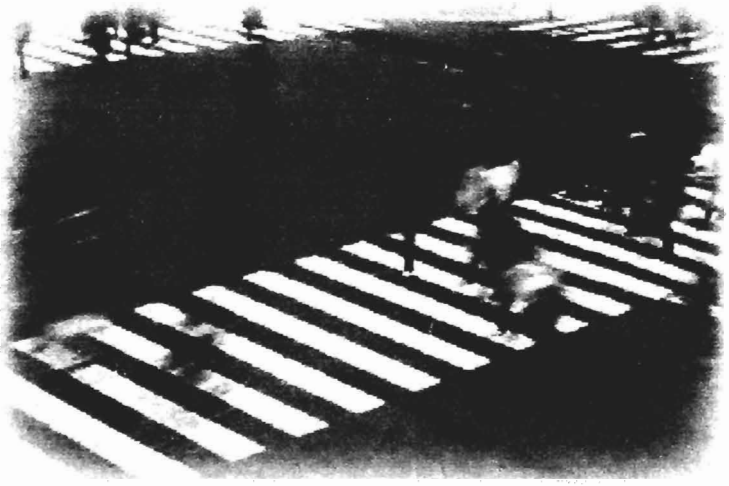
TURING 图灵程序设计丛书

Erlang程序设计

Programming Erlang Software for a Concurrent World



[瑞典] Joe Armstrong 著
赵东炜 金尹 译

人民邮电出版社
北 京

图书在版编目（CIP）数据

Erlang 程序设计 /（瑞典）阿姆斯特朗（Armstrong, J.）著；赵东炜，金尹译 . —北京：人民邮电出版社，2008.12（2009.1 重印）
（图灵程序设计丛书）
书名原文：Programming Erlang: Software for a Concurrent World
ISBN 978-7-115-18869-4

Ⅰ. E… Ⅱ. ①阿…②赵…③金… Ⅲ. 程序语言 — 程序设计 Ⅳ. TP312

中国版本图书馆CIP数据核字（2008）第142666号

内 容 提 要

本书是讲述下一代编程语言 Erlang 的权威著作，主要涵盖顺序型编程、异常处理、编译和运行代码、并发编程、并发编程中的错误处理、分布式编程、多核编程等内容。本书将帮助读者在消息传递的基础上构建分布式的并发系统，免去锁与互斥技术的羁绊，使程序在多核 CPU 上高效运行。本书讲述的各种设计方法和行为将成为设计容错与分布式系统中的利器。

图灵程序设计丛书

Erlang程序设计

◆ 著 [瑞典] Joe Armstrong
译 赵东炜 金 尹
责任编辑 傅志红
执行编辑 杨 爽

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京顺义振华印刷厂印刷

◆ 开本：800×1000 1/16
印张：27.75
字数：691 千字 2008 年 12 月第 1 版
印数：3 001 – 5 000 册 2009 年 1 月北京第 2 次印刷

著作权合同登记号 图字：01-2008-3858号
ISBN 978-7-115-18869-4/TP

定价：79.00元

读者服务热线：(010)88593802 印装质量热线：(010)67129223
反盗版热线：(010)67171154

版 权 声 明

这个世界是并行的。

如果希望将程序的行为设计得与真实世界物体的行为相一致，那么程序就应具有并发结构。

使用专门为并发应用设计的语言，开发将变得极为简便。

Erlang 程序模拟了我们如何思考，如何交互。

——Joe Armstrong

推 荐 序

Erlang算不上是一种“大众流行”的程序设计语言，而且即使是Erlang的支持者，大多数也对于Erlang成为“主流语言”并不持乐观态度。然而，自从2006年以来，Erlang语言确实在国内外一批精英程序员中暗流涌动，光我所认识和听说的，就有不少于一打技术高手像着了魔一样迷上了这种已经有二十多年历史的老牌语言。这是一件相当奇怪的事情。因为就年龄而言，Erlang大约与Perl同年，比C++年轻四岁，长Java差不多十岁，但Java早已经是工业主流语言，C++和Perl甚至已经进入其生命周期的下降阶段。照理说，一个被扔在角落里二十多载无人理睬的老家伙合理的命运就是坐以待毙，没想到Erlang却像是突然吃了返老还童丹似的在二十多岁的“高龄”又火了一把，不但对它感兴趣的人数量激增，而且还成立了一些组织，开发实施了一些非常有影响力的软件项目。这是怎么回事呢？

根本原因在于Erlang天赋异禀恰好适应了计算环境变革的大趋势：CPU的多核化与云计算。

自2005年C++标准委员会主席Herb Sutter在*Dr. Dobb's Journal*上发表《免费午餐已经结束》一文以来，人们已经确凿无疑地认识到，如果未来不能有效地以并行化的软件充分利用并行化的硬件资源，我们的计算效率就会永远停滞在仅仅略高于当前的水平上，而不得动弹。因此，未来的计算必然是并行的。Herb Sutter本人曾表示，如果一个语言不能够以优雅可靠的方式处理并行计算的问题，那它就失去了在21世纪的生存权。“主流语言”当然不想真的丧失掉这个生存权，于是纷纷以不同的方式解决并行计算的问题。就C/C++而言，除了标准委员会致力于以标准库的方式来提供并行计算库之外，标准化的OpenMP和MPI，以及Intel的Threading Building Blocks库也都是可信赖的解决方案；Java在5.0版中引入了意义重大的concurrency库，得到Java社区的一致推崇；而微软更是采用了多种手段来应对这一问题：先是在.NET中引入APM，随后又在Robotics Studio中提供了CCR库，最近又发布了Parrallel FX和MPI.NET，可谓不遗余力。然而，这些手法都可以视为亡羊补牢，因为这些语言和基础设施在创造时都没有把并行化的问题放到优先的位置来考虑。与它们相反，Erlang从其构思的时候起，就把“并行”放到了中心位置，其语言机制和细节的设计无不从并行角度出发和考虑，并且在长达二十年的发展完善中不断成熟。今天，Erlang可以说是为数不多的天然适应多核的可靠计算环境，这不能不说是一种历史的机缘。

另一个可能更加迫切的变革，就是云计算。Google的实践表明，用廉价服务器组成的服务器集群，在计算能力、可靠性等方面能够达到价格昂贵的大型计算机的水准，毫无疑问，这是大型、超大型网站和网络应用梦寐以求的境界。然而，要到达这个境界并不容易。目前一般的网站为了达成较好的可延展性和运行效率，需要聘请有经验的架构师和系统管理人员，手工配置网络服务

端架构，并且常备一个高水准的系统运维部门，随时准备处理各种意外情况。可以说，虽然大多数Web企业只不过是想在这些基础设施上运行应用而已，但仅仅为了让基础设施正常运转，企业就必须投入巨大的资源和精力。现在甚至可以说，这方面的能力成了大型和超大型网站的核心竞争力。这与操作系统成熟之前人们自己动手设置硬件并且编写驱动程序的情形类似——做应用的人要精通底层细节。这种格局的不合理性一望便知，而解决的思路也是一目了然——建立网络服务端计算的操作系统，也就是类似Google已经建立起来的“云计算”那样的平台。所谓“云计算”，指的是结果，而当前的关键不是这个结果，而是作为手段的“计算云”。计算云实际上就是控制大型网络服务器集群计算资源的操作系统，它不但可以自动将计算任务并行化，充分调动大型服务器集群的计算能力，而且还可以自动应对大多数系统故障，实现高水平的自主管理。计算云技术是网络计算时代的操作系统，是绝对的核心技术，也正因此，很多赫赫有名的中外大型IT企业都在不惜投入巨资研发计算云。包括我在内的很多人都相信，云计算将不仅从根本上改变我们的计算环境，而且将从根本上改变IT产业的盈利模式，是真正几十年一遇的重大变革，对于一些企业和技术人员来说是重大的历史机遇。恰恰在这个主题上，Erlang又具有先天的优势，这当然也是归结于其与生俱来的并行计算能力，使得开发计算云系统对于Erlang来说格外轻松容易。现在Erlang社区已经开发了一些在实践中被证明非常有效的云计算系统，学习Erlang和这些系统是迅速进入这个领域并且提高水平的捷径。

由此可见，Erlang虽然目前还不是主流语言，但是有可能会在未来一段时间发挥重要的作用，因此，对于那些愿意领略技术前沿风景的“先锋派”程序员来说，了解和学习Erlang可能是非常有价值的投资。即使你未来不打算使用Erlang，也非常有可能从Erlang的设计和Erlang社区的智慧中得到启发，从而能够在其他语言的项目中更好地完成并行计算和云计算相关的设计和实现任务。再退一步说，就算只是从开启思路、全面认识计算本质和并行计算特性的角度出发，Erlang也值得了解。所以，我很希望这本书在中国程序员社区中不要遭到冷遇。

本书是由Erlang创造者Joe Armstrong亲自执笔撰写的Erlang语言权威参考书，原作以轻松引导的方式帮助读者在实践中理解Erlang的深刻设计思路，并掌握以Erlang开发并行程序的技术，在技术图书中属于难得的佳作。两位译者我都认识，他们都是技术精湛而思想深刻的“先锋派”，对Erlang有着极高的热情，因此翻译质量相当高，阅读起来流畅通顺，为此书中译本添色不少。有兴趣的读者集中一段时间按图索骥，完全有可能就此踏上理解Erlang、应用Erlang的大路。

孟岩

CSDN首席分析师兼《程序员》杂志技术主编

2008年10月

译 者 序

另辟蹊径——从容面对容错、分布、并发、多核的挑战。

作为一名程序员，随着工作经验的增长，如果足够幸运的话，终有一日，我们都将会直面大型系统的挑战。最初的手忙脚乱总是难免的，经历过最初的迷茫之后，你会惊讶地发现这是一个完全不同的“生态系统”。要在这样的环境中生存，我们的代码需要具备一些之前我们相当陌生或者闻所未闻的“生存技能”。容错、分布、负载均衡，这些词会频繁出现在搜索列表之中。经历过几轮各种方案的轮番上阵之后，我们会开始反思这一系列问题的来龙去脉，重新审视整个系统架构，寻找瓶颈所在。你可能会和我一样，最终将目光停留在那些之前被认为是无懈可击的优美代码上。开始琢磨：究竟是什么让它们在新的环境中“水土不服”，妨碍其更加有效地利用越来越膨胀的计算资源？

其实，早在多年以前硬件领域上的革命就已经开始，现在这个浪潮终于从高端应用波及常规的计算领域——多核芯片已经量产，单核芯片正在下线——这场革命正在我们的桌面上演。时代已经改变，程序员们再也不能继续稳坐家中装作什么事情也没发生，问题已经自己找上门来了。由单核CPU频率提升带来软件自动加速的时代已经终止，性能的免费大餐已经结束，“生态环境的变化”迫使代码也必须“同步进化”。锁、同步、线程、信号量，这些之前只是在教科书中顺带提及的概念，越来越多地出现在我们日常的编程中，接踵而至的各类问题也开始折磨我们的神经。死锁、竞态，越来越多的锁带来了越来越复杂的问题。

在多核CPU的系统上，程序的性能不增反降，或者暴露出隐藏的错误？

在更强的硬件平台上，程序的并发处理能力却没有得到提升，瓶颈在哪里？

在分布式计算网络中，不得不对程序结构进行重大调整，才能适应新的运行环境？

在系统的关键应用上，不得不为软件故障或者代码升级，进行代价高昂的停机维护？

……

这一系列的问题，归根结底，都是因为主流技术体系在基本模型上存在着与并发计算相冲突的设计。换句话说，问题广泛地存在于我们所写的每一行代码中。在大厦初具规模时，却猛然发现每一块砖石都不牢固，这听来很有一些耸人听闻，但这种事并不罕见。

比如：`x = x + n`，即使是这个再常见不过的语句，也暗藏着烦恼的根源（你也可以把它称做共享内存陷阱）。从机器指令的角度来思考，这个语句可能做了这么几件事（仅仅只是在概念上）。

(1) `mov ax, [bp+x]`；将寄存器ax赋值为变量x所指示的内存中的数据。

(2) `mov bx, n`；将寄存器bx赋值为n。

(3) add ax, bx；将寄存器ax加n。

(4) mov [bp+x],ax；将变量x所指示的内存赋值为寄存器ax中的数据。

理论上，这是一个原子操作，在单核CPU下情况也确实如此，但在多核CPU下，就全然不同了。如果在每个核心上都有一个正在执行上述代码的进程（也就是试图对这个代码执行并行计算），问题就出现了。这里的x是在两个进程之间共享的内存。很明显，在第(1)步到第(4)步之间，需要某种机制来保证“某一时刻”只有一个进程正在执行，否则就会破坏数据的完整性。这也就意味着，这样的代码无法充分利用多核CPU的运算能力。也罢，咱们不加速就是了，但更糟糕的是，为了保证不出错，还需要引入锁的机制来避免数据被破坏。现有的主流技术体系全都建立在共享内存的模型之上，像x = x + n这样的代码几乎无处不在，但在多核环境下，每一处这样的代码（逻辑上的或者事实上的）都需要小心地处理锁。更糟糕的是，大量的锁彼此互相影响又会导致更为复杂的问题。这迫使程序员们在实现复杂的功能之余，还要拿出极度的耐心和娴熟的技巧来处理好这些沉闷和易错的锁，且不说是否可能，但这至少也是一个极为繁重的额外负担。

Erlang为了并发而生。20多年前，它的创建者们就已经意识到了这一问题，转而选择了一条与主流语言完全不同的路（还有为数不多的另外几种语言也是如此）。它采用的是消息模型，进程之间并不共享任何数据，因而也就完全地避免了引入锁的必要。对于多核系统而言，完全无锁，也就意味着相同的代码在更多核心的CPU上会很容易具有更高的性能，而对于分布式系统，则意味着尽可能地避免了顺序瓶颈，可以把更多的机器无缝地加入到计算网络中来。甩掉了锁的桎梏，无疑是对程序员们的解放。

不仅如此，Erlang在编程模型上走得更远。它在语言级别提供了一系列的并发原语，通过这些原语，我们可以用进程＋消息的模型来建模现实世界中多人协作的场景。一个进程表示一个人，人与人之间并不存在任何共享的内存，彼此之间的协作完全通过消息（说话、打手势、做表情，等等）交互来完成。这正是我们每一个人生而知之的并发模式！软件模拟现实世界协作和交互的场景——这也就是所谓的COP（面向并发编程）思想。

在错误处理上，Erlang也有与众不同的设计决策，这使得实现“容错系统”不再遥不可及。COP假设进程难免会出错——不像其他某些语言一样假设程序不会出错。它假设程序随时可能会出错，如果发生出错的情况，则不要尝试自行处理，而是直接退出，交给更高级的进程来对这种情况进行处理。通过引入“速错”和“进程监控”的概念，我们将错误分层，并由更高层的进程来妥善处理（比如，重启进程，或重启一系列进程）。有了这样的概念作为支撑，构造“容错系统”就会变得易如反掌。在这样的系统之下，软件错误不会导致整个系统的瘫痪，发现错误也无须停机就可直接更新代码，在配置了备份硬件之后，硬件的错误也不会影响服务的正常运行。这么做的结果相当惊人，使用Erlang的电信关键产品，达到了传说中的99.9999999%可用性（即9个9的最高可用性标准）。

Erlang采用虚拟机技术实现，用它编写的程序可以不经修改直接运行在从手机到大型机几乎所有的计算平台之上。这是一项有着20多年历史的成熟技术，有着相当多的成熟库（OTP）和开源软件，这些资产使得它有极高的实用价值。Erlang本身也是开源软件，这扫清了对于语言本身生命力的疑惑。Erlang还是一个充满活力的语言，在它的社区，常常能够见到Joe Armstrong等语

言的创建者在回答问题，这一点尤其宝贵。在熟悉了Erlang的思维方法和社区之后，很多人都发出了相见恨晚的感慨。

虽说对于并发而言，Erlang确实是非常好的选择，但这么多年以来，业界对于并发预料之中的增长却一直没有真正发生。此前，这类应用更多地局限在一些相对高端的领域，而Erlang身上浓厚的电信背景，又使得第一眼看来它似乎只适用于电信行业（实际情况远非如此）。长期以来Erlang的使用群体仅局限在一个狭小的技术圈子之内，它处于“非主流语言”的边缘位置已经很久了。这种情况直到最近才有所改观，最近两年，Google的成功使得其引为核心的大规模分布式应用模式广为人知，而多核CPU进入桌面也迫使“工业主流”开始认真对待并发计算。直到此时，解决这类问题最为成熟的Erlang技术，才因为其难以忽视的优势而引起人们的广泛关注。

从历史的眼光来看，在计算机语言的荣誉堂内，上演着一代又一代程序设计语言的繁荣和更替，潮来潮往让人难以捉摸。这与其说是技术，还不如说是时尚。对于Erlang这种有些怪异的小众语言来说，是否真的会成为“下一个Java”？实难预测，而且也不重要。但是有一点已经毫无疑问，那就是“下一代语言”至少也要像Erlang一样，处理好与并发相关的一系列问题（或者做得更好）。也许将来的X++（或X#）语言在吸收了它的精髓之后，又会成为新的工业主流语言。但即便如此，先跟随本书作者开辟的小径信步浏览这些饶有趣味的问题肯定也会大有帮助。

对于想要学习Erlang的读者，虽说语言本身相当简单，但想要运用自如也有一些难度。比如，在适应COP之后会觉得非常自然，但对于有OOP背景的程序员而言，从固有的思维习惯转换到COP和FP上（主要是和自己的思维惯性较劲）需要有一个过程。此外OTP和其他Erlang社区多年积累的财富（这些好比JDK、EJB之于Java）也需要一些时间才能被充分地理解和吸收。但这些有价值的资料大多零星地散落于邮件列表和独立的文档之中，给学习造成了很多不必要的麻烦。现在好了，有了Erlang创建者Joe Armstrong为我们撰写的“官方教程”，这些问题都已迎刃而解。

一般而言，由语言的创建者亲自撰写的教程，常常都是杰作。在翻译的过程中，译者也常常会发出这种赞叹。在本书中，Joe Armstrong不仅全面地讲述了Erlang语言本身，还详细交代了这些语言特性的来龙去脉。为什么要这么设计？除了掌握语言本身之外，能有幸窥见大师精微思辨的轨迹，也是难得的机缘。书中的例子，还会将你为之惊异的那些Erlang特性一一解密。通常是从一个不起眼的小问题开始，从宏观分析到微观实现，层层深入细细道来。问题是什么？要如何建模？该怎么重构？各个版本之间的精微演化全然呈现，但这些微小的改进，最终演化出了那些让人惊喜的特性，整个过程可谓相当精彩。

本书由Erlang中文社区（erlang-china.org）组织翻译。其中，第1章到第14章由金尹翻译，第15章到附录F由赵东炜翻译，全书由赵东炜统稿润色和审校。

