码如下：

```
var grid=Ext.getCmp("ProductGrid"),
    form=this.up("form");
    f=form.getForm(),
    id=f.findField("ProductID").getValue();
if(f.isValid() && f.isDirty()){
    f.submit({
            url:"ProductManager.ashx?act="+((id>0)?"edit":"add"),
            waitMsg:"正在保存 ...",
          success: function(form, action){
              var result = action.result,
              f=this.getForm(),
              companyname=f.findField("SupplierID").getRawValue(),
              categoryname=f.findField("CategoryID").getRawValue(),
              store=Ext.StoreManager.lookup("ProductStore"),
              rec=f.getRecord();
   f.updateRecord(rec);
rec.set("CompanyName",companyname);
rec.set("CategoryName",categoryname);
   if(result.ProductID){
              rec.set("ProductID",result.ProductID)
              store.insert(0,rec);
              rec.commit();
                      m=Ext.ModelManager.getModel('Product');
                      f.loadRecord(new m());
   }else{
              rec.commit();
              Ext.getCmp("productWin").close();
   }
   },
   failure: function(form, action){
         if (action.failureType === "connect") {
               Ext.Msg.alert('错误',
                  '状态:'+action.response.status+': '+
                  action.response.statusText);
               return;
       }
         if(action.result){
```

```
                if(action.result.msg)
                Ext.Msg.alert('错误', action.result.msg);
                }
                },
            scope:form
    });
}else{
    Ext.Msg.alert("修改","请修改数据后再提交。")
}
```

在代码中，url 中 act 的值是由 ProductID 的值决定，这基于的是新增记录的 ProductID 值都等于 0 的情况，如果是其他情况，则具体情况具体分析。定义配置项 waitMsg 可显示遮蔽，从而避免提交期间用户进行多次操作。

与 Ajax 提交有所不同，当返回的结果的 success 为 false 时，会调用 failure 回调函数，因此所有 success 为 false 的情况都要在这里处理。如果是验证错误，则不用提示，在表单中可以看到错误提示；如果不是验证错误，则提示错误信息。

本示例的服务器端代码，基本就是之前示例的服务器端代码的综合，在此就不多说了，读者可以通过资源包中的代码进行研究。

（3）页面效果

在浏览器中打开页面，将看到如图 12-30 所示的效果。

产品管理

第 1 页,共 4 页 | 新增 | 编辑 | 保存

编号	名称	供应商	类别	单位	单价	库存	订购量	再订购量	停产
77	Original Frankfurter grü	Plutzer Lebensmittelgro	Condiments	12 boxes	$13.00	32	0	15	否
76	Lakkalikööri	Karkki Oy	Beverages	500 ml	$18.00	57	0	20	否
75	Rhönbräu Klosterbier	Plutzer Lebensmittelgro	Beverages	24 - 0.5 l bottles	$7.75	125	0	25	否
74	Longlife Tofu	Tokyo Traders	Produce	5 kg pkg.	$10.00	4	20	5	否
73	Röd Kaviar	Svensk Sjöföda AB	Seafood	24 - 150 g jars	$15.00	101	0	5	否
72	Mozzarella di Giovanni	Formaggi Fortini s.r.l.	Dairy Products	24 - 200 g pkgs.	$34.80	14	0	0	否
71	Flotemysost	Norske Meierier	Dairy Products	10 - 500 g pkgs.	$21.50	26	0	0	否
70	Outback Lager	Pavlova, Ltd.	Beverages	24 - 355 ml bottles	$15.00	15	10	30	否
69	Gudbrandsdalsost	Norske Meierier	Dairy Products	10 kg pkg.	$36.00	26	0	15	否
68	Scottish Longbreads	Specialty Biscuits, Ltd.	Confections	10 boxes x 8 pieces	$12.50	6	10	15	否
67	Laughing Lumberjack L	Bigfoot Breweries	Beverages	24 - 12 oz bottles	$14.00	52	0	10	否
66	Louisiana Hot Spiced O	New Orleans Cajun Del	Condiments	24 - 8 oz jars	$17.00	4	100	20	否
65	Louisiana Fiery Hot Pep	New Orleans Cajun Del	Condiments	32 - 8 oz bottles	$21.05	76	0	0	否

图 12-30　本示例的页面效果

单击“新增”按钮，然后输入如图 12-31 所示内容，单击“保存”按钮后，关闭窗口将看到 Grid 在第一行中添加了如图 12-32 所示的一条记录，选择它后，单击“删除”按钮，将看到如图 12-33 所示的确认窗口，单击“是”按钮后，等服务器端处理完成后，将看到如图 12-34 所示的提示信息，将此提示框关闭后会刷新 Grid。

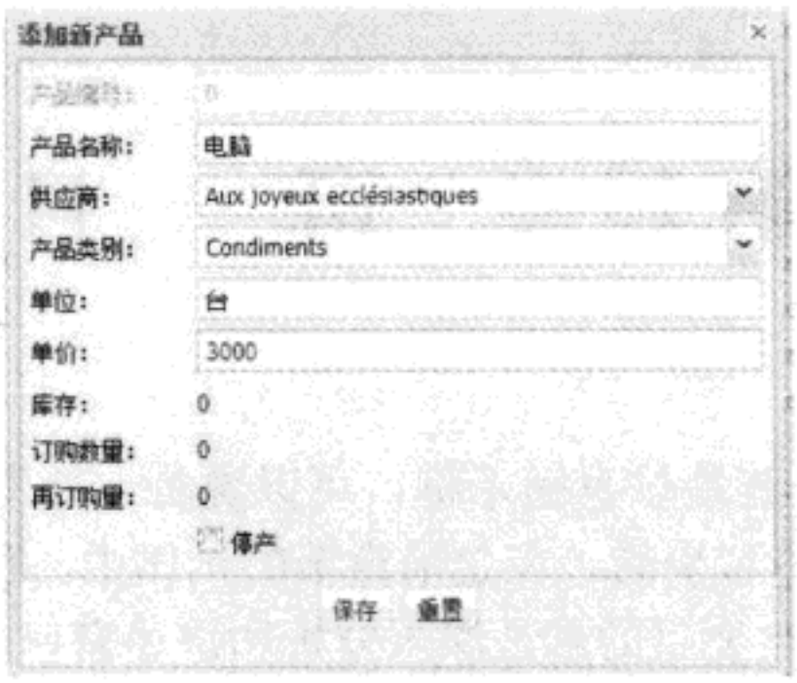


图 12-31　新增记录的内容

编号	名称	供应商	类别	单位	单价	库存	订购量	再订购量	停产
108	电脑	Aux Joyeux ecclésiastiq	Condiments	台	$3000.00	0	0	0	否
77	Original Frankfurter grü	Plutzer Lebensmittelgrc	Condiments	12 boxes	$13.00	32	0	15	否

图 12-32　新增记录后 Grid 的显示

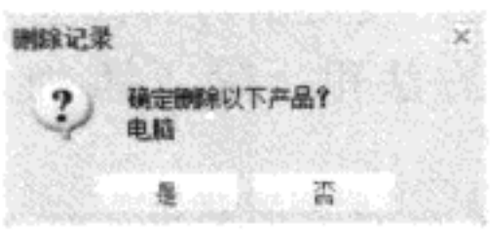


图 12-33　确认删除记录的显示

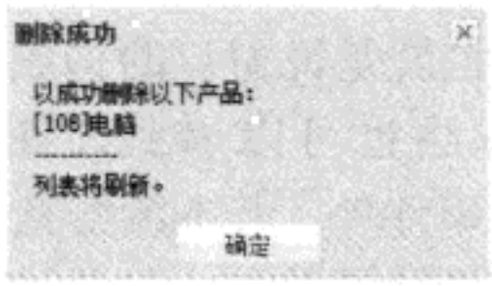


图 12-34　记录删除后的提示信息

12.8　本章小结

表单涉及的组件很多，使用起来也相当的复杂多变，因此要使用好表单不是一件轻松的事情，其中会涉及很多设计上问题，因此好的表单很多时候是在设计阶段就定义好了的，功能上实现的难度多半在设计阶段已经确定。多花点工夫在设计上是必要的。当然，熟练地掌握各种字段的用法，以及 BasicForm 对象、FormPanel 对象等的配置项、属性、方法和事件也是很重要的。

笔者建议在学习表单时应该尽量抽时间把表单涉及的对象的 API 仔细看一遍，不需要像背书一样看，大概知道有哪些配置项，有哪些属性方法就可以了，能够在使用时第一时间想到该用什么配置项、属性、方法或事件，那么这已经成功一半了。

第13章　窗　　口

窗口的应用太广泛了，尤其是信息提示框，可以说是无处不在，对于 Ext JS 这样提供整体 UI 的开发框架，当然是必不可少的。在之前章节的示例中，窗口已经使用过了，只是没有深入进行分析和讲述，本章的主要目的就是深入分析和讲述窗口的构成和使用。

13.1　窗口：Ext.window.Window.

13.1.1　窗口的构成

Window 对象派生于 Panel 对象，因而，简单来说，窗口就是一个浮动的、可调整大小及可拖动的面板，也就是说窗口的构成与面板是一样的，其使用与面板也大同小异。这个从其 initComponent 方法也可以看到，在 initComponent 方法里，除了添加 resize、maximize、minimize 和 restore 事件外，基本流程就是面板的流程。

当然，为了实现可调整大小以及拖动的功能，需要在面板的基础上添加一些操作。而最重要的，当然是通过 Stateful 对象记录窗口的状态。记录状态的事件包括 maximize、restore、resize 和 dragend，涵盖了窗口的最大化、最小化、调整大小和拖动等操作。

窗口是使用 ComponentDragger 对象实现拖动功能，在 initDraggable 方法中会创建其实例。

在 addTools 方法中，会根据配置项 minimizable 和 maximizable 的定义，显示最小化、最大化和恢复按钮。

以上就是窗口的构成及其功能，除了最小化、最大化和拖动等功能外，其他的与面板没什么区别。

13.1.2　窗口的配置项、属性、方法和事件

1. 配置项

- animateTarget：设置窗口显示时动画开始的目标，其值可以是模板元素也可以是其 id。默认值为 null。
- baseCls：应用于窗口的基本样式，默认值是“x-window”。
- closable：布尔值，默认值为 true，会在右上角显示关闭图标，允许窗口关闭。
- collapse：布尔值，如果值为 true，窗口初始状态是折叠的。默认值为 false，窗口初始状态是展开的。

- constrain：布尔值，如果设置为 true，会限制窗口只能在其容器内移动，默认值为 false，允许窗口在任何位置移动。默认情况下，窗口会渲染到 Body 元素下。通过 renderTo，可将其渲染到指定的元素下。
- constrainHeader：布尔值，如果设置为 true，窗口的标题栏只能在其容器内移动，默认值为 false，窗口可在任何位置移动。
- defaultFocus：设置窗口显示后，其焦点位置。值可以是页脚工具栏按钮的索引，组件配置项 itemId 的值，或者是组件实例。
- draggable：布尔值，默认值为 true，允许拖动窗口。如果值为 false，则不允许拖动窗口。
- expandOnShow：布尔值，默认值为 true，在窗口显示时，会展开窗口。如果值为 false，会保持窗口的当前状态。
- hidden：默认值为 true，窗口渲染时是隐藏的，hide 方法会被立即调用。
- maximizable：布尔值，如果为 true，会在右上角显示最大化图标及允许用户最大化窗口。默认值为 false。
- maximized：布尔值，如果为 true，窗口在初始显示时会以最大化方式显示。默认值为 false。
- minimizable：布尔值，如果为 true，会在右上角显示最小化图标，允许用户最小化窗口。默认值为 false。要注意，最小化按钮图标的行为需要自己进行定义，默认状态下是没有最小化操作的，只是显示图标而已。
- modal：布尔值，如果为 true，窗口显示为模态窗口。
- onEsc：复写 onEsc 函数，重新定义 Esc 键的作用。默认情况下，按下 Esc 键会关闭窗口。
- plain：布尔值，如果设置为 true，会渲染窗口的主体为透明背景。默认值为 false，会以一个比 frame 颜色浅的颜色作为窗口主体的背景。
- resizable：默认值为 true，允许用户调整窗口大小，为 false 则禁止调整大小。值也可以为 Resizer 对象的配置对象，重新设置调整大小的方式。
- x：设置窗口左上角的 x 坐标。
- y：设置窗口左上角的 y 坐标。

2. 属性

- dd：指向 ComponentDragger 对象示例。

3. 方法

- getFocusEl：获取 defaultFocus 配置项定义的元素。
- maximize：最大化窗口。
- minimize：最小化窗口。
- restore：恢复窗口最大化前的状态。
- toggleMaximize：在 minimize 和 restore 之间进行切换。

4. 事件

- activate：当窗口成为活动窗口时会触发该事件。

- deactivate：当窗口变成非活动窗口时会触发该事件。
- maximize：当窗口最大化后会触发该事件。
- minimize：当窗口最小化后会触发该事件。
- resize：当窗口调整大小后会触发该事件。
- restore：当窗口恢复最大化前的状态后，会触发该事件。

13.1.3 使用窗口

1. 最简单的窗口

打开模板页，在命令行中输入以下命令：

```
Ext.create('Ext.window.Window', {
    title:"最简单的窗口",width:300,height:300
}).show();
```

运行后将在屏幕中间看到如图 13-1 所示的效果。其实标题、宽度和高度都可以不要，不过这样显示起来就不大好看了。

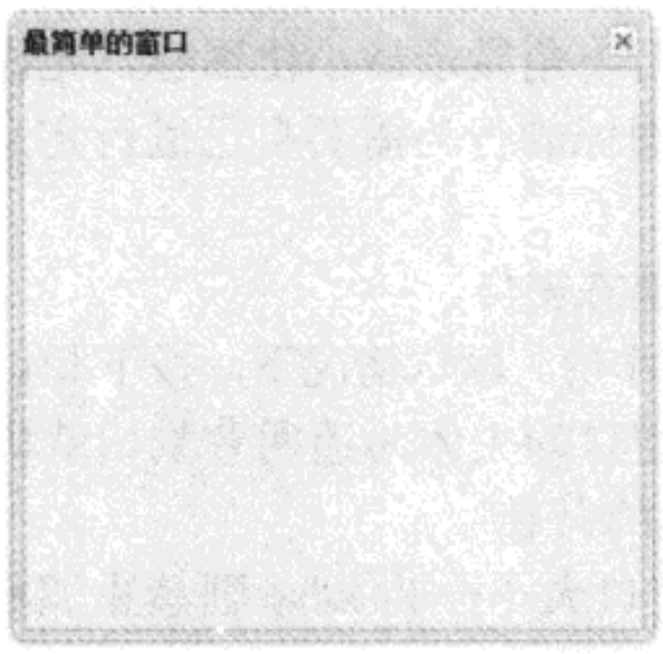


图 13-1　最简单的窗口的效果

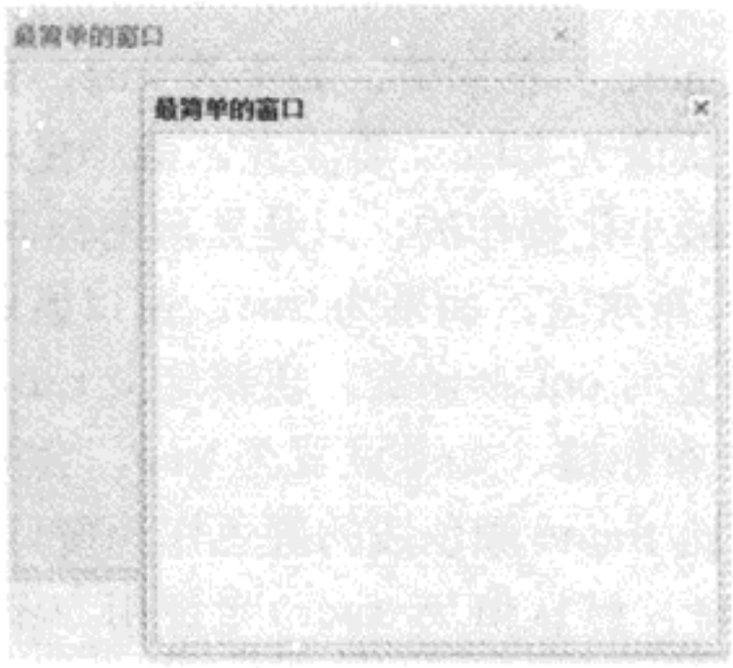


图 13-2　模态窗口

2. 模态窗口

在刚才的窗口定义中加入配置项 modal，值为 true，再次运行，并拖动一下窗口，将看到如图 13-2 所示的效果。因为第二个窗口是模态窗口，所以其显示后，会把页面遮蔽起来，包括刚才定义的窗口。

3. 可最小化、最大化的窗口

关闭模态窗口，然后把窗口定义中的 modal 配置项去掉，加入 maximizable、maximized 和 minimizable 配置项，值都设为 true。运行后，将会看到窗口初始时是最大化的，而右上角从左到右依次有最小化、恢复和关闭 3 个图标，单击恢复图标，可将窗口恢复到图 13-1 所示的状态。单击最小化图标，窗口并没有任何动作，这是因为默认情况下，没有定义最小化的行为，这个需要根据应用的情况自行定义。

4. 自定义起始位置的窗口

在窗口定义中去掉 maximizable、maximized 和 minimizable 配置项，加入配置项 x 和 y，值为 50，运行后，可以看到窗口不是处于页面的中央，而是左上角，在窗口的 HTML 代码中，可以看到窗口 div 的样式 left 和 top 值都是 50。

5. 不允许拖动和调整大小的窗口

在窗口定义中加入配置项 draggable 和 resizable，值都为 false，运行后将发现新窗口是不能拖动和调整大小的。

将 resizable 的值修改为以下值：

```
{handles:"s e se"}
```

运行后，可看到窗口只能往下和往右调整大小。

6. 可重用的窗口

把刚才创建的窗口全部关闭，然后在 Firebug 的 HTML 面板中，可看到窗口的生成的 HTML 代码都没了，这说明窗口默认情况只能使用一次。在刚才的窗口定义中加入 closeAction 配置项，其值为 hide。运行后，注意观测 HTML 面板。关闭窗口后，看到窗口的 Style 属性中，其 visibility 值为 hidden，说明窗口只是隐藏了。

7. 限制窗口的移动范围

刷新一下模板页，然后输入以下代码：

```
var p=Ext.create("Ext.panel.Panel",{
    renderTo:Ext.getBody(),
    width:300,height:300
});
Ext.create("Ext.window.Window",{
    title:"constrain",renderTo:p.getEl(),
    width:50,height:50,constrain:true
}).show()
Ext.create("Ext.window.Window",{
    title:"constrainHeader",renderTo:p.getEl(),
    width:50,height:50,constrainHeader:true
}).show()
```

运行后，可看到标题为 constrain 的窗口，只能在面板定义的范围内移动，不能超出范围，而标题为 constrainHeader 的窗口，标题栏不能移除面板。

要限制窗口的移动范围，要点是将窗口渲染到容器的元素内，然后设置 constrain 或 constrainHeader 为 true。

13.1.4 在窗口内使用布局

窗口派生于面板，是面板的一种。清楚这点，就能很好地在窗口中应用布局了。例如，要在窗口中使用边框布局，就当它是面板那样使用就行了，代码如下：

```
Ext.create("Ext.window.Window",{
```

```
        width:500,height:300,layout:"border",
        items:[
            {xtype:"panel",region:"west",split:true,width:200},
            {xtype:"panel",region:"center"}
        ]
}).show()
```

运行后，可看到如图 13-3 所示的效果。

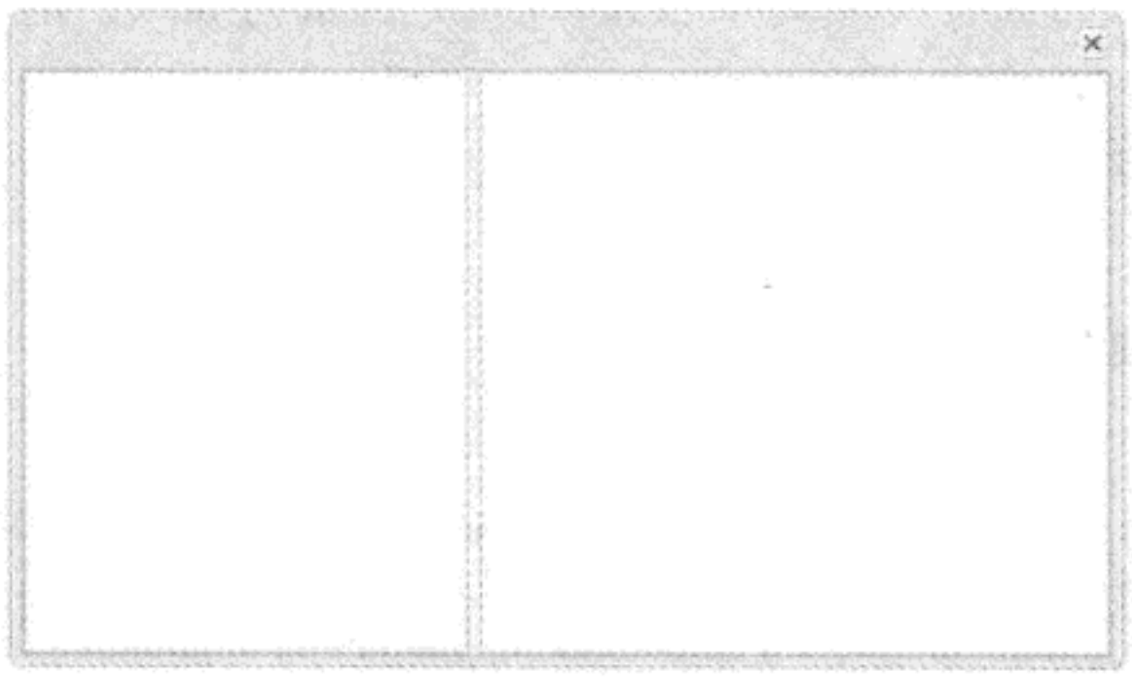


图 13-3　在窗口中输入 BorderLayout 进行布局的示例效果

代码相当的简单，与在面板中使用 BorderLayout 一样。

13.2　信息提示窗口：Ext.window.MessageBox

13.2.1　概述

信息提示窗口是 Window 对象的一个实例，在其内部固化了一些信息提示用的组件，然后通过组件的组合，就可实现不同的信息提示功能。而这些不同的信息提示功能是通过调用不同方法实现的。

13.2.2　信息提示窗口的构成

MessageBox 对象的 initComponent 方法代码如下：

```
initComponent: function() {
    var me = this,
        i, button;

me.title = ' ';

me.topContainer = new Ext.container.Container({
    layout: 'hbox',
    style: {
        padding: '10px',
        overflow: 'hidden'
    },
```

```
items: [
    me.iconComponent = new Ext.Component({
        cls: me.baseCls + '-icon',
        width: 50,
        height: me.iconHeight
    }),
    me.promptContainer = new Ext.container.Container({
        flex: 1,
        layout: {
            type: 'anchor'
        },
        items: [
            me.msg = new Ext.Component({
                autoEl: { tag: 'span' },
                cls: me.baseCls + '-text'
            }),
            me.textField = new Ext.form.field.Text({
                anchor: '100%',
                enableKeyEvents: true,
                listeners: {
                    keydown: me.onPromptKey,
                    scope: me
                }
            }),
            me.textArea = new Ext.form.field.TextArea({
                anchor: '100%',
                height: 75
            })
        ]
    })
]
});
me.progressBar = new Ext.ProgressBar({
    margins: '0 10 0 10'
});

me.items = [me.topContainer, me.progressBar];

me.msgButtons = [];
for (i = 0; i < 4; i++) {
    button = me.makeButton(i);
    me.msgButtons[button.itemId] = button;
    me.msgButtons.push(button);
}
me.bottomTb = new Ext.toolbar.Toolbar({
    ui: 'footer',
    dock: 'bottom',
    layout: {
        pack: 'center'
    },
    items: [
        me.msgButtons[0],
        me.msgButtons[1],
        me.msgButtons[2],
        me.msgButtons[3]
```

```
            ]
        });
        me.dockedItems = [me.bottomTb];

        me.callParent();
    },
```

从代码可以看到，它会创建一个如图 13-4 所示的窗口。在 bottomTb 中，会有“确定（OK）”、“是（Yes）”、“否（No）”和“取消（cancel）”4 个按钮。通过这些组件的组合就可以创建出不同的信息提示框。

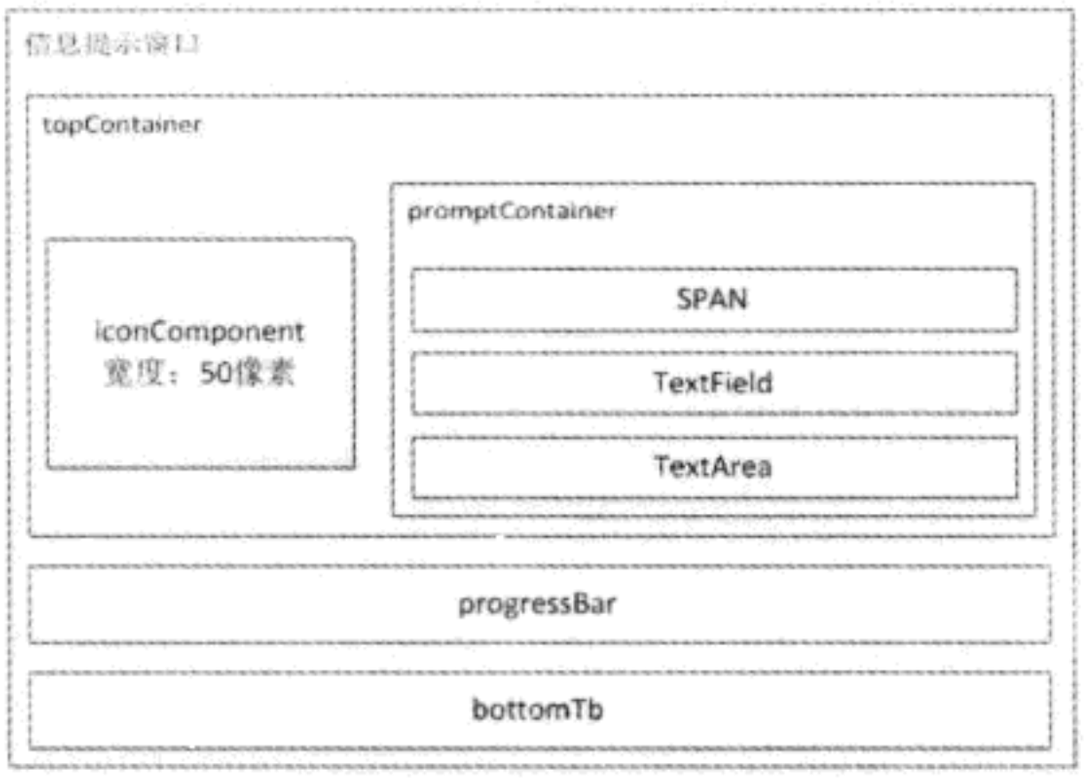


图 13-4　信息提示窗口的结构

13.2.3　使用信息提示窗口

要使用信息提示窗口，可使用 show 方法，自定义窗口内的组件的显示，也可以直接调用已预定义了组件的方法。本节主要介绍 show 方法，其余的方法会在后面介绍。使用 show 方法显示信息提示窗口的格式如下：

```
Ext.MessageBox.show(config);
```

或

```
Ext.Msg.show(config)
```

格式中的 config 为配置对象，可用的配置项如下：

- ❑ animateTarget：设置在窗口显示时执行动画的启始对象，值可以是模板元素也可以是其 id。默认值为 null。
- ❑ buttons：设置要显示的按钮。其值为要显示按钮值的累加结果。“确定”按钮的值是 1；“是”按钮的值是 2；“否”按钮的值是 4；“取消”按钮的值是 8，因而要显示“确定”和“取消”按钮，设置值为 9 就行了。如果不要显示按钮，则设置值为 false。
- ❑ closable：布尔值，设置 false 不显示关闭图标。默认值为 true，显示关闭图标。
- ❑ cls：应用到信息提示窗口容器的样式。

- defaultTextHeight：设置 Textarea 字段的高度，默认值是 75 像素。
- fn：用户单击窗口上的按钮后，执行的回调函数。接收的参数包括 buttonId、text 和 opt 这三个。其中，buttonId 表示用户单击的是哪个按钮，值包括 ok、yes、no 或 cancel；text 表示在 prompt 或 multiline 配置项为 true 时，在 Text 字段或 Textarea 中输入的值；opt 则为调用 show 方法时传递的配置对象。
- scope：作用域。
- icon：设置 iconComponent 的样式。可以使用已有的样式，也可以自定义样式。
- iconCls：设置窗口标题栏的图标样式。
- maxWidth：设置窗口的最大宽度，默认值为 600。
- minWidth：设置窗口的最小宽度，默认值为 100。
- modal：设置窗口是否为模态窗口，默认值为 true，表示显示为模态窗口。
- msg：在 SPAN 标记中显示的文字信息。
- multiline：如果为 true，显示 Textarea 字段，默认值为 false，不显示 Textarea 字段。
- progress：如果为 true，显示进度条，默认值为 false，不显示进度条。
- progressText：显示在进度条内的文字信息。默认值为空值。
- prompt：如果为 true，显示 Text 字段。默认值为 false，不显示 Text 字段。
- proxyDrag：如果为 true，当窗口被拖动时，显示一个轻型代理。默认值为 false。
- title：窗口的标题。
- value：设置 Text 或 Textarea 字段的值。
- wait：如果为 true，显示进度条。默认值为 false。
- waitConfig：该值为进度条的配置对象，可用于定义进度条。
- width：设置窗口的宽度。

打开模板页，在命令行输入以下代码：

```
Ext.Msg.show({
    title:"提示信息",msg:"自定义的信息提示窗口",
    buttons:1
})
```

运行后会看到如图 13-5 所示的提示信息窗口。可以看到，其效果与 alert 方法显示的效果差不多。

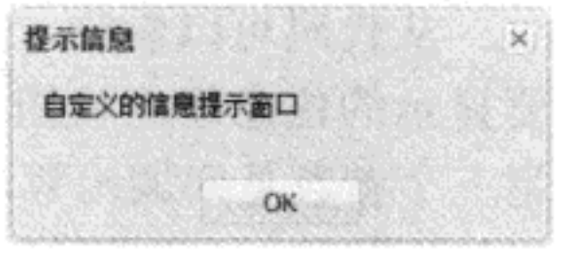


图 13-5　使用 show 显示的提示信息窗口

一般情况下，使用预定义的信息提示方法就足够了，不需要使用到 show 方法，当然，特殊情况特殊处理。

13.2.4　信息提示窗口按钮的本地化

在本地化文件其实有本地化的代码，不过不起作用，原因是它只是修改了 buttonText 对象里的文本，但是，这时候按钮已经在 initComponent 方法中实例化了，也就是说，按钮的文本已经在固化在按钮里了，修改 buttonText 并不会改变按钮的文本。因而，要实现本地化，只能直接修改按钮的文本了。比较幸运的是，所有按钮都是在 msgButtons 指向的数组里，遍历

该数组，然后使用 setText 方法修改就行了，代码如下：

```
if(Ext.MessageBox){
   var buttonText={
      ok     : "确定",
      cancel : "取消",
      yes    : "是",
      no     : "否"
   },
    btns=Ext.MessageBox.msgButtons;
    for(var i=btns.length-1;i>=0;i--){
          var btn=btns[i];
          btn.setText(buttonText[btn.itemId]);
   }
}
```

所有按钮都会有 itemId 属性，来标记其是什么按钮，根据该 id 对文本进行修改就行了。替换掉原来的本地化代码，就可以看到显示中文的按钮了。

13.2.5 使用 alert 方法

先看看 alert 方法使用了那些配置项，其源代码如下：

```
alert: function(cfg, msg, fn, scope) {
    if (Ext.isString(cfg)) {
        cfg = {
            title : cfg,
            msg : msg,
            buttons: this.OK,
            fn: fn,
            scope : scope,
            minWidth: this.minWidth
        };
    }
    return this.show(cfg);
},
```

从代码可以看到，第一个参数 cfg 可以是配置对象，也可以是标题。第二个参数，就是要显示的信息了。按钮固定了只显示“确定”按钮，因而只能用来做一般的信息提示窗口。第三个参数是回调函数。第 4 个参数为作用域。最小宽度可在 OnReady 函数中先预设好，这样就比较统一，不用每次都修改了。

测试一下，在模板页的命令行中输入以下代码：

```
Ext.Msg.alert("提示信息","自定义的信息提示窗口");
```

运行后，将看到与图 13-5 一样的窗口。

13.2.6 使用 confirm 方法

方法 confirm 的源代码如下：

```
confirm: function(cfg, msg, fn, scope) {
```

```
if (Ext.isString(cfg)) {
    cfg = {
        title: cfg,
        icon: this.QUESTION,
        msg: msg,
        buttons: this.YESNO,
        callback: fn,
        scope: scope
    };
}
return this.show(cfg);
},
```

从代码可以看到，它与 alert 的主要区别是，会显示带问号的图标，按钮会显示“是”和“否”两个按钮。

在命令行中输入以下代码：

```
Ext.Msg.confirm("提示信息","你确认删除这些记录？",
    function(btn){
        console.log(btn);
    }
);
```

运行后，会看到如图 13-6 所示的结果。单击“是”，在控制会显示 yes，单击“否”，可看到 no，通过参数 btn 就可判断用户单击了那个按钮。

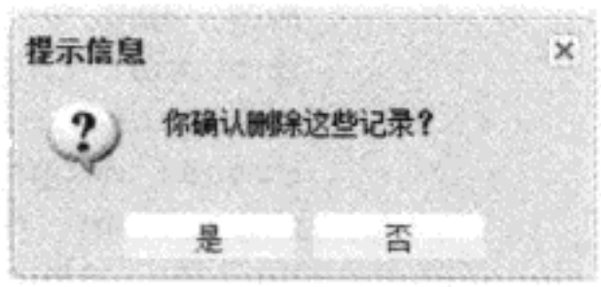


图 13-6 confirm 方法显示的提示窗口

13.2.7 使用 progress 方法

方法 progress 的源代码如下：

```
progress : function(cfg, msg, progressText){
    if (Ext.isString(cfg)) {
        cfg = {
            title: cfg,
            msg: msg,
           progress: true,
            progressText: progressText
        };
    }
    return this.show(cfg);
}
```

从代码可以看到，progress 方法值配置了标题、信息、显示进度条（progress 为 true）和进度条里显示的文本这 4 个配置项。

在命令行中输入以下代码：

```
Ext.Msg.progress("提示信息","正在上传 ...","1/10");
```

运行后可看到图 13-7 所示的效果。要让进度条前进，需要使用 updateProgress 方法。这里要注意，updateProgress 方法第一个参数是控制进度条显示的百分比的，值为 0 到 1 中的数字，例如输入：

```
Ext.Msg.updateProgress(0.1,"2/10");
```

滚动条将会前进 10%，而滚动条内文字会显示为“2/10”。如果最后完成了，可输入：

```
Ext.Msg.updateProgress(1,"10/10","上传结束");
```

当进度条就到达 100% 时，“正在上传…”会变成“上传结束”。最后还要调用 close 方法关闭窗口。

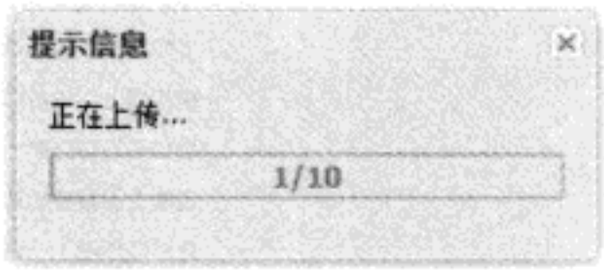


图 13-7　progress 方法显示的提示窗口

13.2.8　使用 prompt 方法

方法 prompt 的源代码如下：

```
prompt : function(cfg, msg, fn, scope, multiline, value){
    if (Ext.isString(cfg)) {
        cfg = {
            prompt: true,
            title: cfg,
            minWidth: this.minPromptWidth,
            msg: msg,
            buttons: this.OKCANCEL,
            callback: fn,
            scope: scope,
            multiline: multiline,
            value: value
        };
    }
    return this.show(cfg);
},
```

代码默认配置了 prompt 为 true，也就是说默认情况下，会显示 Text 字段，允许用户输入。按钮会显示“确定”和“取消”按钮。第 5 个参数 multiline 如果为 true，则会显示 Textarea 字段。第 6 个参数用来设置默认值。该方法多用于一些需要做简单输入的地方，例如，作为修改树节点的文本。

在命令行中输入以下命令：

```
Ext.Msg.prompt("输入","请输入：",
    function(btn,v){
        if(btn=="ok"){
            console.log(v)
        }
    },
    null,null,"值"
);
```

运行后，将看到如图 13-8 所示的效果。单击“确定”后可看到控制台中会输出“值”。修改第 6 个参数为 true，将看到如图 13-9 所示的效果。

图 13-8　显示 Text 字段的 prompt 提示窗口

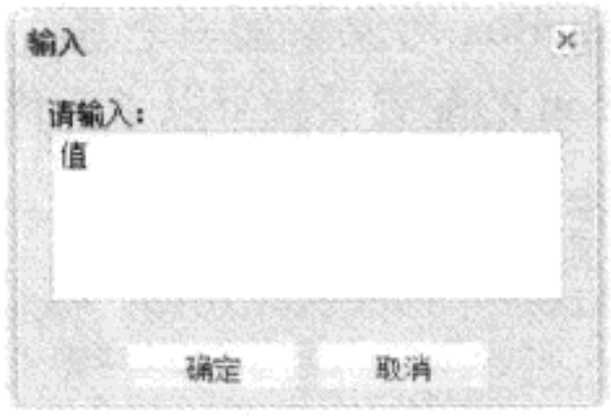


图 13-9　显示 Textarea 字段的 prompt 提示窗口

13.2.9　使用 wait 方法

方法 wait 的代码如下：

```
wait : function(cfg, title, config){
    if (Ext.isString(cfg)) {
        cfg = {
            title : title,
            msg : cfg,
            closable: false,
            wait: true,
            modal: true,
            minWidth: this.minProgressWidth,
            waitConfig: config
        };
    }
    return this.show(cfg);
},
```

代码只有 3 个参数，不过与其他方法不同，要注意，第一个参数是信息，第二个参数才是标题，第三个参数是进度条 wait 方法配置对象。注意，配置项里 closable 为 false，也就是不允许用户关闭窗口，只能等待任务执行完毕后由代码控制关闭，这也就是为什么“wait”是方法名了。

在命令行中输入以下命令：

```
Ext.Msg.wait("正在加载...","请等待...");
```

运行后，将会看到如图 13-10 所示的效果。进度条默认是 1 秒前进 10% 的，如果没有调用 close 方法，窗口会一直显示，进度条满 100% 后会重新从 10% 开始前进。因而，使用 wait 方法，一定要注意，任务完成后要调用 cloase 方法关闭窗口。

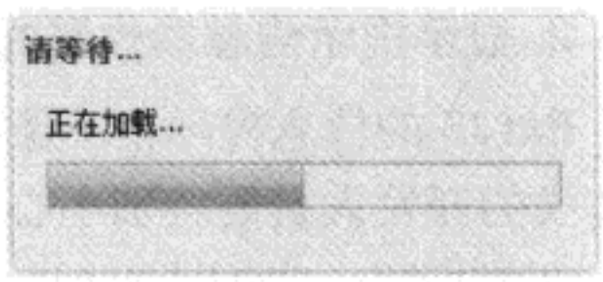


图 13-10　wait 方法显示的提示窗口

13.2.10　使用信息提示窗口要注意的问题

1. 使用 alert 不能显示信息图标

从 13.2.5 节的 alert 的源代码知道，alert 没有定义 icon 属性，因而不能显示提示图标，这样就只能通过 show 方法自定义显示了，但这样使用起来太麻烦。好的方法是在本地化文件

中为 MessageBox 对象添加一个方法，例如，myAlert 方法，在参数 msg 参数后插入 1 个参数 icon 作为 icon 的配置项就可以了，代码如下：

```
Ext.apply(Ext.MessageBox,{
    // 省略重写的 progress 方法
    myAlert: function(cfg, msg, icon,fn, scope) {
        if (Ext.isString(cfg)) {
            cfg = {
            icon:icon,
                    // 省略原有 alert 的配置参数
            };
            }
            return this.show(cfg);
    },
})
```

在使用 alert 时，fn 和 scope 这两个参数不常用，因此，为了避免在编码时加入多余的逗号，将 icon 作为第三个参数。

刷新一下模板页，在命令行中输入以下代码：

```
Ext.Msg.myAlert(" 警告 "," 发生了错误。",Ext.Msg.WARNING);
```

运行后，将看到如图 13-11 所示的效果。

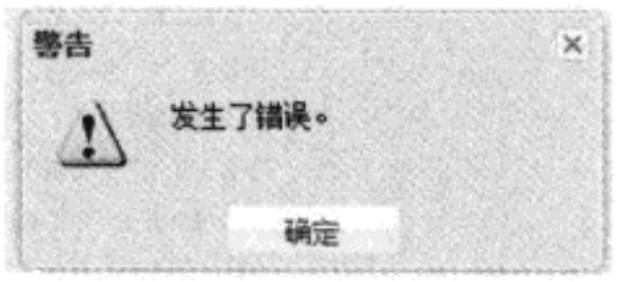


图 13-11　myAlert 方法的显示效果

表 13-1　MessageBox 对象提供的 4 个 icon 值及其图标

值	图标
Ext.Msg.INFO	
Ext.Msg.WARNING	
Ext.Msg.QUESTION	
Ext.Msg.ERROR	

2. 显示提示信息后代码的执行问题

Ext JS 的信息提示框不像 JavaScript 的 alert 方法一样，会停止代码的执行，因而，习惯了该方式的开发人员要特别小心。做个简单测试就知道了。

在模板页，先测试 JavaScript 的 alert 方法，在命令行中输入以下代码：

```
alert("Test");
console.log("Test");
```

运行后会看到，只要不关闭 alert 对话框，后面的代码就不会执行，代码在这里停止了。单击“确定”后，才会在控制台看到“Test”。

现在将 alert 换成 MessageBox 的 alert 方法，运行后会看到，信息提示窗口还没关闭，在控制台就已经看到“Test”了。这说明 MessageBox 的 alert 方法不会停止代码的执行，而是在

显示对话框的同时，继续执行后续代码。

因而，以前在 alert 方法后放置后续代码的方式就要将后续代码放到回调函数里执行，不然会出现意想不到的错误。例如，将 MessageBox 的 alert 方法的代码修改成以下代码：

```
Ext.MessageBox.alert("Test","Test",function(){
    console.log("Test");
});
```

这样，效果才会像 JavaScript 的 alert 方法一样。不单是 alert 方法，使用 MessageBox 的其余方法也会有这样的问题要注意。

在回调函数里执行后续代码的问题是作用域会改变，因而在调用时设置作用域也是相当重要的。具体方式可参考 12.7 节示例的“删除”按钮。

13.3　综合实例：实现登录窗口

（1）功能描述

实现一个登录窗口。

（2）实现代码

使用模板页创建一个名称为 13-1.html 的页面文件，先创建一个窗口，代码如下：

```
Ext.create("Ext.window.Window",{
    title:" 登录 ",modal:true,width:300,height:250,
    closable:false,resizable:false,closeAction:"hide",
    items:[
    ]
}).show();
```

因为登录窗口不允许关闭，所以要设置 closable 为 false。

接着完成表单，代码如下：

```
{xtype:"form",border:false,bodyPadding:5,id:"loginForm",
    currentTabIndex:1,
    bodyStyle:"background:#DFE9F6",
    defaultType:"textfield",
    fieldDefaults: {
            labelWidth:80,labelSeparator:": ",anchor:"0"
    },
    items:[
            {fieldLabel:" 用户名 ",name:"username",allowBlank:false,
                    tabIndex:1
            },
            {fieldLabel:" 密码 ",name:"password",allowBlank:false,
            inputType:"password",tabIndex:2
    },
    {fieldLabel:" 验证码 ",name:"vcode",minLength:6,minLength:6,
            allowBlank:false,tabIndex:3
    },
    {xtype:"container",height:80,anchor:"-5",layout:"fit",
    items:[
            {xtype:"image",id:"vImage",
```

```
                        src:"vcode.ashx?_dc="+(new Date()).getTime()
                    }
                ]
                }
        ],
        dockedItems: [
                {xtype: 'toolbar',dock:'bottom',ui:'footer',layout:{pack:"center"},
            items: [
                    {text:"登录",disabled:true,formBind:true,
                        handler:function(){
                        },
                    {text:"重置",handler:function(){
                    }},
                    {text:"刷新验证码",handler:function(){
                    }}
                ]}
        ]
}
```

要输入密码，只要定义 inputType 为 password 就行了。如果直接把图片放在字段下，因为没容器，进行布局计算时会忽略图片的高度，所以在其上面加另一个 Container，然后使用 Fitlayout，让图片完全使用容器的空间就可以了。刷新验证码使用按钮比添加文字链接方便得多，不然又要自己写链接的脚本。发现很多组件都没单击事件，很是奇怪。

代码中，为 3 个 Text 字段添加了 tabIndex，目的是方便使用回车键，在它们之间切换。表单的附加属性 currentTabIndex 可记录焦点在哪里。定义回车键功能代码如下：

```
var form=Ext.getCmp("loginForm");
Ext.create("Ext.util.KeyNav",form.getEl(),{
    enter:function(){
            var me=this;
            me.currentTabIndex++;
            if(me.currentTabIndex>3)
                    me.currentTabIndex=1;
            me.query("[tabIndex="+me.currentTabIndex+"]")[0].focus();
    },
    scope:form
});
```

使用 KeyNav 对象，绑定表单的元素，这样当字段中按下了回车键的时候，就可触发 enter 指向的函数了。函数中，先找到下一个回车的索引，然后通过 query 查询出 tabIndex 属性等于索引的字段，并调用 focus 方法设置焦点。这里还有点问题，就是当用户随机点进某个字段的时候，currentIndex 的值就会乱，所以，还要在字段获取焦点的时候重新设置 currentIndex 的值，代码如下：

```
var fieldFocus=function(el){
    var form=this.up("form");
    form.currentTabIndex=this.tabIndex;
};
var f=form.getForm();
f.findField("username").on("focus",fieldFocus);
```

```
f.findField("password").on("focus",fieldFocus);
f.findField("vcode").on("focus",fieldFocus);
```

按钮因为按下回车键的时候等于单击的效果，所以不能设置回车键进行切换，除非已经屏蔽掉回车键的单击作用了。

为了在窗口显示时，让光标停留在 username 字段，需要为窗口定义 show 事件，代码如下：

```
listeners:{
   show:function(){
          var f=this.down("form").getForm();
          f.findField("username").focus();
   }
}
```

余下就是三个按钮的操作了，先完成“重置”按钮，这个简单，代码如下：

```
var f=this.up("form").getForm();
f.reset();
```

再完成“刷新验证码”按钮的操作，代码如下：

```
var img=Ext.getCmp("vImage");
img.setSrc("Vcode?_dc="+(new Date()).getTime());
```

最后完成“登录”按钮，这里要考虑的问题是登录成功后使用何种方式切换显示，可以是跳转到另一个页面，可以是服务器端跳转，可以在客户端跳转，也可以是关闭登录窗口，然后从服务端下载脚本重新定义渲染页面。具体方式在本示例就不探讨了。最后完成的“登录”按钮代码如下：

```
var f=this.up("form").getForm();
if(f.isValid()){
   f.submit({
          url:"check.ashx",
          success:function(form,action){
                 // 页面调整或重新渲染页面
                 Ext.Msg.myAlert("提示信息",action.result.msg,Ext.Msg.INFO);
          },
        failure: function(form, action){
           if (action.failureType === "connect") {
               Ext.Msg.myAlert('错误',
                   '状态:'+action.response.status+': '+
                   action.response.statusText,Ext.Msg.ERROR);
               return;
           }
          if(action.result){
                 if(action.result.msg)
                 Ext.Msg.myAlert('错误', action.result.msg,Ext.Msg.ERROR);
           }
        },
        scope:form
   });
}
```

服务器端主要考虑的问题是，错误信息如何返回，使用 errors 对象是一个好的方式，这样错误会显示在字段上，具体代码如下：

C#

```
public void ProcessRequest (HttpContext context) {
    string vcode = context.Request.Params["vcode"] ?? "";
    string username = context.Request.Params["username"] ?? "";
    string password = context.Request.Params["password"] ?? "";
    JObject jo = new JObject();
    jo.Add("success", true);
    if (context.Session["vcode"] == null)
    {
        jo.Property("success").Value = false;
        jo.Add("errors", new JObject(
            new JProperty("vcode", "错误的验证码。")
        ));
    }
    else
    {
        if (string.Compare(vcode, context.Session["vcode"].ToString(), false) == 0 )
        {
            if (username == "admin" && password == "123456")
            {
                jo.Add("msg", "登录成功");
            }
            else
            {
                jo.Property("success").Value = false;
                jo.Add("errors", new JObject(
                    new JProperty("username", "用户名或密码错误。"),
                    new JProperty("password", "用户名或密码错误。")
                ));
            }
        }
        else
        {
            jo.Property("success").Value = false;
            jo.Add("errors", new JObject(
                new JProperty("vcode", "错误的验证码。")
            ));
        }
    }
    context.Response.Write(jo.ToString());
}
```

Java

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
   String vcode="";
   if(request.getParameter("vcode")!=null){
        vcode=request.getParameter("vcode");
   }
```

```
String username="";
if(request.getParameter("username")!=null){
        username=request.getParameter("username");
}
String password="";
if(request.getParameter("password")!=null){
        password=request.getParameter("password");
}
HttpSession session=request.getSession(true);
JsonObject jo =new JsonObject();
JsonObject errors=new JsonObject();
jo.add("success", new JsonPrimitive(true));
if(session.getAttribute("vcode")==null){
        jo.add("success", new JsonPrimitive(false));
        errors.addProperty("vcode", "错误 的验证码。");
        jo.add("errors", errors);
}else{
        if(vcode.equals(session.getAttribute("vcode").toString())){
                if(username.equals("admin") && password.equals("123456")){
                        jo.add("success", new JsonPrimitive(true));
                        jo.addProperty("msg","登录成功");
                }else{
                        jo.add("success", new JsonPrimitive(false));
                errors.addProperty("username", "用户名或密码错误。");
                errors.addProperty("password", "用户名或密码错误。");
                        jo.add("errors", errors);
                }
        }else{
                jo.add("success", new JsonPrimitive(false));
                errors.addProperty("vcode", "错误 的验证码。");
                jo.add("errors", errors);
        }
    }
    response.setContentType("text/javascript;
        charset=utf-8");
    response.getWriter().write(jo.toString());
}
```

因为用户名和密码同时验证了，不知道是用户名还是密码错，所以在添加错误的时候，把两个字段都加上了，而错误信息显示的是一样的。

（3）页面效果

在浏览器中打开页面，将看到图 13-12 所示的效果。

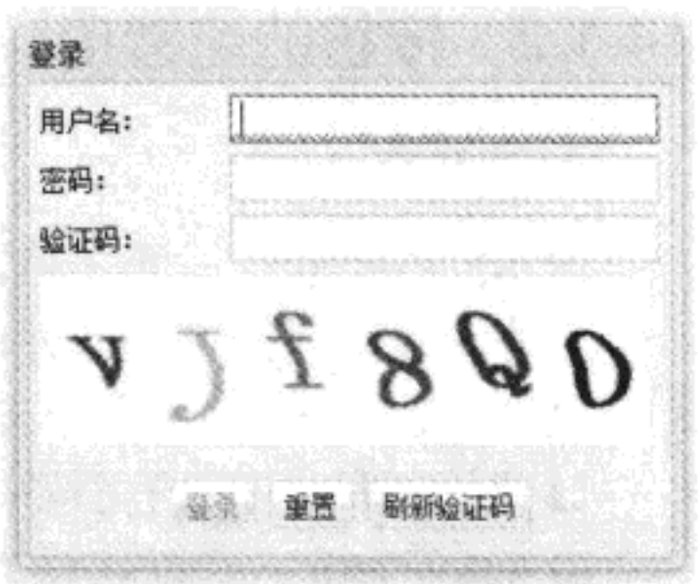


图 13-12　示例的页面效果

13.4　本章小结

窗口是应用中频繁使用的组件，因而必须熟悉其用法，尤其是信息提示窗口。只要清楚窗口是一个特殊的面板，在使用上就不会有太大的困难，关键是一些细节的地方要注意，如可复用的窗口要设置 closeAction 配置项等。多点看 API，熟悉其配置项是好的办法。

第 14 章　按钮、菜单与工具条

按钮、菜单和工具条也是使用比较频繁的组件，可以说是简单而平凡的组件。本章将讲述这些简单而平凡的组件是如何构成的，以及如何去使用它们。

14.1　按钮

14.1.1　按钮的构成：Ext.button.Button

Button 对象派生于 Component 对象，因而基本就是由自己的模板构成的，图 14-1 是按钮的元素构成。从图中可以看到，一个按钮总共有 5 个元素，最外层的 div 负责监听单击事件，em 元素的作用是根据定义使用不同的样式显示下拉的小箭头，button 元素的作用是作为图标和文本的容器，两个 span 元素一个用于显示文本，一个用于显示图标，通过不同的样式，就可实现按钮的多种样式。

图 14-1　按钮的构成

当为按钮定义了 url 配置项的时候，button 元素会换成 a 元素，成为一个链接，而且是 target 为“_blank”的链接，这适合用在一些需要打开链接的地方。

14.1.2　按钮的配置项、属性、方法和事件

1. 配置项

- allowDepress ：布尔值，如果设置为 false，会禁止按钮的按下动作。该配置项只有在 enableToggle 为 true 时才有效。默认值是 undefined。
- arrowAlign：设置小箭头的位置，值可以为 right 和 bottom，默认值为 right。
- arrowCls：应用于小箭头的样式名称。
- autoWidth ：在默认状态下，如果没有设置按钮的宽度，它会尝试拉长其宽度以适应容器的宽度。如果按钮的宽度是由可调整大小的布局管理的，该值设置为 false 可阻止按钮自动调整大小。默认值为 undefined。
- baseCls：应用于按钮的样式名称，默认值为“x-btn”。
- baseParams：参数对象，当定义了 url 配置项，该参数将作为链接的提交参数。
- clickEvent ：设置可以触发按钮句柄函数的 DOM 事件，默认值是 click，值还可以是 dblclick 或 contextmenu。

- cls：应用于按钮 div 的样式名称。
- disabled：布尔值，设置为 true 会禁用按钮。默认值为 false。
- enableToggle：布尔值，设置为 true，按钮将会有两个切换状态。默认值为 false。
- focusCls：按钮获得焦点后的样式名称。默认值为“x-btn-focus”。
- handleMouseEvents：布尔值，如果设置为 false，会禁用 mouserover、mouseout 或 mousedown 等的视觉效果。默认值为 true。
- handler：单击鼠标后调用的句柄函数。
- hidden：布尔值，如果设置为 true，会隐藏按钮。默认值为 false。
- icon：图标使用的图片文件及其路径，该图片会作为按钮的背景图片。
- iconAlign：按钮图标的显示位置，值可以为 top、right、bottom 或 left。
- iconCls：图标的样式名称。
- menu：定义按钮的菜单。默认值为 undefined。
- menuActiveCls：当菜单处于活动状态时应用于按钮的样式。默认值为“x-btn-menu-active”。
- menuAlign：菜单的相当于按钮的对象位置。详细请阅读 Element 对象的 alignTo 配置项。默认值为“tl-bl”。
- minWidth：按钮的最小宽度。
- overvCls：鼠标在按钮上方时的样式名称。默认值是“x-btn-over”。
- overflowText：按钮显示在工具栏溢出菜单的文本。
- params：与 baseParams 作用相同，如果两个同时定义时，该配置项会覆盖 baseParams 的设置。
- pressd：布尔值，如果值为 true，按钮处于按下状态。只有在 enableToggle 为 true 时才有效。默认值为 false。
- pressedCls：按钮处于按下状态时的样式名称。默认值为“x-btn-pressed”。
- preventDefault：布尔值，当定义了 clickEvent 配置项的时候，如果该值为 true，会阻止默认操作。默认值为 true。
- repeat：布尔值或 ClickRepeater 对象的配置项对象。如果为 true，当按住鼠标按钮时，会重复触发单击事件。默认值为 false。
- scale：按钮的大小。值可以是 small（16px）、medium（24px）或 large（32px）。
- scope：作用域。
- tabIndex：设置按钮 DOM 元素的 tabIndex 属性。
- text：按钮显示的文本。
- toggleGroup：将同名（toggleGroup 值相同）的按钮组成一个切换组，每组按钮中每次只能按下一个。
- toggleHandler：当 enableToggle 设置为 true 时，单击按钮调用的句柄。
- tooltip：按钮的提示信息。
- tooltipType：按钮提示信息的类型。默认值为 qtip，值还可以是 title，使用 HTML 的

title 属性显示提示信息。

- type：按钮的类型。默认值为 button。也可以设置为 submit 或 reset。

2. 属性

- disabled：只读属性，如果按钮被禁用，该值为 true。
- hidden：只读属性，如果按钮是隐藏的，该值为 true。
- menu：指向按钮的菜单。
- pressed：只读属性，如果按钮处于按下状态，该值为 true。
- template：创建按钮的模板。

3. 方法

- getPressed：返回指定按钮组中被按下的按钮，如果没有，返回 false。
- getText：返回按钮的显示文本。
- hasVisibleMenu：如果按钮有菜单且是隐藏的，返回 true。
- hideMenu：隐藏按钮的菜单。
- setHandler：设置按钮的单击句柄函数。
- setIcon：设置按钮的图标。
- setIconCls：设置按钮的图标样式。
- setParams：设置按钮的提交参数。
- setScale：改变按钮的大小。
- setText：设置按钮的显示文本。
- setTooltip：设置按钮的提示信息。
- showMenu：显示按钮的菜单。
- toggle：切换按钮的状态。

4. 事件

- click：当单击按钮时会触发该事件。
- menuhide：当菜单隐藏后会触发该事件。
- menushow：当菜单显示后，会触发该事件。
- menutriggerout：如果按钮有菜单，当鼠标离开触发元素时，会触发该事件。
- menutriggerover：如果按钮有菜单，当鼠标进入触发元素时，会触发该事件。
- mouseout：当鼠标离开按钮时，会触发该事件。
- mouserover：当鼠标在按钮上方时，会触发该事件。
- toggle：当按钮的 pressed 状态改变时，会触发该事件。

14.1.3 使用按钮

使用按钮，最简单的就是只定义单击句柄就行了，例如，打开模板页，在命令行中输入以下代码：

```
Ext.create(Ext.Button,{
```

```
    renderTo:Ext.getBody(),
    handler:function(){
        Ext.Msg.alert("信息","我是按钮");
    }
})
```

运行后，会看到页面左上角显示一个没有文字和图标的按钮，单击它会显示一个信息提示窗口。

这样的按钮实际用途不大，因而需要使用 text、icon 或 iconCls 等配置项为其加上文字或图标。还可以使用 tooltip 为按钮加上提示信息。

如果要设置按钮的大小，可以使用 width 和 scale 这两个配置项。

而设置配置项 enableToggle 则可将按钮设置为拥有两种状态的按钮，而其单击句柄可以为 toggleHandler，也可以为 handler，两者只能选一个，如果同时使用，只会触发 handler。而 toggleHandler 的优势是会把按钮当前的状态返回到函数。

以上配置项大家都可以通过命令行测试一下，这样对熟悉按钮的使用相当有帮助。

14.1.4　带分割线的按钮：Ext.button.Split

SplitButton 对象派生于按钮，它与按钮最大的不同是，em 元素上的小箭头会有自己的单击操作，而按钮则保持自己的操作。界面显示上，按钮和小箭头之间会多了一条分割线。

因而，它在按钮的基础上增加了以下两个配置项：

- arrowHandler：单击小箭头调用的句柄函数。
- arrowTooltip：小箭头的提示信息。

还增加了 setarrowHandier 方法用来设置小箭头的句柄函数。

当然，为了配合小箭头的操作，还添加了 arrowclick 事件，当单击小箭头时触发。

SplitButton 对象的使用与按钮的使用方法除了突出小箭头外，没其他区别。

14.1.5　多状态按钮：Ext.button.Cycle

CycleButton 对象派生于 SplitButton 对象，它的功能类似 13.3 节示例的登录对话框中使用回车键在 Text 字段中切换的功能，只是回车键变成了按钮的单击操作，而 Text 字段则换成了菜单项的文本，每次单击操作就会切换一次菜单项。通过小箭头展开菜单，直接选择菜单项进行操作也行。

打开模板页，在命令行中输入以下代码：

```
Ext.create(Ext.CycleButton,{
    showText: true,
    renderTo: Ext.getBody(),
    menu: {
        items: [{
            text:'左对齐',
            checked:true
        },
        {
```

```
                text:'居中对齐',
            },
            {
                text:'右对齐',
            }]
        }
})
```

运行后，单击三次按钮，再打开菜单，将看到如图 14-2 所示的效果。

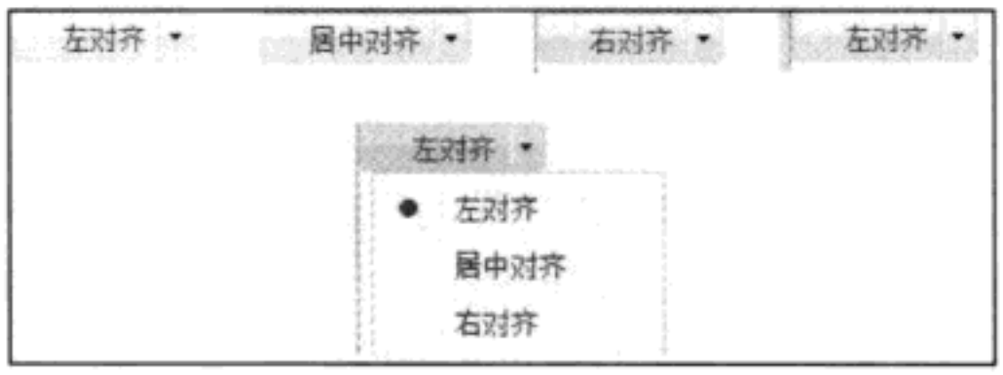


图 14-2　CycleButton 对象的示例效果

为了实现 CycleButton 对象的功能，在对象内添加了以下配置项、属性、方法和事件：

1. 配置项

- changeHandler：当菜单项发生改变的时候，会调用该配置项定义的回调函数。
- forecIcon：设置按钮的图标样式。
- prependText：当没有被选择选项时，在按钮上显示的文本。
- showText：布尔值，如果为 true，会将选项的文本显示到按钮。默认值为 false，只显示按钮定义的文本。

2. 属性

- menu：按钮的菜单。

3. 方法

- getActiveItem：获取当前选择的选项。
- setActiveItem：将指定的菜单项作为当前选择项。
- toggleSelected：相当于单击按钮操作，切换选项。

4. 事件

- change：当选项发生改变时，会触发该事件。

14.1.6　按钮组：Ext.container.ButtonGroup

ButtonGroup 对象派生于 Panel 对象，因而具有面板的所有特性。其内部会使用表格布局作为布局，所以可使用 columns 配置项将按钮分组。

按钮组的使用可参考 9.6.6 节的示例。

14.2　菜单及菜单项

14.2.1　Ext JS 的菜单（Menu 对象）是什么

说到菜单，我们都会习惯性联想到应用程序顶部菜单栏，而 Ext JS 中的菜单概念并不是这样，准确的说，Menu 对象相当于我们习惯的子菜单。精确地来说，Menu 对象是一个子菜单面板，它派生于 Panel 对象，因而具有面板的特性。

Menu 对象是一个容器，是面板容器，这非常重要。虽然是面板，但是它有自己的独特，例如使用垂直布局作为布局，会在菜单管理器（Ext.menu.Manager）中进行注册等。它还有自己的配置项、属性、方法和事件。

1. 配置项

- allowOtherMenus：布尔值，如果为 true，可同时显示多个菜单。默认值为 false。
- defaultAlign：菜单默认的对齐方式。默认值为“tl-bl”。
- floating ：布尔值，默认值为 true，菜单会渲染成使用绝对定位的浮动组件。如果设置为 false，则会渲染成其他容器的子项。
- hidden ：布尔值，如果为 true，则菜单渲染后是隐藏的。当 floating 为 true 时，其默认值为 true。当 floating 为 false 时，其默认值是 false。
- ignoreParentClicks ：布尔值，如果为 true，会忽略有子菜单的菜单项的单击动作，这样单击菜单项就不会收回子菜单。默认值为 false。
- minWidth：菜单的最小宽度。默认值为 120。
- plain ：布尔值，如果为 true，会删除菜单的分隔线，而普通组件将不会缩进。默认值为 false。
- showSeparator：布尔值，默认值为 true，显示分隔线。

2. 属性

没有特别属性。

3. 方法

- canActivateItem：返回指定菜单项是否可以设置为活动项。
- deactivateActiveItem：将活动菜单项的状态设置为非活动状态。
- showBy：将菜单显示在指定的组件或元素内。

4. 事件

- click：单击菜单后，会触发该事件。
- mouseenter：当鼠标移入菜单时，会触发该事件。
- mouseleave：当鼠标移除菜单时，会触发该事件。
- mouseover：当鼠标在菜单上面时，会触发该事件。

14.2.2 菜单管理器：Ext.menu.Manager

菜单管理器是一个单件模式的类，其主要作用是通过 MixedCollection 对象实例来记录创建的菜单实例，然后通过将菜单的 beforehide、hide、beforeshow 和 show 事件绑定到管理器内的方法，统一管理菜单的显示或隐藏。

当菜单创建时，会通过 register 方法将菜单实例注册到菜单管理器，菜单的 id 就是该菜单实例的关键字，而菜单的 beforehide、hide、beforeshow 和 show 方法都会绑定到管理器对应的方法，从而实现管理功能。

具体的源代码比较简单，与其他的管理器作用类似，在此就不详细说明了，有兴趣可以自己研究一下。

14.2.3 菜单项：Ext.menu.Item

MenuItem 对象派生于 Component 对象。如图 14-3 所示，一个菜单项主要由容器、a 标记、用于显示图标的 image 标记以及用于显示菜单文字的 span 标记构成。从构成上可以看到，菜单项可以具有 a 标记的一些特性，这在菜单项的配置项中可以看到。

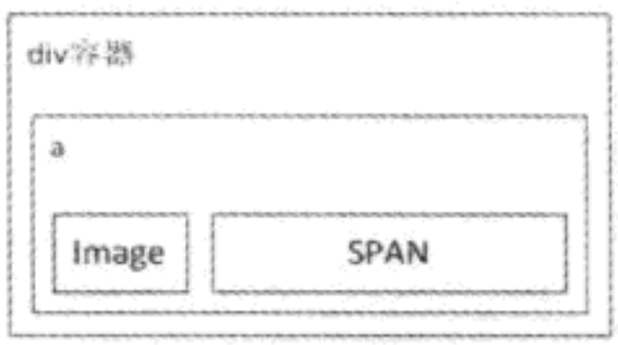


图 14-3 菜单项的构成

1. 配置项

- activeCls：当菜单项为活动状态时的样式名称。默认值为“x-menu-item-active”。
- canActivate：布尔值，默认值为 true，表示该菜单项可以被设置为活动状态。
- clickHideDelay：设置单击菜单项后，菜单隐藏前等待的时间，单位是微秒。该项只有在 hideOnClick 为 true 时才有效。默认值为 1 微秒。
- destroyMenu：布尔值，默认值为 true，当菜单项被销毁时，其子菜单也会被销毁。
- disabledCls：当菜单项被禁用时应用的样式，默认值为“x-menu-item-disabled”。
- handler：单击菜单项后执行的句柄函数。
- hideOnClick：布尔值，默认值为 true，当单击菜单项时，会隐藏菜单。
- href：设置菜单项 a 元素的 href 属性。
- hrefTarget：设置菜单项 a 元素的 target 属性。
- icon：设置菜单项使用的图标。
- iconCls：设置菜单项图标的样式。
- menuAlign：设置菜单项的子菜单的对齐方式，默认值为“tl-tr”。
- menuExpandDelay：设置鼠标移动到菜单项上时，显示子菜单前的等待时间，默认值为 200 微秒。
- menuHideDelay：设置鼠标移除菜单项时，隐藏子菜单前的等待时间，默认值为 200 微秒。
- plain：布尔值，如果为 true，菜单项将没有图标和活动时的可视效果。默认值为 false。
- text：菜单项显示的文本。

2. 属性

- activated：如果菜单项当前为活动状态，返回 true，否则返回 false。
- menu：菜单项的子菜单。

3. 方法

- setHandler：设置单击菜单项后的句柄函数。
- setIconCls：设置菜单项的图标样式。
- setText：设置菜单项的显示文本。

4. 事件

- active：当菜单项成为活动状态时，会触发该事件。
- click：单击菜单项后，会触发该事件。
- deactivate：当菜单项成为非活动状态时，会触发该事件。

14.2.4　可复选的菜单项：Ext.menu.CheckItem

可复选的菜单项是在菜单项的基础上，将其图标显示为一个可复选的图片，然后通过控制菜单项的单击操作模拟菜单项的选择与取消选择两种操作。

1. 配置项

- checkedCls：复选图标被选择时的样式名称，默认值是“x-menu-item-checked”。
- groupCls：当将可复选的菜单项组合成一个单选组时，图标应用的样式名称，默认值为“x-menu-group-icon”。
- hideOnClick：布尔值，默认值为 true，当单击菜单项时，会隐藏菜单。
- uncheckedCls：复选图标被选择时的样式名称，默认值是“x-menu-item-unchecked”。

2. 属性

没有特殊属性。

3. 方法

- disableCheckChange：禁用可复选菜单项的复选功能。
- enableCheckChange：启用可复选菜单项的复选功能。
- setChecked：设置可复选菜单项的选择状态。

4. 事件

- beforecheckchange：当可复选菜单项的选择状态发生改变前，会触发该事件，如果返回 false，可中止状态改变。
- checkchange：当可复选菜单项的选择状态改变后，会触发该事件。

14.2.5　菜单分隔条：Ext.menu.Separator

菜单分隔条派生于 MenuItem 对象，是用来分隔菜单项的。在使用时不需要定义配置项对

象，只要简单地使用“-”代替就可以了，例如在模板页命令中输入以下代码：

```
Ext.create(Ext.menu.Menu,{
    renderTo: Ext.getBody(),
        items: [{
            text:' 菜单项 1'
        },
        '-',
        {
            text:' 菜单项 2'
        }]
}).show()
```

运行后可看到如图 14-4 所示的效果，中间的分隔条就是根据“-”创建的。

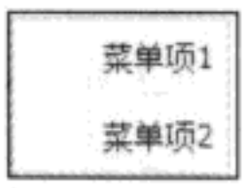


图 14-4 菜单分隔条的效果

14.2.6 颜色选择器菜单：Ext.menu.ColorPicker

ColorPicker 对象派生于 Menu 对象，它会在菜单面板内放置一个 ColorPalette 对象实例，然后将实例的 select 事件传播到 ColorPicker，这样通过 select 事件就可以获取选择的颜色了。

14.2.7 日期选择菜单：Ext.menu.DatePicker

MenuDatePicker 对象派生于 Menu 对象，它会在菜单面板内放着一个 DatePicker 对象实例，然后将实例的 select 事件传播到 MenuDatePicker，这样通过 select 事件就可以获取选择的日期了。

14.2.8 使用菜单

本节通过一个示例程序演示一下如何使用菜单及其所有的菜单项。

使用模板页创建一个名称为 14-1.html 的页面文件，然后定义一个面板，在面板的顶部工具栏内定义一个“菜单示例”按钮，代码如下：

```
Ext.create(Ext.panel.Panel,{
    renderTo:Ext.getBody(),width:100,height:100,
    tbar:[
            {text:" 菜单示例 "}
    ]
})
```

在按钮内定义配置项 menu，其值可以是一个菜单实例，也可以是菜单的菜单项组成的数组。如果是数组，会在菜单管理器中自动创建一个菜单实例。本示例将使用数组的方式进行定义，主要是代码书写方便，代码如下：

```
{text:" 纯文本菜单项 "},
{text:" 带图标的菜单项 ",icon:"../images/view.png"},
{xtype:"menucheckitem",text:" 可复选的菜单项 "},
"-",
{xtype:"menucheckitem",text:" 选项 1",group:"group",checked:true},
```

```
{xtype:"menucheckitem",text:"选项 2",group:"group"},
{xtype:"menucheckitem",text:"选项 3",group:"group"},
"-",
{text:"选择日期",menu:{xtype:"datemenu",
   listeners:{
          select:function(cm,date){
                 Ext.Msg.alert("日期选择","选择的日期是: "+date);
          }
   }
}},
{text:"选择颜色",menu:{xtype:"colormenu",
   listeners:{
          select:function(cm,color){
                 Ext.Msg.alert("颜色选择","选择的颜色是: "+color);
          }
   }
}}
]}
```

代码依次定义了“纯文本菜单项”、“带图标的菜单项”、“可复选的菜单项”、分隔条、同一组的三个可复选的菜单项、分隔条、带日期选择菜单的“选择日期”菜单项及带颜色选择菜单的“选择颜色”菜单项。

在浏览器中打开页面，单击按钮，并将鼠标移动到选择颜色上，将看到如图 14-5 所示的效果。

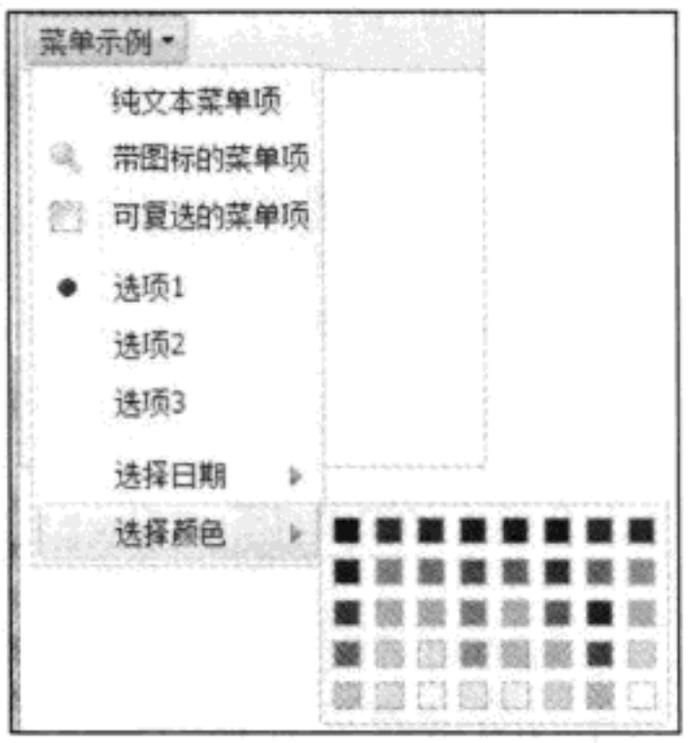


图 14-5　菜单使用示例的效果

14.3　工具栏及工具栏组件

14.3.1　工具栏：Ext.toolbar.Toolbar

Toolbar 对象派生于 Container 对象，是一个容器。如果工具栏是横向的，会使用水平布局对容器内的组件进行布局；如果是竖向的，则使用垂直布局对容器内的组件进行布局。

使用工具栏最重要的是要清楚知道，如果没有定义其子组件的类型，默认会创建按钮，因而，如果不是创建按钮，一定要定义 xtype 配置项，除非你修改工具栏的配置项 defaultType。

工具栏主要有以下两配置项：

- enableOverflow：布尔值，如果设置为 true，工具栏将会提供一个按钮显示一个下拉子菜单显示超出工具栏宽度的条目。默认值为 false。
- vertical：布尔值，如果设置为 true，工具栏的显示将会是垂直方式。默认值为 false。

工具栏提供了一个 overflowchange 事件，它会在溢出状态改变时触发。

14.3.2　非交互式工具栏条目的基类：Ext.toolbar.Item

ToolbarItem 对象派生于 Component 对象，是 ToolBarTextItem、Fill 和 Separator 这 3 个非

交互式对象的基类。它主要是在 Component 对象的基础上添加 enable、disable 和 focus 这三个空函数。

14.3.3 文本项：Ext.toolbar.TextItem

ToolBarTextItem 对象派生于 ToolbarItem 对象，用于在工具栏上显示纯文本。因为没有交互功能，所以只有 text 一个配置项，用于设置显示的文本。

它还有一个 setText 方法用于设置显示的文本。

14.3.4 填充项：Ext.toolbar.Fill

Fill 对象派生于 ToolbarItem 对象，用于填充工具栏的空位，以便在工具栏右边放置组件。

因为横向工具栏使用的是水平布局，所以只要将 Fill 对象的 flex 设置为 1，就可以填充工具栏的空位了。

使用时可以像菜单分隔条一样，使用“->”符号组合代替对象的定义。

14.3.5 工具栏分隔条：Ext.toolbar.Separator

Separator 对象派生于 ToolbarItem 对象，用于在工具栏上显示一个分隔条。其使用方法与菜单分隔条一样，可使用“|”代替对象的定义。

14.3.6 空白项：Ext.toolbar.Spacer

Spacer 对象派生于 Component 对象，主要作用是在工具栏上两个条目之间增加空位，如果不设置其宽度，默认宽度是两个像素。

可以像菜单分隔条一样，使用空格代替对象的定义，不过，要设置宽度的话，必须使用配置对象。

灵活使用空白项可实现一些特殊效果，例如希望工具栏的按钮居中显示，可分别在工具栏一头一尾放置一个空白项，设置其 flex 配置项的值为 1 就可以了。例如，打开模板页，然后在命令行输入以下命令：

```
Ext.create(Ext.toolbar.Toolbar,{
    renderTo:Ext.getBody(),width:400,
    items:[
            {xtype:"tbspacer",flex:1},
            {text:"运行"},
            {xtype:"tbspacer",flex:1}
    ]
});
```

运行后就可看到“运行”按钮居中显示了。

14.3.7 分页工具栏：Ext.toolbar.Paging

PagingToolbar 派生于 Toolbar 对象，是将分页所需的导航、刷新、提示信息等封装起来的

工具栏，它大大减轻了开发人员基于分页的工作。开发人员只需要将 Store 绑定到分页工具栏就可实现分页，无需再做其他编码工具，非常方便。

在使用 Ext JS 3 之前的版本的分页工具栏，会为在固有的按钮中插入自己的按钮而烦恼。在 Ext JS 4 中，这将是一个简单的工作，使用组件的 insert 方法在工具栏中某个位置插入一个组件很容易；而工具栏全部子组件的实例都能在 items 属性中可以找到，因而可以在 items 中找到该组件，然后使用 remove 方法删除该组件，不过最好的办法还是使用组件的 hide 方法将其隐藏。下面来实践一下，打开模板页，在命令中输入以下命令在页面显示一个分页工具栏：

```
var p=Ext.create(Ext.PagingToolbar);
Ext.create(Ext.Panel,{
    width:400,height:400,renderTo:Ext.getBody(),
    tbar:p
});
```

现在要隐藏“刷新”按钮，得先知道“刷新”按钮的索引是多少，这个不难，输入以下命令：

```
console.dir(p.items.items)
```

在控制面板中将看到以下输出信息：

```
0  [Trial] Ext.button.Button { itemId="first", id="button-1015"}
1  [Trial] Ext.button.Button { itemId="prev", id="button-1016"}
2  [Trial] Ext.toolbar.Separator { id="tbseparator-1017"}
3  [Trial] Ext.toolbar.TextItem { id="tbtext-1018", text=" 第 "}
4  [Trial] Ext.form.field.Number { itemId="inputItem", name="inputItem", id="number-
            field-1019"}
5  [Trial] Ext.toolbar.TextItem { itemId="afterTextItem", text=" 页，共 1   页 ",
            id="tbtext-1020"}
6  [Trial] Ext.toolbar.Separator { id="tbseparator-1021"}
7  [Trial] Ext.button.Button { itemId="next", id="button-1022"}
8  [Trial] Ext.button.Button { itemId="last", id="button-1023"}
9  [Trial] Ext.toolbar.Separator { id="tbseparator-1024"}
10 [Trial] Ext.button.Button { itemId="refresh", id="button-1025"}
```

可以看到，工具栏里有 11 个组件，而“刷新”按钮的索引是 11，因而可以使用以下代码隐藏刷新按钮：