由于时间仓促，加之译者水平有限，译文难免会有不足之处，欢迎读者批评指正。

赵东炜

2008年3月于北京

译者介绍

赵东炜（Jackyz）

独立软件顾问，一直专注于Web应用开发，曾负责设计和维护某大型门户网站的多个核心应用，对高并发大容量的分布式应用领域有独到见解。曾担任过软件开发工程师、系统架构师、技术经理、产品经理、创业者等多种不同的角色。闲暇时以思考技术问题为乐，从事软件行业10余年来，从最初的ASP/PHP到之后的Java/.NET以及现在的Ajax和Erlang，一直都活跃在技术的最前沿。2006年作为主要译者参与了*Ajax in Action*（中译本《Ajax实战》，由人民邮电出版社出版）的翻译工作。之后为Erlang强大的并发能力所吸引，是国内学习和传播Erlang技术的第一批人，迄今已有2年多的实际开发经验。在2007年3月创建了Erlang中文社区（erlang-china.org），现在是国内Erlang爱好者聚集和分享资料的主要网站。

金尹（TrustNo1）

金尹，长期从事电信行业的大规模语音通信程序的研发，有丰富的并发/分布式网络系统的开发经验。业余从事于数学与编程语言理论，以及并行计算方面的研究。致力于在国内推广函数式语言的发展，分别在2001年和2006年在《程序员》杂志上介绍Python、Erlang等前卫的编程理念。

目　录

第1章

引　言

哦，不！别再来一种编程语言了！我一定要再学另外一种吗？现在的这些难道还不够吗？我能理解你的反应。程序语言浩若烟海，再学一种理由何在？

学习Erlang的理由，可列出如下5条。

- 希望编写能在多核计算机上运行更快的程序。
- 希望编写不停机即可修改的可容错性程序。
- 希望尝试传闻中的“函数式语言”是否切实可行。
- 希望使用一种语言，它既在大规模工业产品中经过实战检验，又不乏优秀的类库与活跃的社区。
- 不希望在冗长烦琐的代码中耗费时间。

我们能如愿以偿吗？在20.3节中，我们会看到一些能在32核计算机上以线性增速运行的程序。在第18章中，我们将会关注如何构造可以历经数年全天候运行的高可靠性系统。在16.1节中，我们还将讨论编写服务器程序的技术，这些服务器可以在不停机的情况下更新软件。

很多时候，我们会不断地夸耀函数式语言的各种长处。函数式语言禁止代码具有“副作用”，副作用与并发水火不容。你要么编写有副作用的顺序代码，要么编写无副作用的并发代码。在这两者之间你必须选择，没有中间情况。

Erlang的并发特性源自语言本身而非操作系统。它把现实世界模拟成一系列的进程，其间仅靠交换消息进行互动，由此Erlang简化了并行编程。在Erlang世界中，存在并行进程但是没有锁，没有同步方法，也不存在共享内存污染的可能，因为Erlang根本没有共享内存。

Erlang程序可以由几百万个超轻量级的进程组成。这些进程可以运行在单处理器、多核处理器或者处理器网络上。

1.1　路线图

- 第2章让你能对Erlang快速起步。
- 第3章是有关顺序型编程的第一章。它介绍了模式匹配和非破坏性赋值的概念。
- 第4章是异常处理。程序总免不了出错。该章讲述了Erlang顺序型编程中的错误侦测和处理。
- 第5章是有关顺序型编程的第二章。它从一些高级的主题开始，涵盖了顺序型编程的其他

所有细节。

- 第6章主要讲述了编译和运行程序的几种不同方法。
- 第7章开始新话题。这是一个非技术型章节，主要讨论两个问题："这种编程方式背后的思想是什么"以及"如何用Erlang的视角来看待世界"。
- 第8章主要讲述并发性。如何创建Erlang中的并行进程？如何处理进程间通信？创建一个并行进程有多快？

2

- 第9章讨论了并行程序中的错误。当一个进程错误退出时会发生什么？如何检测进程错误，又该如何处理？
- 第10章开始讲述分布式编程。在这一章我们会写几个小型分布式程序，来展示如何在Erlang集群节点或者使用基于套接字分布模型的独立主机上运行它们。
- 第11章是一个纯应用的章节。我们会把并发和基于套接字分布的主题与我们第一个不平凡的应用紧密联系起来，构造一个迷你的类IRC客户/服务器程序。
- 第12章介绍如何将Erlang与外部程序语言衔接起来。
- 第13章给出了几个与文件编程有关的样例。
- 第14章展现如何使用套接字进行编程。我们会看到如何在Erlang中构建顺序和并行的服务器。该章的最后给出第2个颇具规模的应用：SHOUTcast服务器。这是一个流媒体服务器，它可以使用SHOUTcast协议来流化MP3数据。
- 第15章描述了`ets`和`dets`两个底层模块。`ets`模块主要用于快速、可更改的、内存散列表操作，`dets`则专门针对低级磁盘存储。
- 第16章专门介绍OTP。OTP是一批Erlang库和操作过程的总称，用来构建具有工业规模的Erlang应用。该章介绍了行为（behavior，一种OTP的核心概念）思想。通过行为，我们可以着眼于模块的功能性部分，而让框架去处理问题的非功能性部分。比如说，框架关心应用的容错和可伸缩性，而行为的回调函数则集中处理问题的细节。该章首先概要论述了如何构建你自己的行为，然后进一步地深入描述作为Erlang标准库一部分的`gen_server`行为。

3

- 第17章讨论了Erlang数据库管理系统（DBMS）Mnesia。Mnesia是一个内建的集成数据库管理系统，它极为快速且拥有软实时的响应速度。可以通过配置来让它在数个分散的物理节点上复制数据，从而提供容错功能。
- 第18章再次讨论OTP。它涵盖了拼接OTP应用所涉及的各个实践性层面。实际应用程序中充斥了大量零散的细节，它们必须以一种互相协调的方式来启动和关闭。如果它们自身或者它们的子模块崩溃，就必须重新启动。我们需要依靠错误日志来找出在崩溃前后到底发生了什么。这一章讲述了打造一个真正成熟的OTP应用程序所需的实质性细节。
- 第19章简要介绍了为什么说Erlang是为多核计算机编程量身定做的。我们将概略地讨论共享内存并发和消息传递并发，并探究为何我们坚信无可变状态且支持并发的语言是契合多核编程的理想选择。
- 第20章讲述多核编程。我们会讨论确保在多核计算机上使得Erlang程序高效运行的各种技

术。为在多核计算机上加速顺序型程序，我们会引入一系列与之相关的抽象原则。最后我们会对优化的效果执行一些测量，然后开发第3个主要程序：一个全文检索引擎。为此我们会首先实现mapreduce函数，这是一个高阶函数，可以对一群处理元素进行并行化计算。

- 附录A讲述了用于撰写Erlang函数文档的类型系统。
- 附录B讲述了如何在Windows系统上建立Erlang（以及如何在所有操作系统上配置Emacs）。
- 附录C列出了Erlang资源列表。
- 附录D讲述了lib_chan，一个用于编写基于套接字分布的库。
- 附录E关注于代码的分析、优化、调试和跟踪。
- 附录F简要地介绍了Erlang标准库中最常用的模块。

1.2 正式起航

从前，一名程序员偶然读到了一本古怪的程序语言图书。相等其实不是相等，变量实际上不能改变，它的语法一切都那么陌生。更糟糕的是，它甚至都不是面向对象的。这些程序，呃，实在太另类了……

另类的不仅仅是程序，编程的教学步骤也特立独行。它的作者一直喋喋不休地教授并发、分布和容错，不断地唠叨着一种叫做COP（Concurrency Oriented Programming，面向并发编程）的方法——管它叫什么呢。

不过有些例子看上去很好玩。那天夜里，这个程序员注视着那个聊天程序的小例子。它是多么的小巧可爱而又通俗易懂，只是那些语法有那么一丁点儿古怪，但它确实简单到不能再简单了。

开始编程并不困难，用不了几行代码，文件共享和加密通信便跃然纸上。于是这个程序员开始敲起了他的键盘……

这是一本什么样的书

这是一本讲述并发的书，也是讲述分布式的书，既是一本讲述容错的书，也是讲述函数编程的书。这本书会帮助你在消息传递的基础上构建分布式的并发系统，免去锁与互斥技术的羁绊。它会让你的程序在多核CPU上风驰电掣，亦会展示分布式互动程序的所有设计蓝图。这本书传授的各种设计方法和行为将成为设计容错与分布式系统的利器。它所蕴含的模型化并发思想，及从模型映射到代码的过程，则被称为面向并发编程。

我独乐撰此书，亦望众乐读此书。

幸望诸君，学而时习，乐其乐也。

1.3 致谢

在撰写本书时，很多人提供了帮助，我想在这里感谢所有人。

首先感谢的是我的编辑Dave Thomas。Dave一直以来都在教导我写作，用没完没了的问题来

轰炸我。为什么这样？为什么那样？我开始写书时，Dave认为我的写作风格近似于“站在石头上布道”。他说：“我希望你是在和读者交流，而不是布道。”这本书在这方面大有改观。谢谢Dave。

接下来感谢的是我身后的一群语言专家组成的智囊团。他们协助我决定内容的取舍，帮我澄清一些难以解释的小问题。在这里一并谢谢他们（排名不分先后）：Björn Gustavsson、Robert Virding、Kostis Sagonas、Kenneth Lundin、Richard Carlsson和 Ulf Wiger。

感谢Claes Vikström，他提供了关于Mnesia很有价值的建议，感谢Rickard Green在SMP Erlang上的帮忙，也为在全文检索中使用的词干分析算法感谢Hans Nilsson。

Sean Hinde和Ulf Wiger帮我弄懂了如何使用不同的OTP内部构件，Serge Aleynikov向我解释活动套接字，我才能清楚地了解它们。

Helen Taylor（我的妻子）帮我校对了好几章。还要感谢她那数百杯香茶，它们总是在我最需要的时候出现在我的手边。更重要的是，她忍受了我长达7个月之久的忘我工作。感谢Thomas和Claire，感谢Bach、Handel、Zorro、Daisy和Doris（五只可爱的小家伙），抚摸它们的时候为我欢叫，帮我保持清醒，为我找到正确的方向。

最后，要感谢所有填写勘误表的初稿阅读者。我对他们既恨又爱。当第一版草稿释出时，我没有料到他们可以在两天内读完，他们的批注把每一页改得体无完肤。但是这个过程最终催生了一本比我预想中还要优秀的书。当一堆人对我说“我看不懂这页”时（发生过不止一次），我就被迫回头审视他们关心的问题然后推倒重来。感谢所有人给予的帮助。

Joe Armstrong
2007年5月

第2章

入　门

2.1 概览

同任何其他学习经历一样，在掌握Erlang的过程中你也将会经历若干个阶段。下面就来看看，在这本书里会经历哪些阶段，而在这些阶段之中我们又能学到些什么。

2.1.1 阶段 1：茫然无绪

作为一个初学者，你能学到如何启动系统，在shell里运行命令，编译简单的程序，然后逐渐地熟悉Erlang（Erlang是一门小巧的语言，这用不了多久）。

让我们再细化一下。作为一个初学者，需要做下面这些事情。

- 确保你计算机上的Erlang系统能正常工作。
- 学习启动和关闭Erlang shell。
- 了解如何在Erlang shell里输入表达式，对其求值，并且弄懂返回结果的意思。
- 了解如何在你惯用的文本编辑器中创建和修改程序。
- 练习一下如何在Erlang shell中编译和运行程序。

2.1.2 阶段 2：初窥门径

到这个阶段你已经具备了初步的知识来运用这个语言。如果继续深入学习语言而碰到难点，正好可以运用这些背景知识去弄懂第5章。

在这个阶段中我们已经熟悉Erlang，因此可以转到更多有趣的主题。

- 你会发掘出Erlang shell更多的高级功能。与你初次接触它时所学的那些功能相比，Erlang shell可以做到的事情远远超乎我们的想象。（比如说，你可以重新调用之前录入的表达式并对它们进行编辑，这个话题包含在6.5节中。）
- 你会开始逐步学习各种库（Erlang中称为模块）。我所撰写的大多数程序中都会用到5个模块：`lists`、`io`、`file`、`dict`和`gen_tcp`，而且，在本书中我们会频繁地使用这些模块。
- 随着代码规模的逐渐膨胀，你有必要学会如何去自动地编译和执行它们。`make`是进行这些工作的必备工具。我们会学到如何运用makefile来控制编译和执行，这个话题包含在6.4节中。

- Erlang编程更高的层次是能灵活地运用一批扩展库——OTP[①]。随着Erlang使用经验的积累，你会意识到，通晓OTP将能事半功倍。毕竟，如果有人已经写好了所需的功能，又何必另起炉灶？我们会学习OTP主要的行为（behavior），特别是`gen_server`，这个话题包含在16.2节中。
- Erlang主要用来编写分布式的应用程序，这个阶段正是开始尝试的好时机。你可以从第10章的样例程序开始，而后尽情地扩展它们。

2.1.3　阶段 2.5：观其大略，不求甚解

第一次通读本书时不必苛求每一章都能学得面面俱到。

你先前可能接触过很多语言，Erlang与它们中的大部分都不相同，它是一种并发编程语言，这使得它可以和分布式、多核/SMP[②]编程结合得天衣无缝。大多数Erlang程序在多核或者SMP机器上会运行得更快。

用Erlang编程总是与一种编程范式形影不离，我称之为COP（面向并发编程）。

使用COP，你可以分解问题，识别出它本身（天然）的并发模式。这正是面向并发编程实质性的第一步。

2.1.4　阶段 3：运用自如

至此，你已然可以灵活地运用这一语言，编写一些有用的分布式程序。但是要达到游刃有余的境界，你还需要学习更多的知识。

- Mnesia。Erlang的发行版包含一个内置的、完整的、快速的、可复制的数据库——Mnesia。它原本用于那些性能和容错都至关重要的电信应用，现在也广泛地用于很多非电信场合。
- 如何和其他语言编写的代码进行接口，如何使用内联的驱动，这些内容包含在12.4节中。
- 利用各种OTP行为（behavior）来构建监控树（supervision tree）和启动脚本（start script）等，这些内容包含在第18章中。
- 如何在多核计算机上运行和优化你的程序，这些内容包含在第20章中。

2.1.5　重中之重

在通读此书时，有一条原则你必须谨记于心：编程乐在其中。就我个人看来，与传统的顺序型应用程序相比，编写诸如聊天、即时消息这样的分布式程序有着更多的乐趣。如果说在一台计算机上你能做的事情往往会被其能力限制，那么，构建于网络之上的计算机集群则是海阔凭鱼跃天高任鸟飞的另一番天地。Erlang提供了一个非常理想的环境，在它之中你可以尝试网络化的应用程序，构建产品级的系统。

为了让你能在这些领域里大展拳脚，我在那些技术性的章节里揉合了一些真实世界中的应

① Open Telecom Platform（开放电信平台）。

② Symmetric multiprocessing（对称式多处理器）。

用。毫无疑问，你可以将这些应用作为进一步实践的基石。接受并修改它们，把它们应用到超乎我所能想象的领域，而我将会为此感到万分欣慰。 9

2.2 Erlang 安装

在动手做任何事之前，你得先确保系统上装有一个可运行的Erlang版本。转到命令提示符界面，输入erl：

```
$ erl
Erlang (BEAM) emulator version 5.5.2 [source] ... [kernel-poll:false]

Eshell V5.5.2  (abort with ^G)
1>
```

在Windows系统中，必须先安装Erlang，并修改环境变量PATH以包含Erlang所在的目录，之后，erl命令才能正确执行。如果你是按标准方式安装的程序，还可以通过菜单开始→所有程序→Erlang OTP来启动Erlang。在附录B中，我会谈到如何配置Erlang以便让它与MinGW和MSYS一起运行。

说明 我只会偶尔展示(“Erlang (BEAM) ... (abort with ^G)”)这个提示。这个提示只在你需要提交一个Erlang的bug时有用。我在这里展示它仅仅是为了让你安心，不会因为你看到它却不理解而感到担心。在大多数例子中它们都不再出现，除非是某些特定的情况。

如果看到上述的shell提示，那就表明你已成功地安装了Erlang。在提示符>后面，按下Ctrl+G，输入字母Q，然后回车[①]就可退出shell。你现在可以跳过下面的内容直接阅读2.3节。

如若不然，你得到了erl是一个未知命令的错误信息，就需要先在你的机器上安装好Erlang。这通常意味着你需要做一个决定——是用一个预先做好的二进制发布版呢？还是一个打包好的发布版（仅支持OS X）？或者是从源代码中编译Erlang？此外还可以使用CEAN（Comprehensive Erlang Archive Network，Erlang综合档案网络）。

2.2.1 二进制发布版

只有Windows和基于Linux的操作系统才可以使用Erlang的二进制发布版。二进制发布版的具体安装步骤要视不同系统而定。因此，我们会逐个将这些系统上的安装方法介绍一遍。 10

1. Windows

你可以从http://www.erlang.org/download.html中找到各个版本的下载列表。选择最新版本的条目，单击Windows二进制文件的链接，它指向你下载的Windows可执行文件。单击链接，按照提示执行安装步骤。它是标准的Windows安装程序，因此你无须担心任何问题。

2. Linux

只有基于Debian的系统才有相应的二进制包。在基于Debian的系统中，输入如下命令：

① 也可在shell中输入命令q()。

```
> apt-get install erlang
```

3．在Mac OS X上的安装

Mac用户可以使用MacPorts系统来安装Erlang的预打包版本，也可以通过编译源代码来进行安装。相比之下，使用MacPorts极为方便，而且它可以一直保持软件更新。但MacPorts的Erlang发布总是要落后几版。在本书撰写最初的一段时间里，MacPorts的Erlang落后当时的官方发布大约两个版本。这种情况下，我建议你还是咬咬牙，按照下一节的描述，啃下源代码编译这块硬骨头。为了完成这项工作，你需要确保机器上已经装好了开发工具（它们一般都在随机附带的软件DVD上）。

2.2.2　从源代码创建 Erlang

除了二进制安装外，另外一个方法就是从源代码来编译Erlang。Windows上每一个新的发布版都附带了完整的二进制代码和源代码，所以对于Windows来说，这种方式没有什么特别的好处。但对于Mac和Linux平台就不同了，在新的Erlang发布版和可用的二进制安装包之间总存在着一些延时。对于所有的类Unix操作系统，从源代码安装的步骤都是相同的。

(1) 获取最新的Erlang源代码[①]。代码会打包在一个文件之中，它的名字类似`otp_src_R11B-4.tar.gz`（意思是这个文件包含了Erlang第11版的第4个维护发布版）。

11

(2) 按照下面的步骤进行解包、配置、编译以及安装：

```
$ tar -xzf otp_src_R11B-4.tar.gz
$ cd otp_src_R11B-4
$ ./configure
$ make
$ sudo make install
```

说明　在编译系统之前，你可以使用使用命令`./configure --help`来查看可用的配置选项。

2.2.3　使用 CEAN

CEAN试图将所有主流的Erlang应用集中起来统一在一个通用安装界面上。CEAN的特点在于不仅可以管理基本的Erlang系统，还能管理大量使用Erlang编写的程序包。这也就意味着你自己编写的程序包也能像基本的Erlang系统那样得到持续的更新与维护。

CEAN针对不同的操作系统和处理器架构预编译了各种不同的二进制版本。想通过CEAN来安装系统，那么就前往http://cean.process-one.net/download/，按照上面指示的步骤安装［注意，有一些读者反映CEAN可能不会安装Erlang编译器。如果你恰好碰上了这个问题，那么，启动Erlang shell并输入命令`cean:install(compiler)`。这样就能装好编译器了］。

2.3　本书代码

我们所展示的大多数代码片段都来自完整的、可运行的样例程序。这些程序你也能通过网络

① 可从http://www.erlang.org/download.html下载。

下载[①]。为便于查找，如果本书的代码清单也包含在下载文件中，那么在该代码片段上会有一个条目（就像下面的这个样子）：

shop1.erl

```
-module(shop1).
-export([total/1]).

total([{What, N}|T]) -> shop:cost(What) * N + total(T);
total([])            -> 0.
```

这个条目表明了代码在下载点中的路径。如果你读的是本书的PDF版，而且你的PDF阅读器也支持超链接，那么在单击这个条目之后，相关代码就会在浏览器中显示出来。

12

2.4 启动 shell

现在我们正式开始。shell是一个交互工具，我们常用它来完成与Erlang的互动。启动shell之后，我们可以输入表达式，然后shell就会返回这些表达式的值。

如果你已经安装好了Erlang（参照2.2节所述），那么Erlang shell——`erl`也就同时安装好了。要运行它，请开启一个传统的操作系统命令行界面（Windows上是`cmd`，而在类Unix的系统上则是`bash`这样的shell程序）。在命令提示符下，输入`erl`来启动Erlang shell：

```
❶ $ erl
  Erlang (BEAM) emulator version 5.5.1 [source] [async-threads:0] [hipe]

  Eshell V5.5.1  (abort with ^G)
❷ 1> % I'm going to enter some expressions in the shell ..
❸ 1> 20 + 30.
❹ 50
❺ 2>
```

让我们看看刚才做的动作。

❶ 这是在Unix命令下启动Erlang shell。shell返回了一个提示，告诉你正在运行的是哪个版本的Erlang。

❷ shell显示了提示符1>，然后我们输入了一串注释，百分号（%）表示一个注释的开始。从百分号开始到这行结束的所有文本都被看作是注释，它会被shell和Erlang编译器忽略。

❸ 由于我们并没有输入一个完整的命令，所以shell重复显示1>。在此时，我们输入表达式20+30，而后紧跟一个句点和一个回车（初学者往往会忘记输入句点。没有句点，Erlang就认为我们还没有输完整个表达式，我们也不会看到显示的结果）。

❹ shell对表达式求值，然后打印结果（这里的结果是50）。

❺ shell打印出另外一个提示符，这次显示命令数为2（因为命令数会随着每次新命令的输入而增加）。

有没有在你的系统上试着运行过shell？如果没有，那么现在就放下书去试一下。可能你会认

① 可从http://pragmaticprogrammer.com/titles/jaerlang/code.html下载。

13 为自己对于前面的内容了如指掌，但是如果一味地死读书而不去加以实践，那就会眼高手低，不能将知识融入实践。如果将编程比作运动的话，那么它绝对不是表演项目，而是竞技项目。因此，你也应该像一个勤奋的运动员那样，不断地去练习。

照着书中的样子，输入例子中的表达式，用这些例子作些试验，对它们做些修改。如果出错了，那就停下来，研究一下出错的原因。即便是经验老到的Erlang程序员也会花上大把的时间去和shell打交道。

随着经验的积累，你会发现shell其实是一个非常强大的工具。之前录入shell的命令可以用Ctrl+P和Ctrl+N来找回，也能用类似Emacs的编辑命令来编辑它们。这些话题我们会留到6.5节中继续讨论。更妙的是，当开始编写分布式的程序时，一个集群内会有许多正运行着Erlang系统的节点，你将发现可以将shell随意地附着到它们中的任何一个上。你甚至可以用安全shell（ssh）向一个运行着Erlang系统的远程计算机发起一个直接连接。通过这种方法，在Erlang的节点集群中，你能与其中的任何一个节点上的任意一个程序打交道。

警告 本书之中，也不是所有的东西都能输入shell。特别要注意的是，你不能往shell里面输入Erlang文件当中的代码。.erl文件中的句法形式不是表达式，它不能被shell所接受。shell仅仅能够对Erlang表达式求值，除此之外的其他事情，它都做不了。另外需要特别注意的是，你不能在shell中输入模块注解，这些注解以连字号开始（比如-module、-export等）。

本章剩余的部分依然会采用这种“与Erlang shell进行数次短小对话”的形式。为了不破坏行文，很多时候，我不会解释全部的细节，这些内容会在5.4节中补充说明。

2.5 简单的整数运算

先计算几个算术表达式：

```
1> 2 + 3 * 4.
14
2> (2 + 3) * 4.
20
```

要点 你会注意到这段对话以命令行数1开始（也就是说shell打印了1>）。这表明我们开启了一
14 个新的shell，如若不然，则意味着shell继续上一个样例的会话。为了正确地再现本书中的样例，你必须在看到对话从1>开始时重新启动shell，反之则不必。

shell没有任何响应

你输入命令后，如果shell不响应，那么大概就是因为你忘记了用句点加回车[或者叫做点-空行（dot-whitespace）]来结束命令。

另一种可能会出错的地方是，你用单引号或者双引号开始一段文字，但是没有用相匹配的引号去结束它。

如果上述的任何一种情况发生，最好的办法就是把遗漏的符号补上。

如果已经到了无可挽回的地步，系统不再有任何响应，那么就按Ctrl+C（Windows上是Ctrl+Break）。你会看到下面这行提示：

```
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
```

这个时候，按A就可以中止当前Erlang会话。

进阶 你可以开启关闭数个shell，参见6.7节中的详细说明。

说明 照着书中的例子依葫芦画瓢，不失为一种绝佳的学习方式。如果你正准备这么做，最好先浏览一下6.5节。

此外，Erlang遵守标准算术表达式的法则，因此，2+3*4应该是2+(3*4)而不是(2+3)*4。

Erlang采用不定长的整数来进行整数的算术演算。在Erlang中，整数运算没有误差，因此你不用担心运算溢出，也不用为了一个固定字长容纳不下一个大整数而伤脑筋。 15

变量记号

我们经常需要讨论特定变量的值，为此我会用Var ↦ Value这种记号，例如，A ↦ 42就表示变量A的值为42。如果有多个变量的话，我会这样写{A ↦ 42,B ↦ true...}，意思是A为42，B为true，以此类推。

为何不尝试一下这个特性？超大整数的计算一定会给你的朋友们留下深刻的印象。

```
3> 123456789 * 987654321 * 112233445566778899 * 998877665544332211.
13669560260321809985966198898925761696613427909935341
```

你还可以用很多种不同的方式来输入整数[①]，下面这个例子就使用了16进制和32进制记号：

```
4> 16#cafe * 32#sugar.
1577682511434
```

2.6 变量

怎样才能把一个命令的结果保存起来，以供后面使用呢？这就是变量的职责所在。下面是一个例子：

```
1> X = 123456789.
123456789
```

① 参见5.4节。

这是什么意思呢？首先，我们向变量X赋了一个值，然后shell打印出了变量的值。

说明　所有的变量都必须以大写字母开头。

如果你要查看一个变量的值，那么只要输入变量的名字即可：

```
2> X.
123456789
```

现在X有了值，你可以这样使用它：

```
3> X*X*X*X.
232305722798259244150093798251441
```

16

代数式的单一赋值

在我上初中时，数学老师就告诉过我“如果在同一个方程的不同地方都有X，那么这些X指的都是同一个东西”。解方程就靠它了，比如，我们有X+Y=10和X-Y=2，那么根据这两个方程可得：X为6，Y为4。

但是当我学习第一门程序语言时，却看到老师在黑板上写出这样的式子：

X=X+1

大家都懵了，“这是个无解的等式”。但老师却说，我们错了，我们应该忘了在数学课上学到的东西。X不是一个数学变量，它就像一个鸟笼……

而在Erlang中，变量恢复了它在数学中的涵义。当把一个变量和值关联在一起时，你其实就做出了一项断言，也就是对一个事实的陈述，这个变量的值是多少，仅此而已。

然而，你要是想尝试着给变量X赋上一个另外的值，那么系统会无情地抛给你一个错误消息。

```
4> X = 1234.
=ERROR REPORT==== 11-Sep-2006::20:32:49 ===
Error in process <0.31.0> with exit value:
  {{badmatch,1234},[{erl_eval,expr,3}]}

** exited: {{badmatch,1234},[{erl_eval,expr,3}]} **
```

这到底是怎么回事？嗯，要解释它，我们需要破除对于表达式X=1234的两种错误想法，如下。

- 首先，X不是一个变量，至少不是你在Java或者C当中碰到的那种变量。
- 其次，=不是一个赋值操作符。

对于Erlang新手来说，这可能是最让人犯晕的地方之一，为此，很有必要来深入地探讨一下这个问题。

2.6.1　变量不变

Erlang的变量是单一赋值变量。恰如其名，单一赋值变量的值只能一次性地给定。一个变量一旦被赋了值，你想再次改变它，就会得到一个错误（实际上，我们刚才得到的是一个匹配失败的错误）。一个变量如果含有一个被赋予的值，就称为绑定变量，否则，则被称作自由变量。一

17

开始，所有的变量都是自由的。

当Erlang遇到语句X=1234时，它就将值1234绑定到X。而在被绑定之前，X可以接受任何值，它就像是一个需要被填满的空洞。但是，一旦它得到了某个值，那么它就永远保持这个值。

讲到这儿，你有可能会问，既然如此，使用变量这个术语的意义又在哪里。主要有两个原因。

- 它们的确也是变量，只不过它们的值只能改变一次（也就是，它们从自由变量变成了绑定变量）。
- 它们看上去与传统编程语言中的变量很相似，因此，当遇到一行这样的代码：

  X=...

  我们心里就会开始想，“嗯，这个我知道，X是一个变量，=是一个赋值操作符”。实际上，我们的这些想法也没什么大错，X近似于一个变量，=也近似于一个赋值操作符。

  说明，出现在Erlang代码之中的省略号（...）意味着“此处省略掉了一些代码”。

就其本质而言，=是一个模式匹配运算符。当X是一个自由变量时，它的行为与赋值一致。

最后，定义一个变量的词法单元就是这个变量的作用域。因此，如果在一个函数语句范围内使用X，那么X的值就不能“跳出”语句之外。在同一个函数的不同子句中，彼此之间也不存在全局或者共享的私有变量。如果X出现在许多个不同的函数当中，那么这些X的值也都是各自独立的。

2.6.2 模式匹配

在大多数的语言中，=都表示赋值语句。然而，在Erlang中，=表示一个模式匹配操作。Lhs=Rhs实际上是这样一个过程，对右端求值（Rhs），然后将结果与左端（Lhs）进行模式匹配。

一个变量，比如X，就是一种最简单的模式。如前所述，变量只能被赋值一次。当我们第一次输入X=SomeExpression时，Erlang会问自己，“要怎么做才能让这个语句的值变为true？[①]”由于X没有被赋值，因此可以把SomeExpression的结果绑定到X上，同时也使得语句有效，于是皆大欢喜。

18

但是，如果随后又输入X=AnotherExpression，那么只有当SomeExpression与AnotherExpression一致时这个语句才会成立。下面是一个例子：

```
Line 1   1> X = (2+4).
         6
         2> Y = 10.
         10
5        3> X = 6.
         6
         4> X = Y.
         =ERROR REPORT==== 27-Oct-2006::17:25:25 ===
         Error in process <0.32.0> with exit value:
10          {{badmatch,10},[{erl_eval,expr,3}]}
```

① Erlang的每一个语句都会有值，请参看2.8节相关部分。——译者注

```
5> Y = 10.
10
6> Y = 4.
=ERROR REPORT==== 27-Oct-2006::17:25:46 ===
Error in process <0.37.0> with exit value:
  {{badmatch,4},[{erl_eval,expr,3}]}
7> Y = X.
=ERROR REPORT==== 27-Oct-2006::17:25:57 ===
Error in process <0.40.0> with exit value:
  {{badmatch,6},[{erl_eval,expr,3}]}
```

在上面这个例子中，在第1行，系统对表达式2+4求值，得到答案6。在第1行之后，shell就维护了这样一张绑定表{X ↦ 6}。在第3行被求值后，绑定表就变成这样：{X ↦ 6，Y ↦ 10}。

现在到了第5行。由于之前的运算，现在有X ↦ 6，于是X=6这个匹配成立。

但当运行到第7行X=Y时，现在的绑定表是{X ↦ 6，Y ↦ 10}，因此匹配失败，然后打印了一个错误消息。

第4～7行的表达式成功与失败都取决于X与Y的值。在继续下面的话题之前，你应该反复地温习这些知识，以确保真的弄懂了它们。

到目前为止，你可能会觉得我在模式匹配上所着的笔墨似乎有点多。这是因为目前所遇到的情况，在等号的左边，无论是绑定的还是自由的，这些模式都只是变量。但在后续的内容中，我们会看到，可以让“=”去匹配任意复杂的模式。在回到这一主题之前，我们还会介绍元组（tuple）和列表（list），它们通常都用于存储复合数据。

19

2.6.3　单一赋值为何有益于编写质量更高的代码

Erlang里面的变量仅是对值的一个引用，就具体实现而言，一个绑定变量就是一个指针，这个指针指向存放那个值的存储区。而那个值是无法改变的。

不能改变一个变量的值是极为重要的事实，因为这与C、Java这样的命令式语言中的变量是不同的。

现在来看看，如果允许改变变量，又会发生什么情况。先定义一个变量X：

```
1> X = 23.
23
```

现在用X进行运算：

```
2> Y = 4 * X + 3.
95
```

假定我们可以修改X的值（好可怕）：

```
3> X = 19.
```

幸运的是，Erlang不许这么做，shell会神经质似地报告：

```
=ERROR REPORT==== 27-Oct-2006::13:36:24 ===
Error in process <0.31.0> with exit value:
  {{badmatch,19},[{erl_eval,expr,3}]}
```

也就是说我们已经给X赋了23，它就不能是19。

但是，假设我们可以这么做，那么Y的当前值就是不正确的，我们就不能把语句2看作是一个等式。此外，如果在程序中不同的地方允许X多次改变自己的值，那么一旦某些部分出错了，我们会很难确定具体是哪个X值引起的，也就是说，会很难精确找到出错误语句。

在Erlang中，变量一旦被赋值就不能再改变的特性还可以简化调试。要知其所以然，我们就要想一想，错误是什么？怎么才能发现错误？

在代码错误的排查中相当常见的一种错误就是，变量被赋了一个错误的值。在这种情况下，你需要找出程序从哪儿获得的错误值。如果变量在程序的不同地方多次修改了值，那么要找出哪些修改是错误的是非常困难的。

20

抛弃“副作用”意味着我们的程序可以并行化

用术语来说，我们把可修改的内存区域称为可变状态（mutable state）。Erlang是一个函数式语言，不存在可变状态。

本书的后续章节，将会关注于如何在多核CPU上编写程序。当多核编程来临的时候，采用不可变状态所带来的好处是难以估量的。

如果你用C、Java这样的传统编程语言为多核CPU编写程序，就不得不应付共享内存带来的问题。要想不破坏共享内存，就必须在访问时对其加锁。程序还要保证在操纵共享内存时不会崩溃。

而Erlang没有可变状态，也就没有共享内存，更没有锁，这一切都有利于并行化程序的编写。

在Erlang中就不存在这个问题，变量赋值一次之后永不再变。一旦发现某个变量出错，我们就能立刻确定程序之中绑定这个变量的代码，它就是错误产生之处。

那么，你可能又会问，没有了变量该怎样去编程？在Erlang中要如何去描述X=X+1这样的表达式呢？答案也很简单，声明一个新变量，它的名字之前没有被用过，比如说X1，然后写X1=X+1。

2.7 浮点数

让我们用浮点数做一些运算：

```
1> 5/3.
1.66667
2> 4/2.
2.00000
3> 5 div 3.
1
4> 5 rem 3.
2
5> 4 div 2.
2
6> Pi = 3.14159.
3.14159
7> R = 5.
5
```

21

```
8> Pi * R * R.
78.5397
```

这里有件事情不要混淆，第1行的末尾是整数3。句点代表表达式的结束而不是小数点。如果要表示一个浮点数，那么可以写3.0。

“/”永远返回浮点数，因此，4/2计算结果（在shell中）就是2.0000。N div M和N rem M是用于整数除和取余数，因此，5 div 3是1，5 rem 3是2。

浮点数必须含有小数点且小数点后至少有一位十进制数。当你用“/”来除两个整数的时候，其结果会自动转换为浮点数。

2.8　原子

在Erlang中，原子用来表示不同的非数字常量值。

如果以前使用过C或Java中的枚举类型，不管是否意识到了，你都是在使用和原子非常类似的东西。

使用符号常量来增加代码的可读性，C程序员对此大多非常熟悉。一个典型的C程序会在包含文件中定义一大堆的全局常量，比如文件glob.h可能包括这些：

```
#define OP_READ 1
#define OP_WRITE 2
#define OP_SEEK 3
...
#define RET_SUCCESS 223
...
```


而典型的C代码会像下面这样使用这些符号常量：

```
#include "glob.h"
int ret;
ret = file_operation(OP_READ, buff);
if( ret == RET_SUCCESS ) { ... }
```

在C程序中，这些常量具体的值并不重要，重要的是它们的值都不相同，而且它们之间可以进行比较。

与之等价的Erlang代码会是这样：

```
Ret = file_operation(op_read, Buff),
if
    Ret == ret_success ->
       ...
```

Erlang中的原子是全局有效的，而且无需使用宏定义或者包含文件。

如果想编写一个涉及日期和星期的程序。该如何在Erlang中表示某一天呢？显然，最好使用monday、tuesday这样的一些原子。

原子是一串以小写字母开头，后跟数字字母或下划线（_）或邮件符号（@）的字符[①]。例如，red、december、cat、meters、yards、joe@somehost以及a_long_name等。

① 你可能会发现句点（.）也能在原子中使用，但这并不是一个正规的Erlang扩展。

使用单引号引起来的字符也是原子。使用这种形式，我们就能使得原子可以用大写字母作为开头或者包含非数字字符。例如，'Monday'、'Tuesday'、'+'、'*'、'an atom with spaces'。你还可以将原本不需要使用引号的原子引起来，'a'实际上就等同于a。

一个原子的值就是原子自身。因此，如果输入的命令只有原子，那么Erlang shell会打印那个原子的值：

```
1> hello.
hello
```

讨论原子的值或整数的值，这听上去多少还是有些奇怪。但是因为Erlang是一个函数式语言，每一个表达式都必须有值，整数和原子这些特别简单的表达式也不例外。

23

2.9 元组

你若想将一定数量的项组成单一的实体，那么就可以使用元组（tuple）。将若干个以逗号分割的值用一对花括号括起来，就形成了一个元组。例如，想要描述某个人的名字和他的身高，你可以用{joe, 1.82}。这个元组包括了一个原子和一个浮点值。

元组类似于C语言中的结构，除了元组是匿名的之外，它们之间相差无几。在C语言中要定义类型point的变量p，要这么做：

```
struct point {
  int x;
  int y;
} P;
```