```
p.items.items[10].hide()
```

要在第一个分隔条后插入一个 Text 字段，可使用以下代码：

```
p.insert(3,new Ext.form.TextField());
```

经过以上操作后，分页工具栏将如图 14-6 所示。

图 14-6　经过隐藏和插入操作后的分页工具栏

使用分页工具栏最重要的两个配置项是 store 和 pageSize，store 就不用说了，pageSize 是

每页的记录数，工具栏会根据该值以及服务器端返回的总数，计算出页数。

下面看看分页工具栏的配置项、属性、方法和事件。

1. 配置项

- afterPageText：“下一页”按钮前的文本信息，默认值为“of {0}”，其中“{0}”会被格式化函数替换为实际总页数。该文本已在本地化文件中实现本地化，修改要最后在本地化文件中修改。
- beforePageText：在 Text 字段前的文本，默认值为“Page”。已在本地化文件中实现本地化。
- displayInfo：布尔值，如果设置为 true，会在工具栏最右边显示记录总数及当前页的记录的起始位置和结束位置。默认值为 false。
- displayMsg：displayInfo 为 true 时，在工具栏最右边显示的文本，默认值为“Displaying {0} - {1} of {2}”。已在本地化文件中实现本地化。
- emptyMsg：没有记录时显示的信息。默认值为“No data to display”。已在本地化文件中实现本地化。
- firstText：“第一页”按钮的提示信息，默认值为“First Page”。已在本地化文件中实现本地化。
- inputItemWidth：工具栏中 Text 字段的宽度。默认值为 30。
- lastText：“最后一页”按钮的提示信息，默认值为“Last Page”。已在本地化文件中实现本地化。
- nextText：“下一页”按钮的提示信息，默认值为“Next Page”。已在本地化文件中实现本地化。
- pageSize：必需项，每页记录数。
- prependButtons：如果设置为 true，可将配置项 items 中的组件插入到分页按钮的前面。默认值为 false。
- prevText：“上一页”按钮的提示信息，默认值为“Previous Page”。已在本地化文件中实现本地化。
- refreshText：“刷新”按钮的提示信息，默认值为“Refresh”。已在本地化文件中实现本地化。
- store：必需项，为工具栏定义数据源。

2. 属性

没有特别属性。

3. 方法

- bindStore：绑定 Store。
- doRefresh：执行刷新，相当于单击“刷新”按钮。
- moveFirst：移动到第一页，相当于单击“第一页”按钮。
- moveLast：移动到最后一页，相当于单击“最后一页”按钮。

- moveNext：移动到上一页，相当于单击“上一页”按钮。
- movePrevious：移动到上一页，相当于单击“上一页”按钮。
- unbind：取消 Store 的绑定。

4. 事件

- beforechange：在当前页发生改变的之前，会触发该事件。如果返回 false 会中止改变。
- change：在当前页发生改变后，会触发该事件。

14.3.8　使用工具栏

本节通过一个示例程序演示一下如何使用工具栏及其所有的条目。

使用模板页创建一个名称为 14-2.html 的页面文件，然后定义一个工具栏并渲染到 body 内，代码如下：

```
Ext.create(Ext.toolbar.Toolbar,{
    renderTo:Ext.getBody(),width:400,
    items:[
            {text:"按钮"},
            {xtype:"tbspacer",width:50},
            {xtype:"tbtext",text:"文本"},
            "-",
            {text:"按钮1"},
            "->",
            {text:"右边的按钮"}
    ]
});
```

代码一次创建了按钮、宽为 50 像素空白项、文本、分隔条、按钮、填充项和按钮。

在浏览器中打开页面，将看到如图 14-7 所示的效果。

图 14-7　工具栏示例的效果

14.4　使用 Ext.Action

14.4.1　概述

Action 对象是可重用的功能对象，熟悉 Delphi 的开发人员对这个应该不陌生。通过 Action 可以将其配置及行为共享到菜单的菜单项、工具栏的按钮等组件，从而减少重复代码，例如，在应用中，很多时候，工具栏和右键菜单都有相同的配置和操作，如果按钮和菜单项都分开定义，那么有很多重复的代码，而如果使用 Action 对象，就只需定义一次，减少重复代码，而且，当要启用或禁用按钮和菜单项时，只需要禁用 Action 就行，不需要分别对按钮或菜单项执行禁用操作。

要使用 Action 对象，组件必须支持以下方法：setText、setIconCls、setDisable、setVisible

和 setHandler。

14.4.2 Action 对象配置项和方法

1. 配置项

- disabled：布尔值，如果为 true，会禁用使用 Action 的组件。默认值为 false。
- handler：定义句柄函数。
- hidden：布尔值，如果为 true，使用该 action 的组件初始状态为隐藏状态。默认值为 false。
- iconCls：使用该 Action 的组件的图标样式名称。
- itemId：组件的 itemId。
- scope：作用域。
- text：使用该 Action 的组件的显示文本。

2. 方法

- disable：禁用所有使用该 Action 的组件。
- each：枚举使用该 Action 的组件。
- enable：启用所有使用该 Action 的组件。
- execute：执行 Action 在示例化时使用 handler 定义的或使用 sethandler 设置的句柄函数。
- getIconCls：获取图标样式。
- getText：获取显示文本。
- hide：隐藏使用该 Action 的文本。
- isDisabled：如果使用该 Action 的组件的状态为禁用状态，返回 true，否则返回 false。
- isHidden：如果使用该 Action 的组件的状态为隐藏状态，返回 true，否则返回 false。
- setDisabled：根据指定的值设置使用该 Action 的组件的启用或禁用状态。
- setHandler：设置句柄函数。
- setHidden：根据指定的值设置使用该 Action 的组件的隐藏或显示状态。
- setIconCls：设置图标样式。
- setText：设置显示文本。
- show：显示使用该 Action 的组件。

14.4.3 使用示例

使用模板页创建一个名称为 14-3.html 的页面文件，先定义一个样式用于显示图标：

```
.info{background:url("../images/view.png");}
```

接着要定义 Action，定义方法有两种，一种是一个个的定义，不过笔者更喜欢将所有 Action 定义在一个数组中，这样的话，如果按钮或者菜单是连着显示的，就不用一个个将变量写在 items 里。以下是 Action 的两种定义方式：

```
var single=Ext.create(Ext.Action,{
    text:"禁用或启用其他Action",
    handler:function(){
            var ma=multAction;
            for(var i=ma.length-1;i>=0;i--){
                    ma[i].setDisabled(!ma[i].isDisabled());
            }
    }
});
var multAction=[
    Ext.create(Ext.Action,{
            text:"查看",
            iconCls:"info",
            handler:function(){
                    Ext.Msg.alert("操作",this.text)
            }
    }),
    Ext.create(Ext.Action,{
            text:"新增",
            handler:function(){
                    Ext.Msg.alert("操作",this.text)
            }
    }),
    Ext.create(Ext.Action,{
            text:"编辑",
            handler:function(){
                    Ext.Msg.alert("操作",this.text)
            }
    }),
    Ext.create(Ext.Action,{
            text:"删除",
            handler:function(){
                    Ext.Msg.alert("操作",this.text)
            }
    })
]
```

变量 single 是常用的 Action 定义方法，一个变量对应一个 Action，该 Action 可禁用或启用下面数组内的 Action。变量 multAction 是把 Action 定义直接放在数组内，这样有时候挺方便。

现在定义一个面板，在面板的顶部工具栏使用定义的 Action 创建 5 个按钮，在面板内定义一个菜单，菜单内使用 Action 创建 5 个菜单项，然后再根据 Action 创建 5 个独立的按钮，代码如下：

```
var bar=[single].concat(multAction);
Ext.create(Ext.panel.Panel,{
    renderTo:Ext.getBody(),width:400,height:400,
    tbar:bar,
    items:[
            {xtype:"menu",floating:false,items:bar,width:200},
            Ext.create(Ext.button.Button, single),
            Ext.create(Ext.button.Button, multAction[0]),
            Ext.create(Ext.button.Button, multAction[1]),
```

```
                Ext.create(Ext.button.Button, multAction[2]),
                Ext.create(Ext.button.Button, multAction[3])
    ]
});
```

工具栏、菜单的定义相当简单，把 Action 直接放进去就行了。

注意 因为 4.1 Beta 1 版不支持“[single,multAction]”这样的写法，因而必须先将 single 和 multAction 组合成数组再使用。

在浏览器中打开页面，然后单击“禁用或启用其他 Action”，无论是按钮还是菜单项，结果是一样的，会看到如图 14-8 所示的效果，无论是按钮还是菜单项，通过 Action，只要执行一次就可把按钮或菜单项禁用了，这是 Action 的优势。

图 14-8 使用 Action 示例的效果

14.5 综合实例：在 Grid 中使用右键菜单

（1）功能描述

为 10.7.6 节示例添加右键菜单。

（2）实现代码

在实现代码前要注意，只有在视图类对象内才有右键事件 containercontextmenu 和 itemcontextmenu，containercontextmenu 的作用是单击容器时的操作，而 itemcontextmenu 是单击记录时的操作。如果要在其他位置添加右键事件，只能自己扩展该组件，自行添加事件。

首先将 10-8.html 复制一份，然后将复制文件的文件名修改为 14-4.html。

现在要定义一个 Action 数组，将工具栏里的按钮定义剪切出来，然后贴到数组，加上创建 Action 的代码。因为菜单的分隔条与工具栏分隔条的代码不同，所以要删除所有分隔条定义。因为菜单不在 Grid 内，使用 this.up 方法是找不到 Grid 的，所以要为 Grid 加上 id，值为 TerritoriesGrid，修改 Action 定义中相应的地方，最后 Action 数组的代码如下：

```
var op=[
    Ext.create(Ext.Action,{text:"增加",id:"add",disabled:true,handler:function(){
            //var grid=this.up("gridpanel");
            var grid=Ext.getCmp("TerritoriesGrid");
            var edit=grid.plugins[0];
            var rec=new Territories({
                TerritoryDescription:"",
```

```
                RegionDescription:""
            });
            edit.cancelEdit();
            grid.store.insert(0,rec);
            edit.startEdit(0,0);

    }}),
    Ext.create(Ext.Action,{text:"删除",id:"delete",disabled:true,handler:function(){
            //var grid=this.up("gridpanel");
            var grid=Ext.getCmp("TerritoriesGrid");
            var rs=grid.getSelectionModel().getSelection();
            if(rs.length > 0){
                    var content="确定删除以下地区？</br>";
                    for(var i=0;ln=rs.length,i<ln;i++){
                            content+=rs[i].data.TerritoryDescription+"<br/>";
                    }
                    Ext.Msg.confirm("删除记录",content,function(btn){
                            if(btn=="yes"){
                                    this.store.remove(this.getSelectionModel().
                                        getSelection());
                            }
                    },grid)
            }
    }}),
    Ext.create(Ext.Action,{text:"保存",handler:function(){
            //this.up("gridpanel").store.sync();
            Ext.getCmp("TerritoriesGrid").store.sync();
    }}),
    Ext.create(Ext.Action,{text:"刷新",handler:function(){
            //this.up("gridpanel").store.load();
            Ext.getCmp("TerritoriesGrid").store.load();

    }})
];
```

接着创建一个菜单，代码如下：

```
Ext.create(Ext.menu.Menu,{
    id:"contentMenu",
    items:op
});
```

代码中 id 是为了从菜单管理器中获取菜单。

直接为 Grid 添加视图的配置项，绑定 itemcontextmenu 事件，代码如下：

```
viewConfig:{
    listeners:{
            itemcontextmenu:function(view,record,item,index,e){
                    var menu=Ext.menu.Manager.menus["contentMenu"];
                    e.stopEvent();
                            menu.showAt(e.getXY());
                    return false;
            }
    }
}
```

使用右键菜单很简单，调用 stopEvent 停止事件的传播，阻止显示默认菜单就行了，然后使用 showAt 方法将菜单显示在发生鼠标单击事件的地方，使用事件的 getXY 方法可返回坐标。

因而，工具栏的按钮换成了 Action，所以还要修改两处地方，第一处是 Store 的 datachanged 事件里，激活增加按钮的地方，要修改为激活 Action，代码如下：

```
//Ext.getCmp("add").enable();
op[0].enable();
```

还要在 selectionchange 事件中激活“删除”按钮，修改后的代码如下：

```
//Ext.getCmp("delete").setDisabled(rs.length==0);
op[1].setDisabled(rs.length==0);
```

至此，修改就完成了。

（3）页面效果

在浏览器中，打开页面，并单击鼠标右键，将看到如图 14-9 所示的效果。

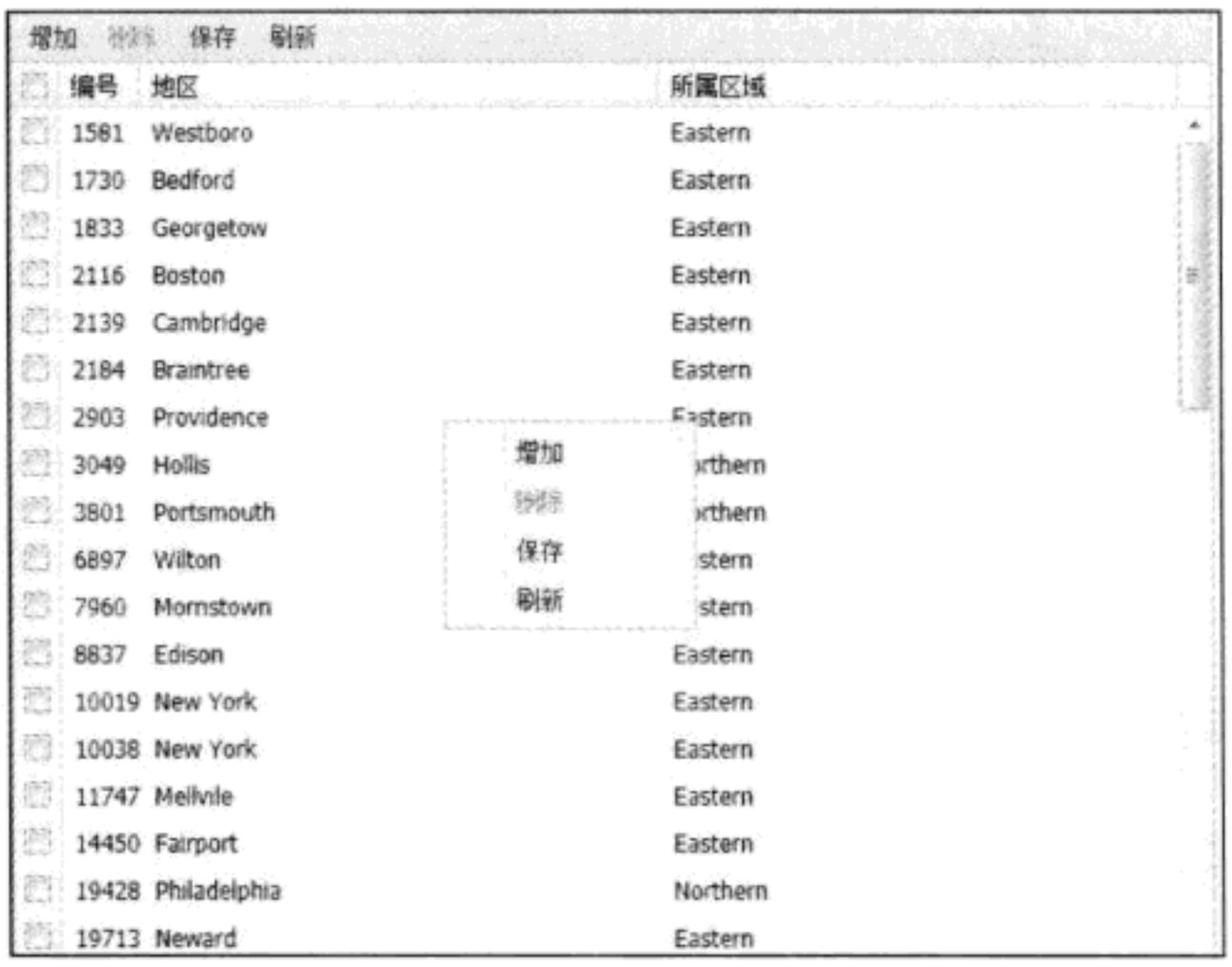


图 14-9　在 Grid 中使用右键菜单示例的效果

14.6　本章小结

按钮、菜单、工具栏虽然简单，但是如何方便地使用、减少重复代码等，是需要认真去理解和学习的，当然，这更需要经验的积累。在经验不足的时候，就不要去搞很炫很酷的菜单或工具栏，尽量先用简单的方法完成任务，当累积到一定经验，自然就会水到渠成了。

第 15 章　图形与图表

在 Ext JS 3 中，加入了图表功能，当时是使用 flash 的形式来画图表的，所以有不少问题，功能限制还很多。在 Ext JS 4 中，为了减少 flash 的影响，引入了图形功能，通过图形功能，不但能画出图表，能画图，还解决了面板标题栏竖向显示的问题。而这些，都是以浏览器的支持图形格式为基础构建的，IE[㊀]使用的是 VML 引擎，而其他浏览器使用的是 SVG 引擎。

本章将首先介绍 VML 和 SVG 这两个图形格式的基本知识，然后介绍图形功能及其使用，而重点则是图表的使用。

15.1　基础知识[㊁]

15.1.1　SVG 简介

SVG 是 Scalable Vector Graphics 的简写，中文名称是可缩放矢量图形，是基于 XML，用于描述二维矢量图形的一种图形格式。它是由 W3C 制订的一个开放标准。[㊂]

简单来说，就是通过 XML 的标记和属性来描述图形，例如以下标记：

```
<rect width="300" height="100" />
```

就表示在画一个 300×100 的矩形。

SVG 预定义的形状标记主要有：矩形（rect）、圆形（circle）、椭圆（ellipse）、直线（line）、折线（polyline）、多边形（polygon）和路径（path）。

而每个形状标记又由不同的属性来定义坐标（x、y）、长宽（width、height）或颜色（color）等。

为了实现特殊的显示效果，还有滤镜、渐变等标记。

下面通过一个简单示例来说明如何在页面中使用 SVG。使用模板页创建一个名称为 15-1.html 文件。首先要做的是在页面中添加 svg 标记，表示该部分是 SVG 图形定义：

```
<svg width="100%" height="100%" version="1.1" xmlns="http://www.w3.org/2000/svg">
</svg>
```

其中的宽度和高度表示图形的宽度和高度。和 HTML 的定义一样。

现在，要画一个从坐标（50，50）开始，使用红色填充的 100×100 的正方形，可在 svg 标记内加入以下代码：

㊀ IE 从 9.1 版开始支持 SVG。

㊁ 因为笔者对 SVG 和 VML 研究不深，而且本书重点是 Ext JS，因而有错漏在所难免，请见谅。

㊂ SVG 与 VML 资料均来自于维基百科。

```
<rect x="50" y="50" width="100" height="100" fill="red" />
```

看代码是否觉得很简单，和普通的 HTML 代码没什么区别。简单来说，SVG 就是那么简单，但是对于复杂的图形，例如图表，需要定义的代码就可以很多了。不过，使用 Ext JS，我们不需要自己画这些图表，Ext JS 已经为我们代劳了。

好了，现在的问题是如何使用 JavaScript 控制这个 SVG 图形，既然 SVG 标记和 HTML 标记没什么区别，也就说是，可以使用选择器找到 SVG 标记，然后通过 DOM 在其内部追加节点就可以了，例如，以下代码：

```
var svg=Ext.query("svg")[0];
var el = document.createElementNS("http:/" + "/www.w3.org/2000/svg", "circle");
el.setAttribute("cx","100");
el.setAttribute("cy","100");
el.setAttribute("r","50");
el.setAttribute("fill","green");
svg.appendChild(el);
```

创建 SVG 节点要使用 createElementNS 方法，这与 HTML 有点不同。而 setAttribute 和 appendChild 方法则是我们属性的 DOM 操作语法了。代码在刚才的正方形中画了一个绿色圆。

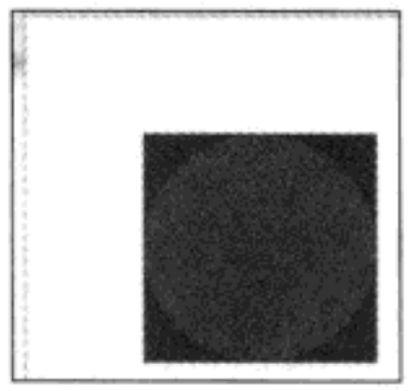


图 15-1　SVG 的示例

在浏览器打开页面，将看到如图 15-1 所示的效果。

通过示例和简单的介绍，不知道你有点头绪没有？没有的话，就找点 SVG 的资料继续研究，笔者推荐 W3school 这个网站（http://www.w3school.com.cn/svg/index.asp），对 SVG 有很清晰的讲述。

15.1.2　VML 简介

VML 是 Vector Markup Language 的简写，中文名称是矢量标记语言，是一种基于 XML 用于绘制矢量图形的语言。1998 年 VML 建议书由微软和 Macromedia 等向 W3C 提出审核，但遭到拒绝，因为 Adobe、Sun 等提出了 PGML 计划书。最后，这两套标准合并成更具潜力的 SVG。

VML 并没有因为遭到拒绝而被放弃，微软利用其浏览器霸主地位，从 IE 5 开始，将 VML 加入了浏览器中，延续至 IE 8。IE 9 开始使用 SVG，对 VML 不再支持。

VML 的使用比 SVG 复杂一点，首先是要声明页面的名称空间，代码如下：

```
<HTML xmlns:v="urn:schemas-microsoft-com:vml">
```

然后是要定义以下样式：

```
v\:*{behavior:url(#default#VML) }
```

其作用是把命名空间“v”和系统预定义的行为 VML 连接。现在就可以开始使用 VML 画图了，先使用模板创建一个名称为 15-2.html 的页面文件，然后把 HTML 的名称空间加上，再把样式定义加上。接着与上一节示例一样，先画一个正方形：

```
<v:rect FillColor="red" style='position:absolute;top:50;left:50;width:100;height:
    100;'></v:rect>
```

VML 语句都以“v：”开头，后面接形状标记，有些定义是以属性定义的，有些则以样式形式定义的，这个比 SVG 复杂多了。例如中，填充颜色要使用属性 FillColor。而要定义位置，要像 HTML 那样，先确定元素的定位方式，示例中使用了绝对定位。然后像 HTML 元素一样，定义左上角坐标，然后定义宽度和高度，不过这些值又不需要加“px”，够混乱的。还要记得加结束标记“</v:rect>”。

使用脚本操作画图 VML 倒比 SVG 简单些，例如画上一节示例的圆，代码如下：

```
var el=document.createElement("<v:oval FillColor='green' style='position:relative
    ;top:50;left:50;width:100;height:100;'></v:oval>");
document.appendChild(el);
```

在 IE 8 及以下版本的浏览器中打开页面，将看到类似图 15-1 所示的效果，不同的是，正方形和圆会多一个边框。

VML 随着浏览器的升级，估计会逐步被淘汰，但由于 IE 的传统统治地位，估计 IE 6 到 IE 8 这些浏览器离最终消失还有几年的时间，所以，还是得了解一下 VML。笔者推荐到 http://www.itlearner.com/code/vml/index.html 这里学习。更详细的信息还是要到微软 MSDN 网站查找。

15.2　图形介绍

15.2.1　概述

从 15.1 节可以了解到，SVG 与 VML 在语法上差异很大，因而要实现跨平台，就必须像事件一样，统一接口，而这是通过 Surface 对象实现的。Surface 对象作为 Svg 对象和 Vml 对象的基类，为图形提供了统一的画图接口。

使用 Photoshop 或 AutoCAD 画一个图，必不可少的几样东西是画布、画笔、调色板和图层。在 Ext JS 的图形包中，DrawComponent 对象就是画布，DrawSprite 对象就是画笔，DrawColor 对象就是调色板，DrawCompositeSprite 对象就是图层。画好的图形要输出到页面，就要调用接口 Surface 对象，将其转换为浏览器支持的图形格式描述文本，然后渲染到页面。而这就是 Ext JS 画图功能的基本原理。

明白了原理，对画图功能的理解就容易多了。

15.2.2　画布的工作流程：Ext.draw.Component

DrawComponent 对象派生于 Component 对象，在 initComponent 方法中，只是添加了 mousedown、mouseup、mousemove、mouseenter、mouseleave 和 click 6 个事件。不过，其重点是在渲染上，下面是其 onRender 方法的代码：

```
onRender: function() {
```

```
        var me = this,
            viewBox = me.viewBox,
            autoSize = me.autoSize,
            bbox, items, width, height, x, y;
        me.callParent(arguments);

        if (me.createSurface() !== false) {
            items = me.surface.items;

            if (viewBox || autoSize) {
                bbox = items.getBBox();
                width = bbox.width;
                height = bbox.height;
                x = bbox.x;
                y = bbox.y;
                if (me.viewBox) {
                    me.surface.setViewBox(x, y, width, height);
                }
                else {
                    me.autoSizeSurface();
                }
            }
        }
    },
```

调用完组件的 onRender 后，调用 createSurface 方法，开始创建接口，代码如下：

```
createSurface: function() {
    var me = this,
        cfg = Ext.apply({}, {
            width: me.width,
            height: me.height,
            renderTo: me.el
        }, me.initialConfig),
        surface;

    // ensure we remove any listeners to prevent duplicate events since we refire
        them below
    delete cfg.listeners;
    surface = Ext.draw.Surface.create(cfg);
    if (!surface) {
        // In case we cannot create a surface, return false so we can stop
        return false;
    }
    me.surface = surface;

    function refire(eventName) {
        return function(e) {
            me.fireEvent(eventName, e);
        };
    }

    surface.on({
        scope: me,
```

```
            mouseup: refire('mouseup'),
            mousedown: refire('mousedown'),
            mousemove: refire('mousemove'),
            mouseenter: refire('mouseenter'),
            mouseleave: refire('mouseleave'),
            click: refire('click')
        });
},
```

首先构建一个配置对象，然后调用 Surface 对象的静态方法 create。注意，它会把 DrawComponent 对象的配置项复制到 Surface 对象里，因而在 DrawComponent 对象的配置对象中，可直接使用 Surface 对象的配置项。Surface 对象的 create 方法代码如下：

```
create: function(config, enginePriority) {
    enginePriority = enginePriority || ['Svg', 'Vml'];
    var i = 0,
        len = enginePriority.length,
        surfaceClass;
    for (; i < len; i++) {
        if (Ext.supports[enginePriority[i]]) {
            return Ext.create('Ext.draw.engine.' + enginePriority[i], config);
        }
    }
    return false;
}
},
```

这里会调用 Ext 的 supports 方法，检查浏览器是支持 SVG 还是 VML 格式，然后根据支持的格式创建 Svg 对象或 Vml 对象并返回。

回到 createSurface 方法，余下要做的是将 Svg 对象或 Vml 对象的 6 个事件绑定方法，实现鼠标操作。

如果图像是允许缩放、旋转或移动位置的，则需要为每个画笔创建 Matrix 对象，通过矩阵运算实现画笔的转换。如果是简单的自动调整大小，则调用 autoSizeSurface 方法，否则调用 Surface 对象的 setViewBox 方法。

从以上的代码可以看到，DrawComponent 对象的主要作用就是提供一个 DOM 节点以加载图形，而主要的工作是在 Surface 对象内完成的。

15.2.3　图形引擎及接口：Ext.draw.Surface、Ext.draw.engine.Svg 和 Ext.draw.engine.Vml

Surface 对象是 Svg 和 Vml 对象的基类，它的主要作用是提供接口，把画笔的数据转换为图形语言，然后渲染到画布上。从 15.2.2 节可以看到，实际上不会创建 Surface 实例，只是创建了其子类 Svg 或 Vml 的实例。

Svg 对象和 Vml 对象都没有构造函数，都是执行 Surface 对象的构造函数，代码如下：

```
constructor: function(config) {
    var me = this;
    config = config || {};
```

```
        Ext.apply(me, config);
        me.domRef = Ext.getDoc().dom;
        me.customAttributes = {};
        me.addEvents(
            'mousedown',
            'mouseup',
            'mouseover',
            'mouseout',
            'mousemove',
            'mouseenter',
            'mouseleave',
            'click'
        );
        me.mixins.observable.constructor.call(me);
        me.getId();
        me.initGradients();
        me.initItems();
        if (me.renderTo) {
            me.render(me.renderTo);
            delete me.renderTo;
        }
        me.initBackground(config.background);
    },
```

代码中，会让 domRef 属性指向 HTML 文档的根节点，因为 VML 格式需要修改根节点的命名空间。

接着是创建一个自定义属性的空对象，添加事件，初始化事件，设置实例的 id。

接着调用 initGradients 方法，初始化渐变效果，其代码如下：

```
initGradients: function() {
    var gradients = this.gradients;
    if (gradients) {
        Ext.each(gradients, this.addGradient, this);
    }
},
```

代码会枚举 gradients 配置项定义的效果配置对象，调用 addGradient 方法将其转换为对应的图形格式代码。

处理完效果后，就调用 initItems 方法处理画笔了，代码如下：

```
initItems: function() {
    var items = this.items;
    this.items = Ext.create('Ext.draw.CompositeSprite');
    this.groups = Ext.create('Ext.draw.CompositeSprite');
    if (items) {
        this.add(items);
    }
},
```

代码先创建两个 CompositeSprite 对象，用来记录每个画笔和图层。最后调用 add 方法添加画笔，其代码如下：

```
add: function(){
```

```
var args = Array.prototype.slice.call(arguments),
    sprite,
    index;

var hasMultipleArgs = args.length > 1;
if (hasMultipleArgs || Ext.isArray(args[0])) {
    var items = hasMultipleArgs ? args : args[0],
        results = [],
        i, ln, item;

    for (i = 0, ln = items.length; i < ln; i++) {
        item = items[i];
        item = this.add(item);
        results.push(item);
    }

    return results;
}
sprite = this.prepareItems(args[0], true)[0];
this.normalizeSpriteCollection(sprite);
this.onAdd(sprite);
return sprite;
},
```

如果是数组，会递归调用 add 方法。

如果不是数组，则调用 prepareItems 方法做预处理，代码如下：

```
prepareItems: function(items, applyDefaults) {
    items = [].concat(items);
    var item, i, ln;
    for (i = 0, ln = items.length; i < ln; i++) {
        item = items[i];
        if (!(item instanceof Ext.draw.Sprite)) {
            item.surface = this;
            items[i] = this.createItem(item);
        } else {
            item.surface = this;
        }
    }
    return items;
},
```

如果参数 items 已经是 DrawSprite 对象实例了，则把其 surface 属性指向当前对象，否则调用 createItem 方法创建 DrawSprite 对象实例，其代码如下：

Svg 对象的：

```
createItem: function (config) {
    var sprite = new Ext.draw.Sprite(config);
    sprite.surface = this;
    return sprite;
},
```

Vml 对象的：

```
createItem: function (config) {
    return Ext.create('Ext.draw.Sprite', config);
},
```

两者的差异是 Vml 没设置 suface 属性。至于为什么这样，笔者估计是和销毁对象有关。

方法 initItems 执行完后，如果存在 renderTo 配置项，就调用 render 方法，渲染图形，实际上就是把生成的图形语句通过 appendChild 方法写到 DrawComponent 提供的元素上。

15.2.4 画笔：Ext.draw.Sprite

DrawSprite 对象的作用就是把 SVG 或 VML 的基本形状平滑得在 Ext JS 中做成统一的形状接口，在渲染时，会根据 type 属性将其转换为对应的图形代码。因而其构造函数的主要工作就是调用 setAttributes 方法设置形状的属性。具体的代码就不详细说了，有兴趣可以自己进行研究。

15.2.5 图层：Ext.draw.CompositeSprite

CompositeSprite 派生于 MixedCollection 对象，因而具有 MixedCollection 对象的功能，在其基础上添加了 mousedown、mouseup、mouseover、mouseout 和 click 方法，以及添加了图层操作需要的 hide 和 show 等方法。

15.2.6 调色板：Ext.draw.Color

DrawColor 对象说是调色板，但实际上只是用来表示 RGB 颜色，并提供辅助功能在 HSL 颜色空间中获取颜色组件。

15.2.7 辅助对象：Ext.draw.Draw 与 Ext.draw.Matrix

Ext.draw.Draw 对象是单件模式的对象，主要作用是提供画图的功能函数。

Matrix 对象的作用是为图形提供矩阵运算的功能。

15.3 使用图形功能

15.3.1 简单的开始

本节将演示使用 Ext JS 的图形功能画出图 15-1 所示的效果。

使用模板页创建一个名称为 15-3.html 的页面文件，然后先创建一个 DrawComponent 组件，代码如下：

```
var draw= Ext.create(Ext.draw.Component,{
    width:400,height:400,renderTo:Ext.getBody(),
    viewBox:false,autoSize:false
});
```

接着画正方形，代码如下：

```
var rect=draw.surface.add({
    type:"rect",x:50,y:50,height:100,width:100,fill:"#f00"
});
rect.show(true);
```

填充颜色属性 fill 内的颜色，千万别用 red 或 green 这样的颜色名称，不然 IE 会出问题。

这里要注意，调用 Surface 的 add 方法并没有渲染 DrawSprite 对象实例，只是创建了对象而已，因而还需要调用 show 方法渲染它，而且必须设置参数为 true 才会进行渲染。

最后完成圆的代码：

```
var circle=draw.surface.add({
   type:"circle",x:100,y:100,radius:50,fill:"#008000"
});
circle.show(true);
```

至此，代码就全部完成了。不过，这样写有点累赘，在已知图形的情况下，代码不用写得那么麻烦，只要在 DrawComponent 的定义代码中，加入 items 配置项，并将 add 方法内的配置项放进 items 配置项的数组内就可以了。这样不用 show 方法就可以显示了。

从以上代码可以看到，在 Ext JS 中使用图形功能还是很简单的，关键是要熟悉这些对象的配置项、属性、方法和事件，最重要的还是要有想象力能把简单的图形组合出复杂的图形。

15.3.2　DrawComponent 对象的配置项、属性、方法和事件

1. 配置项

- autoSize：布尔值，如果为 true。绑定图形的 div 元素将会根据内容字段调整大小。默认值为 false。
- gradients：数组，由渐变效果的配置项对象组合而成。这些效果主要用于 DrawSprite 对象的 fill 属性。每个效果都有 id、angle 和 stops 三个配置项。其中，id 为效果的唯一名称，fill 属性将根据该 id 引用效果；angle 为角度，是渐变的角度；stops 用于定义从什么颜色过渡到什么颜色，其值为以 0 到 100 之间的数字为属性的对象，而属性值也是对象。
- items：由 DrawSprite 对象组成的数组。
- viewBox：布尔值，默认值为 true，图形将支持缩放、改变位置或进行旋转。

2. 属性

没有定义属于自己的属性。

3. 方法

- createSurface：创建 Surface 实例。

4. 事件

没有定义属于自己的事件。

15.3.3　Surface 对象的配置项、属性、方法和事件

1. 配置项

- height：图形的高度，默认值为 auto。
- width：图形的宽度，默认值为 auto。

2. 属性

没有定义属于自己的属性。

3. 方法

- add：在图形中增加一个 DrawSprite 对象。注意，这时不会渲染新增的 DrawSprite 对象。
- addCls：为指定的 DrawSprite 对象添加样式。
- addGradient：增加渐变效果。
- getGroup：根据指定 id 获取图层（CompositeSprite 对象实例）。
- getId：返回组件的 id。
- remove：删除指定的 DrawSprite 对象。
- removeAll：删除所有 DrawSprite 对象。
- removeCls：移除指定的 DrawSprite 对象的样式。
- setSize：设置 Surface 对象的大小。
- setSytle：设置指定的 DrawSprite 对象的样式属性。
- setText：设置指定的 DrawSprite 对象的显示文本。

4. 事件

- create：当 Surface 实例创建后会触发该事件。

15.3.4 DrawSprite 对象的配置项、属性、方法和事件

1. 配置项

- fill：填充的颜色。如果是渐变效果，格式为“url(#gradientId)”，其中 gradientId 为 gradients 配置项中渐变效果配置对象内的 id 配置项的值。
- font：定义字体。语法与 CSS 的 font 属性一样。
- group：DrawSprite 对象根据该属性分组，同组的 DrawSprite 对象可同时隐藏或显示，简单来说，就是图层的 id。
- height：当 type 为 rect 时，设置矩形的高度。
- opacity：设置不透明度。
- path：由路径组成的数组。使用类似 SVG 图形的 path 语法。
- radius：当 type 为 circle 时，设置圆的半径。
- size：当 type 为 square 时，设置正方形的尺寸。
- stroke：设置形状的边框颜色。
- stroke-width：设置形状的边框宽度。
- text：设置显示文本。
- type：设置图形的类型，值可以是 circle（圆形）、rect（矩形）、text（文本）、path（路径）、ellipse（椭圆）、square（正方形，没实现）和 imag（图片）。
- width：当 type 为 rect 时，设置矩形的宽度。
- x：图形位置的 x 坐标值。

- y：图形位置的 y 坐标值。
- translate：移动图形到指定坐标。值为包含 x 和 y 坐标的对象。
- rotate：旋转图形。值为配置对象。配置项 degrees 用于指定旋转角度，配置项 x、y 可用于旋转中心。
- scale：缩放图形。值为配置对象，配置项 x、y 用于指定向 x 轴或 y 轴缩放的倍数。而配置项 cx、cy 则可用于指定缩放的中心点。

2. 属性

- dd：执行 DragSource 对象实例。

3. 方法

- addCls：添加样式。
- getBBox：返回边界。
- hide：隐藏 DrawSprite 对象，如果参数设置为 true，会重新画对象。
- redraw：重画对象。
- remove：删除对象。
- removeCls：删除对象的样式。
- setAttributes：设置对象属性。
- setStyle：设置对象的 Style 属性。
- show：显示对象，如果参数设置为 true，会重画对象。

4. 事件

没有定义属于自己的事件。

15.3.5　CompositeSprite 对象的配置项、属性、方法和事件

1. 配置项

没有定义属于自己的配置项。

2. 属性

没有定义属于自己的属性。

3. 方法

- add：增加 DrawSprite 对象到图层。
- getBBox：返回图层的边界。
- hide：隐藏图层，也就是隐藏组内所有 DrawSprite 对象。
- remove：从图层中删除 DrawSprite 对象。
- setAttributes：为图层内所有 DrawSprite 对象添加属性。
- show：显示图层，也就是显示组内所有 DrawSprite 对象。

4. 事件

没有定义属于自己的事件。

15.3.6 使用基本图形

目前 Ext JS 图形功能只支持圆形、矩形、椭圆形和文本这 4 种基本图形，而且实现上还有些错误，使用时一定要注意。在使用前要先搞清楚每种基本图形都有什么配置项，API 只是列出了部分，具体的还是要看 Svg 和 Vml 对象定义脚本中的 minDefaults 属性，而且两者之间还有差异，笔者经过试验后总结出了一些可以共用的配置项。

矩形通用的配置项：

- x、y：左上角坐标。
- width、height：矩形的宽度和高度。
- fill：填充颜色。
- stroke、stroke-width 和 stroke-opacity：边框、边框宽度和边框的不透明度。
- opacity 和 fill-opacity：不透明度和填充的不透明度。

圆形的配置项：

- x、y：圆心坐标，一定要用 x、y，千万别用 cx、cy，不然 Vml 格式会显示不了。
- radius：半径。
- 边框、填充、不透明度等配置项与矩形相同。

椭圆形的配置项：

- x、y：圆心坐标。
- rx、ry：rx 为水平半径，ry 为垂直半径。
- 边框、填充、不透明度等配置项与矩形相同。

文字的配置项：

- x、y：左上角坐标。
- font-family：字体名称。
- font-size：字体大小。
- font-weight：字体的粗细。
- font-style：字体样式。
- text-anchor：文字锚固的位置。
- 边框、填充、不透明度等配置项与矩形相同。

下面通过一个示例演示如何使用基本图形及其配置项，每个图形都有一个不设置透明度及设置了透明度的示例。

使用模板创建一个名称为 15-4.html 的样式文件，然后添加以下代码：

```
var draw= Ext.create(Ext.draw.Component,{
    renderTo:Ext.getBody(),width:600,height:600,
    viewBox:false,
    items:[
          {type:"rect",x:20,y:20,width:100,height:50,fill:"#0f0",
```

```
            stroke:"#f00","stroke-width":20
    },
    {type:"rect",x:160,y:20,width:100,height:50,fill:"#0f0",
            stroke:"#f00","stroke-width":20,
            opacity:.3,"stroke-opacity":.5,"fill-opacity":.3
    },
    {type:"circle",x:70,y:150,radius:50,fill:"#0f0",
            stroke:"#f00","stroke-width":20
    },
    {type:"circle",x:210,y:150,radius:50,fill:"#0f0",
            stroke:"#f00","stroke-width":20,
            opacity:.3,"stroke-opacity":.5,"fill-opacity":.3
    },
    {type:"ellipse",x:100,y:270,rx:80,ry:30,fill:"#0f0",
            stroke:"#f00","stroke-width":20
    },
    {type:"ellipse",x:280,y:270,rx:80,ry:30,fill:"#0f0",
            stroke:"#f00","stroke-width":20,
            opacity:.3,"stroke-opacity":.5,"fill-opacity":.3
    },
    {type:"text",x:20,y:360,text:"测试",fill:"#0f0",
            "font-size":"40px","font-weight":"bold","font-style":"italic",
            "font-family":"黑体"
    },
    {type:"text",x:120,y:360,text:"测试",fill:"#0f0",
            "font-size":"40px","font-weight":"bold","font-style":"italic",
            "font-family":"黑体",
            stroke:"#f00","stroke-width":1
    },
    {type:"text",x:240,y:360,text:"测试",fill:"#0f0",
            "font-size":"40px","font-weight":"bold","font-style":"italic",
            "font-family":"黑体",
            stroke:"#f00","stroke-width":1,
            opacity:.3,"stroke-opacity":.5,"fill-opacity":.3
    }
  ]
});
```

分别在 Firefox 和 IE 6 中打开页面，将看到如图 15-2 所示的效果。除了文字，效果基本一样。文字的情况比较复杂，主要差异是在 Firefox 6 中，笔者测试过，在 IE 6、IE 9、Chorme 和 Oprea 中，显示是基本一致的，只有在 Firefox 6 中才出现图 15-2 的问题。估计因为版本不同，存在错误，这不奇怪，在使用的时候只能做足测试和尽量不要使用比较特殊的效果，毕竟现在浏览器更新太快，“要一招鲜，吃遍天”比较困难。

15.3.7　使用图片

在图形中可以加入图片，其通用配置项有：

- width、height：图片的宽度和高度。
- src：图片的路径。

其他的配置项，不是只能在 SVG 格式用就是只能在 VML 格式用，尤其是左上角坐标，

只能 SVG 格式才能使用，而要将图片使用 VML 定位到某个位置，暂时没找到解决办法。例如，以下图片的定义：

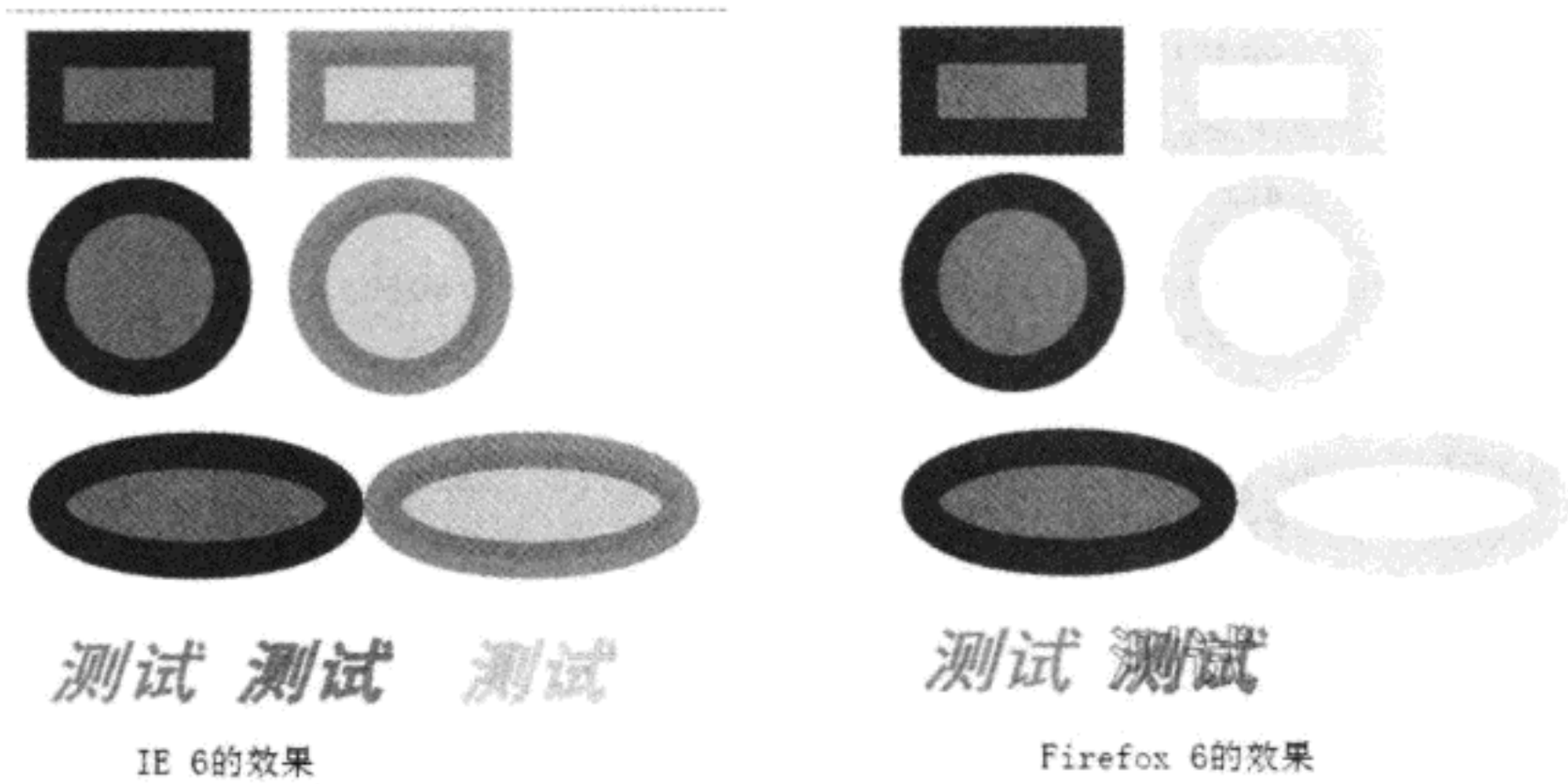


图 15-2　示例在不同浏览器上的效果

```
{type:"image",x:20,y:20,width:160,height:107,
   src:"../images/s1.jpg"
}
```

在使用 SVG 格式的浏览器，图片左上角坐标是（20，20），而在使用 VML 格式的浏览器，图片左上角坐标是（0，0），经研究源代码，因为 VML 格式的代码没将坐标信息写到代码里，所以图片显示位置不正确。因此，使用图片要注意这个问题，或者等补丁出来。

15.3.8　使用路径

路径（Path）的作用是描画一个形状的轮廓，它可以用来填充，画线，作为路径的片段，或者是以上三者的任意组合。

路径是通过点来描述的，在点与点之间，可以使用直线，也可以使用曲线来描画。描述点与点之间的描画是通过命令实现的，一共有以下 10 个命令：

- M：移动到某个点，要给出点的坐标（x y）。
- Z：结束路径。
- L：在当前点与指定点之间画一条直线，要给出指定点的坐标（x y）。
- H：在当前点与指定点之间画一条水平线，要给出指定点的 x 坐标。
- V：在当前点与指定点之间画一条垂直线，要给出指定点的 y 坐标。
- C：在当前点与指定点之间画一条三次贝塞尔曲线，要给出 3 个点的坐标（x1 y1 x2 y2 x y）。
- S：在当前点与指定点之间画一条平滑三次贝塞尔曲线，要给出 2 个点的坐标（x2 y2 x y）。
- Q：在当前点与指定点之间画一条二次贝塞尔曲线，要给出 2 个点的坐标（x1 y1 x y）。
- T：在当前点与指定点之间画一条平滑二次贝塞尔曲线，要给出 1 个点的坐标（x y）。

- A：在当前点与指定点之间画一条椭圆弧线，要给的参数包括椭圆的两个半径、椭圆相对于 x 轴的旋转角度、大圆弧标志、弯曲标准及指定点坐标。

以上命令要注意字母的大小写，大写字母表示使用绝对定位，小写字母表示使用相对定位。

以上命令的贝塞尔曲线的坐标等的详细内容请参阅 W3C 中有关 SVG 的说明（http://www.w3.org/TR/SVG11/paths.html#PathDataCurveCommands），在此就不详细说明了。

本节的重点是如何使用 path 来画图，下面通过示例来演示一下如何使用 path 来画一个从坐标（100，10）开始的边长为 100 的正六边形。

使用模板页创建一个名称为 15-5.html 的页面文件，然后先添加一个 DrawComponent 的定义，代码可以从 15.3.6 节示例中复制过来。

现在考虑怎么画正六边形，第一步肯定是先使用 M 命令将其移动到开始点（100，10），然后画一条 100 像素的水平线。接着是要先计算一下第三个点的坐标，横坐标是 250，纵坐标要计算一下，结果是 87 加 10，等于 97，也就是使用 L 命令将线画到坐标（250，97），接着使用 L 命令再画一条线（200，184）。然后画水平线到（100，184），再画线到（50，97）。最后将线连到起点。点和命令确定后就可以加入路径代码了：

```
{type:"path",stroke:"#f00",
   path:['M100 10 H200 L250 97 L200 184 H100 L50 97 L100 10 Z']

}
```

这里要注意，如果不设置 stroke 配置项，那么渲染出来的线是没有颜色的，因而看不到图形。如果不是描画轮廓，可使用 fill 配置项对图形进行填充。路径要在配置项 path 中定义，其值和模板的定义类似，使用数组组合出路径的字符串。注意绘图命令和绘图坐标的定义格式，先是命令，然后才是坐标和参数，坐标之间和命令之间要使用空格分隔。

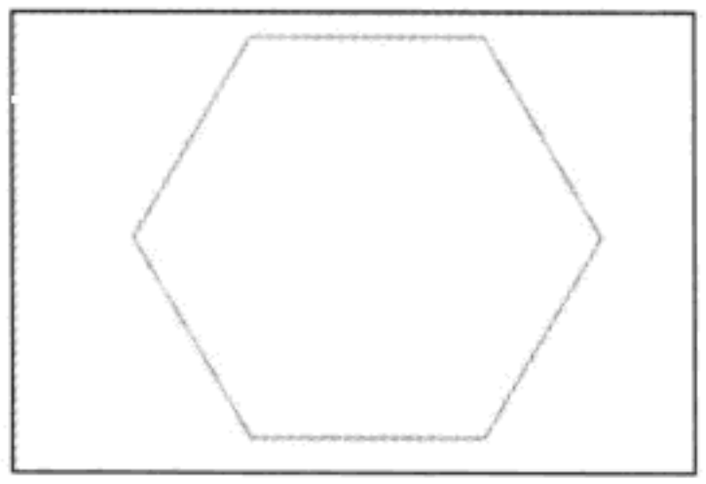


图 15-3　使用 Path 画出的正六边形

在浏览器中打开页面，将看到如图 15-3 所示的效果。

15.3.9　移动、旋转和缩放图形

本节将通过示例演示图形的移动、旋转和缩放属性。先使用模板页创建一个名称为 15-6.html 的页面文件。先根据允许的格式定义 4 个对象，代码如下：

```
var move={translate:{x:50,y:50}},
rotate1={rotate:{degrees:45}},
rotate2={rotate:{x:50,y:50,degrees:45}},
scale={scale:{x:0,y:0}};
```

第一个 move 是用来实现移动的，其中 x 和 y 分别代表在 x 轴和 y 轴上的移动距离。第二个 rotate1 是用来实现旋转的，其中 degrees 表示顺时针旋转的角度。第三个也是用来实现旋转的，与第二个的区别是可以通过 x、y 来定义旋转的支点，旋转将围绕这个点进行旋转，而第二个方法是围绕图形的中心点进行旋转的。第四个 scale 用来实现缩放，x、y 的值，如

果在 0 到 1 之间，表示图形在 x 轴或 y 轴上的缩小百分比，例如 0.5 表示缩小一半；如果值是大于 1，则表示图形在 x 轴或 y 轴上放大的比例，例如 2 表示放大一倍。

接着定义一个左上角坐标为（50，50），大小为 50×50 的矩形：

```
var rect=Ext.create(Ext.draw.Sprite,{
   type:"rect",x:50,y:50,height:200,width:200,fill:"#f00"
});
```

接着定义一个面板，面板内放一个 DrawComponent，用来显示矩形。还要在底部定义 4 个工具条，分别用于执行移动、两个旋转和缩放的操作。具体代码如下：

```
Ext.create(Ext.panel.Panel,{
   renderTo:Ext.getBody(),width:600,height:400,
   layout:'fit',
   items:[
          {xtype:"draw",viewBox:false,id:"Draw",items:[
                 rect
          ]}
   ],
   dockedItems:[
          {xtype:'toolbar',dock:"bottom",
                 items:[
                 {xtype:"numberfield",labelWidth:20,fieldLabel:"X",
                        value:50,width:80,minValue:0,maxValue:550,
                        listeners:{
                               change:function(f,n){
                                      move.translate.x=n;
                               }
                        }
                 },
                 {xtype:"numberfield",labelWidth:20,fieldLabel:"Y",
                        value:50,width:80,minValue:0,maxValue:350,
                        listeners:{
                               change:function(f,n){
                                      move.translate.y=n;
                               }
                        }
                 },
                 {text:"移动",handler:function(){
                        rect.setAttributes(move,true);
                 }}
          ]
   },
   {xtype:'toolbar',dock:"bottom",
          items:[
   {xtype:"numberfield",labelWidth:40,fieldLabel:"角度",
          value:45,width:100,minValue:0,maxValue:360,
          listeners:{
                 change:function(f,n){
                        rotate1.translate.degrees=n;
                 }
          }
   },
   {text:"旋转（仅角度）",handler:function(){
```

```
                        rect.setAttributes(rotate1,true);
                }}
        ]
},
{xtype:'toolbar',dock:"bottom",
        items:[
                {xtype:"numberfield",labelWidth:20,fieldLabel:"X",
                        value:50,width:80,minValue:0,maxValue:550,
                        listeners:{
                                change:function(f,n){
                                        rotate2.rotate.x=n;
                                }
                        }
},
{xtype:"numberfield",labelWidth:20,fieldLabel:"Y",
        value:50,width:80,minValue:0,maxValue:350,
        listeners:{
                change:function(f,n){
                        rotate2.rotate.y=n;
                }
        }
},
{xtype:"numberfield",labelWidth:40,fieldLabel:"角度",
        value:45,width:100,minValue:0,maxValue:360,
        listeners:{
                change:function(f,n){
                        rotate2.rotate.degrees=n;
                }
        }
                },
                {text:"旋转（角度+支点）",handler:function(){
                        rect.setAttributes(rotate2,true);
                }}
        ]
},
{xtype:'toolbar',dock:"bottom",
        items:[
                {xtype:"numberfield",labelWidth:20,fieldLabel:"X",
                        value:0,width:80,minValue:0,maxValue:5,step:0.1,
                        listeners:{
                                change:function(f,n){
                                        scale.scale.x=n;
                                }
                        }
                },
                {xtype:"numberfield",labelWidth:20,fieldLabel:"Y",
                value:0,width:80,minValue:0,maxValue:5,step:0.1,
                listeners:{
                        change:function(f,n){
                                scale.scale.y=n;
                        }
                }
},
{text:"缩放",handler:function(){
        rect.setAttributes(scale,true);
```

```
                        }}
                ]
        }
        ]
});
```

第一个工具栏，有两个数字字段，字段值的改变会直接修改 move 内的 x、y 值；第二个工具栏对应于 rotate1，字段值的改变会修改 degrees 的值；第三个工具栏对应于 rotate2；第四个工具栏对应于 scale。

在浏览器中打开页面，然后在缩放中将 x、y 值修改为 0.5，单击“缩放”；接着直接单击“旋转（角度 + 支点）”；接着单击“旋转（仅角度）”；最后将移动按钮所在工具栏的 x 值改为 300 后，单击“移动”按钮。最终将看到如图 15-4 经过处理后叠加在一起的效果。根据效果，可以知道如果同时存在旋转和缩放效果，会先执行旋转，再执行缩放，不然第一旋转的支点应该是（75，75），而不是（50，50），这个要注意。

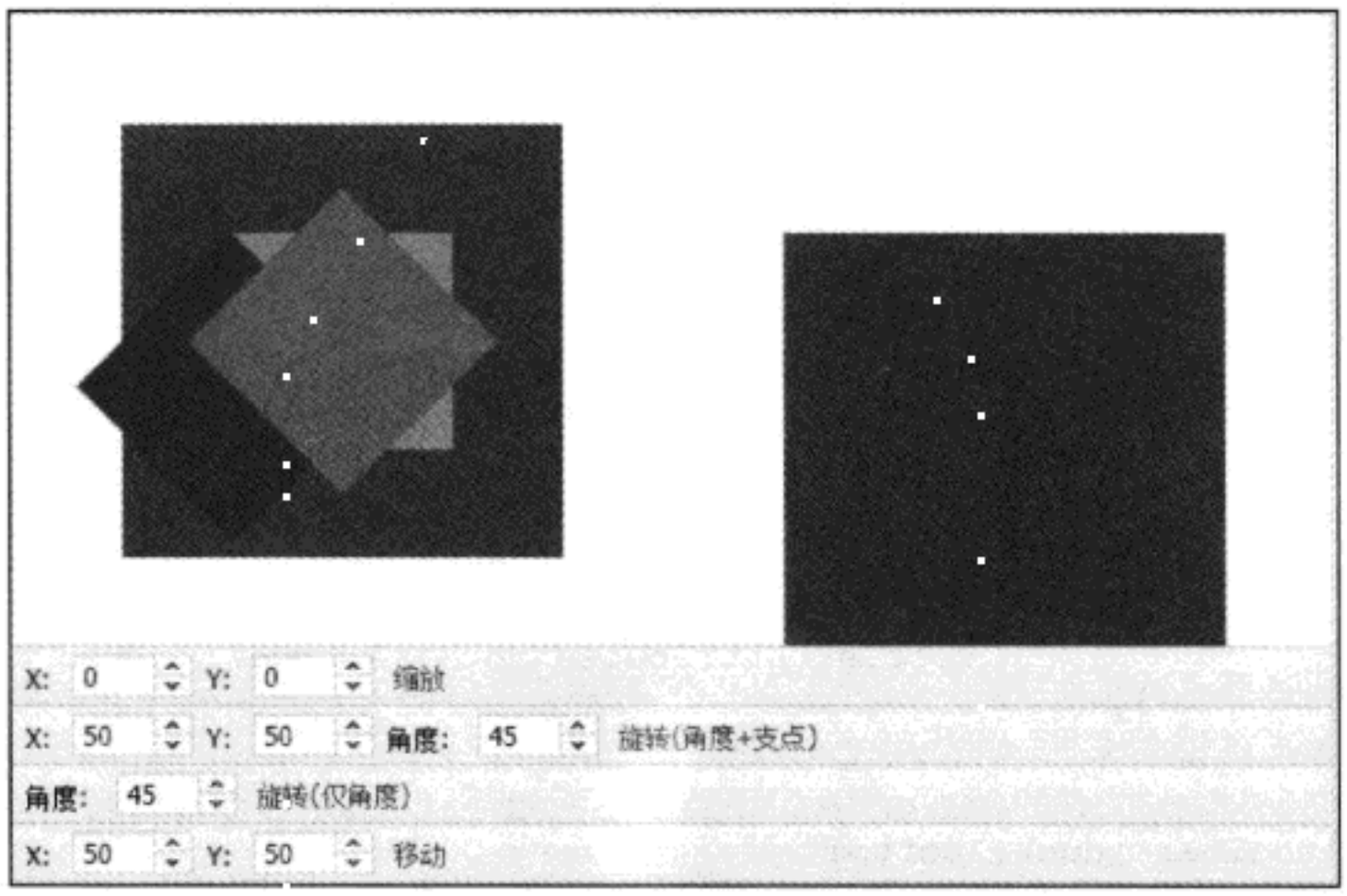


图 15-4　移动、旋转和缩放图形后叠加的效果

页面还有个问题在底部工具栏，因为是从底往上加工具栏的，所以会和定义的顺序相反，这个也要注意。

15.3.10　使用渐变效果

渐变效果要在 DrawnComponent 对象的配置项 gradients 中定义，每一个渐变效果就是一个配置对象，使用 id 来区分这些渐变效果。每个渐变对象除了 id 配置项，还包含以下两个配置项：

- angle：渐变的角度。
- stops：一个对象，用来定义渐变过程的颜色值。对象内，以 0 到 100 的数字表示渐变路径上某个位置最终的颜色，因而 0 和 100 两个属性是必需的，表示开始使用什么颜色，结束使用什么颜色。

在图形中使用渐变效果，要使用 fill 配置项，其值类似 CSS 背景图的设置，使用 url 表明

使用一个指向，url 内使用“#”符号加上 id 指向 gradients 配置项中定义的渐变效果。

下面通过一个示例来演示一下如何使用渐变效果。使用模板页创建一个名称为 15-7.html 的页面文件，先加入 DrawComponent 的定义：

```
Ext.create(Ext.draw.Component,{
    renderTo:Ext.getBody(),width:400,height:400,
    viewBox:false,
});
```

接着加入渐变的定义：

```
gradients:[
   {id:"g1",angle:0,stops:{
           0:{color:"#f00"},
           100:{color:"#0f0"}
   }},
   {id:"g2",angle:0,stops:{
           0:{color:"#f00"},
           50:{color:"#00f"},
           100:{color:"#0f0"}
   }},
   {id:"g3",angle:45,stops:{
           0:{color:"#f00"},
           100:{color:"#0f0"}
   }}
],
```

在定义中，angle 虽然是 0，也必须加入，不然 VML 默认情况下，是从上往下渐变的，而 SVG 是从左往右渐变的，如果没有 angel 配置项，它们的显示就有差异了。第一个渐变是从红色渐变到绿色；第二渐变是先从红色渐变到蓝色，然后再从蓝色渐变到绿色；第三个渐变是从红色渐变到绿色，但是渐变角度是从左上角到右下角。

最后定义两个矩形分别使用第一和第二个渐变效果，定义一个圆形使用第三个渐变效果，代码如下：

```
items:[
   {type:"rect",x:20,y:20,width:300,height:50,fill:"url(#g1)"},
   {type:"rect",x:20,y:80,width:300,height:50,fill:"url(#g2)"},
   {type:"circle",x:70,y:190,radius:50,fill:"url(#g3)"}
]
```

注意 fill 的写法。

在浏览器中打开页面，将看到如图 15-5 所示的效果。

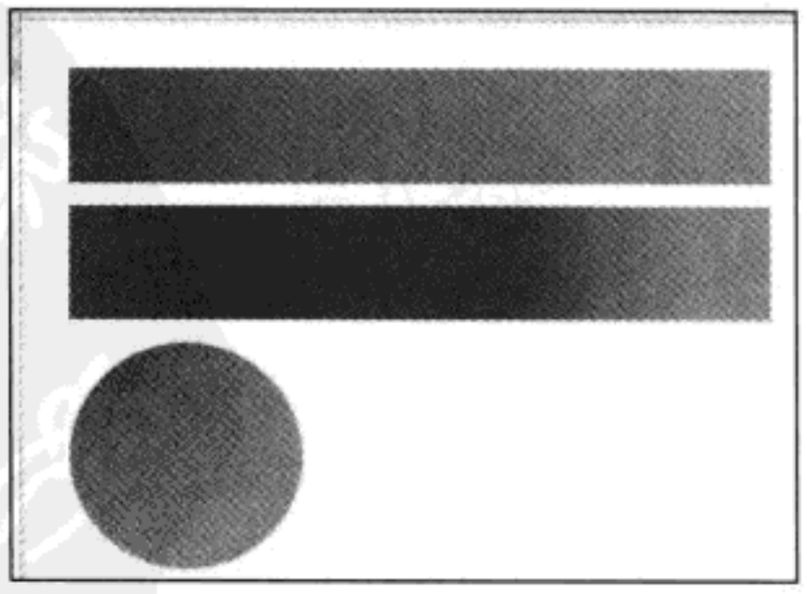


图 15-5　渐变示例的效果

在 W3C 有关 SVG 渐变的效果其实还有很多配置项，但是 VML 则没有太多，因而目前版本的画图功能只支持到颜色配置项。

15.3.11　使用图层

要使用图层，在定义 DrawSprite 对象时，加入 group

配置项就可以了。下面通过一个示例演示如何使用图层。使用模板页创建一个名称为 15-8.html 的页面文件，然后定义一个面板，在面板中放一个 DrawComponent 组件，在里面画两个矩形和两个圆，将填充颜色为红色的圆和矩形画到同一图层，图层名称为 group1，把填充颜色为绿色的圆和矩形画到同一图层，名称为 group1。在面板的底部工具栏中加一个按钮用来切换两个图层。具体代码定义如下：

```
Ext.create(Ext.panel.Panel,{
    renderTo:Ext.getBody(),width:600,height:200,
    layout:'fit',
items:[
{xtype:"draw",viewBox:false,id:"Draw",items:[
{type:"rect",x:20,y:20,width:100,height:100,fill:"#f00",group:"group1"},
{type:"circle",x:70,y:180,radius:50,fill:"#f00",group:"group1"},
{type:"rect",x:20,y:20,width:100,height:100,fill:"#0f0",group:"group2",hidden:true},
{type:"circle",x:70,y:180,radius:50,fill:"#0f0",group:"group2",hidden:true}
        ]}
],
bbar:[{
        text:"切换",handler:function(){
                var surface=this.up("panel").down("draw").surface,
                        g1=surface.getGroup("group1"),
                        g2=surface.getGroup("group2");
                if(g1.items[0].attr.hidden){
                        g1.show(true);
                        g2.hide(true)
                }else{
                        g2.show(true);
                        g1.hide(true)
                }
        }
    }]
});
```

第二组图形默认是隐藏的，因而只有 show 的时候才会显示。切换过程很简单，检查第一个矩形的属性（attr）里的 hidden 属性是否为 true，如果是，说明当前第一组是隐藏的，要调用 show 方法显示该组，隐藏第二组；否则显示第二组，隐藏第一组。

在浏览器中打开页面，将看到如图 15-6 所示的结果，单击“切换”按钮，将看到图形在红色和绿色之间切换。

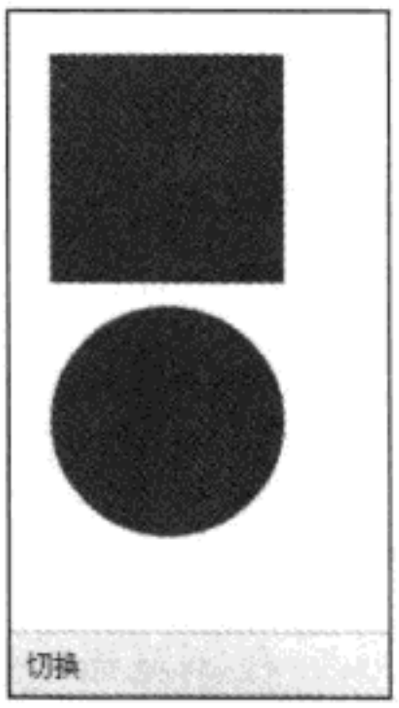


图 15-6 使用图层示例的效果

15.4 图表介绍

15.4.1 概述

图表是图形与数据的结合，图形可以使用 DrawComponent 的功能画出来，而数据则可通过 Store 提供，这就是 Ext JS 4 提供的图表功能。

为了实现和丰富图表功能，图表库中定义了很多类，表 15-1 详细说了这些类的作用。

表 15-1　图表库中类的详细说明

对象名	说明
Chart	派生于 DrawComponent 对象，混入了 Theme、Mask 和 Navigation 功能，作用是创建一个图表
Theme	为图表提供主题功能
BaseTheme	为新的图表主题提供基类
Mask	为图表提供区域选择功能，这样就可通过 select 事件实现一些特殊功能，如放大区域
Navigation	为图表提供平移和缩放功能
AbstractAxis	坐标轴类的抽象基类
Axis	派生于 AbstractAxis 对象，为图表提供坐标轴
Category	派生于 Axis 对象，为图表提供条目明确的坐标轴
Numeric	派生于 Axis 对象，为图表提供数字坐标轴
TimeAxis	派生于 Category 对象[⊖]，为图表提供时间坐标轴，坐标轴可实现分组和动态更新
GaugeAxis	派生于 AbstractAxis 对象，为表盘图表提供坐标轴
Radial	派生于 AbstractAxis 对象，为雷达图表提供坐标轴
Series	各系列图表的抽象基类，混入了 Label、Highlight、ChartTip 和 Callout 功能，为绘制图表提供公共的逻辑运算
Label	为图表提供标签功能
Highlight	为图表提供突出显示功能
ChartTip	为图表提供提示信息功能
TipSurface	为提示信息提供图形接口
Callout	为图表提供标注功能
Gauge	派生于 Series 对象，绘制表盘图表
Cartesian	派生于 Series 对象，是基于 x、y 坐标图表的基类
Scatter	派生于 Cartesian 对象，绘制散点图表
Line	派生于 Cartesian 对象，绘制折线图表
Bar	派生于 Cartesian 对象，绘制条形图表
ColumnChart	派生于 Bar 对象，绘制竖向条形图表
Area	派生于 Cartesian 对象，绘制面积图表
Pie	派生于 Series 对象，绘制饼图
Radar	派生于 Series 对象，绘制雷达图表
Legend	为图表提供图例
LegendItem	图例的条目
Shape	为图表提供基本图形

15.4.2　图表的工作流程

Chart 对象派生于 DrawComponent 对象，因而是一个画布。在画布内混入了 Theme、

⊖ API 中是派生于 Axis 对象，但实际代码是派生于 Category 对象，这里按实际代码定义说明。

Mask 和 Navigation 对象功能，在构造函数内会对混入功能进行初始化。而在 initComponent 方法内，才开始处理图表要显示的对象，具体代码如下：

```
initComponent: function() {
    var me = this,
        axes,
        series;
    me.callParent();
    me.addEvents(
            // 省略事件定义代码
    );
    Ext.applyIf(me, {
        zoom: {
            width: 1,
            height: 1,
            x: 0,
            y: 0
        }
    });
    me.maxGutter = [0, 0];
    me.store = Ext.data.StoreManager.lookup(me.store);
    axes = me.axes;
    me.axes =new Ext.util.MixedCollection(false, function(a) { return a.position; });
    if (axes) {
        me.axes.addAll(axes);
    }
    series = me.series;
    me.series = new Ext.util.MixedCollection(false, function(a){ return a.seriesId
        || (a.seriesId = Ext.id(null, 'ext-chart-series-')); });
    if (series) {
        me.series.addAll(series);
    }
    if (me.legend !== false) {
        me.legend = new Ext.chart.Legend(Ext.applyIf({chart:me}, me.legend));
    }

    me.on({
        mousemove: me.onMouseMove,
        mouseleave: me.onMouseLeave,
        mousedown: me.onMouseDown,
        mouseup: me.onMouseUp,
        scope: me
    });
},
```

代码调用 DrawComponent 的 initComponent 方法后，调用 addEvents 方法添加事件。接着初始化图表四边的最大空白值，设置 store。

接着就是创建 MixedCollection 实例，记录坐标轴（axes）和使用的图表（series）。

如果定义了 legend 配置项，则创建 Legend 对象实例。

最后绑定一些事件。

至此初始化组件工作就完成了。不过工作还没完成，与视图一样，在 afterRender 方法中，调用 bindStore 方法，将 Store 中的数据操作绑定到 Chart 中的方法，也就是当数据改变时，调用 refresh 方法更新图表的显示。在 refresh 方法内，是调用 redraw 方法重新刷新图表的，其代码如下：

```
redraw: function(resize) {
    var me = this,
        chartBBox = me.chartBBox = {
            x: 0,
            y: 0,
            height: me.curHeight,
            width: me.curWidth
        },
        legend = me.legend;
me.surface.setSize(chartBBox.width, chartBBox.height);
me.series.each(me.initializeSeries, me);
me.axes.each(me.initializeAxis, me);
me.axes.each(function(axis) {
    axis.processView();
});
me.axes.each(function(axis) {
    axis.drawAxis(true);
});
if (legend !== false) {
    legend.create();
}
me.alignAxes();
    if (me.legend !== false) {
        legend.updatePosition();
    }
    me.getMaxGutter();
    me.resizing = !!resize;
    me.axes.each(me.drawAxis, me);
    me.series.each(me.drawCharts, me);
    me.resizing = false;
},
```

代码在设置了画图区域后，会调用 initializeSeries 方法初始化 Series 对象；调用 initializeAxis 方法初始化坐标轴。如果有集合数据，调用坐标轴的 processView 方法，处理这些集合数据。然后调用 drawAxis 初始化图表，注意参数是 true，表示只做初始化，不渲染。

如果定义了 legend 配置项，调用 create 方法创建 Legend 对象实例。

接着调用坐标轴的 alignAxes 方法，调整坐标轴的位置。如果 legend 存在，调用其 updatePosition 方法调整位置。接着，调用 getMaxGutter 方法获取最大空白值。

最后调用 drawAxis 方法和 drawCharts 方法画出坐标轴和图表。在 drawAxis 方法中还是会调用坐标轴的 drawAxis 方法来画坐标轴。而在 drawCharts 方法中，会调用 Series 对象的 drawSeries 方法画出图表。

经过以上处理，图表就画出来了。

15.5 使用图表

15.5.1 从一个简单例子开始

图表目前有 8 种类型，每种类型都有与之相关的坐标轴或图例，而坐标轴等又有许多的配置项搭配，因而，我们先从一个简单示例开始，逐步来学习如何使用图表。

使用模板页创建一个名称为 15-9.html 的页面文件，然后加入以下代码：

```
Ext.create(Ext.data.ArrayStore,{
    id:"RevenueStore",
    fields:["Month",{name:"2010",type:"int"}],
    data:[["一月",30],["二月",32],["三月",29],
            ["四月",35],["五月",24],["六月",26],
            ["七月",31],["八月",33],["九月",35],
            ["十月",32],["十一月",34],["十二月",37]]
});
Ext.create(Ext.chart.Chart,{
    renderTo:Ext.getBody(),width:500,height:300,
     store: "RevenueStore",
     axes: [{
         type: 'Numeric',
         position: 'left',
         fields: ['2010']
     },
     {
         type: 'Category',
         position: 'bottom',
         fields: ['Month']
     }],
     series: [{
         type: 'line',
         axis: 'left',
         xField: 'Month',
         yField: '2010'
     }]
});
```

在浏览器打开，将看到如图 15-7 所示的效果。

我们先来看看示例有哪些关键定义，是必不可少的，Store 当然是。**Store 的重点是数据格式，其关键是，记录总数只影响坐标轴的划分**。这是为什么呢？就例子来说，目前显示的是 2010 年的数据，如果要显示 2011 年的数据，不能通过添加数据的方式来实现，因为这只会改变坐标轴的划分，例如在数据中添加数据“["一月",30]”，那么结果只是在横坐标多了一个一月的刻度，并不是我们预期的结果。如果要显示 2011 年的数据，必须通过加字段的方式实现。这是目前图表使用的数据格式，要谨记。

接着是坐标轴的定义（axes）可以看到，数组内，一个配置对象就是一条坐标轴，而配置项 position 则决定了坐标轴的位置。最关键的配置项是 fields，该配置项定义了该轴将使用哪个字段的值进行刻度划分。可以看到，坐标轴的 fields 使用的都是数组，这是因为一个坐标轴

可以对应多个字段，所以可以使用数组进行定义。当然，如果确定只有一个字段，则可以直接使用字段名称作为值。

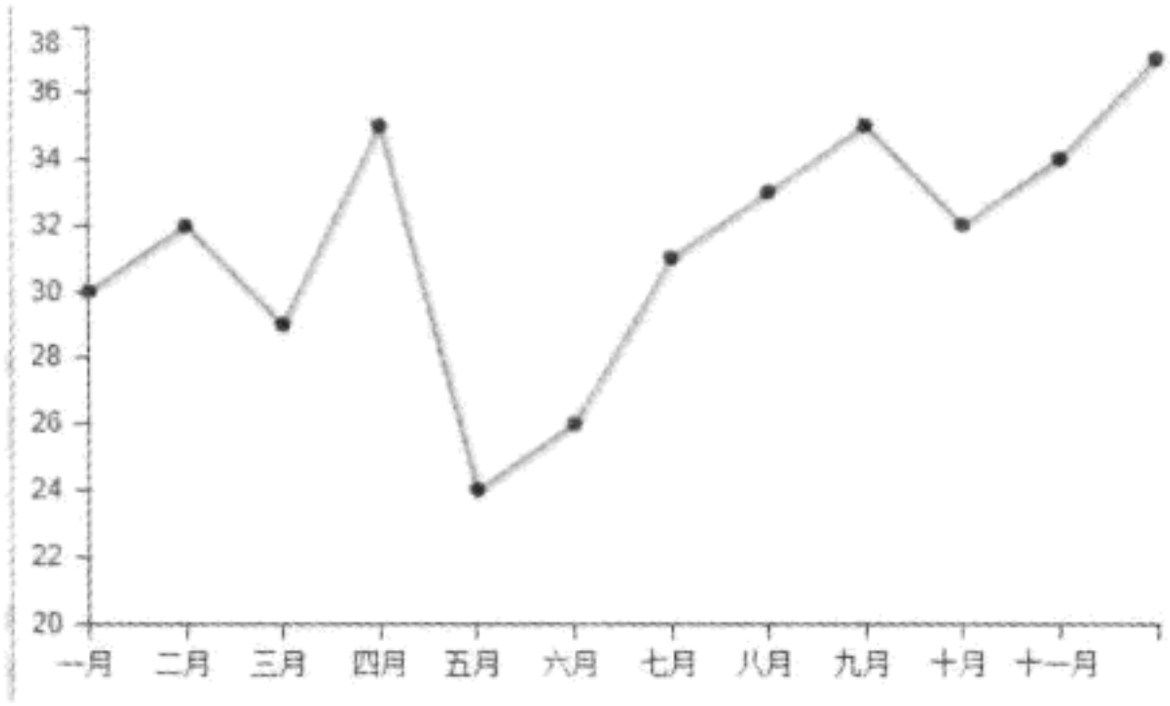


图 15-7　示例的页面效果

最后是图表类型的定义（series），示例中使用的折线图表，因而需要配置 axis、xField 和 yField 三个配置项，axis 的作用是声明数值在哪条坐标轴上，例如示例，因为数值轴是 y 轴，在 left，所以 axis 要设置为 left，否则将会像 x 轴那样使用相对比例。xField 的作用是定义 x 轴上的坐标值使用的是哪个字段的值，而 yField 则是定义 y 轴坐标使用的是哪个字段的值。不同类型的图表的配置项会有差别，这个要注意，例如，饼图就没有 axis、xField 和 yField 配置项。

本示例说明了图表的基本要素包括了 Store、坐标轴和图表类型。当然，如果是饼图，坐标轴也可以不要。

15.5.2　坐标轴的配置项

实际使用的坐标轴有 Numeric、Category、TimeAxis 和 GaugeAxis 这 4 种，除了 GaugeAxis 是直接继承自 AbstractAxis，其余三种都直接或间接继承自 Axis，因而这三种都有以下共同的配置项：

- dashSize：刻度线的长度或高度，默认值是 3。
- grid ：布尔值或对象。如果设置为 true，会在图表中显示网格线。当为对象时，可通过配置项 odd 设置奇数行的显示格式，而 even 则可设置偶数行的显示格式。无论是 ood 还是 even，其值都为一个 DrawSprite 对象。
- length：坐标轴位置的偏移量，默认值是 0。
- majorTickSteps ： 设置坐标轴如何划分坐标的主刻度，例如，坐标轴最小值为 0，最大值为 40，该值为 40，则主刻度的间隔是 1。使用时要注意，如果该值大于最大值和最小值之差，且计算出来的每格刻度四舍五入之后不是 1，那么就会出现错误。设置该值，应该保证每格间隔至少是 1。
- minorTickSteps ： 指定在两个主刻度之间如何划分小刻度，默认值为 0，没有小刻度。该功能类似尺子中的每 1 厘米刻度内每隔 1 毫米有 1 个小刻度。该值就是用来指定有

多少个小刻度的。

- position：坐标轴的位置，值可以为 left、bottom、right 和 top。默认值为 bottom。
- width：坐标轴宽度的偏移量。默认值为 0。
- label：为 DrawSprite 对象的配置项对象，用来设置在刻度上标签的显示方式。
- title：坐标轴的标题。

Numeric 自有的配置项：

- adjustMaximumByMajorUnit：布尔值，如果设置为 true，在最接近数据最大值的刻度上再加一个刻度。不过貌似有错误，在示例 15-9 中将该值设置为 true，会看到最后一个点的值是 37，但却显示在 36 之下，估计是四舍五入造成的。默认值为 false。
- adjustMinimumByMajorUnit：布尔值，如果设置为 true，会在最接近数据最小值的刻度下再加一个刻度。不过和 adjustMaximumByMajorUnit 一样，因为四舍五入会造成显示错误，使用时要小心。默认值为 false。
- decimals：小数的显示位数，默认值为 2。
- maximum：设置坐标轴上的最大值。
- minimum：设置坐标轴上的最小值。
- scale：坐标轴缩放时使用的算法，值可以是 linear 或 logarithmic。

Category 没有自有的配置项。

TimeAxis 自有的配置项：

- aggregateOp：定义分组的时候使用的聚合操作，值可以是 sum（合计）、avg（平均值）、max（最大值）或 min（最小值）。默认值是 sum。
- constrain：布尔值，如果为 true，图表只会渲染在 fromDate 到 toDate 之间的数据。默认值为 false，图表会根据新值重新计算坐标轴的刻度。
- dateFormat：指定渲染的日期格式。
- fromDate：坐标轴的起始时间。
- setp：一个包含两个值的数组。第一个值表示坐标轴的单位，第二个值表示每一个刻度包含多少个单位。默认值为“[Ext.Date.DAY, 1]”，也就是每一刻度就是一天。
- groupBy：设置每一刻度的时间单位，值可以是 day、month、year 或由以上三个使用逗号组合起来的值，如“year.month”。
- toDate：坐标轴的结束时间。

GaugeAxis 对象的配置项对象：

- margin：标签和刻度线之间的偏移量，默认值是 10。
- maximum：坐标轴的最大值。
- minimum：坐标轴的最小值。
- steps：设置刻度之间的间隔。
- title：坐标轴的标题。
- lable：为 DrawSprite 对象的配置项对象，用来设置刻度上标签的显示方式。

在 15.5.1 节示例中，为了能清晰的看清楚坐标点的位置，可以使用 gird 设置网格线，并

将其坐标原点设置为 0，每格刻度细化到 2，并加上标题“营收（单位：万）”具体代码如下：