在C语言的结构中，通过点操作符来访问一个结构的字段。比如，要设置Point中x、y的值，你可能会这么写：

```
P.x = 10; P.y = 45;
```

Erlang并没有类型声明，因此创建一个“point”则会是这个样子：

```
P = {10,45}
```

这条语句创建了一个元组并且将其绑定到变量P。和C语言不同的是，元组中的字段没有名字。因为这个元组只包含了两个整数，所以我们必须记住其用处。为了方便记忆，通常可以使用一个原子作为元组的第一个元素来标明这个元组所代表的含义。因此我们可以用{point, 10, 45}来代替{10,45}，这能使得程序更为清晰易懂。[①]

元组可以嵌套。你若想表达一个人信息的某些方面,如他们的名字、高度、鞋码和眼睛的颜色，我们可以这么做：

```
1> Person = {person,
             {name, joe},
             {height, 1.82},
             {footsize, 42},
             {eyecolour, brown}}.
```

24

① 这种标记元组的方法并非语言的规定，但确实是一个值得推荐的编程风格。

注意观察我们是如何使用原子作为元组的标签，同时给字段赋值（在name、和eyecolour这些地方）。

2.9.1　创建元组

在声明元组时，就自动创建了元组，不再使用它们时，元组也随之销毁。Erlang使用垃圾搜集器去收回没有使用的内存，因此我们不用担心内存分配的问题。

如果你创建的一个新元组引用了一个已绑定的变量，那么新元组就会享有这个变量所引用的数据结构。下面就是一个例子：

```
2> F = {firstName, joe}.
{firstName,joe}
3> L = {lastName, armstrong}.
{lastName,armstrong}
4> P = {person, F, L}.
{person,{firstName,joe},{lastName,armstrong}}
```

而若在创建数据结构时试图引用一个未定义的变量，系统就会给出一个错误。比如，下面这一行，尝试使用未定义的变量Q会得到一个错误消息。

```
5> {true, Q, 23, Costs}.
** 1: variable 'Q' is unbound **
```

这就意味着变量Q未被定义。

2.9.2　从元组中提取字段值

正如前文所述，=看似赋值语句，实乃模式匹配操作符。呵呵，这么啰嗦，你大概要开始嘀咕为什么我们这么迂腐？这么说吧，模式匹配作为Erlang的基础，用来完成很多不同的任务：可以用它从数据结构中提取字段值，在函数中进行流程控制，或者当你向一个进程发送消息时，从并行程序中筛选那些需要处理的消息。

当想从元组中提取一些字段值时，就会用到模式匹配操作符=。

让我们先回到用元组表示点的例子：

```
1> Point = {point, 10, 45}.
{point, 10, 45}.
```


若想从Point中提取字段然后存放于X、Y两个变量，可以这么做：

```
2> {point, X, Y} = Point.
{point,10,45}
3> X.
10
4> Y.
45
```

在命令2中，X被绑定到10，Y被绑定到45。这里Lhs=Rhs表达式所定义的值是Rhs，因此shell打印{point,10,45}。

如你所见，位于等号两边的元组必须含有相同数量的元素，两边相对应的元素必须绑定相同

的值。

现在如果输入这样的命令：

```
5> {point, C, C} = Point.
=ERROR REPORT==== 28-Oct-2006::17:17:00 ===
Error in process <0.32.0> with exit value:
{{badmatch,{point,10,45}},[{erl_eval,expr,3}]}
```

会发生什么呢？模式{point, C, C}与{point, 10, 45}不能匹配，因为C不可能同时等于10和45。因此，模式匹配失败了，[①]系统会打印一个错误消息。

如果你有一个复杂的元组，那么可以使用相同结构的模式来提取所要的字段值，并且只要在需要提取的字段位置上使用未绑定变量。[②]

作为演示，首先要定义一个含有复杂数据结构的Person变量：

```
1> Person={person,{name,{first,joe},{last,armstrong}},{footsize,42}}.
{person,{name,{first,joe},{last,armstrong}},{footsize,42}}
```

现在，编写一个模式去提取这个人的姓：

```
2> {_,{_,{_,Who},_},_} = Person.
{person,{name,{first,joe},{last,armstrong}},{footsize,42}}
```

26

最后打印出Who的值：

```
3> Who.
joe
```

注意 在前面的样例中，将_作为占位符，表示那些我们不关心的变量。符号_称为匿名变量，与常规变量不同，在同一个模式中的不同地方，各个_所绑定的值不必相同。

2.10 列表

我们用列表存储数目可变的东西，如在商店里所买到的商品、行星的名字、从因式分解函数中返回的素数，等等。

将若干个以逗号分割的值用一对方括号括起来，就形成了一个列表。下面的例子就演示了如何创建一个购物清单：

```
1> ThingsToBuy =  [{apples,10},{pears,6},{milk,3}].
[{apples,10},{pears,6},{milk,3}]
```

列表之中的各个元素可以有各自不同的类型，比如，可以这样写：

```
2> [1+7,hello,2-2,{cost, apple, 30-20},3].
[8,hello,0,{cost,apple,10},3]
```

① 致熟悉Prolog的读者：Erlang将匹配失败当作错误处理，也不会在匹配中回溯。

② 这种使用模式匹配来提取变量的方法称为unification，很多函数式编程语言和逻辑式编程语言都使用这种方法。

2.10.1　术语

列表的第一个元素称为列表的头（head）。那么你可以想象一下如果从列表中移除头，所剩下的东西就称为列表的尾（tail）。

例如，如果有列表[1,2,3,4,5]，那么列表的头就是整数1，它的尾为[2,3,4,5]。注意，列表的头可以是任何东西，但是列表的尾通常还是一个列表。

访问列表的头是一个非常高效的操作，因此，实际上所有的列表处理函数都是从提取列表头
27 开始的，先对头进行处理，然后继续处理列表的尾。

2.10.2　定义列表

如果T是一个列表，那么[H|T]也是一个列表[①]，这个列表以H为头，以T为尾。竖线符号（|）可以将列表的头和尾分隔开来，而[]则是空列表。

无论何时，当我们用[...|T]来构造一个列表时，都应该保证T是一个列表。如果T是一个列表，那么新的列表就是“正规形式”的，反之，新列表就被称为“非正规列表”。大多数的库函数都假定列表是正规的，它们不能正确地处理非正规列表。

可以用[E1,E2,..,En|T]这种形式向T的起始处加入多个新元素。例如：

```
3> ThingsToBuy1 = [{oranges,4},{newspaper,1}|ThingsToBuy].
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

2.10.3　从列表中提取元素

我们可以用模式匹配操作从一个列表中提取元素。假定现在有一个非空的列表L，那么表达式[X|Y]=L（这里的X、Y都是自由变量），可以把列表的头提取到X，将列表的尾提取到Y。

如果我们在商店，并且有一个购物清单ThingsToBuy1，首先要做的是把列表分解成头和尾：

```
4> [Buy1|ThingsToBuy2] = ThingsToBuy1.
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

这个匹配的结果是：

```
Buy1|->{oranges,4}
```

和

28

```
ThingsToBuy2|->[{newspaper,1}, {apples,10}, {pears,6}, {milk,3}].
```

我们根据Buy1去买橘子，然后再来继续提取下面两个元素：

```
5> [Buy2,Buy3|ThingsToBuy3] = ThingsToBuy2.
{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

得到了Buy2 ↦ {newspaper,1}，Buy3 ↦ {apples,10}和ThingsToBuy3 ↦ [{pears,6},{milk,3}]

---

① LISP程序员注意：[H|T]其实是一个带有CAR H和CDR T的CONS单元。在模式中，这个语法可以分解成CAR和CDR。在表达式中，它构造了一个CONS单元。

2.11 字符串

严格地讲，Erlang中并没有字符串，字符串实际上就是一个整数列表。用双引号（"）将一串字符括起来就是一个字符串，比如，我们可以这样写：

```
1> Name = "Hello".
"Hello"
```

说明 在某些编程语言中，字符串既可以使用单引号也可以使用双引号。而在Erlang中，必须使用双引号。

这里的"Hello"仅仅是一个速记形式，实际上它意味着一个整数列表，列表中每一个元素都是相应字符的整数值。

shell打印一串列表值时，只有列表中的所有整数都是可打印字符，它才把这个列表当作字符串来打印：

```
2> [1,2,3].
[1,2,3]
3> [83,117,114,112,114,105,115,101].
"Surprise"
4> [1,83,117,114,112,114,105,115,101].
[1,83,117,114,112,114,105,115,101].
```

表达式2的列表[1，2，3]被原封不动地打印出来，这是因为1、2、3并不是可打印字符。

表达式3的列表中，所有的项都是可打印字符，因此列表就打印成字符串的形式。

表达式4与表达式3大体相同，但列表的开始元素为1，并非可打印字符，因此这个列表被原样打印。

我们无须死记硬背哪一个整数表示哪一个特定的字符（ASCII码表），可以使用$符号来表示字符的整数值。例如，$a实际上是一个整数，表示字符a。比如说：

```
5> I = $s.
115
6> [I-32,$u,$r,$p,$r,$i,$s,$e].
"Surprise"
```

字符串中使用的字符集

字符串中的字符是Latin-1（ISO-8859-1）编码的字符。例如，一个含有瑞典名字的字符串Håkan会被编码成[72,229,107,97,110]。

29

说明 如果在shell中将[72,229,107,97,110]作为表达式输入，你可能不会看到想要的结果：

```
1> [72,229,107,97,110].
"H\345kan"
```

“Håkan” 哪里出错了？怎么面目全非了？这实际上是显示终端的字符集和区域设定有问

题，在这类问题上Erlang束手无策。

Erlang所关心的，只是以某种编码方式编码的一串整数值列表。如果碰巧是在Latin-1编码下，那么它们应该可以正确显示（如果终端显示设定无误的话）。

2.12　再论模式匹配

在本章接近尾声的时候，我们要再一次回到模式匹配的话题。

表2-1列出了一些模式和它们所对应的值。[①]在表2-1第3列结果栏中的内容说明了这个模式是否匹配对应的例子，如果是，那么显示变量的绑定情形。逐行研究这些样例，确保真正明白了它们的含义。

表 2-1

模　式	值	结　果
{X,abc}	{123,abc}	成功，X↦123
{X,Y,Z}	{222,def,"cat"}	成功，X↦222，Y↦def，Z↦"cat"
{X,Y}	{333,ghi,"cat"}	失败，元组结构不同
X	true	成功，X↦true
{X,Y,X}	{{abc,12},42,{abc,12}}	成功，X↦{abc,12}，Y↦42
{X,Y,X}	{{abc,12},42,true}	失败，X不能同时为{abc,12}和true
[H|T]	[1,2,3,4,5]	成功，H↦1，T↦ [2,3,4,5]
[H|T]	"cat"	成功，H↦99，T↦"at"
[A,B,C|T]	[a,b,c,d,e,f]	成功，A↦a，B↦b，C↦c，T↦ [d,e,f]

如果你对其中的任何一点仍然心存疑惑，那么应该试着在shell中输入表达式Pattern=Term
30 来查看具体的运行结果。

例如：

```
1> {X, abc} = {123, abc}.
{123,abc}.
2> X.
123
3> f().
ok
4> {X,Y,Z} = {222,def,"cat"}.
{222,def,"cat"}.
5> X.
222
6> Y.
```

① 值就是Erlang的数据结构。

```
def
...
```

说明 命令f()会让shell释放它所绑定过的所有变量。执行这个命令后，所有的变量都变成自由变量，因此第4行的X与第1行和第2行的X也就没有任何关系。

现在，我们对基本的数据类型已经非常熟悉了，对单一赋值和模式匹配也有了初步的了解，因此可以加快步伐进入新的一章，学习如何定义函数和模块。

第3章 顺序型编程

本章中，我们会学到如何用Erlang来编写简单的顺序型程序。3.1节主要会讨论模块和函数。而在学习函数的过程之中，还会进一步地了解第2章中谈到的模式匹配，它们如何发挥更多的作用。

此外，我们还会继续第2章中提到过的购物清单的例子。这一次，要编写一些代码来计算购物清单中的价格总和。

随着学习的深入，我们还会循序渐进地改良之前编写的代码。经过这样的过程，就能学到如何去演化一个原始的创意，将其变成优美的代码，而不是一堆知其然而不知其所以然的死知识。通过剖析其中的每一个步骤，你将会领悟到一些可以应用到自己编程实践中的理念，这会让你受益匪浅。

再进一步，我们会讨论高阶函数（称为fun），学习如何用它创建自己的控制抽象。最后，我们还会谈及断言（guard）、记录（record）、case语句和if语句。

好了，继续上路……

3.1 模块

模块是Erlang中代码的基本单元，我们编写的所有函数都存于模块之中。模块文件通常存放在以.erl为扩展名的文件中。

要运行一个模块，首先需要编译它，编译成功之后的模块文件其扩展名是.beam。[①]

在我们动手编写第一个模块之前，先来回想一下有关模式匹配的知识。下面的内容中，我们会创建两个数据结构，分别用来表示矩形和圆。之后再来解析这些数据结构，从矩形中取边长，从圆中取半径，下面是其实现：

```
1> Rectangle = {rectangle, 10, 5}.
{rectangle, 10, 5}.
2> Circle = {circle, 2.4}.
{circle,2.40000}
3> {rectangle, Width, Ht} = Rectangle.
```

① beam是Bogdan's Erlang Abstract Machine（Bogdan的Erlang抽象机）的缩写。Bogumil（Bogdan）Hausman在1993年实现了一个Erlang编译器，并且设计了一套新的Erlang指令集。

```
{rectangle,10,5}
4> Width.
10
5> Ht.
5
6> {circle, R} = Circle.
{circle,2.40000}
7> R.
2.40000
```

第1行和第2行分别创建了矩形和圆。第3行和第6行用模式匹配分别提取了矩形和圆中的字段。在第4行、第5行和第7行，打印这些通过模式匹配获得的值。运行到第7行之后，shell中的变量绑定是这样的：{Width ↦ 10,Ht↦5,R ↦ 2.4}。

将模式匹配从shell挪到函数中只需稍加改变。首先，我们来创建一个名为area的函数，用它来计算矩形和圆的面积。我们把这个函数放在gemoetry模块中，并把这个模块存到geometry.erl文件中。下面就是这个模块的完整内容：

geometry.erl

```
-module(geometry).
-export([area/1]).
area({rectangle, Width, Ht}) -> Width * Ht;
area({circle, R})            -> 3.14159 * R * R.
```

先别理会-module和-export声明（我们稍后会再讨论它们），现在我们只关注area函数。 34

area函数由两个子句构成，子句间以分号分隔，最后一条子句的后面以句点作为结束符。每一个子句都有一个函数头和一个函数体，函数头由函数名和随后的以括号括起来的模式组成，函数体则由一系列表达式组成，[①]如果函数头中的模式与调用参数匹配成功的话，其对应的表达式就会进行运算。模式将按照它们出现在函数定义中的先后顺序进行匹配。

注意，形如{rectangle, Width, Ht}的模式是area函数定义的一部分。每个模式都明确地与一个子句相对应。下面看看area函数的第一个子句：

```
area({rectangle, Width, Ht}) -> Width * Ht;
```

这是一条计算矩形面积的规则。当我们的调用是geometry: area({rectangle, 10, 5})时，最前面那个模式被匹配，绑定变量{Width ↦ 10, Ht ↦ 5}。匹配完之后，->号之后的代码会被执行。这里是Width * Ht，即10*5，结果为50。

现在，编译并运行它：

```
1> c(geometry).
{ok,geometry}
2> geometry:area({rectangle, 10, 5}).
50
3> geometry:area({circle, 1.4}).
6.15752
```

① 参见5.4节。

上面的演示是在做什么呢？在第1行，我们输入了命令c(geometry)，编译geometry.erl文件中的源代码。编译器返回{ok,geometry}，意味着编译成功，模块geometry已经被编译并加载。第2行和第3行是在geometry模块之中调用定义的函数。注意，如何同时使用模块名和函数名以精确定位希望调用的函数。

扩展这个程序

假如现在我们想要扩展这个程序，加入对正方形这种几何对象的支持，可以这么做：

```
area({rectangle, Width, Ht}) -> Width * Ht;
area({circle, R})            -> 3.14159 * R * R;
area({square, X})            -> X * X.
```

35

或者这么写：

```
area({rectangle, Width, Ht}) -> Width * Ht;
area({square, X})            -> X * X;
area({circle, R})            -> 3.14159 * R * R.
```

在这个例子中，子句的顺序并不重要，无论这些子句的顺序如何，对程序来说，运行的效果都是一样的。这是因为（这个例子中）各子句的模式彼此互不相干。这使编写和扩展程序变得很简单，只须添加新的模式就行了。不过，通常来说子句的顺序还是有点关系的，因为进入一个函数的时候，调用是按照模式在文件中的顺序依次进行匹配的。

继续深入之前，小结一下，对于area函数的编写方式，我们应该注意以下两点。

(1) area函数由若干个不同的子句构成。当调用这个函数时，对其调用参数的匹配过程从第一个子句开始依次向下进行。

(2) 函数不能处理模式匹配失败的情形，此时程序会失败并抛出一个运行时错误。这一点是故意为之。

很多编程语言，比如C语言，每个函数只有一个入口点。如果用C语言来写这个程序的话，代码可能会是这个样子：

```
enum ShapeType { Rectangle, Circle, Square };

struct Shape {
    enum ShapeType kind;
    union {
        struct { int width, height; } rectangleData;
        struct { int radius; }        circleData;
        struct { int side;}           squareData;
    } shapeData;
};

double area(struct Shape* s) {
    if( s->kind == Rectangle ) {
      int width, ht;
      width = s->shapeData.rectangleData.width;
      ht    = s->shapeData.rectangleData.ht;
```

```
    return width * ht;
  } else if ( s->kind == Circle ) {
    ...
```

36

代码要放在哪儿

如果你下载本书的样例代码或想要编写自己的示例程序，在进入shell运行编译器之前，为确保系统能找到代码，需要确认当前的工作目录是否正确。

在尝试编译示例程序之前，如果你在使用操作系统的命令行shell，那么需要先把目录切换到代码所在的目录。

如果你运行的是Windows上的Erlang标准发布版，也需要将目录切换到存储代码的目录上。Erlang shell中有两个命令可以帮你切换到正确的目录。如果你现在不知道当前位于哪个目录，pwd()可以打印当前的工作目录。cd(Dir)则可以将当前目录切换到Dir所在的目录。在shell中你应该使用正斜杠来分隔目录名。

```
1> cd("c:/work").
c:/work
```

给Windows用户的一个小技巧。创建一个名为C:/Program Files/erl5.4.12/.erlang的文件（根据实际的安装路径进行调整），然后在文件中加入如下的内容：

```
io:format("consulting .erlang in ~p~n",
          [element(2,file:get_cwd())]).
%% Edit to the directory where you store your code
c:cd("c:/work").
io:format("Now in:~p~n", [element(2,file:get_cwd())]).
```

保存之后，每次启动Erlang时，它都能自动切换到目录C:/WORK。

37

C代码向我们原原本本地展示了参数和函数进行模式匹配的过程。在C语言中，程序员必须自己编写模式匹配代码，并保证它们正确无误。

在Erlang中，做相同的事，只需要编写模式，Erlang编译器会自动生成优化的模式匹配代码，帮程序找到正确的入口点。

我们还可以再来看看Java中的等价代码：[①]