```
title:"营收（单位：万）",
minimum:0,
maximum:40,
majorTickSteps:20,
grid:{
    even:{fill:"#ccc"}
}
```

从图中可以看到“十二月”这标签因为位置不足，显示不了，可以将标签逆时针旋转 90 度，竖向看就可以了，再加个标题，代码如下：

```
label:{
    rotate:{degrees:-90}
},
title:"月份"
```

刷新一下页面，将看到如图 15-8 所示的效果。

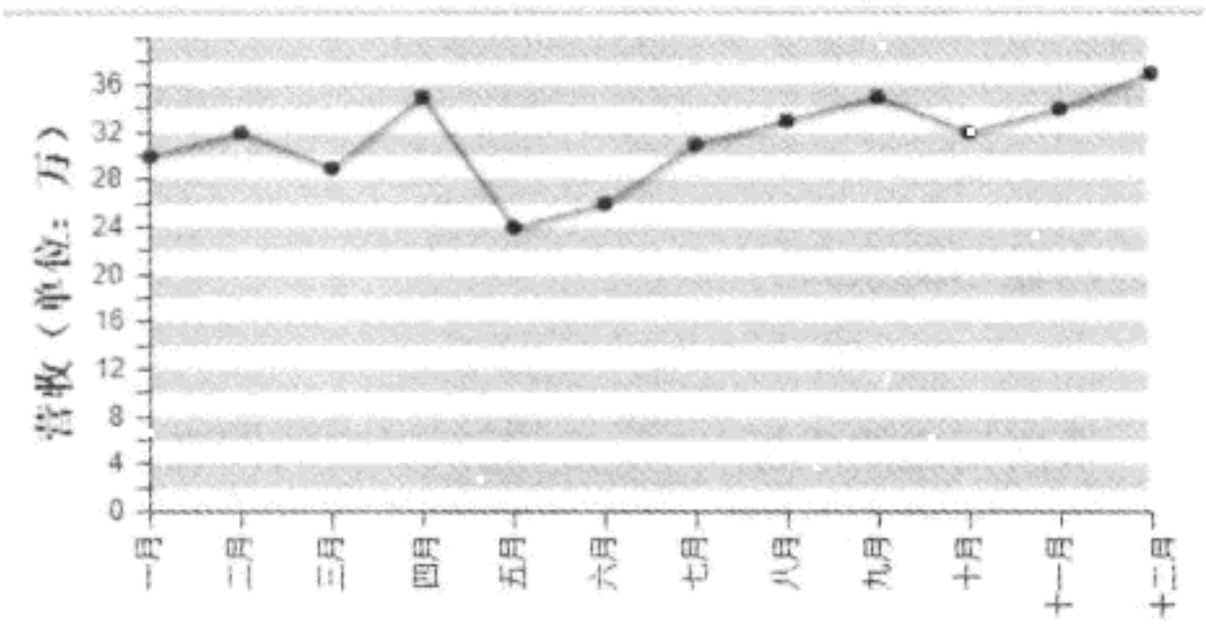


图 15-8　修改过坐标轴配置的示例的页面效果

15.5.3　Series 对象的配置项、属性、方法和事件

因为所有类型的图表都派生于 Series 对象，所以它们会继承 Series 对象的配置项、属性、方法和事件，因而要使用图表。必须先了解 Series 对象的配置项、属性、方法和事件。

1. 配置项

- callouts：Callout 对象的配置对象。
- color：标签的颜色，默认值是“#000”（黑色）。
- display：指定是否在饼图的每一个切片显示标签及标签位置。其值可以是 rotate、middle、insideStart、insideEnd、outside、over、under 或者 none，其中 none 会禁止标签渲染。默认值为 none。
- field：要显示为标签的字段名称。默认值为 name。
- font：标签的字体。默认值为“11px Helvetica, sans-serif”。

- highlight：布尔值或对象，如果设置为 true，当鼠标移动到图表标记上面时，会突出显示该标记。也可以定义为与 DrawSprite 对象样式属性相同的配置项，这样当突出显示时，会将样式应用到标记。默认值为 false。
- label：Label 对象的配置对象，定义标签。
- listeners：定义监听事件。
- minMargin：设置标签到原始显示效果的距离。该配置项通常用于饼图宽度成为饼图切片的长度。默认值为 50。
- orientation：设置图表的方向，值可以为 horizontal（水平方向）或 vertical（垂直方向）。
- renderer：与 Grid 一样，为字段渲染样式。函数的参数依次是 DrawSprite 对象、Record（记录）、attributes（属性）、index（索引）和 Store 对象。
- shadowAttributes：阴影属性组成的数组。
- showInLegend：布尔值，如果为 true，会将该图表显示在图例中。
- tips：ToolTip 的配置对象，定义标记的提示信息。
- title：图表的标题。
- type：图表的类型。

2. 属性

- highlight：图表突出显示的条目。

3. 方法

- getItemForPoint：根据指定的坐标（x，y）返回对应的图表。
- getLegendColor：根据指定索引返回图例的颜色。
- hideAll：隐藏图表的所有元素。
- highlightItem：突出显示指定的条目。
- setTitle：设置标题。
- showAll：显示图表的所有元素。
- unHighlightItem：将当前突出显示的条目取消突出显示效果。

4. 事件

- titlechange：当图表的标题发生改变的时候会触发该事件。

15.5.4 折线图的配置项

折线图在继承 Series 对象的配置项基础上，添加了以下配置项：

- axis：声明数值在哪条坐标轴上。例如示例，因为数值轴是 y 轴，在左边，因而 axis 要设置为 left，否则将会像 x 轴那样使用相对比例。
- fill：布尔值，如果为 true，就会使用 style 配置项中的 fill 配置项定义的颜色填充折线下的区域，不透明度则采用 opacity 配置项定义的值。默认值为 false。
- markerConfig：标记的配置对象，用于定义标记的显示样式，值可以为 DrawSprite 对象

的配置对象。只有 showMarkers 设置为 true 时才有效。

- selectionTolerance：定义光标位置与折线位置之间的可触发事件的距离。默认值为 20。
- showMarkers：布尔值，如果设置为 true，会在数据点上显示标记。默认值为 false。
- smooth ：如果设置为 true 或者非 0 值，点之间的连线会很平滑的环绕数据点。否则会使用直线连接数据点。
- style：定义连接线的样式，该样式会覆盖主题样式。
- xField：定义应用在 x 轴的数据的字段名称。
- yField：定义应用在 y 轴的数据的字段名称。

现在在 15.5.1 节示例的图表定义中加入突出显示效果；利用提示信息显示点的数字；在折线下填充绿色，并设置不透明度为 0.2；将点的标记放大点；使用平滑线。具体代码如下：

```
highlight:true,
tips: {
      renderer: function(rec) {
                  this.update(rec.get('2010'));
      }
},
fill:true,
style:{
   fill:"#0f0",
   opacity:0.2
},
showMarkers:true,
markerConfig:{
   type:"circle",
   radius:5,
   fill:"#f00"
},
smooth:true
```

要显示什么信息，则要在 renderer 函数内使用 update 方法更新 ToolTip 对象的内容。要在线下填充颜色，则要先设置 fill 为 true，然后在 style 中设置样式。标记也要先设置 showMarkers 为 true，然后在 markerConfig 中配置标记的类型。

在浏览器中刷新页面将看到如图 15-9 所示的效果。

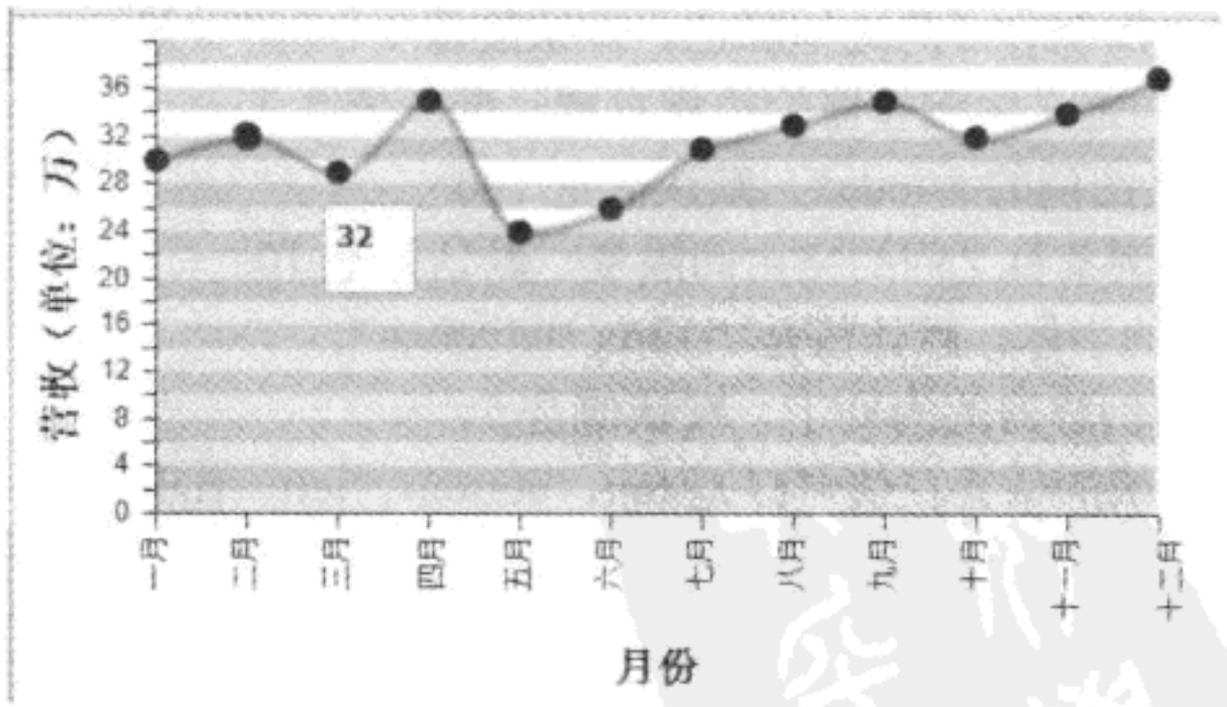


图 15-9　添加图表配置项后示例的页面效果

15.5.5 显示多个折线图及使用图例

在 15.5.1 节示例中只显示了一个年度的折线图，如果想添加一个年度做对比，很简单，下面通过示例演示如何添加多个折线图，以及如何使用图例对曲线标识。

先将文件 15-9.html 复制一份，将复制后的文件的文件名修改为 15-10.html。首先是修改 Store 的定义，在 15.5.1 节中，已经说明要添加年度，必须添加字段，因而，在 Store 定义中添加 2011 年度字段，并在数据中添加 2011 年度的数据，最后的代码如下：

```
Ext.create(Ext.data.ArrayStore,{
    id:"RevenueStore",
    fields:["Month",{name:"2010",type:"int"},
            {name:"2011",type:"int"}
    ],
    data:[["一月",30,32],["二月",32,28],["三月",29,33],
            ["四月",35,35],["五月",24,26],["六月",26,35],
            ["七月",31,43],["八月",33,31],["九月",35,42],
            ["十月",32,38],["十一月",34,45],["十二月",37,34]]
});
```

接着要修改 Y 轴，把 minimum、maximum 和 majorTickSteps 配置项删除，在 fields 中添加字段 2011，把 grid 的值修改为 true，具体代码修改如下：

```
fields:['2010','2011'],
grid:true
```

先把原图表中与填充和标记有关的代码删除，把 smooth 也删除。然后复制一份 2010 年度图表，粘贴在 series 中，把 yField 修改为 2011。把 tips 中 setTitle 内的 2010 也修改为 2011。

这样，两个年度的折线图就做好了，现在的问题是无法区分哪条线是 2010 年度的，哪条线是 2011 年度的，因而要加入图例。首先是在 Chart 的定义中加入 legend 配置项，在配置对象里要定义其显示位置，这里显示在图表顶部比较合适，因而要加入以下代码：

```
legend:{position:"top"},
```

Legend 对象还有以下配置项：

- boxFill：定义图例容器的填充颜色。默认值是“#FFF”（白色）。
- boxStroke：定义图例容器的边框颜色。默认值是“#000”（黑色）。
- boxStrokeWidth：定义图例容器的边框宽度。默认值是 1。
- boxZIndex：定义图例的 z-index。默认值是 100。
- itemSpacing：图例条目之间的空白间隔量。默认值是 10。
- labelFont：标签的字体，默认值是“12px Helvetica”。
- padding：定义图例容器的内补丁。默认值是 5。
- position：定义图例的位置，值可以是 top、bottom、left、right 或 float。如果值是 float，图例会根据 x、y 配置项进行定位。默认值为 bottom。
- visible：布尔值，如果为 false，将不显示图例。默认值为 true。
- x、y：定义图例的左上角坐标，只有 position 值为 float 时才有效。

如果不希望某个图表显示在图例上，可设置该图表的 showInLegend 值为 false。

好了，代码已全部完成，现在在浏览器中打开页面，将看到如图 15-10 所示的效果。这里要注意，**图例中显示标签的文本是由图表的 title 配置项和 yField 配置项决定的，如果定义了 title，优先使用 title 配置项的值**。这样就无需担心字段名称是否与图例显示名称一致，因为可通过 title 配置项定义。单击图例中的标签，可显示或隐藏对应的图表，这是非常好的一个功能。

这里要注意，折线图要显示两条线，需要定义两个 Line 的配置对象，这与面积图和条形图等不同。

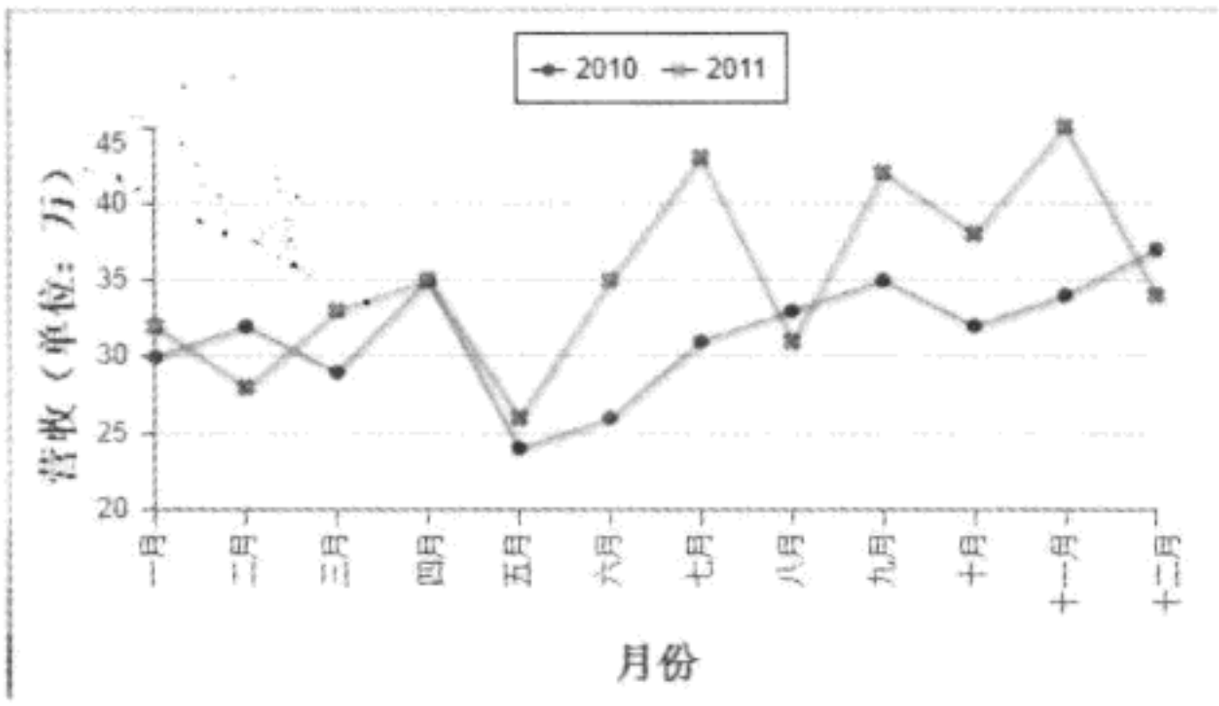


图 15-10　示例的页面效果

15.5.6　使用面积图

面积图通常用于比较一段时间内值的变化情况，例如表 15-2 列出了 2008 年到 2011 年操作系统的市场份额，通过面积图能很好地看到在这段时间内份额的变化。

表 15-2　2008 年到 2011 年操作系统的市场份额

系统 / 年度	XP	Win7	Vista	MacOSX	Linux	其他
2008	75.98	0	16.94	4.16	0.69	2.23
2009	69.57	1.84	22.09	4.27	0.69	1.53
2010	56.11	17.95	18.28	5.82	0.78	1.06
2011	46.21	32.31	13.04	6.41	0.77	1.26

要在面积图中显示表 15-2 的数据，首先要将其转换为符合 Ext JS 图表格式的数据，根据 15.5.1 节 Store 的格式要求，可以得出 Store 的字段包括 Year、XP、Win7、Vista、MacOSX、Linux 和 Other。

数据格式定了后，就可以开始写代码了。使用模板页创建一个名称为 15-11.html 的页面文件，首先加入 Store 的定义，定义时要注意数据类型，不然会发生错误。具体代码如下：

```
Ext.create(Ext.data.ArrayStore,{
```

```
    id:"Store1",
    fields:["Year",{name:"XP",type:"float"},
            {name:"Win7",type:"float"},
            {name:"Vista",type:"float"},
            {name:"MacOSX",type:"float"},
            {name:"Linux",type:"float"},
            {name:"Other",type:"float"}
    ],
    data:[
            ["2008",75.98,0,16.94,4.16,0.69,2.23],
            ["2009",69.57,1.84,22.09,4.27,0.69,1.53],
            ["2010",56.11,17.95,18.28,5.82,0.78,1.06],
            ["2011",46.21,32.31,13.04,6.41,0.77,1.26]
    ]
});
```

最后完成图表，代码如下：

```
Ext.create(Ext.chart.Chart,{
    renderTo:Ext.getBody(),width:500,height:500,
     store: "Store1",
     legend:{position:"top"},
     axes: [{
         type: 'Numeric',
         position: 'left',
         title:"操作系统",
         fields:['XP','Win7','Vista','MacOSX','Linux','Other']
     },
     {
         type: 'Category',
         position: 'bottom',
         fields: ['Year'],
         title:"年度"
    }],
    series: [
           {
            type: 'area',
            axis: 'left',
            xField: 'Year',
            yField: ['XP','Win7','Vista','MacOSX','Linux','Other'],
            title:['XP','Win7','Vista','MacOSX','Linux',' 其他 '],
            highlight:true,
            tips: {
                  trackMouse:true,
                renderer: function(rec,item) {
                  this.update(rec.get("Year")+"年度"+item.storeField
                      +"份额："+rec.get(item.storeField)+"%");
                }
            }
        }
    ]
});
```

定义面积图与折线图最大的不同是，在面积图中，多个图是在同一个 Series 对象内的，而不是分成多个 Series 对象的。图例的标签（title）要与 yField 的字段对应。提示信息要根

据 renderer 函数的第二个参数获取当前是哪个字段的数据，要想了解第二个参数 item 包括什么内容，可使用 consol.log 语句输出到控制台再分析；要注意加上配置项 trackMouse，值为 true，不然会提示错误。

在浏览器中打开页面可看到如图 15-11 所示的效果。面积图中有个小小的错误，就是鼠标移动到 2011 年位置，不能突出显示 2011 年及提示信息。

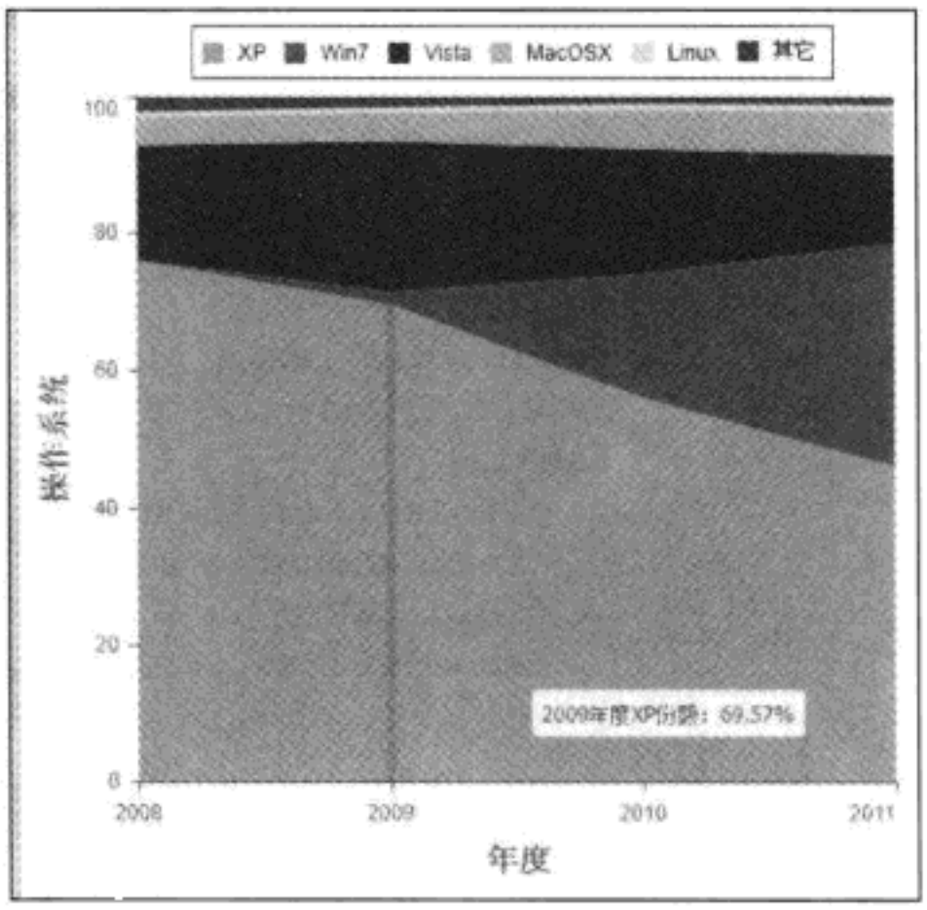


图 15-11　示例的页面效果

15.5.7　简单条形图（Bar 和 ColumnChart）及使用标签（Label 对象）

使用条形图，一定要清楚 Bar 是水平方向的，ColumnChart 才是垂直方向的，因而它们的坐标轴不能放错位置。

在使用条形图前先熟悉一下 Bar 对象的配置项：

- column：布尔值，如果为 true，条形图是垂直方向的。默认值为 false，条形图是水平方向的。**ColumnChart 对象派生于 Bar 对象用于显示垂直的条形图，就是将 column 参数设置为 true 而已。**
- groupGutter：设置两组条形图之间的空白间隔，值为条形图宽度的百分百值。
- gutter：设置两个条形图之间的空白间隔，值为条形图宽度的百分百值。
- stacked：布尔值，如果为 true，将会显示堆积条形图。
- style：条形图的样式，会覆盖主题的样式。
- xPadding：左边或右边坐标轴与条形图之间的间隔。Bar 对象默认值是 0，CloumnChart 对象默认值是 10。
- yPadding：顶部或底部坐标轴与条形图之间的间隔。Bar 对象默认值是 10，CloumnChart 对象默认值是 0。

下面通过示例演示如何使用条形图（水平和垂直）显示表 15-2 中 XP 的市场份额。使用模板页创建一个名称为 15-12.html 的页面文件，先把 15.5.6 节示例的 Store 定义复制过来。

接着定义一个面板，使用 HBoxLayout 将面板区域分成两部分，左边显示水平条形图，右边显示垂直条形图。

```
Ext.create(Ext.panel.Panel,{
    renderTo:Ext.getBody(),width:800,height:500,
    layout:{type:"hbox",align:"stretch"},
    items:[
    ]
});
```

接着定义第一个条形图，要注意坐标位置，代码如下：

```
{xtype:"chart",store: "Store1",flex:1,
    axes: [{
        type: 'Numeric',position: 'bottom',
        title:"操作系统(XP)",fields:['XP'],
        minimum:0,maximum:100
    },
    {
        type: 'Category',position: 'left',
        fields: ['Year'],title:"年度"
    }],
    series: [{
        type: 'bar',axis: 'bottom',
        xField: 'Year',yField: 'XP',
        label:{
           display:"insideEnd",
           'text-anchor':'middle',
           field:"XP",
           renderer:Ext.util.Format.numberRenderer('0.00%')
        }
    }]
},
```

因为条形图是水平方向的，所以数字轴要在底部。在图表定义中，axis 要跟随坐标轴的改变而修改为 bottom。不过，**虽然坐标轴变了，但是 xField 和 yField 还是要根据原来的方式进行定义**。

在图表中多了一个标签（Label 对象）的定义，用来显示条形图的数值。它具体的配置项可参阅 Series 对象中与标签相关的配置项。代码中，标签将显示在条形图内部靠近图尾部的位置；“text-anchor”的作用是确定标签固定的位置，值为 middle 表示固定在中间；使用 renderer 配置项将显示格式变回了百分数。

最后定义垂直显示的条形图，代码如下：

```
{xtype:"chart",store: "Store1",flex:1,
    axes: [{
        type: 'Numeric',position: 'left',
        title:"操作系统(XP)",fields:['XP'],
        minimum:0,maximum:100
    },
    {
        type: 'Category',position: 'bottom',
        fields: ['Year'],title:"年度"
    }],
    series: [{
        type: 'column',axis: 'left',
        xField: 'Year',yFiel: 'XP',
        label:{
           display:"outside",
           'text-anchor':'middle',
           field:"XP",
           renderer:Ext.util.Format.numberRenderer('0.00%')
        }
    }]
}
```

垂直条形图与水平条形图最大的不同是坐标轴要换过来；图表定义中的 axis 也要改为 left；还有标签的显示位置改为了条形图的顶部。

在浏览器中打开页面将看到如图 15-12 所示的效果。

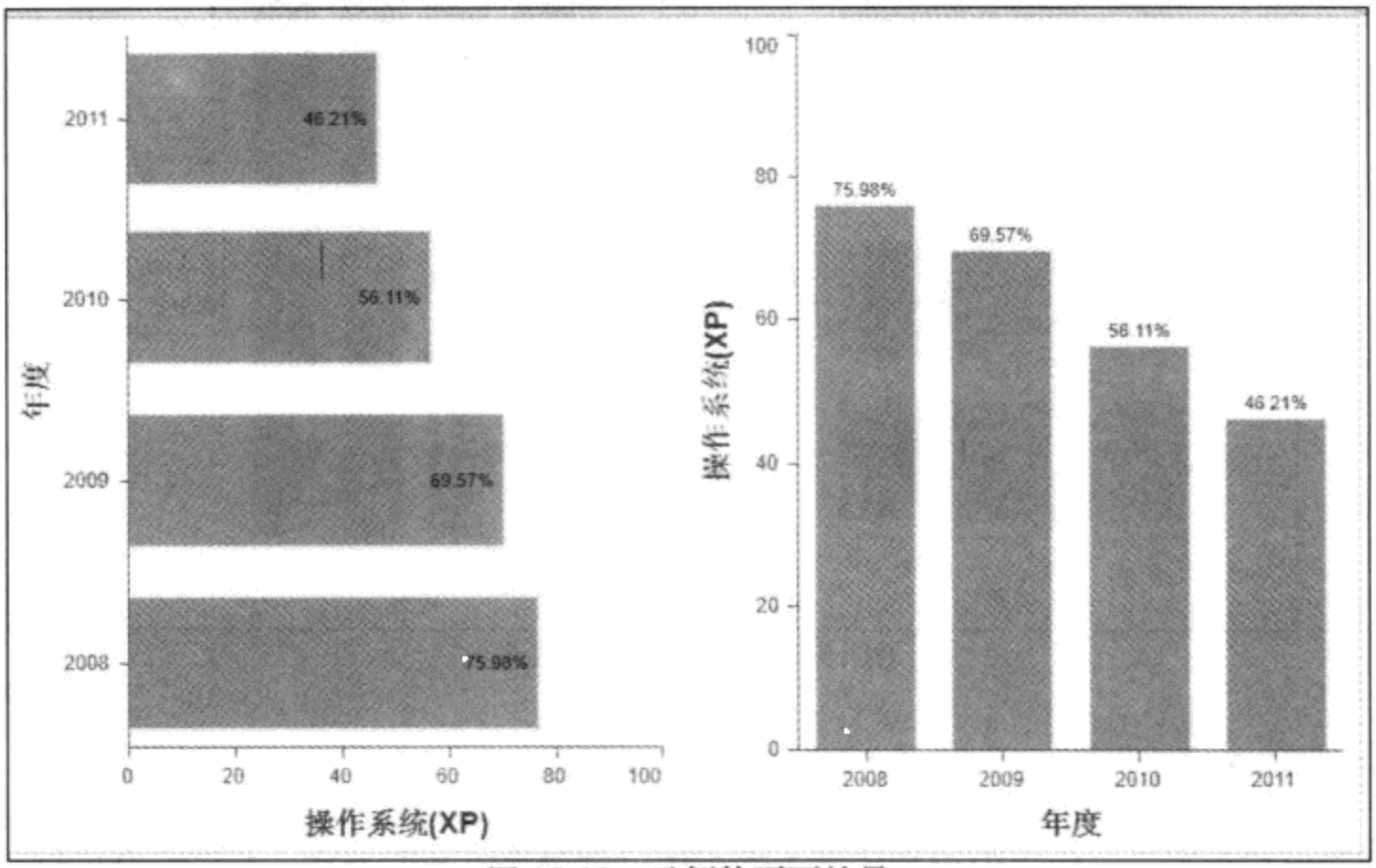


图 15-12　示例的页面效果

15.5.8　堆积条形图

堆积条形图的作用是显示单个项目与整体之间的关系，例如可将各年度操作系统的份额堆积起来显示。

要使用堆积条形图，只要将图表配置项 stacked 定义为 true 就行了。将上一节的示例复制一份，然后将复制后的文件的文件名修改为 15-13.html。首先把数字轴的 fields 配置项和 yField 修改为“['XP','Win7','Vista','MacOSX','Linux','Other']”；然后把数字轴的最大值和最小值配置项删除，因为这里总数是 100，不需要设置了；在两个 Chart 的定义中加入 legend 配置项，图例的位置在右边；在两个条形图中加入配置项 stacked，值为 true；在条形图定义中加入 title，值为“['XP','Win7','Vista','MacOSX','Linux',' 其他 ']”；删除 label 配置项，因为不能显示所有标签，所以只能通过加入提示信息解决。

最后为两个条形图加入提示信息定义，代码如下：

```
tips:{
    trackMouse:true,
    renderer:function(rec,item){this.update(item.value[1]+"%");}
}
```

在页面中打开浏览器将看到如图 15-13 所示的页面效果。图中垂直条形图的底部有些灰

色线条，这个是 Ext JS 的错误，还有待改善。

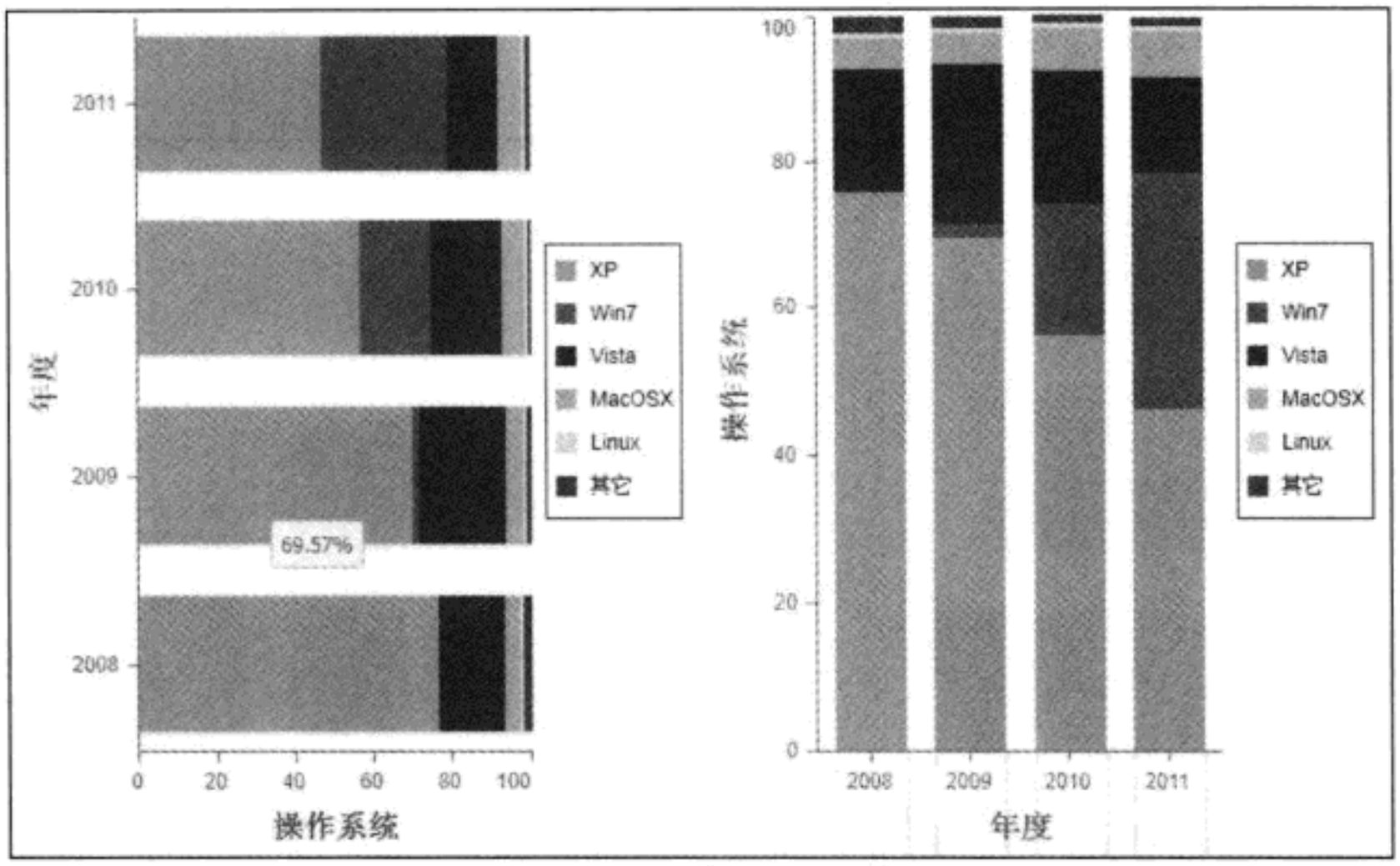


图 15-13　示例的页面效果

15.5.9　分组条形图

如果不是希望像堆积条形图那样显示单个项目与整体之间的关系，而是想做比较，则可使用分组条形图。要使用分组条形图，只要简单地把堆积条形图中的 stacked 删除或设置为 false 就行了。

例如将上一节的示例中的配置项 stacked 设置为 false，将看到如图 15-14 的效果。

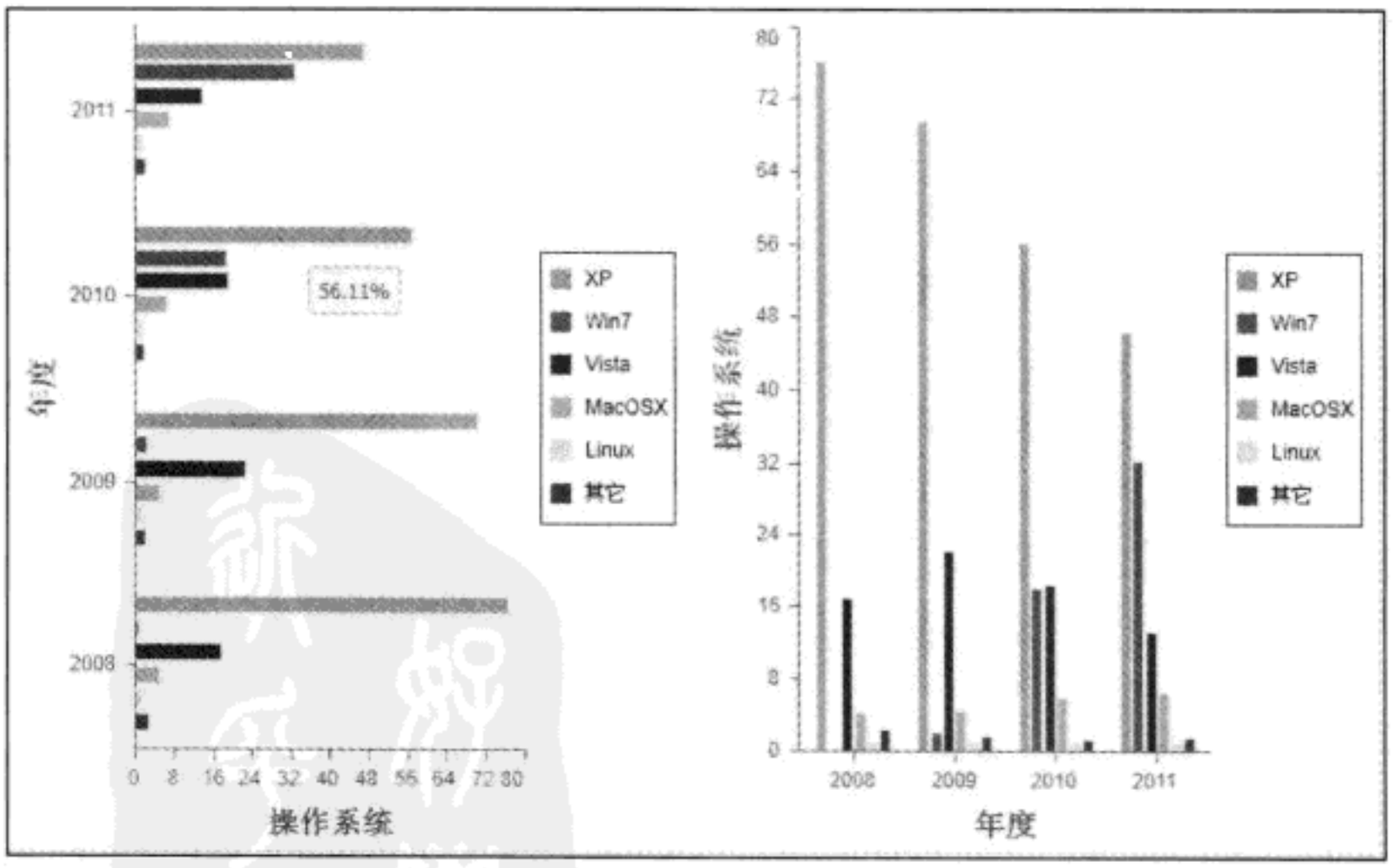


图 15-14　将 15.5.8 节示例中的 stacked 修改为 false 后的页面效果

15.5.10　自定义条形颜色

有时候在使用条形图时，希望当数据小于某个值时就显示为红色作为告警，这时候就需要自定义条形的颜色了。要自定义条形颜色，只需要在 Series 配置对象中定义 renderer 函数就行了，函数可依次接收以下 5 个参数：

- sprite：条形的 DrawSprite 对象。
- record：记录。
- attr：DrawSprite 对象的属性。
- index：索引。
- store：Store 对象。

例如在 15.5.7 节示例中，如果数值小于 50 就显示为红色，可在 Series 配置对象加入以下代码：

```
renderer:function(s,rec,attr,index,store){
    var color= rec.get('XP')<= 50?"#f00":attr.fill;
    return Ext.apply(attr, {fill: color});
}
```

代码中，当值小于或等于 50 时，改为红色，否则保留原来的颜色。

加入以上代码后，2011 年的条形就会显示为红色了。

15.5.11　使用散点图

散点图（Scatter）通常用于显示和比较数值，例如科学数据、统计数据和工程数据等。例如表 15-3 有关温度与雪糕销量的统计数据，通过散点图，就可做温度与雪糕销量的分析。

表 15-3　温度与雪糕销量的统计数据

温度	15	17	18	19	20	21	23	24	26	27	29	30
数量	50	54	58	62	66	70	74	80	86	92	96	100

要使用 Ext JS 将表 15-3 显示为散点图，首先是将其转换为图表认识的格式，这里需要定义 Temperatrue 和 Amount 这 2 个字段。

数据格式定义好以后，可以开始工作了，使用模板页创建一个名称为 15-14.html 的页面文件，然后加入 Store 的定义：

```
Ext.create(Ext.data.ArrayStore,{
    id:"Store1",
    fields:[
            {name:"Temperatrue",type:"int"},
            {name:"Amount",type:"int"}
    ],
    data:[
            [15,50],[17,54],[18,58],[19,62],[20,66],
            [21,70],[23,74],[24,80],[26,86],[27,92],
            [29,96],[30,100]
    ]
});
```

最后定义图表：

```
Ext.create(Ext.chart.Chart,{store: "Store1",
   renderTo:Ext.getBody(),width:500,height:500,
    axes: [{
        type: 'Numeric',position:'left',grid:true,
        title:"销量",fields:['Amount']
    },
    {
        type: 'Numeric',position:'bottom',grid:true,
        fields:"Temperatrue",
        title:"温度",
        label:{
           renderer:Ext.util.Format.numberRenderer('0° ')
        }
    }],
    series: [{
        type:'scatter',axis:'left',
        xField: 'Temperatrue',yField:['Amount'],
        tips:{
           trackMouse:true,
           renderer:function(rec,item){
                  this.update(item.value[0]+"° : "+item.value[1]);
           }
        }
    }]
});
```

因为温度和数量都是数字，所以都使用数字坐标轴。而且为了更好观察，两条坐标轴都加了网格线。温度轴在显示时，特意通过 renderer 函数在数字后加上温度标记。

Series 的定义除了 type 以外，与折线图是基本一样的，可以这样说，把线加上，它就是折线图了。但在点分布不均匀且很多的时候，有线的话会很乱。

在浏览器中打开页面将看到如图 15-15 所示的效果。

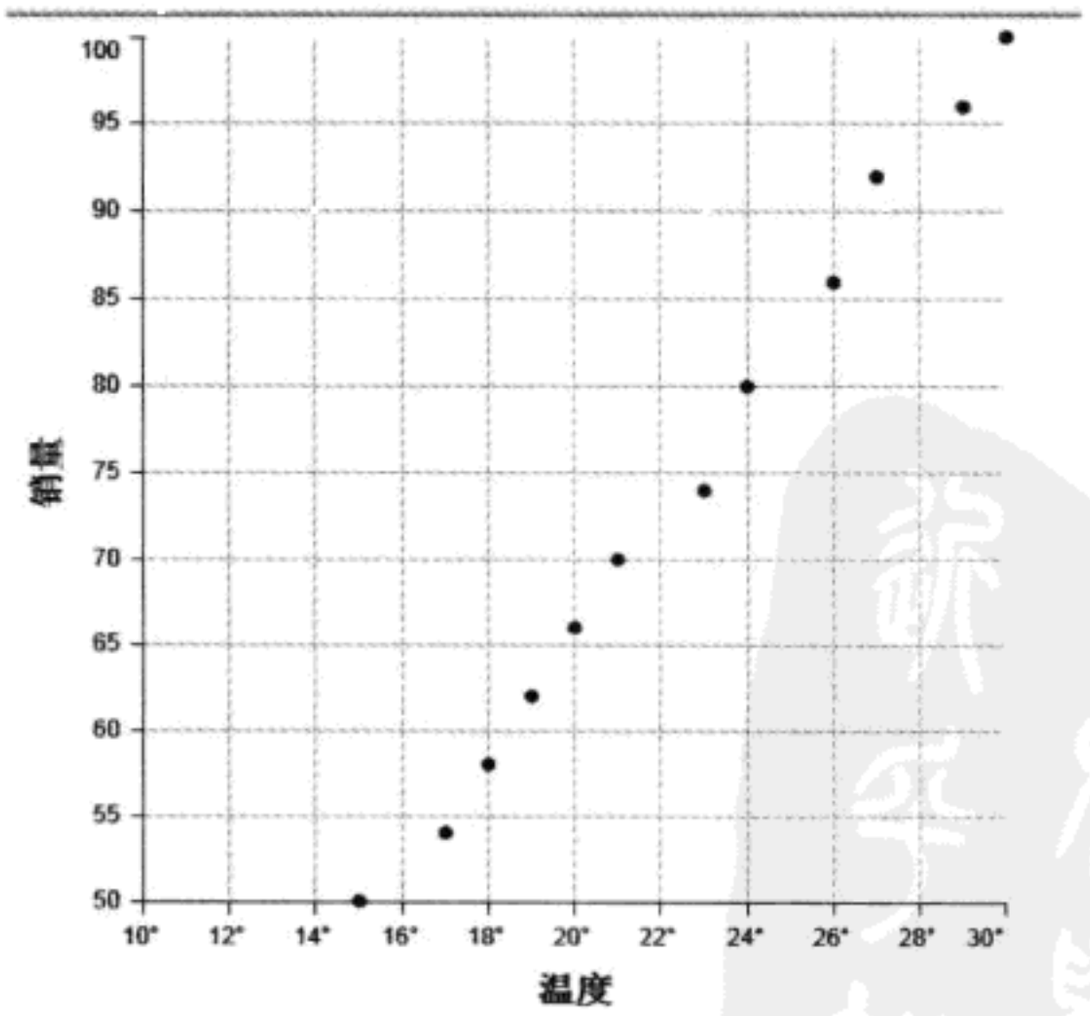


图 15-15　示例的页面效果

如果要多加一组数据，像 15.5.5 那样，添加一个字段，并添加数据。然后把字段名称添加到销量的 fields 数组。最后添加一个 Scatter 对象的配置对象就可以了。在此就不演示，有兴趣可以自己实验一下。

15.5.12　使用饼图

饼图（Pie）通常用于显示百分比，例如显示 15.5.6 节示例中的 2011 年的操作系统市场份额。但是，它的数据格式就不能像 15.5.6 节示例那样定义。这个稍后再说，我们先看看饼图有什么配置项。

饼图主要有以下配置项：

- angleField：设置用于计算饼图角度的字段，该字段数据必须是数字类型。该配置项也可以使用 field 或 xField 代替。
- colorSet：由用于填充每个饼块的颜色组成的数组列表。如果数据很多，则让代码自己生成颜色比较合适。
- donut：布尔值或数字，如果值为 true 或数字，饼图将会变成圆环图。如果是数字，那么该值就是圆环半径占饼图半径的百分比。默认值为 false。
- highlightDuration：设置饼块突出显示效果的持续时间。默认值为 150。
- lengthField：设置饼块长度的字段。如果长度不同，那么全部饼块构成的饼并不是一个圆形。该字段数据必须是数字类型的。
- showInLegend：是否显示图例。默认值为 false，不显示图例。
- style：饼图的样式。

好了，现在回头研究一下饼图的数据格式，直接使用 15.5.6 节示例的数据，情况就是如果指定 angleField 是“XP”字段，那么，饼图显示的将是在 4 年中“XP”每年所占的份额，而不是一年中各操作系统所占的份额。因而，要正确显示一年中各操作系统所占的份额，必须将数据原来的列转换为行，原来的行转换为列。例如，要显示 2011 年的各操作系统的饼图，就要定义 System 和 Share 两个字段，每个记录由系统名称及其所占份额组成。

数据格式定义好以后，可以开始工作了，使用模板页创建一个名称为 15-15.html 的页面文件，然后加入 Store 的定义：

```
Ext.create(Ext.data.ArrayStore,{
    id:"Store1",
    fields:["System",{name:"Share",type:"float"}],
    data:[
            ["XP",46.21],["Win7",32.31],["Vista",13.04],
            ["MacOSX",6.41],["Linux",0.77],[" 其他 ",1.26]
    ]
});
```

现在定义饼图，代码如下：

```
Ext.create(Ext.chart.Chart,{store: "Store1",
    renderTo:Ext.getBody(),width:500,height:500,
    insetPadding:30,
```

```
legend:{position:"right"},
 series: [{
     type:'pie',
     field:'Share',
     showInLegend:true,
     title:["XP","Win7","Vista","MacOSX","Linux","其他"],
     highlight:{
         segment: {margin:20}
     },
     label: {
         field:'System',
         display: 'rotate',
         contrast: true
     },
     tips:{
         trackMouse: true,
         renderer:function(rec,item){
             this.update(rec.get("System") + ': ' + rec.get("Share") + '%');
        }
     }
 }]
});
```

这里有几个关键地方需要注意，首先是饼图没有坐标轴；如果不设置配置项 insetPadding，那么饼图将会使用宽度或高度的最大值作为直径，这样当有突出显示的时候，饼块边缘部分就会被隐藏；如果显示图例，不设置 label，就必须定义 title，不然图例不知道从哪里取标签文本；配置项 contrast 的作用是使标签颜色与饼块的颜色有反差；配置项 segment 的作用是当突出显示时，让饼块偏移中心 20 像素，这样做的主要原因是如果只将其设置为 true 来突出显示，效果实在太差。

示例存在的问题是，很难设置饼块的标签，主要是最小的两个饼块的显示问题，因而只好使用提示信息和图例搭救。

在浏览器中打开页面将看到如图 15-16 所示的效果。

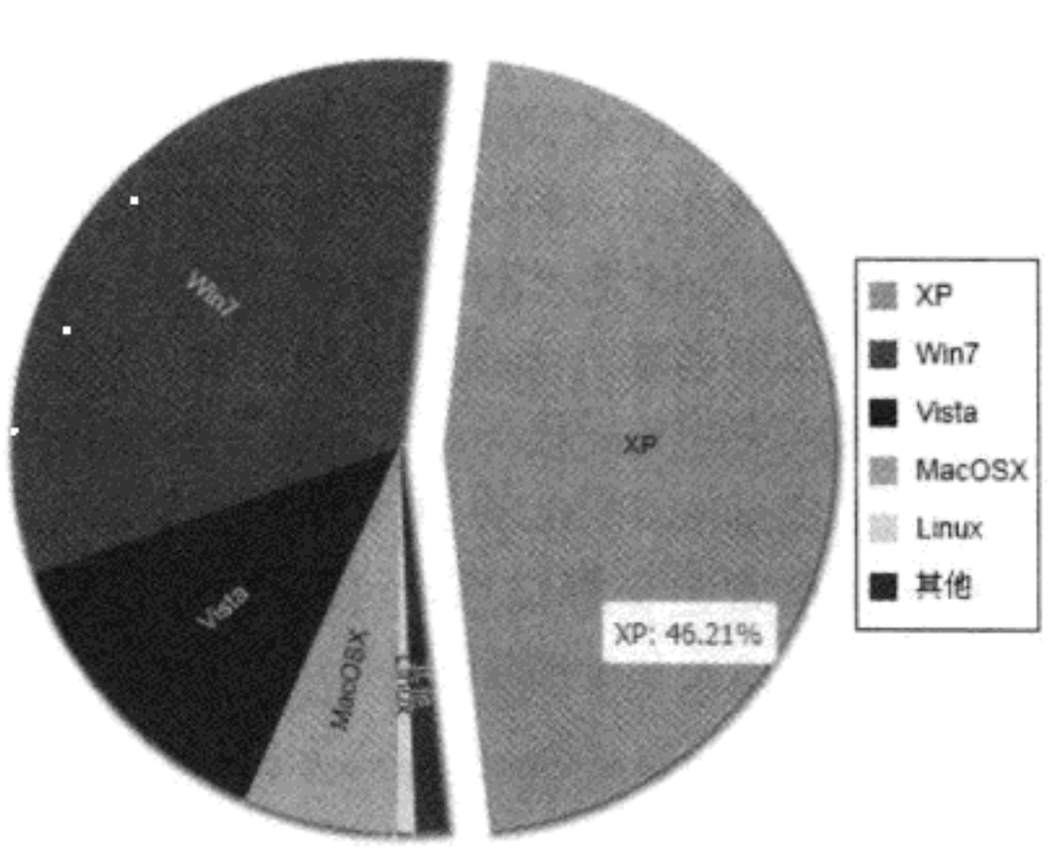


图 15-16　示例的页面效果

15.5.13　自定义饼块颜色

饼块的颜色可使用 15.5.10 节自定义条形颜色的方法进行自定义，在此就不做示例了，自己可以试验一下。

15.5.14　使用表盘图

表盘图（Gauge）有点类似进度条，因而只需要一个值，该值可由 Store 提供，也可以使

用 setValue 方法更新。不过，因为定义 Chart 必须定义 Store，而且使用 setValue 方法也还存在问题，所以不建议使用。图表的表示方式有两种，一种是通过不同颜色区分。先看看表盘图有什么配置项，以下是其配置项：

- angleField：设置用于计算表盘图角度的字段，该字段数据必须是数字类型的。该配置项也可以使用 field 或 xField 代替。
- colorSet：设置表盘的颜色，只设置两种颜色就行了。
- donut：布尔值或数字，如果值为 true，表盘图将会变成圆环表盘图。如果是数字，那么该值就是圆环半径占表盘图半径的百分比。默认值为 false。
- highlightDuration：设置表盘图突出显示效果的持续时间。默认值为 150。
- needle：设置表盘的指针。
- showInLegend：是否显示图例。默认值为 false，不显示图例。
- sytle：表盘图的样式。

下面通过示例演示如何通过两种方式显示表盘图，以及动态更新表盘图。使用模板创建一个名称为 15-16.html 的页面文件，先定义一个 Store：

```
Ext.create(Ext.data.ArrayStore,{
    id:"Store1",
    fields:[{name:"Value",type:"float"}],
    data:[[47]]
});
```

然后定义好图表：

```
Ext.create(Ext.panel.Panel,{
    renderTo:Ext.getBody(),width:800,height:300,
    layout:{type:"hbox",align:"stretch"},
    items:[
            {xtype:"chart",store: "Store1",flex:1,
                    insetPadding:50,
             axes: [{
                 type: 'gauge',position: 'gauge',
                 minimum:0,maximum:100,margin:-10
             }],
             series: [{
                type: 'gauge',field:"Value",needle:true
             }]
            },
            {xtype:"chart",store:"Store1",flex:1,
                    insetPadding:50,
             axes: [{
                 type:'gauge',position:'gauge',
                 minimum:0,maximum:100,margin:10,
             }],
             series: [{
                type: 'gauge',donut:50,field:"Value",
             }]
            }
    ]
});
```

和饼图一样，需要定义 insetPadding 控制表盘的半径。第一个表盘图使用指针指示数值，第二个则使用颜色区块指示数值。在坐标轴的定义中，margin 可控制数字的显示位置，如果是负数，会显示在表盘内，默认是显示在表盘外，没有刻度条。

现在要定义一个任务，定时刷新数值，代码如下：

```
var taskManager=new Ext.util.TaskRunner();
var task = {
   run:function(){
            var v=Math.floor(Math.random()*100),
                store=Ext.StoreManager.lookup("Store1");
          store.getAt(0).set("Value",v);
   },
   interval:3000
};
```

任务内，随机生成一个 0 到 99 的数字，然后使用 set 方法修改记录的值。这里一定要用 set 方法，千万别直接使用 data 属性去修改，因为只有 set 方法才会触发 refresh 事件。记录只有一个，所以使用 getAt 返回第一个记录就行了。

再在面板的顶部工具栏添加一个按钮用来触发任务：

```
tbar:[
   {text:"开始",handler:function(){
            if(this.text=="开始"){
                  this.setText("停止");
                  taskManager.start(task);

            }else{
                  this.setText("开始");
                  taskManager.stop(task);
            }
   }}
],
```

至此，示例就完成了，在浏览器中打开将看到如图 15-17 所示的效果。单击“开始”按钮，会看到表盘指针或色块会 3 秒变化一次。

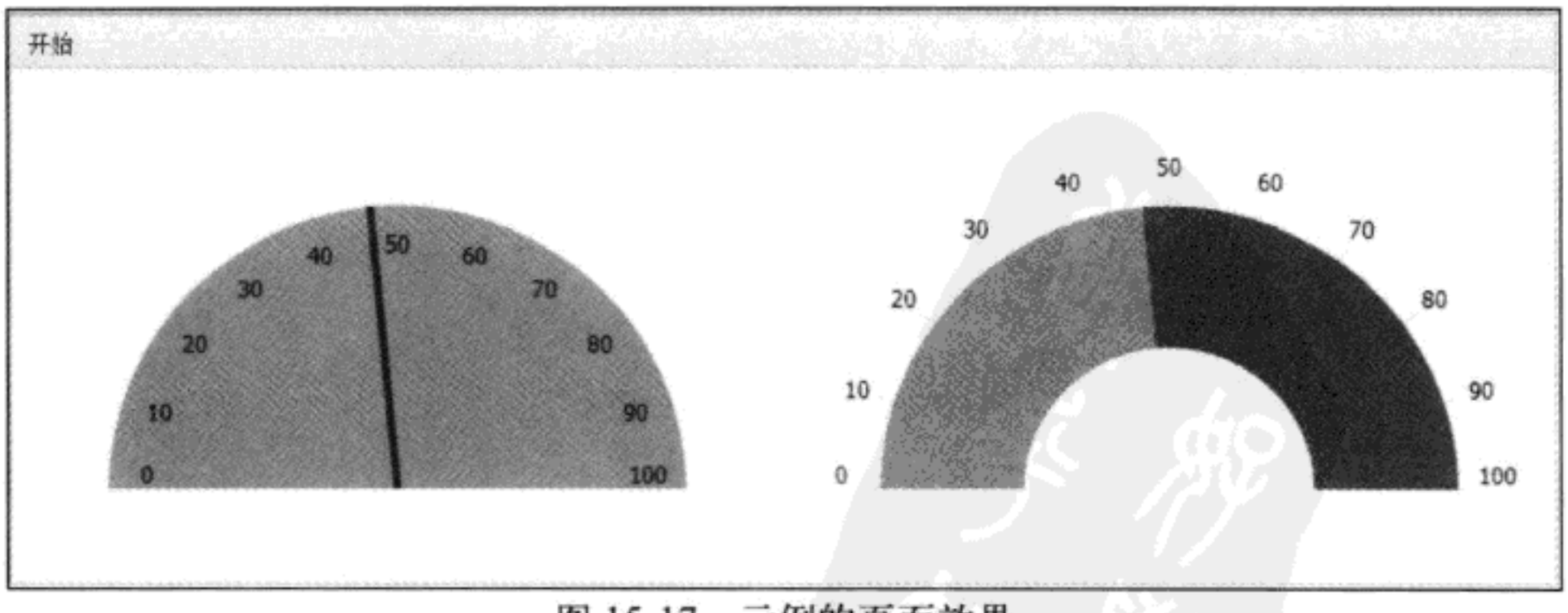


图 15-17　示例的页面效果

标题图存在的问题是不能显示标签和提示信息，因而无法标识当前值是什么。这个要通

过其他办法来解决了。

15.5.15　使用雷达图

雷达图（Radar）的作用主要用于数据比较，例如游戏中，经常使用雷达图来比较人物、装备之间的性能参数差别。

雷达图与折线图有点类似，只是由纵横坐标换成了圆形的雷达坐标。因而其使用方法除了坐标轴使用 Radial 对象外，与折线图的数据格式和定义基本一致。例如要在雷达图中显示和比较表 15-4 的游戏人物数据，首先要定义好数据格式。根据表格，一个人物就是一个字段，而每个字段有 5 个数据，可以定义字段为：Ability、GuanYu、ZhugeLiang、CaoCao、ZhaoYun 和 HuangZHong。

表 15-4　游戏人物数据

能力 / 姓名	统率	武力	智力	政治	魅力
关羽	95	97	75	62	93
诸葛亮	92	38	100	95	92
曹操	96	72	91	94	96
赵云	91	96	76	65	81
黄忠	86	93	60	52	75

数据格式确定后，就可以创建演示页面了，使用模板页创建一个名称为 15-17.html 的页面文件，先定义好 Store，代码如下：

```
Ext.create(Ext.data.ArrayStore,{
    id:"Store1",
    fields:["Ability",
            {name:"GuanYu",type:"int"},{name:"ZhugeLiang",type:"int"},
            {name:"CaoCao",type:"int"},{name:"ZhaoYun",type:"int"},
            {name:"HuangZHong",type:"int"}
    ],
    data:[
            ["统率",95,92,96,91,86],["武力",97,38,72,96,93],
            ["智力",75,100,91,76,60],["政治",62,95,94,65,52],
            ["魅力",93,92,96,81,75]
    ]
});
```

接着定义图表，代码如下：

```
Ext.create(Ext.chart.Chart,{
    renderTo:Ext.getBody(),width:500,height:500,
    insetPadding:50,store:"Store1",
    legend:{position:"top"},
    axes:[{
            type: 'Radial',
            position: 'radial',
            label:{display:true}
```

```
    }],
     series: [{
         type: 'radar',xField: 'Ability',yField: 'GuanYu',
         title:"关羽",showInLegend: true,
         style: {opacity:0.2}
     },{
         type: 'radar',xField: 'Ability',yField: 'ZhugeLiang',
         title:"诸葛亮",showInLegend: true,
         style: {opacity:0.2}
     },{
         type: 'radar',xField: 'Ability',yField: 'CaoCao',
         title:"曹操",showInLegend: true,
         style: {opacity:0.2}
     },{
         type: 'radar',xField: 'Ability',yField: 'ZhaoYun',
         title:"赵云",showInLegend: true,
         style: {opacity:0.2}
     },{
         type: 'radar',xField: 'Ability',yField: 'HuangZHong',
         title:"黄忠",showInLegend: true,
         style: {opacity:0.2}
     }]
});
```

与折线图对比，会发现定义坐标轴时不需要定义字段，而定义每个图时不需要 axis 配置项。其余定义基本相同。

在浏览器中打开页面将看到如图 15-18 的效果。从图可以看到，使用填充方式，图多的时候比较混乱。如果不需要填充，可在 style 配置项中添加 fill，值为 none。不过这时候要定义连线的颜色和宽度，不然会看不到连线。如果要显示大的点，添加 showMarkers 配置项，显示标记，如果想自定义标记，则添加 markerConfig 配置项。

雷达图存在个问题，要使用提示信息显示当前点的数值，而 renderer 函数返回的参数根本确定不了点的数值。

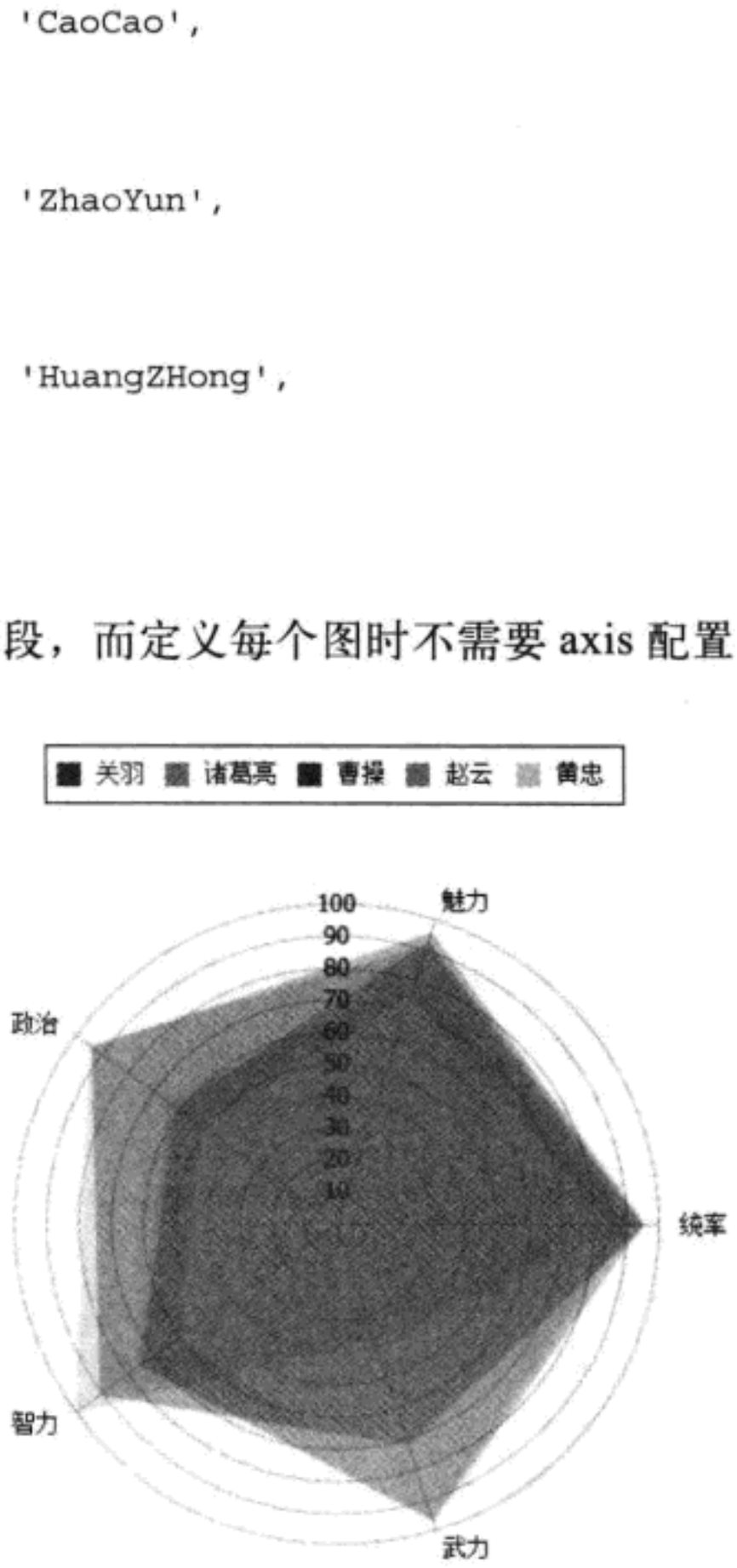


图 15-18　示例的页面效果

15.5.16　使用时间轴

时间轴（TimeAxis）目前只能支持到天数这个级别，对于显示实时性比较强的时间，如要到分钟级别的则不适合。**使用中一定要注意，在 fromDate 和 toDate 指定的时间区间内，必须每一个日期都有数据，例如区间是 2011-8-1 到 2011-8-7，间隔是天，那么在 Store 中的数据必须有这 7 天的数据，不能缺少任何一天。反过来，如果 Store 中的数据是日期是从 2011-8-1 到 2011-8-20，那么 fromDate 和 toDate 设置的日期区间必须在 Sotre 的数据范围内，例如，toDate 就不能设置为 2011-8-31，不然会出错。**

下面通过一个示例演示如何使用时间轴显示从 2011-8-1 到 2011-8-7 的温度值。使用模板页创建一个名称为 15-18.html 的页面文件，然后定义 Store，代码如下：

```
Ext.create(Ext.data.ArrayStore,{
    id:"Store1",
    fields:[
            {name:"Date",type:"date"},{name:"Temperatrue",type:"int"}

    ],
    data:[[new Date(2011,8,1),31],[new Date(2011,8,2),32],
          [new Date(2011,8,3),34],[new Date(2011,8,4),36],
          [new Date(2011,8,5),33],[new Date(2011,8,6),32],
          [new Date(2011,8,7),28]
    ]
});
```

要注意，数据中日期必须是不间断的，如果是服务器端返回的数据，要注意把没有的日期补充完整，如果觉得麻烦，就不要用时间轴。

最后定义图表，代码如下：

```
Ext.create(Ext.chart.Chart,{
    renderTo:Ext.getBody(),width:500,height:500,
    store:"Store1",insetPadding:20,
    axes:[{
            type: 'Numeric',grid: true,position: 'left',
          fields: ['Temperatrue'],title: '温度',
          minimum:0,maximum:40
    },
    {
            type: 'Time',
            position: 'bottom',
            fields: 'Date',
            title: '8 月份',
            dateFormat: 'd',
            constrain: true,
            fromDate: new Date(2011,8,1),
            toDate: new Date(2011,8,7)
    }],
    series: [{
            type: 'line',axis: ['left',"bottom"],
            xField: 'Date',yField: 'Temperatrue'
    }]
});
```

注意 fromDate 和 toDate 的定义，值必须是在 Store 中存在的值。

在浏览器中打开页面，将看到如图 15-19 所示的效果。

15.5.17　实现实时动态的图表

在监控应用中，很多时候会使用图表实时显示监控数据，要实现该功能只要更新 Store 的数据就行，可以用 Store 的 add、load 或 loadData 等操作方法。下面通过一个示例演示如何显示监控数据，模拟的是从早上 10 点开始每分钟对温度的监控，为了方便演示，数据将会 1 秒

更新一次。

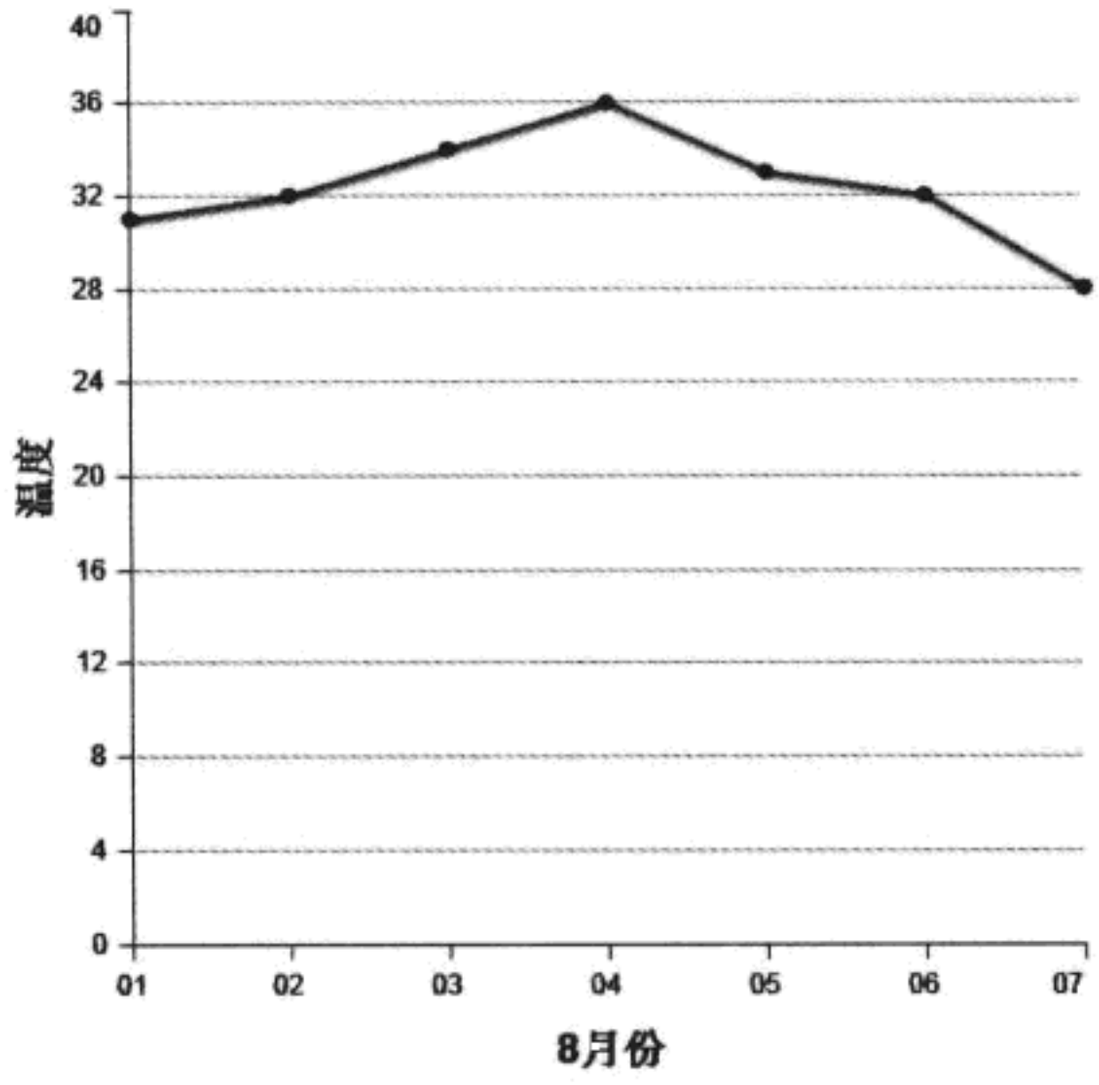


图 15-19　示例的页面效果

使用模板页创建一个名称为 15-19.html 的页面文件，首先定义一个当前时间来记录开始时间：

```
var currentTime=new Date(2011,8,1,10,0,0);
```

接着定义一个 Store：

```
Ext.create(Ext.data.ArrayStore,{
    id:"Store1",
    fields:[
            "Date",{name:"Temperatrue",type:"int"}

    ],
    data:[          [Ext.Date.format(Ext.Date.add(currentTime,Ext.Date.MINUTE,-
                        1),"H:i"),0],
    [Ext.Date.format(Ext.Date.add(currentTime,Ext.Date.MINUTE,-2),"H:i"),0]
    ]
});
```

在定义 Store 时要注意，如果开始时没有数据，横轴将不会有显示。因而为了在初始化到时候能看到横轴，必须先预置**两个**数据，这里不能只预置一个数据，因为是根据最小值和最大值画坐标轴的，只有一个数据，最大值就为 undefined 了，会出现问题。这里直接将日期转换为了时间的字符串，目的是为了方便显示，因为时间轴不支持时间间隔，只能用 Category 代替，将日期转换为字符串后，就可以直接输出，不用定义 label 的 renderer 函数重置坐标轴的标签文本。

接着定义图表。为了添加开始按钮，需要将图表放到面板内，具体代码如下：

```
Ext.create(Ext.panel.Panel,{
    renderTo:Ext.getBody(),width:800,height:300,
    layout:"fit",
    tbar:[
            {text:"开始",handler:function(){
            }}
    ],
    items:[{xtype:"chart",store:"Store1",insetPadding:20,
            axes:[{
            type: 'Numeric',grid: true,position: 'left',
              fields: ['Temperatrue'],title: '温度',
                minimum:0,maximum:40
           },
           {type: 'Category',position:'bottom',
              fields:'Date',title:'8月1月'
           }],
           series: [{
              type: 'line',
              axis: 'left',
              xField: 'Date',
              yField: 'Temperatrue'
              }]
      }]
});
```

图表定义没什么特殊配置项，按正常定义就行了。

接着从 15.5.4 节示例中将 taskManager、task 和“开始”按钮的句柄复制过来。最后将 task 的代码修改成以下代码：

```
var task = {
    run:function(){
            var v=Math.floor(Math.random()*40),
                    store=Ext.StoreManager.lookup("Store1");
            if(store.count()>20){
                    store.remove(store.getAt(0));
            }
            store.add({"Date":
    Ext.Date.format(Ext.Date.add(currentTime,Ext.Date.MINUTE,this.taskRunCount-
        1),"H:i")
            ,Temperatrue:v});
    },
    interval:1000
};
```

任务 task 将 1 秒触发一次，每触发一次，就产生一个随机的温度数据；如果 Store 的记录超过了 20 个，就需要删除索引为 0 的记录，不然图表随着数据的不断增加，刻度会越来越少，显示的点就会粘连在一起了。至于保持多少个数据，可根据自己需要设置，这里保留 20 个数据；通过 Store 的 add 方法为 Store 添加一个记录，时间为当前时间的分钟加上任务的执行次数。

在浏览器中打开页面，然后单击“开始”，等待 30 秒，单击“暂停”，将看到如图 15-20 的页面效果。

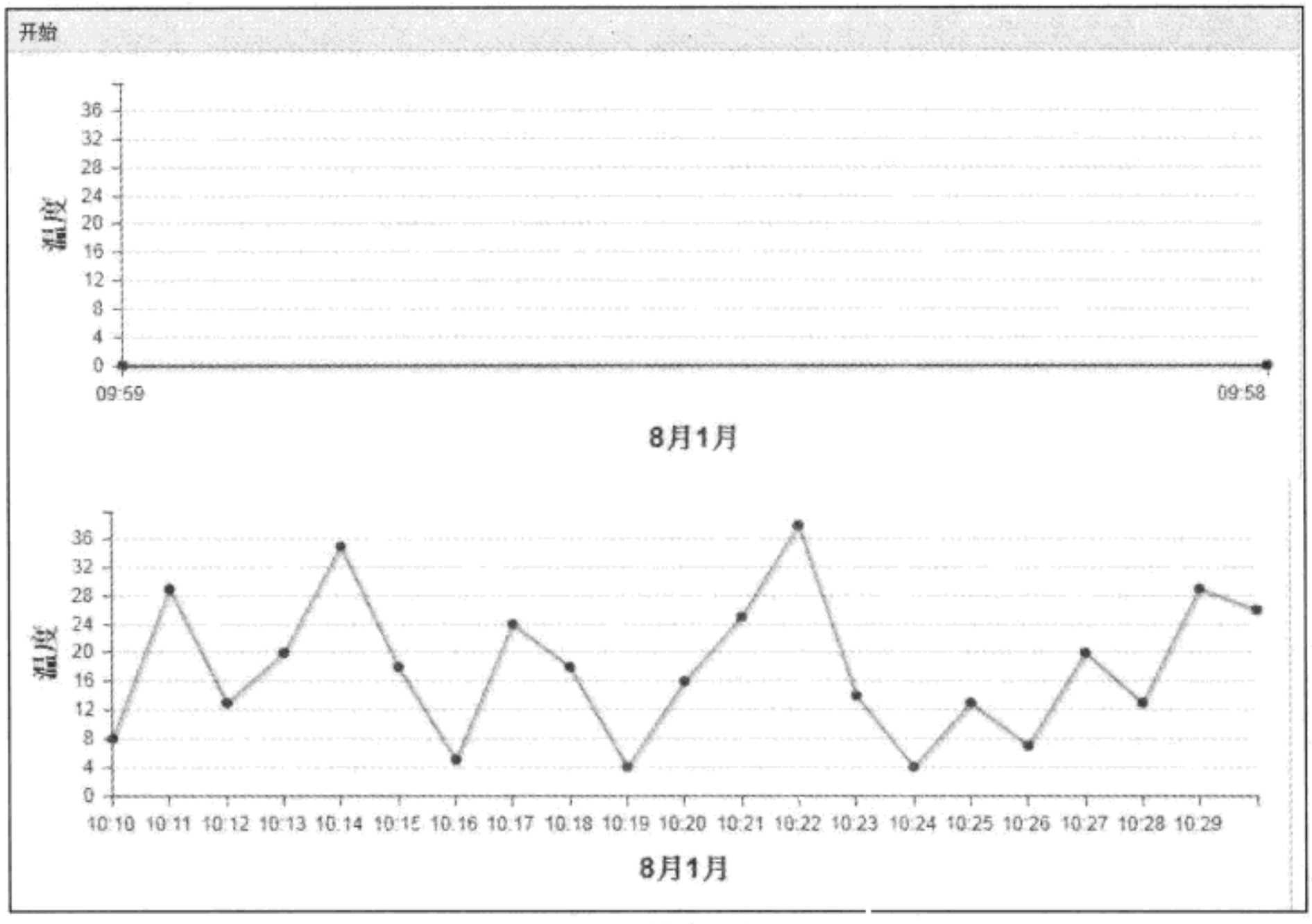


图 15-20　示例的页面效果

15.5.18　使用组合图

组合图使用了两种或两种以上的图表类型，用来强调图表中含有不同类型的信息，例如在图表中使用条形图显示计划数据，而使用折线图显示实际数据。现在尝试一下在 15.5.1 节示例的基础上，使用条形图显示计划数据。

使用模板页创建一个名称为 15-20.html 的页面文件，然后将 15.5.1 节示例的代码复制过来。首先修改 Store，添加名称为 Plan 的字段用来存放计划数据，其类型为 int。然后在数据中添加计划数据。

接着在图表定义中添加图例显示：

```
legend:{position:"top"},
```

在数字轴的 fields 配置项中添加 Plan 字段，把 grid 配置项内的 even 配置项注释掉。

在折线图前加入条形图，代码如下：

```
{
    type:"column",axix:"left",
    xField:"Month",yField:"Plan",
    title:"计划"
}
```

如果折线图在前，条形图在后，则条形图会遮盖住折线图的点，因而图表的先后顺序一定要注意。

注释掉折线图的 fill 和 smooth 配置项，并添加 title 配置项，值为“实际”。
在浏览器中打开页面将看到如图 15-21 所示的效果。

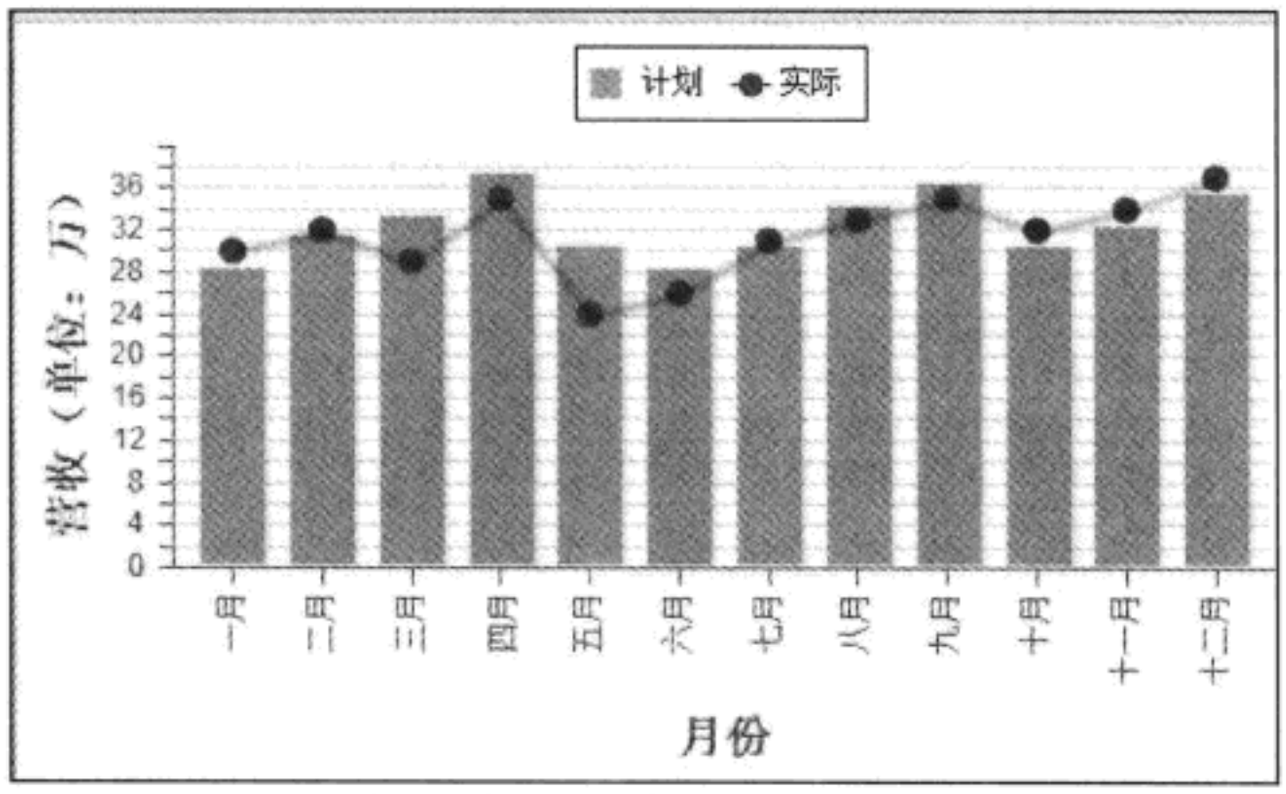


图 15-21　示例的页面效果

15.5.19　在图表中使用背景

在图表中使用背景，可在定义图表的时候添加 background 配置项，该配置项的值可以为 true，使用主题预设的背景颜色，也可以为对象，在对象内使用 fill 配置项设置填充颜色，也可使用 image 配置项设置背景图片，还可以使用 gradient 配置项设置渐变效果。

下面通过一个示例演示如何自定义图表背景，使用模板页创建一个名称为 15-21.html 的页面文件，然后添加以下 Store 的定义：

```
Ext.create(Ext.data.ArrayStore,{
    id:"Store1",
    fields:["Year",{name:"Revenue",type:"int"}
    ],
    data:[["2009",300],["2010",320],["2011",350]]
});
```

接着定义一个面板，使用水平布局将面板分成 3 份，分别用于 fill、image 和 gradient 配置项的背景，具体代码如下：

```
Ext.create(Ext.panel.Panel,{
    renderTo:Ext.getBody(),width:800,height:300,
    layout:{type:"hbox",align:"stretch"},
    items:[
   {xtype:"chart",store: "Store1",flex:1,
        background:{fill:"#def"},
    axes:[{
        type: 'Numeric',position: 'left',
        fields:['Revenue'],title:"营收（单位：万）",
        grid:true
    },{
        type: 'Category',position: 'bottom',
        fields: ['Year'],title:"年度"
```

```
                }],
                series: [{
                        type:"line",axix:"left",
                        xField:"Month",yField:"Revenue"
    }]
},
    {xtype:"chart",store: "Store1",flex:1,
            background:{image:"../images/bg.jpg"},
     axes: [{
         type: 'Numeric',position: 'left',
         fields:['Revenue'],title:"营收（单位：万）",
         grid:true,label:{color:"#00f",fill:"#00f"},
         labelTitle:{color:"#00f",fill:"#00f"}
     },{
            type: 'Category',position: 'bottom',
            fields: ['Year'],title:"年度",
            label:{color:"#00f",fill:"#00f"},
            labelTitle:{color:"#00f",fill:"#00f"}
            }],
            series: [{
                    type:"line",axix:"left",
                    xField:"Month",yField:"Revenue"
    }]
},
    {xtype:"chart",store: "Store1",flex:1,
    background: {
        gradient: {
            id: 'gradientId',
            angle: 90,
            stops: {
                0: {color:'#fff'},
                100: {color:'#def'}
            }
        }
    },
     axes: [{
         type: 'Numeric',position: 'left',
         fields:['Revenue'],title:"营收（单位：万）",
         grid:true
     },{
            type: 'Category',position: 'bottom',
            fields: ['Year'],title:"年度"
            }],
            series: [{
                    type:"line",axix:"left",
                    xField:"Month",yField:"Revenue"
    }]
}
    ]
});
```

第一个图表使用颜色“def”为填充颜色。第二图表使用图片“bg.jpg”为背景图片，因为背景会造成标签黑色字体看不清楚，因而需要在坐标轴内定义配置项 label 和 labelTitle 来改变字体颜色。其中 label 是用来修改刻度上的标签文字的，labelTitle 是用来设置 title 配置项定义的标题颜色的。第三个图表则使用了从顶部到底部的渐变效果作为背景。

在浏览器中打开页面将看到如图 15-22 所示的效果（VML 图使用图片背景有问题）。

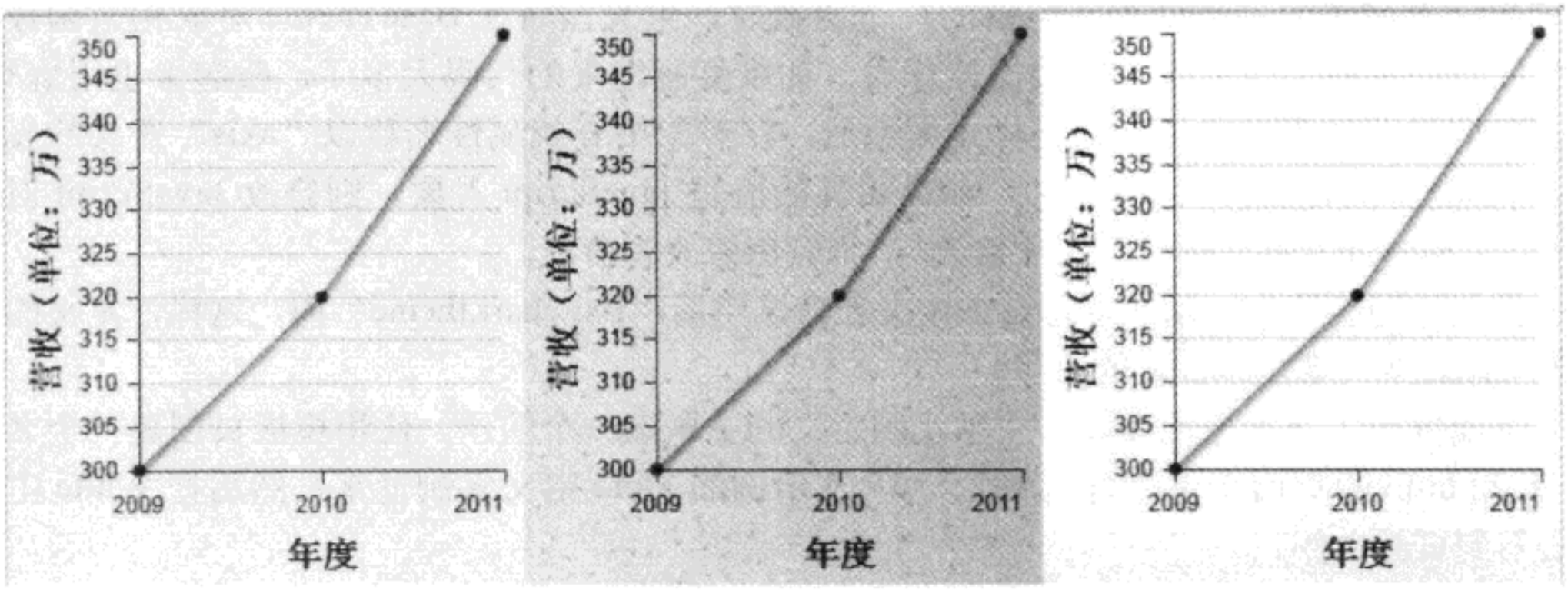


图 15-22　示例的页面效果

注意　示例在 4.0.7 下运行正常，而在 4.1 Beta 1 中存在错误，不能显示背景。

15.5.20　在图表中自定义主题

图表的主题可以控制形状、颜色、字体、坐标轴的样式及图表背景颜色，具体的配置项在 BaseTheme 对象的构造函数内，只要扩展 BaseTheme 对象，复写样式定义。例如，要将图表背景修改为黑色的，坐标轴和标签及图表的标签都为白色的，可这样扩展：

```
Ext.define('Ext.chart.theme.White', {
    extend: 'Ext.chart.theme.Base',

    baseColor:"#fff",

     constructor: function(config) {
            var me=this;
          me.callParent([Ext.apply({
            //可加重写样式
            background:{fill:"#000"},
              seriesLabel: {
                  font: '12px Arial',
                  fill: '#fff'
              }
          }, config)]);
          for(var key in me){
            if(me.hasOwnProperty(key) && key.substr(0,4)=="axis"){
              var o=me[key];
              if(Ext.isObject(o)){
                    if(o.hasOwnProperty("fill"))o["fill"]=me.baseColor;
                    if(o.hasOwnProperty("stroke"))o["stroke"]=me.baseColor;
              }
            }
          }
    }
});
```

在以上代码中，在调用父对象的构造函数时，可直接将在 BaseTheme 对象构造函数内定义的样式复制到注释下，直接修改样式。如果复制过来的代码太多了，改起来也很麻烦，可以在调用父对象构造函数绑定样式属性后，在对象中查找属性名称以“axis”开头的属性（这些都是与坐标轴有关的样式），如果属性的值是 JavaScript 对象，则修改 JavaScript 对象内的 fill 和 stroke 配置项（存在才修改），将颜色修改为白色。

扩展主题要注意主题的名称必须定义在命名空间“Ext.chart.theme”内，这样，定义的名称就可以作为配置项 theme 的值了。

下面做个示例来把预设好的主题及刚定义的主题做一个演示。使用模板页创建一个名称为 15-22.html 的页面文件。首先将 15.5.19 节示例的 Store 定义复制过来，然后把 White 主题扩展写到页面文件。

现在定义一个标签页，代码如下：

```
Ext.create(Ext.tab.Panel,{
    renderTo:Ext.getBody(),width:500,height:500,
    items:[
     ]
});
```

现在在 items 中加入一个标签页，显示自定义模板的图表，代码如下：

```
{layout:"fit",title:" 自定义主题 ",items:[
    {xtype:"chart",store: "Store1",
            theme:"White",
          axes:[{
           type: 'Numeric',position: 'left',
           fields:['Revenue'],title:" 营收（单位：万）",
           grid:true,minimum:200,maximum:400
       },{
              type: 'Category',position: 'bottom',
              fields: ['Year'],title:" 年度 "
     }],
     series: [{
              type:"column",axix:"left",
              xField:"Year",yField:"Revenue",
      label:{
              display:"outside",
              'text-anchor':'middle',
              field:"Revenue"
      }
     },{
              type:"line",axix:"left",fill:true,
              xField:"Year",yField:"Revenue"
             }]
    }
]},
```

图表中显示了两个图，一是条形图，一个是折线图，折线图还要显示填充，目的是展示出基本的主题。我们的重点是配置项 theme，使用的是自定义的主题 White。

直接复制 13 份标签，将它们的标签页标题（title）和 theme 配置项修改为 Base、Green、

Sky、Red、Purple、Blue、Yellow 以及 Category1 至 Category6。这样，就把自定义的主题和默认的 13 个主题都编写完成了。

在浏览器中打开页面将看到如图 15-23 所示的效果。自定义的主题只修改了坐标轴及其标签的颜色、图表标签的颜色，因而条形图和折线的颜色没有改变。

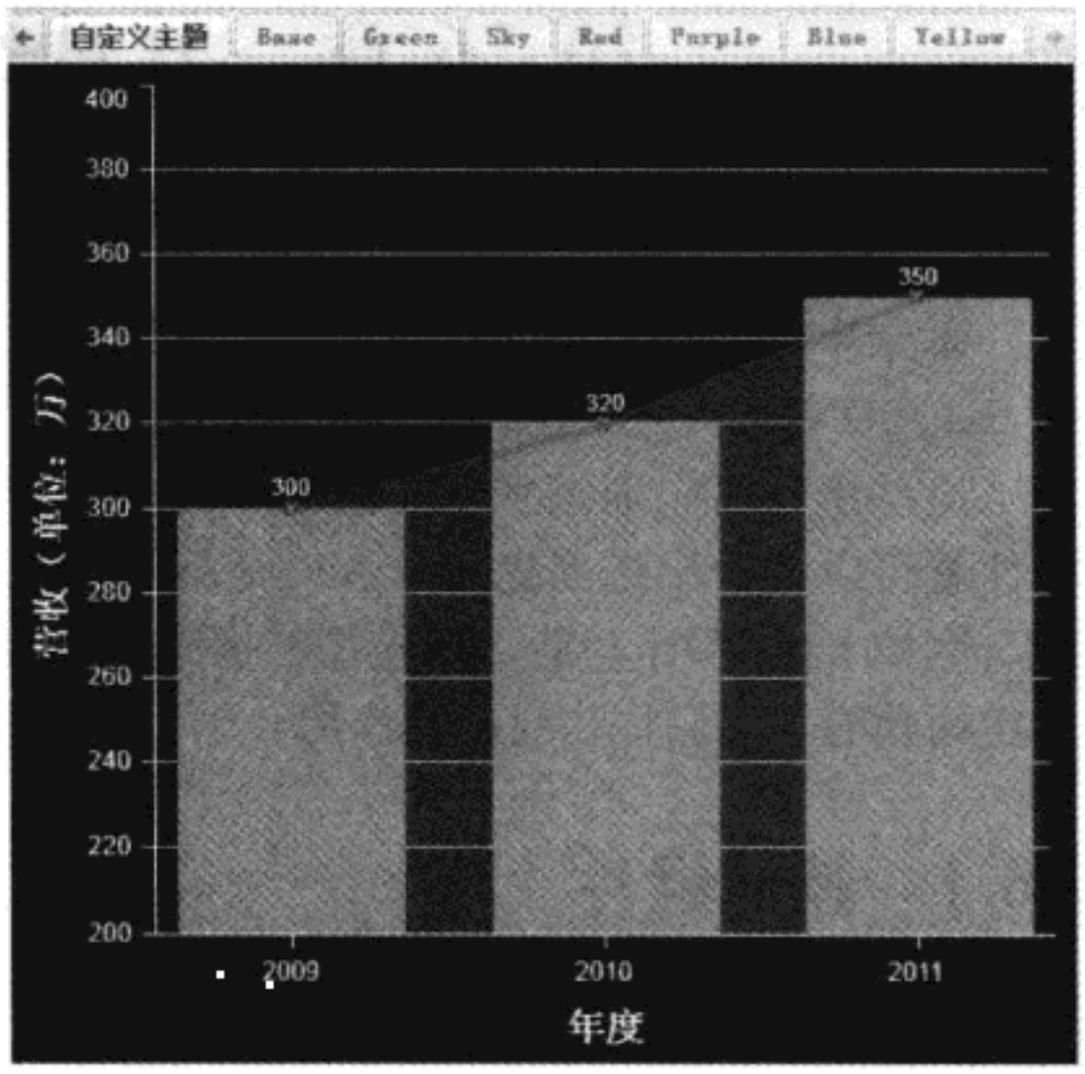


图 15-23　示例的页面效果

注意　因 4.1 Beta 1 版本中设置背景存在错误，不能正确显示背景，因而本示例自定义主题显示存在问题，文字是改成了白色了，但是背景没有修改为黑色。

15.6　本章小结

利用浏览器支持的图形功能在框架中实现画图功能和图表功能，是很有创意的想法，而且实现得不错，应该对 Ext JS 团队赞一个！随着 HTML 5 的画布功能的发展，估计未来实现起来会更简单，可以有更多的创意，期待……

本章的重点是图表的使用，而使用图表的重点是数据格式。很多时候，使用图表发生问题往往是在数据格式上，因而，如果要经常使用图表，必须对各种图表的数据格式好好研究一番。

第 16 章　其他组件及实用功能

本章主要介绍在前面章节中还没有讲到的组件及实用功能，如进度条、图片组件等组件及 Ext.JSON 等功能函数。

这些组件可以通过重新构造或与其他组件组合在一起成为新的组件，例如进度条和窗口组件就可以组合出进度提示信息对话框。而实用功能主要是为组件或其他对象提供实用功能的，例如 Ext.JSON 功能，在模型的提交和数据读取等场合都会用到。因而不要小看这些组件和实用功能，它们往往会起大作用。

16.1　其他组件

1. 使用内联编辑组件：Ext.Editor

Editor 对象的作用就是单击页面上的元素的时候，会显示一个编辑器，让用户编辑当前元素的内容，编辑完成后可用编辑过的内容替换旧的内容。

使用内联编辑器，首先要定义编辑器使用的表单字段，还要定义如何激活编辑器进入编辑状态，常用的操作是单击或双击元素。下面通过一个示例演示如何使用内联编辑器。在页面中显示“姓名”和“学历”两个项，单击“姓名”使用 Text 字段编辑姓名，单击“学历”使用 ComboBox 选择学历。

使用模板页创建一个名称为 16-1.html 的页面文件。先定义好样式和页面中显示的内容，其中样式代码如下：

```
.demo{padding:20px 0 0 20px;font-size:14px;line-height:30px;width:165px;display:
    block;}
.demo label,span{float:left;display:block;}
.demo lable{width:60px}
.demo span{width:80px}
```

页面 HTML 代码如下：

```
<div class="demo">
    <label>姓名：</label><span id="name">张三</span>
    <label>学历：</label><span id="edu">本科</span>
</div>
```

元素 SPAN 加了 id 的目的是要为其绑定单击操作。

现在定义两个编辑器，一个使用 Text 字段，一个使用 ComboBox 字段，代码如下：

```
var editor = new Ext.Editor({
        autoSize:{width:"boundEl"},
```

```
        updateEl:true,
        field:{xtype: 'textfield',allowBlank:false,width:80}
});

var store1=Ext.create("Ext.data.ArrayStore",{
        fields:["text"],
        data:[["小学"],["初中"],["高中"],["本科"],["研究生"],["博士"],["博士后"]]
});

var combo= new Ext.Editor({
        autoSize:{width:"boundEl"},
        updateEl:true,
        field:{xtype:'combobox',allowBlank:false,queryMode:'local',
            store:store1
    }
});
```

在编辑器定义中，定义了配置项 autoSize，目的是让编辑器显示在元素 SPAN 内，其长度为 SPAN 的长度。若配置项 updateEl 为 true，则会在编辑完成后更新 SPAN 的内容。

最后要做的是为两个 SPAN 元素绑定单击事件，在单击后进入编辑状态，代码如下：

```
var el=Ext.get("name");
el.on("click",function(){
        editor.startEdit(this);
},el);

var el1=Ext.get("edu");
el1.on("click",function(){
        combo.startEdit(this);
},el1);
```

在 startEdit 方法内指定要编辑的元素就可进入编辑状态了。

在浏览器中打开页面，单击“张三”，修改为“李四”，然后单击“本科”，修改为“高中”，将看到如图 16-1 所示的过程。

图 16-1　示例的页面效果

2. 使用进度条：Ext.ProgressBar

进度条大家都很熟悉了，在 Windows 应用中经常能看到，而在页面中，因没有现成的组件，所以只能模拟出进度条的效果。在 Ext JS 中，是在一个带边框的 DIV 元素内，通过一个有背景的 DIV，根据比例修改其宽度来达到进度条的效果。

使用进度条，主要是使用其方法控制进度。使用进度条的主要方法有以下 5 个：

- ❑ isWaiting：如果进度条当前正在进行 wait 操作，返回 true。
- ❑ reset：重置进度条的值为 0，文本为空字符串。如果指定参数为 true，会隐藏进度条。
- ❑ updateProgress：更新进度条，带三个参数，依次为进度条的百分比值、进度条中显示

的文本和是否使用动画。

- ❑ updateText：更新进度条内的文本。
- ❑ wait：指示进度条自动进行更新操作，参数为一个配置对象，表 16-1 列出了其配置项及说明。

表 16-1 wait 方法参数的配置项及说明

配置项	类 型	说 明
duration	数字	设置进度在重置之前运行的时间长度，单位为微秒。默认值为 undefined
interval	数字	设置进度条更新的时间，默认值为 1000，也就是 1 秒更新一次
animate	布尔值	进度条切换是否显示动画，默认值为 undefined
increment	布尔值	进度条每次更新的增量，默认值为 10
text	字符串	进度条中显示的文本
fn	函数	进度条自动更新完成后执行的回调函数。如果没有设置 duration，该回调函数会被忽略
scope	对象	作用域

在知道进度的情况下，可使用 updateProgress 方法更新进度。如果不知道，则可使用 wait 方法让进度条自动更新，在接收到完成信息后，隐藏进度条，信息提示窗口的 wait 就是使用该方式实现的，多用在表单提交时。

下面演示一下如何使用进度条。在浏览器中打开模板页，然后在命令行中运行以下代码创建一个进度条：

```
var p=new Ext.ProgressBar({
    renderTo:Ext.getBody(),width:200,text:"1/10"
});
```

这样，在页面中就会看到图 16-2 中编号 1 所示的效果。再执行以下代码：

```
p.updateProgress(0.2,"2/10",true)
```

就会看到进度条以动画形式前进了 10%，最后看到如图 16-2 中编号 2 所示的效果。不断调整 updateProgress 内的参数，直到 1，进度条就会占满整个区间，表示操作完成，这个就不演示了。

下面演示一下 wait 方法的使用。在演示前，先用 reset 方法重置一下滚动条，用 upateText 更新一下滚动条内的问题为空，代码如下：

```
p.reset();
p.updateText("");
```

先测试一下配置项的 wait 方法，执行后，会看到进度条在 10 秒后会重新回到起点，继续运作，如果不调用 reset 方法，则会不停地进行下去。现在执行一次 reset 方法，重置一下进度条，然后执行以下代码：

```
p.wait({
```

```
        duration:3000,
        fn:function(){
            p.updateText("完成");
        }
    })
```

这样，进度条在执行 3 秒后，就会自动停止，并调用回调函数，如图 16-2 中编号 3 所示的效果，更新进度条内的文本为“完成”。

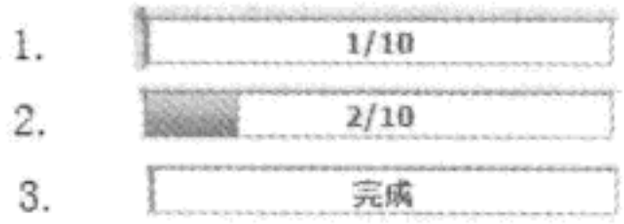


图 16-2　进度条的演示效果

3. 使用颜色选择器：Ext.picker.Color

ColorPalette 对象就是颜色列表，是用来选择颜色的。它与 Menu 结合，就成了一个颜色选择菜单。其运行原理就是使用模板，以 SPAN 元素为单元，利用其背景将 colors 数组中的颜色显示出来，而在 SPAN 元素外有一个 A 元素，其 class 属性的值是“color-”加上颜色值。这样，当单击或双击 A 元素时，就可根据 class 属性获取颜色值。

使用 ColorPalette 对象重要的配置项有 colors、handler 和 value 这三个。其中，colors 用于设置颜色列表，**要注意，颜色值必须是 6 位且必须是大写字母**，不然在使用正则表达式获取颜色值的时候会出错；handler 用于选择颜色后的操作；value 用于初始化选择值。

使用 getvalue 方法可返回选择的颜色值。使用 select 事件则可跟踪用户的选择。

在浏览器打开模板页，在命令行输入以下命令：

```
var c = new Ext.picker.Color({
    renderTo: Ext.getBody(),
    colors:["000000","FFFFFF","FF0000"],
    value:"FFFFFF",
    handler:function(picker,color){
        console.log(color);
    }
});
```

运行后会看到如图 16-3 所示的效果，单击红色后，在控制台可看到输出信息“FF0000”。


图 16-3　只有 3 个选择项的颜色选择器

4. 使用日期选择器：Ext.picker.Date

DatePicker 对象的主要作用是进行日期选择。它与 Trigger 对象结合，就成了日期选择字段；与 Menu 对象结合，就成了日期选择菜单。其工作原理与颜色选择器差不多，主要区别是它的小部件很多，例如年份选择、月份选择等。

使用日期选择器的主要配置项有以下几个：

- format：设置日期的格式。
- handler：选择日期后调用的函数。
- maxDate：设置日期选择的最大值。
- minDatre：设置日期选择的最小值。

- showToday：布尔值，默认值为 true，会显示今日按钮。

要获取选择日期值，可使用 getValue 方法；要设置日期值，则可使用 setValue 方法。

在浏览器打开模板页，在命令行输入以下命令：

```
var d = new Ext.picker.Date({
    renderTo: Ext.getBody(),
    showToday:false,
    format:"Y-m-d",
    minDate:new Date(2011,7,10),
    maxDate:new Date(2011,7,20),
    handler:function(picker,date){
        console.log(date);
    }
});
```

代码中，minDate 和 maxDate 必须是日期值，这里限制了日期的选择范围是 2011-8-10 到 2011-8-20。运行后，将看到如图 16-4 所示的页面效果。图中，今日按钮没有了，而除了规定范围的日期外，其余日期都是不能进行选择的。单击 17，将看到控制台输出“Date {Wed Aug 17 2011 00:00:00 GMT+0800}”，因为返回的是日期对象，所以不会返回日期格式定义的字符串，这个做提交处理的时候要注意。

图 16-4　日期选择器的页面效果

5. 使用时间选择器：Ext.picker.Time

TimePicker 对象派生于 BoundList 对象，也就是说它是一个视图。这个对象主要用来选择时间。其工作原理就是根据配置项生成时间，并把生成的时间放到内部创建的 Store 中，然后使用视图将其显示出来。

使用 TimePicker 对象，主要需定义以下 4 个配置项：

- format：定义时间的格式，默认值为“g:I A”。
- increment：每个时间值之间的增量，默认值是 15 分钟。
- maxValue：允许选择的最大时间值。
- minValue：允许选择的最小时间值。

若要从时间选择器中获取选择值，要根据视图的方式，通过选择模型获取选择的记录。

在浏览器打开模板页，在命令行输入以下命令：

```
var t=new Ext.create(Ext.picker.Time, {
    renderTo: Ext.getBody(),
    width:40,
    format:"H:i",
    increment:60,
    minValue: Ext.Date.parse('09:00', 'H:i'),
    maxValue: Ext.Date.parse('18:00', 'H:i'),
});
```

代码中，宽度如果不设置，则会填充整个浏览器的宽度；时间格式改回了习惯的 24 小

时格式；时间间隔设置为 1 小时；显示的时间是从 9 点到 18 点。

运行后将看到如图 16-5 所示的效果。在选择器中选择 10 点，然后输入以下命令：

```
console.dir(t.getSelectionModel().getSelection()[0].data);
```

图 16-5　时间选择器的页面效果

在控制台会看到以下输出：

```
date        Date {Tue Jan 01 2008 10:00:00 GMT+0800}
disp        "10:00"
```

从输出可以看到，内部 Store 包括 date 和 disp 两个字段，需要的值在 disp 字段，因而可使用 get 方法或直接通过 data 属性获取 disp 的值。

6. 使用图片组件：Ext.Img

在 Ext JS 3 及之前版本的 Ext JS 中，若要在组件中添加一个图片，需要定义一个容器，然后在容器的 html 配置项内加入 IMG 标记。如果图片是需要动态改变的，则需要在 IMG 标记内加入 id，然后使用 Ext.get 或 Ext.fly 获取元素再设置，相当麻烦。在 Ext JS 4 中终于加入了 Img 组件，显示和控制图片就方便多了。Img 组件并不复杂，就是生成一个 IMG 标记。但它具备了组件的特性。

使用 Img 组件很简单，定义对象时使用 src 配置项设置图片路径就可以了。如果要改变图片，调用 setSrc 方法就可以了。

在浏览器中打开模板页，在命令行执行以下命令：

```
var img=new Ext.Img({
    renderTo:Ext.getBody(),
    src:"../images/s1.jpg"
})
```

然后再执行以下命令：

```
img.setSrc("../images/s2.jpg");
```

页面中将会显示如图 16-6 所示的效果。

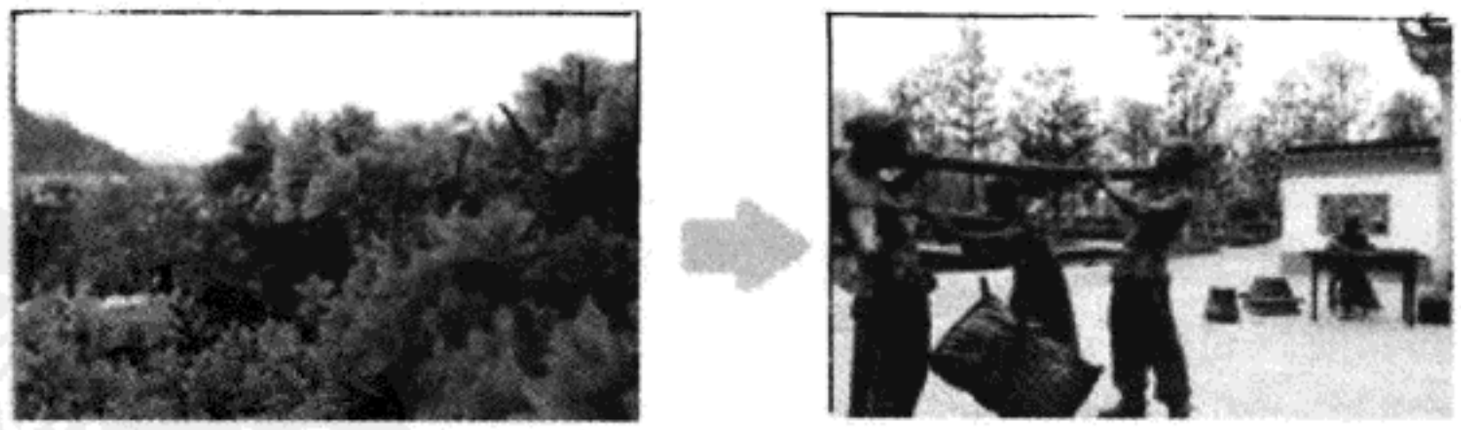


图 16-6　使用 Img 组件的页面效果

7. 使用 Flash 组件：Ext.FlashComponent

FlashComponent 对象是 Ext JS 4 新增的组件，用于播放 Flash 动画。它需要 SWFObject 库支持，该库可以从 http://code.google.com/p/swfobject/downloads/detail?name=swfobject_2_2.

zip&can=2&q= 下载。

要使用 FlashComponent 对象，首先要在页面中加入 SWFObject 库，代码如下：

```
<script type="text/javascript" src="Path/swfobject.js"></script>
```

代码中，Path 是 Swfobject.js 相对于页面文件的路径。

FlashComponent 对象会根据配置项创建一个 swfobject 对象的实例，然后由该实例生成 Flash 的 HTML 代码。这是使用 FlashComponent 对象比较麻烦的地方。

FlashComponent 对象一般设置好 url 配置项就可以了。如果要设置 Flash 的宽度和高度，不是使用组件习惯的 width 和 height 配置项，而要使用 swfWidth 和 swfHeight 配置项。

下面通过一个示例演示一下如何使用 FlashComponent 对象。使用模板页创建一个名称为 16-2.html 的页面文件，首先在页面 HEAD 部分加入 SWFObject 库的引用。

然后创建 FlashComponent 对象的代码：

```
Ext.create(Ext.FlashComponent,{
        renderTo:Ext.getBody(),
        swfWidth:320,
        swfHeight:220,
        url:"Bridge.swf"
})
```

代码很简单，只定义了 Flash 的宽度和高度，url 是必需的。

在浏览器中打开页面，Flash 加载完成后将看到如图 16-7 所示的页面效果。

图 16-7　示例的页面效果

16.2　使用滑块

1. 概述

MultiSlider 对象派生于 BaseField 对象，所以 MultiSlider 对象实际上是一个表单字段，有表单字段的特性。之所以没介绍表单时讲述滑块，主要原因是滑块在表单中的使用几率实在小，更多时候，滑块都是相对独立的组件。

滑块的工作原理是通过 DIV 标记在页面上虚拟出一条轴（横向或纵向），然后通过 Thumb 对象在轴上虚拟出一个个滑块，Thumb 对象内置了拖动功能，因而可模拟出滑块移动的效果，从而实现通过滑块改变数值的功能。再加上派生于 Tip 对象的 SliderTip 对象提供的提示信息，滑块的功能就完整了。

轴上有多少个滑块是由配置项 values 决定的。如果只提供一个值，那就是单滑块，也就是 SingleSlider 对象。SingleSlider 对象派生于 MultiSlider 对象，是 MultiSlider 对象的一个特例。

2. 使用多滑块组件：Ext.slider.Multi

使用多滑块组件要定义的配置项主要如下。

- constrainThumbs：布尔值，默认值为 true，表示不允许滑块叠在一起。

- decimalPrecision：保留的小数位数。默认值为 0，没有小数点。
- increment：滑块拖动时的调整量。默认值为 0。
- keyIncrement：使用键盘移动滑块时的调整量。默认值为 1。
- maxValue：设置滑块允许的最大值。默认值为 100。
- minValue：设置滑块允许的最小值。默认值为 0。
- tipText：一个函数，用来自定义滑块的提示信息，有点类似 renderer 配置项。
- useTips：布尔值，默认值为 true，显示提示信息，提示信息显示的是滑块的当前值。
- value：设置滑块的初始值，默认值为最小值（minValue 配置定义的值）。
- values：由数字组成的数组，一个数字代表一个滑块。
- vertical：布尔值。默认值为 false，表示滑块组件是水平放置的；若设置为 true，表示滑块组件将垂直放置。

在浏览器中打开模板页，在命令行输入以下命令：

```
var s1=Ext.create(Ext.slider.Multi,{
        renderTo: Ext.getBody(),
        width: 300,
        values: [25,50,74,90]
});
var s2=Ext.create(Ext.slider.Multi,{
        renderTo: Ext.getBody(),
        height: 300,
        values: [25],
        vertical:true
});
```

运行后，可看到如图 16-8 所示的效果。相信大家一定很好奇，滑块的数量既然是由 values 配置项决定的，为什么还要定义 SingleSlider 配置项呢？主要原因就是这二者的 setValue 方法和 getValue 方法的使用方式不同。例如，要设置示例中 s2 的值，不能直接将值传递给 setValue 方法，而应该使用以下方式：

```
s2.setValue(0,50);
```

在代码中，第 1 个参数是滑块的索引值，第 2 个参数才是滑块的数值。运行后，将会看到滑块移动到中间了。

同理，使用 getValue 方法也要在参数中指定索引值，指定返回哪个滑块的值，例如要想返回 s1 中第 2 个滑块的值，需要执行：

图 16-8　多滑块的示例效果

```
console.log(s1.getValue(1))
```

可以看到控制台会输出“50”。

如果要返回全部滑块的值，可使用 getValues 方法，返回一个数组，在命令行中输入以下

代码：

```
console.log(s1.getValues())
```

在控制台将看到以下输出：

```
[25, 50, 74, 90]
```

3. 使用单滑块组件：Ext.slider.Single

上一节已经提到，单滑块组件与多滑块组件的最大不同是 setValue 方法和 getValue 方法不同，因而使用单滑块组件与使用多滑块组件的方法基本是一样的。

刷新一下模板页，在命令行中输入以下命令：

```
var s1=Ext.create(Ext.slider.Single,{
        renderTo: Ext.getBody(),
        width: 300,
        value:30
});
```

运行后，在页面中会看到一个滑块，停留在 30 的位置。在命令行中输入以下命令：

```
s1.setValue(50)
```

滑块将会移动到中间，与多滑块语法进行比较，会发现这里不需要索引了。

再执行以下命令：

```
s1.getValue();
```

命令行将会输出“50”，这里也不要指定索引。

16.3 使用提示信息

1. 概述

在前面章节已经多次使用过提示信息，而且很多组件都有自己特殊的提示信息组件。那么提示信息到底是怎样的一个组件呢？

Tip 对象是所有提示信息的基类，它派生于 Panel 对象，也就是所有提示信息都是面板，准确地说，是一个浮动的面板。明白这点很重要。也就是说，可以通过 items 配置项，在提示信息中加入任何需要的组件，当然，你需要的话，加入工具栏也没问题。

从 Tip 对象派生出了 SliderTip 和 ToolTip 两个子对象。SliderTip 主要用于显示滑块的提示信息。而 ToolTip 则可应用到任何 HMTL 元素或 Ext JS 组件。而从 ToolTip 对象派生出的 QuickTip 对象，与 QuickTipManager 对象一起，可方便地为 HTML 元素添加提示信息。

2.ToolTip 的工作原理

ToolTip 的 initComponent 方法代码如下：

```
initComponent: function() {
    var me = this;
```

```
        me.callParent(arguments);
        me.lastActive = new Date();
        me.setTarget(me.target);
        me.origAnchor = me.anchor;
    },
```

在调用父对象的 initComponent 方法后，会设置 lastActive 属性，其主要作用是控制提示信息在什么时候自动关闭。接着，调用了 setTarget 方法，代码如下：

```
    setTarget: function(target) {
        var me = this,
            t = Ext.get(target),
            tg;
        if (me.target) {
            tg = Ext.get(me.target);
            me.mun(tg, 'mouseover', me.onTargetOver, me);
            me.mun(tg, 'mouseout', me.onTargetOut, me);
            me.mun(tg, 'mousemove', me.onMouseMove, me);
        }
        me.target = t;
        if (t) {
            me.mon(t, {
                freezeEvent: true,
                mouseover: me.onTargetOver,
                mouseout: me.onTargetOut,
                mousemove: me.onMouseMove,
                scope: me
            });
            if (me.anchor) {
                me.anchorTarget = me.target;
            }
        },
```

如果 target 属性存在，先将 target 指向的目标元素解除事件的绑定。然后再为新目标元素的 mouseover、mouserout 和 mousemove 事件绑定函数。这样每当鼠标移入、移出元素或在元素上移动，都会触发 Tooltip 内的方法了，从而可以显示、隐藏或移动提示信息。

我们看一下 onTargetOver 方法，其代码如下：

```
    onTargetOver: function(e) {
        var me = this,
            t;
        if (me.disabled || e.within(me.target.dom, true)) {
            return;
        }
        t = e.getTarget(me.delegate);
        if (t) {
            me.triggerElement = t;
            me.clearTimer('hide');
            me.targetXY = e.getXY();
            me.delayShow();
        }
    },
```

如果提示信息当前状态为被禁用，或事件不在目标元素内，则直接返回，不显示提示信息。

注意下面一句，调用 getTarget 方法，去寻找事件的委托目标，使用委托的好处是可以减少 ToolTip 对象的创建[⊖]。

当目标元素存在时，就可以调用 delayShow 方法显示提示信息了。

3. QuickTip 的工作原理

QuickTip 派生于 ToolTip 对象，其基本工作原理与 ToolTip 对象一样，不过 QuickTip 是绑定事件的目标元素页面的 Document 对象，也就是委托了文档对象来处理鼠标移入、移出或移动等操作。它重写了 onTargetOver 和 onTargetOut 方法。我们来看看 onTargetOver 方法，代码如下：

```
onTargetOver : function(e){
    var me = this,
        target = e.getTarget(me.delegate),
        hasShowDelay,
        delay,
        elTarget,
        cfg,
        ns,
        tipConfig,
        autoHide;

    if (me.disabled) {
        return;
    }

    me.targetXY = e.getXY();

    if(!target || target.nodeType !== 1 || target == document.documentElement ||
        target == document.body){
        return;
    }

    if (me.activeTarget && ((target == me.activeTarget.el) || Ext.fly(me.active-
        Target.el).contains(target))) {
        me.clearTimer('hide');
        me.show();
        return;
    }

    if (target) {
        Ext.Object.each(me.targets, function(key, value) {
        var targetEl = Ext.fly(value.target);
        if (targetEl && (targetEl.dom === target || targetEl.contains
            (target))) {
            elTarget = targetEl.dom;
            return false;
        }
```

⊖ 相关信息可阅读：http://iamtotti.com/blog/2011/05/what-makes-your-Ext JS-application-run-so-slow/
译文：http://blog.csdn.net/tianxiaode/article/details/6525486。

```
});
if (elTarget) {
    me.activeTarget = me.targets[elTarget.id];
    me.activeTarget.el = target;
    me.anchor = me.activeTarget.anchor;
    if (me.anchor) {
        me.anchorTarget = target;
    }
    hasShowDelay = Ext.isDefined(me.activeTarget.showDelay);
    if (hasShowDelay) {
        delay = me.showDelay;
        me.showDelay = me.activeTarget.showDelay;
    }
    me.delayShow();
    if (hasShowDelay) {
        me.showDelay = delay;
    }
    return;
  }
}

elTarget = Ext.get(target);
cfg = me.tagConfig;
ns = cfg.namespace;
tipConfig = me.getTipCfg(e);

if (tipConfig) {

    if (tipConfig.target) {
        target = tipConfig.target;
        elTarget = Ext.get(target);
    }
    autoHide = elTarget.getAttribute(ns + cfg.hide);

    me.activeTarget = {
        el: target,
        text: tipConfig.text,
        width: +elTarget.getAttribute(ns + cfg.width) || null,
        autoHide: autoHide != "user" && autoHide !== 'false',
        title: elTarget.getAttribute(ns + cfg.title),
        cls: elTarget.getAttribute(ns + cfg.cls),
        align: elTarget.getAttribute(ns + cfg.align)

    };
    me.anchor = elTarget.getAttribute(ns + cfg.anchor);
    if (me.anchor) {
        me.anchorTarget = target;
    }
    hasShowDelay = Ext.isDefined(me.activeTarget.showDelay);
    if (hasShowDelay) {
        delay = me.showDelay;
        me.showDelay = me.activeTarget.showDelay;
    }
    me.delayShow();
    if (hasShowDelay) {
```

```
                me.showDelay = delay;
            }
        }
    },
```

从第 2 个 if 语句可以看到，如果事件的目标不是元素、文档或 body 元素，都会直接返回，不再执行。

如果是当前活动的提示信息，则关闭它，并返回。

接着，是从 targets 指向的对象中寻找目标元素，如果找到了，则停止枚举操作，并且将 elTarget 指向该元素，从 targets 中取出提示信息的配置对象，显示提示信息。

如果不存在，则要使用 getTipCfg 方法检查元素的属性中是否有带有 QuickTip 规定的属性，如果有，说明该元素要显示提示信息，调用 delayShow 方法显示提示信息。

从代码可以看到，QuickTip 对象显示提示信息的主要方式是通过 register 方法将需要显示提示信息的对象注册到 QuickTip 对象的 targets 数组，另一种方式是通过属性进行判断。

QuickTip 对象存在的问题是在 Document 对象绑定事件会严重影响性能，因而使用时要注意。

4. 使用 ToolTip

使用 ToolTip 很简单，只要在创建时，定义好 target 和 html 配置项就可以了，其中，target 用于指定显示提示信息的元素，值可以为元素的 id 或元素对象本身；html 则用于定义要显示的提示信息。

在浏览器打开模板页，然后在命令行，命令创建一个容器，容器里包含一个 id 为 TipTest 的 DIV 元素，准备用来设置提示信息，代码如下：

```
Ext.create(Ext.container.Container,{
    renderTo:Ext.getBody(),
    html:"<div id='TipTest'>这里用来显示提示信息</div>"
});
```

运行后，就可附加提示信息了，代码如下：

```
Ext.create(Ext.tip.ToolTip,{
    target: 'TipTest',
    html:'提示信息'
});
```

运行后，将鼠标移动到“这里用来显示提示信息”上，将看到如图 16-9 所示的效果。

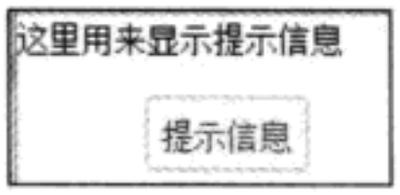


图 16-9 使用提示信息的示例

5. 在工具栏使用委托显示组件提示信息

在《What makes your Ext JS application run so slow ?》一文中，建议使用委托的方法来为按钮提供提示信息，而不要为每一个组件都创建一个 ToolTip 实例。该文中没有提供例子，所以本节将为其补充这样的例子。不过，在这里要说明一点，实际上很多可以定义提示信息的组件，使用的是 QuickTip 对象显示提示信息，而不是 ToolTip，所以除非你不想使用 QucikTip，才需要考虑使用本节的方法。

使用模板页创建一个名称为 16-3.html 的页面文件。在开始编码前，首先要考虑如何获取

目标元素，因为 delegate 配置项的值是 DomQuery 对象的选择符，需要根据该定义选择出元素。如果工具栏全部是按钮，那很简单，使用按钮的特征样式“x-btn”就可以了，但是如果还需要给其他组件，如 Text 字段加提示信息，该方法就行不通了。解决办法就是使用 cls 配置项为这些组件添加一个不起作用的样式，例如“Tips”，就可根据这个样式名称获取元素了。还有个问题是如何取得各元素的提示信息，办法就是为实例添加一个用于放置提示信息的属性。问题解决后，就可以开始编码了，最终编码如下：

```
Ext.create(Ext.toolbar.Toolbar,{
renderTo:Ext.getBody(),width:500,
items:[
        {text:"增加",cls:"Tips",t:"增加记录"},
        {text:"删除",cls:"Tips",t:"删除记录"},
        "|",
        "搜索",
        {xtype:"textfield",width:80,cls:"Tips",
                t:"请输入搜索文本"
        },
        "->",
        {text:"注销",cls:"Tips",t:"注销"}
],
listeners:{
        render:function(tb){
                tb.tip=Ext.create('Ext.tip.ToolTip', {
                        target:tb.el,
                        delegate:".Tips",
                        trackMouse: true,
                        renderTo: Ext.getBody(),
                    listeners:{
                        beforeshow:function(tip){
                            var obj=tb.items.getByKey(tip.triggerElement.id);
                             tip.update(obj.t);
                        }
                    }
                    });
                }
    }
})
```

代码中，凡是定义了 cls 配置项和添加了 t 配置项的组件都会显示提示信息，而 t 就是要显示的提示信息。当工具栏渲染后（render 事件），就可创建 ToolTip 对象实例了，其 target 配置项指向工具栏的容器 el。定义 renderTo 配置项的目的是渲染 ToolTip 对象实例，不然只是创建了组件，但没有渲染，就不起作用了。要更换提示信息的内容，则要在 ToolTip 对象实例 beforeshow 事件内进行，因为这是 triggerElement 属性才会指向触发的事件的元素，才可能根据其 id 从工具栏的 items 中找到组件，然后提取到 t 的值，用于更新提示信息的内容。

在浏览器中打开页面，然后将鼠标移动到 Text 字段上，将看到如图 16-10 所示的页面效果。

图 16-10　示例的页面效果

6. 在 Grid 中使用委托显示提示信息

在 Grid 中使用委托显示提示信息，如果不考虑列的情况，可直接定义 delegate 的值为视图的 itemSelector 属性，这样就可根据目标元素使用 getRecord 返回当前记录，取得数据。但是，很多时候是要考虑列的情况的，每个单元格可能显示的提示信息都不同，因而采用 itemSelector 属性作为 delegate 的值并不适合。那么使用 cellSelector 属性是否适合呢？也不适合，因为获取元素后，你还要根据元素去查找这是第几列的，而且现在不知道是在哪一行了。因而最好的解决办法，还是自定义 delegate 的选择符，例如在 renderer 函数内，专门为 td 加上一个特殊属性，例如 tip，其值就是要显示的提示信息。这样，delegate 的值就可以为“td[Tip]”。然后在 beforeshow 中，直接用 getAttribute 返回 Tip 属性的值作为提示信息就可以了。

将文件 10-1.html 复制一份，将其文件名修改为 16-4.html，然后将姓名列中 renderer 函数内的“data-qtip”修改为“Tip”。

最后在 viewConfig 的 listeners 配置对象内加入以下代码：

```
render:function(view){
   view.tip=Ext.create('Ext.tip.ToolTip', {
          target:view.el,
          delegate:"td[Tip]",
          trackMouse: true,
          renderTo: Ext.getBody(),
        listeners:{
            beforeshow:function(tip){
                tip.update(tip.triggerElement.getAttribute("Tip"));
            }
        }
   });
}
```

代码中，delegate 配置项使用了 td[Tip] 来查找目标元素。在 beforeshow 中，直接使用 Tip 的值来更新提示信息。

这样修改后，与原来使用 QuickTip 对象显示提示信息效果是一样的。

7. 使用 QuickTip 对象显示提示信息

要使用 QuickTip 对象显示提示信息，首先要在 OnReady 函数内执行以下语句：

```
Ext.tip.QuickTipManager.init();
```

该语句的作用是创建 QuickTip 对象的实例，·为 Document 对象绑定事件。

接着要做的是为元素绑定提示信息，有以下两种方法：

（1）在 HTML 元素内绑定 QuickTip 对象规定的属性

从 16.3.3 节可以知道，使用该方法，QuickTip 对象会直接使用 getAttribute 方法获取提示信息进行显示。其主要的属性有以下 8 个：

- data-qtip：定义提示信息的内容。
- data-qtitle：定义提示信息面板的标题（因为是面板，可以显示标题）。

- data-qwidth：定义提示信息面板的宽度。
- data-target：定义获取显示提供信息的目标元素的选择符。在 onTargetOver 方法内会调用 Ext.get 方法获取目标元素，因而其值可定义为符合 Ext.get 方法要求的选择符。
- data-hide：如果值为 user，相当于设置面板的 autoHide 配置项为 false；其余值相当于设置面板的 autoHide 配置项为 true。
- data-qclass：定义应用于提示信息的样式名称，相当于面板的 cls 配置项。
- data-qalign：定义提示信息的对齐方式。
- data-anchor：定义提示信息窗口的锚固位置。

（2）调用 QuickTipManager 对象 register 方法注册提示信息

使用该方式会将提示信息的元素及配置对象保存在 targets 指向的对象内，当要显示提示信息的时候，会枚举对象，找到与目标元素匹配的项后，取配置对象，显示提示信息。在组件中，大部分有提示信息配置项的组件都会使用该方式显示提示信息。

使用该方式要考虑枚举的效率问题，当目标元素很多的时候，效率不会太高，因而不建议太多的使用 register 方法。尤其是像视图这些经常更新的，如果不 unregister，会造成大量的冗余数据在对象中，效率会更低下。

该方式可直接使用 QuickTip 对象的配置项。

下面通过示例演示如何使用这两种方式显示提示信息。使用模板页创建一个名称为 16-5.html 的页面文件，然后定义两个 DIV 元素，代码如下：

```
<div data-qtip="属性里的提示信息 1">在这里使用属性方式显示提示信息 </div></br>
<div id="div2">在这里使用 register 方式显示提示信息 </div>
```

第一个 DIV 定义了“data-qtip”属性，可直接显示提示信息。现在为第二个 DIV 注册提示信息，代码如下：

```
Ext.tip.QuickTipManager.register({
    target: 'div2',
    title: '提示信息',
    text: 'registerd 的提示信息'
});
```

配置项 target 和 text（也可用 html，参考面板的配置项）是必需的。

别忘了把初始化语句加上。

在浏览器中打开页面，一次将鼠标移动到两行文本上，将看到如图 16-11 所示的提示信息。

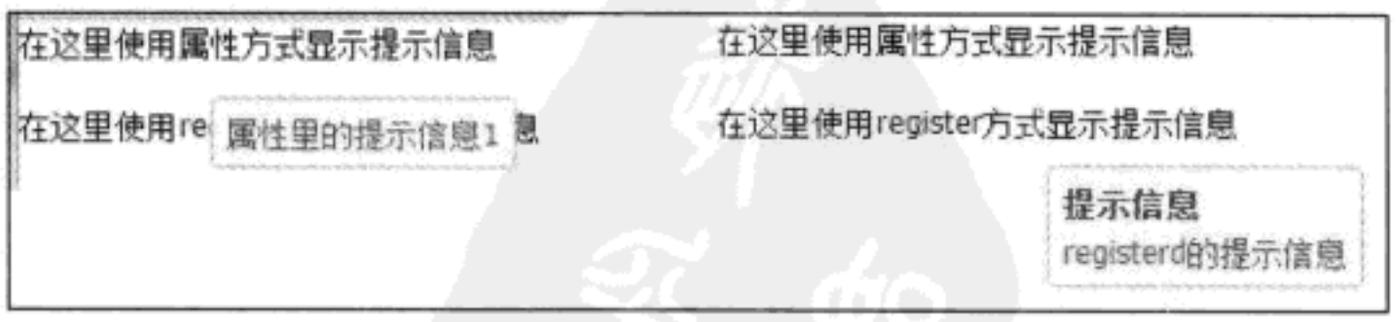


图 16-11　示例的页面效果

8. 关于使用 QuickTip 对象显示提示信息的问题

在 16.3.3 节，我们可以知道，QuickTip 对象会为 Document 对象绑定 3 个事件，这样，只

要鼠标在文档中移动，就有机会触发这 3 个事件，这是否会影响应用性能，值得商榷。不过暂时还没有这方面的反馈信息，笔者做的应用目前也没发现什么大问题。

如果对这个有担心，可以在执行初始化时（调用 init 方法），改变 QuickTip 对象实例的 target 指向，缩小其范围，例如在使用边框布局的应用中，只对 Center 区域使用 QuickTip，不过，这样就要等 Center 区域渲染后才能进行初始化，不然没有目标元素。

彻底的解决办法是完全使用 ToolTip 对象的委托代替 QuickTip 对象，但这样会增加许多不必要的代码量。

总之，目前不存在完美的解决方案，只有在项目开始根据项目情况权衡使用哪一种方式。

16.4 实用功能

1. 使用遮蔽功能：Ext.LoadMask

LoadMask 对象的作用是在加载组件、加载数据或进行耗时的运算时，为了避免用户操作影响过程而显示一个遮蔽层，禁止用户进行操作。在前面章节，进行 Store 加载或表单提交的时候，都已经使用过 LoadMask 对象了。

LoadMask 对象实际上是使用 Element 对象的 mark 方法和 unmark 方法来实现遮蔽功能。而 mark 方法的基本过程是在指定元素上放置一个样式是“x-mark”的 DIV 遮盖住元素，如果需要显示信息，则在遮蔽的 DIV 上再放置一个 DIV 用于显示信息。

要使用 LoadMask 对象很简单，在创建 LoadMask 对象实例时指定要实用遮蔽的元素，如果需要显示信息，则可添加配置对象，在配置对象内添加 msg 配置项，定义显示的信息。

在浏览器中打开模板页，然后在命令行创建一个 id 为 c1，300×300 的容器，代码如下：

```
Ext.create(Ext.container.Container,{
    renderTo:Ext.getBody(),
    id:"c1",
    height:300,width:300,
    html:"遮蔽演示"
})
```

运行后页面只能看到“遮蔽演示”这 4 个字。

现在创建一个 LoadMask 对象实例，指定元素为 c1，显示提示信息，代码如下：

```
var lm=Ext.create(Ext.LoadMask,"c1",{msg:"正在加载..."});
```

运行后页面不会有任何变化，要想激活遮蔽，需要调用其 show 方法，运行以下代码：

```
lm.show();
```

这时页面将会显示如图 16-12 所示的效果。要停止遮蔽，一般情况下是在加载完成后调用 hide 方法关

图 16-12　调用 show 方法后页面的显示

闭，因为现在没有加载，所有直接调用 hide 方法就可关闭遮蔽了。

2. 提供 JSON 处理功能：Ext.JSON

Ext.JSON 的主要功能有两个，一是将字符串转换为 JSON 对象，一是将 JSON 对象转换为字符串。还有一个辅助功能是将日期编码。

将字符串转换为 JSON 对象，使用的是 decode 方法，它有两种转换方式，一种是使用原生 JSON 的 parse 方法转换，一种是使用 eval 转换。使用原生的 parse 方法转换，对字符串的格式要求比较严格，例如以下格式的字符串都是错误的：

```
"{name:'张三',age:50}"
"{'name':'张三','age':50}"
```

正确的格式是：

```
'{"name":"张三","age":50}'
```

字符串中，JSON 对象的属性必须用双引号引起来，要注意，一定要双引号，不能是单引号。如果是字符串值，也必须是双引号引起来，而不能是单引号。使用 JSON 对象返回的 JSON 文本不存在格式问题，可以不用担心，但手动组织的就要注意了。

使用 evel 方法将字符串转换为 JSON 对象，虽然简单，但是不建议使用，因为它有性能和安全性问题。在 Ext JS 中，它的转换代码如下：

```
doDecode = function(json) {
    return eval("(" + json + ')');
},
```

使用 encode 方法将对象转换为字符串，也有两种方式，一种是使用原生 JSON 的 stringify 方法，一种是根据对象类型进行转换。

而在第二种方式中，对应日期对象，附加了一个 encodeDate 方法，用于转换日期，转换后的日期格式为“yyyy-mm-ddThh:mm:ss”。

3. 单词转换：Ext.util.Inflector

Inflector 对象的主要功能有 3 个：将单数的英文单词转换为复数（pluralize 方法）、将复数的英文单词转换为单数（singularize 方法）和将数字转换为英文序数（ordinalize）。

这些功能主要用于模型中创建关联模型数据读取的属性名称。

该功能知道就行了，使用不多。

4. 区域功能：Ext.util.Region

Region 对象的主要作用是通过 top、left、right 和 bottom 4 个属性，标示出一个矩形区域，以及进行几何转换或对区域进行检测。该功能主要用于拖放时获取区域信息，例如窗口在拖动时，要知道其能移动的区域及获取窗口本身的区域大小。

Region 对象提供了 getRegion 和 from 两个静态方法用于从对象创建 Region 对象实例。其中，getRegion 方法是使用 Element 对象的 getPageBox 方法创建实例的，因而其参数可以为 Element 对象实例的 id、Element 对象实例本身或 HTMLElement 对象；而 from 则是从一个带

有 top、left、right 和 bottom 属性的 JavaScript 对象中创建实例。

Region 对象还提供了检测点是否在区域内、检测点与区域的偏移量、检测两个区域是否相等、检测两个区域是否相交、检测区域是否包含另一个区域及根据偏移量转换区域等方法。

5. 坐标点：Ext.util.Point

Point 对象派生于 Region 对象，用于表示一个坐标点。其主要功能就是从事件（fromEvent 方法）中返回一个 Point 对象实例，用于几何转换以及与其他坐标点进行比较。

6. 测量文本尺寸：Ext.util.TextMetrics

TextMetrics 对象主要用于按钮、表单字段的标签、Textarea 字段根据文本的尺寸重新调整包裹文本元素的宽度和高度。测量主要通过静态方法 measure 实施。

7. 散列表：Ext.util.HashMap

HashMap 对象就是一个散列表，通过关键字和值的方法来访问数据，简单来说，就是一个 JavaScript 对象，但是 JavaScript 对象没有适合的方法来管理它，例如统计数量、返回所有关键字等。这些功能，都要自己做封装，或者在每次需要的时候写一堆的代码，这样很不方便，于是就有了 HashMap 对象。MixedCollection 对象其实也具有散列表的功能，但是它更庞大，包含了过滤、排序等功能，在某些时候我们并不需要这么多的功能，这时一个简单的 HashMap 对象就能满足我们的需求。

HashMap 对象提供了以下 14 个方法用于管理 JavaScript 对象：

- add：添加一个项。参数依次为关键字、对象。
- clear：清空散列表。把 map 属性指向一个空对象。
- clone：克隆散列表。
- contains：检查散列表是否包含指定的值。
- containsKey：检查散列表是否包含指定的关键字。
- each ：枚举散列表。枚举函数依次可接收 key（关键字）、value（值）和 length（散列表包含的数据总数）。
- get：根据指定的关键字获取值。
- getCount：返回散列表的数据总数。
- getKey：根据指定的值返回关键字。
- getKeys：返回由散列表内所有关键字组成的数组。
- getValues：返回由散列表内所有的值组成的数组。
- remove：根据指定的值删除一个数据。
- removeAtKey：根据指定的关键字删除一个数据。
- replace：替换散列表内的一个数据。

要使用散列表，直接创建一个 HashMap 对象实例就可以了。在浏览器中打开模板页，然后在命令行执行以下代码就可创建一个散列表：

```
var hm = Ext.create(Ext.util.HashMap);
```

现在可通过 add 方法为散列表添加数据了，代码如下：

```
hm.add("name","张三");
```

执行以下依据就可以看到散列表内的数据：

```
console.dir(hm.map)
```

在控制台会看到以下输出：

```
name          "张三"
```

正是刚才添加的数据。

有兴趣可以使用命令行把散列表的所有方法都测试一次，在这里就不做过多演示了。

8. 样式维护：Ext.util.CSS

CSS 对象是 Ext JS 新增的一个功能类，是一个单件模式的类，主要用于维护样式。目前的主要功能包括在 head 标记内创建 Style 标记、移除 Style 标记、替换 Style 标记、从样式表中根据选择符返回样式和更新样式。目前的主要应用是在面板使用画图功能画出垂直标题时，获取标题样式，然后将其作为图形的文字样式。至于其他方面的应用，还有待发掘。

9. 操作 Cookies：Ext.util.Cookies

对于框架来说，操作 Cookies 是必不可少的功能，目前来说，Cookies 使用太广泛了，没有该功能会很不方便。Cookies 对象也是一个单件模式的类，提供了以下三个方法用于操作 Cookies：

- clear：清除指定名称的 Cookies。
- get：返回指定名称的 Cookies。
- set：设置 Cookies。

10. 历史记录：Ext.util.History

对于使用浏览器的用户来说，使用浏览器提供的前后按钮切换页面是再正常不过的事了，但是对于单页面的应用来说，页面只有一个，如果直接使用前后按钮，将会跳出应用，因而需要模拟出多页面切换的效果。对于旧版本的 IE，通常的做法是在页面中插入一个 Iframe 来实现历史记录功能，而对于其他浏览器，在页面插入一个隐藏字段就可以了。

在 Ext JS 中，History 对象就是实现该功能的对象，它是一个单件模式的类。它会根据设置修改 Iframe 或隐藏字段的内容，从而达到模拟历史记录的功能。

要实现历史记录功能，首先要在页面中加入以下 HTML 代码：

```
<form id="history-form" class="x-hide-display">
    <input type="hidden" id="x-history-field" />
    <iframe id="x-history-frame"></iframe>
</form>
```

代码中 input 的 id 必须与 History 对象的 fieldId 配置项的值相同，而 Iframe 的 id 必须 iframeId 的值相同，不然 History 对象就找不到隐藏字段和 Iframe 了，这样就修改不了 input 或 Iframe 的内容，实现不了历史记录的功能。fieldId 的默认值是“x-history-field”，而

iframeId 的默认值为“x-history-frame”，因而没必要画蛇添足对它们做修改了，直接使用在 id 上就好了。

还要做的事情是调用 History 对象的 init 执行初始化工作；绑定它的 change 事件，在监测到用户单击了浏览器导航按钮后，触发该事件做处理；当用户切换了内容面板的时候，调用 History 对象的 add 添加一个令牌，通过该令牌可以将内容面板显示回切换前的状态。

下面尝试为 9.8.2 节示例添加历史记录功能，学习一下如何使用 History 对象。复制一份 9-5.html 文件，将其文件名修改为 16-6.html。首先要做的是在页面中添加 HTML 代码，然后加入调用 init 方法的代码：

```
Ext.History.init();
```

现在要考虑如何添加令牌，以及在哪里添加令牌，内容页的切换是根据节点 id 在 itemclick 函数内进行，因而这是一个比较理想的添加令牌的地方，而令牌就用节点 id，这样恢复的时候，可以直接根据 id 切换面板了。在 itemclick 函数内 if 语句之前添加以下语句：

```
Ext.History.add(node.data.id);
```

因为初始的面板是产品管理面板，而其切换 id 为“Product”，因而可加入一个初始状态的令牌，在 init 语句下加入以下语句：

```
Ext.History.add("Product");
```

接着要绑定 change 事件监测导航按钮的动作了，代码如下：

```
Ext.History.on('change', function(token) {
    var cmp=Ext.getCmp(token+"Content"),
            contentPage=Ext.getCmp("ContentPage");
    if(cmp){
            contentPage.getLayout().setActiveItem(cmp);
    }
});
```

这和 itemclick 函数差不多，根据返回的令牌值找到内容面板，然后将其设置为活动面板就行了。

在浏览器中打开页面，然后单击“菜单”切换一下内容面板，再单击浏览器的导航按钮，就可看到切换的是内容面板了。

16.5 本章小结

本章很多组件或实用功能是为 Ext JS 组件的内部运作而制作的，因而有机会研究一下它们的结构对理解 Ext JS 的运作和组件的工作原理，尤其是研究其源代码非常有帮助。而且 HashMap 对象、提示信息、LoadMask、进度条等对象在其他组件的方法调用中也会经常遇到，必须掌握其使用方法。

第 17 章　可简化通信的 Ext.Direct

Ext.Direct 是在 Ext JS 3 中加入的对象，其主要目的是简化客户端和服务器端之间的通信，提供一个单一的接口，从而减少代码量，例如对请求的数据验证和返回数据的处理。这些都是通过路由将客户端参数与服务器端方法参数对接起来实现的，例如提交的参数类型与方法定义的参数数据类型不对，就会报错，无需自己再对这些参数进行验证。而这也在一定程度上实现了客户端代码与服务器平台和语言的无关性。

本章将介绍 Ext.Direct 的工作原理、构成及如何使用它。

17.1　准备工作

使用 Ext.Direct 首先要做的是根据开发环境下载相应的路由库，目前的路由库已支持 PHP、Java、.NET、ColdFusion、Ruby 和 Perl，下载回来就可以使用了。不过每一种语言会根据使用环境不同而有不同的库，例如基于 .NET 的，有支持 MVC 的，有支持 ASP.NET 等库的，在下载前一定要结合项目使用的开发框架选择这些库。这些库都可以在 Sencha 的论坛的 Ext.Direct 子板块中找到。

因为这些库太多，本章不可能全部讲述，因而选取了基于 ASP.NET 和 servlet 的两个库进行讲述，这两个库是比较常用的，下载地址如下：

- ASP.NET：http://www.sencha.com/forum/showthread.php?68161-Ext.Direct-.NET-Router
- Java：http://code.google.com/p/directjngine/downloads/list

在本书的资源包的资源目录里也可找到这两个库，基于 ASP.NET 的 Router 库文件名是 Router-1.0.zip，基于 Java 的 directjngine 库文件名是 directjngine.2.0-alpha1.zip。

17.2　Ext.Direct 的工作原理及构成

17.2.1　工作原理

Ext.Direct 的工作原理如图 17-1 所示，API 会收集所有要映射到页面的对象及方法，然后将对象名称和方法映射到页面，DirectManager 根据添加的 DirectProvider 对象，从 API 返回的数据中读取对象名称及其方法，然后根据对象名称和方法创建对应的对象及其方法，在调用客户端的对象的方法时，会自动生成提交数据并将这些数据提交到路由，让路由根据提交的对象名称和方法，调用对应的对象的方法，将数据返回页面。

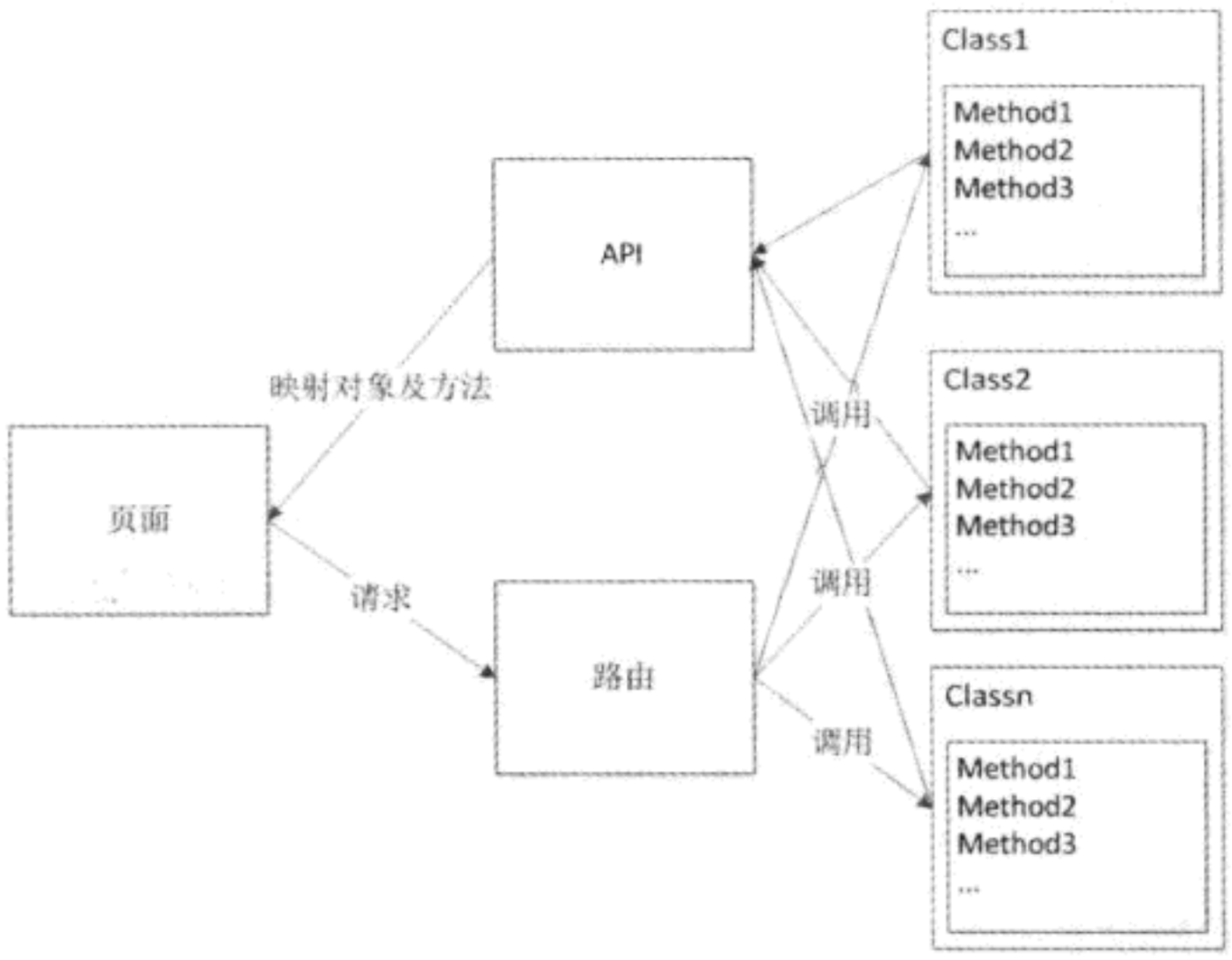


图 17-1　Ext.Direct 的工作原理

根据工作原理，可以知道，使用 Ext.Direct 有两个必不可少的配置，一个是将服务器端的对象及其方法映射到页面的 API，一个是根据请求调用对应方法的路由，缺一不可。

17.2.2　Ext.Direct 的构成

Ext.Direct 一共有 14 个类，根据功能可分为提供者、事件处理、数据通讯处理、数据存储及其他等几类。

提供者类的抽象基类是 DirectProvider，它为提供者类搭建了一个拥有 connect、disconnect、data 和 exception 这 4 个事件，以及 isConnected、connect 和 disconnect 这 3 个方法的框架。

JsonProvider 对象派生于 DirectProvider 对象，用于处理 JSON 数据，它在 DirectProvider 对象的基础上添加了 parseResponse 方法，用于将服务器端返回的数据转换为对象；createEvents 方法用于根据返回的数据创建不同类型的事件；createEvent 方法用于创建 Ext.Direct 的事件实例。

根据格式处理要求，应该还要有处理 XML 数据的提供者，不过目前还没实现，目前使用 Ext.Direct 只能使用 JSON 格式的数据，这个要特别注意。或者自己写一套支持 XML 格式的扩展也行。

PollingProvider 对象派生于 JsonProvider 对象，其主要作用是定时向服务器端发送请求，实现轮询功能，适合于实时性比较强的工作。这个以后再详细讲述。

RemotingProvider 对象也派生于 JsonProvider 对象，它是 Ext.Direct 的主角，主要工作是根据映射的对象及其方法创建客户端对应的对象和方法，然后对象调用方法时，创建 Transaction 对象实例，通过队列发送请求，请求返回后的处理请求等。

DirectEvent 是所有 Ext.Direct 事件对象的基类，主要结构就是包括一个 status 属性和 getData 方法。

RemotingEvent 对象派生于 DirectEvent 对象，在 DirectEvent 对象基础上添加了 getTransaction 方法，用于返回 Transaction 对象实例。

ExceptionEvent 对象派生于 RemotingEvent 对象，当服务器端返回错误的时候，会创建 ExceptionEvent 对象的实例，它只是将 status 属性修改为 false。

每一个服务器的方法，都会创建一个 RemotingMethod 对象实例，这就是 RemotingMethod 对象的作用。通过其 getCallData 方法，可以将方法的参数组织成提交参数，用于发送请求。

Transaction 对象类似于 Operation 对象，每一个 Transaction 对象就是一个发送请求，用于记录请求的信息。

为了支持 Grid、Tree 等需要使用 Store 的组件，根据 Ext.Direct 的数据格式及处理方式，创建了 DirectProxy 对象和 DirectStore 对象。

DirectProxy 对象派生于 ServerProxy 对象，主要改变是根据 paramOrder 对提交参数进行排序；重写 doRequest 方法，以 Ext.Direct 的提交方式提交数据。

DirectStore 对象派生于 Store 对象，主要变化是设置其默认 Proxy 为 DirectProxy，并将配置对象中的 paramOrder、paramsAsHash、directFn、API 和 simpleSortMode 等配置项复制到 DirectProxy；设置默认的 Reader 对象为 JsonRender 对象，还要将配置项对象中的 totalProperty、root、idProperty 等配置项复制到 JsonRender 对象中。

相应的，表单的加载和提交也必须符合 Ext.Direct 的格式要求，因而创建了 DirectLoad 和 DirectSubmit 这两个对象。

DirectLoad 对象主要工作是将提交参数根据 paramOrder 排序，以符合方法的参数和数据返回后的处理。

因为 DirectSubmit 是表单提交，参数只有一个，其主要工作是创建一个表单，然后调用方法提交数据。还要对返回信息依据 Ext.Direct 的方法进行处理。

把以上对象串联起来有序进行工作的是 DirectManager 对象，它是一个单件模式对象。它用两个 MixedCollection 对象实例来记录提供者对象实例和 Transaction 对象实例，并提供方法对这两个数据集合进行管理。它会提供了 event 和 exception 两个事件，用于处理服务器端事件。

17.2.3　RemotingProvider 对象的具体工作流程

Ext.Direct 的工作是从调用 DirectManager 的 addProvider 方法创建提供者开始的，其代码如下：

```
addProvider : function(provider){
    var me = this,
        args = arguments,
        i = 0,
        len;
    if (args.length > 1) {
```

```
        for (len = args.length; i < len; ++i) {
            me.addProvider(args[i]);
        }
        return;
    }
    if (!provider.isProvider) {
        provider = Ext.create('direct.' + provider.type + 'provider', provider);
    }
    me.providers.add(provider);
    provider.on('data', me.onProviderData, me);
    if (!provider.isConnected()) {
        provider.connect();
    }
    return provider;
},
```

如果参数 provider 是数组，会逐个递归调用 addProvider 方法，如果对象没有 isProvider 属性，说明不是提供者实例，则创建提供者实例。以下是 RemotingProvider 对象的构造函数：

```
constructor : function(config){
    var me = this;
    me.callParent(arguments);
    me.addEvents(
        'beforecall',
        'call'
    );
    me.namespace = (Ext.isString(me.namespace)) ? Ext.ns(me.namespace) : me.names-
        pace || window;
    me.transactions = Ext.create('Ext.util.MixedCollection');
    me.callBuffer = [];
},
```

要想看明白这段代码，要先清楚 API 返回的数据格式，基本的格式如下：

```
Ext.ns('Ext.app.REMOTING_API');
Ext.app.REMOTING_API = {
    "type":"remoting",
    "url":"",
    "namespace":"",
    "actions":{
            "Class1":[
                    {"name":"Method1","len":1},
                    {"name":"Method2","len":3},
                    {"name":"Method3","len":1,"formHandler":true}
            ]
    }
};
```

代码中，第一行是在 Ext 注册一个命名空间，从而可以访问到 Ext.app.REMOTING_API，在 addProvider 添加的就是该命名空间指向的对象，也就是，在创建 RemotingProvider 对象实例的时候，服务器端返回的对象就是配置对象，于是在构造函数中，如果返回的数据中 namespace 不为空，那么就会调用 ns 创建这个命名空间，不然 RemotingProvider 实例的 namespace 属性就指向 window 对象，通过全局变量方式调用映射的对象及方法。

在创建了 RemotingProvider 实例后，会将其加入到 providers 属性中（MixedCollection 对象实例），然后将 RemotingProvider 实例的 data 事件绑定到 onProviderData 方法，其代码如下：