```
abstract class Shape {
    abstract double area();
}

class Circle extends Shape {
    final double radius;
    Circle(double radius) { this.radius = radius; }
    double area() { return Math.PI * radius*radius; }
}
```

① 修改自http://java.sun.com/developer/Books/shiftintojava/page1.html。

```
class Rectangle extends Shape {
    final double ht;
    final double width;

    Rectangle(double width, double height) {
        this.ht = height;
        this.width  = width;
    }

    double area() { return width * ht; }
}

class Square extends Shape {
    final double side;

    Square(double side) {
        this.side = side;
    }

    double area() { return side * side; }
}
```

比较Erlang和Java的代码，你会发现，Java代码中的area函数分布在3个不同的地方，而在
38 Erlang的代码中，area的所有的代码都在一起。

3.2 购物系统——进阶篇

回顾一下之前我们讨论过的购物清单，它大致是这个样子：

```
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

假如现在还想知道所购物品的价格，我们必须知道购物清单中各个商品的单价。我们假定这个计算总和的功能由一个名为shop的模块来实现。那么，现在就开始动手，打开你惯用的编辑器，将如下内容输入到一个名为shop.erl的文件中：

shop.erl

```
-module(shop).
-export([cost/1]).

cost(oranges)   -> 5;
cost(newspaper) -> 8;
cost(apples)    -> 2;
cost(pears)     -> 9;
cost(milk)      -> 7.
```

函数cost/1[①]由5个子句组成，每个子句的头部都包括了一个模式（本例之中的模式非常简单，

① 符号Name/N表示一个带有N个参数的名为Name的函数。N称为函数的运算目。

只是一个原子而已）。当我们对shop:cost(X)求值时，系统会用X对这些子句的每一个模式进行匹配。如果匹配了某个模式，那么紧跟在->之后的代码就会执行。

cost/1函数必须从模块之中导出，如果你想从模块的外部调用它，这是必须的。[①]

我们来测试一下。在Erlang shell中编译和运行这个程序：

```
1> c(shop).
{ok,shop}
2> shop:cost(apples).
2
3> shop:cost(oranges).
5
4> shop:cost(socks).
=ERROR REPORT==== 30-Oct-2006::20:45:10 ===
Error in process <0.34.0> with exit value:
        {function_clause,[{shop,cost,[socks]},
        {erl_eval,do_apply,5},
        {shell,exprs,6},
        {shell,eval_loop,3}]}
```

39

第1行编译了shop.erl文件中的模块，第2行、第3行得到了apples和oranges的价格（结果分别是2个和5个货币单位[②]），第4行想得到socks的价格，但因为没有任何子句可以与之匹配，所以得到了一个模式匹配失败的错误，系统打印一条错误消息。[③]

回到购物清单，假设我们的购物清单是这样的：

```
1> Buy = [{oranges,4}, {newspaper,1}, {apples,10}, {pears,6}, {milk,3}].
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

若想计算清单之中所有物品的总价格，可以用下面这种方式来达到目的：

shop1.erl

```
-module(shop1).
-export([total/1]).

total([{What, N}|T]) -> shop:cost(What) * N + total(T);
total([])            -> 0.
```

让我们试试以下代码：

```
2> c(shop1).
{ok,shop1}
3> shop1:total([]).
0
```

这个结果为什么会是0？这是因为total/1的第2个子句是total([])-> 0：

```
4> shop1:total([{milk,3}]).
21
```

① 即用-compile(export_all)，来导出模块之中的所有函数。

② 这里我们并不关心货币单位，因此直接返回数值就好。

③ 错误消息中的“function_clause”部分表明由于没有可匹配参数的子句，从而导致的函数调用失败。

调用时，total([{milk,3}])匹配了子句total([{What,N}|T]，T=[]。[①]匹配之后，此时的变量绑定为{What ↦ milk，N ↦ 3，T ↦ []}。然后，进入函数体(shop:cost(What) * N + total(T))。函数体中的所有变量都被替换为绑定标记的值。那么，函数体的结果就是表达式shop:cost(milk) * 3 + total([])。

shop:cost(milk)的值为7，total([])的值为0，因此，函数体的计算结果也就是7*3+0=21。

如果换上一个更加复杂的参数，结果又会如何？

```
5> shop1:total([{pears,6},{milk,3}]).
75
```

40

在哪里使用分号

我们在Erlang中会遇到3种标点符号。

逗号（,）用来分隔函数调用、数据构造器以及模式中的参数。

句号（.）（后跟一个空白符号）用来在shell中分隔完整的函数和表达式。

分号（;）用来分隔子句，在这几种情况下都会用到子句：分段的函数定义、case语句、if语句、try...catch语句以及receive表达式。

无论何时，我们只要看到一组后面跟有表达式的模式，都会使用分号进行分隔。

```
Pattern1 ->
    Expressions1;
Pattern2 ->
    Expressions2;
...
```

此次，total函数第1个子句的匹配情况是{What↦pears，N↦6，T↦[{milk,3}]}。相对应的结果是shop:cost(pears) * 6 + total([{milk,3}])，也就是9 * 6 + total([{milk,3}])。

上次我们已经计算过total([{milk,3}])的值是21，因此，最终的值也就是：9*6+21=75。

最后：

```
6> shop1:total(Buy).
123
```

在我们结束本节的内容之前，应该更加细致地观察total函数,total(L)通过分析参数L的各种不同条件来工作。这里的L存在两种不同的情况，L是一个非空列表，或者是一个空列表。对于这里的每一种情况，我们都写了一个子句来进行处理：

```
total([Head|Tail]) ->
   some_function_of(Head)  + total(Tail);
total([]) ->
   0.
```

在这个例子中，Head是一个模式{What,N}。当第一个子句匹配一个非空列表时，它会从列表中选出头部，对其进行一些处理，然后调用自身去处理列表的尾部。当列表被缩减至空列表时([])，就会匹配到第2个子句。

41

① 这是因为[X]就是[X|[]]的缩写。

函数total/1实际上做了两件事。第一，它查找列表中每件物品的价格，然后把它们的价格加起来。我们可以对其进行重写，把查找单个物品和价格和对价格进行总计这两个部分分开，代码将会更加清晰易懂。为此我们需要设计两个小的列表处理函数，分别是sum和map。但在此之前，我们需要先了解fun的概念。在此之后，我们将会编写sum和map函数，以及total的改进版本。

3.3 同名不同目的函数

函数的目（arity）就是它所拥有的参数数量。在Erlang中，同一个模块中的两个函数，如果它们同名但是目并不相同，这样的两个函数被认为是完全不相同的。它们之间除了名字恰巧相同之外，彼此之间再无其他关联。

为了方便起见，同名但不同目的函数经常被Erlang程序员用来作为辅助函数。下面就是一个例子：

lib_misc.erl

```
sum(L) -> sum(L, 0).

sum([], N)    -> N;
sum([H|T], N) -> sum(T, H+N).
```

函数sum(L)用于计算列表L之中所有元素的总和。它使用了一个辅助函数sum/2，但这个辅助函数其实可以叫任何其他的名字，你甚至可以把它叫做辅助函数hedgehog/2，而程序的含义保持不变。显然，作为一个命名，sum/2是一个不错的选择，因为它能给程序的读者一点提示，告诉他们函数在干些什么，与此同时你也不用去为此发明一个新的名字（大部分时候，这都不是什么轻松的活儿）。

3.4 fun

fun就是匿名函数。被称为匿名函数，是因为它并没有名字。我们来做点试验，先定义一个fun，然后把它赋给一个变量Z：

```
1> Z = fun(X) -> 2*X end.
#Fun<erl_eval.6.56006484>
```

当定义一个fun时，Erlang shell会打印#Fun<...>，这里的...常常是一些奇怪的数字。不过现在不用管它。

42

我们可以将一个参数应用到一个fun上，对于fun来说，这是我们唯一能做的事，像这样：

```
2> Z(2).
4
```

对这个fun来说，Z显然不是一个好名字，叫做Double可能更好些，这恰好表述了fun的功能：

```
3> Double = Z.
#Fun<erl_eval.6.10732646>
4> Double(4).
8
```

fun可以拥有任意数量的参数。我们可以像下面一样，编个函数来计算直角三角形的斜边：

```
5> Hypot = fun(X, Y) -> math:sqrt(X*X + Y*Y) end.
#Fun<erl_eval.12.115169474>
6> Hypot(3,4).
5.00000
```

如果调用参数的个数不正确，会得到一个错误：

```
7> Hypot(3).
** exited: {{badarity,{#Fun<erl_eval.12.115169474>,[3]}},
            [{erl_eval,expr,3}]} **
```

这个错误为何被称作badarity？回忆一下arity（目）的含义——一个函数可以接受的参数个数。badarity表明在Erlang中找不到由所调用的函数名及其给定的参数数量所表明的函数。在这个例子中，给出了函数Hypot，它需要两个参数，但我们只传了一个。

fun也可以有若干个不同的子句。下面是在华氏气温与摄氏气温之间进行转换的函数：

```
8> TempConvert = fun({c,C}) -> {f, 32 + C*9/5};
8>                   ({f,F}) -> {c, (F-32)*5/9}
8>                end.
#Fun<erl_eval.6.56006484>
9> TempConvert({c,100}).
{f,212.000}
10> TempConvert({f,212}).
{c,100.000}
11> TempConvert({c,0}).
{f,32.0000}
```

说明　第8行的表达式跨越了好几行。在输入这个表达式时，每输入一个新行shell都会重复打印“8>”，这意味着这个表达式并未结束，shell还在等待后续的输入。

Erlang是一种函数式的编程语言，也就是说，除了极个别的情况外，fun既可以作为函数的参数，也可以作为函数（或者fun）的结果。

这些能够返回fun或接受fun作为参数的函数，都被称作高阶函数（high-order function）。在下一节中我们会看到一些与此有关的例子。

现在这些内容看上去好像没什么特别，这只是因为我们目前还未领略到fun的强大威力。虽然到目前为止，fun中的代码与模块之中其他的常规函数的代码看起来没有区别，但实际上，距离真相我们仅有一步之遥。高阶函数是函数式语言的灵魂所在，它能使程序脱胎换骨。一旦掌握了fun的用法，你就会爱上它们。后面我们还会看到更多用到fun的地方。

3.4.1　以 fun 为参数的函数

list是标准库中的一个模块，从中导出的很多函数都是以fun作为参数的。其中，最有用的是lists:map(F,L)。这个函数将fun F应用到列表L的每一个元素上，并返回一个新的列表。

```
12> L = [1,2,3,4].
[1,2,3,4]
```

```
13> lists:map(Double, L).
[2,4,6,8].
```

另一个常用的函数是lists:filter(P,L)，它返回一个新列表，新列表由列表L中每一个能满足P(E)为true的元素组成。

让我们定义一个函数Even(X)，当X为奇数则返回true。

```
14> Even = fun(X) -> (X rem 2) =:= 0 end.
#Fun<erl_eval.6.56006484>
```

其中X rem 2是对X除2取余数，而=:=是一个恒等测试符号。现在我们可以测试一下Even，然后用它作为map和filter的参数。

```
15> Even(8).
true
16> Even(7).
false
17> lists:map(Even, [1,2,3,4,5,6,8]).
[false,true,false,true,false,true,true]
18> lists:filter(Even, [1,2,3,4,5,6,8]).
[2,4,6,8]
```

44

我们将像map和filter这样在一个函数调用中处理整个列表的操作称为list-at-a-time操作。list-at-a-time操作能让程序变得简洁易懂，有了它，我们就可以把处理整个列表的程序看作是一个的抽象步骤。这就是我们的程序能变得更为精炼的原因。否则，我们必须将处理列表元素的操作分解为一系列独立的步骤。

3.4.2 返回 fun 的函数

fun不仅可以用作函数的参数（比如map和filter），而且其他函数也可以将fun当作返回值。

下面是一个例子，假设有一个名为fruit的列表:

```
1> Fruit = [apple,pear,orange].
[apple,pear,orange]
```

现在可以定义一个函数MakeTest(L)，将这个列表（L）转换为一个测试函数，这个测试函数将会检查它所传入的参数是否为列表L中的成员：

```
2> MakeTest = fun(L) -> (fun(X) -> lists:member(X, L) end) end.
#Fun<erl_eval.6.56006484>
3> IsFruit = MakeTest(Fruit).
#Fun<erl_eval.6.56006484>
```

如果X是列表L的成员，那么函数lists:member(X, L)返回true，反之则返回false。现在我们已经编写了一个测试函数，可以进行一些测试：

```
4> IsFruit(pear).
true
5> IsFruit(apple).
true
6> IsFruit(dog).
false
```

同样，我们还能将其用作lists:filter/2的参数：

```
7> lists:filter(IsFruit, [dog,orange,cat,apple,bear]).
[orange,apple]
```

返回fun的fun，这个语法多少还是有些让人迷惑，下面我们再花一点时间来把它说清楚。一个返回“正常”值的fun一般是这样的：

```
1> Double = fun(X) -> ( 2 * X ) end.
#Fun<erl_eval.6.56006484>
2> Double(5).
10
```

45

括号之中的代码（更明确地说，也就是2*X）很明显就是函数的返回值。现在我们试着把这个fun放进括号之中。请记住，括号之中的东西就是返回值：

```
3> Mult = fun(Times) -> ( fun(X) -> X * Times end ) end.
#Fun<erl_eval.6.56006484>
```

现在，括号之中的fun就是fun(X) -> X * Times end，这是关于X的一个函数，但Times是哪里冒出来的？答案就是，它是更外层的fun的参数。

对Mult(3)求值返回fun(X) -> X * 3 end，也就是内层fun的函数体，其中的Times为3所替代。现在我们可以测试一下：

```
4> Triple = Mult(3).
#Fun<erl_eval.6.56006484>
5> Triple(5).
15
```

因此，Mult是Double的一个泛化版本。它并非计算一个值然后返回它，而是会返回一个函数，这个函数只有在被调用的时候才会计算具体的值。

3.4.3　定义你自己的抽象流程控制

等一下，你有没有发现，到目前为止，我们还没有涉及任何if、switch、for、while这些语句，而且似乎也没发现有何不妥，原本应当出现这些语句的位置现在都被模式匹配和高阶函数替代了。到目前为止，我们不需要任何额外的控制结构。

如果需要额外的控制结构，那么就在我们的手边，一个现成的超强的胶水随时可用，我们可用它来自制所需的控制结构。先来看一个例子，Erlang没有for循环，那么就来做一个：

lib_misc.erl

```
for(Max, Max, F) -> [F(Max)];
for(I, Max, F)   -> [F(I)|for(I+1, Max, F)].
```

就这么简单，例如，对for(1,10,F)求值会生成列表[F(1), F(2), ..., F(10)]。

这个for循环中的模式匹配是如何工作的呢？第一个子句仅当第1个参数和第2个参数相等时才会匹配。那么，在调用for(10,10,F)时，第一个子句就会将Max绑定到10，结果就是列表[F(10)]。而如果调用参数是for(1,10,F)，第一个子句不会匹配，因为Max无法同时匹配成1和10。这时，得到匹配的是第二个子句，这个子句返回[F(I)|for(I+1,10,F)]，此时变量绑定为

I↦1，Max↦10，也就是I被1替换，Max被10替换，结果就是[F(1)|for(2,10,F)]。 46

何时使用高阶函数

正如我们看到的，在使用高阶函数时，可以创建自己的新的抽象控制结构。可以将函数作为函数的参数传入，可以编写返回fun的函数。但在实践中，这些技巧并不常用。

- 实际上，我写的所有模块都会用到类似于lists:map/2这样的函数——它如此通用以至于我几乎要把Map当作是Erlang语言的一部分。调用类似map、filter、partition这些lists模块中的函数特别常见。
- 有时，我也创建自己的抽象控制结构，但这远不如调用标准库模块中的高阶函数频繁。在一些大的模块中，也只是偶尔才会用到这种技术。
- 编写返回fun的函数是我很少做的。编写上百个模块，可能也只有一到两个模块会用到这种编程技术。返回fun的函数通常都不容易调试。但从另一方面来说，这一技术可以用来解决诸如延迟求值、可重入的解析器、解析组合子等问题。因为这些问题本身就是返回解析器的函数。

现在我们有了一个简单的for循环，[①]可以用它去创建一个从1到10的整数列表：

```
1> lib_misc:for(1,10,fun(I) -> I end).
[1,2,3,4,5,6,7,8,9,10]
```

或者用它去计算从1到10的整数的平方：

```
2> lib_misc:for(1,10,fun(I) -> I*I end).
[1,4,9,16,25,36,49,64,81,100]
```

随着经验的不断积累，你会发现创建自己的控制结构可以很大地降低程序的代码量，有时它还能让程序变得更为清晰。这是因为你可以根据需求量身定做最为合适的控制结构，超越琐碎呆板的控制结构，从语言的桎梏中解脱。 47

常见错误

有些读者错误地将源代码中的代码片段输入shell，它们并不是有效的shell命令。如果尝试这么做，你会看到一些很奇怪的错误消息。因此再次提醒你，不要这么做。

如果你的模块恰巧使用了与系统模块相同的名字，那么在编译时会得到一个奇怪的消息，告诉你不能从一个保留目录中加载模块。你只需要重新命名，并删掉之前编译所生成的beam文件即可。

3.5 简单的列表处理

现在我们已经介绍过fun了，可以回头继续编写sum和map，它们是编写total函数改进版的必

① 这与一般的命令语言并不完全相同，但是它满足我们的基本需求。

备部分（我相信你还没有忘记）。

我们先从sum开始，它计算列表之中所有元素的和：

mylists.erl

```
❶  sum([H|T]) -> H + sum(T);
❷  sum([])    -> 0.
```

注意，sum中两个子句的顺序是无关紧要的。因为第一个子句匹配一个非空列表，第二个子句匹配空列表，这两种情况是不会互相干扰的。我们可以这样来测试sum：

```
1> c(mylists).  %% <-- Last time I do this
{ok, mylists}
2> L = [1,3,10].
[1,3,10]
3> mylists:sum(L).
14
```

第1行编译了mylists模块。从今往后，在这本书中，我都会忽略掉编译模块的命令，你要自己记得做这件事。sum函数的内部工作机制极易理解。让我们跟踪一下具体的执行情况：

(1) sum([1,3,10])

(2) sum([1,3,10]) = 1 + sum([3,10]) (by ❶)

48

(3) = 1 + 3 + sum([10]) (by ❶)

(4) = 1 + 3 + 10 + sum([]) (by ❶)

(5) = 1 + 3 + 10 + 0 (by ❷)

(6) = 14

最后，看看我们早先论述过的map/2。下面是它的定义：

mylists.erl