```
onProviderData : function(provider, event){
    var me = this,
        i = 0,
        len;

    if (Ext.isArray(event)) {
        for (len = event.length; i < len; ++i) {
            me.onProviderData(provider, event[i]);
        }
        return;
    }
    if (event.name && event.name != 'event' && event.name != 'exception') {
        me.fireEvent(event.name, event);
    } else if (event.type == 'exception') {
        me.fireEvent('exception', event);
    }
    me.fireEvent('event', event, provider);
}
```

从代码可以看到，如果 event（服务器返回的数据）是数组，会递归调用 onProviderData 方法，如果 event 的 name 属性不是 event 或 exception，就触发 name 的值指定的事件。如果 type 为 exception，则触发 exception 事件。最后触发 event 事件。

回到 addProvider 方法，RemotingProvider 实例在创建时，connected 属性为 undefined，因而 isConnected 会返回 fasle，要调用 connect 方法，代码如下：

```
connect: function(){
    var me = this;
    if (me.url) {
        me.initAPI();
        me.connected = true;
        me.fireEvent('connect', me);
    } else if(!me.url) {
        Ext.Error.raise('Error initializing RemotingProvider, no url configured.');
    }
},
```

如果 url 属性存在，也就是服务器返回的数据存在 url 属性，则调用 initAPI 方法，代码如下：

```
initAPI : function(){
    var actions = this.actions,
    namespace = this.namespace,
    action,
    cls,
    methods,
    i,
    len,
    method;
```

```
for (action in actions) {
    if (actions.hasOwnProperty(action)) {
        cls = namespace[action];
        if (!cls) {
            cls = namespace[action] = {};
        }
        methods = actions[action];

        for (i = 0, len = methods.length; i < len; ++i) {
            method = new Ext.direct.RemotingMethod(methods[i]);
            cls[method.name] = this.createHandler(action, method);
        }
    }
},
```

代码中的 actions 就是服务器返回的数据中的 actions 属性，也就是在这里会处理服务器端返回的对象名称及其方法。注意命名空间，根据构造函数的代码，如果是 window，就都成了全局变量了，因而不是好方法，所以在服务器端返回的数据一定要定义 namespace 属性。

如果在定义的命名空间内，以服务器对象名称为属性的对象不存在，则将其指向一个空对象。

接着是读取方法了，注意粗体字体的代码，每一个方法会创建一个 RemotingMethod 对象实例。RemotingMethod 对象的构造函数如下：

```
constructor: function(config){
    var me = this,
        params = Ext.isDefined(config.params) ? config.params : config.len,
        name;
    me.name = config.name;
    me.formHandler = config.formHandler;
    if (Ext.isNumber(params)) {
        me.len = params;
        me.ordered = true;
    } else {
        me.params = [];
        Ext.each(params, function(param){
            name = Ext.isObject(param) ? param.name : param;
            me.params.push(name);
        });
    }
},
```

根据服务器端返回的数据基本格式，返回的只有参数格式，也就是 len 属性，因而现在 params 的值就是这个 len 属性的值，也就是数字。

如果 params 是数字，那么就定义 len 属性为该值，而且要设置 ordered 属性为 true 表示变量要依次序提交。如果不是，则将返回的参数依次存到 params 数组里，提交的参数将依据该数组的格式次序提交。

RemotingMethod 对象实例创建后，调用 createHandler 方法，这是关键一步，其代码如下：

```
createHandler : function(action, method){
    var me = this,
        handler;

    if (!method.formHandler) {
        handler = function(){
            me.configureRequest(action, method, Array.prototype.slice.call(argu-
                ments, 0));
        };
    } else {
        handler = function(form, callback, scope){
            me.configureFormRequest(action, method, form, callback, scope);
        };
    }
    handler.directCfg = {
        action: action,
        method: method
    };
    return handler;
},
```

属性 formHandler 表示方法是以表单方式提交的。不是表单提交的，匿名函数调用的是 configureRequest 方法，而表单方式提交的，则调用 configureFormRequest 方法。注意最后返回的就是这个匿名函数。也就是说，当利用模拟的远程对象调用其方法的时候，客户端实际执行的是 configureRequest 或 configureFormRequest 方法。

除了创建匿名函数外，还为这个匿名函数添加了一个 directCfg 属性，该属性指向一个对象，包含了 action 和 method 两个属性，action 就是对象名称，method 指向的是 RemotingMethod 对象实例。

这样，initAPI 方法就执行完了，回到 connect 方法，设置 connected 属性为 true，并触发 connect 事件。

至此，addProvider 方法就执行完毕了。

从代码可以看到，使用 addProvider 方法添加 RemotingProvider 对象的作用就是将服务器端的对象和方法模拟成客户端的对象和方法，并将其绑定到 configureRequest 或 configureFormRequest 方法。

下面看看这两个方法会做什么操作，以下是 configureRequest 的代码：

```
configureRequest: function(action, method, args){
    var me = this,
        callData = method.getCallData(args),
        data = callData.data,
        callback = callData.callback,
        scope = callData.scope,
        transaction;

transaction = new Ext.direct.Transaction({
    provider: me,
    args: args,
    action: action,
    method: method.name,
```

```
            data: data,
            callback: scope && Ext.isFunction(callback) ? Ext.Function.bind(callback,
                scope) : callback
        });

        if (me.fireEvent('beforecall', me, transaction, method) !== false) {
            Ext.direct.Manager.addTransaction(transaction);
            me.queueTransaction(transaction);
            me.fireEvent('call', me, transaction, method);
        }
    },
```

代码先调用 RemotingMethod 对象的 getCallData 方法组织数据，其代码如下：

```
getCallData: function(args){
    var me = this,
        data = null,
        len  = me.len,
        params = me.params,
        callback,
        scope,
        name;

    if (me.ordered) {
        callback = args[len];
        scope = args[len + 1];
        if (len !== 0) {
            data = args.slice(0, len);
        }
    } else {
        data = Ext.apply({}, args[0]);
        callback = args[1];
        scope = args[2];


       for (name in data) {
            if (data.hasOwnProperty(name)) {
                if (!Ext.Array.contains(params, name)) {
                    delete data[name];
                }
            }
        }
    }

    return {
        data: data,
        callback: callback,
        scope: scope
    };
}
```

如果 ordered 为 true，也就是说服务器端方法的参数为 len 个，因而，客户端方法在调用时，方法的参数中前面“len”个参数就是对应的服务器端方法的参数，将变量 data 指向该数据。而第 len 个就是回调函数，len 加 1 个参数就是作用域。

如果 ordered 不为 true，则说明要根据服务器端参数名组织数据，这时候第一个参数就是提交的数据，第二个参数是回调函数，第 3 个是作用域。这时候要将对象转换为数组。

最后返回一个对象，成员包括提交参数、回调函数和作用域。

数据返回后，就创建一个 Transaction 对象实例。

如果 beforecall 没有中止操作，则使用 addTransaction 方法将 Transaction 对象实例加入 DirectManager 的 transactions 中。然后调用 queueTransaction 方法，其代码如下：

```
queueTransaction: function(transaction){
    var me = this,
        enableBuffer = me.enableBuffer;
    if (transaction.form) {
        me.sendFormRequest(transaction);
        return;
    }
    me.callBuffer.push(transaction);
    if (enableBuffer) {
        if (!me.callTask) {
            me.callTask = new Ext.util.DelayedTask(me.combineAndSend, me);
        }
        me.callTask.delay(Ext.isNumber(enableBuffer) ? enableBuffer : 10);
    } else {
        me.combineAndSend();
    }
},
```

如果是表单方式提交，则调用 sendFormRequest 方法，再调用 Ajax 的 request 方法将表单发送到服务器。

如果开启了缓冲功能，则延时调用 combineAndSend 方法发送请求，否则直接发送请求。方法 combineAndSend 的代码如下：

```
combineAndSend : function(){
    var buffer = this.callBuffer,
        len = buffer.length;

    if (len > 0) {
        this.sendRequest(len == 1 ? buffer[0] : buffer);
        this.callBuffer = [];
    }
},
```

无论有没有缓冲，都是调用 sendRequest 方法发送请求，其代码如下：

```
sendRequest : function(data){
    var me = this,
        request = {
            url: me.url,
            callback: me.onData,
            scope: me,
            transaction: data,
            timeout: me.timeout
        }, callData,
```

```
        enableUrlEncode = me.enableUrlEncode,
        i = 0,
        len,
        params;
    if (Ext.isArray(data)) {
        callData = [];
        for (len = data.length; i < len; ++i) {
            callData.push(me.getCallData(data[i]));
        }
    } else {
        callData = me.getCallData(data);
    }
    if (enableUrlEncode) {
        params = {};
        params[Ext.isString(enableUrlEncode) ? enableUrlEncode : 'data'] = Ext.
            encode(callData);
        request.params = params;
    } else {
        request.jsonData = callData;
    }
    Ext.Ajax.request(request);
},
```

代码先创建了一个 request 对象，和 Proxy 中处理请求一样。接着要做的是调用 getCallData 方法组织提交参数，其代码如下：

```
getCallData: function(transaction){
    return {
        action: transaction.action,
        method: transaction.method,
        data: transaction.data,
        type: 'rpc',
        tid: transaction.id
    };
},
```

可以看到，提交的参数有 5 个，action 是服务器端对象的名称，method 为其方法，data 就是数据，请求类型为 rpc，tid 为 Transaction 对象的实例。

数据准备好以后就发送请求，结束 sendRequest 方法。

回调 configureRequest 方法，触发 call 事件，方法执行完毕。

方法 configureFormRequest 与 configureRequest 方法差不多，最终会调用 sendFormRequest 方法发送请求。

将请求发送到服务器端后，路由就可根据提交参数的 action 找到对应的类对象，根据 method 找到对象内的方法，然后调用找到的方法，并将提交的数据传递到方法内进行处理，之后返回数据。

注意 request 对象中，callback 属性指向的是 onData，也就是数据返回后会调用 onData 方法，其代码如下：

```
onData: function(options, success, response){
    var me = this,
```

```
        i = 0,
        len,
        events,
        event,
        transaction,
        transactions;

    if (success) {
        events = me.createEvents(response);
        for (len = events.length; i < len; ++i) {
            event = events[i];
            transaction = me.getTransaction(event);
            me.fireEvent('data', me, event);
            if (transaction) {
                me.runCallback(transaction, event, true);
                Ext.direct.Manager.removeTransaction(transaction);
            }
        }
    } else {
        transactions = [].concat(options.transaction);
        for (len = transactions.length; i < len; ++i) {
            transaction = me.getTransaction(transactions[i]);
            if (transaction && transaction.retryCount < me.maxRetries) {
                transaction.retry();
            } else {
                event = new Ext.direct.ExceptionEvent({
                    data: null,
                    transaction: transaction,
                    code: Ext.direct.Manager.self.exceptions.TRANSPORT,
                    message: 'Unable to connect to the server.',
                    xhr: response
                });
                me.fireEvent('data', me, event);
                if (transaction) {
                    me.runCallback(transaction, event, false);
                    Ext.direct.Manager.removeTransaction(transaction);
                }
            }
        }
    }
},
```

如果 success 属性为 true，那么就调用 createEvents 方法（JsonProvider.js）创建事件，其代码如下：

```
createEvents: function(response){
    var data = null,
        events = [],
        event,
        i = 0,
        len;
    try{
        data = this.parseResponse(response);
    } catch(e) {
        event = new Ext.direct.ExceptionEvent({
```

```
                data: e,
                xhr: response,
                code: Ext.direct.Manager.self.exceptions.PARSE,
                message: 'Error parsing json response: \n\n ' + data
            });
            return [event];
        }
        if (Ext.isArray(data)) {
            for (len = data.length; i < len; ++i) {
                events.push(this.createEvent(data[i]));
            }
        } else {
            events.push(this.createEvent(data));
        }
        return events;
    },
```

首先尝试将返回的数据转换为对象。如果发生错误，创建 ExceptionEvent 对象实例，并返回。

如果是 data 是数组，则依次调用 createEvent 方法，否则直接调用 createEvent 方法，其代码如下：

```
createEvent: function(response){
    return Ext.create('direct.' + response.type, response);
}
```

就是根据返回的 type，创建一个实例，返回的类型有 rpc、exception 和 event，分别对应 RemotingEvent、ExceptionEvent 和 DirectEvent。

事件创建完后返回 onData 方法，开始遍历事件，根据事件调用 getTransaction 方法找到提交请求的 Transaction 对象实例，触发 data 事件，调用实例中定义的回调函数，然后从 transactions 中删除该实例。

如果不成功，则调用 retry 方法重新发送请求，如果重试次数超出最大尝试数，则创建 ExceptionEvent 对象实例，触发 data 事件，并删除 Transaction 对象实例。

这就是一个完整的 Ext.Direct 处理过程。

17.2.4 PollingProvider 对象的具体工作流程

PollingProvider 对象也是从 addProvider 方法创建对象实例开始的，其构造函数如下：

```
constructor : function(config){
    this.callParent(arguments);
    this.addEvents(
        'beforepoll',
        'poll'
    );
},
```

只是简单的添加了 beforepoll 和 poll 两个事件。

接着是执行 connect 方法，代码如下：

```
connect: function(){
    var me = this, url = me.url;
    if (url && !me.pollTask) {
        me.pollTask = Ext.TaskManager.start({
            run: function(){
                if (me.fireEvent('beforepoll', me) !== false) {
                    if (Ext.isFunction(url)) {
                        url(me.baseParams);
                    } else {
                        Ext.Ajax.request({
                            url: url,
                            callback: me.onData,
                            scope: me,
                            params: me.baseParams
                        });
                    }
                }
            },
            interval: me.interval,
            scope: me
        });
        me.fireEvent('connect', me);
    } else if (!url) {
        Ext.Error.raise('Error initializing PollingProvider, no url configured.');
    }
},
```

这里简单，有 url 的话，二话不说，就开始调用 TaskManager 开始执行任务。回调函数还是 onData 方法，比 RemotingProvider 对象的简单些，success 为 true 时，创建事件后，只是触发 data 事件。发生错误时，不用再重试，直接创建 ExceptionEvent 对象实例，并触发 data 事件。

17.3　配置 Ext.Direct 的使用环境

17.3.1　概述

要使用 Ext.Direct，需要先配置路由包，这是必须做的事情，不然页面会找不到 API，也不知道如何发送请求。而每个路由包的配置大相径庭，所以下载后一定要详细阅读其说明文档，养成这个好习惯。

下面将分两节介绍如何配置本书示例所需要的 Ext.Direct 的使用环境。

17.3.2　.NET 环境的配置

将 Router 1.0 包中的文件提取出来后，将 Ext.Direct\Ext.Direct\bin\Release 目录下的两个 dll 文件复制到 ASP.NET 网站的 bin 目录下。这里一定要注意，Newtonsoft.Json.dll 文件建议使用同目录下的文件，因为不同版本的文件 Ext.Direct.dll 会产生错误。

接着是创建 API 文件，在根目录创建一个名称为 Api.ashx 的一般处理程序，先加入对

Ext.Direct 的引用：

```
using Ext.Direct;
```

接着把类 Api 的继承由 IHttpHandler 修改为 DirectHandler，把原来的代码全部删除，然后加入以下代码：

```
public class Api : DirectHandler
{
    public override string ProviderName
    {
        get
        {
            return "Ext.app.REMOTING_API";
        }
    }

    public override string Namespace
    {
        get
        {
            return "Ext.app";
        }
    }

    protected override void ConfigureProvider(DirectProvider provider)
    {
        this.Configure(provider, new object[] { /* 在此加入对象实例 */});
    }
```

代码中，ProviderName 方法中返回的是将要在 addProvider 方法中加入的提供者名称。Namespace 则是调用对象的命名空间，这里可根据你的项目进行定义，使用“Ext.app”可把自定义的对象统一在 Ext.app 下。ConfigureProvider 方法则会收集实例中要映射到客户的方法，生成 actions 属性里的对象。需要映射到客户端的对象，就要在注释中使用 new 方法创建该对象的实例，例如要把 Test 对象的方法映射到客户端，就要将语句修改为：

```
this.Configure(provider, new object[] { new Test()});
```

如果有多个对象，则用逗号将其分隔，例如要增加映射 Test2 对象，则代码可以这样写：

```
this.Configure(provider, new object[] { new Test(),new Test2()});
```

那么在这些对象中要做什么配置呢？我们做个简单示例来演示一下。

在解决方案的 App_Code 目录下创建两个类，类名分别为 Test 和 Test2。

先打开类 Test，然后添加对 Ext.Direct 的引用。然后在类定义的上方添加属性 DirectAction，代码如下：

```
[DirectAction]
public class Test
```

这就表示 Test 类是一个 Action 类，要映射到客户端。接着为类添加一个不带参数的 SayHi 方法，代码如下：

```
[DirectMethod]
public string SayHi()
{
    return "Hi";
}
```

代码中，DirectMethod 属性表示该方法要映射到客户端，不过有了该属性还不行，还必须定义方法为公共方法，不然也不会映射到客户端。

再定义带一个参数的 SayHello 方法，代码如下：

```
[DirectMethod]
public string SayHello(string name)
{
    return "Hello,"+name;
}
```

再定义一个不带 DirectMethod 属性的 Say 方法，代码如下：

```
public string Say()
{
    return "Say";
}
```

现在使用模板页创建一个名称为 17-1.html 的页面文件，首先在 head 部分加入对 Api.ashx 的引用，代码如下：

```
<script type="text/javascript" src="api.ashx"> </script>
```

然后加入 addProvider 方法，添加提供者，代码如下：

```
Ext.direct.Manager.addProvider(Ext.app.REMOTING_API);
```

“Ext.app.REMOTING_API”正是 Api.ashx 中 ProviderName 方法返回的字符串。

接着加入对 SayHi 和 SayHello 方法的调用：

```
Ext.app.Test.SayHi(function(){console.log(arguments)});
Ext.app.Test.SayHello("张三",function(){console.log(arguments)});
```

注意参数，回调函数也是一样的，在控制台输出回调函数的参数。

在浏览器中打开页面后，将 Firebug 切换到网络面板，展开 Api.ashx，并切换到响应面板，将看到以下数据：

```
Ext.ns('Ext.app.REMOTING_API');Ext.app.REMOTING_API = {"type":"remoting","url":"/
    ext-book3/chapter17/dotnet/api.ashx","namespace":"Ext.app","actions":{"Test":
    [{"name":"SayHi","len":0},{"name":"SayHello","len":1}]}};
```

从数据可以看到，Test2 因为没有定义属性，所以虽然加入到了 ConfigureProvider 方法中，但还是不会映射到客户端，Test 的 Say 方法也没做映射。

再切换回控制台，展开提交的请求，并切换到 Post 面板，将看到如图 17-2 所示的结果。图中，SayHi 和 SayHello 合并到一起提交了，这是因为 RemotingProvider 对象的 enableBuffer 属性默认值为 10，也就是在这时间内，把所有请求合并在一起再提交。提交参数固定为 5 个，数据都在 data 里。

从图中最后两个输出结果可以看到，回调函数的第一个参数为返回的结果，第二个参数为 RemotingEvent 对象实例，这个可单击对象进入 DOM 中看类名。

图 17-2 控制台的显示结果

将标签页切换到响应面板，可看到返回的数据为：

```
[{"type":"rpc","tid":1,"action":"Test","method":"SayHi","result":"Hi"},{"type":"r
pc","tid":2,"action":"Test","method":"SayHello","result":"Hello,张三 "}]
```

从数据可以看到，所有返回结果都是在 result 属性里，因而要找数据，得从 reuslt 属性里找。

如果提交的数据是表单，为方法添加 DirectMethodForm 属性，参数可以为 HttpRequest 对象，例如在 Test 类中添加一个接收表单形式数据的 Save 方法，可定义如下：

```
[DirectMethodForm]
public JObject Save(HttpRequest request)
{
    return new JObject();
}
```

代码可以直接返回 JSON 对象，路由会将其转换为字符串再返回。

刷新一下页面，可以看到 Save 方法映射数据为：

```
{"name":"Save","len":1,"formHandler":true}
```

这里多了一个 formHandler 属性，表示这方法是接收表单数据的。

以上就是 Router 的环境配置，不算太复杂。

17.3.3 Java 环境的配置

把下载的 directjngine.2.0-alpha1.zip 文件解压后，首先把以下文件复制到应用的 WEB-

INF/lib 目录：

- deliverables 目录的 directjngine.2.0-alpha1.jar 文件，该文件可能随着版本的更新，文件名会不同，不过肯定是 directjngine.xxx.jar 方式的文件，其中 xxx 为版本号。
- lib 目录下所有 jar 文件，不包括子目录。
- lib/runtimeonly 目录下所有 jar 文件。

接着要做的是在 web.xml 里配置 DirectJNgine 的 servlet。打开 WebContent\WEB-INF\web.xml 文件，然后加入以下配置：

```
<!-- DirectJNgine servlet -->
 <servlet>
        <servlet-name>DjnServlet</servlet-name>
        <servlet-class>
                com.softwarementors.Ext JS.djn.servlet.DirectJNgineServlet
        </servlet-class>
        <init-param>
        <param-name>providersUrl</param-name>
        <param-value>djn/directprovider</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
 </servlet>
 <servlet-mapping>
        <servlet-name>DjnServlet</servlet-name>
        <url-pattern>/djn/directprovider/*</url-pattern>
 </servlet-mapping>
```

以上配置如果没有特殊情况，建议不要修改。

下面要配置 API，DirectJNgine 会自动生成 Api.js，所以只需要在 web.xml 文件中配置好文件的路径和文件名就可以了，具体配置代码如下：

```
<init-param>
   <param-name>apis</param-name>
   <param-value>app</param-value>
</init-param>
<init-param>
   <param-name>app.apiFile</param-name>
   <param-value>app/Api.js</param-value>
</init-param>
<init-param>
   <param-name>app.apiNamespace</param-name>
   <param-value>Ext.app</param-value>
</init-param>
<init-param>
   <param-name>app.actionsNamespace</param-name>
   <param-value>Ext.app</param-value>
</init-param>
```

代码中，4 个参数的作用如下：

- apis：设置 API 的名称，如果有多个，可用逗号分隔，名称不允许重复。
- xxx.apiFile：定义 API 的路径和文件名，其中 xxx 表示 apis 中定义 API 的名称。
- xxx.apiNamespace：定义 addProvider 方法添加的提供者的命名空间，其中，xxx 表示

apis 中定义 API 的名称。例如示例代码定义了命名空间为 Ext.app，那么提供者的定义就是 Ext.app.REMOTING_API。

- xxx.actionsNamespace：定义 actions 的命名空间，其中，xxx 表示 apis 中定义 API 的名称。例如示例代码中，定义了命名空间为 Ext.app，那么要访问 Test 对象的 SayHi 方法要这样写：

```
Ext.app.Test.SayHi();
```

下一步要做的是将服务器端的对象及其方法映射到客户端，这就要使用到 xxx.classes 配置，其中，xxx 表示 apis 中定义 API 的名称。其值为由逗号分隔的类名，例如要映射缺省包的 Test 类和 test2 包的 Test2 类，代码如下：

```
<init-param>
   <param-name>app.classes</param-name>
   <param-value>
          Test,test2.Test
   </param-value>
</init-param>
```

这样还不能将对象及其方法映射到客户端，还需要在定义对象时加入属性。例如，在项目中创建 Test 类，要将其映射到客户端，需要这样写：

```
@DirectAction(action="Test")
public class Test {
```

代码中，“@DirectActio”表示要将该类映射到客户端，而 action 的值表示映射的对象名称为“Test”。接着定义一个不带参数的 SayHi 方法，代码如下：

```
@DirectMethod
public String SayHi(){
   return "Hi";
}
```

代码中，“@DirectMethod”表示要将该方法映射到客户端。

再定义一个带一个参数的 SayHello 方法，代码如下：

```
@DirectMethod
public String SayHello(String name){
   return "Hello,"+name;
}
```

最后定义一个不映射到客户端的 Say 方法，代码如下：

```
public String Say(){
   return "Say";
}
```

现在创建一个在 test2 包的 Test2 对象。

好了，现在使用模板页创建一个名称为 17-1.html 的页面文件，来测试一下刚才创建的方法。

首先要做的是把 web.xml 中定义的 API 文件加入页面 head 标记中，代码如下：

```
<script type="text/javascript" src="app/Api.js"> </script>
```

这里一定要注意路径与定义的路径要匹配，不然找不到文件。
接着是添加提供者，代码如下：

```
Ext.direct.Manager.addProvider(Ext.app.REMOTING_API);
```

注意提供者名称要与生成的匹配。
接着调用 SayHi 和 SayHello 方法，代码如下：

```
Ext.app.Test.SayHi(function(result){console.log(arguments)});
Ext.app.Test.SayHello("张三",function(result){console.log(arguments)});
```

在浏览器中打开页面文件，首先将 Firebug 面板切换到网络面板，展开 Api.js，在响应面板可以看到以下输出：

```
Ext.namespace("Ext.app");
Ext.namespace("Ext.app");
Ext.app.PROVIDER_BASE_URL=window.location.protocol+"//"+window.location.
    host+"/"+(window.location.pathname.split("/").length>2?window.location.
    pathname.split("/")[1]+"/":"")+"djn/directprovider";
Ext.app.POLLING_URLS={};
Ext.app.REMOTING_API={url:Ext.app.PROVIDER_BASE_URL,type:"remoting",namespace:Ex
    t.app,actions:{Test:[{name:"SayHello",len:1,formHandler:false},{name:"SayHi",
    len:0,formHandler:false}]}};
```

这就是 DirectJNgine 自动生成 Api.js 的内容。为什么会有两个一样的命名空间定义呢？这是因为 apiNamespace 和 actionsNamespace 定义了相同的命名空间，每一个都会生成一个定义，因而就有两个一样的定义了。

Ext.app.PROVIDER_BASE_URL 定义的是路由的地址，注意最后的“djn/directprovider”，就是 web.xml 中 providersUrl 变量定义的值，这个一般不需要修改。

Ext.app.REMOTING_API 就是要添加的提供者，其中的“Ext.app”就是 apiNamespace 定义的值，这个要注意。而在对象中，namespace 属性的值是 actionsNamespace 定义的值。

在 actions 属性中，可以看到，“Test”就是“@DirectAction(action="Test")”定义中的 action 的值。没有定义“@DirectMethod”属性的 Say 方法或没有定义“@DirectAction”属性的 Test2 对象就都不会被映射到客户端。

将面板切换回控制台面板，展开提交的地址，并切换到 Post 面板，将看到如图 17-2 的结果（地址会不同）。图中，SayHi 和 SayHello 会合并到一起提交，这是因为 RemotingProvider 对象的 enableBuffer 属性默认值为 10，也就是会在这时间内，把所有请求合并一起再提交。提交参数固定为 5 个，数据都在 data 里。

从图中最后两个输出结果可以看到，回调函数的第一个参数为返回的结果，第二个参数为 RemotingEvent 对象示例，可单击对象进入 DOM 中可看类名。

将标签页切换到响应面板，可看到返回的数据为：

```
[{"type":"rpc","tid":1,"action":"Test","method":"SayHi","result":"Hi"},{"type":"r
    pc","tid":2,"action":"Test","method":"SayHello","result":"Hello, 张三 "}]
```

从数据可以看到，所有返回结果都是在result属性里，因而如果要找数据，则要从reuslt属性里找。

如果提交的数据是表单，为方法添加“@DirectFormPostMethod”属性，参数为HttpServletRequest对象就行了，例如在Test类中添加一个接收表单形式数据的Save方法，可定义如下：

```
@DirectFormPostMethod
public string Save(Map<String, String> formParameters,
        Map<String, FileItem> fileFields){
    return "";
}
```

注意参数，第一个参数formParameters可直接获取表单提交的数据。如果是上传文件，可通过第二个参数获取。

刷新一下页面，可以看到Save方法映射数据为：

```
{"name":"Save","len":1,"formHandler":true}
```

这里的formHandler属性为true，表示这方法是接收表单数据的。

以上只是DirectJNgine的其中一种配置方法，还可以不使用web.xml文件，直接使用注册配置类进行注册，在用户手册里有说明。在用户手册里，还有一些其他的配置项及优化策略，希望读者能仔细阅读一下，在此就不详细解释了。

17.4 使用Ext.Direct

17.4.1 概述

使用Ext.Direct，重点关注的是服务器端方法接收参数的数据类型和返回参数的数据类型，不同的路由库，设计思想的不同，会有很大的不同，因而在使用路由前，必须先搞清楚这些，例如，本书中用到的.NET路由库，在接收Store的CURD请求时，服务器端方法的默认参数的类型是“Dictionary<string,object>”。如果在定义方法时设置属性ParseAsJson，那么接收参数的类型就可以是JArray或JObject，而数据的返回格式可以是任何类型。基于Java的DirectJNgine是以“JsonArray”作为接收参数类型的，其返回的数据类型必须是集合或对象类数据，不能是JsonObject。更重要的问题是很多的库在这方面都没有具体的说明文档，完全要靠自己去摸索，这是使用Ext.Direct存在的最大问题。

在客户端，DirectStore只是将DiretProxy的配置项放到Store这层进行定义，因而，建议尽量不要使用DirectStore，而是在Store中定义proxy配置项，这样的好处是使定义更清晰，代码更统一、易读。

17.4.2 使用DirectProxy及进行CURD操作

Ext JS 4与Ext JS 3的DirectProxy主要区别是在DirectProxy提交参数上做了调整，在

Ext JS 3，需要为提交参数定义 paramOrder 配置项，来确定参数的顺序，这对应分页等参数多的情况，相当的麻烦，在 Ext JS 4 中对此做了改进，全部参数都放到一个对象里提交了，该对象包含了 extraParams 配置项定义的参数，因而方法的参数就可统一为 Dictionary 等集合类型了，不用为参数顺序，尤其是带有额外参数的顺序而头疼了。

下面将 10.7.3 节示例修改成使用 DirectProxy 提交数据，练习一下如何使用 DirectProxy 及进行 CURD 操作。使用模板也创建一个名称为 17-2.html 的页面文件，然后将 10.7.3 节示例的代码复制过来。

要修改的地方不多，首先是加入 API 的引用：

C#

```
<script type="text/javascript" src="api.ashx"> </script>
```

Java

```
<script type="text/javascript" src="app/Api.js"> </script>
```

还要添加增加提供者的代码：

```
Ext.direct.Manager.addProvider(Ext.app.REMOTING_API);
```

以上代码，是使用 Ext.Direct 必须加的，本章后续示例就不再赘述了。还有就是每添加一个映射类，都要修改 Api.ashx 或 web.xml 文件，将该类加到定义中，具体说明可参考配置使用环境章节的内容。

最后修改 Store 中 proxy 定义如下：

```
proxy:{
    type:"direct",
    batchActions:false,
    //directFn:Ext.app.Product.List,
    api:{
            read:Ext.app.Product.List,
            create:Ext.app.Product.Add,
            update:Ext.app.Product.Edit,
            destroy:Ext.app.Product.Delete
    },
    reader:{
            type:"json",
            root:"data"
    }
}
```

代码中，batchActions 配置项是根据服务器端方法接收参数的处理情况而设置的。其默认值为 true，会把相同操作的数据合并在一个数组里提交，如果路由库的接收参数能接收数据组成的数组，则可使用该方法提交。DirectJNgine 是支持该方式的，而遗憾的是，.NET 库不支持，只能处理单个数据，因而要设置 batchActions 为 false。

配置项 directFn 相当于 api 中的 read 配置项，在仅需读取数据的情况下使用。

配置项 reader 与使用 url 的配置没什么区别。Writer 对象的配置项不建议做修改，尤其是

不要进行 encode，不然就没提交参数了。

这样，客户端就修改完成了，要写服务器端代码，与以前习惯的方式主要区别是参数不再是从 HTTP 请求对象获取数据了，而是接收经过路由处理好以后的数据。返回数据格式也要注意，例如本示例的 Grid 数据，千万别返回字符串数据，不然 Reader 读不出数据，因为 result 的结果是字符串，而不是 JSON 对象，读不出数据。

首先完成 List 方法，创建一个名称为 Product 的类，加入 List 方法，代码如下：

C#

```
[DirectMethod]
public JObject List(Dictionary<string,Object> data)
{
    int page = 1;
    if (data.ContainsKey("page"))
    {
        int.TryParse(data["page"].ToString(), out page);
    }
    int start = 0;
    if (data.ContainsKey("start"))
    {
        int.TryParse(data["start"].ToString(), out start);
    }
    int limit = 20;
    if (data.ContainsKey("limit"))
    {
        int.TryParse(data["limit"].ToString(), out limit);
    }
    string sort = "it.ProductID";
    string filter = "true";
    if (data.ContainsKey("sort"))
    {
        sort = "";
        object[] b = (object [])data["sort"];
        for (int i = 0; i < b.Count(); i++)
        {
            Dictionary<string, object> c = (Dictionary<string, object>)b[i];
            sort += "it." + c["property"].ToString() + " " + c["direction"].
                ToString() + ",";
        }
        sort = sort.Substring(0, sort.Length - 1);
    }
    if (data.ContainsKey("filter"))
    {
        filter = "";
        object[] b = (object[])data["filter"];
        for (int i = 0; i < b.Count(); i++)
        {
            Dictionary<string, object> c = (Dictionary<string, object>)b[i];
            filter += "it." + c["property"].ToString() + " LIKE '%" + c["value"].
                ToString() + "%',";
        }
        filter = filter.Substring(0, filter.Length - 1);
    }
```

```
using (NorthwindEntities ne = new NorthwindEntities())
{
    int total = ne.Products.Where(filter).Count();
    int a = total;
    if (total < start) start = 0;
    var q = ne.Products.Where(filter).OrderBy(sort).Skip(start).Take(limit).
        Select(m => m).ToList();

    return new JObject
    {
        new JProperty("total",total),
        new JProperty("success",true),
        new JProperty("data",new JArray(
            from p in q
            select new JObject(
                new JProperty("ProductID",p.ProductID),
                new JProperty("ProductName",p.ProductName),
                new JProperty("QuantityPerUnit",p.QuantityPerUnit),
                new JProperty("ReorderLevel",p.ReorderLevel),
                new JProperty("SupplierID",p.SupplierID),
                new JProperty("UnitPrice",p.UnitPrice),
                new JProperty("UnitsInStock",p.UnitsInStock),
                new JProperty("UnitsOnOrder",p.UnitsOnOrder),
                new JProperty("CategoryID",p.CategoryID),
                new JProperty("Discontinued",p.Discontinued)
            )
        ))
    };
}
```

Java

```
@DirectMethod
public Map<String, Object> List(JsonArray data){
   JsonObject joData=(JsonObject)data.get(0).getAsJsonObject();
   int page= joData.has("page")? joData.get("page").getAsInt(): 1;
   int start= joData.has("start")? joData.get("start").getAsInt(): 0;
   int limit= joData.has("limit")? joData.get("limit").getAsInt(): 20;
    String sort="ProductID";
    if(joData.has("sort")){
          sort="";
          JsonArray jaArray=(JsonArray)joData.get("sort").getAsJsonArray();
       for (JsonElement je : jaArray) {
          JsonObject jo = je.getAsJsonObject();
          sort+=jo.get("property").getAsString()+ " "
                  + jo.get("direction").getAsString()+",";
    }
    sort=sort.substring(0,sort.length()-1);
}
String filter="";
if(joData.has("filter")){
       filter="";
       JsonArray jaArray=(JsonArray)joData.get("filter").getAsJsonArray();
    for (JsonElement je : jaArray) {
```

```
            JsonObject jo = je.getAsJsonObject();
            filter+=jo.get("property").getAsString()+ " like '%"
                    + jo.get("value").getAsString()+"%',";
        }
        filter=filter.substring(0,filter.length()-1);
}
String connectionUrl = "jdbc:sqlserver://192.168.0.254:1433;" +
"databaseName=Northwind;;user=sa;password=abcd-1234";
Connection con = null;
Statement stmt = null;
ResultSet rs = null;
ResultSet rscount=null;
Map<String, Object> result = new HashMap<String, Object>();
try {

      Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
      con = DriverManager.getConnection(connectionUrl);

      int count = 0;
      stmt = con.createStatement();
      rscount=stmt.executeQuery("select  count(ProductID) as  count  from
          products"+
                  (filter.length()>0 ? " where " + filter : ""));
      rscount.next();
      count=rscount.getInt(1);
      if(start>count){
          start=0;
      }
      StringBuilder sb = new StringBuilder();
      sb.append("select top 20 * from products where ProductID ");
      sb.append(" not in (select top ");
      sb.append(start);
      sb.append(" ProductID from products  ");
      if(filter.length()>0){
          sb.append(" where ");
          sb.append(filter);
      }
      sb.append(" order by ");
      sb.append(sort);
      sb.append(")");
      if(filter.length()>0){
          sb.append(" and ");
          sb.append(filter);
      }
      sb.append("  order by ");
      sb.append(sort);

      rscount.close();
      rs = stmt.executeQuery(sb.toString());

      //返回JOSN格式数据
      ArrayList<Object> array =new ArrayList<Object>();
      while (rs.next()) {
          Map<String, Object> obj= new HashMap<String, Object>();
          obj.put("ProductID", rs.getInt("ProductID"));
```

```
            obj.put("ProductName", rs.getString("ProductName"));
            obj.put("SupplierID", rs.getInt("SupplierID"));
            obj.put("CategoryID", rs.getInt("CategoryID"));
            obj.put("QuantityPerUnit", rs.getString("QuantityPerUnit"));
            obj.put("UnitPrice", rs.getInt("UnitPrice"));
            obj.put("UnitsInStock", rs.getInt("UnitsInStock"));
            obj.put("UnitsOnOrder", rs.getInt("UnitsOnOrder"));
            obj.put("ReorderLevel", rs.getInt("ReorderLevel"));
            obj.put("Discontinued", rs.getBoolean("Discontinued"));
            array.add(obj);
        }
            result.put("total", count);
            result.put("success", true);
            result.put("data", array);

            rs.close();

            return result;
        }
        catch (Exception e) {
            result.put("success", false);
            result.put("msg", e.getMessage());
                    return result;
        }
        finally {
            if (rscount != null) try { rscount.close(); } catch(Exception e) {}
            if (rs != null) try { rs.close(); } catch(Exception e) {}
            if (stmt != null) try { stmt.close(); } catch(Exception e) {}
            if (con != null) try { con.close(); } catch(Exception e) {}
                }

        }
```

在 C# 中，参数返回的数据其实是 data 数组里的第一对象，因而参数的数据类型是“Dictionary<string,Object>”。排序信息和过滤信息都是数组，因而需要再转换一下。也可以通过附加 ParseAsJson 属性接收 JObject 对象。返回格式倒是很方便，直接返回 JObject 对象就可以了，和前面的示例的没太大区别。

Java 中，参数返回的是 data 提交的数组，所以要先从数组中提取参数对象，再提取参数，这个处理起来就方便多了。但缺点是不能直接返回 JsonObject，必须返回能构建出 JSON 格式的数据格式，这里使用了“Map<String, Object>”类型和 JsonObject 的处理区别不大。

现在完成 Add 方法，Store 的 CRUD 要求返回完整数据，这里要注意，接收的参数与 List 方法是一样的，代码如下：

C#

```
[DirectMethod]
public JObject Add(Dictionary<string,object> data)
{
    JObject jo = new JObject();
    int pid=999;
    int sid=0;
    int.TryParse(data["SupplierID"].ToString(), out sid);
```

```
int cid=0;
int.TryParse(data["CategoryID"].ToString(), out cid);
int unitPrice=0;
int.TryParse(data["UnitPrice"].ToString(), out unitPrice);
int unitsInStock=0;
int.TryParse(data["UnitsInStock"].ToString(), out unitsInStock);
int unitsOnOrder=0;
int.TryParse(data["UnitsOnOrder"].ToString(), out unitsOnOrder);
int reorderLevel = 0;
int.TryParse(data["ReorderLevel"].ToString(), out reorderLevel);
bool discontinued = false;
bool.TryParse(data["Discontinued"].ToString(), out discontinued);
//插入数据库操作代码
return new JObject(
    new JProperty("data",new JArray(
        new JObject(
        new JProperty("ProductID", pid),
        new JProperty("ProductName", data["ProductName"].ToString()),
        new JProperty("QuantityPerUnit", data["QuantityPerUnit"].ToString()),
        new JProperty("ReorderLevel", reorderLevel),
        new JProperty("SupplierID", sid),
        new JProperty("UnitPrice", unitPrice),
        new JProperty("UnitsInStock", unitsInStock),
        new JProperty("UnitsOnOrder", unitsOnOrder),
        new JProperty("CategoryID", cid),
        new JProperty("Discontinued", discontinued)
))));
}
```

Java

```
@DirectMethod
public Map<String, Object> Add(JsonArray data){
   Map<String, Object> result = new HashMap<String, Object>();
   JsonObject joData=(JsonObject)data.get(0).getAsJsonObject();
   int pid= 999;
   int sid= joData.has("SupplierID")? joData.get("SupplierID").getAsInt(): 0;
   int cid= joData.has("CategoryID")? joData.get("CategoryID").getAsInt(): 0;
   int unitPrice= joData.has("UnitPrice")? joData.get("UnitPrice").getAsInt(): 0;
   int unitsInStock= joData.has("UnitsInStock")? joData.get("UnitsInStock").
       getAsInt(): 0;
   int unitsOnOrder= joData.has("UnitsOnOrder")? joData.get("UnitsOnOrder").
       getAsInt(): 0;
   int reorderLevel= joData.has("ReorderLevel")? joData.get("ReorderLevel").
       getAsInt(): 0;
    boolean discontinued = joData.has("Discontinued")? joData.get("Discontinued").
       getAsBoolean(): false;
String productName = joData.has("ProductName")? joData.get("ProductName").
       getAsString(): "";
String quantityPerUnit = joData.has("QuantityPerUnit")? joData.
       get("QuantityPerUnit").getAsString(): "";
//插入数据库操作代码
ArrayList<Object> array =new ArrayList<Object>();
Map<String, Object> obj= new HashMap<String, Object>();
obj.put("ProductID", pid);
```

```
            obj.put("ProductName", productName);
            obj.put("SupplierID", sid);
            obj.put("CategoryID", cid);
            obj.put("QuantityPerUnit", quantityPerUnit);
            obj.put("UnitPrice", unitPrice);
            obj.put("UnitsInStock", unitsInStock);
            obj.put("UnitsOnOrder", unitsOnOrder);
            obj.put("ReorderLevel", reorderLevel);
            obj.put("Discontinued", discontinued);
            array.add(obj);
        result.put("data", array);

        return result;
    }
```

C# 返回的就是数据对象，因而可直接读取数据，而 Java 返回的还是数组，要先提取数据对象，再提取数据。然后就把数据写进数据库（参见代码中的注释部分），最后构建返回对象返回。

Edit 方法和 Delete 方法与 Add 方法大同小异，在此就不列出代码了。

至此，页面就完成了，在浏览器中打开页面，显示结果与 10.7.3 节示例的一样。这里要注意使用 Firebug 观测提交参数与 10.7.3 节示例的区别。没有排序和过滤的时候提交的参数是：

```
{"action":"Product","method":"List","data":[{"page":1,"start":0,"limit":20}],"typ
    e":"rpc","tid":1}
```

标准的 Ext.Direct 提交方式是数据都在 data 的数组中。C# 方法接收的是数组内的对象，而 Java 则是将整个数组作为参数。不同的库会有不同，在使用的时候要注意。

带查询的提交数据如下：

```
{"action":"Product","method":"List","data":[{"page":1,"start":0,"limit":20,"sort"
    :[{"property":"ProductName","direction":"ASC"}]}],"type":"rpc","tid":2}
```

“sort”只是对象的一个属性而已，要取得排序信息，需要从数组中提取对象。filter 和 group 的处理和 sort 是一样的。

下面在命令行中执行以下代码：

```
var store=Ext.StoreManager.lookup("ProductStore");
store.add({
    ProductName:" 电脑 ",
    SupplierID:1,
    CategoryID:1,
    QuantityPerUnit:" 台 ",
    UnitPrice:3000,
    UnitsInStock:10,
    UnitsOnOrder:10,
    ReorderLevel:0,
    Discontinued:false
})
store.sync();
```

这会添加一个记录并同步到服务器，其提交参数是：

```
{"action":"Product","method":"Add","data":[{"ProductID":0,"ProductName":"\u7535\u
    8111","SupplierID":1,"CategoryID":1,"QuantityPerUnit":"\u53f0","UnitPrice":30
    00,"UnitsInStock":10,"UnitsOnOrder":10,"ReorderLevel":0,"Discontinued":false}
    ],"type":"rpc","tid":3}
```

可以看到，数据对象包含了所有字段的值，直接提取值就可以了。

数据的返回格式如下：

```
[{"type":"rpc","tid":3,"action":"Product","method":"Add","result":{"data":[{"Prod
    uctID":999,"ProductName":"电脑","QuantityPerUnit":"台","ReorderLevel":0,"Supp
    lierID":1,"UnitPrice":3000,"UnitsInStock":10,"UnitsOnOrder":10,"CategoryID":1
    ,"Discontinued":false}]}}]
```

与列表的返回格式相比，这里少了 success 属性和 total 属性，具体数据格式则是一样的。

有兴趣的读者可以继续使用 set 方法修改一下字段值，然后利用 removeAt 方法删除记录，接着再执行同步操作，观察一下数据提交情况。这是一个很好的学习处理流程的方法，要充分利用。

17.4.3 使用 Ext.Direct 实现树的动态加载及节点维护

在 11.3.2 节示例中已经实现过了树的动态加载和节点维护，现在尝试将它修改为使用 Ext.Direct 实现的。创建一个名称为 17-3.html 的页面文件，然后将 11.3.2 节示例的代码复制过来的。

与上一节一样，只需要修改 proxy 配置项就行了，修改代码如下：

```
proxy: {
    type: 'direct',
    batchActions:false,
    api:{
            read:Ext.app.Tree.List,
            create:Ext.app.Tree.Add,
            destroy:Ext.app.Tree.Delete,
            update:Ext.app.Tree.Edit
    },
    reader:{
            messageProperty:"msg"
    }
}
```

主要的修改是添加了 type 和 api 中的各配置项的值，还添加了 batchActions 配置项。

不过这里要注意，因为 proxy 是定义在模型中的，而模型是在 onReady 函数外的，所以 addProvider 方法必须定义在模型前，不然会提示找不到提供者。

现在来完成服务端代码，要创建一个名称为 Tree 的类，先创建 List 方法，代码如下：

C#

```
[DirectMethod]
public JArray List(Dictionary<string,object> data)
{
    string node="";
```

```
if (data.ContainsKey("node"))
{
    node = data["node"].ToString();
    if (node == "-1" || node == "")
    {
        node = "it.ParentID=0";
    }
    else
    {
        node = "it.ParentID=" + node + "";
    }
    }
    using (NorthwindEntities ne = new NorthwindEntities())
    {
        var q = ne.TreeTest.Where(node);
        JArray ja = new JArray();
        foreach (var c in q)
        {
            ja.Add(
                new JObject(
                new JProperty("id", c.ID),
                new JProperty("parentId", c.ParentID == 0 ? -1 : c.ParentID),
                new JProperty("text", c.Text)
            ));
        }
        return ja;

    }
}
```

Java

```
@DirectMethod
public ArrayList<Node> List(JsonArray data){
   ArrayList<Node> array = new ArrayList<Node>();
   JsonObject joData=(JsonObject)data.get(0).getAsJsonObject();
   int id=joData.has("node")? joData.get("node").getAsInt():0;
   if(id==-1)id=0;
    String connectionUrl = "jdbc:sqlserver://192.168.0.254:1433;" +
    "databaseName=Northwind;;user=sa;password=abcd-1234";
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;
    try {
            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
            con = DriverManager.getConnection(connectionUrl);
    stmt = con.createStatement();
    rs = stmt.executeQuery("select * from TreeTest where ParentID=" + String.
        valueOf(id));

//返回 JOSN 格式数据
    int parentid=0;
    while (rs.next()) {
            Node obj= new Node();
            obj.id=rs.getInt("ID");
```

```
                obj.text=rs.getString("Text").trim();
                parentid=rs.getInt("ParentID");
                obj.parentId =  parentid==0?-1:parentid;
                array.add(obj);
            }

            rs.close();
            return array;

        }
        catch (Exception e) {
                throw new RuntimeException(e.getMessage());
        }
        finally {
             if (rs != null) try { rs.close(); } catch(Exception e) {}
             if (stmt != null) try { stmt.close(); } catch(Exception e) {}
             if (con != null) try { con.close(); } catch(Exception e) {}
         }
    }
```

因为树要求返回数组格式数据，所以在 C# 中，返回的是 JArray 类型数据，而 Java 不允许返回 JsonArray，只能以“ArrayList<Node>”代替。因为 C# 接收参数不是 JSON 对象，所以将 Dictionary 转为 Node 对象又多一层操作，直接取值就好了。因为 Java 接收的是 JsonArray，所以使用 Node 对象比较方便，在生成 ArrayList 时，也派上了用场。

因为路由中有错误处理机制，发生错误时会返回错误事件，所以在发生错误时会抛出一个异常交给路由处理并返回正确的做法。

接着完成 Add 方法，代码如下：

C#

```
[DirectMethod]
public JArray Add(Dictionary<string, object> data)
{
    int parentId = 0;
    int.TryParse(data["parentId"].ToString(), out parentId);
    int depth = 0;
    int.TryParse(data["depth"].ToString(), out depth);
    int index = 0;
    int.TryParse(data["index"].ToString(), out index);

    string text = "新节点";
    JArray ja = new JArray();
    try
    {
    using (NorthwindEntities ne = new NorthwindEntities())
    {
        NorthwindModel.TreeTest d = new NorthwindModel.TreeTest
        {

            Text = text,
            ParentID = parentId == -1 ? 0 : parentId
        };
        ne.AddToTreeTest(d);
```

```
            ne.SaveChanges();
            ja.Add(new JObject(
                new JProperty("id", d.ID),
                new JProperty("parentId", parentId),
                new JProperty("text", text),
                new JProperty("depth", depth),
                new JProperty("index", index),
                new JProperty("checked", null)
            ));
        }
        catch (Exception e)
        {
            throw new DirectException(e.Message);
        }
        return ja;
    }
```

Java

```
@DirectMethod
public ArrayList<Node> Add(JsonArray data){
   ArrayList<Node> array = new ArrayList<Node>();
   Node[] nodes=ProcessData(data);
   if(nodes.length>0){
        String connectionUrl = "jdbc:sqlserver://192.168.0.254:1433;" +
        "databaseName=Northwind;;user=sa;password=abcd-1234";
       Connection con = null;
       CallableStatement callstmt=null;
       ResultSet rs = null;
       try {
            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
            con = DriverManager.getConnection(connectionUrl);
            int iUpdCount =0;
            boolean bMoreResults = true;
            callstmt=con.prepareCall("INSERT INTO TreeTest (Text,Parentid)
                VALUES (?,?);SELECT @@IDENTITY;");
            for (int i = 0; i < nodes.length; i++) {
            Node t= nodes[i];
            callstmt.setNString(1, t.text);
            callstmt.setInt(2, t.parentId==-1?0:t.parentId);
            callstmt.execute();
            iUpdCount=callstmt.getUpdateCount();
            bMoreResults=true;
            rs=null;
                while (bMoreResults || iUpdCount!=-1)
            {
                rs = callstmt.getResultSet();
                if (rs != null)
                {
                    rs.next();
                    nodes[i].id = rs.getInt(1);
                }
                bMoreResults = callstmt.getMoreResults();
                iUpdCount = callstmt.getUpdateCount();
```

```
                    }
                array.add(nodes[i]);
            }
             return array;
}
catch (Exception e) {
            throw new RuntimeException(e.getMessage());
}
finally {
    if (rs != null) try { rs.close(); } catch(Exception e) {}
    if (callstmt != null) try { callstmt.close(); } catch(Exception e) {}
        if (con != null) try { con.close(); } catch(Exception e) {}
    }
}
return array;
}
```

C# 在这里不能转对象，所以会麻烦点，要将全部参数都取一次，将其添加后返回。

Java 的代码则可以处理批量提交的数据，通过 ProcessData 方法可将返回的 JSON 数组转换为 Node 对象实例，然后再将其添加。返回的数据类型也是 ArrayList。

Edit 方法、Delete 方法与 Add 方法区别不大，在此就不列出来了，请读者自己看示例源代码。

在浏览器中打开页面，所能做的操作与 11.3.2 节示例没有区别。

17.4.4 使用 DirectLoad 为表单加载数据

使用 DirectLoad 的重点是要定义 paramOrder 配置项设置参数顺序，一般情况下只有 id 一个参数，但也必须进行设置，否则就和方法的参数对应不上。下面将 12.5.2 节示例修改为使用 DirectLoad 加载数据。

使用模板页创建一个名称为 17-4.html 文件，然后把 12.5.2 节示例的代码复制过来，把与 XML 有关的 Store 和表单面板删除。

修改的重点还是 proxy 配置项，先改 Product 模型的 proxy 配置项，修改后代码如下：

```
proxy:{
    type:"direct",
     directFn:Ext.app.Product.Details,
     reader:{
          type:"json",
          root:"data"
     }
}
```

因为不需要提交，所以使用 directFn 配置项就可以了，别忘了为 reader 加 root 配置项。

接着修改 SupplierStore 的 proxy 配置项，代码如下：

```
proxy:{
    type:"direct",
    directFn:Ext.app.Supplier.ComboList,
    reader:{
            type:"json",
            root:"data"
```

```
    }
}
```

继续修改 CategoryStore 的 proxy 配置项，代码如下：

```
proxy:{
    type:"direct",
    directFn:Ext.app.Category.ComboList,
    reader:{
            type:"json",
            root:"data"
    }
}
```

最后要为表单面板加入 paramOrder 和 api 配置项，以实现加载功能，代码如下：

```
paramOrder:"id",
api:{
    load:Ext.app.Product.Details2
},
```

还要修改一下“使用 load 加载”按钮的代码，把 url 配置项去掉。

现在客户端代码全部完成了，接着开始编写服务器端代码。可能大家已经注意到，使用了模型加载和表单加载两个不同的方法，模型使用的是 Details，而表单使用的是 Details2，这是为什么呢？原因是使用模型获取的数据，参数类型必然是 Dictionary 或 JsonArray，而使用表单获取的数据则是整型，虽然 C# 和 Java 都支持重载，但是路由不识别重载方法，也无法根据参数类型去调用适合的方法，因此这里必须通过两个方法来实现。

先完成 Details 方法，代码如下：

C#

```
[DirectMethod]
public JObject Details(Dictionary<string,object> data)
{
    int id = 0;
    if (data.ContainsKey("id"))
    {
        int.TryParse(data["id"].ToString(), out id);
    }
    if (id > 0)
    {
        using (NorthwindEntities ne = new NorthwindEntities())
        {
try
{
    var q = ne.Products.Single(m => m.ProductID == id);
    return new JObject(
        new JProperty("success", true),
        new JProperty("data",  new JObject(
            new JProperty("ProductID", q.ProductID),
            new JProperty("ProductName", q.ProductName),
            new JProperty("CategoryID", q.CategoryID),
            new JProperty("SupplierID", q.SupplierID),
```

```
                    new JProperty("UnitPrice", q.UnitPrice),
                    new JProperty("UnitsInStock", q.UnitsInStock),
                    new JProperty("UnitsOnOrder", q.UnitsOnOrder),
                    new JProperty("QuantityPerUnit", q.QuantityPerUnit),
                    new JProperty("ReorderLevel", q.ReorderLevel),
                    new JProperty("Discontinued", q.Discontinued),
                    new JProperty("CategoryName", q.Categories.CategoryName),
                    new JProperty("CompanyName", q.Suppliers.CompanyName)
                ))
            );
                }
                catch (Exception ee)
                {
                    throw new DirectException(ee.Message);
                }
            }
        }
        else
        {
            throw new DirectException("错误的产品编号: " + id.ToString());
        }
}
```

Java

```
@DirectMethod
public Map<String, Object> Details(JsonArray data){
    JsonObject joData = data.get(0).getAsJsonObject();
    int id=joData.has("id")?joData.get("id").getAsInt():0;
    if(id>0){
         Map<String, Object> result = new HashMap<String, Object>();
         String connectionUrl = "jdbc:sqlserver://192.168.0.254:1433;" +
         "databaseName=Northwind;;user=sa;password=abcd-1234";
        Connection con = null;
        Statement stmt=null;
        ResultSet rs = null;
        try {
           Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
           con = DriverManager.getConnection(connectionUrl);
           stmt = con.createStatement();

           rs = stmt.executeQuery("SELECT   Products.*, Categories.CategoryName, " +
            " Suppliers.CompanyName FROM Products LEFT OUTER JOIN" +
            " Categories ON Products.CategoryID = Categories.CategoryID LEFT " +
           " OUTER JOIN Suppliers ON Products.SupplierID = Suppliers.SupplierID" +
            " where Products.ProductID =" + String.valueOf(id));

           ArrayList<Object> array=new ArrayList<Object>();
           while (rs.next()) {
               Map<String, Object> obj = new HashMap<String, Object>();
               obj.put("ProductID", rs.getInt("ProductID"));
               obj.put("ProductName", rs.getString("ProductName"));
               obj.put("CategoryID", rs.getInt("CategoryID"));
               obj.put("SupplierID", rs.getInt("SupplierID"));
```

```
            obj.put("UnitPrice", rs.getFloat("UnitPrice"));
            obj.put("UnitsInStock", rs.getInt("UnitsInStock"));
            obj.put("UnitsOnOrder", rs.getInt("UnitsOnOrder"));
            obj.put("QuantityPerUnit", rs.getString("QuantityPerUnit"));
            obj.put("ReorderLevel", rs.getInt("ReorderLevel"));
            obj.put("Discontinued", rs.getString("Discontinued"));
            obj.put("CategoryName", rs.getString("CategoryName"));
            obj.put("CompanyName", rs.getString("CompanyName"));
            array.add(obj);
        }
        rs.close();
            result.put("success", true);
            result.put("data", array);
                return result;
        }
        catch (Exception e) {
                    throw new RuntimeException(e.getMessage());
        }
        finally {
            if (rs != null) try { rs.close(); } catch(Exception e) {}
            if (stmt != null) try { stmt.close(); } catch(Exception e) {}
            if (con != null) try { con.close(); } catch(Exception e) {}
        }
    }else {
            throw new RuntimeException("错误的产品编号: "+String.valueOf(id));
    }
}
```

使用模型读取单个数据的最大麻烦是要把 id 取出来再进行余下操作，Details2 方法则不用如此麻烦，因为参数传递过来的就是 id，可以直接使用。Details2 方法与 Details 方法的最大区别除了参数不同外，就是没有取值部分代码，还要特别注意的是返回格式，Details 方法中的数据要求以数组形式返回，而 Details2 方法返回的必须是对象格式。因为差别不大，Details2 方法这里就不列出来了。

接着创建 Supplie 类，并加入 ComboList 方法，代码如下：

C#

```
[DirectMethod]
public JObject ComboList(Dictionary<string,object> data)
{
    JArray ja = new JArray();
    string query="true";
    if (data.ContainsKey("query"))
    {
        query = string.Format("it.CompanyName like '%{0}%'", data["query"].
        ToString());
}
using (NorthwindEntities ne = new NorthwindEntities())
{
    var q = ne.Suppliers.Where(query).OrderBy(m => m.CompanyName);
    foreach (var c in q)
    {
        ja.Add(new JObject(
```

```
                    new JProperty("SupplierID", c.SupplierID),
                    new JProperty("CompanyName", c.CompanyName)
                ));
            }
        }
        return new JObject(
            new JProperty("success", true),
            new JProperty("data", ja)
        );

    }
```

Java

```
@DirectMethod
public Map<String, Object> ComboList(JsonArray data){
   JsonObject joData = data.get(0).getAsJsonObject();
   String query = joData.has("query")?joData.get("query").getAsString():"";
    String connectionUrl = "jdbc:sqlserver://192.168.0.254:1433;" +
    "databaseName=Northwind;;user=sa;password=abcd-1234";
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;
    try {
          Map<String, Object> resutl=new HashMap<String, Object>();
        Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
        con = DriverManager.getConnection(connectionUrl);
        stmt = con.createStatement();

        rs = stmt.executeQuery("select SupplierID,CompanyName from Suppliers" +
                (query.length()>0 ? " where CompanyName like '%" + query +"%'"
                    : "")
                +" order by CompanyName");

        ArrayList<Object> array=new ArrayList<Object>();
        while (rs.next()) {
           Map<String, Object> obj=new HashMap<String, Object>();
            obj.put("SupplierID", rs.getInt("SupplierID"));
            obj.put("CompanyName", rs.getString("CompanyName"));
            array.add(obj);
        }
        rs.close();
           resutl.put("success", true);
           resutl.put("data", array);
           return resutl;
    }
    catch (Exception e) {
          throw new RuntimeException(e.getMessage());
    }
    finally {
        if (rs != null) try { rs.close(); } catch(Exception e) {}
        if (stmt != null) try { stmt.close(); } catch(Exception e) {}
        if (con != null) try { con.close(); } catch(Exception e) {}
    }

}
```

在代码中要注意，提取 query 的值来过滤数据，提取方法与 List 方法提取 page 或 start 等参数的方法是一样的，格式是固定的。

最后创建 Category 类，并添加 ComboList 方法，代码与 Supplie 类的 ComboList 方法相似，这里就不列出了。

至此，使用 DirectLoad 为表单加载数据的代码就全部完成了，打开页面，注意观察使用模型加载和使用 Load 加载之间数据提交的差别。

以下是使用模型加载时提交的数据：

```
{"action":"Product","method":"Details","data":[{"id":27}],"type":"rpc","tid":3}
```

以下是使用 Load 加载时提交的数据：

```
{"action":"Product","method":"Details2","data":[29],"type":"rpc","tid":4}
```

从提交数据中可以看到，模型加载提交的数据格式基本都固定为 JSON 对象格式的，而 Load 方式是把参数作为单独一个数据项提交，而且没有声明提交的是什么参数，因而这要求服务器端必须清楚知道参数的用途，不然会出错。这样做是因为使用 DirectProxy 提交的数据，有时候参数很多，有时候参数又很少，而且 CURD 操作提交的参数基本是不同的。如果让开发人员定义 paramOrder，那么代码维护起来就很困难了，而且容易出错，如果使用对象方式提交，则简单很多，因为不再需要指定 paramOrder，编写服务器端代码的开发人员，只要根据提交的对象取值就很容易统一接口。但是，又不能完全都以这样的方式运行，因为类似表单加载数据这样的操作，只需要一个 id 参数就行了，维护和使用上并不困难，而且更便捷。

17.4.5　使用 DirectSubmit 提交表单及使用 Session

使用 DirectSubmit 提交表单与使用 Submit 提交表单基本没什么区别，服务器端验证的错误信息返回等都是一样的，最大的不同是路由库不同．方法接收参数的数据类型不同而已。

要使用 Session，C# 需要在 Api.ashx 中加入以下引用：

```
using System.Web.SessionState;
```

然后还要加入对 IRequiresSessionState 的继承，代码如下：

```
public class Api : DirectHandler,IRequiresSessionState
```

在要使用的 Session 的类中，也要加入 System.Web 和 System.Web.SessionState 的引用。

在 Java 中，只需要在方法内加入以下代码就可以使用 Session 了：

```
WebContext context = WebContextManager.get();
HttpSession session=context.getSession();
```

下面通过修改 13.3 节示例的登录对话框演示一下如何使用 DirectSubmit 提交表单和使用 Session。使用模板页创建名称为 17-5.html 的页面文件，然后把 13.3 节示例的代码复制过来。首先为表单面板添加 api 的定义，代码如下：

```
api:{submit:Ext.app.Login.Check},
```

因为是以表单形式提交的，所以不需要定义 paramOrder 配置项。

最后把登录按钮中 submit 方法调用的 url 去掉，这样就完成了客户端代码。

验证码的图片是通过 Ext.Direct 输出的吗？根据 Ext.Direct 的处理机制，这样会很麻烦，因为数据都是以 JSON 格式返回的，如果返回图片，还需要自己从数据中对图片进行读取处理，再绑定到图片上。如果不怕麻烦，读者可以尝试一下。笔者的看法是，没有完美的解决办法，只能根据情况做出适当选择。

现在来完成服务器端代码，创建一个名称为 Login 的类，然后添加 Check 方法，代码如下：

C#

```
[DirectMethodForm]
public JObject Check(HttpRequest request)
{
    string vcode = request.Params["vcode"] ?? "";
    string username = request.Params["username"] ?? "";
    string password = request.Params["password"] ?? "";
    JObject jo = new JObject();
    jo.Add("success", new JValue(true));
    if (HttpContext.Current.Session["vcode"] == null)
    {
        jo.Property("success").Value = new JValue(false);
        jo.Add("errors", new JObject(
            new JProperty("vcode", "错误的验证码。")
        ));
    }
    else
    {
        if (string.Compare(vcode, HttpContext.Current.Session["vcode"].ToString(),
            false) == 0)
        {
            if (username == "admin" && password == "123456")
            {
                jo.Add("msg", new JValue("登录成功"));
            }
            else
            {
                jo.Property("success").Value = new JValue(false);
                jo.Add("errors", new JObject(
                    new JProperty("username", "用户名或密码错误。"),
                    new JProperty("password", "用户名或密码错误。")
                ));
            }
        }
        else
        {
            jo.Property("success").Value = new JValue(false);
            jo.Add("errors", new JObject(
                new JProperty("vcode", "错误的验证码。")
            ));
        }
    }
    return jo;
}
```

Java

```
@DirectFormPostMethod
public Map<String, Object> Check(Map<String, String> formParameters,
            Map<String, FileItem> fileFields){
   String vcode= formParameters.containsKey("vcode")?
                  formParameters.get("vcode"):"";
   String username= formParameters.containsKey("username")?
                  formParameters.get("username"):"";
   String password= formParameters.containsKey("password")?
                  formParameters.get("password"):"";
   WebContext context = WebContextManager.get();
   HttpSession session=context.getSession();
   Map<String, Object> result = new HashMap<String, Object>();
   Map<String, Object> errors = new HashMap<String, Object>();
   if(session.getAttribute("vcode")==null){
          result.put("success", false);
          errors.put("vcode", "错误的验证码。");
          result.put("errors", errors);
   }else{
          if(vcode.equals(session.getAttribute("vcode").toString())){
                 if(username.equals("admin") && password.equals("123456")){
                        result.put("success", true);
                        result.put("msg","登录成功");
                 }else{
                        result.put("success", false);
                        errors.put("username", "用户名或密码错误。");
                        errors.put("password", "用户名或密码错误。");
                        result.put("errors", errors);
                 }
          }else{
                 result.put("success", false);
                 errors.put("vcode", "错误的验证码。");
                 result.put("errors", errors);
          }
   }
   return result;
}
```

C# 的代码与 13.3 节示例的服务器端代码处理上没什么不同，都是从 HttpRequest 对象中获取值。主要区别是，这里需要从 HttpContext 对象的 Current 属性中获取 Session，而不是像 13.3 节示例中一样直接取值了。

在 Java 中，区别是取值不是从 HttpServletRequest 对象中取得，而是从 formParameters 中取得，而且需要加入访问 Session 的两句代码，返回数据的组织与原来也大不相同了。

在浏览器中打开页面，可以看到提交的数据格式如下：

```
extAction  Login
extMethod  Check
extTID     1
extType    rpc
extUpload  false
password   1231321
username   123123
vcode       123456
```

这与之前示例的提交有很大不同，不再是 JSON 格式的数据，而是我们熟悉的提交方式了。Ext.Direct 的参数也包含在数据中，而且是以 ext 开头的，因而要注意，在定义字段名称时，千万别与之发生冲突。

17.4.6 使用 Ext.Direct 上传文件

通过上一节的介绍可以知道，DirectSubmit 提交与普通的表单提交没什么区别，因而在处理文件上传方面也就没太大区别。下面通过修改 12.2 节的第 11 小节示例演示如何使用 Ext.Direct 上传文件。

使用模板页创建一个名称为 17-6.html 的页面文件，然后把 12.2 节第 11 小节示例的代码复制过来。要改的代码只有一句，把 defaults 配置项中的 url 配置项修改为以下语句：

```
api:{submit:Ext.app.Upload.Save},
```

现在来完成服务器端代码，创建名称为 Upload 的类，添加 Save 方法，代码如下：

C#

```
[DirectMethodForm]
public JObject Save(HttpRequest request)
{
    System.Drawing.Image original_image = null;
    System.Drawing.Bitmap final_image = null;
    System.Drawing.Graphics graphic = null;
    try
    {
        HttpPostedFile jpeg_image_upload = request.Files["Filedata"];
        string original_fielname = jpeg_image_upload.FileName.ToLower();
        string extname = original_fielname.Substring(original_fielname.
            LastIndexOf(".") + 1).ToLower();
        if (",jpg,gif,bmp,png,".IndexOf(extname) >= 0)
        {
            Guid guid = Guid.NewGuid();
            string filename = guid.ToString() + "." + extname;
            string path = request.RequestContext.HttpContext.Server.MapPath(".") + "\\";
            jpeg_image_upload.SaveAs(path + filename);
            return new JObject(
                    new JProperty("success", true),
                    new JProperty("msg", filename)
                );

        }
        else
        {
            return new JObject(
                    new JProperty("success", false),
                    new JProperty("errors", new JObject(
                        new JProperty("Filedata", "错误的图片类型。")
                    ))
                );
        }
    }
```

```
catch (Exception ee)
{
    throw new DirectException(ee.Message);
}
finally
{
    if (final_image != null) final_image.Dispose();
    if (graphic != null) graphic.Dispose();
    if (original_image != null) original_image.Dispose();
}
```

Java

```
@DirectFormPostMethod
public Map<String, Object> Save(Map<String, String> formParameters,
          Map<String, FileItem> fileFields) throws Exception{
   UUID uuid =UUID.randomUUID();
   Map<String, Object> result =new HashMap<String, Object>();
   try {
          FileItem fileitem = fileFields.get("Filedata");
          String name = fileitem.getName();
          String extname = name.substring(name.lastIndexOf(".")+1).toLowerCase();
          if(",jpg,gif,png,bmp,".indexOf(","+extname+",")>=0)
          {
                String filename = uuid+"."+extname;
                WebContext context = WebContextManager.get();
                String loadpath= context.getServletContext().getRealPath("/");
                File fNew= new File(loadpath,filename);
                FileOutputStream fos = new FileOutputStream(fNew);
                Streams.copy(fileitem.getInputStream(), fos,true);
                result.put("success", true);
                result.put("msg", filename);
          }else{
                result.put("success", true);
                Map<String, Object> errors = new HashMap<String, Object>();
                errors.put("Filedata", "错误的文件类型。");
                result.put("errors", errors);
          }

   }catch (FileUploadIOException  e) {
          throw (FileUploadException) e.getCause();
   }catch (Exception  e) {
          throw new RuntimeException(e.getMessage());
   }
   return result;
}
```

C# 的代码与示例 12-9 中的代码比较，只修改了取值的对象、获取路径的对象和抛出异常的代码。

对 Java 的修改就不太轻松，使用原来的代码，只修改从 fileFields 中获取 FileItem、获取路径及返回的数据格式，总是抛出异常“Cannot write uploaded file to disk!”，这是为什么呢？

查看文件名、路径都没错。查看 DirectJNgine 的源代码才会发现 DiskFileItem2 对象的 write 方法只抛出异常不干活。出现这样的错误，还不如直接给出 HttpServletRequest 对象，由开发者自行处理来得便利。没办法，只能通过数据流复制的办法将 getInputStream 返回的数据流复制到文件输出的数据流中才解决了这个问题。

还好，这些路由库基本都是开源的，有源代码可以参考。

Ext.Direct 在文件上传中的好处就是不用考虑 ContentType 问题了，路由库会处理。

17.4.7 使用 PollingProvider 对象

PollingProvider 对象适用于实时性比较强的场合，例如聊天、股票信息更新等，在默认情况下，它每 3 秒就会发送一个请求。其配置项主要有以下 3 个。

- ❑ baseParams：定义要提交的参数。
- ❑ interval：定义刷新时间，默认值为 3000。
- ❑ url：轮询的地址。

使用 PollingProvider 对象和自己创建一个定时任务，定时发送请求没什么区别。有些路由库会带有 DirectPollMethod 属性，如 DirectJNgine，有些则没有，比如本书用到的 .NET 版本。有没有 DirectPollMethod 属性其实关系不大，只要按固定格式返回数据即可。对于封装好的 DirectPollMethod 属性库，只要求方法返回数据，没有封装的，则需要全格式返回，返回的数据格式如下：

```
{
    "type":"event",
    "name":" 定义的事件名称 ",
    "data":" 要返回数据 "
}
```

在以上格式中，type 的值是固定不能变的，而在 name 中定义的值，在客户端可根据该值为 Ext.Direct 绑定一个函数处理返回的数据，函数只有一个参数，是 DirectEvent 对象的实例。

下面通过一个示例演示如何使用 PollingProvider 对象，示例中页面会显示两个为 0 的数，这两个值会提交到服务器端，在服务器端，如果随机产生的数是双数，就对第一个数值加 1，name 的值为 E1，如果随机产生的数是单数，就对第二个值加 1，name 的值为 E2，然后将数据返回客户端并刷新显示。

使用模板页创建一个名称为 17-7.html 的页面文件，这里千万别添加 API，因为不需要。如果 Java 使用 DirectPollMethod 属性，则必须添加 API，不过本示例不采用这方式，因为设置比较麻烦，不够直接。首先添加显示数值的 HTML 代码：

```
<div style="padding:20px 0 0 20px;font-size:18px;line-height:30px;">
    值 1：<span id="v1">0</span></br>
    值 2：<span id="v2">0</span>
</div>
```

接着先将两个显示值的 SPAN 转换为 Element 对象实例，刷新显示时要用到，代码如下：

```
var e1=Ext.get("v1");
```

```
var e2=Ext.get("v2");
```

接着添加提供者，代码如下：

```
Ext.direct.Manager.addProvider({
    id:"PollTest",
    type:'polling',
    baseParams:{value1:0,value2:0},
    url:"Poll.ashx" //.Net
    //url:"Poll"  //Java
});
```

设置 id 的目的是要在事件中通过 id 获取提供者，然后获取其 baseParams 中的值刷新显示。baseParams 中有两个值，这里会将其提交到服务器端，也会以对象形式返回，直接修改其值。

现在为 Ext.Direct 绑定事件 E1 和 E2，用于修改 baseParams 的值及刷新显示，代码如下：

```
Ext.Direct.on("E1",function(e){
    var p=Ext.direct.Manager.getProvider("PollTest")
    p.baseParams=e.data;
    e1.dom.innerHTML=p.baseParams.value1;
});

Ext.Direct.on("E2",function(e){
    var p=Ext.direct.Manager.getProvider("PollTest")
    p.baseParams=e.data;
    e2.dom.innerHTML=p.baseParams.value2;
});
```

当服务器端返回的 name 属性的值为 E1 时，会执行第一个事件中的代码，并且会修改“值 1”的显示；当属性值为 E2 时，将会刷新“值 2”的显示。

现在来完成服务器端代码，除了返回格式，没有其他需要特别设置的地方，具体代码如下：

C#

```
public void ProcessRequest (HttpContext context) {
    context.Response.ContentType = "text/javascript";
    int v1 = 0;
    int v2 = 0;
    int.TryParse(context.Request.Params["value1"], out v1);
    int.TryParse(context.Request.Params["value2"], out v2);
    Random ro = new Random(); ;
    string e="E1";
    if (ro.Next(1,10) % 2 == 0)
    {
        v1++;
    }
    else
    {
        v2++;
        e = "E2";
    }
    context.Response.Write(
        new JObject(
```

```
                new JProperty("type", "event"),
                new JProperty("name", e),
                new JProperty("data", new JObject(
                    new JProperty("value1",v1),
                    new JProperty("value2",v2)
                ))
            ).ToString()
        );

    }
```

Java

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
     throws ServletException, IOException {
    int v1=0;
    int v2=0;
    if(request.getParameter("value1") != null){
            v1=Integer.parseInt(request.getParameter("value1"));
    }
    if(request.getParameter("value2") != null){
            v2=Integer.parseInt(request.getParameter("value2"));
    }
    String e="E1";
    Random ro = new Random();
    if(ro.nextInt()%2==0){
            v1++;
    }else{
            v2++;
            e="E2";
    }
    JsonObject j1=new JsonObject();
    j1.addProperty("value1", v1);
    j1.addProperty("value2", v2);
    JsonObject result =new JsonObject();
    result.addProperty("type", "event");
    result.addProperty("name", e);
    result.add("data", j1);
    response.setContentType("text/javascript; charset=utf-8");
     response.getWriter().write(result.toString());

}
```

在以上代码中，在取得两个值后，将随机数与 2 的模数进行运算，变量 e 的值将会作为 name 属性的值，以触发相应的客户端事件。最后根据返回格式构建 JSON 对象后返回。

在浏览器中打开页面，将看到页面的值每隔 3 秒会刷新一次。在控制台会看到很多发送请求，打开第一个请求，会看到提交参数为：

```
value1=0&value2=0
```

而返回值为（值可能不同）：

```
{"type":"event","name":"E2","data":{"value1":0,"value2":1}}
```

这与不使用 Ext.Direct 的处理没太大区别，重点就在可通过返回的 name 值，触发不同的事件。可以专门观察一下返回值后会刷新哪个值的数据，就知道触发的是哪个事件了。

17.4.8　错误处理

在提供者的工作流程中，在 onProviderData 方法中，只有服务器端返回的对象的 type 值为 exception，才会触发 exception 事件。而每当服务器端代码运行错误的时候，路由都会以对象的 type 值为 exception 的形式返回错误，因而可通过监听 exception 事件来进行错误处理。例如在 17.3.2 节示例中，添加以下代码监听 exception 事件：

```
Ext.direct.Manager.on("exception",function(e){
    console.log("Exception:",e);
})
```

然后调用带有错误参数的 SayHi 和 SayHello 方法，代码如下：

```
Ext.app.Test.SayHello(1,function(result){console.log(arguments)});
```

在浏览器中打开 17.3.2 节示例，可看到最后两个输出为：

```
Exception: Object { type="exception", tid=3, action="Test", 更多 ...}
[undefined, Object { type="exception", tid=3, action="Test", 更多 ...}]
```

第一个输出是 exception 事件的输出。单击对象，可在 DOM 面板中看到如图 17-3 所示的内容。在图 17-3 中，该对象为 Exception 对象实例，message 属性包含了服务端返回的错误信息，可用于显示提示信息。

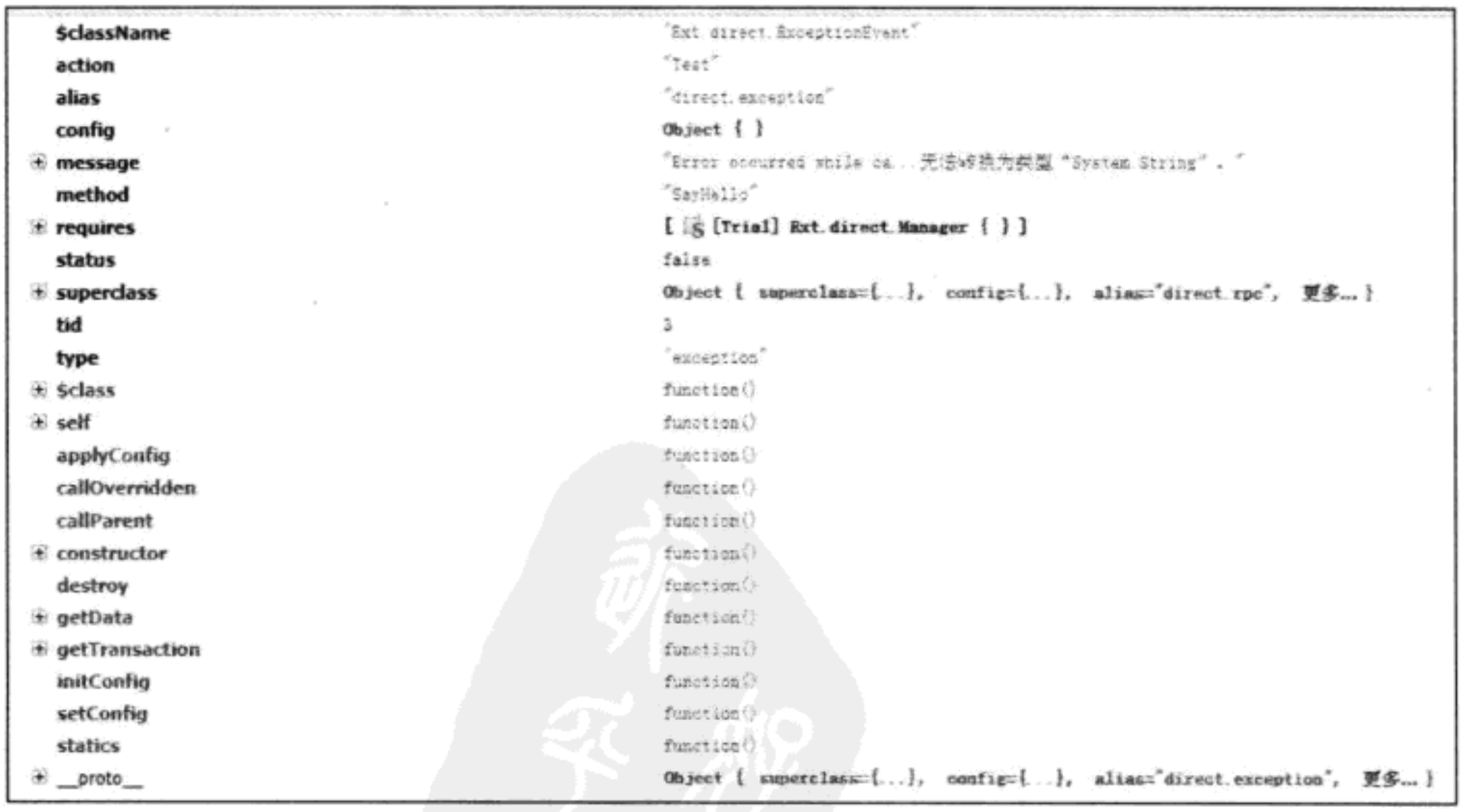


图 17-3　Exception 对象实例

而第二个输出是调用 SayHello 方法时回调函数的输出，所以，通过回调函数也可以对错

误信息进行处理，只要判断第二参数的 type 属性是否为 exception 就可以了。

17.5 本章小结

通过本章的示例可以看到，Ext.Direct 确实可简化通信，不过这太依赖路由库了。路由库的好坏决定了 Ext.Direct 使用是否简单便利，因而在没有找到满意的路由库前，还是多考虑是否值得使用 Ext.Direct。就目前的路由库来说，笔者都不大满意，都有不完善的地方，例如基于 .NET 的，在调试时，缓存功能的使用就很麻烦，在新添一个映射类时，由于没文档说明如何清理缓存，只能通过重启 IIS 的方式刷新。而 Java 的也差不多，需要重启 Tomcat。

不过，路由库是开源的，都有源代码可以参考，有时间的读者也可以根据自己的需要完善其功能。Ext.Direct 还是不错的。

第18章　动画功能

动画功能是框架必不可少的功能之一，它能提供用户体验效果。Ext JS 在这方面当然也不会落后，Ext JS 的动画可根据对象类型的不同而使用不同的配置，从而大大简化了基于动画的定义更方便实用。

本章将讲述 Ext JS 中与动画有关的对象及如何使用这些对象。

18.1　动画功能的构成及工作流程

18.1.1　概述

这里说的动画是指一个目标（target），当其大小、位置、颜色或其他属性改变时，实现一种过渡的效果。在这个过渡的过程中，会通过平滑函数计算出一个值，通过该值计算出目标在某一个时间点内的属性值，然后将这些属性值应用到目标上，从而虚拟出目标在这个过渡过程中的动画效果。

与动画功能有关的类不少，但用于实现动画功能的类是 Anim 对象，其余类都是为 Anim 类提供支持的。表 18-1 列出了所有与动画功能有关的类及其说明。

表 18-1　与动画功能相关的类及其说明

类　名	详 细 说 明
Anim	创建一个动画
Animator	用于定义动画的关键帧
Easing	为动画提供过渡用的功能函数
CubicBezier	提供贝塞尔曲线计算
FxManager	单件模式对象，对所有动画进行跟踪和管理
Queue	混入 FxManager 对象的对象，为动画提供队列功能
Ext.fx.PropertyHandler	为目标提供属性句柄用于处理属性
FxTarget	为动画指定目标
FxElement	派生于 FxTarget 对象，指定动画目标为 Element 对象
FxElementCSS	派生于 FxElement，表示动画目标为 Element 对象且支持 CSS
FxCompositeElement	派生于 FxElement，表示动画目标为 CompositeElement 对象
FxCompositeElementCSS	派生于 FxCompositeElement，表示动画目标为 CompositeElement 对象且支持 CSS
FxComponent	派生于 FxTarget 对象，指定动画目标为组件
FxSprite	派生于 FxTarget 对象，指定动画目标为 DrawSprite 对象
FxCompositeSprite	派生于 FxSprite 对象，指定动画目标为 CompositeSprite 对象

18.1.2 动画功能的工作流程：Ext.fx.Anim

动画功能是从创建 Anim 对象实例开始的，其构造函数的代码如下：

```
constructor: function(config) {
    var me = this,
       curve;
    config = config || {};
    if (config.keyframes) {
        return Ext.create('Ext.fx.Animator', config);
    }
    config = Ext.apply(me, config);
    if (me.from === undefined) {
        me.from = {};
    }
    me.propHandlers = {};
    me.config = config;
    me.target = Ext.fx.Manager.createTarget(me.target);
    me.easingFn = Ext.fx.Easing[me.easing];
    me.target.dynamic = me.dynamic;
    if (!me.easingFn) {
        me.easingFn = String(me.easing).match(me.bezierRE);
        if (me.easingFn && me.easingFn.length == 5) {
            curve = me.easingFn;
            me.easingFn = Ext.fx.cubicBezier(+curve[1], +curve[2], +curve[3],
                +curve[4]);
        }
    }
    me.id = Ext.id(null, 'ext-anim-');
    me.addEvents(
        'beforeanimate',
        'afteranimate',
        'lastframe'
    );
    me.mixins.observable.constructor.call(me, config);
    if (config.callback) {
        me.on('afteranimate', config.callback, config.scope);
    }
    Ext.fx.Manager.addAnim(me);
},
```

如果定义了 keyframes，说明动画要分帧，那么就要创建 Animator 对象实例，这个稍后再讨论，先往下看。

如果没有定义开始动画的开始值，则设置其为空对象。属性 propHandlers 用于记录属性处理句柄，现在为空对象。

接着调用 FxManager 的 createTarget 创建目标，其代码如下：

```
createTarget: function(target) {
    var me = this,
        useCSS3 = !me.forceJS && Ext.supports.Transitions,
        targetObj;
    me.useCSS3 = useCSS3;
```

```
if (Ext.isString(target)) {
    target = Ext.get(target);
}
if (target && target.tagName) {
    target = Ext.get(target);
    targetObj = new Ext.fx.target['Element' + (useCSS3 ? 'CSS' : '')](target);
    me.targets.add(targetObj);
    return targetObj;
}
if (Ext.isObject(target)) {
    if (target.dom) {
        targetObj =new Ext.fx.target['Element' + (useCSS3 ? 'CSS' : '')]
            (target);
    }
    else if (target.isComposite) {
        targetObj = new Ext.fx.target['CompositeElement' + (useCSS3 ? 'CSS' :
            '')](target);
    }
    else if (target.isSprite) {
        targetObj = new Ext.fx.target.Sprite(target);
    }
    else if (target.isCompositeSprite) {
        targetObj = new Ext.fx.target.CompositeSprite(target);
    }
    else if (target.isComponent) {
        targetObj =new Ext.fx.target.Component(target);
    }
    else if (target.isAnimTarget) {
            return target;
        }
        else {
            return null;
        }
        me.targets.add(targetObj);
        return targetObj;
    }
    else {
        return null;
    }
},
```

如果没有设置 FxManager 的 forceJS 属性（强制使用脚本）为 true，并且浏览器支持 CSS 3 样式，则设置 useCSS3 属性为 true。

下面的代码很清楚了，如果参数 target 是字符串，则认定它为 DOM 节点的 id，通过 get 方法返回 Element 对象。

如果 target 是 HTMLElement 对象，则创建 FxElement 或 FxElementCSS 对象实例。属性 targets 是在 Queue 对象中创建的，是 HashMap 对象实例。也就是说，动画的目标对象都会存在 FxManager 对象的 targets 属性中。

如果定义时 target 是对象，且带 dom 属性，说明是 Element 对象，创建 FxElement 或 FxElementCSS 对象实例。

如果 isComposite 属性为 true，则说明是 CompositeElement 对象，创建 FxCompositeElement 或 FxCompositeElementCSS 对象实例。

如果 isSprite 属性为 true，则说明是 DrawSprite 对象，创建 FxDrawSprite 对象实例。

如果 isCompositeSprite 属性为 true，则说明是 CompositeSprite 对象，创建 FxisCompositeSprite 对象实例。

如果 isComponent 属性为 true，则说明是组件，创建 FxComponent 对象实例。

如果 isAnimTarget 属性为 true，则说明已经是 Target 对象实例，直接返回。

若都不是以上情况，只能返回 null。

创建完目标对象后，根据 easing 配置项，将 easingFn 属性指向 Easing 对象提供的动画功能函数。

把配置项 dynamic 的值复制给目标对象的 dynamic 属性。

如果没有对应的动画方式，则尝试使用贝塞尔曲线。

接着为对象设置一个 id，并调用 addAnim 方法将其添加到 FxManager 对象中，代码如下：

```
addAnim: function(anim) {
    var items = this.items,
        task = this.task;
    items.add(anim.id, anim);
    if (!task && items.length) {
        task = this.task = {
            run: this.runner,
            interval: this.interval,
            scope: this
        };
        Ext.TaskManager.start(task);
    }
},
```

FxManager 对象的 items 属性指向一个 MixedCollection 对象实例，因而 addAnim 方法把动画对象加入管理器。如果当前没有活动任务，且存在动画对象，则创建一个任务，在 interval 指定的时间后运行动画。任务会调用 runner 方法执行动画，其代码如下：

```
runner: function() {
    var me = this,
        items = me.items.getRange(),
        i = 0,
        len = items.length,
        anim;

    me.targetArr = {};

    me.timestamp = new Date();

    for (; i < len; i++) {
        anim = items[i];

        if (anim.isReady()) {
            me.startAnim(anim);
```

```
        }
    }

    for (i = 0; i < len; i++) {
        anim = items[i];

        if (anim.isRunning()) {
            me.runAnim(anim);
        } else if (!me.useCSS3) {
        }

        me.applyPendingAttrs();
    },
```

在以上代码中先遍历目标，如果 isReady 方法返回 true，表示已准备好，调用 startAnim 方法开始运行动画。如果 isRunning 方法返回 true，表示已经在运行的动画暂停，调用 runAnim 方法，继续运行。最后调用 applyPendingAttrs 方法将改变的属性应用到目标对象。

回到 Anim 对象的构造函数，在添加 3 个事件后，初始化事件对象，如果定义了回调函数，那么绑定回调函数，调用 addAnim 方法将实例自身加入到管理器后，结束 Anim 对象的创建。

18.1.3 分步动画的工作流程：Ext.fx.Animator

在上一节中，当创建 Anim 对象时，如果定义了 keyframes 配置项，则会创建 Animator 对象实例，其构造函数如下：

```
    constructor: function(config) {
        var me = this;
        config = Ext.apply(me, config || {});
        me.config = config;
        me.id = Ext.id(null, 'ext-animator-');
        me.addEvents(
            'beforeanimate',
            'keyframe',
            'afteranimate'
        );
        me.mixins.observable.constructor.call(me, config);
        me.timeline = [];
        me.createTimeline(me.keyframes);
        if (me.target) {
            me.applyAnimator(me.target);
            Ext.fx.Manager.addAnim(me);
        }
    },
```

在设置 id 和添加事件后，会调用 createTimeline 方法，其代码如下：

```
createTimeline: function(keyframes) {
    var me = this,
        attrs = [],
        to = me.to || {},
```

```
        duration = me.duration,
        prevMs, ms, i, ln, pct, anim, nextAnim, attr;
    for (pct in keyframes) {
        if (keyframes.hasOwnProperty(pct) && me.animKeyFramesRE.test(pct)) {
            attr = {attrs: Ext.apply(keyframes[pct], to)};
            if (pct == "from") {
                pct = 0;
            }
            else if (pct == "to") {
                pct = 100;
            }
            attr.pct = parseInt(pct, 10);
            attrs.push(attr);
        }
    }
    Ext.Array.sort(attrs, me.sorter);
    ln = attrs.length;
    for (i = 0; i < ln; i++) {
        prevMs = (attrs[i - 1]) ? duration * (attrs[i - 1].pct / 100) : 0;
        ms = duration * (attrs[i].pct / 100);
        me.timeline.push({
            duration: ms - prevMs,
            attrs: attrs[i].attrs
        });
    }
},
```

第一个循环会先将 keyframes 从对象转换成数组，而且会把 from 属性转换为 0，将 to 属性转换为 100。

接着根据 pct 属性将属性进行排序，也就是把每个动画的运行次序排序。

最后一个循环是通过动画的时间总长度，根据百分比计算出每帧的时间长度。

方法 createTimeline 执行完毕后，如果此时动画目标已存在，就调用 applyAnimator 方法，为每帧创建一个 Anim 对象实例，其代码如下：

```
applyAnimator: function(target) {
    var me = this,
        anims = [],
        timeline = me.timeline,
        reverse = me.reverse,
        ln = timeline.length,
        anim, easing, damper, initial, attrs, lastAttrs, i;

    if (me.fireEvent('beforeanimate', me) !== false) {
        for (i = 0; i < ln; i++) {
            anim = timeline[i];
            attrs = anim.attrs;
            easing = attrs.easing || me.easing;
            damper = attrs.damper || me.damper;
            delete attrs.easing;
            delete attrs.damper;
            anim = new Ext.fx.Anim({
                target: target,
                easing: easing,
                damper: damper,
                duration: anim.duration,
```

```
                paused: true,
                to: attrs
            });
            anims.push(anim);
        }
        me.animations = anims;
        me.target = anim.target;
        for (i = 0; i < ln - 1; i++) {
            anim = anims[i];
            anim.nextAnim = anims[i + 1];
            anim.on('afteranimate', function() {
                this.nextAnim.paused = false;
            });
            anim.on('afteranimate', function() {
                this.fireEvent('keyframe', this, ++this.keyframeStep);
            }, me);
        }
        anims[ln - 1].on('afteranimate', function() {
            this.lastFrame();
        }, me);
    }
},
```

如果 beforeanimate 方法没有中止代码，则根据 timeline 数组为每帧创建 Anim 对象实例，最后将这些动画保存在 animations 属性中。

接着要做的是为每个动画指定下一个要执行动画是哪个对象，在当前动画完成后，设置下一个动画的 paused 属性为 false，准备执行下一个动画。接着绑定 afteranimate 事件，因为要设置作用域，所以分开两次绑定。

最后将最后一个动画的 afteranimate 事件绑定到 lastFrame 方法，代码如下：

```
lastFrame: function() {
    var me = this,
        iter = me.iterations,
        iterCount = me.currentIteration;

    iterCount++;
    if (iterCount < iter) {
        me.startTime = new Date();
        me.currentIteration = iterCount;
        me.keyframeStep = 0;
        me.applyAnimator(me.target);
        me.animations[me.keyframeStep].paused = false;
    }
    else {
        me.currentIteration = 0;
        me.end();
    }
},
```

如果当前的运行次数（currentIteration）少于设置的运行次数（iterations），则会重新设置动画，继续执行。否则调用 end 方法结束动画，触发 afteranimate 事件。

从代码中可以看到，Animator 对象就是动画组合，Animator 对象的作用就是将它们串起来显示。

18.2 使用动画

18.2.1 由最简单的动画开始

打开模板页，然后在命令行中创建一个 50×50，背景为蓝色的面板，代码如下：

```
var p=Ext.create(Ext.Panel,{
    renderTo:Ext.getBody(),
    width:50,height:50,
    bodyStyle:"background:blue"
})
```

现在将面板由页面的左上角移动到（100，100）的位置，代码如下：

```
Ext.create(Ext.fx.Anim,{
    target:p,
    duration:1000,
    to:{x:100,y:100}
})
```

在代码中，其实只需要 target 和 to 配置项即可，duration 的作用是让动画持续时间长一些，默认值为 250 微秒。

从代码中可以看到，使用动画功能很简单，可以通过以下 11 个配置项来定义动画。

- alternate：布尔值，该配置项要结合 iterations 配置项，以实现每次动画完成就反向再执行一次。默认值为 false。
- delay：数值，定义延迟时间，单位为微秒，在延迟时间过后再执行动画。默认值为 0。
- durati：数值，定义动画持续的时间长度，单位为微秒。默认值为 250。
- dynamic：布尔值，默认值为 false，会绕过组件的布局，只对最外层元素实施动画效果。如果设置为 true，则包括组件的布局一起执行动画效果。
- easing：要使用的功能函数，值可以是 backIn、backOut、bounceIn、bounceOut、ease、easeIn、easeOut、easeInOut、elasticIn、elasticOut、cubic-bezier(x1, y1, x2, y2)，各值的用途可阅读 18.2.2 节。
- from：JavaScript 对象，定义动画起始时目标的状态。如果不设置，则使用目标当前状态。
- iterations：整数，定义动画的执行次数。默认值为 1。
- keyframes：对象，定义动画的帧。
- reverse：布尔值，设置为 true，动画会反向执行。默认值为 false。
- target：定义动画的目标对象。
- to：JavaScript 对象，定义动画结束时目标的状态。

18.2.2 过渡效果使用的功能函数介绍

Ext JS 4 提供了以下 12 种过渡使用的功能函数。

- linear：默认值，匀速变化。

- backIn：退后开始。
- backOut：越界后，退后结束。
- bounceIn：跳动开始。
- bounceOut：跳动结束。
- ease：低速开始，然后加速，最后减速。
- easeIn：加速。
- easeOut：减速。
- easeInOut：低速开始，减速结束。
- elasticIn：弹性结束。
- elasticOut：弹性开始。
- cubic-bezier(x1, y1, x2, y2)：自定义贝塞尔曲线用于过渡运算。需要提供两个点的坐标值，这些值必须在 0 和 1 之间。

下面通过一个示例演示一下这些功能函数的效果。使用模板页创建一个名称为 18-1.html 的页面文件。然后定义一个面板，在面板顶部的工具栏上定义 12 个以功能函数名称为显示文本的按钮，在面板内放置一个 Draw 组件，在其中画一个圆，具体代码如下：

```
var panel=Ext.create(Ext.Panel,{
    renderTo:Ext.getBody(),
    width:800,height:200,
    layout:"fit",
    tbar:[
        {text:"linear",handler:show},
        {text:"backIn",handler:show},
        {text:"backOut",handler:show},
        {text:"bounceIn",handler:show},
        {text:"bounceOut",handler:show},
        {text:"ease",handler:show},
        {text:"easeIn",handler:show},
        {text:"easeOut",handler:show},
        {text:"easeInOut",handler:show},
        {text:"elasticIn",handler:show},
        {text:"elasticOut",handler:show},
        {text:"cubic-bezier",handler:show}
],
items:[
        {xtype:"draw",viewBox:false,
            items: [{
                type:'circle',
                fill: '#00f',
                radius:20,
                x:100,y:40
            }]
        }
    ]
})
```

最后定义按钮的句柄 show 函数，代码如下：

```
var show=function(el){
    var ersing=el.text;
    if(ersing=="cubic-bezier"){
            ersing="cubic-bezier("+Math.random()+
                    ","+Math.random()+","+Math.random()+","+
                    Math.random()+")";
            console.log(ersing);
    }
    Ext.create(Ext.fx.Anim,{
        easing:ersing,
        target:panel.down("draw").surface.items.items[0],
        duration:3000,
        from:{x:100},
        to:{x:700}
    })
}
```

使用“cubic-bezier”功能，需要随机产生 4 个坐标，如果按钮文本为“cubic-bezier”，重新组织一下数据，使用 console.log 语句将其显示出来，有兴趣的读者可以根据这些值研究一下效果是怎样的。

因为目标是画布中的圆，因而需要从面板开始，先找到画布，再从图层中把圆的对象找出来。每次横坐标都会从 100 开始，到 700 结束。

因为 Anim 构造函数在调用 cubicBezier 方法时有 bug，所以需要在 OnReady 前加入以下代码：

```
Ext.fx.cubicBezier=Ext.fx.CubicBezier.cubicBezier;
```

这样构造函数中的 Ext.fx.cubicBezier 能正确调用 cubicBezier 方法。

在浏览器中打开页面，将看到如图 18-1 所示的效果，单击按钮就可看到各功能函数的效果了。

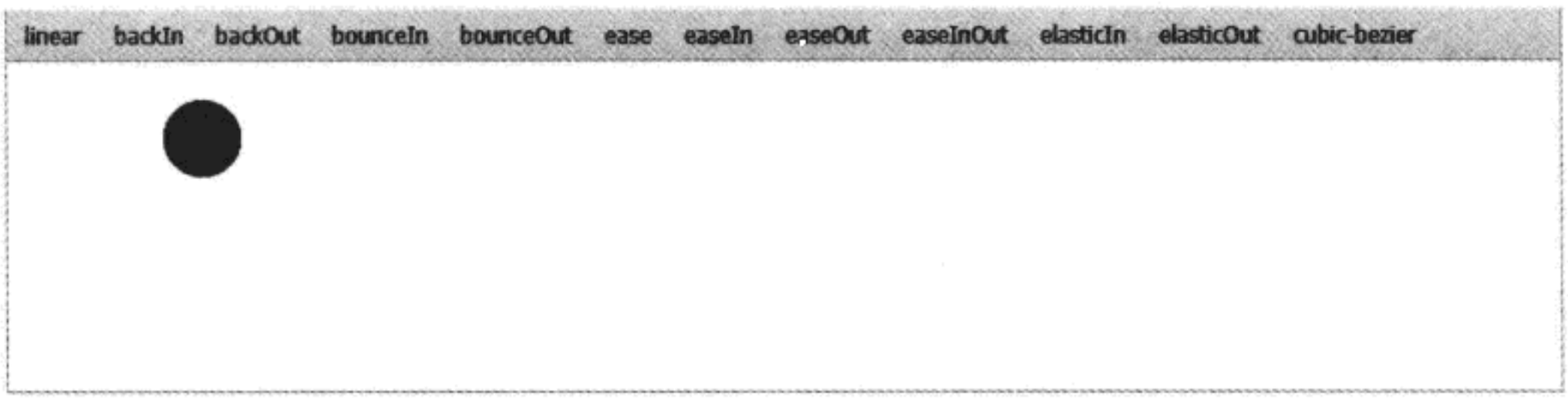


图 18-1　示例的页面效果

18.2.3　使用分步动画

Animator 对象的配置项与 Anim 的配置项是一样的，这是因为 Animator 对象其实是多个 Anim 对象的组合。

使用 Animator 对象一般不需要定义 from 或 to 配置项，因为在 keyframes 中已经包含了这两

个配置项，而且从 18.1.3 节可以知道，from 或 to 配置项最终会转换为 keyframes 中的配置项。

下面通过一个示例演示如何使用 Animator 对象。在页面中创建一个 50×50，背景色为蓝色的面板，面板的宽度从 50 扩大到 200 后，接着把高度从 50 扩大到 200，然后把宽度缩小回 50，最后把高度缩小回 50。

使用目标页创建一个名称为 18-2.html 的页面文件，先创建面板，代码如下：

```
var panel=Ext.create(Ext.Panel,{
    renderTo:Ext.getBody(),
    width:50,height:50,
    bodyStyle:"background:blue;"
});
```

接着创建 Animator 对象，代码如下：

```
Ext.create(Ext.fx.Animator,{
    target:panel,
    duration:25000,
    keyframes:{
            25:{width:200},
            50:{height:200},
            75:{width:50},
            100:{height:50}
    }
});
```

因为是从原始状态开始，所以序数为 0 的状态不用定义也可以。在以上代码中，序数可以用百分数进行定义。

在浏览器中打开页面将看到预定的效果。

注意　4.1 Beta 1 中 Ext.fx.Manager 的 runner 方法有错误，会造成示例运行不了，打开 Firebug 可看到错误提示。

18.2.4　注意的问题

在使用动画时，设置需要变化的属性，一定要注意目标的对象类型，例如对于面板，能设置的属性是左上角坐标和大小，如果将其设置为 DOM 节点的属性，将会出错，因为面板没有 DOM 对象的属性。

18.3　在 Element 对象中使用动画

1. 概述

在 Element 对象中内置了很多动画方法，这样，当组件要实现动画的时候，就不需要另外为其创建 Amin 对象实例或 Animator 对象实例了，在内部直接使用即可。

Element 对象内置了 9 个动画方法，这些方法都可根据 6.4 节中介绍 alignTo 时提到的定

位点开始或结束动画。

2. 滑进 / 滑出效果：slideIn/slideOut

(1）基本语法

```
el.slideIn([postion],[options],[slideOut])
el.slideOut([postion],[options])
```

其中，el 为 Element 对象实例；postion 值可选，为动画的开始或结束位置，默认值为 t；options 值可选，为 Anim 对象的配置对象；slideOut 为可选的布尔值，如果为 true，与 sildeOut 效果一样。

(2）示例

使用模板页创建一个名称为 18-3.html 的页面文件，先创建一个 Store，准备用于显示动画的开始或结束位置，代码如下：

```
var store=Ext.create(Ext.data.ArrayStore,{
    fields:["text","value"],
    data:[["左上角","tl"],
          ["顶边中心","t"],
          ["右上角","tr"],
          ["左边中心","l"],
          ["右边中心","r"],
          ["左下角","bl"],
          ["底边中心","b"],
          ["左上角","br"]
    ]
})
```

接着定义一个 500×500 的面板，在面板内加两个按钮分别用于执行 sildeIn 和 slideOut 方法，再放置一个下拉列表框用于选择位置，在面板内放置一个宽度为 500 的图片，该图片就是实现滑进或滑出效果的主角，具体代码如下：

```
var panel=Ext.create(Ext.Panel,{
    renderTo:Ext.getBody(),
    width:500,height:500,
    tbar:[
          {text:"slideIn",handler:doAnim},
          {text:"slideOut",handler:doAnim},
          "位置：",
          {xtype:"combo",id:"pos",store:store,
                displayField:"text",valueField:"value",
                value:"tl"
          }
    ],
    items:[
          {xtype:"image",width:500,src:"../images/1.jpg"}
    ]
});
```

最后定义 doAnim 函数，代码如下：

```
var doAnim=function(el){
    var img=panel.down("image"),
            cb=Ext.getCmp("pos"),
            pos=cb.getValue()?cb.getValue():"t";
    img.el[el.text](pos)
}
```

因为 img 的 el 就是 Element 对象，所以可以直接调用方法，要调用通过按钮的文本指定方法，必须使用代码中的形式，这方法在某些时候很常用，其实就是访问对象的属性，指向一个函数，加括号后就可执行该函数。通过参数 pos 可指定效果的开始位置与结束位置，至于什么时候是结束位置，什么时候是开始位置，可通过源代码确定。

在浏览器打开页面，将看到如图 18-2 所示的效果。单击 slideOut 按钮，会看到图片向左上角滑出，最后隐藏了。继续单击 sildeIn 按钮，图片会从左上角慢慢滑进，恢复到图 18-2 所示的效果。

图 18-2　示例的页面效果

3. 膨胀效果：puff

(1）基本语法

```
el.puff([options]);
```

其中，options 为可选参数，为 Anim 对象的配置对象。

(2）示例

在上一小节中，在 sildeOut 按钮后加入一个 puff 按钮，代码如下：

```
{text:"puff",handler:function(){
    var img=panel.down("image");
    img.el.puff();
}},
```

刷新页面，然后单击 puff 按钮，会看到图片逐渐膨胀，直至隐藏。要注意，puff 方法会修改元素的尺寸，因此再用其他方法的时候，一定要将元素的尺寸修改回原来的尺寸。

4. 关闭效果：switchOff

（1）基本语法

```
el.switchOff([options]);
```

其中，options 为可选参数，为 Anim 对象的配置对象。

（2）示例

在前面示例中，在 puff 按钮后加入一个 switchOff 按钮，代码如下：

```
{text:"switchOff",handler:function(){
   var img=panel.down("image");
   img.el.switchOff();
}},
```

刷新页面，然后单击 switchOff 按钮，会看到图片向中间收缩成一条线后，最后隐藏，类似老式电视机关机的画面。

5. 淡入淡出效果：fadeIn/fadeout

（1）基本语法

```
el.fadeIn([options]);
el.fadeOut([options]);
```

其中，options 为可选参数，为 Anim 对象的配置对象。

（2）示例

在前面示例中，先把面板的宽度改 800，然后在 switchOff 按钮后加入 fadeIn 和 fadeOut 两个按钮，代码如下：

```
{text:"fadeIn",handler:function(){
   var img=panel.down("image");
   img.el.fadeIn({duration:3000});
}},
{text:"fadeOut",handler:function(){
   var img=panel.down("image");
   img.el.fadeOut({duration:3000});
}},
```

因为默认动画时间太短，看不出效果，所以调整动画时间为 3 秒。

刷新页面，然后单击 fadeOut 按钮，会看到图片逐渐从不透明变成透明，直至隐藏。单击 fadeIn 按钮，会看到图片逐渐从透明变成不透明，直至完全显示。

6. 渐隐效果：ghost

（1）基本语法

```
el.ghost([postion],[options])
```

其中，postion 值可选，为动画的开始或结束位置，默认值为 t；options 值可选，为 Anim 对象的配置对象。

（2）示例

在前面示例的 fadeOut 后添加 ghost 按钮，代码如下：

```
{text:"ghost",handler:doAnim},
```

因为与 sildeIn 方法的参数一样，所以可以使用 doAnim 进行调用。
刷新页面，然后单击 ghost 按钮，会看到图片在向左上角移出的同时，渐渐变得透明。

7. 爆炸波纹效果：frame

(1）基本语法

```
el.frame([color],[count],[options]);
```

其中，color 为可选参数，设置爆炸波纹的颜色，颜色值必须以“#”开头，以 6 位字符方式表示颜色值；count 为可选参数，为效果的显示次数，默认值为 1；options 为可选参数，为 Anim 对象的配置对象。

(2）示例

在前面示例中，先在图片下添加一个面板，在面板内显示“测试文字”这 4 个字，代码如下：

```
{xtype:"panel",height:60,
   bodyStyle:"font-size:40px;line-height:60px;text-align:center",
   html:"<span id='test'>测试文字</span>"
}
```

再在 ghost 按钮后添加 frame 按钮，单击此按钮后对 id 为 test 的 SPAN 元素调用 frame 方法，代码如下：

```
{text:"frame",handler:function(){
   var el=Ext.get("test");
   el.frame("#0000ff",3);
}},
```

刷新页面，然后单击 frame 按钮，会看到页面在图片下添加了如图 18-3 所示的文字。单击 frame 按钮，将看到文字四周会出现向四周扩散的爆炸波纹。

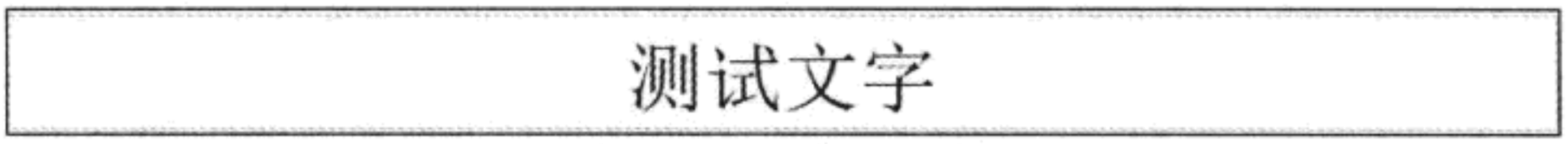


图 18-3　加入的文字面板

8. 突出显示效果：highlight

（1）基本语法

```
el.frame([color],[options]);
```

其中，color 为可选参数，设置文字高亮显示时颜色，颜色值必须以“#”开头，以 6 位字符方式表示颜色值；options 为可选参数，为 Anim 对象的配置对象。

（2）示例

在前面示例中，在 frame 按钮后添加 highlight 按钮，单击后对 id 为 test 的 SPAN 元素调

用 highlight 方法，代码如下：

```
{text:"highlight",handler:function(){
    var el=Ext.get("test");
    el.highlight("#0000ff");
}},
```

刷新页面，然后单击 highlight 按钮，将看到文字背景变成了蓝色，然后背景颜色逐渐由不透明到透明。

注意 以上示例部分在 Ext JS 4.1 Beta 1 中不能运行，主要原因还是 18.2.3 节提到的错误，所以最后要在 Ext JS 4.0.7 或修正了错误的后续版本中测试。

18.4 本章小结

Ext JS 只是提供了最基本的动画效果，要实现更多更炫的动画效果，还得进行扩展。目前 Ext JS 最大的问题是使用组件多，而对于 Ext Core 的扩展太少了，尤其是像 JQuery 一样专门针对页面的一些效果及插件，希望在 Ext JS 4 之后能有大的发展。

第19章　拖放功能

在我们使用的系统中，拖放功能可以说随处可见，而且经常被用到，因此，在一个页面中添加拖放功能也是提高用户体验的一种有效方式。Ext JS 不单实现了基于元素的拖放功能，还实现了与 Store 有关的 Grid、树和视图的拖放功能，这大大提高了组件的用户体验，又为开发人员提供一种数据处理方式。

本章将介绍拖放功能的构成、工作流程及使用方法。

19.1　拖放功能的构成及工作流程

19.1.1　概述

图 19-1 列出了所有与拖放功能有关的类，DragDrop 对象为拖放功能提供了接口和基本操作，其分出两个分支，左边分支用于实现拖动操作，右边分支用于实现放置操作。

在左边分支中，DD 对象用于实现元素的基本拖动操作。

DDProxy 对象的主要作用是创建一个空的带边界的 DIV 元素，代替要拖动对象本身进行拖动操作，相当于一个代理。当发生拖动操作时，会在 DIV 内显示拖动对象的副本，跟随光标移动。该对象只是为后续的对象提供代理接口。

DragSource 对象在 DDProxy 对象基础上将代替拖动对象的 DIV 替换为 StatusProxy 对象实例，提供带状态标识（⊘、✓或+）的拖动代理，并为标识的更改提供相应操作。

DragZone 对象的作用是，作为一个容器为 DragSource 对象提供可拖动的节点。ViewDragZone 的作用是把视图作为容器，将数据节点作为拖动节点。TreeVeiwDragZone 就是把 TreeView 作为容器，把树节点作为拖动节点，因为树节点的数据格式与视图的数据格式不同，所以需要根据树节点的特性扩展出该对象。GridHeaderDragZone 则把列标题（Grid）作为容器，把允许拖动列的标题作为拖动节点，以实现每列标题的拖动操作。

PanelDD 对象是以 PanelProxy 对象实例作为代理来实现拖动的。PanelProxy 对象会把面板自身作为代理，实现拖动，这是因为面板的拖动，尤其是窗口的拖动，不能像记录那样，可以通过一个副本来实现，否则看到两个面板，或者两个窗口，就很怪异了，而且使用副本重新复制一个面板或窗口，开销会很大。那么为什么不直接从 DD 对象继承呢？在拖动面板的时候，只能通过标题栏实现拖动，也就是说拖动的元素是标题栏的元素，而显示要求是拖动整个面板。如果使用 DD 对象，设置标题栏的元素为拖动对象，那么就只能看到标题栏在动了。如果设置整个面板作为拖动对象，那么面板内不单是标题栏，任何其他位置都可以拖动了，这就有可能与面板内的允许拖动的对象造成冲突。

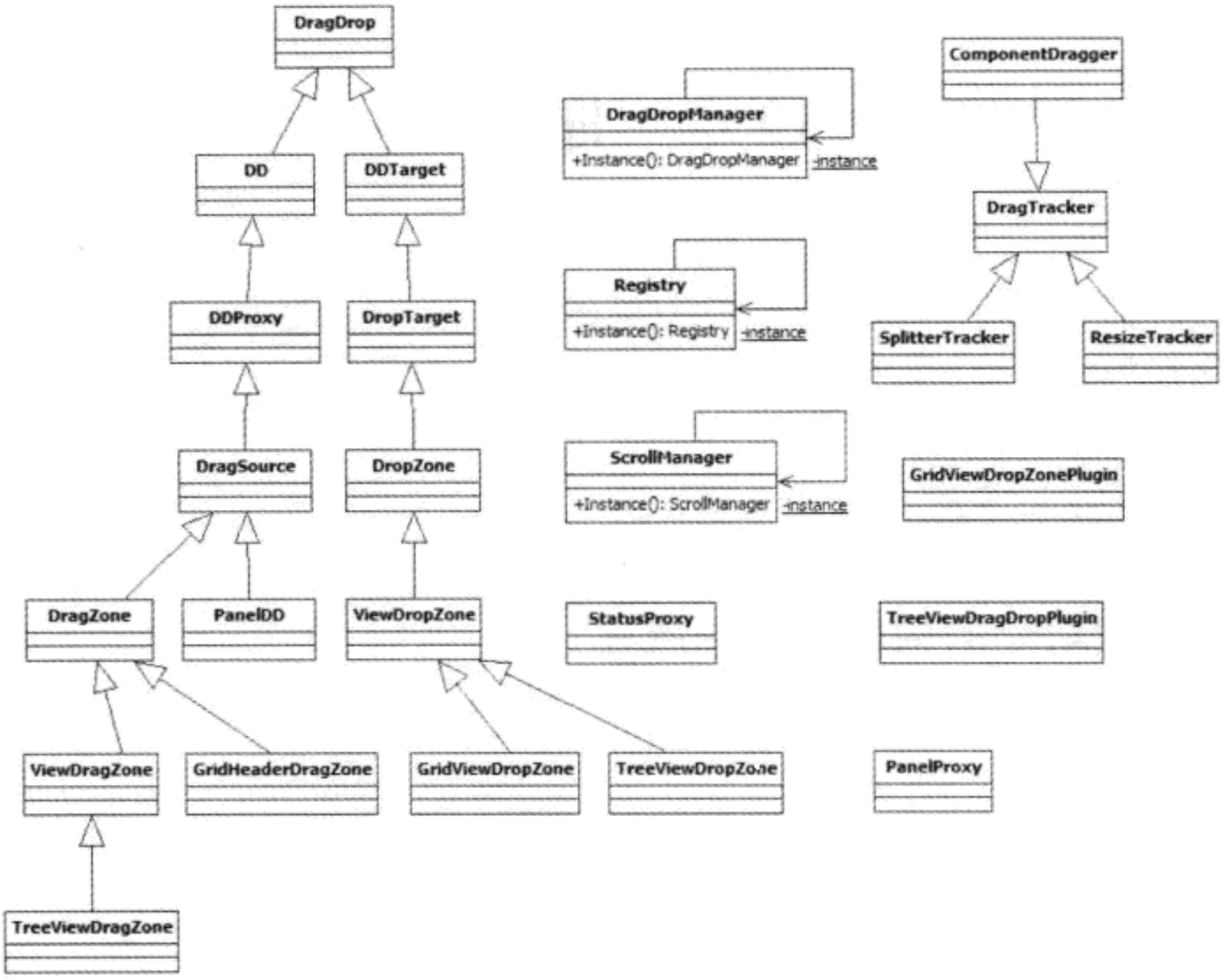


图 19-1　拖放类图

右边分支就简单多了，因为只是放置功能，不需要太多的拖动运算及拖动功能，所以DDTarget 对象的作用就是把 DragDrop 对象的拖动功能屏蔽，只保留放置功能。

DropTarget 对象在 DDTarget 对象的基础上添加了基本的放置操作。

DropZone 对象的作用是把一个元素作为容器，允许放置多个子节点。

ViewDropZone 对象在 DropZone 对象的基础上，把视图作为放置容器，可以接收视图节点。GridViewDropZone 对象和 TreeViewDropZone 对象的主要作用是分别把 Grid 的行和树节点作为其放置对象。

DragDropManager 对象是单件模式对象，其作用是跟踪窗口内元素的拖动操作。

Registry 对象是单件模式对象，其作用是记录拖动对象，在拖放事件中，通过事件目标找到这些对象来实施操作。

ScrollManager 对象是单件模式对象，其作用是在拖动时根据拖动操作实现滚动条的自动滚动。

DragTracker 对象的作用是监听元素的拖动事件，并在拖动开始或结束的时候触发事件。ComponentDragger 对象是专门用于处理组件拖动事件的对象。ResizeTracker 对象是为 Resizer 对象提供拖动事件处理的对象。SplitterTracker 是为分隔条提供拖动事件处理的对象。

因为重构后的 Grid 是通过插件来插入功能的，所以为了实现 Grid 的拖动功能，创建了

GridViewDropZonePlugin 对象。树与 Grid 同源，因此其拖动功能也必须通过插件实现，但树与 Grid 又有区别，因此创建了 TreeViewDragDropPlugin 对象。

19.1.2　DragDropManager 对象的工作流程

由于 DragDropManager 对象没有构造函数，因此在创建该对象后，会调用 _addListeners 方法，代码如下：

```
_addListeners: function() {
    if ( document ) {
        this._onLoad();
    } else {
        if (this._timeoutCount > 2000) {
        } else {
            setTimeout(this._addListeners, 10);
            if (document && document.body) {
                this._timeoutCount += 1;
            }
        }
    }
},
```

如果文档已经准备好，调用 _onLoad 方法，不然继续等待，直到可以调用 _onLoad 方法，代码如下：

```
_onLoad: function() {
  this.init();
    var Event = Ext.EventManager;
    Event.on(document, "mouseup",   this.handleMouseUp, this, true);
    Event.on(document, "mousemove", this.handleMouseMove, this, true);
    Event.on(window,   "unload",    this._onUnload, this, true);
    Event.on(window,   "resize",    this._onResize, this, true);
},
```

首先调用 inti 方法将 initialized 属性设置为 true，表示已经初始化了。接着为 document 对象绑定 mouseup 和 mousemove 事件，为 window 对象绑定 unload 和 resize 事件。

首先介绍 handleMouseUp 方法，代码如下：

```
handleMouseUp: function(e) {
    if(Ext.tip.QuickTipManager){
        Ext.tip.QuickTipManager.ddEnable();
    }
    if (! this.dragCurrent) {
        return;
    }
    clearTimeout(this.clickTimeout);
    if (this.dragThreshMet) {
        this.fireEvents(e, true);
    } else {
    }
    this.stopDrag(e);
    this.stopEvent(e);
},
```

如果 QuickTipManager 对象存在，则调用 ddEnable 方法开启 QuickTip 功能。
如果当前不是拖动操作（没有拖动对象）触发的事件，则直接返回。
清理了计时事件后，如果 dragThreshMet 为 true，触发事件 e。
接着调用 stopDrag 方法，停止拖动，代码如下：

```
stopDrag: function(e) {
    if (this.dragCurrent) {
        if (this.dragThreshMet) {
            this.dragCurrent.b4EndDrag(e);
            this.dragCurrent.endDrag(e);
        }
        this.dragCurrent.onMouseUp(e);
    }
    this.dragCurrent = null;
    this.dragOvers = {};
},
```

如果存在拖动对象，并且 dragThreshMet 为 true，调用 b4EndDrag 方法，在 DDProxy 对象中，此方法会隐藏代理元素。接着调用 endDrag 方法，结束拖动操作。接着调用 onMouseUp 方法。

最后将 dragCurrent 设置为 null，表示没有拖动对象，设置 dragOvers 为空对象。

回到 handleMouseUp 方法中，最后调用 stopEvent 事件，停止事件传播和默认事件。

下面介绍 handleMouseMove 方法，代码如下：

```
handleMouseMove: function(e) {
    if (! this.dragCurrent) {
        return true;
    }
    if (Ext.isIE && (e.button !== 0 && e.button !== 1 && e.button !== 2)) {
        this.stopEvent(e);
        return this.handleMouseUp(e);
    }
    if (!this.dragThreshMet) {
        var diffX = Math.abs(this.startX - e.getPageX());
        var diffY = Math.abs(this.startY - e.getPageY());
        if (diffX > this.clickPixelThresh ||
                    diffY > this.clickPixelThresh) {
            this.startDrag(this.startX, this.startY);
        }
    }
    if (this.dragThreshMet) {
        this.dragCurrent.b4Drag(e);
        this.dragCurrent.onDrag(e);
        if(!this.dragCurrent.moveOnly){
            this.fireEvents(e, false);
        }
    }
    this.stopEvent(e);
    return true;
},
```

如果方法不是通过拖动操作触发的，则直接返回 true。如果浏览器是 IE，并且按下的按钮不是左右键或中间键，则停止事件，返回经 handleMouseUp 方法处理后的结果。

如果 dragThreshMet 为 false，那么计算开始坐标，此时调用 startDrag 方法，其代码如下：

```
startDrag: function(x, y) {
    clearTimeout(this.clickTimeout);
    if (this.dragCurrent) {
        this.dragCurrent.b4StartDrag(x, y);
        this.dragCurrent.startDrag(x, y);
    }
    this.dragThreshMet = true;
},
```

调用 b4StartDrag 方法在 DDProxy 对象中显示代理元素。

接着调用 startDrag 方法，在 DragSource 对象中，会克隆节点，并将克隆后的节点加入到代理元素内，然后调用 onStartDrag 方法。在其余的拖动对象中，不是调用空函数就是调用 onStartDrag 方法。

最后设置 dragThreshMet 为 true，完成 startDrag 方法。

回到 handleMouseMove 方法，如果 dragThreshMet 为 true，调用 b4Drag 方法，在 DD 对象中，会根据事件坐标设置拖动对象的坐标，让拖动对象跟随鼠标移动。

接着调用 onDrag 方法，该方法一般需要开发人员重写。

如果拖动对象纯粹是移动功能，则触发事件。

最后停止事件，并返回 true 来结束 handleMouseMove 方法。

window 的 upload 事件的作用是在页面刷新、关闭或后退时，清理管理器存储的对象，而 resize 事件的作用是当页面大小发生改变时，重新计算容器的大小。

19.1.3 注册节点：Ext.dd.Registry

Registry 对象的构造函数如下：

```
constructor: function() {
    this.elements = {};
    this.handles = {};
    this.autoIdSeed = 0;
},
```

简单的创建了两个空对象。

其 register 方法的代码如下：

```
register : function(el, data){
    data = data || {};
    if (typeof el == "string") {
        el = document.getElementById(el);
    }
    data.ddel = el;
    this.elements[this.getId(el)] = data;
    if (data.isHandle !== false) {
        this.handles[data.ddel.id] = data;
```

```
        }
        if (data.handles) {
            var hs = data.handles;
            for (var i = 0, len = hs.length; i < len; i++) {
                this.handles[this.getId(hs[i])] = data;
            }
        }
    },
```

在 elements 对象中，会以元素的 id 作为关键字，指向 data 对象，而 data 对象包含一个指向元素的 ddel 属性。如果 data 的 isHandle 属性为 true，则在 handles 对象中将元素 id 作为关键字，指向 data 对象。如果 data 存在 handles 属性，则在 handles 对象中添加 handles 内的元素。

这样，当这些元素触发事件时，就可根据其 id 从 Registry 对象中找到拖动的对象，以便进一步处理。

19.1.4 一般拖动功能的工作流程：Ext.dd.DD

DD 对象的构造函数的代码如下：

```
constructor: function(id, sGroup, config) {
    if (id) {
        this.init(id, sGroup, config);
    }
},
```

如果指定了元素，调用 init 方法（DragDrop.js），代码如下：

```
init: function(id, sGroup, config) {
    this.initTarget(id, sGroup, config);
    Ext.EventManager.on(this.id, "mousedown", this.handleMouseDown, this);
},
```

先调用 initTarget 方法（DragDrop.js），代码如下：

```
initTarget: function(id, sGroup, config) {
    this.config = config || {};
    this.DDMInstance = Ext.dd.DragDropManager;
    this.groups = {};
    if (typeof id !== "string") {
        id = Ext.id(id);
    }
    this.id = id;
    this.addToGroup((sGroup) ? sGroup : "default");
    this.handleElId = id;
    this.setDragElId(id);
    this.invalidHandleTypes = { A: "A" };
    this.invalidHandleIds = {};
    this.invalidHandleClasses = [];
    this.applyConfig();
    this.handleOnAvailable();
},
```

如果元素没有 id，先为元素添加 id，然后调用 addToGroup 方法，其代码如下：

```
addToGroup: function(sGroup) {
    this.groups[sGroup] = true;
    this.DDMInstance.regDragDrop(this, sGroup);
},
```

首先在 groups 对象中添加关键字为 sGroup，true 为值的成员。接着调用 DragDrop-Manager 对象的 regDragDrop 方法，其代码如下：

```
regDragDrop: function(oDD, sGroup) {
    if (!this.initialized) { this.init(); }

    if (!this.ids[sGroup]) {
        this.ids[sGroup] = {};
    }
    this.ids[sGroup][oDD.id] = oDD;
},
```

也就是在 ids 内，根据组名记录该组中所有的 DD 对象实例。

回到 initTarget 方法中，接着设置 handleElId（拖动对象的句柄）的值为 id，调用 setDragElId 设置 dragElId（拖动对象）的值为 id。

接着设置合法的句柄类型，默认值是 A 标记。

接着调用 applyConfig 方法，将配置对象中 padding、isTarget、maintainOffset 和 primary-ButtonOnly 配置项的值应用到对应属性中。如果没有定义，则使用默认值。

接着调用 handleOnAvailable，代码如下：

```
handleOnAvailable: function() {
    this.available = true;
    this.resetConstraints();
    this.onAvailable();
},
```

设置 available 属性为 true 后，调用 resetConstraints 方法设置初始坐标及拖动约束。

最后调用 onAvailable 方法（空函数）。

回到 init 方法中，接着将元素的 mousedown 事件绑定到方法中，其代码如下：

```
handleMouseDown: function(e, oDD){
    if (this.primaryButtonOnly && e.button != 0) {
        return;
    }
    if (this.isLocked()) {
        return;
    }
    this.DDMInstance.refreshCache(this.groups);
    var pt = e.getPoint();
    if (!this.hasOuterHandles && !this.DDMInstance.isOverTarget(pt, this)) {
    } else {
        if (this.clickValidator(e)) {
            this.setStartPosition();
            this.b4MouseDown(e);
```

```
                this.onMouseDown(e);
                this.DDMInstance.handleMouseDown(e, this);
                this.DDMInstance.stopEvent(e);
            } else {
            }
        }
    },
```

如果 PrimaryButtonOnly 为 true（只允许鼠标左键）且触发事件的鼠标按钮不是鼠标左键（button 不为 0），那么直接返回。

如果 islocked 为 true（锁定状态），那么也直接返回。

如果都不属于以上情况，那么直接调用 DragDropManager 对象的 refreshCache 方法刷新缓冲中拖动对象的左上角和右下角坐标。

接着，调用 getPoint 方法返回事件的坐标点。

紧接着的这个判断语句，只要 hasOuterHandles 为 true 或 isOverTarget（检查是否在放置容器上）返回 true 才会执行。首先调用 clickValidator 方法检查单击的是否是用于拖动的句柄，如果是，调用 setStartPosition 方法设置开始坐标。接着调用 b4MouseDown 设置拖动对象的坐标，调用 onMouseDown 方法（空函数），此方法需要开发人员重写，调用 DragDropManager 对象的 handleMouseDown 方法，代码如下：

```
handleMouseDown: function(e, oDD) {
    if(Ext.tip.QuickTipManager){
        Ext.tip.QuickTipManager.ddDisable();
    }
if(this.dragCurrent){
    this.handleMouseUp(e);
}
this.currentTarget = e.getTarget();
this.dragCurrent = oDD;
var el = oDD.getEl();

this.startX = e.getPageX();
this.startY = e.getPageY();
this.deltaX = this.startX - el.offsetLeft;
this.deltaY = this.startY - el.offsetTop;
this.dragThreshMet = false;
this.clickTimeout = setTimeout(
        function() {
            var DDM = Ext.dd.DragDropManager;
            DDM.startDrag(DDM.startX, DDM.startY);
        },
        this.clickTimeThresh );
},
```

首先禁用 QuickTips。接下来，如果当前已存在拖动对象，调用 handleMouseUp 方法结束其拖动，以免破坏将要进行的拖动操作。接着从事件中获取拖动目标，设置 dragCurrent 属性指向拖动对象。计算好坐标并设置 dragThreshMet 为 false 后，创建一个时间任务，在指定时间（默认值为 350 微秒）后执行 startDrag 方法开始拖动对象。

回到 handleMouseDown 方法，最后调用 stopEvent 方法停止事件。

因为拖动对象就是元素本身，所以在鼠标移动事件中，调用 b4Drag 方法修改的是对象的坐标，对象会跟随鼠标移动，其停止时的坐标就是最后时刻的坐标。

19.1.5　DragSource 对象的工作流程

DragSource 对象的构造函数如下：

```
constructor: function(el, config) {
    this.el = Ext.get(el);
    if(!this.dragData){
        this.dragData = {};
    }
    Ext.apply(this, config);
    if(!this.proxy){
        this.proxy = new Ext.dd.StatusProxy({
           id: this.el.id + '-drag-status-proxy',
            animRepair: this.animRepair
        });
    }
    this.callParent([this.el.dom, this.ddGroup || this.group,
          {dragElId : this.proxy.id, resizeFrame: false, isTarget: false, scroll:
              this.scroll === true}]);
    this.dragging = false;
},
```

首先将 el 指向元素，直接初始化 dragData，默认值为空对象。将配置项复制到对象中。

如果代理不存在，创建 StatusProxy 对象实例。

接着调用 DDProxy 对象的构造函数，代码如下：

```
constructor: function(id, sGroup, config) {
    if (id) {
        this.init(id, sGroup, config);
        this.initFrame();
    }
},
```

先调用 init 方法初始目标并绑定 mousedown 事件。接着调用 initFrame 方法，也就是调用 createFrame 方法，代码如下：

```
createFrame: function() {
    var self = this;
    var body = document.body;
    if (!body || !body.firstChild) {
        setTimeout( function() { self.createFrame(); }, 50 );
        return;
    }
    var div = this.getDragEl();
    if (!div) {
        div    = document.createElement("div");
        div.id = this.dragElId;
        var s  = div.style;
```

```
            s.position   = "absolute";
            s.visibility = "hidden";
            s.cursor     = "move";
            s.border     = "2px solid #aaa";
            s.zIndex     = 999;
            body.insertBefore(div, body.firstChild);
        }
    },
```

如果文档还没加载完成，就不断地调用 setTimeout 方法，直到 document 对象存在才继续执行后续代码。接着调用 getDragEl 方法获取拖动对象，代码如下：

```
    getDragEl: function() {
        return Ext.getDom(this.dragElId);
    },
```

就是根据 id 返回 DOM 节点。注意，当在构造函数中调用 callParent 方法时，dragElId 的值是 StatusProxy 对象实例的 id，即拖动对象已经存在，不需要执行下面代码。否则，创建一个 DIV 标记，将其插入到 body 中。

至此，DragSource 对象的实例化过程就完成了。要注意一点，DragSource 对象的拖动实现的只是代理对象在动，实际对象并没有移动，因而不会像 DD 对象那样，跟随鼠标到达指定位置，要让拖动的对象移动，必须有放置容器接收该对象。

19.1.6 DropTarget 对象的工作流程

DropTarget 的构造函数如下：

```
constructor : function(el, config){
    this.el = Ext.get(el);
    Ext.apply(this, config);
    if(this.containerScroll){
        Ext.dd.ScrollManager.register(this.el);
    }
    this.callParent([tnis.el.dom, this.ddGroup || this.group,
          {isTarget: true}]);
},
```

先将 el 指向放置对象，应用配置项到对象，如果有滚动条，那么在 ScrollManager 对象中注册放置对象。最后调用 DDTarget 的构造函数，其代码如下：

```
constructor: function(id, sGroup, config) {
    if (id) {
        this.initTarget(id, sGroup, config);
    }
},
```

只是调用 initTarget 方法那么简单。

纵观整个初始化操作，会有雾里看花的感觉，完全不知道是如何接收对象的。这主要是因为 DropTarget 只有在拖动对象进入其范围内，才会判断是否要接收对象，因而它要根据拖动对象的操作来执行接收操作。在 DragDropManager 对象的 fireEvents 方法中，会根

据当前鼠标的坐标点判断当前坐标是否处于与拖动目标同组的放置容器内，如果是，会调用 onDragOut、onDragEnter、onDragOver 和 onDragDrop 这 4 个方法，而在 DragSource 的 onDragDrop 方法中，如果放置容器的 isNotifyTarget 属性为 true，则会调用 notifyDrop 方法，然后通过重写该方法就可实现放置操作了。

19.1.7　DragZone 对象的工作流程

DragZone 对象的构造函数与 DragSource 对象的构造函数相比，只是多了在 ScrollManager 对象中注册拖动对象而已，其主要的改变是在 getDragData 方法中，代码如下：

```
getDragData : function(e){
    return Ext.dd.Registry.getHandleFromEvent(e);
},
```

getDragData 会从 Registry 对象中根据事件返回拖动的数据。很多时候可以通过重写 getDragData 方法，绕开在 Registry 对象中注册数据这个过程，以简化代码。

19.1.8　DropZone 对象的工作流程

DropZone 对象没有自己的构造函数，因此它使用的是父对象的构造函数。在 DropTarget 对象的基础上，DropZone 对象添加了 getTargetFromEvent 方法，用于从 Registry 对象中根据事件返回放置对象。

19.2　使用拖放功能

19.2.1　最简单的拖动效果

在创建 DD 对象实例时指定元素 id 就可以实现最简单的拖动效果。

使用模板页创建一个名称为 19-1.html 的页面文件，然后先加入一个图片和一个 DIV，代码如下：

```
<img id="img1" style="position:absolute;left:10px;top:10px;" src="../images/
    s1.jpg">
<div id="div1" style="position:absolute;left:10px;top:250px;">
   <span> 节点 1</span><br/>
   <span> 节点 2</span><br/>
   <span> 节点 3</span><br/>
</div>
```

这里一定要使用绝对定位，不然元素会受其他元素定位的影响，不能随意拖动了。

最后创建两个 DD 对象实例，代码如下：

```
Ext.create(Ext.dd.DD,"img1");
Ext.create(Ext.dd.DD,"div1");
```

就是这么简单，在浏览器中打开页面，将看到如图 19-2 所示的效果。在图片或 DIV 内按下鼠标左键，就能在页面中随意拖动图片或 DIV 了。

19.2.2 使用 DragSource 对象与 DropTarget 对象

使用 DragSource 对象，一次只能拖动一个对象，例如 DIV 中有 3 张图片，要为这 3 张图片各创建一个 DragSource 对象实例，才能分别拖动这些图片，如果只在 DIV 中创建一个 DragSource 对象实例，那么 DIV 就会连带 3 张图片一起拖动。在 DropTarget 对象中则没有这个限制，只要是允许放置的对象，它都可以接收，除非自定义限制，这个可通过定义 group 或 ddGroup 配置项实现。下面通过一个示例来演示如何使用 DragSource 对象和 DropTarget 对象。示例中有三角形、正方形和圆形 3 张图片，需要将它们拖入对应的位置。

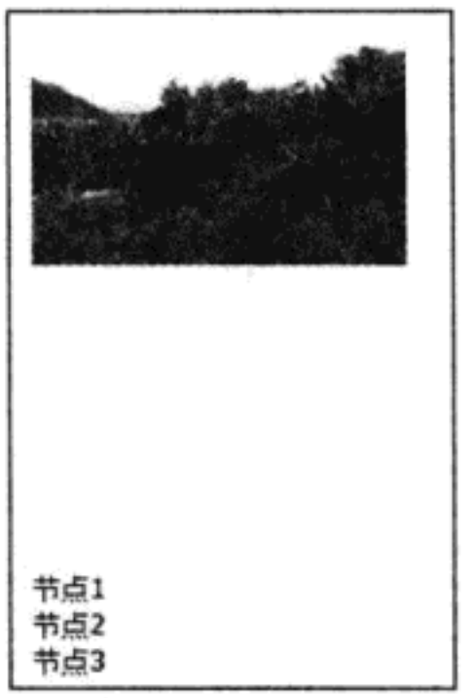


图 19-2 示例的页面效果

使用模板页创建一个名称为 19-2.html 的页面文件，然后先加入显示图片和放置图片的 HTML 代码：

```
<div id="source">
   <img id="source-t" title="三 角 形" src="../images/
       triangle.png">
           
   <img id="source-s" title="正方形" src="../images/square.png">
           
   <img id="source-c" title="圆形" src="../images/circle.png">
</div>
<div id="target">
   <span id="target-t" style="background:url(../images/triangle-none.png) no-
       repeat"></span>
   <span></span>
   <span id="target-s" style="background:url(../images/square-none.png) no-
       repeat"></span>
   <span></span>
   <span id="target-c" style="background:url(../images/circle-none.png) no-
       repeat"></span>
</div>
```

还要定义些样式，例如：

```
#source {margin:0 0 0 20px;}
#source img{width:100px;height:100px;}
#target {margin:100px 0 0 20px;}
#target span{width:100px;height:100px;display:block;float:left;}
```

现在为“source-t”创建一个 DragSource 对象实例，代码如下：

```
Ext.create(Ext.dd.DragSource,"source-t",{
group:"group-t",
getDragData:function(e){
        var sourceEl = e.getTarget();
        if (sourceEl) {
                return {
                ddel: sourceEl,
                sourceEl: sourceEl
                }
```

```
            }
        }
    });
```

使用 DragSource 对象需要提供 dragData 数据，但是对象本身并不能提供这个数据，只能自己创建，所以要重写 getDragData 方法，根据事件获取目标元素，然后生成数据返回。

现在为“target-t”创建一个 DropTarget 对象实例，用于放置“source-t”，代码如下：

```
Ext.create(Ext.dd.DropTarget,"target-t",{
    group:"group-t",
    notifyDrop:function (source, event, dragNodeData) {
            var dragged = source.dragData.ddel;
            var destinationContainer = this.getEl();
            destinationContainer.appendChild(dragged);
            return true;
    }
});
```

注意，group 配置项必须与“source-t”的一样，不然就接收不了。DropTarget 对象如何接收对象要自己定义，所以需要重写 notifyDrop 方法，这里直接使用 appendChild 将其加入 SPAN 元素内就可以了。

根据“source-t”的定义代码，创建“source-s”和“source-c”的代码，注意 group 配置项的值要分别定义为“group-s”和“group-c”。

同样，根据“target-t”的定义代码，创建“target-s”和“target-c”的代码，group 值分别为“group-s”和“group-c”。

这样，页面就完成了。在浏览器中打开页面，将看到如图 19-3 左上部分所示的效果，拖动三角形到正方形上方，会看到如图 19-3 右上角所示的效果，提示图标显示禁止放置图标，而拖动三角形到三角形位置，则会看到如图 19-3 右下角所示的效果，图标显示为允许放置图标。最终将所有图形拖动到对应位置，将看到如图 19-3 左下部分所示的效果。

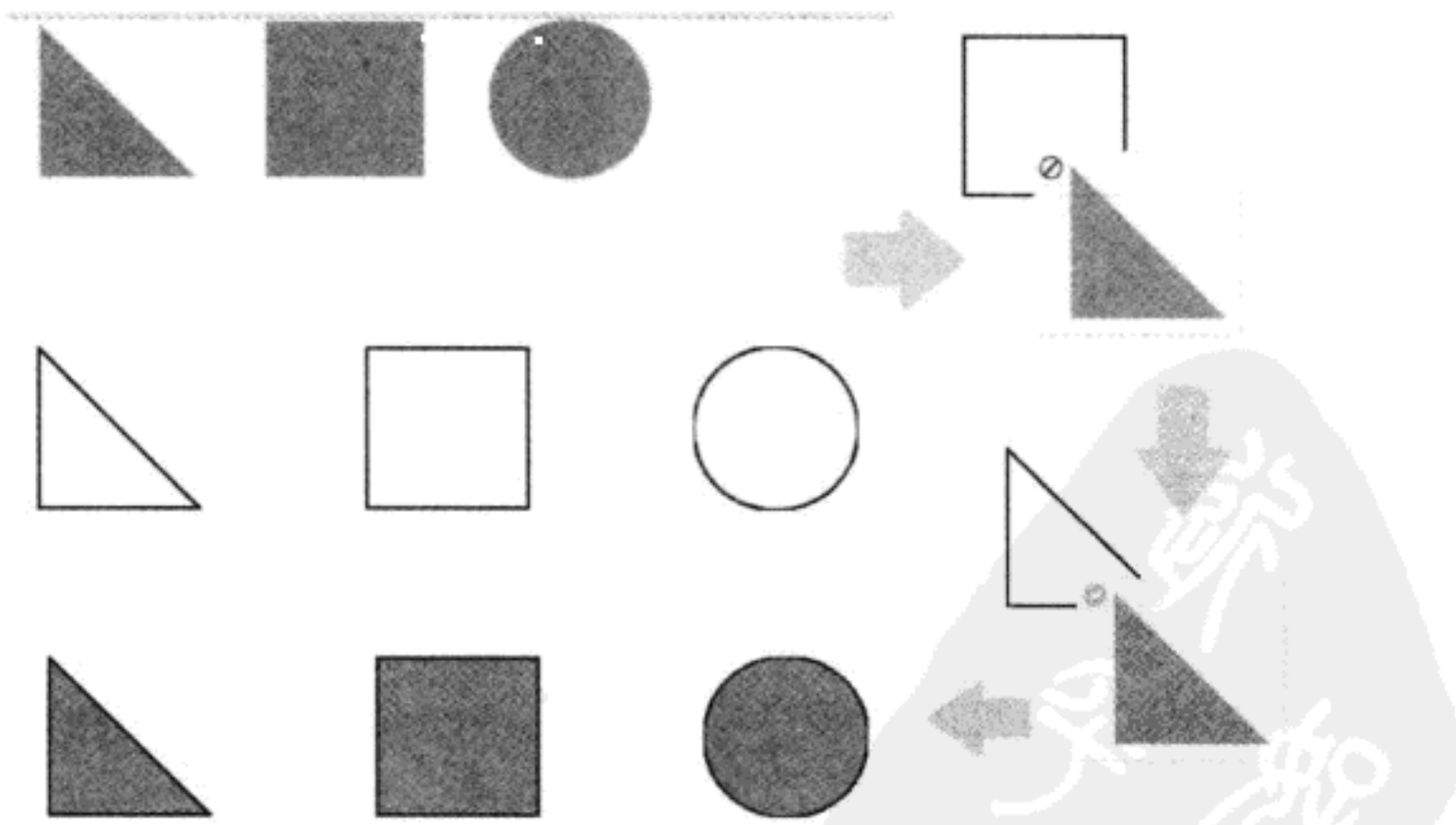


图 19-3　示例的页面效果

19.2.3 使用 DragZone 对象与 DropZone 对象（使用 Registry 对象）

在上一节中，在同一个 DIV 内的图片和 SPAN 都要分开定义，才能进行拖放操作，而使用 DragZone 对象与 DropZone 对象则不需要这样，而是通过两种方式进行处理，本节先讲述使用 Registry 对象处理的方式。

使用模板页创建一个名称为 19-3.html 的页面文件，然后，把样式和 HTML 代码复制过来，需要修改 target 的样式，修改后的代码如下：

```
#target {margin:100px 0 0 20px;width:600px;height:150px;}
```

如果不设置宽度和高度，DropZone 对象就计算不出范围，从而找不到放置容器，因而必须设置宽度和高度，而且其值必须大于子节点的总宽度和总高度。

下面定义 DragZone 对象，代码如下：

```
Ext.create(Ext.dd.DragZone,"source");
```

定义 DragZone 对象后，需要将子节点在 Registry 对象中注册，代码如下：

```
var els=Ext.query("#source>img");
for(i=els.length-1;i>=0;i--){
   Ext.dd.Registry.register(els[i]);
}
```

不注册就找不到对应的拖动对象，除非重写 getDragData 方法。

接着定义 DropZone 对象，代码如下：

```
Ext.create(Ext.dd.DropZone,"target",{
   onNodeDrop:function(dropData, source, e, dragData){
          var sid=dragData.ddel.id,tid=dropData.ddel.id;
          if(sid.substr(sid.length-1,1)==tid.substr(tid.length-1,1)){
                var sc = source.el.dom;
                sc.removeChild(dragData.ddel);
                dropData.ddel.appendChild(dragData.ddel);
                return true;
          }
          return false;
   }
});
```

DropZone 对象与 DropTarget 对象的不同，前者使用了 onNodeDrop 方法改变显示，而不是 notifyDrop 方法。

为了使图形放置正确，需要检查它们 id 中最后一位字符是否相同，如果相同，说明需要删除和插入节点，否则不做处理。

参数 dropData 和 dragData 的 ddel 属性分别指向放置节点和拖动节点，因而需要从 source 对象中移除，而添加是在放置节点内添加，可直接调用 appendChild 方法添加。

最后要将放置节点在 Registry 对象中注册，代码如下：

```
var elsT=Ext.query("#target>span[id]");
for(i=elsT.length-1;i>=0;i--){
```

```
        Ext.dd.Registry.register(elsT[i]);
    }
```

在浏览器中打开页面，可看到与上一节示例一样的效果，不过在拖动时有小小区别，就是将三角形拖动到正方形或圆形的时候也显示允许放置，如果要像上节示例一样显示不允许放置，则需要重写 DragZone 对象的 onDragOver 方法。

是否使用 Register 对象可根据自己喜好选择，没有绝对的好与不好，笔者更喜欢不使用，这样代码维护性好些。

19.2.4 使用 DragZone 对象与 DropZone 对象（不使用 Registry 对象）

不使用 Registry 对象，就要重写 DragZone 对象的 getDragData 方法和 DropZone 对象的 getTargetFromEvent 方法，使方法返回需要的 DragData 对象。

使用模板页创建一个名称为 19-4.html 的页面，将上一节示例中的样式和 HTML 代码复制过来，先定义 DragZone 对象，代码如下：

```
Ext.create(Ext.dd.DragZone,"source",{
    getDragData:function(e){
            var t = e.getTarget("img");
            if (t) {
                    return {
                    ddel: t,
                    sourceEl: t
                    }
            }
    }
});
```

通过 getTarget 方法找到对象后，根据 DragData 对象的数据结构构建一个对象返回即可。

接着定义 DropZone 对象，代码如下：

```
Ext.create(Ext.dd.DropZone,"target",{
    getTargetFromEvent:function(e){
            var t = e.getTarget("span[id]");
            if (t) {
                    return {
                    ddel: t,
                    sourceEl: t
                    }
            }
    },
    onNodeDrop:function(dropData, source, e, dragData){
            // 省略代码，参阅 19.2.3 节示例
    }
});
```

重写的 getTargetFromEvent 方法与 getDragData 方法除了选择符不同外，其余都是一样的。

在浏览器中打开页面，运行效果与上一节示例是一样的。

19.2.5 通过拖动实现节点排序

要实现节点排序，关键是要找到插入点，然后使用 insertBefore 在插入点前插入拖动节点即可，实现起来不难。

使用模板页创建一个名称为 19-5.html 的页面文件，然后加入以下 HTML 代码：

```
<ul id="list">
        <li> 节点 1</li>
        <li> 节点 2</li>
        <li> 节点 3</li>
        <li> 节点 4</li>
        <li> 节点 5</li>
        <li> 节点 6</li>
</ul>
```

再加入样式：

```
#list{margin:20px 0 0 20px;font-size:30px;line-height:40px;}
```

因为 getDragData 方法和 getTargetFromEvent 方法的代码相同，因此可先写一个公共函数 getDragData，代码如下：

```
var getDragData=function(e){
    var t = e.getTarget("li");
    if (t) {
            return {
                    ddel: t,
                    sourceEl: t
            }
    }
}
```

然后再定义 DragZone 对象，代码如下：

```
Ext.create(Ext.dd.DragZone,"list",{
    getDragData:getDragData
});
```

最后定义 DropZone 对象，代码如下：

```
Ext.create(Ext.dd.DropZone,"list",{
    getTargetFromEvent:getDragData,
    onNodeDrop:function(dropData, source, e, dragData){
            var sc = source.el.dom;
            sc.removeChild(dragData.ddel);
            sc.insertBefore(dragData.ddel,dropData.ddel);
            return true;
    }
});
```

在 onNodeDrop 方法中，删除原来节点，然后调用 insertBefore 方法将拖动对象插入到插入点之前即可。

在浏览器中打开页面，然后将节点 6 拖动到节点 2 位置放下，将看到如图 19-4 所示的变化。

19.2.6　使用 GridViewDropZonePlugin 插件

图 19-4　示例的页面效果

在 Ext JS 4 的 Grid 中使用拖放功能比在 Ext JS 3 中简单多了，只要在视图中定义好 GridViewDropZonePlugin 插件就行了，而且在插件中已经实现了通过拖动进行排序的功能，这都要归结于 GridViewDropZone 对象是使用 Store 的 insert 方法插入记录的，所以实现起来就比较简单了。以下是 GridViewDropZone 对象中记录操作的代码：

```
handleNodeDrop : function(data, record, position) {
    var view = this.view,
        store = view.getStore(),
        index, records, i, len;
    if (data.copy) {
        records = data.records;
        data.records = [];
        for (i = 0, len = records.length; i < len; i++) {
            data.records.push(records[i].copy(records[i].getId()));
        }
    } else {
        data.view.store.remove(data.records, data.view === view);
    }
    index = store.indexOf(record);
    if (position !== 'before') {
        index++;
    }
    store.insert(index, data.records);
    view.getSelectionModel().select(data.records);
}
```

如果 data 对象的 copy 属性为 true，表示要复制记录，那么使用模型的 copy 方法复制一份记录，否则，直接调用 remove 方法删除记录。

接着通过 indexOf 方法获取插入点，这里还可根据参数 position 指示在当前插入点前还是后插入。参数 position 是根据鼠标位置确定的，如图 19-5 所示的位置就表示在任务 4 之后插入，如果绿色的线在上面，则表示在任务 4 之前插入，很人性化。

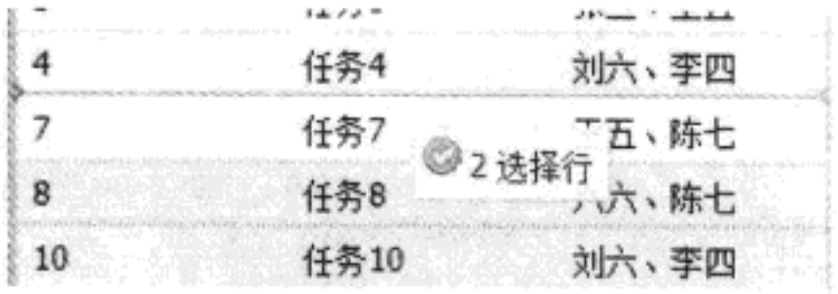


图 19-5　指示插入位置

最后调用 insert 方法插入记录，并让选择模型重新选择这些记录，非常方便实用。

那么是使用复制方式还是删除方式设置记录呢？在 ViewDragZone 对象的 getDragData 方法中，有以下代码：

```
copy: this.view.copy || (this.view.allowCopy && e.ctrlKey),
```

这说明，可以在视图中定义配置项 copy 为 true，或定义配置项 allowCopy 为 true，并且要按下 Ctrl 键，也就是说，如果定义配置项 copy 为 true，则在任何情况下都使用复制方式；

如果定义配置项 allowCopy 为 true，那么在按下 Ctrl 键的时候要使用复制方式，否则使用删除方式。

下面通过一个示例演示如何使用 GridViewDropZonePlugin 插件。使用模板页创建一个名称为 19-6.html 的页面文件，然后定义一个包含数据的 Store，代码如下：

```
Ext.create("Ext.data.ArrayStore",{
    storeId:"Store1",
    fields:[{name:"id",type:"int"},
            "task","rel"
    ],
    data:[
            [1,"任务 1","张三、李四"],
            [2,"任务 2","王五"],
            [3,"任务 3","张三、王五"],
            [4,"任务 4","刘六、李四"],
            [5,"任务 5","张三、李四"],
            [6,"任务 6","陈七、李四"],
            [7,"任务 7","王五、陈七"],
            [8,"任务 8","六六、陈七"],
            [9,"任务 9","张三、李四"],
            [10,"任务 10","刘六、李四"],
            [11,"任务 11","王五、陈七"]
    ]
});
```

再定义一个空的 Store：

```
Ext.create("Ext.data.ArrayStore",{
    storeId:"Store2",
    fields:[{name:"id",type:"int"},
            "task","rel"
    ]
});
```

最后定义一个面板，使用 HBoxLayout 将其分成两部分，用于显示两个 Grid，代码如下：

```
Ext.create(Ext.Panel,{
    width:600,height:350,
    renderTo:Ext.getBody(),
    layout:{type:"hbox",align:'stretch'},
    defaults:{
            xtype:"grid",flex:1,
            selModel:{mode:"MULTI"},
        viewConfig: {
            plugins: {
            ptype:'gridviewdragdrop',
            ddGroup:'groupGrid',
            dragText : '{0} 选择行',
        }
    },
    columns:[
            {text:"编号",dataIndex:"id",flex:1},
            {text:"任务",dataIndex:"task",flex:1},
            {text:"参与人员",dataIndex:"rel",flex:1}
```

```
            ]
        },
        items:[
                {store:"Store1"},
                {store:"Store2"}
        ]
});
```

因为对于两个 Grid 的定义，除了 Store 外都是一样的，所以可以在面板的 defaults 配置项中定义大部分代码。重点关注粗体代码中视图的配置项，没有定义 copy 配置项，说明将使用删除模式。在插件的定义中，ddGroup 配置项用于定义拖放操作的分组，在这里不定义也可以，不过如果页面中有很多 Grid，并且 Store 不同，就要定义了，不然拖放操作都在一个组了，就会允许互相进行拖放操作，由于 Store 的字段不同，会发生错误，因此定义 ddGroup 配置项是一个好的习惯。因为在本地化文件中没有对 dragText 属性进行本地化，所以需要在这里将其本地化。

在浏览器中打开页面，并随意选择一些数据进行拖动，将看到如图 19-6 所示的效果。从图 19-6 中的指示箭头可以看到，效果比 Ext JS 3 的效果人性化多了。

编号	任务	参与人员
1	任务1	张三、李四
2	任务2	王五
3	任务3	张三、王五
4	任务4	刘六、李四
5	任务5	张三、李四
6	任务6	陈七、李四
7	任务7	王五、陈七
8	任务8	刘六、陈七
9	任务9	张三、李四
10	任务10	刘六、李四
11	任务11	王五、陈七

编号	任务	参与人员

3 选择行

图 19-6　示例的页面效果

在 Grid 与 Grid 中使用拖放功能时，要注意 Store 的字段必须是相同的，只有 ddGroup、dragGroup 或 dropGroup 中定义的组名称相同的 Grid 之间才允许拖放操作。当同时定义这三个配置项时，如果 enableDrag 配置项为 true，在创建 ViewDragZone 对象实例时，会优先使用 dragGroup 的定义。如果 enableDrop 配置项为 true，在创建 GridViewDropZone 对象实例时，会优先使用 dropGroup 的定义。

19.2.7　使用 TreeViewDragDropPlugin 插件

因为在 Ext JS 4 中，树和 Grid 是同源的，所以很容易理解在树中使用拖放功能与在 Grid 中是一样的，只是数据处理上的不同而已。在树中使用拖放功能是通过 TreeViewDragDropPlugin 插件实现的，其使用方式与 GridViewDropZonePlugin 插件一样，都

是在视图中定义的，不过，其配置项比 GridViewDropZonePlugin 插件多了 7 个，以下是其特有的配置项。

- ❑ allowContainerDrop：布尔值，如果设置为 true，则视图可接收树容器。默认值为 false。
- ❑ allowParentInsert ：布尔值，如果设置为 true，则允许在父节点及其第一子节点之间插入拖动节点，该节点将变成与父节点同层的节点。默认值为 false。
- ❑ appendOnly：布尔值，如果设置为 true，树只运行追加拖动节点。默认值为 false。
- ❑ expandDelay ：数值，指定当拖动节点在目标节点上方时，目标节点准备展开时的等待时间。默认值为 1000（毫秒）。
- ❑ nodeHighlightColor ：6 位十六进制值的颜色值，不需要前导符“#”。指定拖动节点或放置节点的突出显示颜色。
- ❑ nodeHighlightOnDrop ：布尔值，指示是否在成功放置节点后突出显示节点。默认值为 Ext.enableFx。
- ❑ nodeHighlightOnRepair：布尔值，指示在修复一个不成功的拖放操作后，是否突出显示节点。默认值为 Ext.enableFx。