```
❶  map(_, [])     -> [];
❷  map(F,  [H|T]) -> [F(H)|map(F, T)].
```

❶ 第一个子句表示该对一个空列表做什么处理。把任何函数映射到一个空列表（它没有任何元素）上只能产生一个空列表。

❷ 第二个子句是一个处理非空列表的规则，它的头是H，尾是T。同样非常简单，只是创建一个新列表，其头是F(H)，尾是map(F,T)。

说明　mylists的这个map/2定义是从标准库的lists模块中复制过来的。在任何情况下都不要企图把你自己的模块名修改为lists，除非你确切地知道会发生什么后果。

下面我们用两个函数来运行map，一个是对列表中的元素乘2，一个是对列表中的元素求平方。

```
1> L = [1,2,3,4,5].
[1,2,3,4,5].
2> mylists:map(fun(X) -> 2*X end, L).
[2,4,6,8,10]
```

```
3> mylists:map(fun(X) -> X*X end, L).
[1,4,9,16,25]
```

对于map的探索是否就要告一段落了呢？嗯，没有，绝对不会！在稍后的章节里，我们将用列表解析来展示一个极为精简的map版本。在20.2节中，我们还会看到如何并行化地计算map中的所有元素（在多核计算机上这将会显著提升程序的运行效率），不过这对于现在的我们还言之尚早。有了sum和map，我们就能利用这两个函数来重写total：

shop2.erl

```
-module(shop2).
-export([total/1]).
-import(lists, [map/2, sum/1]).

total(L) ->
    sum(map(fun({What, N}) -> shop:cost(What) * N end, L)).
```

49

我如何写程序

在写程序的时候，我的方法是写一点测试一点。从一个没多少函数的小模块开始，然后在shell中用几个命令来编译和测试它们。等到测试结果令我满意，才会继续写其他函数，再对新代码编译测试，整个过程都这么展开。

通常，我也不会草率地决定程序需要什么样的数据结构。在运行简单例子的同时，我会不断地审视之前选择的数据结构是否合理。

我写程序倾向于循序渐进地扩展代码，而不是在动手之前就已经完整地构思出来。这种方法使我不会在发现错误前出大的错误。最重要的是，这种方法很有趣，它能让我获得即时的反馈，几乎是在输入的同时就能让我知道我的想法是否可行。

一旦找到某个问题在shell中的解决办法，我通常就会立刻写个makefile以及一些代码来重新生成我从shell中所得的收获。

我们可以通过下面这些分解步骤来了解这个函数是如何工作的：

```
1> Buy = [{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}].
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
2> L1=lists:map(fun({What,N}) -> shop:cost(What) * N end, Buy).
[20,8,20,54,21]
3> lists:sum(L1).
123
```

这个模块里的-import和-export声明的使用也是需要注意的。

- 声明-import(lists, [map/2, sum/1])意味着函数map/2是从lists模块中导入的。也就是说我们可以用map(Fun,...)而不必去写lists:map(Fun,...)。cost/1由于没有在导入声明中声明，所以我们不得不使用完整的名称shop:cost。
- 声明-export([total/1])意味着函数total/1能够在模块shop2之外调用。只有从一个模块中导出的函数才能在模块之外调用。

如果你现在认为我们的total函数已经没有什么改良的余地，那就大错特错了。我们还能进
50 一步地改进它，这就需要用到列表解析技术。

3.6 列表解析

列表解析是一种无须使用fun、map或filter来创建列表的表达式。它能让程序更为简洁且更加容易理解。

先从一个例子开始。假设我们有一个列表L：

```
1> L = [1,2,3,4,5].
[1,2,3,4,5]
```

现在，假设想要把列表当中的每个元素加倍。这我们之前已经做过，这里再重申一下：

```
2> lists:map(fun(X) -> 2*X end, L).
[2,4,6,8,10]
```

与之相比，我们还有一个更为精炼的方式，那就是使用列表解析：

```
4> [2*X || X <- L ].
[2,4,6,8,10]
```

记号[F(X) || X <- L]代表“由F(X)组成的列表，其中X是取值于列表L”。因此，[2*X || X <- L]意味着“列表L中每一个元素X乘以2后的列表”。

为了知道如何使用列表解析，可以先在shell中输入几行表达式，观察一下结果。首先定义Buy：

```
1> Buy=[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}].
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}].
```

现在，把原始列表中每一个元素的个数乘以2：

```
2> [{Name, 2*Number} || {Name, Number} <- Buy].
[{oranges,8},{newspaper,2},{apples,20},{pears,12},{milk,6}]
```

注意，记号（||）右边的元组{Name,Number}是用于匹配列表Buy中每个元素的模式。左边的元组{Name, 2*Number}则是一个构造器。

如果现在想要计算原始列表中所有元素的价格总和，可以这么做，首先用列表中每个元素的单价代替它的名字：

```
3> [{shop:cost(A), B} || {A, B} <- Buy].
[{5,4},{8,1},{2,10},{9,6},{7,3}]
```

然后将价格与数量相乘：

```
4> [shop:cost(A) *  B || {A, B} <- Buy].
[20,8,20,54,21]
```

51 再把它们加起来：

```
5> lists:sum([shop:cost(A) *  B || {A, B} <- Buy]).
123
```

最后，如果想把这些都整合到一个函数中，可以这么写：

```
total(L) ->
    lists:sum([shop:cost(A) *  B || {A, B} <- L]).
```

列表解析能明显地缩短代码，同时也让它更加清晰易懂。我们可以用列表解析来编写一个更为简洁的map定义，看看到底能简洁到什么程度：

```
map(F, L) -> [F(X) || X <- L].
```

下面这个表达式就是一个列表解析的最常见形式：

```
[X || Qualifier1, Qualifier2, ...]
```

X可以是任意一个表达式，每个限定词（qualifier）可以是一个生成器或者是一个过滤器。

- 生成器通常写为Pattern<-ListExpr，其中ListEpr必须是一个对列表项求值的表达式。
- 过滤器可以是一个谓词（返回true或false的函数），也可以是一个布尔表达式。

注意，列表解析中的生成器部分也可以像过滤器一样工作，比如：

```
1> [ X || {a, X} <- [{a,1},{b,2},{c,3},{a,4},hello,"wow"]].
[1,4]
```

下面我们用几个例子来对本节的内容做一个总结。

3.6.1 快速排序

下面这个例子展示了如何使用两个列表解析来完成一个排序算法：[①]

lib_misc.erl

```
qsort([]) -> [];
qsort([Pivot|T]) ->
        qsort([X || X <- T, X < Pivot])
        ++ [Pivot] ++
        qsort([X || X <- T, X >= Pivot]).
```

52

（这里的++是一个中缀添加操作符）：

```
1> L=[23,6,2,9,27,400,78,45,61,82,14].
[23,6,2,9,27,400,78,45,61,82,14]
2> lib_misc:qsort(L).
[2,6,9,14,23,27,45,61,78,82,400]
```

为了弄懂它是如何工作的，我们可以一步一步地跟踪它的执行情况。首先定义一个列表L，然后调用qsort(L)，接下来它匹配了qsort的第二个子句：

```
3> [Pivot|T] = L.
[23,6,2,9,27,400,78,45,61,82,14]
```

其中的变量绑定情况为{Pivot↦23}和{T↦[6,2,9,27,400,78,45,61,82,14]}。

现在，将T分为两个列表，一个列表中的所有元素都是列表T中小于Pivot的，另一个列表中的所有元素都是列表T中大于或等于Pivot的：

```
4> Smaller = [X || X <- T, X < Pivot].
[6,2,9,14]
5> Bigger  = [X || X <- T, X >= Pivot].
[27,400,78,45,61,82]
```

① 这个代码着重于展现代码的优雅性而不是执行效率。这样使用++，一般而言不是一个良好的编程习惯。

现在，对Smaller和Bigger进行排序，然后用Pivot将它们合并起来：

```
qsort( [6,2,9,14] ) ++ [23] ++ qsort( [27,400,78,45,61,82] )
= [2,6,9,14] ++ [23] ++ [27,45,61,78,82,400]
= [2,6,9,14,23,27,45,61,78,82,400]
```

3.6.2　毕达哥拉斯三元组

毕达哥拉斯三元组是一个整数集合{A,B,C}，它使得$A^2+B^2=C^2$。

函数pythag(N)产生一个列表，包含了所有满足$A^2+B^2=C^2$，且3条边之和小于等于整数N的整数集合{A,B,C}。

lib_misc.erl

```
pythag(N) ->
    [ {A,B,C} ||
        A <- lists:seq(1,N),
        B <- lists:seq(1,N),
        C <- lists:seq(1,N),
        A+B+C =< N,
        A*A+B*B =:= C*C
    ].
```

这里简单地解释几句：lists:seq(1,N)返回一个由1到N整数组成的列表，所以A<-lists:seq(1, N)意味着A的取值范围是1到N的所有整数。因此我们的程序可以这么理解，“从1到N中得到A的所有可能取值，从1到N中获得B的所有可能取值，从1到N中获得C的所有可能取值，使得A+B+C小于等于N且A*A + B*B = C*C”。

53

```
1> lib_misc:pythag(16).
[{3,4,5},{4,3,5}]
2> lib_misc:pythag(30).
[{3,4,5},{4,3,5},{5,12,13},{6,8,10},{8,6,10},{12,5,13}]
```

3.6.3　变位词

如果你着迷于英语式的纵横字迷，你自己就会发现所谓的变位词。现在让我们用Erlang编写一个漂亮的小函数perms去寻找一个字符串所有可能的排列。下面就是这个函数的具体代码。

lib_misc.erl

```
perms([]) -> [[]];
perms(L)  -> [[H|T] || H <- L, T <- perms(L--[H])].
```

```
1> lib_misc:perms("123").
["123","132","213","231","312","321"]
2> lib_misc:perms("cats").
["cats", "cast", "ctas", "ctsa", "csat", "csta", "acts", "acst",
"atcs", "atsc", "asct", "astc", "tcas", "tcsa", "tacs", "tasc",
"tsca", "tsac", "scat", "scta", "sact", "satc", "stca", "stac"]
```

X--Y是列表的分离操作符，它从列表X中分离出元素Y，在5.4节中有更为详尽的定义。

这一次，我就不去解释perms是如何工作的了，因为解释的篇幅会是程序本身的几倍，你可以自己去研究一下这个函数！[不过，这里有一个小提示：要求出X123的所有排列，那么就要先求出123的全部排列（它们是123 132 213 231 312 321）。然后往每个排列的所有可能的位置上插入X，把X插入123就有X123 1X23 12X3 123X，将X插入132就有X132 1X32 13X2 132X，以此类推，递归地应用这些规则。]

3.7 算术表达式

在表3-1中展示了所有可能的算术表达式。每种算术操作符有1个或者2个参数，在表3-1中将用Integer或者Numer来指代这些这些参数（Number意味着参数可以是整数或者浮点数）。

每个操作符都有其优先级。对一个复杂的算术表达式求值的顺序依赖于操作符的优先级，所有优先级为1的操作符首先会被求值，其次是所有优先级为2的操作符，以此类推。

54

你可以使用括号来改变默认的求值顺序——任何括号中的表达式都会先于其他符号求值。相同优先级的操作符会被看作是左结合的，也就是从左向右进行求值。

表3-1 算术表达式

操 作	描 述	参数类型	优 先 级
+X	+X	Number	1
-X	-X	Number	1
X*Y	X*Y	Number	2
X/Y	X/Y（浮点数相除）	Number	2
bnot X	对X按位取反	Integer	2
X div Y	X整除Y	Integer	2
X rem Y	X除Y取余数	Integer	2
X band Y	对X和Y按位取与	Integer	2
X+Y	X+Y	Number	3
X-Y	X-Y	Number	3
X bor Y	对X和Y按位取或	Integer	3
X bxor Y	对X和Y按位进行异或	Integer	3
X bsl N	对X按位左移N位	Integer	3
X bsr N	对X按位右移N位	Integer	3

3.8 断言

断言（guard）是一种用于强化模式匹配功能的结构。使用断言，可以在一个模式上做一些简单的变量测试和比较。假如我们想要写一个函数max(X,Y)求出X与Y之间的最大值，那么可以使用断言来实现它:

```
max(X, Y) when X > Y -> X;
max(X, Y) -> Y.
```

当X大于Y的时候，第一个子句就得到匹配，其结果为X。

如果第一个子句不匹配，那么尝试匹配第二个子句，第二个子句总是返回第2个参数Y。此时Y必定大于等于X，否则，第一个子句会先被匹配。

你可以在函数定义的头部使用断言，此时断言必须以关键字when开头，当然也可以在任何语
55 言中允许使用表达式的地方使用断言。当断言用于表达式时，它们会被求值为一个原子true或者false。如果断言求值为true，那么我们认为求值成功，否则就认为求值失败。

3.8.1　断言序列

断言序列可以是单个断言也可以是一系列用分号（;）分开的断言集合。在断言集合G1; G2; ...; Gn中，只要任何一个断言为true，那么整个断言序列就为true。

断言也可以是一系列用逗号分开的断言集合。断言集合GuardExpr1, GuardExpr2, ..., GuardExprN中，只有所有的断言都为true，整个断言序列才为true。

合法的断言表达式只是合法Erlang表达式的子集。我们把断言表达式限制为Erlang表达式的子集是为了保证断言表达式没有副作用。由于断言是模式匹配的一种扩展，而模式匹配又必须是无副作用的，所以保证断言没有副作用也就成为一种必然。

另外，断言不能使用用户定义的布尔表达式，因为无法确保它们没有副作用，也不能确保它们可以从函数里退出。

下面是断言表达式的合法的语法形式。

- 原子true。
- 其他常量（条件或者绑定变量），这些在断言表达式中都会被求值为false。
- 列在表3-2中的断言谓词，或者表3-3中的BIF[①]。
- 比较表达式（图5-3）。
- 算术表达式（表3-1）。
- 布尔表达式（5.4.5节）。
- 短路布尔表达式（5.4.20节）。

56 在对断言表达式求值时，采用5.4.17节中描述的优先规则。

表3-2　断言谓词

谓　词	含　义	谓　词	含　义
is_atom(X)	X是原子	is_integer(X)	X是整数
is_binary(X)	X是二进制数据	is_list(X)	X是列表
is_constant(X)	X是常数	is_number(X)	X是整数或浮点数
is_float(X)	X是浮点数	is_pid(X)	X是进程标识符
is_function(X)	X是函数	is_port(X)	X是端口
is_function(X, N)	X是有N个参数的函数	is_reference(X)	X是引用

① BIF是built-in function（内建函数）的缩写，参见5.1节。

（续）

谓　词	含　义	谓　词	含　义
is_tuple(X)	X是元组	is_record(X,Tag,N)	X是标记为Tag大小为N的记录
is_record(X,Tag)	X是标记为Tag的记录		

表3-3 断言BIF

函　数	含　义
abs(X)	X的绝对值
element(N, X)	元组X的第N个元素
float(X)	将数字X转换为浮点数
hd(X)	列表X的头部
length(X)	列表X的长度
node()	当前节点
node(X)	创建X的节点，X可以是进程标识符、引用或端口
round(X)	将数字X转换为整数（四舍五入）
self()	当前进程的进程标识符
size(X)	X的大小，X为元组或二进制数据
trunc(X)	将数字X转换为整数（截取）
tl(X)	列表X的尾部

3.8.2 断言样例

```
f(X,Y) when is_integer(X), X > Y, Y < 6 -> ...
```

这个例子表示“当X是一个整数，且X大于Y，且Y小于6”，这里将断言分开的逗号表示and（与）操作。

```
is_tuple(T), size(T) =:= 6, abs(element(3, T)) > 5
element(4, X) =:= hd(L)
...
```

第1行表示T是具有6个元素的元组，且T的第3个元素的绝对值大于5。第2行表示X的第4个元素与列表L中的头元素相等。

```
X =:= dog; X =:= cat
is_integer(X), X > Y ; abs(Y) < 23
...
```

第一个断言表示X是cat，或是dog。第二个断言表示X是一个整数且大于Y，或者Y的绝对值小于23。

下面是几个使用短路布尔表达式的断言样例：

```
A >= -1.0 andalso A+1 > B
is_atom(L) orelse (is_list(L) andalso length(L) > 2)
```

进阶 在断言中允许使用布尔表达式是因为可以让断言的语法与其他表达式保持一致。使用orelse和andalso操作符的原因是布尔操作符and/or的原始定义中需要对它们的参数都

求值。在断言中，and和andalso之间，或者or和orelse之间是有区别的。例如，考虑如下两个断言：

```
f(X) when (X == 0) or (1/X > 2) ->
    ...

g(X) when (X == 0) orelse (1/X > 2) ->
    ...
```

当X为0时，f(x)中的断言求值为false，但是g(x)中的断言为true。

57 在日常编程中，很少有代码会使用复杂的断言，简单的（,）断言可以满足大多数程序的需要。

3.8.3 true 断言的使用

你可能会想为何需要true断言？原因是原子true可以在if表达式的末尾用作“catchall”断言。就像这样：

```
if
    Guard -> Expressions;
    Guard -> Expressions;
    ...
    true -> Expressions
end
```

我们会在3.10.2节中讨论if。

3.8.4 过时的断言函数

如果你看到过一些几年前写的老式Erlang代码，断言测试的名字可能非常不同。老式代码的断言测试都是使用诸如atom(X)、constant(X)、float(X)、integer(X)、list(X)、number(X)、pid(X)、port(X)、reference(X)、tuple(X)和binary(X)这样的函数名。在新版Erlang中，与之相对应的测试函数的名称都形如is_atom(X)，它们的意义相同。但是我强烈建议你不要在新

58 版的Erlang中使用老式的函数名。

3.9 记录

在用元组进行编程时，通常会遇到这样一种情况：当元组的元素的数量变得非常庞大时，我们会很难记住元组中每一个元素的确切含义。记录（record）就提供了一种方法把一个名称与元组中的一个元素对应起来，从而解决这一烦恼。

我们时常都能看到操作小型元组的程序，这是因为小型元组对程序并不构成什么威胁，并且不会混淆元素之间的不同含义。记录通过下面这种语法来定义：

```
-record(Name, {
                %% the next two keys have default values.
                key1 = Default1,
                key2 = Default2,
                ...
                %% The next line is equivalent to
                %% key3 = undefined
```

```
        key3,
        ...

    }).
```

警告 record不是一个shell命令（在shell中使用rr，参看本节稍后的描述）。记录的声明只能用于Erlang源代码而不能用于shell。

59

在前面的例子中，Name是记录的名字，key1、key2等是记录中的字段名，这些名字都必须是原子。记录中的每个字段都可以有默认值。如果在记录创建的时候，没有为某个特定的字段指定值，那么就会使用默认值。

例如，假如我们想要操纵一个未完成列表（to-do list）。首先定义一个todo记录，然后将其存入一个文件（记录的定义可以包含在Erlang源代码或以.hrl为扩展名的文件中，这些文件可以被Erlang源代码引用[①]）。

records.hrl

```
-record(todo, {status=reminder,who=joe,text}).
```

一旦定义了记录，就能创建记录的实例。如果在shell中也想这么做，那么在定义一个记录之前必须在shell中读取记录的定义。可以用rr（read record的缩写）这个命令:

```
1> rr("records.hrl").
[todo]
```

3.9.1 创建和更新记录

现在可以开始定义并操纵记录了：

```
2> X=#todo{}.
#todo{status = reminder,who = joe,text = undefined}
3> X1 = #todo{status=urgent, text="Fix errata in book"}.
#todo{status = urgent,who = joe,text = "Fix errata in book"}
4> X2 = X1#todo{status=done}.
#todo{status = done,who = joe,text = "Fix errata in book"}
```

在第2行和第3行各创建了一个新记录。语法#todo{key1=Val1,..., keyN=ValN}用于创建类型为todo的新记录。key都是原子，而且必须和记录定义中的一样。若省略key，那么就使用记录定义中的默认值。

第4行，复制了一个存在的记录。语法X1#todo{status=done}表示创建一个X1的副本（X1必须为todo类型），并将value字段的值修改为done。记住，这只是原记录的一个副本，原记录本身并未改变。

60

3.9.2 从记录中提取字段值

同其他情况一样，这里也可以使用模式匹配：

① 只有这种方法才能保证不同的Erlang模块使用同一个记录定义。

```
5> #todo{who=W, text=Txt} = X2.
#todo{status = done,who = joe,text = "Fix errata in book"}
6> W.
joe
7> Txt.
"Fix errata in book"
```

匹配操作符的左边，我们使用了自由变量W和Txt来定义一个记录模式。如果匹配成功，这些变量将会绑定到记录中对应的字段值。如果仅想提取记录之中某个字段的值，那么我们可以使用“点语法”来获取：

```
8> X2#todo.text.
"Fix errata in book"
```

3.9.3　在函数中对记录进行模式匹配

我们可以写这样的函数，它可以对一个记录的字段进行模式匹配并创建一个新的记录。代码通常是这样的：

```
clear_status(#todo{status=S, who=W} = R) ->
    %% Inside this function S and W are bound to the field
    %% values in the record
    %%
    %% R is the *entire* record
    R#todo{status=finished}
    %% ...
```

为匹配一个特定类型的记录，可以这样来定义函数：

```
do_something(X) when is_record(X, todo) ->
    %% ...
```

当X是一个类型为todo的记录时这个子句就会得到匹配。

3.9.4　记录只是元组的伪装

记录就是元组。现在让我们告诉shell释放掉todo的定义：

```
11> X2.
#todo{status = done,who = joe,text = "Fix errata in book"}
12> rf(todo).
ok
13> X2.
{todo,done,joe,"Fix errata in book"}
```

61

第12行告诉shell释放掉todo这个记录的定义。因此当打印X2的时候，shell就把x2显示为元组。在系统的内部只有元组，而记录只是使命名元组中不同的元素的语法简单而已。

3.10　case/if表达式

到目前为止，我们用模式匹配来处理所有的事，这让Erlang代码更加灵巧和协调。但有的时候为每件事情都定义一个独立的函数子句会显得相当不便。如果遇到这种情况，就有必要使用

case/if表达式。

3.10.1 case表达式

case的语法是这样的：

```
case Expression of
    Pattern1 [when Guard1] -> Expr_seq1;
    Pattern2 [when Guard2] -> Expr_seq2;
    ...
end
```

case运算的过程是这样的：首先，对Expression进行求值（假定最后的结果存为Value），然后，Value依次对Pattern1（带有可选的断言Guard1）、Pattern2等进行模式匹配，直到找到一个可以匹配的分支。一旦找到这样的分支，就对其后相应的表达式序列进行求值——这个表达式序列的求值结果也就是case表达式的返回值。如果没有任何分支被匹配到，那么就会抛出一个异常。

在前面的内容中，我们曾使用过一个叫做filter(P,L)的函数，它将L中的所有元素对函数P求值，并且把P(X)为true的元素组成一个列表返回。现在用模式匹配来定义这个filter：

```
filter(P, [H|T]) ->  filter1(P(H), H, P, T);
filter(P, [])    ->  [].

filter1(true, H, P, T)  -> [H|filter(P, T)];
filter1(false, H, P, T) -> filter(P, T).
```

但这样的定义并不优雅，因为我们不得不引入一个叫做filter1的辅助函数，然后把filter/2的参数都传给它。

62

有了case结构，可以定义一个更为清晰的版本：

```
filter(P, [H|T]) ->
    case P(H) of
        true  -> [H|filter(P, T)];
        false -> filter(P, T)
    end;
filter(P, []) ->
    [].
```

3.10.2 if表达式

Erlang也提供另外一种条件原语if。下面是它的语法：

```
if
  Guard1 ->
    Expr_seq1;
  Guard2 ->
    Expr_seq2;
  ...
end
```

它按照这样的方式来求值：先对Guard1求值，若求值为true，那么if的值就是随后的表达式序列Expr_seq1的运行结果。若Guard1不成功，那么就继续对Guard2求值，以此类推，直到有断言成功。在if表达式的这些断言中，至少要有一个结果为true，否则就会抛出一个异常。

通常if表达式的最后一个断言会是原子true，如果其他的断言全都失败，它可以保证表达式的最后一个分支会被求值。

3.11　以自然顺序创建列表

创建一个列表最有效率的方法是把元素加在一个现有的列表头部，因此通常会看到这样的匹配模式：

```
some_function([H|T], ..., Result, ...) ->
    H1 = ... H ...,
    some_function(T, ..., [H1|Result], ...);
some_function([], ..., Result, ...) ->
    {..., Result, ...}.
```

这个代码依序遍历一个列表，同时提取列表的头部H，用它在函数中计算某个值（称为H1），
63 然后将H1加入到输出列表Result。

当输入列表耗尽，此时最后一个子句得到匹配，输出变量Reuslt将从函数中返回。Reuslt中元素的顺序与原列表中元素的顺序相反，这可能也不是什么大问题。但如果它们的顺序确实有意义，我们在最后一步可以非常简单地反转顺序。

这里的基本概念其实相当简单。

(1) 总是在列表头部添加元素。

(2) 从一个输入列表的头部提取元素，然后把它们加在一个输出列表的头部。输出列表中的结果与输入列表的顺序相反。

(3) 如果顺序至关重要，那么调用经过高度优化的函数lists:reverse/1。

(4) 避免违反这些原则。

说明　无论何时想要反转一个列表，都应该调用lists:reverse，而不是试图寻找其他方法。如果你看过lists模块的源代码，会找到反转的定义，但这个定义仅仅用于简单的说明。因为对于编译器来说，一旦它发现调用了lists:reverse，就会调用这个函数在系统内部更为高效的版本。

如果你曾见过这样的代码：

```
List ++ [H]
```

那么请在脑袋里为此设置一个警钟——这是一个极为低效的操作，只有在列表非常短的时候才适用。

3.12　累加器

如何在函数之外得到两个列表？如何写一个函数将一个整数列表分解为偶数列表和奇数列

表？我们可以这样做：

lib_misc.erl

```
odds_and_evens(L) ->
    Odds  = [X || X <- L, (X rem 2) =:= 1],
    Evens = [X || X <- L, (X rem 2) =:= 0],
    {Odds, Evens}.
```

```
5> lib_misc:odds_and_evens([1,2,3,4,5,6]).
{[1,3,5],[2,4,6]}
```

64

这段代码的问题在于遍历了列表两次——如果列表很短，那这并不构成问题，但是如果它很长，那么就会降低效率。

为了避免遍历列表两次，可以这样来重构代码：

lib_misc.erl

```
odds_and_evens_acc(L) ->
    odds_and_evens_acc(L, [], []).

odds_and_evens_acc([H|T], Odds, Evens) ->
    case (H rem 2) of
        1 -> odds_and_evens_acc(T, [H|Odds], Evens);
        0 -> odds_and_evens_acc(T, Odds, [H|Evens])
    end;
odds_and_evens_acc([], Odds, Evens) ->
    {Odds, Evens}.
```

现在，通过引入odd和even这两个参数，只需遍历一次列表就能把结果输出到对应的列表上去（这就是累加器）。这段代码还有一个不太明显的好处，使用累加器的版本要比使用`[H || filter(H)]`类型构造器的版本更加节省空间。

对它进行测试，我们会得到与之前版本相同的结果。

```
1> lib_misc:odds_and_evens_acc([1,2,3,4,5,6]).
{[5,3,1],[6,4,2]}
```

不同之处在于odd和even中的元素顺序恰好相反。这是列表构造的结果，如果想要维持与原列表相同的顺序，我们要做的只是在函数的最后一个子句中反转列表，把odds_and_evens_acc的第2个子句改成下面这样：

```
odds_and_evens_acc([], Odds, Evens) ->
    {lists:reverse(Odds), lists:reverse(Evens)}.
```

至此我们学到了什么

现在我们可以编写Erlang模块和一些简单的顺序型代码，也具备了几乎所有编写顺序型Erlang程序的知识。

下一章我们会简要地介绍一下错误处理。随后，会再回到顺序型编程，去学习目前我们跳过的那些遗留细节。

65

第4章

异　常

4.1 异常

如果你一步一个脚印地跟着第3章中的代码学下来，那么就应该看到过一些运行中产生的Erlang错误报告和处理。在我们更加深入学习顺序型编程之前，需要花一些时间去了解一下与异常相关的细节。这看起来有点舍近求远，但是如果你阅读本书的最终目的是想构建一个健壮的分布式应用程序，那么透彻理解错误处理的工作机制无疑是关键所在。

一个函数可能成功地返回一个值，也可能出现执行错误。每次在Erlang中调用函数时，结果肯定是二者中的一个。现在看一下第3章中出现过的例子，还记得cost函数吗？

shop.erl

```
cost(oranges)    -> 5;
cost(newspaper) -> 8;
cost(apples)     -> 2;
cost(pears)      -> 9;
cost(milk)       -> 7.
```

它的运行结果为：

```
1> shop:cost(apples).
2
2> shop:cost(socks).
=ERROR REPORT==== 30-Oct-2006::20:45:10 ===
Error in process <0.34.0> with exit value:
       {function_clause,[{shop,cost,[socks]},
       {erl_eval,do_apply,5},
       {shell,exprs,6},
       {shell,eval_loop,3}]}
```


当调用cost(socks)时，函数崩溃了，这是因为函数定义的所有子句中没有一个能与调用的参数相匹配。

调用cost(socks)毫无意义。由于socks的价格根本就没有定义，函数也就不可能返回任何有意义的值。因此，系统抛出一个异常而不是返回一个值。是的，就是用“异常”这个专业术语来称呼平时我们所说的“崩溃”。

我们没有必要尝试去修复这个错误，因为它是不可能修复的。既然我们并不知道socks的价格，显然也就不可能返回它。只有cost(socks)的调用者才知道函数崩溃之后该去做什么。

遇到系统内部错误，或者在代码中显式地调用throw(Exception)、exit(Exception)或erlang:error(Exception)，系统就会抛出异常。

Erlang有两种方法去捕获异常：一种是用try...catch表达式将一个会抛出异常的函数括起来，另一种是把函数调用包含在catch表达式里。

4.2 抛出异常

当系统遇到错误时，就会自动地抛出异常。最常见的错误就是模式匹配失败（没有可匹配的函数子句），或者是在调用BIF时传入了错误类型的参数（比如，调用atom_to_list时传了整数参数）。

也可以通过调用下面几个内建的异常产生函数来显式地产生一个错误：

```
exit(Why)
```

当想要终止当前进程时，就需要用到这个函数。如果这个异常未被捕获，那么系统会向所有与当前进程相连接的进程广播{'EXIT',Pid,Why}消息。在9.1节中，我们会着重讨论这个话题，在此先不深入讨论它。

```
throw(Why)
```

这个函数用于抛出一个调用者可能会捕获的异常。在这种情况下，有必要为函数添加注释，说明它会抛出这个异常。这个函数的调用者有两种选择：要么大胆放心地忽略这些异常，然后一如既往地编写代码；要么将这个调用包含在try...catch表达式中并对错误进行处理。

68

```
erlang:error(Why)
```

这个函数用于抛出那些“崩溃错误”。这些异常应该是那些调用者不会真正意识到要去处理的致命错误，可以将它等同于内部产生的系统错误。

好，下面试着来捕获这些错误。

4.3 try...catch

如果你熟悉Java，那么相信你不会对try...catch表达式感到陌生。Java中可以使用下面这样的语法来捕获一个异常：

```
try {
    block
} catch (exception type identifier) {
    block
} catch (exception type identifier) {
    block
} ...
finally {
    block
}
```

Erlang有着与之极为相似的语法结构，具体语法如下：

```
try FuncOrExpressionSequence of
    Pattern1 [when Guard1] -> Expressions1;
    Pattern2 [when Guard2] -> Expressions2;
    ...
catch
    ExceptionType: ExPattern1 [when ExGuard1] -> ExExpressions1;
    ExceptionType: ExPattern2 [when ExGuard2] -> ExExpressions2;
    ...
after
    AfterExpressions
end
```

注意try...catch表达式和case表达式之间的相似性：

```
case Expression of
    Pattern1 [when Guard1] -> Expressions1;
    Pattern2 [when Guard2] -> Expressions2;
    ...
end
```

try...catch就像是一个case表达式的增强版本。基本上它就是一个在尾部带有catch和after块的case表达式。

69

try...catch表达式是有值的

记住，Erlang中的任何东西都是表达式，而所有的表达式都有值。这也就意味着try...end表达式也不能例外，它也是有值的。因此，我们可能会写这样的代码：

```
f(...) ->
    ...
    X = try ... end,
    Y = g(X),
    ...
```

不过一般而言，我们并不需要try...catch表达式的值。因此，可以这么写：

```
f(...) ->
    ...
    try ... end,
    ...
    ...
```

try...catch按照如下规则工作：首先，对FuncOrExpessionSeq进行求值。如果它不产生异常地运行完毕，那么函数的返回值就对模式Pattern1（连同可选的断言Guard1）、Pattern2等进行模式匹配，直到有一个模式匹配成功为止。如果有模式匹配到，那么，在这个模式之后的表达式就会被求值，然后它的值就被当作整个try...catch的值返回。

如果在FuncOrExpressionSeq中有异常抛出，那么就会逐个匹配catch下的ExPattern1等模式，然后对相应的表达式序列进行求值。ExceptionType是原子throw、exit或error中的一个，告诉我们异常是以什么方式产生的。若未标明ExceptionType，那么默认的值就是throw。

说明 由Erlang运行时系统所检测到的系统错误一律都有error标签。

跟在after关键字后面的代码用于执行完FuncOrExpressionSeq后的清理工作。无论是否抛出异常，这段代码一定都会被执行。在表达式的try或者catch段中代码运行完毕之后，after段中的代码就会立即被执行。在AfterExpressions中的返回值会被自动舍弃。 70

如果你来自Ruby社区，那么所有这些内容都应该非常熟悉——在Ruby中，我们编写一个类似的模式：

```
begin
   ...
rescue
   ...
ensure
   ...
end
```

只是关键字不同，[1]所有的行为都是相似的。

4.3.1 缩减版本

我们可以忽略try...catch表达式的几个部分，就像这样：

```
try F
catch
   ...
end
```

它的意义等同于下面的表达式：

```
try F of
  Val -> Val
catch
   ...
end
```

当然，after段也是可以忽略的。

4.3.2 使用 try...catch 的编程惯例

当我们设计应用程序时，经常需要保证捕获代码能够捕获这个函数所有可能产生的错误。下面两个函数演示了这种情况。第一个函数能够抛出所有类型的异常：

try_test.erl

```
generate_exception(1) -> a;
generate_exception(2) -> throw(a);
generate_exception(3) -> exit(a);
generate_exception(4) -> {'EXIT', a};
generate_exception(5) -> erlang:error(a).
```

① 在Erlang中没有retry表达式。

71

现在我们再来编写一个包装函数，它在try...catch表达式中调用generate_exception。

try_test.erl

```
demo1() ->
    [catcher(I) || I <- [1,2,3,4,5]].

catcher(N) ->
   try generate_exception(N) of
       Val -> {N, normal, Val}
   catch
       throw:X -> {N, caught, thrown, X};
       exit:X  -> {N, caught, exited, X};
       error:X -> {N, caught, error, X}
   end.
```

运行它我们会得到如下结果:

```
> try_test:demo1().
[{1,normal,a},
 {2,caught,thrown,a},
 {3,caught,exited,a},
 {4,normal,{'EXIT',a}},
 {5,caught,error,a}]
```

这个样例告诉我们可以捕获和区分一个函数抛出的任何形态的异常。

4.4 catch

捕获异常的另一种方式是使用catch原语。当你捕获一个异常时，这个异常会被转换为描述错误的一个元组。为了演示这个语法，我们可以在一个catch表达式中调用函数generate_exception:

try_test.erl

```
demo2() ->
    [{I, (catch generate_exception(I))} || I <- [1,2,3,4,5]].
```

运行它，我们会得到如下结果：

```
2> try_test:demo2().
[{1,a},
{2,a},
{3,{'EXIT',a}},
{4,{'EXIT',a}},
{5,{'EXIT',{a,[{try_test,generate_exception,1},
               {try_test,'-demo2/0-fun-0-',1},
               {lists,map,2},
               {lists,map,2},
               {erl_eval,do_apply,5},
               {shell,exprs,6},
               {shell,eval_loop,3}]}}}]
```

72

如果把这个结果与try...catch块的输出比较，你会发现在分析问题原因时，我们丢失了许多关于错误的详细信息。

4.5 改进错误信息

erlang:error的另一个用处是提高错误信息的质量。当用一个负数参数去调用math:sqrt(X)时，我们会得到如下信息：

```
1> math:sqrt(-1).
** exited: {badarith,[{math,sqrt,[-1]},
                      {erl_eval,do_apply,5},
                      {shell,exprs,6},
                      {shell,eval_loop,3}]} **
```

我们可以为此编写一个包装函数，它可以提高错误消息的质量：

lib_misc.erl

```
sqrt(X) when X < 0 ->
    erlang:error({squareRootNegativeArgument, X});
sqrt(X) ->
    math:sqrt(X).
```

```
2> lib_misc:sqrt(-1).
** exited: {{squareRootNegativeArgument,-1},
      [{lib_misc,sqrt,1},
       {erl_eval,do_apply,5},
       {shell,exprs,6},
       {shell,eval_loop,3}]} **
```

4.6 try...catch 的编程风格

在实际应用中，我们通常会怎样进行错误处理呢？一般会遇到下面几种情形。

4.6.1 经常会返回错误的程序

如果函数包含一个“非常规分支”，那么就应该返回类似{ok, Value}或者{error, Reason}这样的返回值。不过请记住，这将强迫所有的调用者都要去处理返回值。这种方式中有两种不同的风格可以供你选择，你可以这么写：

```
...
case f(X) of
    {ok, Val} ->
        do_some_thing_with(Val);
    {error, Why} ->
        %% ... do something with the error ...
end,
...
```

这种方法会兼顾两种返回值。当然你还可以这么写：

```
...
{ok, Val} = f(X),
do_some_thing_with(Val);
...
```

如果f(x)返回{error, ...}，这种写法就会抛出一个异常。

4.6.2 出错几率比较小的程序

典型的处理错误的代码像下面的样例一样：

```
try my_func(X)
catch
  throw:{thisError, X} -> ...
  throw:{someOtherError, X} -> ...
end
```

检测异常的代码分支应该与函数中的异常抛出分支互相匹配：