下面通过一个示例演示如何使用 TreeViewDragDropPlugin 插件。使用模板页创建一个名称为 19-7.html 的页面文件，然后加入以下代码：

```
Ext.create(Ext.Panel,{
    width:600,height:350,
    renderTo:Ext.getBody(),
    layout:{type:"hbox",align:'stretch'},
defaults:{
        xtype:"treepanel",flex:1,
        useArrows:true,
},
items:[
        {
         root:{
                text:"根节点",id:"tree1",
                expanded:true,
                children:[
                        {text:"节点 1",expanded:true,
                        children:[
                                {text:"节点 1-1",leaf:true},
                                {text:"节点 1-2",leaf:true},
                                {text:"节点 1-3",leaf:true}
                        ]
                },
                                {text:"节点 2",expanded:true,
                children:[
                                {text:"节点 2-1",leaf:true},
                                {text:"节点 2-2",leaf:true},
                                {text:"节点 2-3",leaf:true}
                ]
},
{text:"节点 3",expanded:false,
                children:[
                                {text:"节点 3-1",leaf:true},
```

```
                                        {text:" 节点 3-2",leaf:true},
                                        {text:" 节点 3-3",leaf:true}
                                ]
                        }
                ]
                },
        viewConfig: {
            plugins: {
                ptype:'treeviewdragdrop',
                ddGroup:"TreeGroup",
                dragText:"{0} 个选择节点 "
            }
        }
        },
        {
        root:{
                        text:" 根节点 ",id:"tree1",
                        expanded:true,
                        children:[]
                },
        viewConfig: {
            plugins: {
                ptype:'treeviewdragdrop',
                ddGroup:"TreeGroup",
                dragText:"{0} 个选择节点 "
            }
        }
        ]
    });
```

在以上代码中要注意，不能像 Grid 的定义那样，把配置项 plugins 放到 defaults 里，必须分开定义，不然两棵树将会使用相同的 Store 数据。

在浏览器中打开页面，并拖动节点，将看到如图 19-7 所示的效果。

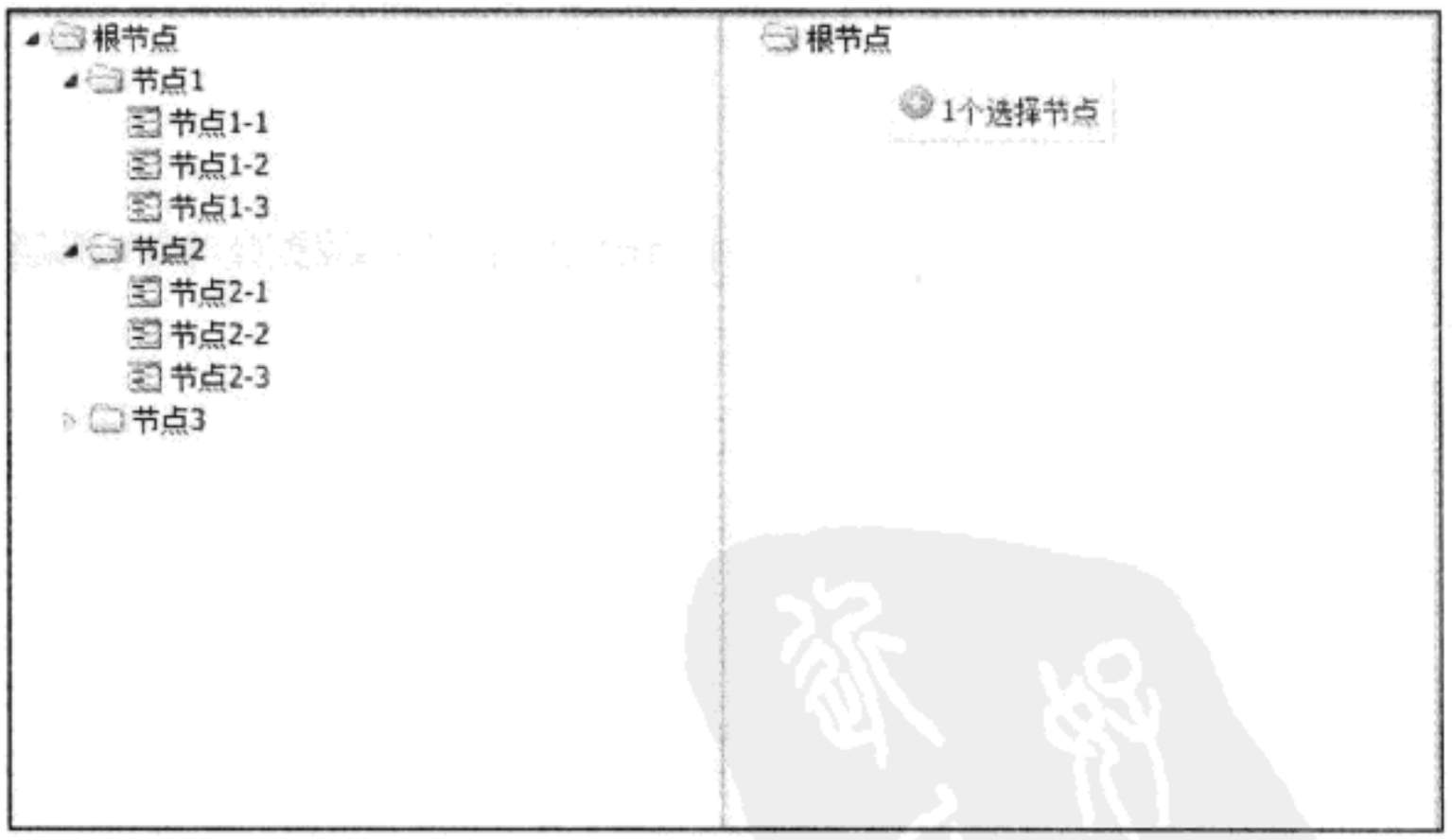


图 19-7 示例的页面效果

19.2.8 关于 Grid 和 Tree 拖动后的数据保存问题

Grid 和 Tree 的拖动，无非是 Store 数据的更改，如果不怕频繁更新，可每改动一次就使用 sync 方法同步一次。否则，增加一个保存按钮，让用户决定是否保存，这应该是比较合适的方法，因为用户操作可能有错误。

如果 store 是有排序的，则可在修改后，按顺序读出记录的编号，然后通过 Ajax 将结果发送到服务器端，或由用户通过保存按钮决定是否保存结果。

19.3 本章小结

在 Ext JS 4 中使用拖放功能相当简单，主要注意以下几点：

- 使用 DragSource 对象和 DropTarget 对象，一次只能拖动和放置一个对象，而且要重写 notifyDrop 方法实现放置操作。
- 使用 DragZone 对象与 DropZone 对象，可设置多个拖动对象与多个放置对象。可使用 Registry 对象注册节点，也可以通过重写 getDragData 方法和 getTargetFromEvent 方法来实现。要实现放置操作，需重写 onNodeDrop 方法。
- 在 Grid 和树中使用拖放功能，是通过插件实现的，而且需要在视图的配置项中定义插件。要注意，如果是在两个 Grid 或两棵树之间实现拖放操作，Store 的字段必须相同，不然数据会出错，要么重写 handleNodeDrop 方法（这个比较复杂了），要么不使用插件，直接使用 DragZone 对象与 DropZone 对象来实现。
- 要在使用 Store 的 Grid、树或视图与不使用 Store 的数据的组件之间实现拖放操作，关键是重写 getDragData 方法，提供正确的数据；或者重写 onNodeDrop 方法，处理数据。

第 20 章　扩展与插件

在一个框架中，要求开发小组把所有功能都一次性写好是不现实的，开发出适应所有情况的功能也是不现实的。因此对于一个好的框架，具有良好的扩展性是很重要的，只有这样，开发人员才能针对实际情况轻松地对其进行扩展以适应不同的场合。

Ext JS 就是这样一个具有良好扩展性的框架，自 Ext JS 2 以来，大量用户为其加入了许多扩展的组件，随着版本的更新，许多好的扩展也逐渐成为了新版本的基础组件，这正是 Ext JS 良好扩展性的体现。

本章将讲述如何书写扩展和插件，以及一些在 Ext 例子中没有提及而在开发中比较常用的扩展和插件。

20.1　扩展与插件的区别

看到扩展与插件，相信大家一定充满疑惑，为什么是扩展与插件，而不是单说扩展或单说插件呢？它们有什么相同的地方，又有什么不同的地方呢？

其实扩展和插件都是针对基准库进行修改或扩充来实现新的功能，它们都不能独立使用，必须依赖基准库的组件或类运行。扩展在 Ext JS 中是指从基类中衍生的子类，它不但继承了基类的属性和方法，还增加了自己的属性和方法。例如 Ext JS 常用的模型，它从 Model 对象扩展，既包含了 Model 对象的属性、方法和事件，又包含自定义的属性和方法，字段就是其自定义的属性。

插件是为 Ext 类服务的类，它不像扩展那样，需要从组件中继承属性和方法，只是为组件添加一些附加功能。

20.2　扩展与插件如何选择

根据扩展与插件的区别，如果不需要从父类继承属性和方法，不需要新增或重写属性和方法，只是为组件添加一个功能，那么就使用插件，否则使用扩展。

20.3　如何编写扩展

20.3.1　命名空间

根据 Ext JS 的命名规则，一般扩展都是以“Ext.ux”作为命名空间的，在其后加入新对象

的名称。当然，也可以使用自己的命名空间，不过，要使用自己的命名空间，必须在扩展定义前加入命名空间的定义，以免找不到对象。

20.3.2 定义扩展

定义扩展，其实在前面章节经常用到，定义的模型就是一个扩展。是的，扩展就是这么简单，在对象的定义中加入 extend 配置项就行了，具体语法如下：

```
Ext.define('Ext.ux.classNmae',{
    extend:"ParentClass",
    ......
})
```

其中，className 是所定义的扩展的类名，ParentClass 是要继承的类的名称。

当然，也有特例，由第 4 章中内容可知，所有继承类的基类都是 Ext.Base 类，因而，如果想直接从 Ext.Base 类继承，则不需要在定义中加入 extend 配置项。

20.3.3 定义别名

别名（alias）的主要作用是可使用 xtype 配置项来创建组件。要定义别名，需要在定义扩展时定义 alias 配置项，其值可以是单个别名的字符串，也可以是由多个别名组成的数组，而别名是由“widget”做前缀，加上小数点，再加别名构成的，例如 Grid 的别名定义：

```
alias: ['widget.gridpanel', 'widget.grid'],
```

20.3.4 定义备用名

备用名（alternateClassName）的作用是简化原类名的名称，作为另一类型的类的名称或兼容旧版本的类名，它使用 alternateClassName 配置项进行定义，值可以是单个类名的字符串，也可以是由多个备用名组成的数组，Grid 的备用名定义如下：

```
alternateClassName: ['Ext.list.ListView', 'Ext.ListView', 'Ext.grid.GridPanel'],
```

20.3.5 要求加载的类：requires 与 uses

当使用动态加载的时候，要实例化当前扩展时需要先加载的类，需要通过配置项 requires 来明确要先加载哪些类，例如 Grid 要求的类：

```
requires: ['Ext.grid.View'],
```

这说明在实例化 Grid 之前必须先加载“Ext.grid.View”。

如果不是实例化时要求先加载的类，可在 uses 配置项中进行定义，例如 FieldSet 使用到的类：

```
uses: ['Ext.form.field.Checkbox', 'Ext.panel.Tool', 'Ext.layout.container.Anchor',
    'Ext.layout.component.FieldSet'],
```

以上两个配置项的值都是由类名组成的数组。

20.3.6　混入功能

如果要在扩展中实现混入功能（mixins），可使用 mixins 配置项，例如要在扩展中添加自定义的事件，则可混入 Observable 对象，代码如下：

```
mixins: {observable: 'Ext.util.Observable'}
```

配置项 mixins 指向一个对象，对象的属性名称可以自定义，其值是要混入的类名。如果有多个混入的类，则在对象中添加属性，并指向这些类即可了。

混入类的属性和方法会直接成为扩展的属性和方法。

20.3.7　构造函数与 initComponent 方法

构造函数（constructor）不是必须的，当创建实例时，与父类有不同的初始化过程的实例才需要重写构造函数。

从组件扩展出新的组件，一般不重写构造函数，只需要使用 initComponent 方法对组件进行初始化。

20.3.8　静态属性和方法与单件模式

静态属性和方法（statics、inheritableStatics）是不需要实例化类就可使用的属性和方法，实际上就是把属性和方法直接定义在类中，而不是其原型中。通过配置项 statics 可定义静态属性和方法，此配置项会指向一个对象，对象的属性名称就是属性的名称或方法名称，值就是属性值或方法的函数定义。例如，在 Model 类中定义的静态属性和方法：

```
statics: {
    PREFIX : 'ext-record',
    AUTO_ID: 1,
    EDIT   : 'edit',
    REJECT : 'reject',
    COMMIT : 'commit',

    id: function(rec) {
        var id = [this.PREFIX, '-', this.AUTO_ID++].join('');
        rec.phantom = true;
        rec.internalId = id;
        return id;
    }
},
```

在这里共定义了 5 个属性和一个方法。

使用 statics 配置项定义的静态属性和方法在类实例化后，就不能再使用了，原因是这些属性和方法不在原型中，而实例化是从原型中创建对象，所以这些属性和方法就会丢失。如果希望在实例化后静态属性和方法能在实例中使用，可使用 inheritableStatics 配置项，它的定义与 statics 的定义是一样的。

单件模式（singleton）的类是类的一个实例，所以其属性和方法可直接使用。在配置项中

加入 singleton 配置项并设置其值为 true，表示该扩展将返回它的一个实例。

在这里，大家一定会有这样的疑问，使用全静态方法的类好还是使用单件模式的类好？这方面争议很多，笔者一般是根据具体情况选择便利的方式，不过原则是能用单件模式尽量用单件模式，遵循 OO 守则是不会错的。

20.3.9 可自动生成 set 和 get 方法的属性与 initConfig 方法

在 config 属性内定义的属性，会自动生成 set 和 get 方法，其值会指向一个对象，对象内的属性名称就是要定义的属性，其值可通过自动生成的 set 和 get 方法进行设置和获取。例如在 Test 对象内，定义了以下属性：

```
config:{
   width:100,
   height:100
}
```

就可通过以下方法设置和访问 width 和 height 的值：

```
var test=new Test();
test.setWidth(200) //width 现在是 200
test.setHeight(200) //height 现在也是 200
test.getWidth()  // 返回 200
test.getHeight() // 返回 200
```

定义了 config 配置项还不行，还需要调用 initConfig 方法才会生成 set 和 get 方法，该方法一般在构造函数内使用。

20.3.10 在扩展中常用的方法

在扩展中经常会用到以下几个方法。

- apply：用来复制对象。
- applyIf：也用来复制对象，不过不会复制原对象中已存在的属性。
- callParent：调用父类同名的方法，要注意参数。
- addEvents ：在混入了 Observable 对象或从 Observable 对象继承的扩展，可使用该方法添加自定义事件。

20.3.11 编写扩展：TreeComboBox

本节将利用 ComboBox，依样画葫芦，从 Ext.form.field.Picker 类派生出一个 TreeComboBox 类，用于在下拉列表中选择树节点作为值。

在开始之前首先要分析 TreeComboBox 与 ComboBox 有哪些异同点，哪些属性和方法可以直接复制过来。

TreeComboBox 与 ComboBox 都要使用 Store，这是肯定的，但是 TreeComboBox 使用的是 TreeStore，由此可以确定 displayField 将是 text，而 valueField 是 id，这与 ComboBox 可以

自定义 displayField 和 valueField 不同，并且 TreeStore 查找节点的操作与 Store 是不同的，所以查找记录的方法要改。

TreeComboBox 要显示的对象为树，不是 BoundList，所以 createPicker 方法要改。

为了简单起见，TreeComboBox 将不实现查询功能，并且不允许编辑，所以与这些相关的属性和方法都不需要。

现在可以开始工作了，创建一个名称为 TreeCombobox.js 的脚本文件，首先加入“Ext.ux”的命名空间和类定义，代码如下：

```
Ext.ns("Ext.ux");
Ext.define('Ext.ux.TreeComboBox', {
    extend:'Ext.form.field.Picker',
});
```

加入命名空间是为了避免出现错误，以防万一。根据 Ext JS 的习惯将 TreeComboBox 定义在“Ext.ux”的命名空间下。与 ComboBox 一样，TreeCombBox 从 Ext.form.field.Picker 类扩展。

接着加入别名、备用名和要求类的定义：

```
requires: [ 'Ext.EventObject', 'Ext.tree.Panel', 'Ext.data.StoreManager'],
alternateClassName: ['Ext.form.TreeComboBox',"Ext.form.field.TreeComboBox"],
alias: ['widget.treecombobox', 'widget.treecombo'],
```

如果不清楚需要加载哪些类，从 ComboBox 类中复制过来，然后在最后的代码中找到那些没有使用的类，将其删除即可。这里要显示树，所以一定要把 TreePanel 加入。

通过备用名可以把 TreeComboBox 加入到字段空间下，这样便于使用。别名的定义则通过 TreeComboBox 将 xtype 定义为 treecombobox 或 treecombo 来创建 TreeComboBox 实例。

现在把 ComboBox 中的属性复制过来，把不需要的删除，并加入扩展需要的一些定义，最后的属性定义代码如下：

```
triggerCls: Ext.baseCSSPrefix + 'form-arrow-trigger',
multiSelect: false,
delimiter: ', ',
displayField:'text',
valueField:'id',
editable:false,
defaultListConfig: {
    emptyText: '',
    maxHeight: 300
},
ignoreSelection: 0,
```

在这里指定了 displayField 为 text，valueField 为 id，就不需要开发人员进行设置了。设置 editable 为 false，这样用户就只能从树中选择数据，不允许编辑。在属性 default-ListConfig 内，配置项 maxHeight 的作用是设置下拉的最大高度是 300 像素，如果没有设置 height，则使用该值作为树面板的高度。

接着把 initComponent 方法复制过来，因为不需要转换功能和查询功能，所以保留以下代

码就足够了。

```
initComponent: function() {
    var me = this,
        isDefined = Ext.isDefined;
        store = me.store;


    //<debug>
    if (!store ) {
        Ext.Error.raise('请定义 Store.');
    }
    //</debug>

    this.addEvents('select');
    me.bindStore(store, true);
    store = me.store;

    if (!me.displayTpl) {
        me.displayTpl = Ext.create('Ext.XTemplate',
            '<tpl for=".">' +
                '{[typeof values === "string" ? values : values.' + me.display-
                    Field + ']}' +
                '<tpl if="xindex < xcount">' + me.delimiter + '</tpl>' +
            '</tpl>'
        );
    } else if (Ext.isString(me.displayTpl)) {
        me.displayTpl = Ext.create('Ext.XTemplate', me.displayTpl);
    }

    me.callParent();

    if (me.store.getCount() > 0) {
        me.setValue(me.value);
    }

},
```

接着把 initComponent 方法中要调用的 bindStore 方法复制过来，并修改为以下代码：

```
bindStore: function(store, initial) {
    var me = this;
    if (store) {
        me.store = Ext.data.StoreManager.lookup(store);
        if (me.picker) {
            me.picker.bindStore(store);
        }
    }
},
```

只需要为树绑定 Store 就好了，不进行其他设置。

接着复制 createPicker 方法，并修改为如下代码：

```
createPicker: function(){
    var me = this,
```

```
        picker,
        menuCls = Ext.baseCSSPrefix + 'menu',
        opts = Ext.apply({
            selModel: {
                mode: me.multiSelect ? 'SIMPLE' : 'SINGLE'
            },
            floating: true,
            hidden: true,
            renderTo:Ext.getBody(),
            ownerCt: me.ownerCt,
            cls: me.el.up('.' + menuCls) ? menuCls : '',
            store: me.store,
            focusOnToFront: false,
        }, me.treeConfig, me.defaultListConfig);

        picker = me.picker = Ext.create(Ext.tree.Panel, opts);
        me.mon(picker, {
            itemclick: me.onItemClick,
            refresh: me.onListRefresh,
            scope: me
        });

        me.mon(picker.getSelectionModel(), 'selectionchange', me.onListSelection-
            Change, me);

        return picker;
    },
```

因为树要根据字段调整显示位置，所以要设置 floating 为 true，并且隐藏初始状态，将它渲染到页面 body 内，就不会受其他的父元素的限制，便于移动。现在创建对象是 TreePanel，因而需要修改 create 方法内的对象。绑定事件的方法不用修改，保留就行。

将方法 onListRefresh、onItemClick、onListSelectionChange 和 select 直接复制过来，不需要做任何修改，操作是一样的。

根据值查找记录的方法只保留 findRecordByValue 方法就行，其代码如下：

```
findRecordByValue: function(value) {
    return this.store.getNodeById(value);
},
```

TreeStore 只支持 getNodeById 方法寻找节点，因此只使用一个方法就行了。

将方法 setValue、getDisplayValue、getSubmitValue、isEqual 和 clearValue 方法直接复制过来，处理过程一样，不需要修改。

将方法 syncSelection 复制过来，需要将其中判断 Store 是否存在该记录的代码“me.store.indexOf(value）>= 0”修改为以下代码：

```
me.store.getNodeById(value.get("id"))
```

原因是 TreeStore 中没有 indexOf 方法，只能通过 id 返回节点判断该节点是否存在。

还有一个至关重要的方法需要修改，在 Ext.form.field.Picker 类中，alignPicker 方法会调用 setSize 方法设置下拉组件的宽度和高度，原方法是根据 Store 的 getCount 方法返回的记录总数

决定高度，而 TreeStore 中没有该方法，高度会是 null，所以必须重写该方法。将 alignPicker 方法复制过来，然后将以下这段代码：

```
picker.setSize(me.bodyEl.getWidth(), picker.store && picker.store.getCount() ?
    null : 0);
```

修改为：

```
picker.setSize(me.bodyEl.getWidth(), picker.height ? picker.height : picker.
    maxHeight);
```

如果开发人员定义了 height 配置项，则使用该配置项的值作为高度，否则使用 maxHeight 的值作为高度。

至此，扩展 TreeComboBox 就完成了。现在可以通过一个示例测试一下这个扩展了。使用模板页创建一个名称为 20-1.html 的页面文件。首先在 head 部分加入一个引用 TreeComboBox.js 的 script 标记，代码如下：

```
<script type="text/javascript" src="TreeComboBox.js"></script>
```

接着定义一个 TreeStore：

```
Ext.create(Ext.data.TreeStore,{
    storeId:"Store1",
    fields:["id","text"],
     root:{
            text:"根节点",id:"tree1",
            expanded:true,
    children:[
            {id:"1",text:"节点 1",expanded:true,
                    children:[
                            {id:"4",text:"节点 1-1",leaf:true},
                            {id:"5",text:"节点 1-2",leaf:true},
                            {id:"6",text:"节点 1-3",leaf:true}
                    ]
            },
            {id:"2",text:"节点 2",expanded:true,
                    children:[
                            {id:"7",text:"节点 2-1",leaf:true},
                            {id:"8",text:"节点 2-2",leaf:true},
                            {id:"9",text:"节点 2-3",leaf:true}
                    ]
            },
            {id:"3",text:"节点 3",expanded:false,
                    children:[
                            {id:"10",text:"节点 3-1",leaf:true},
                            {id:"11",text:"节点 3-2",leaf:true},
                            {id:"12",text:"节点 3-3",leaf:true}
                    ]
            }
    ]
    }
});
```

TreeStore 定义必须有 id 和 text 两个字段。

接着定义一个 FormPanel，在其中放置两个 TreeComboBox，分别用于测试单选和多选两种情况。每个 TreeComboBox 后放置两个按钮，一个用于执行 setValue 方法设置值，一个用于执行 getValue 方法返回值。为了便于在一行中显示 TreeComboBox 和两个按钮，用 FieldContainer 将它们分隔，具体代码如下：

```
Ext.create(Ext.form.Panel,{
    title:"示例 20-1 编写扩展：TreeComboBox",
    renderTo:Ext.getBody(),
    width:500,height:100,
    bodyPadding:"5",
    defaults:{
        xtype:"fieldcontainer",labelWidth:80,
        layout:{type:"hbox"},
        defaults:{margins:'0 5 0 5'}
    },
    items:[
        {fieldLabel:"多选",items:[
            {xtype:"treecombo",anchor:"-5",
                store:"Store1",flex:1,name:"field1",
                treeConfig:{
                    width:200,height:200,
                    useArrows:true
                }
            },
            {xtype:"button",text:"setValue",handler:function(){
                var form=this.up("form").getForm();
                form.findField("field1").setValue("5");
            }},
            {xtype:"button",text:"getValue",handler:function(){
                var form=this.up("form").getForm();
                console.log(form.findField("field1").getValue());
            }},
        ]},
        {fieldLabel:"单选",items:[
            {xtype:"treecombo",anchor:"-5",multiSelect:true,
                store:"Store1",flex:1,name:"field2",
                treeConfig:{
                    width:200,height:200,
                    useArrows:true
                }
            },
            {xtype:"button",text:"setValue",handler:function(){
                var form=this.up("form").getForm();
                form.findField("field2").setValue(["5","7","9"]);
            }},
            {xtype:"button",text:"getValue",handler:function(){
                var form=this.up("form").getForm();
                console.log(form.findField("field2").getValue());
            }},
        ]}
    ]
})
```

完成以上定义后，在浏览器中打开页面，并展开第一个 TreeComboBox，将看到如图

20-1 中数字 1 所在位置的效果。选择“节点 1-2”，在第二个 TreeComboBox 上选择“节点 1-2”、“节点 2-1”，和“节点 2-2”，将看到如图 20-1 中数字 2 所在位置的效果。依次从上往下单击“getValue”按钮，可在控制台上看到如下输出：

```
5
["5", "7", "8"]
```

正是所选的 id 值。

从上往下依次单击“setValue”按钮，并展开第二个 TreeComboBox，将看到如图 20-1 中数字 3 所在位置的效果。

以上测试说明扩展的 TreeComboBox 已经基本可用，不过是否实用，还要进一步测试。

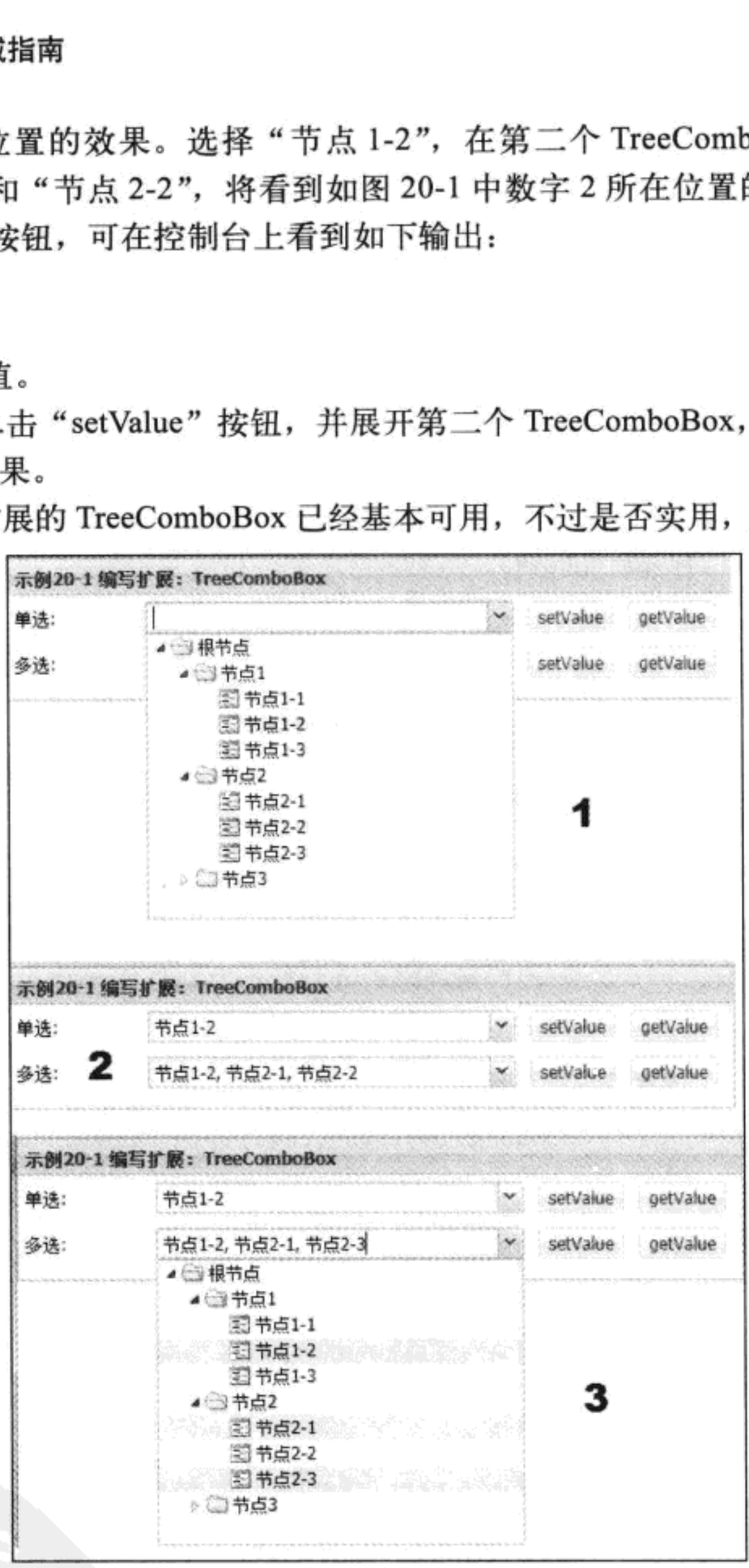


图 20-1　示例的页面效果

20.4　如何编写插件

20.4.1　概述

在 Ext JS 4 中规范了插件的编写，必须定义 init 方法用于初始化插件，定义 destroy 方法

用来销毁在插件中创建的对象。在一般情况下，插件都应以 AbstractPlugin 对象为基类。当然，也可以不从 AbstractPlugin 对象中继承，不过必须遵守插件的编写规范，因为插件的运作是遵循该规范运行的。

20.4.2　AbstractPlugin 对象

AbstractPlugin 对象的源代码如下：

```
Ext.define('Ext.AbstractPlugin', {
disabled: false,
constructor: function(config) {
    //<debug>
    if (!config.cmp && Ext.global.console) {
        Ext.global.console.warn("Attempted to attach a plugin ");
    }
    //</debug>
    Ext.apply(this, config);
},

getCmp: function() {
    return this.cmp;
},
init: Ext.emptyFn,
destroy: Ext.emptyFn,
enable: function() {
    this.disabled = false;
},
    disable: function() {
        this.disabled = true;
    }
});
```

从源代码可以看到，其构造函数规定了插件的配置对象必须定义 cmp 配置项，不然会发送警告信息。一般情况下，在组件创建插件实例时会加入 cmp 的配置，因而不用担心这问题。最后会将配置对象中的配置项复制为插件实例的属性或方法。

在其中定义了 disable 属性，用于禁用或开启插件，为了修改该值，定义了 enable 和 disable 方法。

而 getCmp 方法，用于返回 cmp 属性指向的使用插件的组件。

方法 init 和 destroy 都是空函数，这两个都是其子类要重写的方法，也是插件最重要的两个方法，init 用于进行初始化操作，而 destroy 方法则用于销毁插件中创建的对象。

20.4.3　插件的别名

必须为插件定义别名，原因是插件一般是通过 PluginManager 的 create 方法创建的，而该方法会调用 createByAlias 方法创建实例。如果不定义别名，就只能先创建插件，再放到 plugins 配置项中，不过，这不是好的主意。

插件的别名的前缀是“plugin”，这与扩展的别名前缀不同，要注意。

20.4.4 编写插件：RowColor

本节将编写一个为 Grid 行添加背景的插件 RowColor。在开始前，首先要确定该插件会应用于哪个组件，要对哪个组件进行操作。根据第 10 章的相关内容，可以知道，该插件必须在视图中使用，因此其操作的对象是视图，而插入点是 Store 触发 refresh 事件时。

因为要使用视图的 addRowCls 方法为行添加样式，以达到修改背景颜色的效果，所以需要由开发人员定义一个样式数组，在处理不同结果时应用不同的样式。检测函数也必须由开发人员进行定义，在这里涉及使用哪个数据的问题，因此需要开发人员定义使用哪个字段进行检测，然后从视图的 Store 中提取出该字段值，传送到检测函数内进行比对。最后根据返回值，决定使用数组中哪个值作为行的样式。

基本原理明确后，就可以开始工作了，创建一个名称为 RowColor.js 的脚本文件，先加入基本定义：

```
Ext.define('Ext.ux.RowColor', {
   extend:"Ext.AbstractPlugin",
    alias: 'plugin.rowcolor',
    alternateClassName: 'Ext.view.RowColor',
});
```

别名一定要正确，不然会出错。

接着定义两个必需的属性和方法：

```
cls:["",""],
field:"",
check:Ext.emptyFn(),
```

属性 cls 的作用是作为配置项来定义样式列表。属性 field 是用来定义检测的字段。check 则用于定义检测函数。

现在定义 init 方法，主要工作是绑定 refresh 事件到 onRefresh 方法，代码如下：

```
init: function(){
   this.cmp.on("refresh",this.onRefresh,this);
},
```

最后是完成 onRefresh 方法，代码如下：

```
onRefresh:function(){
   var me=this,
          store=me.cmp.store,
          field=me.field,
          cls=me.cls;
   store.each(function(data){
          var index=me.check(data.get(field));
          if(Ext.isNumber(index) && index>=0){
                me.cmp.addRowCls(data,cls[index])
          }
   },me);
}
```

在代码中会枚举 Store 中的记录，在枚举函数内，通过 get 方法获取字段值后，将其传递给 check 方法进行检测，然后根据返回的值 index 对当前行应用样式。

至此，插件就编写完了。现在创建一个名称为 20-2.html 的页面文件，用于测试插件。从 10.2.3 节示例中把样式和脚本复制过来，不要复制 featrues 和 viewConfig 的代码。

接着在 head 部分加入对插件的引用代码：

```
<script type="text/javascript" src="RowColor.js"></script>
```

重新在 GridPanel 加入以下使用插件的 viewConfig 代码：

```
viewConfig:{
   plugins:[{
           ptype:"rowcolor",
           field:"birthday",
           cls:["","green"],
           check:function(v){
                   var n=new Date(),
                   age=n.getFullYear()-v.getFullYear();
                   if(age>60) return 1;
                   return 0;
           }
   }]
},
```

在以上代码中，check 函数返回的索引一定要与 cls 数组中的索引对应，以避免出现错误，当然，也可以在插件中加入控制代码，当索引超出数组范围时，使用默认值或抛出错误，在这里就不讨论了，有兴趣的读者自己研究一下。

因为在插件中没有创建其他类的实例，所以不需要进行销毁操作，不需要重写 destroy 方法。

完成以上编写后，在浏览器中打开页面，将看到如图 10-3 的效果，Grid 的第一行背景颜色将会是绿色的，这说明插件已经运作正常了。

20.5　扩展和插件介绍

20.5.1　概述

在 Ext JS 4 的库文件中，在 examples\ux 目录下有不少扩展和插件，有些还是比较常用的扩展和插件，表 20-1 列出了这些文件对应的类名及其说明。

20.5.2　本地分页代理：Ext.ux.data.PagingMemoryProxy

经常有人会问，本地数据怎么分页？笔者就会很疑惑，本地还需要分页？不过有需求，就会有扩展。在 Ext JS 4 的包中，目录 examples\ux\data 中有个 PagingMemoryProxy.js 文件，该文件中定义了扩展自 MemoryProxy 的 Ext.ux.data.PagingMemoryProxy，这是一个代理，它在从 Reader 中获取数据时进行分页。其使用方法就是，在定义 Store 的 proxy 时设置 type 为 pagingmemory，并且在 Store 中加入 pageSize 配置项。

表 20-1 Ext JS 4 自带扩展和插件说明

文 件 名	类 名	说 明
BoxReorderer.js	Ext.ux.BoxReorderer	插件，可在工具栏中通过拖放操作改变按钮位置
CheckColumn.js	Ext.ux.CheckColumn	扩展，实现复选框列
DataViewTransition.js	Ext.ux.DataViewTransition	插件，可在视图中实现动画过渡
FieldReplicator.js	Ext.ux.FieldReplicator	插件，可克隆一个表单字段并不断复制它们，直到最后一个为空白才结束
GMapPanel.js	Ext.ux.GMapPanel	扩展，用于实现地图功能
GroupTabPanel.js	Ext.ux.GroupTabPanel	扩展，用于对标签页进行分组
LiveSearchGridPanel.js	Ext.ux.LiveSearchGridPanel	扩展，支持在线搜索的 Grid
PreviewPlugin.js	Ext.ux.PreviewPlugin	插件，在 Grid 行中实现预览功能
ProgressBarPager.js	Ext.ux.ProgressBarPager	扩展，使用进度条进行分页
RowExpander.js	Ext.ux.RowExpander	插件，在 Grid 行中实现折叠功能
SlidingPager.js	Ext.ux.SlidingPager	插件，使用滑块代替分页工具条中的文本字段
Spotlight.js	Ext.ux.Spotlight	扩展，为组件或元素提供 spotlight 功能
TabCloseMenu.js	Ext.tab.TabCloseMenu	插件，为标签提供关闭菜单
TabReorderer.js	Ext.ux.TabReorderer	插件，为标签提供重排功能
TabScrollerMenu.js	Ext.ux.TabScrollerMenu	插件，为标签提供滚动菜单
ToolbarDroppable.js	Ext.ux.ToolbarDroppable	插件，允许拖动一个条目到工具栏使其成为一个新的工具栏条目
PagingMemoryProxy.js	Ext.ux.data.PagingMemoryProxy	扩展，用于实现本地数据分页读取数据
Animated.js	Ext.ux.DataView.Animated	插件，在视图中实现动画过渡
Draggable.js	Ext.ux.DataView.Draggable	混入功能，在视图中混入拖动功能
DragSelector.js	Ext.ux.DataView.DragSelector	插件，在视图中实现通过拖动选择记录
LabelEditor.js	Ext.ux.DataView.LabelEditor	插件，在视图中实现编辑标题功能
ItemSelector.js	Ext.ux.form.ItemSelector	扩展，条目选择器字段
MultiSelect.js	Ext.ux.form.MultiSelect	扩展，可实现多选的字段
SearchField.js	Ext.ux.form.SearchField	扩展，类似搜索引擎中的搜索下拉列表框
FiltersFeatrue.js	Ext.ux.grid.FiltersFeatrue	特性，在 Grid 列标题的菜单中加入过滤特性
TransformGrid.js	Ext.ux.grid.TransformGrid	扩展，可将 HTML 的表格转换为 Grid
Center.js	Ext.ux.layout.Center	扩展，可将组件居中显示
StatusBar.js	Ext.ux.statusbar.StatusBar	扩展，状态栏
ValidationStatus.js	Ext.ux.statusbar.ValidationStatus	扩展，可用于显示表单字段错误信息的状态栏

注：在 Ext.ux.xxx.ClassName 中，xxx 表示目录，例如 Ext.ux.statusbar.ValidationStatus 中，statusbar 表示目录。

下面通过一个示例来演示如何使用 PagingMemoryProxy 对象。使用模板页创建一个名称为 20-3.html 的页面文件，把 10.7.2 节示例的样式定义和脚本先复制过来。然后在 Ext.require 中加入“'Ext.ux.data.PagingMemoryProxy'”，使页面动态加载 PagingMemoryProxy 对象。

因为数据不能使用 url 加载，所以必须定义在页面中，从 pohto-data.js 文件中把 data 部分复制到页面，然后修改为：

```
var data=[
    // 省略数据，该部分不变
]
```

在 Store 定义中，添加 pageSize 配置项，值为 5。配置项 autoLoad 必须保持，不然不会加载数据。将 proxy 中的 type 值修改为 pagingmemory；把 url 删除；添加 data 配置项，值为 data。

在 Grid 的定义中，向其底部工具栏添加一个分页工具栏，具体代码如下：

```
bbar:{xtype:"pagingtoolbar",store:"phoneStore"},
```

这样，本地分页就实现了。

在浏览器中打开页面，将看到如图 20-2 所示的页面效果，分页工具条将数据分成了 4 页。

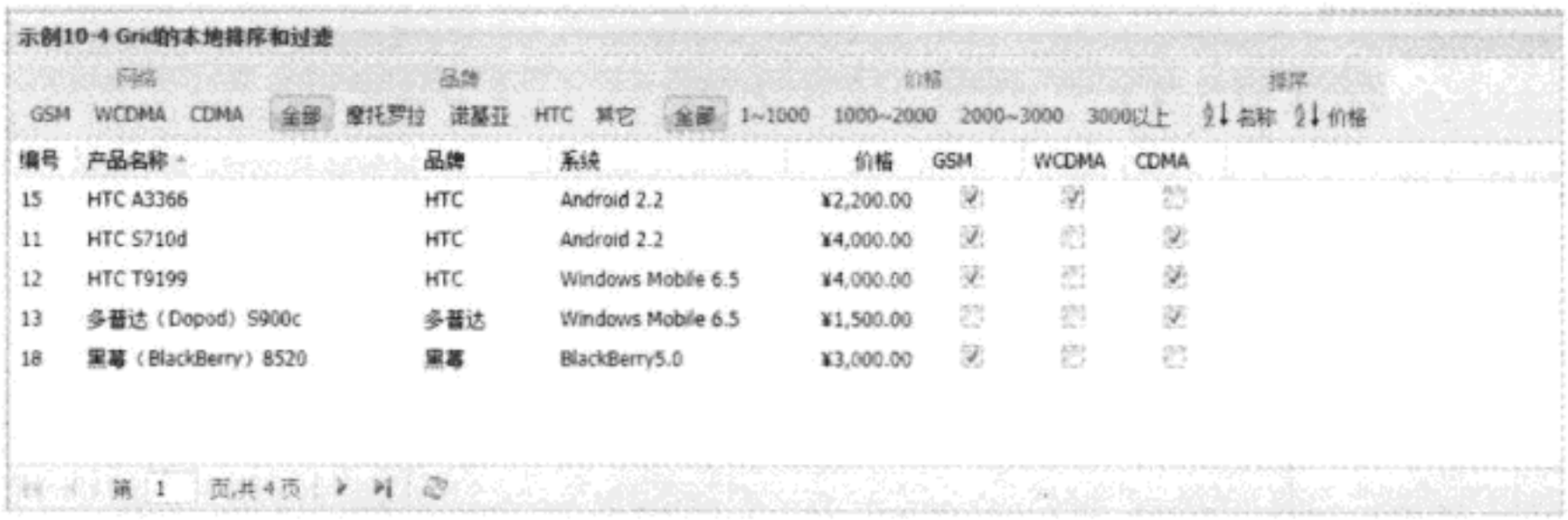


图 20-2　示例的页面效果

20.5.3　标签滚动菜单：Ext.ux.TabScrollerMenu

Ext.ux.TabScrollerMenu 是一个插件，当标签页的标签超过标签容器时，会显示一个菜单，通过菜单可选择标签页。该插件位于 examples\ux 目录，名称为 TabScrollerMenu.js。下面通过一个示例来演示如何使用该插件。

使用模板页创建一个名称为 20-4.html 的页面文件，先添加对 TabScrollerMenu.css 文件的引用，此时菜单需要这个样式文件。

```
<link rel="stylesheet" type="text/css" href="../Ext4/examples/ux/css/TabScroller-
    Menu.css"/>
```

从示例 20-3 中，把动态加载的代码复制过来，删除 require 中要求加载的文件，然后加入“Ext.ux.TabScrollerMenu”。

最后定义一个 TabPanel，先加入一个标签页，然后监听 afterRender 事件，在对其进行渲染后，通过循环为其添加 20 个标签，具体代码如下：

```
Ext.create(Ext.tab.Panel,{
    renderTo:Ext.getBody(),
    width:300,height:300,
    activeTab:0,layout:"fit",
    plugins: [{
```

```
            ptype: 'tabscrollermenu',
            menuPrefixText:"标签"
    }],
    items: [{
            title: '标签1',
            html: '标签1'
    }],
    listeners:{
            afterRender:function(){
                    var me=this;
                    for(var i=0;i<20;i++){
                        me.add({
                                title:"标签"+(i+2),
                                html:"标签"+(i+2)
                        });
                    }
            }
        }
    });
```

在以上代码中，在默认情况下，每 10 个标签就会组成一个子菜单，而配置项 menuPrefix-Text 的作用就是显示子菜单中的文本。

在浏览器中打开页面，然后展开菜单，将看到如图 20-3 所示的效果。

图 20-3　示例页面效果

20.5.4　编辑器 TinyMCE

Ext JS 自带的编辑器功能太简单了，因此很多时候需要结合开源的编辑器来开发自己的扩展。在目前开源的编辑器中，笔者认为做得相当不错的是 TinyMCE，也是用户比较多的一个编辑器。大家可以到 http://www.tinymce.com/download/download.php 下载最新的 3.4.5 版，同时记得下载语言包。

从 Ext JS 3.x 起，一直有 TinyMCE 的扩展，不过目前开发者还没有将其修改成 Ext JS 4 格式的扩展，虽然能用，但会有点小麻烦。在官方论坛发布的 http://www.sencha.com/forum/showthread.php?139663-Another-Ext.ux.form.TinyMCE 中，Ontho 提供了一个基于 Ext JS 4 扩展的 TinyMCE 字段供用户使用。

把帖子中的代码复制到脚本文件中，存为 TinyMCE.js 文件，再把 TinyMCE 包中的

tiny_mce 目录复制过来，然后在页面中引用脚本文件就可以使用了。代码中有个小错误，在 initTinyMce 方法中，要用 applyIf 方法复制 standardConfig 到 config 中，而不是用 apply 方法，因为 apply 方法会覆盖 config 中的配置。

使用模板页创建一个名称为 20-5.html 的页面文件，在 HEAD 部分加入 TinyMCE 扩展和 TinyMCE 包的引用代码：

```
<script type="text/javascript" src="lib/tiny_mce/tiny_mce.js"></script>
<script type="text/javascript" src="TinMCE.js"></script>
```

然后创建一个表单面板，并加入编辑器，代码如下：

```
Ext.create(Ext.form.Panel,{
    width:500,height:500,
    renderTo:Ext.getBody(),
    tbar:[
            {text:"getValue",handler:function(){
                    var f=this.up("form").getForm();
                    console.log(f.findField("content").getValue());

            }},
            {text:"setValue",handler:function(){
                    var f=this.up("form").getForm();
                    f.findField("content").setValue("测试，测试，测试");
            }},
            {text:"保存",handler:function(){
                    var f=this.up("form").getForm();
                    f.submit();
            }}
    ],
    items:[
            {xtype:"xtinymce",name:"content",fieldLabel:"内容",
                    labelAlign:"top",anchor:"0 0",
                    tinyMceConfig:{language:"zh-cn",height:420}
            }
    ]
})
```

面板中有 3 个按钮，分别用于获取编辑器的值、设置编辑器的值及尝试提交表单。

TinyMCE 字段与其他字段的定义区别不大，只是多了 tinyMceConfig 配置项用来定义编辑器的显示。在代码中，将使用中文语言包，高度也必须设置，不然会造成 iframe 高度不足的问题。

在浏览器中打开页面，将看到如图 20-4 所示的效果。单击“setValue”按钮，将看到编辑器中会显示“测试，测试，测试”，单击“getValue”按钮，会在控制台看到以下输出：

```
<p>测试，测试，测试</p>
```

单击“保存”按钮，不要理会错误，展开提交链接，在 Post 面板中将看到以下提交：

```
content     <p>测试，测试，测试</p>
```

这说明 TinyMCE 字段运行正常。

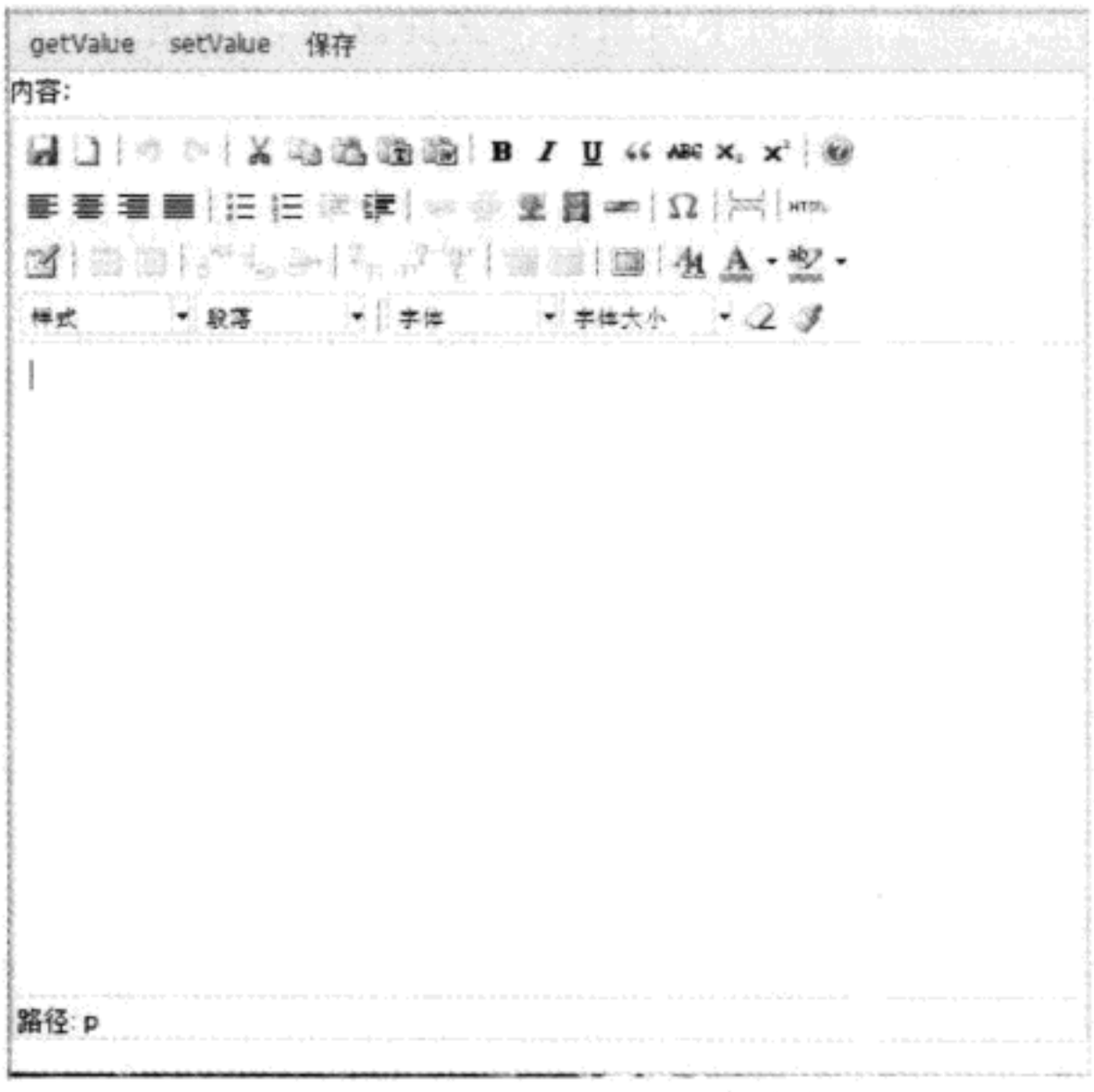


图 20-4　TinyMCE 字段的效果

20.6　本章小结

通过扩展或插件可以扩展 Ext JS 组件，实现所需的功能，因此必须好好掌握相关知识。在使用 MVC 架构时，会经常使用到扩展，多研究一下 Ext JS 4 的源代码有很大好处，因为它们基本是从 Ext.Base 对象扩展出来的。

写扩展或插件，其实并不难，关键是思路要正确，这主要依赖经验。如果有可以参考的类，则尽量按参考的类做出基本效果，在其上做调整，这是一个很好的写扩展的办法，例如本章的 TreeComboBox 扩展就是这样写出来的。

第21章　主题开发

Ext JS除了框架很棒，主题也很棒，这是Ext JS受欢迎的主要原因之一。在Ext JS 4中，使用SASS和Compass可以非常方便地创建自己的主题，这又是一次大的飞跃。本节将讲述如何创建自己的主题。

21.1　准备工作

21.1.1　安装Ruby

使用SASS和Compass需要用到Ruby，可以到http://rubyinstaller.org/下载Ruby的安装包。下载后的文件是“rubyinstaller-1.9.2-p290.exe”。双击运行该文件，将看到如图21-1所示的授权确认窗口。

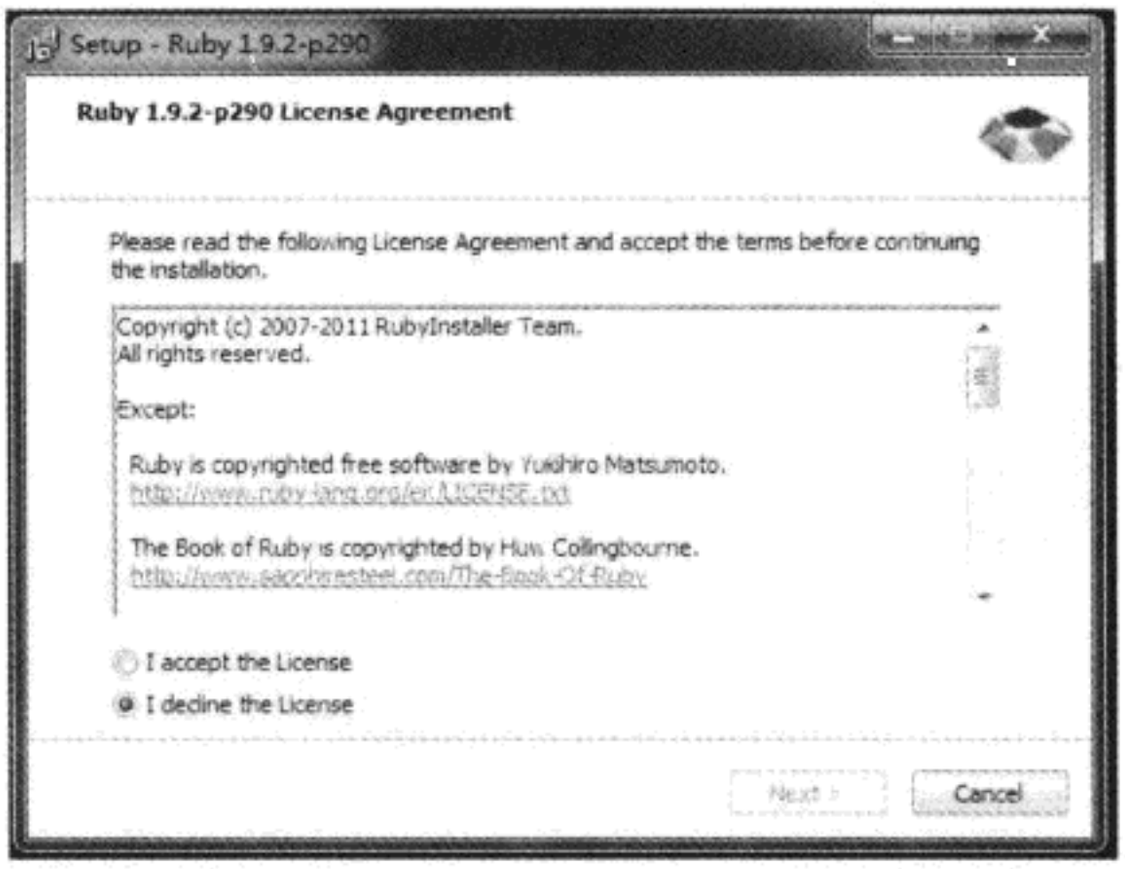


图21-1　安装Ruby的授权确认窗口

选择“I accept the License”，单击“Next”按钮，将看到如图21-2所示的安装目录选择窗口。

把安装目录下的3个选项都选上，单击“Install”按钮，进入安装过程，直到出现如图21-3所示的完成窗口。

单击“Finish”按钮完成安装。

至此，Ruby就安装完成了。

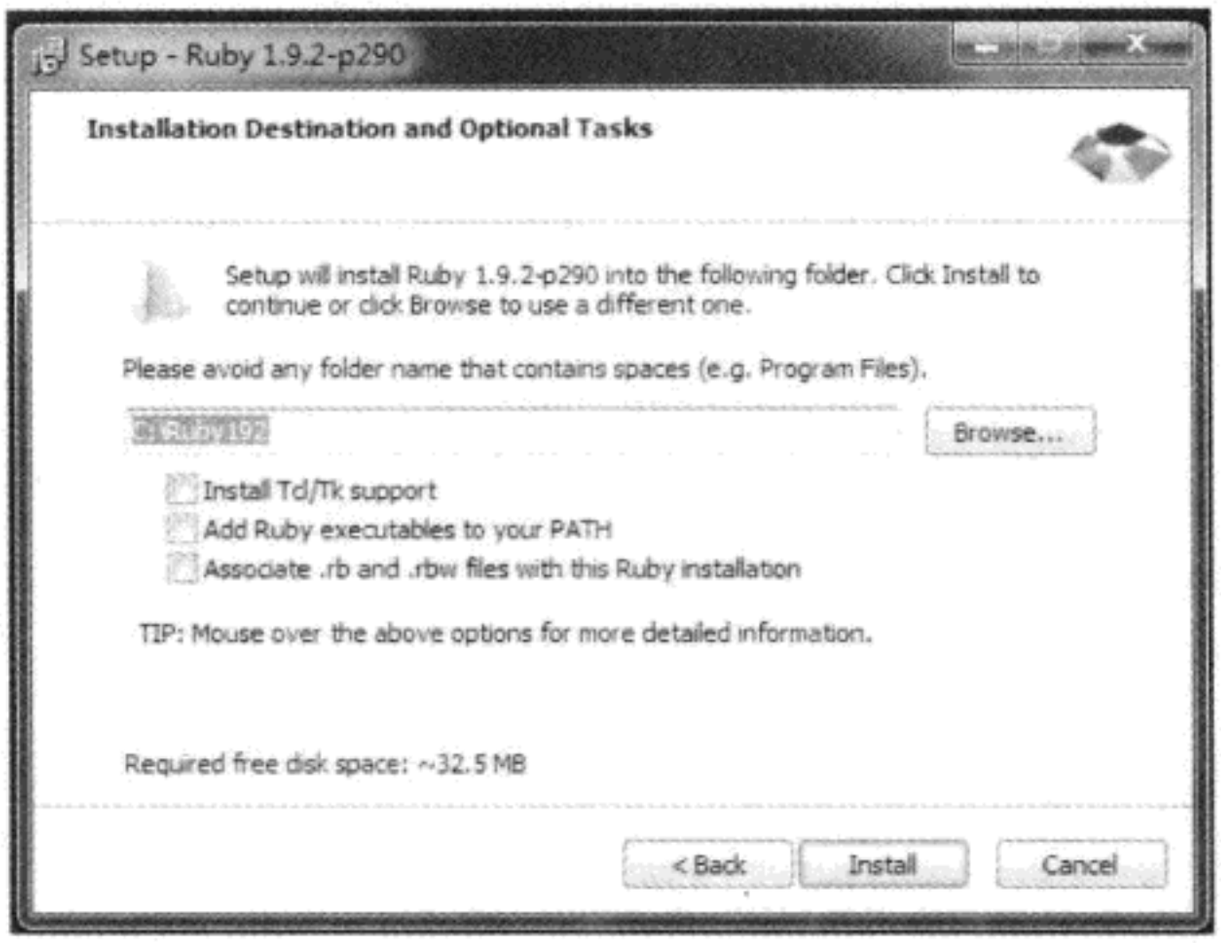


图 21-2　安装目录选择窗口

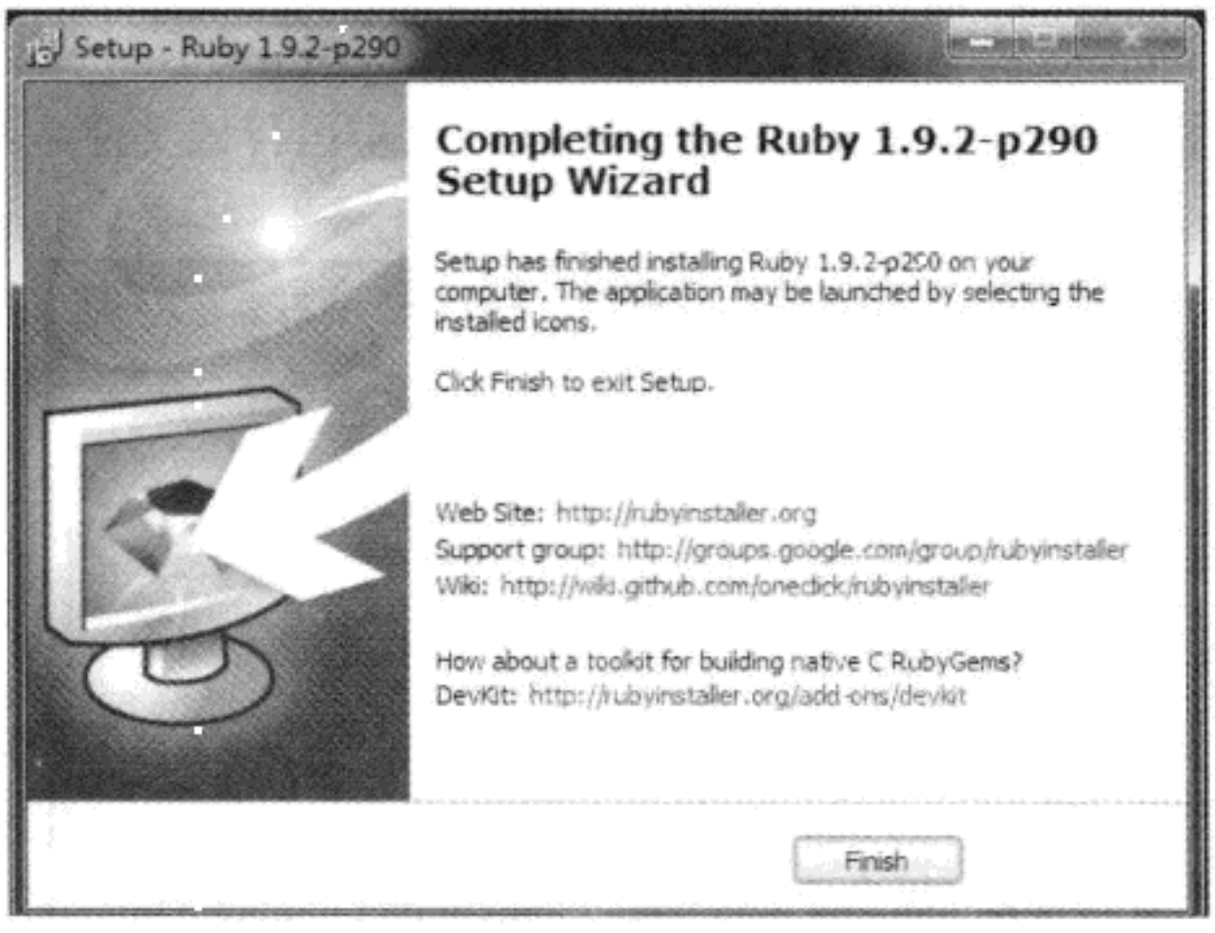


图 21-3　安装完成窗口

21.1.2　安装 Compass[⊖]

Compass 是一个基于 Ruby 的、开源的、用于 CSS 创作的框架。要使用 Compass，首先要在 Ruby 中安装框架。如图 21-4 所示，在开始菜单“Ruby 1.9.2-p290”程序组下，单击“Start Command Prompt with Ruby”，将显示如图 21-5 的命令提示窗口。

图 21-4　“Ruby 1.9.2-p290”程序组

⊖　相关信息请访问 http://compass-style.org/。

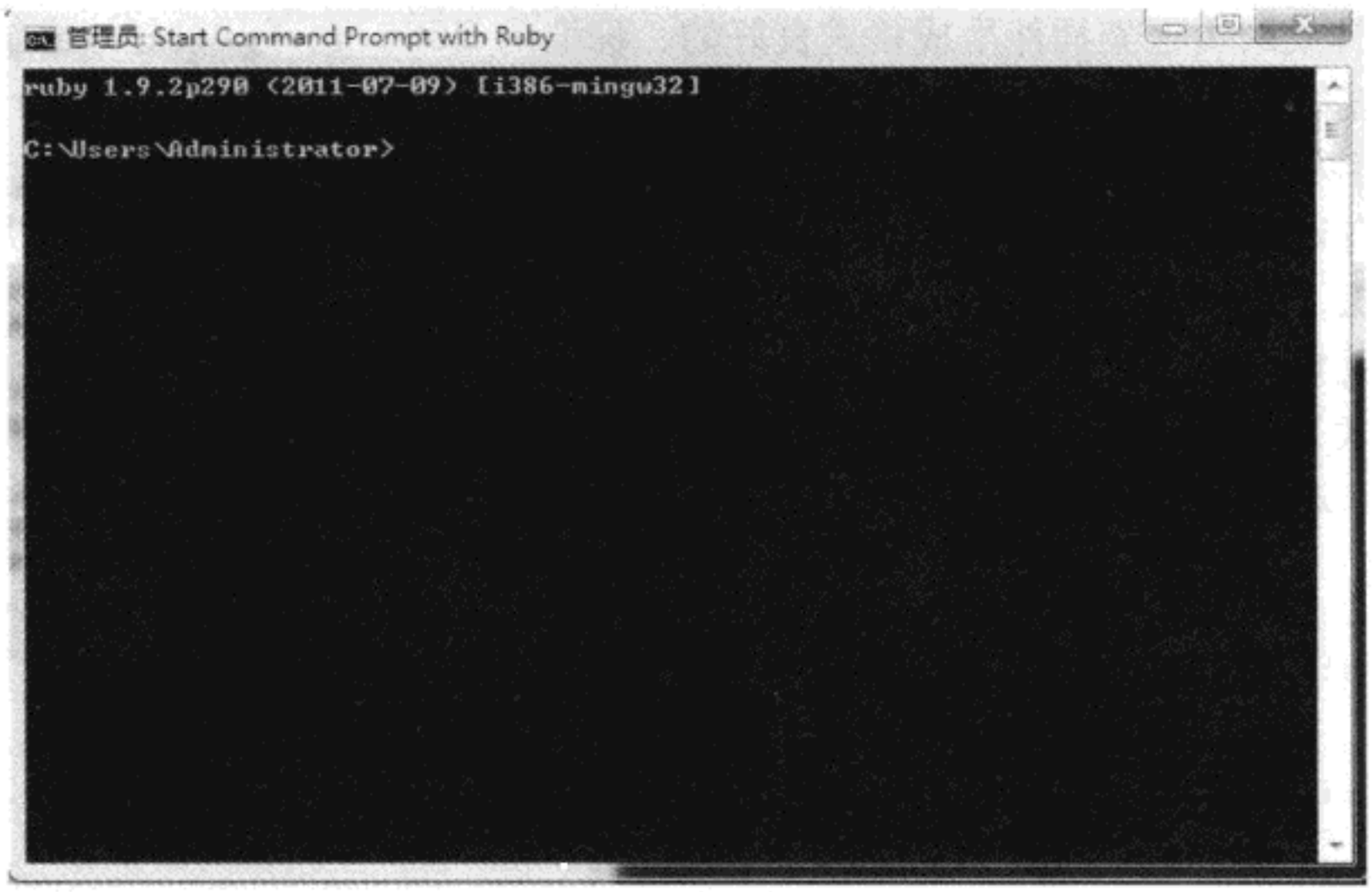


图 21-5 Ruby 的命令提示窗口

在命令行输入以下命令开始安装 Compass：

```
gem install compass
```

当看到如图 21-6 所示窗口中的信息时，表示 Compass 已经安装成功。

图 21-6 Compass 的安装过程

21.1.3 SASS[⊖]介绍

SASS 是 CSS3 的一个扩展，增加了内部规则、变量、混入和选择器继承等特性。它可通

⊖ 相关信息请访问：http://sass-lang.com/

过命令行工具或网页框架插件生成格式良好的、标准的 CSS。它可以很好地维护大的样式表结构，而且能像小样式表一样快速启动和运行，尤其是在 Compass 样式库的协助下。

SASS 有两种语法，第一种语法为 SCSS（Sassy CSS），它是 CSS3 语法的一个扩展，这意味着，任何有效的 CSS3 样式表都是有效的 SCSS 文件。另外，SCSS 具有许多 CSS Hack 和浏览器特有语法。这种语法是下面描述的 SASS 语法的加强版，其文件扩展名为“.scss”。

第二种是旧的语法，叫做缩进语法（indented syntax），很多时候就叫做 SASS，它提供许多简洁的方式来书写 CSS，例如使用缩进格式代替选择器嵌套的括号，使用换行符代替分号的分隔属性。对于以下 CSS 样式：

```
#main {
  color: blue;
  font-size: 0.3em;
}
```

SASS 的写法是：

```
#main
  color: blue
  font-size: 0.3em
```

旧语法的文件扩展名是“.sass”。

简单来说，SASS 通过一些可替换标记来定义样式，然后通过 Compass 库将标记的值替换标记，最终生成样式文件。例如，以下的 SASS 语句：

```
$blue: #3bbfce;
$margin: 16px;

.content-navigation {
  border-color: $blue;
  color:
    darken($blue, 9%);
}

.border {
  padding: $margin / 2;
  margin: $margin / 2;
  border-color: $blue;
}
```

其中“$blue”和“$margin”是变量，修改这两个变量的值，就可以生成不同的样式。将上述语句的 SASS 编译后，样式如下：

```
.content-navigation {
  border-color: #3bbfce;
  color: #2b9eab;
}

.border {
  padding: 8px;
  margin: 8px;
  border-color: #3bbfce;
}
```

21.2 为 Ext JS 4 创建新主题

21.2.1 概述

看了前两节的内容，估计有些人有点晕了。其实一点不复杂，因为 Ext JS 整个样式框架是已经搭建好的，通过先修改 Ext JS 样式框架中的变量值，然后编译生成新的样式文件，即可实现创建新主题的目的。

是的，就这么简单，我们要做的就是了解如何修改这些变量值，以及如何编译新的样式文件。

21.2.2 目录结构

因为主题的编译过程需要寻找对应的文件，所以对目录结构有着比较严格的要求，如果不想花时间去修改配置文件中的路径，那么就用配置好的目录结构去配置编译环境，这样会很简单。

首先要明确的一点是，编译过程是以网站根目录或应用根目录为基准的，也就是说，执行编译要在根目录中执行，如果应用或网站的根目录是 MyApp，那么配置和编译都是以该目录为基准的。

我们还是通过一步一步实践来熟悉这个过程。先创建一个名称为 MyApp 的目录，可以在任何地方创建，但是为了测试方法，就选择在 C 盘根目录创建。

接着，在其下建立一个名称为 lib 的文件夹，再在 lib 文件夹下创建一个名称为 Ext JS 的文件夹，从 Ext JS 的库中，将 ext-all-debug.js、ext-all.js、bootstrap.js 和目录 resources 复制到 extjs 文件夹下。其实可以不复制脚本文件，但 resources 是必需的，因为要在目录中寻找 SASS 的文件。

接着把复制好的 resources 目录中 themes\templates 目录下的 resources 目录复制到根目录（MyApp）下。这个目录中包含一个主题的模板，通过修改模板、编译模板，就会在此目录下的 css 目录中生成一个主题的样式文件。

在 MyApp\resources 目录下创建一个 images 目录，然后把目录 MyApp\lib\Ext JS\resources\themes\images 下的 default 目录复制到 images 目录下。

这样，目录就创建好了，现在要进入配置环节。

21.2.3 修改配置

在 MyApp\resources\sass 目录下有个 my-ext-theme.scss 文件，该文件是主题的模板文件，通过修改该文件，就可对主题进行配置。

使用 Notepad++、UltraEdit、VS2010 或 Eclipse 等代码编辑工具打开该文件，注意不要用记事本打开，不然打开的文字会堆在一起，不便于编辑。打开后，会看到以下代码：

```
    // Unless you want to include all components, you must set $include-default to
false
```

```
// IF you set this to true, you can also remove lines 10 to 38 of this file
$include-default: false;

// Insert your custom variables here.
// $base-color: #aa0000;

@import 'ext4/default/all';

// You may remove any of the following modules that you
// do not use in order to create a smaller css file.
@include Ext JS-boundlist;
@include Ext JS-button;
@include Ext JS-btn-group;
@include Ext JS-datepicker;
@include Ext JS-colorpicker;
@include Ext JS-menu;
@include Ext JS-grid;
@include Ext JS-form;
    @include Ext JS-form-field;
    @include Ext JS-form-fieldset;
    @include Ext JS-form-checkboxfield;
    @include Ext JS-form-checkboxgroup;
    @include Ext JS-form-triggerfield;
    @include Ext JS-form-htmleditor;
@include Ext JS-panel;
@include Ext JS-qtip;
@include Ext JS-slider;
@include Ext JS-progress;
@include Ext JS-toolbar;
@include Ext JS-window;
@include Ext JS-messagebox;
@include Ext JS-tabbar;
@include Ext JS-tab;
@include Ext JS-tree;
@include Ext JS-drawcomponent;
@include Ext JS-viewport;

// This line changes the location of your images when creating UIs to be relative
    instead of within the Ext JS directory.
// You MUST set this to true/string value if you are creating new UIs + supporting
    legacy browsers.
// This only applies to new UIs. It does not apply to default component images (i.e.
    when changing $base-color)
// The value can either be true, in which case the image path will be "../images/"
// or a string, of where the path is
$relative-image-path-for-uis: true; // defaults to "../images/" when true
```

第一个变量是“$include-default”，其作用为设置是否编译所有组件的样式，如果将此变量设置为 true，那么会编译所有组件的样式，不过，也可以删除粗体代码中的组件，删除的组件的样式将不会被编译。此变量的默认值为 true，表示不编译所有组件。要根据需要对此变量进行定义。

接着第二部分用来定义变量值，“$base-color”变量设置的是主题的颜色基调，例如灰色

主题的基调颜色是“#ccc”，这个可以在 MyApp\lib\Ext JS\resources\sass 目录下的 ext-all-gray.scss 文件中看到，此文件是一个很好的学习模板。在 ext-all-gray.scss 文件中，还可以看到许多其他变量的定义，这些变量的名称可以在 MyApp\lib\Ext JS\resources\themes\stylesheets\ext4\default\variables 目录下的文件中找到，该目录下的文件都是根据组件的名称组织的，例如 _window.scss 就是窗口的样式。将要修改的变量名复制到 ext-all-gray.scss 文件中，重定义就可以了。例如，“$base-color”在 _core.scss 文件中的值是“#C0D4ED”，在 ext-all-gray.scss 中，将其修改为“#ffff00”（黄色），让其作为主题的基调颜色（注意要把前面的注释符去掉）。

笔者还打算修改一下样式中默认字体为 11 像素的问题，不过发现根本改不了，原因是变量“$font-size”默认的字体是 12 像素，但是在其他文件中，许多字体的值都是“$font-size”乘以 0.9 后取整值，也就是 11 像素，所以还是得用老办法改。

修改完成后，就可以进行编译了。

21.2.4 编译

在编译前，先检查一下与 ext-all-gray.scss 文件同目录的 config.db 文件中，以下代码定义的路径是否能访问到 Ext JS 目录下的 resources 目录：

```
$ext_path = "../../lib/Ext JS"
```

这是文件的默认配置值，刚才建目录时也是按定义的目录来创建目录结构的，因此一般只是确认一下而已。如果定义的目录不同，就要修改定义了。

确认路径没错后，再检查一下 MyApp\resources\CSS 目录下是否已存在 my-ext-theme.CSS 文件，如果存在，则把文件删除，让 Compass 重新生成该文件。

打开 Ruby 的命令提示窗口，使用 cd 命令，先将当前目录切换到 MyApp 目录下，然后在命令行中输入以下命令进入编译过程：

```
compass compile resources/sass
```

如果窗口中最后一句输出如下：

```
create resources/sass/../css/my-ext-theme.css
```

就表示编译已经完成，已经生成了样式文件了。现在可以测试一下样式了。

21.2.5 测试主题

测试主题当然使用 Ext JS 4 包中的 Theme Viewer 示例，该示例几乎把所有组件都显示出来了，这样就可以测试到哪些组件还存在问题，哪些地方还需要修改了。

把 examples\themes 目录下的 index.html 文件和 themes.js 文件复制到 MyApp 目录下，然后把原来引用的样式的标记删除，加入以下标记：

```
<link rel="stylesheet" type="text/css" href="resources/css/my-ext-theme.css" />
```

好了，在浏览器中打开页面，将看到如图 21-7 所示的页面，所有组件的背景颜色都已经是黄色的了。

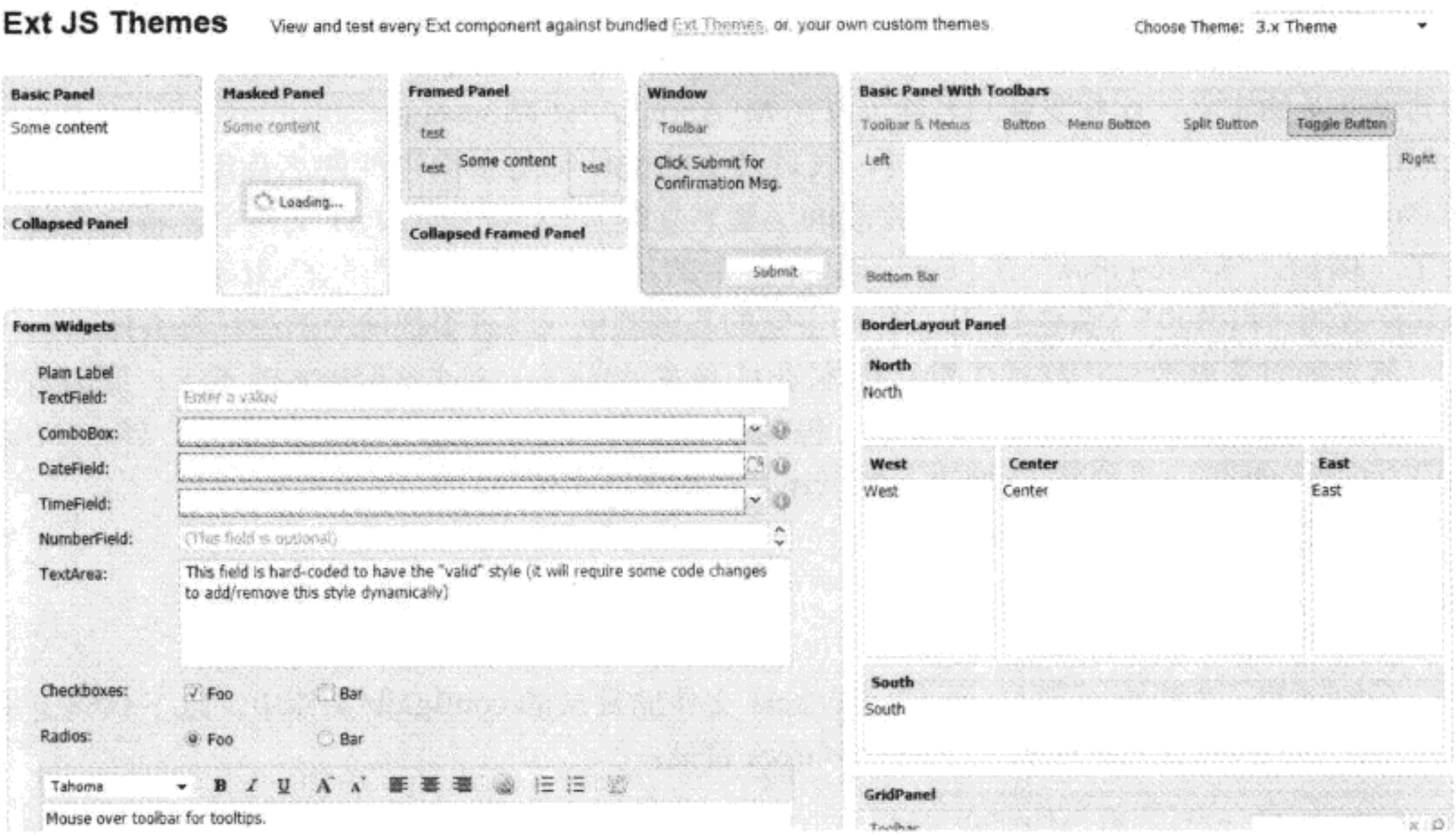


图 21-7　测试新主题的效果

从页面中可以看到，还有一些地方没有修改，例如 Grid 的列标题栏和 Accordion 布局的标题栏等。这时就要通过 Firebug 找出没有修改的样式，例如要找 Accordion 布局的标题栏的问题，在 HTML 面板中使用“点选元素”按钮，在面板中选择 TreePanel 所在位置，然后从外到内一层层地检查 DIV 标记的样式。在如图 21-8 所示的结果中，问题出在样式“x-accordion-hd”内，其背景颜色还是“#D9E7F8”。

图 21-8　查找未转换的样式的结果

根据样式名称，可以知道此样式应该属于什么样的布局的样式。在 variables 目录下，与布局有关的样式文件是 _layout.scss 文件，打开此文件后很容易就找到如下语句：

```
$accordion-header-background-color: #d9e7f8 !default;
```

这正是生成“x-accordion-hd”样式的语句。这时千万不要直接修改 _layout.scss 文件中的变量值，不然每修改一次样式就要找到该文件再修改一次。正确的做法是把变量名复制到 ext-all-gray.scss 文件中，再对其赋值。

页面中还存在图标的问题，这个问题要根据主题找美工人员完成。基本上，每一套主题，

都需要美工做一套对应的图标，工作量还是挺大的，所以在修改主题之前要考虑清楚，并且要做完整测试。

21.3 通过 ui 配置项设置组件样式

在 AbstractComponent 对象中，定义了 ui 配置项，其作用就是为组件设置与主题不同的样式，其配套的方法包括 addClsWithUI、addUIClsToElement、hasUICls、removeClsWithUI、removeUIClsFromElement 和 setUI 6 个，因此使用 UI 的一个好处就是在应用中可以随时根据需要改变组件的样式。

创建自定义 UI 的最大问题是，它与重新生成一个主题没有太大区别，只是把 UI 的样式也包含在样式文件里而已。而且创建自定义 UI 的过程相当复杂，还不如根据需要直接写成 CSS 来得方便，或者干脆就用 cls 配置项解决。

创建自定义 UI 的具体例子可参考 Ext JS 包中示例文件中的 examples/panel/bubble-panel.html 文件，以及阅读 API 用户指南中与主题有关的章节。

21.4 本章小结

在 Ext JS 4 中创建主题可以说相当简单，但是，别忘了需要美工的支持，因而在不确定是否有美工的情况下，千万不要考虑主题的事情，不然，做出来的主题可能会因为缺少某个图标而引起麻烦。同时在开发应用的过程中，完成项目的重要功能才是重点，主题应该放在最后考虑。当然，对于主题优先的项目另当别论。

第 22 章　MVC 应用架构

随着应用规模的扩大，如何管理和维护脚本逐渐成为受开发人员重视的一个问题。在这些脚本中，哪些可以重用，如何重用，也逐渐成为关注的焦点。近来，MVC 模式的兴起为解决这方面的问题提供了一个思路，因此 Ext JS 4 在客户端加入了 MVC 应用架构，为开发人员提供了一种维护和开发脚本的模式。

本章将讲述 Ext JS 4 的 MVC 应用架构如何运作及如何使用。

22.1　MVC 应用架构的构成及工作流程

22.1.1　构成

Ext JS 4 的 MVC 应用架构主要由 Application、Controller 和 EventBus 三个类组成。其中，Controller 类的主要作用是作为所有控制器类的基类，控制应用程序的流程。Application 类派生于 Controller 类，主要作用是初始化应用。EventBus 类的作用是将应用事件转发到对应的控制器上。

22.1.2　控制器的工作流程：Ext.app.Controller

Controller 的构造函数代码如下：

```
constructor: function(config) {
    this.mixins.observable.constructor.call(this, config);
    Ext.apply(this, config || {});
    this.createGetters('model', this.models);
    this.createGetters('store', this.stores);
    this.createGetters('view', this.views);
    if (this.refs) {
        this.ref(this.refs);
    }
},
```

在以上代码中，先初始化事件，然后将配置项复制到控制器中。接着为控制器内使用到的模型、Store 和视图调用 createGetters 方法创建 get 方法，此方法的代码如下：

```
createGetters: function(type, refs) {
    type = Ext.String.capitalize(type);
    Ext.Array.each(refs, function(ref) {
        var fn = 'get',
            parts = ref.split('.');
        Ext.Array.each(parts, function(part) {
```

```
            fn += Ext.String.capitalize(part);
        });
        fn += type;
        if (!this[fn]) {
            this[fn] = Ext.Function.pass(this['get' + type], [ref], this);
        }
        this[fn](ref);
    },
    this);
},
```

在以上代码中，先调用 capitalize 方法将 type 参数的第一个字母转换为大写，然后枚举 refs 数组内的数据。在枚举函数内，先根据小数点将名称拆分后，将 get 字符串与拆分后的字符串（第一个字母大写）结合成方法的名称，然后将方法名称指向由 Ext.Function 的 pass 方法生成的封装函数，在此函数内会调用控制器的 getModel、getStore 或 getView 方法，这 3 个方法又会调用 Application 对象的同名方法返回对象。

回到构造函数，接着调用 ref 方法，处理 refs 数组，其代码如下：

```
ref: function(refs) {
    var me = this;
    refs = Ext.Array.from(refs);
    Ext.Array.each(refs, function(info) {
        var ref = info.ref,
            fn = 'get' + Ext.String.capitalize(ref);
        if (!me[fn]) {
            me[fn] = Ext.Function.pass(me.getRef, [ref, info], me);
        }
    });
},
```

与 createGetters 方法一样，会为 refs 内每一个对象创建一个 get 方法，而该 get 方法会调用 getRef 方法，其代码如下：

```
getRef: function(ref, info, config) {
    this.refCache = this.refCache || {};
    info = info || {};
    config = config || {};
    Ext.apply(info, config);
    if (info.forceCreate) {
        return Ext.ComponentManager.create(info, 'component');
    }
    var me = this,
        selector = info.selector,
        cached = me.refCache[ref];
    if (!cached) {
        me.refCache[ref] = cached = Ext.ComponentQuery.query(info.selector)[0];
        if (!cached && info.autoCreate) {
            me.refCache[ref] = cached = Ext.ComponentManager.create(info,
'component');
        }
        if (cached) {
            cached.on('beforedestroy', function() {
```

```
                me.refCache[ref] = null;
            });
        }
    }

    return cached;
},
```

如果属性 forceCreate 为 true，会调用 ComponentManager 对象的 create 方法创建对象实例，并返回创建的对象实例。

如果在 cache 中还没有该对象，则使用 ComponentQuery 对象的 query 方法根据选择符去查找对象，如果没有找到，并且 autoCreate 属性为 true，则创建对象。

如果找到了，就为对象绑定 beforedestroy 事件，在对象销毁前先删除缓存的指向。

由代码可知，getRef 方法的作用就是创建和缓存对象的。

至此，控制器的工作就完成了。

22.1.3 Application 对象的工作流程

Application 对象的构造函数如下：

```
constructor: function(config) {
    config = config || {};
    Ext.apply(this, config);
    var requires = config.requires || [];
    Ext.Loader.setPath(this.name, this.appFolder);
    if (this.paths) {
        Ext.Object.each(this.paths, function(key, value) {
            Ext.Loader.setPath(key, value);
        });
    }
    this.callParent(arguments);
    this.eventbus = new Ext.app.EventBus;
    var controllers = Ext.Array.from(this.controllers),
        ln = controllers && controllers.length,
        i, controller;
    this.controllers = Ext.create('Ext.util.MixedCollection');
    if (this.autoCreateViewport) {
        requires.push(this.getModuleClassName('Viewport', 'view'));
    }
    for (i = 0; i < ln; i++) {
        requires.push(this.getModuleClassName(controllers[i], 'controller'));
    }
    Ext.require(requires);
    Ext.onReady(function() {
        for (i = 0; i < ln; i++) {
            controller = this.getController(controllers[i]);
            controller.init(this);
        }
        this.onBeforeLaunch.call(this);
    }, this);
},
```

在以上代码中，首先调用 Loader 对象的 setPath 方法设置动态加载的路径。设置的路径分两部分，第一部分是应用路径，由 appFolder 配置项决定，第二部分由 paths 属性决定。

接着调用父对象 Controller 的构造函数，初始化工作完成后，就创建了 EventBus 对象实例，用于处理事件。

接着处理控制器（controllers 配置项），在这里，会使用 MixedCollection 对象实例记录所有控制器。

如果 autoCreateViewport 属性为 true，则在 requires 数组内加入 Viewport 对象，以实现动态加载，也要把控制器对象加入到动态加载数组内。

接着调用 require 方法开始动态加载对象。

当页面准备好以后，就调用 getController 方法开始创建控制器实例，此方法的代码如下：

```
getController: function(name) {
    var controller = this.controllers.get(name);
    if (!controller) {
        controller = Ext.create(this.getModuleClassName(name, 'controller'), {
            application: this,
            id: name
        });
        this.controllers.add(controller);
    }
    return controller;
},
```

以上代码只是实现创建控制器实例并将实例加入到 controllers 中。

创建好控制器实例后，就调用其 init 方法，执行初始化操作。

最后调用 onBeforeLaunch 方法，其代码如下：

```
onBeforeLaunch: function() {
    if (this.enableQuickTips) {
        Ext.tip.QuickTipManager.init();
    }
    if (this.autoCreateViewport) {
        this.getView('Viewport').create();
    }
    this.launch.call(this.scope || this);
    this.launched = true;
    this.fireEvent('launch', this);
    this.controllers.each(function(controller) {
        controller.onLaunch(this);
    }, this);
},
```

如果开启 QuickTip，则进行初始化操作。如果自动创建 Viewport，则创建 Viewport。

接着调用 launch 方法，该方法需要重写以开始运行应用。

触发 launch 事件后，调用每个控制器的 onLaunch 方法，该方法也需要重写以开始运行控制器。

22.2 一步一步实现 MVC 框架

22.2.1 概述

看起来，要使用 MVC 框架相当复杂，笔者认为难的不是如何写控制器、视图、Store 等，而是如何控制显示的切换。在 Ext JS 包中，有 4 个使用 MVC 框架的例子，最好的例子是 API 文档，但是这几个例子都是单一显示界面的，没有界面切换的问题。本节将一步一步地实现一个这样的应用示例。

示例将实现对 Northwind 库的管理，当然，囿于篇幅，不可能实现全部表的管理，只实现部分功能。

示例界面从上到下分为应用标题、菜单栏和内容页等三部分。单击“菜单栏”的按钮可切换内容页的内容，并且菜单栏的菜单可根据登录用户动态显示菜单。

示例将用到 Ext.Direct，具体配置请参考第 17 章的内容。

22.2.2 创建目录

MVC 应用架构对目录有严格的要求，因此必须按目录结构创建目录。首先是创建应用目录，名称为 Northwind。接着创建 MVC 架构需要的目录，如果不修改默认配置，Application 对象的 appFolder 配置项指定的名称默认为 app，那么就创建一个 app 目录。在 app 目录下创建 controller、model、store 和 view 目录，分别用于放置控制器、模型、Store 和视图的脚本文件，这些目录名称是架构规定目录的名称，所以不要搞错了。

以下创建的目录不是规定的，因此可以根据自己的习惯去设置。

接着创建一个 Ext JS 目录，然后把 bootstrap.js、Ext JS-all.js、ext-all-debug.js、local 目录和 resources 目录复制到该目录下，同时创建一个 ux 目录用于放置自定义扩展和插件。

好了，目录就创建完了。

22.2.3 创建首页

使用模板页创建一个名称为 index.html 的页面文件，加入以下代码：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/
    strict.dtd">
<html>
<head>
    <title>MVC 应用机构实例: Northwind 管理系统 </title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <link rel="stylesheet" type="text/css" href="Ext JS/resources/css/ext-all.css"/>
    <script type="text/javascript" src="Ext JS/bootstrap.js"></script>
    <script type="text/javascript" src="Ext JS/locale/Ext-lang-zh_CN.js"></script>
    <script type="text/javascript" src="app.js"></script>
</head>
<body>
</body>
</html>
```

页面中没有太多代码，主要因为它是启动 app.js 后自动生成的，所以页面的主要工作就是把 Ext JS 的样式文件、脚本文件和本地化文件载入页面。当然不能缺少的是启动脚本 app.js 和 Ext.Direct 的 API 文件。

22.2.4　创建启动脚本：app.js

启动脚本首先要考虑的是，作为一个管理系统，页面会先显示一个登录对话框，等待用户登录。那么，如何显示这个登录对话框呢？有两种方式，第一种方式是把登录对话框做成控制器和视图，然后在应用初始化的时候加载并显示；第二种方式与启动 API 文档中的 app.js 文件一样，先不初始化应用，而是等登录对话框验证后才初始化应用。

这两种方式没有好与不好之分，如何用依据使用者的习惯，结果都是一样的。

在示例应用中，登录对话框将作为独立的类加载，在应用初始化之前先登录，登录成功后再初始化页面。

在 Northwind 目录下创建一个名称为 app.js 的脚本文件，加入以下代码：

```
Ext.Loader.setConfig({
    enabled: true,
    paths: {
        'Northwind': 'app'
    }
});

Ext.direct.Manager.addProvider(Ext.app.REMOTING_API);
Ext.ns('Northwind.app');
Ext.require('Northwind.LoginWin');


Ext.onReady(function() {
   Ext.state.Manager.setProvider(new Ext.state.CookieProvider({
       expires: new Date(new Date().getTime()+(1000*60*60))
   }));
   if(Ext.util.Cookies.get("hasLogin")=="true"){
           Ext.create("Northwind.Application");
   }else{
                   Northwind.LoginWin.show();
    }
});
```

使用 MVC 架构，脚本基本是动态加载的，这也是在 Ext JS 4 中加入动态加载的原因，其主要目的并不是实现 Ext JS 库的动态加载，所以不必考虑如何动态加载 Ext JS 库。

要实现动态加载，首先要根据 MVC 的规则，使用 Loader 对象的 setConfig 方法设置动态加载目录，默认的动态加载目录就是 app 目录。

接着设置 Ext.Direct，“Ext.app.REMOTING_API”可以根据项目使用 Northwind 命名空间，不过要先检查路由库是否支持。为了避免麻烦，还是使用默认设置比较好。

调用 ns 创建“Northwind.app”命名空间是为了在应用初始化后，app 会指向应用对象。

接着调用 require 方法加载登录对话框。

在 onReady 函数内，因为要使用状态管理器管理用户信息，所以要在这里进行初始化，并设置其过期时间为 1 小时。然后检查 Cookies 中 hasLogin 值是否为 true，如果是，说明已登录，重新初始化应用显示就行了；如果 hasLogin 值为否，表示还没有登录，则显示登录对话框。

22.2.5 定义登录对话框

登录对话框将直接从 Window 对象扩展，将其定义为单件模式是不错的选择，这样就省去了实例化的过程，这里只需要创建一个实例就足够了，可写下以下基本定义代码：

```
Ext.define("Northwind.LoginWin",{
    extend:"Ext.window.Window",
    singleton:true,
})
```

根据 17.4.5 节示例，可把对话框的基本配置项复制过来，代码如下：

```
hideMode: 'offsets',
closeAction: 'hide',
closable:false,
resizable: false,
title:'Northwind 管理系统登录窗口 ',
width: 300,
height: 250,
modal:true,
currentTabIndex:0,
```

接着定义 initComponent 方法，在方法内要定义表单、表单的字符及按钮，具体代码如下：

```
initComponent: function() {
    var me=this;

    me.fields=[];

    me.fields.push(
            Ext.widget("textfield",{
                    fieldLabel:" 用户名 ",name:"username",
                    allowBlank:false,tabIndex:1,
                    listeners:{
                            scope:me,
                            focus:me.setTabIndex
                    }
            })
    );

    me.fields.push(
            Ext.widget("textfield",{
                    fieldLabel:" 密码 ",name:"password",inputType:"password",
                    allowBlank:false,tabIndex:2,
                    listeners:{
                            scope:me,
                            focus:me.setTabIndex
                    }
```

```
        })
);

me.fields.push(
        Ext.widget("textfield",{
                fieldLabel:"验证码",name:"vcode",
                minLength:6,minLength:6,
                allowBlank:false,tabIndex:3,
                listeners:{
                        scope:me,
                        focus:me.setTabIndex
                }
        })
);

me.image=Ext.create(Ext.Img);

me.form=Ext.create(Ext.form.Panel,{
        border:false,bodyPadding:5,
        api:{submit:Ext.app.Login.Check},
        bodyStyle:"background:#DFE9F6",
        fieldDefaults: {
        labelWidth:80,labelSeparator:": ",anchor:"0"
        },
        items:[
                me.fields[0],
                me.fields[1],
                me.fields[2],
                {xtype:"container",height:80,anchor:"-5",layout:"fit",
                        items:[me.image]
                }
        ],
        dockedItems: [{
                xtype: 'toolbar',dock:'bottom',ui:'footer',layout:{pack:"center"},
            items: [
                {text:"登录",disabled:true,formBind:true,handler:me.onLogin,scope:me},
                {text:"重置",handler:me.onReset,scope:me},
                {text:"刷新验证码",handler:me.onRefrehImage,scope:me}
            ]
        }]
});

me.items=[me.form]

me.on("show",function(){
        me.onReset();
    },me);

    me.callParent(arguments);
},
```

将输入字段记录在 fields 数组内，这样使用回车键切换就简单多了，只要根据数组索引切换就可以了。字段获得焦点后，会调用 setTabIndex 方法改变 currentTabIndex 属性的值，这比直接在 onReady 方法中定义方便多了。

将 image 属性指向验证码图片，这样切换验证码也方便多了。

接着创建表单面板，将 form 属性指向它，这样可通过 form 属性直接访问到表单。创建表单后，把它加入到 items 指向的数组中，这样就可把表单面板加入到窗口了。

接着绑定 show 事件，在显示窗口的时候，重置一下表单。

最后必须有一个调用父类的构造函数，不然就无法初始化登录对话框了。

完成调用的方法如下：

```
initEvents:function(){
    var me=this;
    me.KeyNav=Ext.create("Ext.util.KeyNav",me.form.getEl(),{
            enter:me.onFocus,
            scope:me
    });
},

onLogin:function(){
    var me=this,
            f=me.form.getForm();
    if(f.isValid()){
            f.submit({
                    success:function(form,action){
                            me.hide();
                            Ext.state.Manager.set("userInfo",action.result.userInfo);
                            Ext.create('Northwind.Application');
                    },
              failure: function(form, action){
                    if (action.failureType === "connect") {
                        Ext.Msg.myAlert('错误',
                            '状态:'+action.response.status+': '+
                            action.response.statusText,Ext.Msg.ERROR);
                        return;
                    }
                      if(action.result){
                            if(action.result.msg)
                            Ext.Msg.myAlert('错误', action.result.msg,Ext.Msg.ERROR);
                    }
              },
              scope:me
            });
    }
},

onReset:function(){
    var me=this;
    me.form.getForm().reset();
    me.onRefrehImage();
},

onRefrehImage:function(){
    this.image.setSrc("vcode.ashx?_dc="+(new Date()).getTime());
},
```

```
setTabIndex:function(el){
    this.currentTabIndex=el.tabIndex-1;
},

onFocus:function(){
    var me=this,
            index=me.currentTabIndex;
            index++;
            if(index>2){
                    index=0;
            }
            me.fields[index].focus();
            me.currentTabIndex=index;
}
```

从 8.2.5 节可以知道，initEvents 方法会在组件渲染后执行，而实例化 KeyNav 对象需要使用 HTML 元素，所以在该方法内进行实例化就不会发生找不到元素的错误了。

单击“登录”按钮后会执行 onLogin 方法，在登录成功后，先隐藏窗口，然后将服务器端返回的用户信息保存到状态管理器中。这样，当刷新页面并再次初始化应用的时候，就可以从状态管理器中找到用户信息，而无需再次从服务器端获取了。

将用户信息保存到状态管理器后，就可以开始初始化应用了。

对应的服务器端的 Check 方法的代码与示例 17-5 基本差不多，主要需要修改验证部分，代码如下：

C#

```
HttpCookie login = new HttpCookie("hasLogin");
DateTime dt = DateTime.Now;
TimeSpan ts = new TimeSpan(0,0,10,0);
login.Expires = dt.Add(ts);
login.Value = "true";
if (username == "admin" && password == "123456")
{
    jo.Add("msg", new JValue("登录成功"));
    jo.Add("userInfo", new JObject(
        new JProperty("role","管理员")
    ));
    HttpContext.Current.Response.Cookies.Add(login);
}
else if(username == "user1" && password=="123456"){
    jo.Add("msg", new JValue("登录成功"));
    jo.Add("userInfo", new JObject(
        new JProperty("role", "用户")
    ));
    HttpContext.Current.Response.Cookies.Add(login);
}
else
{
    jo.Property("success").Value = new JValue(false);
    jo.Add("errors", new JObject(
        new JProperty("username", "用户名或密码错误。"),
```

```
            new JProperty("password", "用户名或密码错误。")
        ));
}
```

Java

```
Cookie cookie = new Cookie("hasLogin", "true");
cookie.setMaxAge(1200);
if(username.equals("admin") && password.equals("123456")){
   userinfo.put("role", "管理员");
   result.put("success", true);
   result.put("msg","登录成功");
   result.put("userInfo",userinfo);
   context.getResponse().addCookie(cookie);
}else if(username.equals("user1") && password.equals("123456")){
   userinfo.put("role", "用户");
   result.put("success", true);
   result.put("msg","登录成功");
   result.put("userInfo",userinfo);
   context.getResponse().addCookie(cookie);
}else{
   result.put("success", false);
   errors.put("username", "用户名或密码错误。");
   errors.put("password", "用户名或密码错误。");
   result.put("errors", errors);
}
```

在代码中加入了 Cookie 来确认用户是否登录，不过，在 Java 中，DirectJNgine 似乎不支持输出 Cookie，所以代码在这里不起作用。

如果用户是 admin，则在返回的用户信息中，其 role 值为管理员，user1 为用户，这样，客户端就可根据该值配置菜单和按钮，这些信息可以从状态管理器中获取。

该示例只是演示如何实现功能，所以只加入了这些简单的信息，如果是正式应用，可以在这里将数据库中与用户角色有关的菜单等信息一起返回客户端进行处理。

22.2.6 创建应用脚本：Application.js

在应用脚本中要做的是初始化页面和显示界面，并决定菜单如何工作。当配置项 autoCreateViewport 为 true 时，会自动调用 Viewport.js 初始化页面。现在要考虑的是，如何创建界面中的菜单栏，并让它能够控制内容页的切换。

在 9.8.2 节，已经知道只要加载页面定义，并添加到内容页，然后通过 setActiveItem 方法就可实现页面切换。在使用 MCV 架构的时候，要加载的就是视图，加载视图后将其加入内容页就行了。现在的问题简化为，如何将菜单按钮的单击操作与加载内容页关联起来，并将加载后的视图添加到内容页。

要控制菜单按钮的单击操作，就要在 MVC 架构中创建一个控制器来控制按钮的行为。为了便于维护菜单，将菜单栏独立为一个单独的视图是一个好主意。

清楚结构后，就可创建应用脚本了，在 app 目录下创建 Application.js 文件，并加入以下代码：

```
Ext.define('Northwind.Application', {
    extend: 'Ext.app.Application',
    name: 'Northwind',

    controllers: ["MainMenu"],

    autoCreateViewport: true,

    launch: function() {
        Northwind.app = this;
    }

});
```

这里的代码不多，重点是 name 属性必须定义，因为它将会是视图、控制器、模型和 Store 类的命名空间，所以一定要注意。

在这里必须加载菜单的控制器（MainMenu），不然菜单就不起作用了。

当配置项 autoCreateViewport 为 true 时，将自动加载 Viewport 视图，并渲染到页面。

方法 launch 内的代码将 app 属性指向应用对象实例本身，这样就可通过 app 属性访问到应用实例了。

22.2.7　创建 Viewport 视图

Viewport 视图比较简单，从 Viewport 对象继承，并且将页面整体结构渲染出来。在 view 目录下创建 Viewport.js 文件，加入以下代码：

```
Ext.define('Northwind.view.Viewport', {
    extend: 'Ext.container.Viewport',

    id: 'viewport',
    layout:{type:'vbox',align:"stretch"},
    defaults:{xtype:'container',style:"background-color:#D3E1F1;"},

    initComponent: function() {
          var me=this;

        me.items = [
            {
                height: 40,
                id: 'logo-container',
                html:"Northwind 管理系统 ",
                padding:"0 0 0 20",
            style:"font-size:20px;font-weight:bold;line-height:40px;back-
                ground- color:#D3E1F1;",
        },
        {xtype:"mainmenu"},
        {xtype:"panel",
                        id:"contentPage",
            layout:'card',flex:1,
            padding:"5",
            items: [{
```

```
                    title:"欢迎使用Northwind管理系统",
                     html:"欢迎使用Northwind管理系统"
                }]
            }
        ];

        this.callParent(arguments);
    }

});
```

这里不需要将其设置为单件模式，因为在Application对象中会将其实例化。但一定要注意类名，因为是根据类名加载类的，例如，对于这里定义的类名“Northwind.view.Viewport”，Northwind的目录指向的是app，小数点之间的view是子目录，而最后一个类名加上扩展名“.js”就是文件名，因此最后加载文件的路径为“app\view\Viewport.js”。如果类名为“Northwind.view.product.View”，则表示加载文件的路径为“app\view\product\View.js”。

为每个视图定义一个id是比较好的方式，控制器是通过ComponentQuery对象寻找组件的，使用id查找会比使用其他选择符迅速很多。

因为示例是从上到下的结构，所以使用垂直布局就可以了。在不需要标题栏、停靠区域等附加部件的时候，建议使用容器，而不是面板，这样对提高应用性能有好处。

在items属性中，第一个配置对象定义的是应用标题；第二个配置对象定义的是菜单视图，通过其别名就可以让其实现动态加载并创建了实例；第三个配置对象定义是内容页，这里一定要设置id，因为在控制中，要根据该id找到对象，然后通过add方法添加视图。默认显示一个空面板，当然也可以根据需要显示一些必要的信息。

最后别忘记调用callParent方法，这是必须的。

22.2.8 菜单视图及控制器

菜单视图可直接从Toolbar派生，然后在initComponent方法内根据状态管理器中的用户信息动态设置菜单的按钮就行了。在view目录下创建一个名称为MainMenu.js的文件，加入以下代码：

```
Ext.define('Northwind.view.MainMenu', {
    extend: 'Ext.toolbar.Toolbar',
   alias : 'widget.mainmenu',

    id: 'mainmenu',

initComponent: function() {
       var me=this,
               userInfo=Ext.state.Manager.get("userInfo"),
               buttons=[];

   if(userInfo.role=="管理员"){
       buttons.push({text:"用户",action:"user"});
   }else{
       buttons.push({text:"订单管理",action:"order"});
```

```
            buttons.push({text:"产品管理",action:"product"});
        }
            buttons.push("->");
            buttons.push({text:"退出",action:"quit"});


        me.items = buttons;

        this.callParent(arguments);
    }
});
```

在 Viewport 中要使用 xtype 读取类，需要定义配置项 alias。配置项 id 是必需的，这样便于获取对象。

示例只是简单演示了如何实现动态菜单，因而管理员只显示“用户”和“退出”两个按钮。而用户则显示“订单管理”、“产品管理”和“退出”3 个按钮。本示例的重点在如何切换订单管理与产品管理。

大家可能注意到，按钮中只有 action 配置项，没有 handler 配置项定义单击操作，这是因为操作必须在控制器中进行，而 action 属性是作为选择符区分按钮用的，如果可能，建议使用 id，这样更快捷。

现在创建控制器，在 controller 目录下，创建一个名称为 MainMenu.js 的脚本文件，然后加入以下代码：

```
Ext.define('Northwind.controller.MainMenu', {
    extend: 'Ext.app.Controller',

    requires: [
        'Northwind.view.MainMenu'
    ],

    refs: [
          {ref:'contentPage', selector: '#contentPage'}
    ],

    init: function(app) {
        this.control({
            '#mainmenu button[action=order]': {
                click: this.switchPage
            },
            '#mainmenu button[action=product]': {
                click: this.switchPage
            },
            '#mainmenu button[action=territories]': {
                click: this.switchPage
            },
            '#mainmenu button[action=quit]': {
                click: this.Quit
            }
        });
    },
```

```
switchPage:function(btn){
        var me=this,
                key=btn.action,
                cmp=Ext.getCmp(key+"View");
        if(cmp){
                var layout=me.getContentPage().getLayout();
                if(layout.getActiveItem().id!=cmp.id){
                        layout.setActiveItem(cmp);
                }
                }else{
                me.getController(Ext.String.capitalize(key)).init();
            }
        },

        Quit:function(){
                Ext.app.Login.Quit(function(){
                        window.location="index.html?_dc"+(new Date()).getTime();
                });
        }

});
```

在代码中，属性 requires 指定了该控制器要加载哪些视图，这里要加载菜单视图。

属性 refs 的作用是根据配置对象中的 ref 配置项指定的名称创建一个 get 方法，指向根据 selector 定义的选择符查询到的对象。在代码中会创建一个 getContentPage 方法，指向内容页对应的容器，这样就可以使用 getContentPage 方法返回内容页，并使用其 add 方法添加视图了。

在 init 方法内使用 control 方法，可根据对象中属性定义的选择符查询到对象，然后为其绑定对象中定义的事件，例如，根据第一个属性会查询到主菜单（#mainmenu）中 action 值为 order 的按钮，也就是“订单管理”按钮，然后其单击事件会执行 switchPage 方法。

在 switchPage 方法中，与 9.8.5 节一样，通过 action 值这个关键字，先查询视图是否存在，如果已经存在，并且 getActiveItem 返回的当前视图与其不同，则切换视图；否则，调用 getController 方法加载视图的控制器，并且调用其 init 方法处理视图的加载。

在 Quit 方法内，调用服务器端 Quit 方法注销用户验证，退出应用，刷新后回到登录状态。

22.2.9 实现订单管理

订单管理将实现 11.3.4 节示例的界面，该界面需要定义两个模型、3 个 Store、1 个控制器和 1 个视图。

1. 定义模型

先定义 Order 模型，在 model 目录内创建一个名称为 Order.js 的脚本文件，代码如下：

```
Ext.define('Northwind.model.Order', {
    extend: 'Ext.data.Model',
    fields: [
            {name:'OrderID',type:"int"},
            'CustomerID','CustomerName',
```

```
            {name:'OrderDate',type:"date",format:"Y-m-d"}
        ],
        idProperty:"OrderID",
        hasMany:{model:"Northwind.model.OrderDetail",name:"OrderDetails",foreignKey:"
            OrderID"}
});
```

这里代码与 11.3.4 节示例的区别是模型名称，这里必须根据路径设置，不然加载不了模型。

接着，在 model 目录下创建名称为 OrderDetail.js 的脚本文件，用于定义 OrderDetail 模型，代码如下：

```
Ext.define('Northwind.model.OrderDetail',{
    extend: 'Ext.data.Model',
    fields: [
            {name:'OrderID',type:"int"},
            {name:'ProductID',type:"int"},
            {name:'UnitPrice',type:"float"},
            {name:'Quantity',type:"int"},
            {name:'Discount',type:"float"},
            'ProductName'
    ]
});
```

重点还是类名。

2. 定义 Store

虽然客户的列表树也需要模型，但不是必须的，直接在 Store 中定义就好。在 Store 中创建一个名称为 CustomerTreeStore.js 的脚本文件，与 11.3.4 节示例直接创建 Store 实例不同，这里要做的是定义一个 Store 的扩展，具体代码如下：

```
Ext.define("Northwind.store.CustomerTreeStore",{
    extend: 'Ext.data.TreeStore',
    proxy: {
            type:"direct",
            directFn:Ext.app.Customer.TreeList,
        reader:{
            type: 'json',
            root:"data"
        }
    }
})
```

首先要注意类名及其加载目录。这里除了使用 define 方法和 extend 配置项外，其他与 11.3.4 节示例没什么区别。

接着在 store 目录下创建名称为 Order.js 文件，用于定义订单的 Store，代码如下：

```
Ext.define("Northwind.store.Order",{
     extend: 'Ext.data.Store',
   model:'Northwind.model.Order',
   pageSize:20,
   batchActions:false,
   remoteFilter:true,
```

```
    remoteSort:true,
     proxy: {
            type:"direct",
            directFn:Ext.app.Order.List,
         reader:{
            type: 'json',
            root:"data"
         }
     }
})
```

接着定义订单明细的 Store，在 store 目录下创建名称为 OrderDetail.js 的脚本文件，加入以下代码：

```
Ext.define("Northwind.store.OrderDetail",{
     extend: 'Ext.data.Store',
    model:'Northwind.model.OrderDetail',
     proxy: {
            type:"ajax",
         reader:{
            type: 'json',
            root:"data"
         }
     }
})
```

3. 定义视图

在开始创建前，要考虑的问题是如何划分视图。在界面中将包含一个树面板和两个 Grid 面板，可以考虑的方案有以下三个：

- 全部面板作为一个视图。
- 树面板作为一个视图，两个 Grid 面板作为一个实体。
- 每个面板作为一个视图。

在“Ext JS 应用架构设计”[⊖]一文中，设计的重点是寻找一个平衡点，避免视图过多，也不要让视图变得太通用，这确实是比较恼人的工作。笔者的看法是，如果视图重用的可能性很大，就拆分；如果不是，则尽量避免拆分，这样更易于进行维护。

由于目前没重用需求，因此只需要为订单视图创建一个视图。在 view 目录下创建名称为 order 的子目录，然后在其内创建名称为 View.js 的脚本文件。视图将直接从面板扩展，使用 Border 布局划分 3 个区域，放置 3 个面板，具体代码如下：

```
Ext.define('Northwind.view.order.View', {
 extend: 'Ext.panel.Panel',
 alias : 'widget.orderview',

layout:"border",

title:"订单管理",
id:"orderView",
```

⊖ 相关信息请阅读 http://blog.csdn.net/tianxiaode/article/details/6562113。

```
initComponent: function() {
        var me=this;

        me.tree=Ext.widget("treepanel",{
                title:"客户",region:"west",collapsible: true,
                rootVisible:false,store:"CustomerTreeStore",
                width:200,minWidth:100,split:true,
                viewConfig:{
                        listeners:{
                                scope:me,
                                refresh: me.onTreeRefresh
                        }
                },
                listeners:{
                        scope:me,
                        selectionchange:me.onTreeSelect
                }
        });

        me.orderGrid=Ext.widget("grid",{
                title:"订单",region:"center",minHeight:200,
                tbar:{xtype:"pagingtoolbar",store:"Order",displayInfo:true},
                selMode:{mode:"SINGLE"},store:"Order",
                collapsible: true,
                columns:[
                        {xtype:"rownumberer",sortable:false,width:60},
                        {text:'订单号',dataIndex:'OrderID'},
                        {text:'客户编号',dataIndex:'CustomerID'},
                        {text:'客户名称',dataIndex:'CustomerName',sortable:false,flex:1},
                        {xtype:"datecolumn",text:'订购日期',dataIndex:'OrderDate'
                            ,format:"Y-m-d",width:100}
                ],
                viewConfig:{
                        listeners:{
                                scope:me,
                                refresh:me.onOrderRefresh
                        }
                },
                listeners:{
                        scope:me,
                        selectionchange:me.onOrderSelect
                }
        });

        me.detailsGrid=Ext.widget("grid",{
                title:"订单明细",region:"south",
                split:true,height:300,minHeight:200,
                collapsible: true,store:"OrderDetail",
                columns:[
                        {xtype:"rownumberer",sortable:false,width:60},
                        {text:'产品名称',dataIndex:'ProductName',sortable:false,fl
                            ex:1},
                        {xtype:"numbercolumn",text:'单价',dataIndex:'UnitPrice',
                            align:"right",format:"0,0.00"},
```

```
                        {xtype:"numbercolumn",text:'数 量',dataIndex:'Quantity',
                            format:"0,0"},
                        {xtype:"numbercolumn",text:'折 扣',dataIndex:'Discount',
                            format:"0,0.00"}
                    ]
            });

        this.items=[me.tree,me.orderGrid,me.detailsGrid];
            this.callParent(arguments);

        },

        onTreeRefresh:function(){
            //this.tree.view.select(0);
        },

        onTreeSelect:function(model,sels){
            if(sels.length>0){
                var rs=sels[0],
                        store=this.orderGrid.store;
                store.proxy.extraParams.CustomerID=rs.data.id;
                this.detailsGrid.store.loadRecords([]);
                store.load();
            }
        },

        onOrderRefresh:function(){
            if(this.orderGrid.store.getCount()>0){
                this.orderGrid.view.select(0);
            }
        },

        onOrderSelect:function(model,rs){
            var me=this;
            if(rs.length>0){
                var g=me.detailsGrid;
                g.store.loadRecords(rs[0].OrderDetailsStore.data.items);
            }
        }

    });
```

这里一定要注意 id 的定义，因为要根据 id 获取视图并加入内容页，并且要与按钮的 action 属性配合好。

Store 的定义可以直接使用不带前缀的类名，因为在对 Store 组件初始化时，会使用 StoreManager 的 lookup 方法返回 Store。

在视图内有 4 个内部逻辑要实现，树加载后选择第一个节点，然后刷新订单，在订单数据加载后选择第一行，随后更新订单明细的数据。

根据 MVC 的要求，所有逻辑都应该由控制器进行控制，所以笔者的做法是不标准的。笔者的看法是，所有的逻辑都要放到控制器中，并不见得容易维护，JavaScript 本身的灵活性很大，没有 Java 或 .NET 那样的“刚性”需求，而且页面内部的一些逻辑也不可能全部搬到服

务器端，因而笔者更倾向于灵活处理，尤其是在中小项目中。不过，这又涉及编码一致性的问题，全部逻辑都在控制器内，这样肯定是符合编码一致性的，而笔者的做法，肯定不符合编码一致性，这也是要权衡的一个方面。

笔者是赞成并支持严格在控制器内进行逻辑控制的做法，大家都应该遵循这样的约定。

注意　因 4.1 Beta 1 版本有错误，因而需要屏蔽粗体代码这句。

4. 定义控制器

控制器的作用是加载模型、Store 和视图，然后把视图加入到内容页。在 controller 目录下创建名称为 Orde.js 的脚本文件，然后加入以下代码：

```
Ext.define('Northwind.controller.Order', {
    extend: 'Ext.app.Controller',


    stores: [
        'Order',
        'OrderDetail',
        'CustomerTreeStore'
    ],

    models: [
        'Order',
        'OrderDetail'
    ],

views:["order.View"],

 refs: [
        {ref:'contentPage', selector: '#contentPage'}
],

 init: function() {
        var me=this,
                view=me.getOrderViewView(),
                c=me.getContentPage();
     me.control({
         '#orderView': {
             render: this.onPanelRendered
         }
     });
        c.add(view);
    },

    onPanelRendered :function(panel){
            var me=this,
                    c=me.getContentPage();
            c.getLayout().setActiveItem(panel);
    }


});
```

通过属性 stores、models 和 views 就可为相应的对象创建 get 方法，并加载它们。属性 refs 的作用是为了能访问到内容页。**要注意的是，视图和模型返回的是类的引用（要求实例化），而 Store 和控制器返回的则是实例化后的对象。**

在 init 方法内，虽然创建了视图，但是因为没设置 renderTo 属性，所以还没渲染，只有在调用 add 方法后才会渲染。因为还没渲染，所以不能使用 setActiveItem 方法将其设置为当前显示视图，必须使用 control 方法监听视图的 render 事件，在将视图渲染后，才调用 setActiveItem 方法将其设置为活动视图。

22.2.10 实现产品管理

产品管理将实现 12.7 节示例的功能，涉及一个模型、3 个 Store、两个视图和一个控制器。在视图中，一个视图是用于显示 Grid 和分页工具条的，一个是用于新增和编辑数据的窗口。

1. 定义模型

因为两个下拉列表的数据结构相当简单，所以将它们的字段定义合并到 Store 中，只需要定义产品的模型即可。在 model 目录创建名称为 Product.js 的脚本文件，加入以下定义：

```
Ext.define('Northwind.model.Product', {
    extend: 'Ext.data.Model',
    fields: [
          {name:'ProductID',type:"int"},
          'ProductName',
          {name:'SupplierID',type:"int"},
          {name:'CategoryID',type:"int"},
          'QuantityPerUnit',
          {name:'UnitPrice',type:"float"},
          {name:'UnitsInStock',type:"int"},
          {name:'UnitsOnOrder',type:"int"},
          {name:'ReorderLevel',type:"int"},
          {name:'Discontinued',type:"bool"},
          "CategoryName","CompanyName"
    ],
    idProperty:"ProductID"
});
```

2. 定义 Store

先定义产品的 Store，在 store 目录在创建名称为 Product.js 文件，并且向其中加入以下定义代码：

```
Ext.define("Northwind.store.Product",{
extend:"Ext.data.Store",
model:'Northwind.model.Product',
pageSize:20,
autoLoad:true,
remoteFilter:true,
remoteSort:true,
sorters:[{property :'ProductID',direction: 'DESC'}],
 proxy:{
       type:"direct",
```

```
            batchActions:false,
            api:{
            read:Ext.app.Product.List,
            destroy:Ext.app.Product.Delete
            },
            reader:{
                    type:"json",
                    root:"data"
            }
    }
})
```

接着定义类别的 Store，创建 CategoryCombo.js 文件，加入以下定义：

```
Ext.define("Northwind.store.CategoryCombo",{
    extend:"Ext.data.Store",
    autoLoad:true,
    fields:[
            {name:"CategoryID",type:"int"},
            "CategoryName"
    ],
    idProperty:"CategoryID",
    proxy:{
            type:"direct",
            directFn:Ext.app.Category.ComboList,
            reader:{
                    type:"json",
                    root:"data"
            }
    }
})
```

最后定义供应商的 Store，创建名称为 SupplierCombo.js 的脚本文件，并且向其中加入以下定义：

```
Ext.define("Northwind.store.SupplierCombo",{
    extend:"Ext.data.Store",
    autoLoad:true,
    fields:[
            {name:"SupplierID",type:"int"},
            "CompanyName"
    ],
    idProperty:"SupplierID",
    proxy:{
            type:"direct",
            directFn:Ext.app.Supplier.ComboList,
            reader:{
                    type:"json",
                    root:"data"
            }
    }
})
```

3. 定义视图

视图有两个，先来定义包含 Grid 的主视图，该视图将直接从 Grid 扩展，问题的重点是

哪些按钮需要在控制器中进行控制，哪些不需要。笔者的拆分方法是，对于涉及视图之间的按钮，在控制器内定义操作，对于只涉及视图内部逻辑操作的按钮，在视图内解决。很简单的道理，如果所有按钮都要在控制器中实现操作，那么进行分页操作的按钮都要重新编码的，这样就要重复很多代码，太没必要了。

在主视图中，涉及两个视图操作的按钮是“增加”和“编辑”按钮，用来对显示对话框进行增加或编辑操作。为了便于同时维护，将“删除”按钮与“增加”和“编辑”按钮归入一类。余下的操作在视图内实现逻辑即可。

目标明确后就可以动手了，在 view 目录下创建 product 子目录，在子目录下创建 View.js 文件，然后加入以下代码：

```
Ext.define('Northwind.view.product.View', {
    extend: 'Ext.grid.Panel',
    alias : 'widget.productview',

   title:"产品管理",
   id:"productView",
   store:"Product",
   selType:"checkboxmodel",
   selModel:{mode:"MULTI"},

    initComponent: function() {
          var me=this;
          me.searchField=Ext.widget("textfield",{
                width:160
          });

          me.paging=Ext.widget("pagingtoolbar",{
                id:"productPaging",
                store:me.store,
                displayInfo:true,
                items:[
                      "|",
                      {text:"增加",action:"productadd"},
                      {text:"编辑",action:"productedit",disabled:true},
                      {text:"删除",action:"productdelete",disabled:true},
                      "|",
                      "查找：",
                      me.searchField,
                      {text:"Go",handler:me.search,scope:me},
                      {text:"显示全部",handler:me.showAll,scope:me}
                ]}
          );

          me.tbar=me.paging;

me.columns=[
          {text:'id',dataIndex:'ProductID',text:"产品编号",width:60},
          {text:'ProductName',dataIndex:'ProductName',text:"产品名称",
                renderer:me.onProductNameRenderer
          },
```

```
        {text:'供应商',dataIndex:'CompanyName',titleAlign:"center",
                sortable:false,flex:1
        },
        {text:'类别',dataIndex:'CategoryName',titleAlign:"center",
                sortable:false,flex:1
        },
{text:'单位',dataIndex:'QuantityPerUnit',titleAlign:"center",flex:1},
{xtype:"numbercolumn",text:'单价',dataIndex:'UnitPrice',titleAlign:"center",
        align:"right",format:"$0.00",width:60
},
{xtype:"numbercolumn",text:'库存',dataIndex:'UnitsInStock',align:"center",
        format:"0",width:60
},
{xtype:"numbercolumn",text:'订购量',dataIndex:'UnitsOnOrder',align:"center",
        format:"0",width:60
},
{xtype:"numbercolumn",text:'再订购量',dataIndex:'ReorderLevel',align:"center",
        format:"0",width:60
},
{xtype:"booleancolumn",text:'停产',dataIndex:'Discontinued',
        align:"center",trueText:"是",falseText:"否",width:60
}
]
    this.callParent(arguments);

    me.on("selectionchange",me.onSelectionchange);
},
```

```
search:function(){
        var me=this,
                store=me.store,
                search=me.searchField.getValue();
        if(search && search.length>0){
                store.currentPage=1;
        store.filters.clear();
                store.filter("ProductName",search);
        }
},

showAll:function(){
        var store=this.store;
        store.currentPage=1;
        store.clearFilter();
},

onProductNameRenderer:function(v,meta,rec,row,col,store){
        var filter = store.filters.items;
        if(filter.length>0){
                var value=filter[0].value;
                return v.replace(new RegExp(value,'gi'),function(m){
                        return "<font color='red'>"+m+"</font>";
                })
        }else{
                return v;
        }
```

```
    },

    onSelectionchange:function(seltype,rs){
            var me=this,
                    length=!(rs.length>0);
            me.paging.down("button[action=productedit]").setDisabled(length);
            me.paging.down("button[action=productdelete]").setDisabled(length);
    }

});
```

在代码中，向“增加”、“编辑”和“删除”按钮都添加了 action 属性，以便于在控制器中找到按钮。

扩展的好处就是可根据需要创建一些属性，指向特定组件，进行特定的操作，方法的作用域也可控制在当前对象内，例如搜索输入框可直接通过属性来操作，不需要搜索或获取组件。

接着定义产品编辑对话框，这和登录对话框一样，难度不大，从 Window 对象扩展就行，但是不要设置单件模式，因为这样可能会因为找不到下拉对话框的 Store 而出错。在 product 子目录下创建名称为 EditView.js 的脚本文件，并且向其中加入以下代码：

```
Ext.define("Northwind.view.product.EditView",{
   extend:"Ext.window.Window",
   alias: 'widget.producteditview',
    hideMode: 'offsets',
    closeAction: 'hide',
    resizable: false,
title:'',
width:400,
height:350,
modal:true,

initComponent: function() {
      var me=this;

      me.form=Ext.create(Ext.form.Panel,{
              border:false,bodyPadding:5,
              bodyStyle:"background:#DFE9F6",
              trackResetOnLoad:true,waitTitle:"请等待...",
              api:{submit:Ext.app.Product.Add},
              fieldDefaults: {
              labelWidth:80,labelSeparator:": ",anchor:"0"
              },
              items:[
                      {xtype:"textfield",fieldLabel:"产品编号",
                              name:"ProductID",readOnly:true,
                              readOnlyCls:"x-item-disabled"
                      },
                      {xtype:"textfield",fieldLabel:"产品名称",
                              name:"ProductName",allowBlank:false,
                              maxLength:40
                      },
      {xtype:"combobox",name:"SupplierID",
              fieldLabel:"供应商",valueField:"SupplierID",
```

```
            displayField:"CompanyName",store:"SupplierCombo",
            forceSelection:false,queryMode:"local",
            allowBlank:false,minChars:1
            },
    {xtype:"combobox",name:"CategoryID",
            fieldLabel:"产品类别",valueField:"CategoryID",
            displayField:"CategoryName",store:"CategoryCombo",
            forceSelection:true,queryMode:"local",
            allowBlank:false,minChars:1
            },
    {xtype:"textfield",fieldLabel:"单位",
            name:"QuantityPerUnit",maxLength:20
    },
    {xtype:"numberfield",fieldLabel:"单价",
            name:"UnitPrice",hideTrigger:true,
            minValue:0,autoStripChars:true
    },
    {xtype:"displayfield",fieldLabel:"库存",name:"UnitsInStock"},
    {xtype:"displayfield",fieldLabel:"订购数量",name:"UnitsOnOrder"},
    {xtype:"displayfield",fieldLabel:"再订购量",name:"ReorderLevel"},
    {xtype:"checkbox",fieldLabel:"",boxLabel:'停产',
            name:'Discontinued',inputValue:true,hideEmptyLabel:false
                    }
            ],
            dockedItems: [{
    xtype: 'toolbar',dock:'bottom',ui:'footer',layout:{pack:"center"},
        items: [
    {text:"保存",disabled:true,formBind:true,handler:me.onSave,scope:me},
            {text:"重置",handler:me.onReset,scope:me}
                        ]
                    }]
            });

            me.items=[me.form]
            me.callParent(arguments);

    },

    initEvents:function(){
            this.on("show",this.focusField);
    },
    focusField:function(){
            var f=this.form.getForm();
            f.findField("ProductName").focus();
    },

    onSave:function(){
            var me=this,
                    f=me.form.getForm(),
                    id=f.findField("ProductID").getValue();
            if(f.isValid() && f.isDirty()){
                    f.submit({
                            waitMsg:"正在保存...",
                success: function(form, action){
                    var me=this,
```

```
                result = action.result,
                f=me.form.getForm(),
                companyname=f.findField("SupplierID").getRawValue(),
                categoryname=f.findField("CategoryID").getRawValue(),
                store=Ext.StoreManager.lookup("Product"),
                rec=f.getRecord();
        f.updateRecord(rec);
                rec.set("CompanyName",companyname);
                rec.set("CategoryName",categoryname);
        if(result.ProductID){
                rec.set("ProductID",result.ProductID)
                store.insert(0,rec);
                rec.commit();
                                m=store.model;
                                f.loadRecord(new m());
            }else{
                rec.commit();
                me.close();
            }
        },
        failure: function(form, action){
            if (action.failureType === "connect") {
                Ext.Msg.alert('错误',
                    '状态:'+action.response.status+': '+
                    action.response.statusText);
                return;
            }
          if(action.failureType==="server"){
                                Ext.Msg.alert('错误', "提交失败，运行错误！");
                    }
                },
                scope:me
                  });
            }else{
                    Ext.Msg.alert("修改","请修改数据后再提交。")
            }
    },

    onReset:function(){
            this.form.getForm().reset();
    }

})
```

以上代码与 12.7 节示例的区别是在这里将它写成了一个扩展，基本的代码变化不大。

4. 定义控制器

定义产品控制器要加载 3 个 Store、一个模型和两个视图，将主视图加入内容页，并控制“增加”、“编辑”和“删除”按钮的操作。在 controller 目录创建 Product.js 文件并加入以下代码：

```
Ext.define('Northwind.controller.Product', {
    extend: 'Ext.app.Controller',
```

```
stores: [
    'Product',
    'SupplierCombo',
    'CategoryCombo'
],

models: [
    'Product'
],

views:["product.View","product.EditView"],

 refs: [
       {ref:'contentPage', selector: '#contentPage'}
],

 init: function() {
       var me=this,
              view=me.getProductViewView(),
              c=me.getContentPage();
     me.control({
         '#productView': {
             render: me.onPanelRendered
         },
         '#productPaging button[action=productadd]':{
              click:me.AddProduct
         },
         '#productPaging button[action=productedit]':{
              click:me.EditProduct
         },
         '#productPaging button[action=productdelete]':{
              click:me.DelProduct
         }
     });
       c.add(view);
},

getEditView:function(){
       var me=this;
              view= me.editview;
       if(!view){
              view=me.editview=Ext.widget("producteditview");
       }
       return view;
},

onPanelRendered :function(panel){
       var me=this,
              c=me.getContentPage();
       c.getLayout().setActiveItem(panel);
       me.grid=panel;
},

AddProduct:function(){
```

```
        var me=this,
                win=me.getEditView(),
                f=win.form,
                m=me.getProductStore().model;
        f.getForm().api.submit=Ext.app.Product.Add;
        f.loadRecord(new m());
        win.setTitle("添加新产品")
        win.show();
    },

    EditProduct:function(){
        var me=this,
                rs=me.grid.getSelectionModel().getSelection();
        if(rs.length>0){
                rs=rs[0];
        var win=me.getEditView();
                win.setTitle("编辑产品: "+rs.get("ProductName"));
                win.form.getForm().api.submit=Ext.app.Product.Edit;
                win.form.getForm().loadRecord(rs);
                win.show();
    }else{
                Ext.Msg.alert("提示信息","请选择一条记录进行编辑。");
    }
    },

    DelProduct:function(){
var grid=this.grid,
rs=grid.getSelectionModel().getSelection();
if(rs.length > 0){
        var content=["确定删除以下产品? "];
        for(var i=0;ln=rs.length,i<ln;i++){
                content.push(rs[i].data.ProductName);
        }
        Ext.Msg.confirm("删除记录",content.join("<br/>"),function(btn){
        if(btn=="yes"){
                var me=this,store=me.store,ids=[];
                rs=me.getSelectionModel().getSelection();
                for(var i=0;ln=rs.length,i<ln;i++){
                        ids.push(rs[i].data.ProductID);
                }
                Ext.app.Product.Delete(ids.join(","),function(result,e){
                if(e.type=="exception"){
                        Ext.Msg.alert("提示信息",e.message);
                }else{
                if(result.success){
                var msg=[].concat(["以成功删除以下产品: "],result.msg);
                msg.push("----------");
                msg.push("列表将刷新。");
                Ext.Msg.alert("删除成功",msg.join("<br/>"),function(){
                        me.store.load();
                });
        }else{
                                        Ext.Msg.alert("提示信息",result.msg);
                                        }
                                }
```

```
                                    });
                                }
                            },grid)
                    }
                }

            });
```

在 init 方法中，将视图添加到内容页的操作与订单管理视图的操作相同，都是使用 add 方法添加，并使用 render 方法切换。

通过 control 方法定义了 3 个按钮的操作。

因为此时的产品编辑对话框只是加载了类原型，还没实例化，在何时实例化会是个问题，所以，最适合的方法是创建一个 getEditView 方法，判断 editview 属性是否存在，如果不存在，则创建 EditView 的实例并返回，该方法在 Ext JS 的源代码中经常会看到。

修改了删除记录的提交方式，直接提交一个由 id 组成的字符串到服务器端的方法处理起来比较方便，所以对服务器端的删除方法做了一些改变，代码如下：

C#

```
[DirectMethod]
public JObject Delete(string value)
{
    List<int> ids = new List<int>();
    foreach (string sid in value.Split(new Char[] { ',' }, StringSplitOptions.
        RemoveEmptyEntries))
    {
        int id;
        if (int.TryParse(sid, out id))
        {
            ids.Add(id);
        }
    }
    if (ids.Count > 0)
    {
        using (NorthwindEntities ne = new NorthwindEntities())
        {
            try
            {
                JArray delList = new JArray();
                var q = ne.Products.Where(m => ids.Contains(m.ProductID));
                foreach (var c in q)
                {
                    delList.Add(string.Format("[{0}]{1}", c.ProductID,
                        c.ProductName));
                    ne.DeleteObject(c);
                }
                ne.SaveChanges();
                return new JObject(
                    new JProperty("success", true),
                    new JProperty("msg", delList)
                );
    }
```

```
                catch (Exception ee)
                {
                    return new JObject(
                        new JProperty("success", false),
                        new JProperty("msg", ee.Message)
                    );
                }
            }
        }
        else
        {
            return new JObject(
                new JProperty("success", false),
                new JProperty("msg", "没有要删除的产品。")
            );
        }
    }
```

Java

```
@DirectMethod
public Map<String, Object> Delete(String data){
   String[] ids=data.split(",");
   String sql="";
   if(ids.length>0){
           for (String id : ids) {
                   if (id.matches("\\d*")) {
                           sql=sql + id+",";
                   }
           }
           sql=sql.substring(0,sql.length()-1);
}else {
    Map<String, Object> jo=new HashMap<String, Object>();
   jo.put("success", false);
   jo.put("msg", "没有要删除的产品。");
   return jo;
}
 String connectionUrl = "jdbc:sqlserver://192.168.0.254:1433;" +
 "databaseName=Northwind;;user=sa;password=abcd-1234";
 Connection con = null;
 Statement stmt = null;
 ResultSet rs = null;
 try {

        Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
        con = DriverManager.getConnection(connectionUrl);
stmt=con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.
    CONCUR_UPDATABLE);

rs=stmt.executeQuery("select * from products where productid in ("+
            sql +")");
ArrayList<String> delList= new ArrayList<String>();
while (rs.next()) {
   delList.add(String.format("[%1$s]%2$s", rs.getString ("ProductID"),
       rs.getString("ProductName")));
```

```
            rs.deleteRow();
        }
        Map<String,Object> json=new HashMap<String, Object>();
        json.put("success",new JsonPrimitive(true));
        json.put("msg", delList);
        return json;
    }
    catch (Exception e) {
        throw new RuntimeException(e.getMessage());
    }
    finally {
            if (rs != null) try { rs.close(); } catch(Exception e) {}
            if (stmt != null) try { stmt.close(); } catch(Exception e) {}
            if (con != null) try { con.close(); } catch(Exception e) {}
        }
    }
```

在代码中，因为提交的是字符串，所以处理就简化了，直接使用 split 方法拆分就可以了。

22.2.11　示例效果

在浏览器中打开 index.html 页面，使用 user1（密码 123456）登录后，将看到如图 22-1 所示的页面效果，在其中单击“订单管理”和“产品管理”可看到内容页的切换效果。

注意　要使示例正确运行，必须修改 ext-all-debug.js 文件中 23785 行的 addManagedListener 方法，在 23789 行加入“if(!item){return}”。这还是因为 4.1 Beta 1 会造成错误。

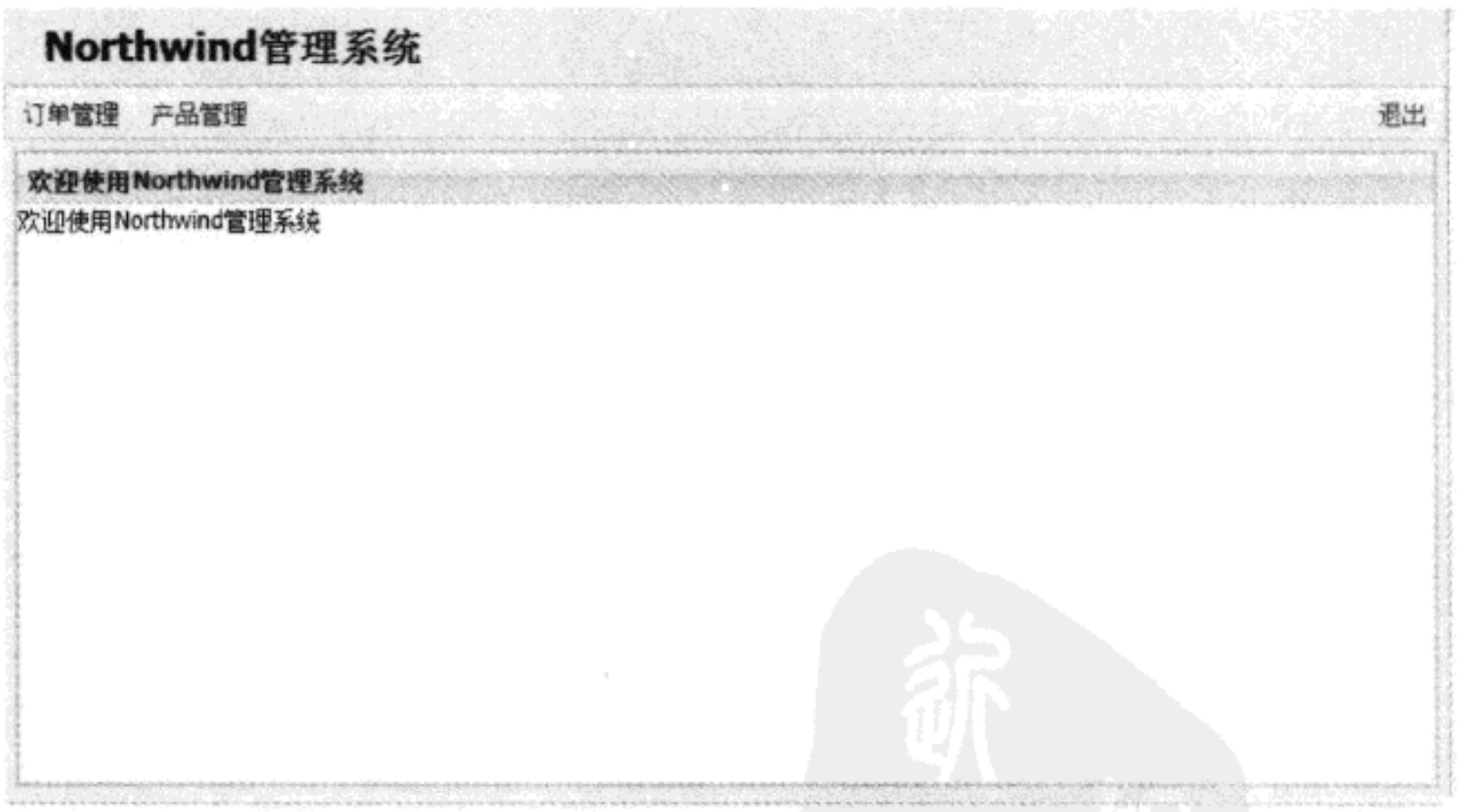


图 22-1　MVC 框架示例的页面效果

22.3 本章小结

使用 MVC 应用架构的好处自然很多，但是如果设计跟不上，尤其是视图划分出现问题的时候，进行调试遇到的问题会比较多。在调试时，很多错误都是显示 ext-all-debug.js 的文件错误，而不是显示具体文件错误，这样增加了调试的难度。

要使用好 MVC 应用架构，重点在设计，而不是编码，这是与一般开发过程的最大不同，也是最容易出问题的地方，要三思而后行。

以下几个链接是笔者翻译的由官方网站提供的 MVC 框架开发的文章，非常值得研究和学习。

- Ext JS 应用架构设计：http://blog.csdn.net/tianxiaode/article/details/6562113
- Ext JS 应用架构设计（二）：http://blog.csdn.net/tianxiaode/article/details/6797680
- Ext JS 应用架构设计（三）：http://blog.csdn.net/tianxiaode/article/details/6799105

附录　简写类名与 Ext JS 类名对照表

简写类名	实际类名
AbsoluteLayout	Ext.layout.container.Absolute
AbstractCardLayout	Ext.layout.container.AbstractCard
AbstractComponent	Ext.AbstractComponent
AbstractManager	Ext.AbstractManager
AbstractContainer	Ext.container.AbstractContainer
AbstractFitLayout	Ext.layout.container.AbstractFit
AbstractMixedCollection	Ext.util.AbstractMixedCollection
AbstractPanel	Ext.panel.AbstractPanel
AbstractPlugin	Ext.AbstractPlugin
AbstractStore	Ext.data.AbstractStore
AbstractView	Ext.view.AbstractView
AccordionLayout	Ext.layout.container.Accordion
Action	Ext.Action
ActionColumn	Ext.grid.column.Action
Ajax	Ext.Ajax
AjaxProxy	Ext.data.proxy.Ajax
AnchorLayout	Ext.layout.container.Anchor
Anim	Ext.fx.Anim
Animate	Ext.util.Animate
Animator	Ext.fx.Animator
Application	Ext.app.Application
Area	Ext.chart.series.Area
ArrayReader	Ext.data.reader.Array
Association	Ext.data.Association
AutoLayout	Ext.layout.container.Auto
Axis	Ext.chart.axis.Axis
GaugeAxis	Ext.chart.axis.Gauge
TimeAxis	Ext.chart.axis.Time
Bar	Ext.chart.series.Bar
Base	Ext.Base
BaseField	Ext.form.field.Base
BasicForm	Ext.form.Basic
Batch	Ext.data.Batch

（续）

简写类名	实际类名
BelongsToAssociation	Ext.data.BelongsToAssociation
BooleanColumn	Ext.grid.column.Boolean
BorderLayout	Ext.layout.container.Border
BoundList	Ext.view.BoundList
BoundListKeyNav	Ext.view.BoundListKeyNav
BoxLayout	Ext.layout.container.Box
Button	Ext.button.Button
ButtonGroup	Ext.container.ButtonGroup
CardLayout	Ext.layout.container.Card
Cartesian	Ext.chart.series.Cartesian
Category	Ext.chart.axis.Category
CellEditing	Ext.grid.plugin.CellEditing
Chart	Ext.chart.Chart
Checkbox	Ext.form.field.Checkbox
CheckboxGroup	Ext.form.CheckboxGroup
CheckboxGroupLayout	Ext.layout.container.CheckboxGroup
CheckboxModel	Ext.selection.CheckboxModel
CheckItem	Ext.menu.CheckItem
Ext.Class	Ext.Class
ClassManager	Ext.ClassManager
ClickRepeater	Ext.util.ClickRepeater
ClientProxy	Ext.data.proxy.Client
ColorPicker	Ext.menu.ColorPicker
ColumnChart	Ext.chart.series.Column
ColumnLayout	Ext.layout.container.Column
ComboBox	Ext.form.field.ComboBox
Component	Ext.Component
ComponentDragger	Ext.util.ComponentDragger
ComponentLoader	Ext.ComponentLoader
ComponentManager	Ext.ComponentManager
ComponentQuery	Ext.ComponentQuery
CompositeElement	Ext.dom.CompositeElement
FxCompositeElementCSS	Ext.fx.target.CompositeElementCSS
CompositeElementLite	Ext.dom.CompositeElementLite
Connection	Ext.data.Connection
Container	Ext.container.Container
Controller	Ext.app.Controller
CookieProvider	Ext.state.CookieProvider

（续）

简写类名	实际类名
Cookies	Ext.util.Cookies
CSS	Ext.util.CSS
CycleButton	Ext.button.Cycle
Model	Ext.data.Model
Ext.Date	Ext.Date
DateColumn	Ext.grid.column.Date
DateField	Ext.form.field.Date
MenuDatePicker	Ext.menu.DatePicker
DatePicker	Ext.picker.Date
DD	Ext.dd.DD
DDProxy	Ext.dd.DDProxy
DDTarget	Ext.dd.DDTarget
DelayedTask	Ext.util.DelayedTask
DirectLoad	Ext.form.action.DirectLoad
DirectManager	Ext.direct.Manager
DirectProvider	Ext.direct.Provider
DirectProxy	Ext.data.proxy.Direct
DirectStore	Ext.data.DirectStore
DirectSubmit	Ext.form.action.DirectSubmit
Display	Ext.form.field.Display
Helper	Ext.dom.Helper
AbstractHelper	Ext.dom.AbstractHelper
DomQuery	Ext.DomQuery
DragDrop	Ext.dd.DragDrop
DragDropManager	Ext.dd.DragDropManager
DragSource	Ext.dd.DragSource
DragTracker	Ext.dd.DragTracker
DragZone	Ext.dd.DragZone
DrawColor	Ext.draw.Color
DrawComponent	Ext.draw.Component
DrawCompositeSprite	Ext.draw.CompositeSprite
DrawSprite	Ext.draw.Sprite
DropTarget	Ext.dd.DropTarget
DropZone	Ext.dd.DropZone
Easing	Ext.fx.Easing
Editing	Ext.grid.plugin.Editing
Editor	Ext.Editor
Element	Ext.dom.Element

（续）

简写类名	实际类名
FxElementCSS	Ext.fx.target.ElementCSS
AbstractElement	Ext.dom.AbstractElement
ElementLoader	Ext.ElementLoader
Errors	Ext.data.Errors
Event	Ext.util.Event
EventManager	Ext.EventManager
EventObject	Ext.EventObject
EventObjectImpl	Ext.EventObjectImpl
ExceptionEvent	Ext.direct.ExceptionEvent
Ext.Array	Ext.Array
Ext.Function	Ext.Function
Ext.is	Ext.is
Ext.JSON	Ext.JSON
Ext.Object	Ext.Object
Ext.String	Ext.String
Ext.supports	Ext.supports
Featrue	Ext.grid.featrue.Featrue
FeatrueDetector	Ext.env.FeatrueDetector
Field	Ext.data.Field
FieldAncestor	Ext.form.FieldAncestor
FieldContainer	Ext.form.FieldContainer
FieldSet	Ext.form.FieldSet
File	Ext.form.field.File
Fill	Ext.toolbar.Fill
Filter	Ext.util.Filter
FitLayout	Ext.layout.container.Fit
FlashComponent	Ext.flash.Component
Floating	Ext.util.Floating
FocusManager	Ext.FocusManager
FormAction	Ext.form.action.Action
Format	Ext.util.Format
FormField	Ext.form.field.Field
FormPanel	Ext.form.Panel
PropertyGrid	Ext.grid.property.Grid
Column	Ext.grid.column.Column
GridViewDropZonePlugin	Ext.grid.plugin.DragDrop
GridPanel	Ext.grid.Panel
GridView	Ext.grid.View

（续）

简写类名	实际类名
Grouper	Ext.util.Grouper
Grouping	Ext.grid.featrue.Grouping
GroupingSummary	Ext.grid.featrue.GroupingSummary
Handle	Ext.resizer.Handle
HashMap	Ext.util.HashMap
HasManyAssociation	Ext.data.HasManyAssociation
HBoxLayout	Ext.layout.container.HBox
Header	Ext.panel.Header
PropertyColumnModel	Ext.grid.property.HeaderContainer
HeaderContainer	Ext.grid.header.Container
HeaderResizer	Ext.grid.plugin.HeaderResizer
Hidden	Ext.form.field.Hidden
History	Ext.util.History
HtmlEditor	Ext.form.field.HtmlEditor
Img	Ext.Img
Inflector	Ext.util.Inflector
JsonP	Ext.data.JsonP
JsonProvider	Ext.direct.JsonProvider
JsonReader	Ext.data.reader.Json
KeyMap	Ext.util.KeyMap
KeyNav	Ext.util.KeyNav
Label	Ext.chart.Label
Labelable	Ext.form.Labelable
Layer	Ext.Layer
Legend	Ext.chart.Legend
LegendItem	Ext.chart.LegendItem
Line	Ext.chart.series.Line
LoadAction	Ext.form.action.Load
Loader	Ext.Loader
LoadMask	Ext.LoadMask
LocalStorageProxy	Ext.data.proxy.LocalStorage
Mask	Ext.chart.Mask
MemoryProyx	Ext.data.proxy.Memory
Menu	Ext.menu.Menu
MenuItem	Ext.menu.Item
MenuManager	Ext.menu.Manager
MenuSeparator	Ext.menu.Separator
MessageBox	Ext.window.MessageBox

（续）

简写类名	实际类名
MixedCollection	Ext.util.MixedCollection
ModelManager	Ext.ModelManager
MultiSlider	Ext.slider.Multi
NodeInterface	Ext.data.NodeInterface
Ext.Number	Ext.Number
NumberColumn	Ext.grid.column.Number
NumberField	Ext.form.field.Number
Numeric	Ext.chart.axis.Numeric
Observable	Ext.util.Observable
Operation	Ext.data.Operation
PagingToolbar	Ext.toolbar.Paging
Panel	Ext.panel.Panel
ColorPalette	Ext.picker.Color
Picker	Ext.form.field.Picker
TimePicker	Ext.picker.Time
Pie	Ext.chart.series.Pie
PluginManager	Ext.PluginManager
Point	Ext.util.Point
PollingProvider	Ext.direct.PollingProvider
ProgressBar	Ext.ProgressBar
PropertyStore	Ext.grid.property.Store
Proxy	Ext.data.proxy.Proxy
QuickTip	Ext.tip.QuickTip
QuickTipManager	Ext.tip.QuickTipManager
Radar	Ext.chart.series.Radar
Radio	Ext.form.field.Radio
RadioGroup	Ext.form.RadioGroup
Reader	Ext.data.reader.Reader
Region	Ext.util.Region
Registry	Ext.dd.Registry
RemotingEvent	Ext.direct.RemotingEvent
RemotingProvider	Ext.direct.RemotingProvider
Request	Ext.data.Request
Resizer	Ext.resizer.Resizer
ResizeTracker	Ext.resizer.ResizeTracker
RestProxy	Ext.data.proxy.Rest
ResultSet	Ext.data.ResultSet
RowBody	Ext.grid.featrue.RowBody

（续）

简写类名	实际类名
RowEditing	Ext.grid.plugin.RowEditing
RowModel	Ext.selection.RowModel
RowNumberer	Ext.grid.RowNumberer
Scatter	Ext.chart.series.Scatter
ScriptTagProxy	Ext.data.proxy.JsonP
ScrollManager	Ext.dd.ScrollManager
SelectionModel	Ext.selection.Model
Series	Ext.chart.series.Series
Gauge	Ext.chart.series.Gauge
Server	Ext.data.proxy.Server
SessionStorageProxy	Ext.data.proxy.SessionStorage
Shadow	Ext.Shadow
SingleSlider	Ext.slider.Single
SliderTip	Ext.slider.Tip
Sortable	Ext.util.Sortable
Sorter	Ext.util.Sorter
SortTypes	Ext.data.SortTypes
Spacer	Ext.toolbar.Spacer
Spinner	Ext.form.field.Spinner
SplitButton	Ext.button.Split
Splitter	Ext.resizer.Splitter
StandardSubmit	Ext.form.action.StandardSubmit
Stateful	Ext.state.Stateful
StateManager	Ext.state.Manager
StateProvider	Ext.state.Provider
StatusProxy	Ext.dd.StatusProxy
Store	Ext.data.Store
StoreManager	Ext.data.StoreManager
Submit	Ext.form.action.Submit
Summary	Ext.grid.featrue.Summary
Surface	Ext.draw.Surface
Svg	Ext.draw.engine.Svg
Tab	Ext.tab.Tab
TabBar	Ext.tab.Bar
TableChunker	Ext.view.TableChunker
TableLayout	Ext.layout.container.Table
TabPanel	Ext.tab.Panel
FxTarget	Ext.fx.target.Target

（续）

简写类名	实际类名
FxCompositeElement	Ext.fx.target.CompositeElement
FxCompositeSprite	Ext.fx.target.CompositeSprite
FxElement	Ext.fx.target.Element
FxSprite	Ext.fx.target.Sprite
TaskManager	Ext.TaskManager
TaskRunner	Ext.util.TaskRunner
Template	Ext.Template
TemplateColumn	Ext.grid.column.Template
Text	Ext.form.field.Text
TextArea	Ext.form.field.TextArea
ToolBarTextItem	Ext.toolbar.TextItem
TextMetrics	Ext.util.TextMetrics
TimeField	Ext.form.field.Time
Tip	Ext.tip.Tip
Tool	Ext.panel.Tool
Toolbar	Ext.toolbar.Toolbar
ToolbarItem	Ext.toolbar.Item
Separator	Ext.toolbar.Separator
ToolTip	Ext.tip.ToolTip
FxComponent	Ext.fx.target.Component
Transaction	Ext.direct.Transaction
TreeData	Ext.data.Tree
TreePanel	Ext.tree.Panel
TreeStore	Ext.data.TreeStore
TreeView	Ext.tree.View
Trigger	Ext.form.field.Trigger
Types	Ext.data.Types
Validations	Ext.data.validations
VBoxLayout	Ext.layout.container.VBox
DataView	Ext.view.View
TreeViewDragDropPlugin	Ext.tree.plugin.TreeViewDragDrop
Viewport	Ext.container.Viewport
TableView	Ext.view.Table
Vml	Ext.draw.engine.Vml
VTypes	Ext.form.field.VTypes
WebStorageProxy	Ext.data.proxy.WebStorage
Window	Ext.window.Window
WindowManager	Ext.WindowManager

（续）

简写类名	实际类名
Writer	Ext.data.writer.Writer
XmlReader	Ext.data.reader.Xml
XmlWriter	Ext.data.writer.Xml
XTemplate	Ext.XTemplate
XTemplateCompiler	Ext.XTemplateCompiler
XTemplateParser	Ext.XTemplateParser
ZIndexManager	Ext.ZIndexManager
XmlStore	Ext.data.XmlStore
JsonStore	Ext.data.JsonStore
JsonPStore	Ext.data.JsonPStore
ArrayStore	Ext.data.ArrayStore
NodeStore	Ext.data.NodeStore
PropGridProperty	Ext.grid.property.Property
TablePanel	Ext.panel.Table
Layout	Ext.layout.Layout
ContainerLayout	Ext.layout.container.Container
ComponentLayout	Ext.layout.component.Component
AbstractDockLayout	Ext.layout.component.AbstractDock
AutoComponentLayout	Ext.layout.component.Auto
BodyLayout	Ext.layout.component.Body
BoundListLayout	Ext.layout.component.BoundList
ButtonLayout	Ext.layout.component.Button
DockLayout	Ext.layout.component.Dock
DrawLayout	Ext.layout.component.Draw
EditorLayout	Ext.layout.component.Editor
FieldSetLayout	Ext.layout.component.FieldSet
ProgressBarLayout	Ext.layout.component.ProgressBar
TabLayout	Ext.layout.component.Tab
TipLayout	Ext.layout.component.Tip
FieldLayout	Ext.layout.component.field.Field
FileLayout	Ext.layout.component.field.File
HtmlEditorLayout	Ext.layout.component.field.HtmlEditor
SliderLayout	Ext.layout.component.field.Slider
TextLayout	Ext.layout.component.field.Text
TextAreaLayout	Ext.layout.component.field.TextArea
TriggerLayout	Ext.layout.component.field.Trigger
ShadowPool	Ext.ShadowPool
LocalStorageProvider	Ext.state.LocalStorageProvider

（续）

简写类名	实际类名
TreeModel	Ext.selection.TreeModel
CellModel	Ext.selection.CellModel
CheckColumn	Ext.ux.CheckColumn
Lockable	Ext.grid.Lockable
LockingView	Ext.grid.LockingView
AbstractSummary	Ext.grid.featrue.AbstractSummary
Callout	Ext.chart.Callout
Highlight	Ext.chart.Highlight
Navigation	Ext.chart.Navigation
Shape	Ext.chart.Shape
ChartTip	Ext.chart.Tip
TipSurface	Ext.chart.TipSurface
Theme	Ext.chart.theme.Theme
BaseTheme	Ext.chart.theme.Base
AbstractAxis	Ext.chart.axis.Abstract
Radial	Ext.chart.axis.Radial
Thumb	Ext.slider.Thumb
Memento	Ext.util.Memento
DirectEvent	Ext.direct.Event
RemotingMethod	Ext.direct.RemotingMethod
CubicBezier	Ext.fx.CubicBezier
FxManager	Ext.fx.Manager
Queue	Ext.fx.Queue
PropertyHandler	Ext.fx.PropertyHandler
TreeViewDropZone	Ext.tree.ViewDropZone
TreeViewDragZone	Ext.tree.ViewDragZone
ViewDragZone	Ext.view.DragZone
ViewDropZone	Ext.view.DropZone
SplitterTracker	Ext.resizer.SplitterTracker
PanelProxy	Ext.panel.Proxy
PanelDD	Ext.panel.DD
GridHeaderDragZone	Ext.grid.header.DragZone
GridViewDropZone	Ext.grid.ViewDropZone
ElementContainer	Ext.util.ElementContainer
ProtoElement	Ext.util.ProtoElement
Renderable	Ext.util.Renderable
Context	Ext.layout.Context
ContextItem	Ext.layout.ContextItem
Bindable	Ext.util.Bindable
PagingScroller	Ext.grid.PagingScroller