```
my_func(X) ->
    case ... of
        ...
        ... ->
                ... throw({thisError, ...})
        ... ->
                ... throw({someOtherError, ...})
```

4.7 捕获所有可能的异常

如果我们想要捕获每一个可能的错误，那么可以采用下面这种习惯用法：

```
try Expr
catch
   _:_ -> ... Code to handle all exceptions  ...
end
```

如果我们忽略异常标签且按下面这么写：

```
try Expr
catch
    _ -> ... Code to handle all exceptions  ...
end
```

74 那么我们将不能捕获所有的错误，因为在这种情况下默认的错误标签是throw。

4.8 新老两种异常处理风格

本节是专门写给Erlang的资深用户的！

try...catch是一个相对新颖的语法结构，它是为了补偿catch...throw机制的不足而引入的。如果你是一个还没有阅读过最新文档的老用户（就像我），那么你会习惯性地写出这样的错误处理代码：

```
case (catch foo(...)) of
    {'EXIT', Why} ->
```

```
        ...
    Val ->
        ...
end
```

这通常没有错，但它的效果不会像下面这种方式那么好：

```
try foo(...) of
    Val -> ...
catch
    exit: Why ->
        ...
end
```

因此，最好使用try...of...，而避免使用case (catch ...) of ...。

4.9 栈跟踪

当捕获异常时，我们可以通过调用erlang:get_stacktrace()来查看最近的栈跟踪信息。下面就是一个样例：

try_test.erl

```
demo3() ->
    try generate_exception(5)
    catch
        error:X ->
            {X, erlang:get_stacktrace()}
    end.
```

```
1> try_test:demo3().
{a,[{try_test,generate_exception,1},
    {try_test,demo3,0},
    {erl_eval,do_apply,5},
    {shell,exprs,6},
    {shell,eval_loop,3}]}
```

75

栈跟踪信息包含了当前调用函数的返回信息，也就是说如果当前调用不发生异常的话，那么当前的函数就会根据栈中的函数列表依次返回。这些信息几乎就是我们运行到当前函数所经过的调用路径（在调试时，这太有用了），不过这些信息不包含任何尾递归[①]调用的路径信息。

从调试程序的角度来说，我们只关心栈跟踪信息的前面几行。之前的那个栈跟踪信息告诉我们，当使用一个参数对try_test模块的generate_exception函数进行求值时，系统会崩溃。而try_test:generate_exception/1函数可能是由try_test:demo3()函数调用的。对于这一点，我们其实是无法肯定的。这是因为try_test:demo3()可能还调用了其他的函数，使尾递归函数调用try_test:generate_exception/1，在这种情况下，栈跟踪信息无法记录任何中间的函数调用。

76

① 参看8.9节。

第5章

顺序型编程进阶

本章我们要回来接着学习Erlang的顺序型编程。在第3章中，我们学习了编写函数的基本方法。本章则要继续涵盖下面这些话题。

- BIF：BIF是built-in function（内建函数）的缩写，是Erlang语言的一个组成部分。虽说看上去好像是用Erlang编写的，但实际上不是，它们是Erlang虚拟机中的基本操作。
- 二进制数据：它是一种数据类型，可以用它实现原始数据的高速存储。
- 比特语法：这是一种模式匹配语法，主要用于对二进制数据中的比特进行封包和解包工作。
- 小问题集锦：这里涉及的少量话题是我们精通顺序型编程所必不可少的。

你掌握了本章的内容，可以说就已经基本地了解了顺序型编程的绝大多数知识，同时也为深入探索神秘的并发编程领域打好了坚实的基础。

5.1 BIF

BIF（内建函数）顾名思义，就是Erlang内建的函数。它们通常用来完成那些无法用Erlang完成的任务。比如，将列表转换为元组或者获取当前的时间和日期。完成这些操作的函数，我们称之为BIF。

例如，`tuple_to_list/1`能将元组转换为列表，`time/0`返回当前时间的时、分、秒：

```
1> tuple_to_list({12,cat,"hello"}).
[12,cat,"hello"]
2> time().
{20,0,3}
```

所有的BIF都在`erlang`模块之中，而且大部分常用的BIF（如`tuple_to_list`）都已被自动导入，因此我们只需要写`tuple_to_list(...)`而不是烦琐的`erlang:tuple_to_list(...)`。

在Erlang发布版关于`erlang`模块的手册中能找到所有BIF的列表，也可以在线访问http://www.erlang.org/doc/man/erlang.html。

5.2 二进制数据

在Erlang中可以使用一种叫做二进制（binary）数据的结构来存储大量的原始数据。相对于

列表或元组，二进制类型更加节省内存，而且运行时系统也对此进行了优化，对二进制数据的输入输出会更加高效。

在书写和打印时，二进制数据以一个整数或者字符序列的形式出现，两端分别用两个小于号和两个大于号括起来，例如：

```
1> <<5,10,20>>.
<<5,10,20>>
2> <<"hello">>.
<<"hello">>
```

在二进制数据中用到的整数，每一个都必须要在0到255之间。由字符序列组成的二进制数据等同于由其每一个字符的ASCII编码组成的二进制数据，即<<"cat">>等同于<<99,97,116>>。

对于字符串来说，如果其二进制数据是可打印的字符串，那么在显示时，shell会将其显示为字符串的形式，否则，就会显示成一串整数。

可以通过BIF来构造二进制数据或者从中提取数据，也可以通过比特语法来完成这一过程（参考5.3节）。本节我们只关注BIF。

78

```
@spec func(Arg1,..., Argn) -> Val
```

@spec是什么意思？

这个例子是一个Erlang类型文档标记。在Erlang社区中，常用这个标记在文档中描述函数的参数及其返回类型。这个格式相当清晰，以至于无需额外说明，当然，对于那些想要刨根问底了解全部细节的读者来说，可以参照附录A。

操纵二进制数据的 BIF

下面这些BIF用于操纵二进制数据。

```
@spec list_to_binary(IoList) -> binary()
```

list_to_binary将IoList中的所有东西转换为一个二进制数据。这里的IoList是一个列表，其中的元素可以是0～255的整数、二进制数据或者另外一个IoList：

```
1> Bin1 = <<1,2,3>>.
<<1,2,3>>
2> Bin2 = <<4,5>>.
<<4,5>>
3> Bin3 = <<6>>.
<<6>>
4> list_to_binary([Bin1,1,[2,3,Bin2],4|Bin3]).
<<1,2,3,1,2,3,4,5,4,6>>
```

```
@spec split_binary(Bin,Pos)->{Bin1,Bin2}
```

这个函数在Pos指定的位置将二进制数据Bin分割为两个部分：

```
1> split_binary(<<1,2,3,4,5,6,7,8,9,10>>, 3).
{<<1,2,3>>,<<4,5,6,7,8,9,10>>}
```

```
@spec term_to_binary(Term) -> Bin
```

这个函数可以将任何Erlang值转化为二进制数据。

由term_to_binary转换得来的二进制数据是以所谓的“外部数据格式”存储的。这些数据可以用于文件存储、网络传输等，而且在转换之后我们还可以从这些二进制数据中还原出原始的
79 数据项[①]。这个函数在需要对复杂的数据结构进行文件存储和网络传输时极为有用。

```
@spec binary_to_term(Bin) -> Term
```

这是term_to_binary的逆函数：

```
1> B = term_to_binary({binaries,"are", useful}).
<<131,104,3,100,0,8,98,105,110,97,114,105,101,115,107,
0,3,97,114,101,100,0,6,117,115,101,102,117,108>>
2> binary_to_term(B).
{binaries,"are",useful}
```

```
@spec size(Bin)->Int
```

这个函数返回二进制数据的字节长度。

```
1> size(<<1,2,3,4,5>>).
5
```

5.3　比特语法

比特语法是模式匹配的一种扩展。经常用于对二进制数据中的单个比特位或者比特位串进行封包和解包。当编写一些底层代码时，常会需要对比特级别的二进制数据进行封包和解包，此时你会发现比特语法会使事情变得无比便捷。比特语法是针对协议编程而设计的（这是Erlang的看家本领）。它可以产生极为高效的代码，用来处理协议数据的封包和解包。

假定我们有3个变量——X、Y和Z。我们希望把它们封装在一个16 bit字长的内存区域，也就是变量M中。X要在结果中占3 bit，Y占7 bit，Z占6 bit。在大多数的语言中这都意味着你将涉及移位、掩码等一系列乏味而凌乱的底层操作。但是在Erlang中，你只需要这么写:

```
M = <<X:3, Y:7, Z:6>>
```

相当简单！

这只是开始，更完整的比特语法比这要稍微复杂一点，因此，有必要循序渐进地学习它。首先会看到一些简单的代码，用来将RGB色彩数据通过封包解包存放在一个16 bit长的字中。随后深入学习比特语法表达式的细节。最后学习3个源自真实代码中的案例，它们都用到了比特语法。

5.3.1　16 bit 色彩的封包与解包

先来看一个最简单的例子。如果想要表示一个16 bit的RGB颜色，我们决定为红色通道分配5
80 bit，绿色通道分配6 bit，蓝色通道分配5 bit（为绿色通道多分配了1 bit是因为人类的眼睛对绿色更为敏感）。

可以像下面一样，创建一个名为Mem的16 bit内存块，用以存放一个RGB三元组。

① 类似于Java的序列化和反序列化。——译者注

```
1> Red = 2.
2
2> Green = 61.
61
3> Blue = 20.
20
4> Mem = <<Red:5, Green:6, Blue:5>>.
<<23,180>>
```

注意，在第4行我们创建了2字节的二进制数据来存放一个16 bit长的字。shell会将这个变量打印为<<23,180>>。

封装这个内存块，只需编写这样的表达式<<Red:5, Green:6, Blue:5>>。而要对这个字进行解包，则可以写一个这样的匹配模式：

```
5> <<R1:5, G1:6, B1:5>> = Mem.
<<23,180>>
6> R1.
2
7> G1.
61
8> B1.
20
```

5.3.2 比特语法表达式

比特语法表达式有如下这些形式：

```
<<>>
<<E1, E2, ..., En>>
```

每一个元素Ei代表了二进制数据中的一个单独区块。每一个元素Ei都可以是如下4种形式的一种：

```
Ei = Value |
     Value:Size |
     Value/TypeSpecifierList |
     Value:Size/TypeSpecifierList
```

无论你用哪种形式，二进制数据中的总比特数必须要恰好能被8整除（这是因为二进制数据中所包含的每个字节都是8 bit，因此二进制数据没有办法表达一个非8倍数长度的比特串）。 81

当创建一个二进制数据时，Value必须是一个绑定变量、文本串或者一个返回值为整数、浮点数或者二进制数据的表达式。而在把比特语法用来做模式匹配操作时，Value则既可以是绑定的变量，也可以是自由的变量，其类型也可以为整数、文本串、浮点数、或者二进制数据。

Size必须是一个返回值为整数的表达式。在模式匹配中，Size必须为一个整型或者是一个整型的绑定变量。Size不能是自由变量。

Size的值指明了单元区块的长度（我们稍候会再讨论），而默认值则依赖于具体的类型（详见下文）。对于整型长度为8，浮点型为64，二进制数据则为其本身的长度。在模式匹配中，这些默认值只对毗邻的元素有效。所有其他的二进制数据元素在模式匹配中必须指明具体的长度。

TypeSpecifierList是一个由连字符间隔的形如End-Sign-Type-Unit的列表。每一项的前置

项都可忽略，而且没有任何顺序要求。如果省略一项，那么这个项的值就使用它的默认值。

Type SpecifierList列表的项有以下值。

```
@type End = big | little | native
```

（@type也是Erlang类型表示法的一部分，具体参见附录A）。

这一项指明了计算机系统的字节序。native是运行时决定的字节序，依赖于计算机的CPU。默认字节序为big。唯一会用到这个项的情形是处理整数和二进制数据之间的封包和解包工作。在不同字节序的机器之间进行整数和二进制数据的封包解包时，你需要注意使用正确的字节序。

提示　有一些罕见的情形，你必须要了解内部的实际情况，此时做一些实验可能会大有帮助。为了确保你做的事情是正确的，可以试试下面的这些shell命令：

```
1> {<<16#12345678:32/big>>,<<16#12345678:32/little>>,
    <<16#12345678:32/native>>,<<16#12345678:32>>}.
{<<18,52,86,120>>,<<120,86,52,18>>,
 <<120,86,52,18>>,<<18,52,86,120>>}
```

82　输出结果明确地显示了使用比特语法时整型是如何被封装在一个二进制数据中的。

如果你仍然无法消除疑虑，那么可以用term_to_binary和binary_to_term，它们会正确地处理整数的封包和解包。因此，你可以在一个big-endian的计算机上创建一个包含整数的元组。然后使用term_to_binary将这个元组转换为二进制数据，并将这个二进制数据传送到一个little-endian机器上。在little-endian机器上，当执行binary_to_term时，元组中的所有整数都会恢复成正确值。

```
@type Sign = signed | unsigned
```

这个参数仅仅用于模式匹配中，默认是unsigned。

```
@type Type = integer | float | binary
```

默认是integer。

```
@type Unit = 1 | 2 | ... 255
```

整个区块长度为Size×Unit bit。整个区块的长度必须大于或等于0而且还必须是8的倍数。Unit的默认值依赖于Type，如果Type是integer或者float则为1，Type为binary则为8。

如果你觉得比特语法的样子有些令人畏惧，请别慌张。写出正确的比特语法是有点小窍门的。熟悉它的最好办法就是在shell中尝试你需要的模式直到能得到正确的值，然后把它们复制粘贴到程序中。我就是这么做的。

5.3.3　高级比特语法样例

学习比特语法过程相当痛苦，但是收获也与之相称。本节里有3个来自真实世界的样例，这些代码都是从实际的程序中摘抄下来的。这3个程序如下。

- ❑ 在MEPG数据中查找同步帧。
- ❑ 解包COFF数据。
- ❑ 从一个IPv4数据报中解析头。

1. 在MEPG数据中查找同步帧

假定我们想写一个程序去操纵MPEG音频数据。我们可能想用Erlang去编写一个流媒体服务器或者提取MPEG音频流中描述内容的数据标签。为了完成这个任务，我们需要去识别和同步MPEG流中的数据帧。

83

MPEG音频数据是由若干个帧构成的。每一帧都有自己的头，紧随其后的就是语音信息——这里不涉及文件头，理论上，你可以把一个MPEG文件分割成若干段，然后随意播放其中一段。任何读取MPEG流的软件都是先找到头帧然后同步MPEG数据。

MPEG头以如下这样的同步帧开始，这个同步帧由连贯的11 bit全1数据组成，后跟描述后续数据的信息：

```
AAAAAAAA AAABBCCD EEEEFFGH IIJJKLMM
```

其中：

AAAAAAAAAAA，是同步字（共11 bit，全1）；

BB，2 bit是MPEG语音的版本号；

CC，2 bit是层描述符；

D，1 bit是保护位；

等等。

其余的比特就不介绍了，我们并不关心这些位的具体意义。一般来说，只要知道了A到M的信息，我们就可以计算整个MPEG帧的长度。

为了要找到同步点，先假定我们正确地定位到一个MPEG帧的开始位置。我们用在这个位置找到的信息来计算帧的长度。如果我们扫描到的是一个无意义的位置，此时帧的长度信息就完全不对了。假设我们从一个帧的头开始，并知道这个帧的长度，那么可以跳到下一帧的开始来检查是否又读到了一个新的MPEG头帧。

这个查找同步点的过程中，先假定我们正确地定位到了MPEG头的开始，然后试图去计算帧的长度。此时会出现下面几种情况之一。

- 我们的假定没有问题，那么当我们跳过这个帧的长度所表明的数据后，就可以找到下一个MPEG头。
- 我们的假定是错的，那么我们要么没有定位到标记头起始的11 bit连贯的全1串上，要么就是这个头的格式错误，因此无法计算这个帧的长度。
- 我们的假定是错的，但是我们定位在了两个恰巧看上去像头的字节上，而这两个字节实际上是音频数据。在这种情况下，我们可以计算帧的长度，但是当跳过那么长的数据时，将无法找到下一个新头。

84

为了确定这一点，我们需要找到3个连贯的头。下面就是同步代码：

mp3_sync.erl

```
find_sync(Bin, N) ->
    case is_header(N, Bin) of
        {ok, Len1, _} ->
```

```
            case is_header(N + Len1, Bin) of
                {ok, Len2, _} ->
                    case is_header(N + Len1 + Len2, Bin) of
                        {ok, _, _} ->
                            {ok, N};
                        error ->
                            find_sync(Bin, N+1)
                    end;
                error ->
                    find_sync(Bin, N+1)
            end;
        error ->
            find_sync(Bin, N+1)
    end.
```

find_sync试图找到3个连贯的MPEG头帧。如果Bin中的字节N是一个头帧的起始，那么is_header(N,Bin)将会返回{ok,Length,Info}。如果is_header返回错误信息，那么N就无法定位到正确的帧的起始位置。我们可以在shell中做一个快速的测试来确保这个程序是正确的：

```
1> {ok, Bin} = file:read_file("/home/joe/music/mymusic.mp3").
{ok,<<73,68,51,3,0,0,0,0,33,22,84,73,84,50,0,0,0,28, ...>>
2> mp3_sync:find_sync(Bin, 1).
{ok,4256}
```

这里使用了file:read_file，它把整个文件读入到一个二进制数据中（参见13.2节）。现在来看is_header：

mp3_sync.erl

```
is_header(N, Bin) ->
    unpack_header(get_word(N, Bin)).

get_word(N, Bin) ->
    {_,<<C:4/binary,_/binary>>} = split_binary(Bin, N),
    C.

unpack_header(X) ->
    try decode_header(X)
    catch
        _:_ -> error
    end.
```


这里要稍微复杂一点。首先从数据中读取了32 bit用于分析（这是通过get_word函数实现的），然后使用decode_header对头进行解包。现在如果参数并不是一个头帧起始的话，decode_header函数会导致崩溃（内部调用了函数exit/1）。为了捕获所有错误，我们把decode_header包在一个try...catch语句中（关于异常捕获详情请参考4.1节）。这个语句也会捕获到由framelength/4函数中错误的代码引起的异常。而decode_header函数则是这个例子的重头戏：

mp3_sync.erl

```
decode_header(<<2#11111111111:11,B:2,C:2,_D:1,E:4,F:2,G:1,Bits:9>>) ->
    Vsn = case B of
              0 -> {2,5};
              1 -> exit(badVsn);
              2 -> 2;
              3 -> 1
          end,
    Layer = case C of
                0 -> exit(badLayer);
                1 -> 3;
                2 -> 2;
                3 -> 1
            end,
    %% Protection = D,
    BitRate = bitrate(Vsn, Layer, E) * 1000,
    SampleRate = samplerate(Vsn, F),
    Padding = G,
    FrameLength = framelength(Layer, BitRate, SampleRate, Padding),
    if
        FrameLength < 21 ->
            exit(frameSize);
        true ->
            {ok, FrameLength, {Layer,BitRate,SampleRate,Vsn,Bits}}
    end;
decode_header(_) ->
    exit(badHeader).
```

这个令人惊异的表达式的奥秘其实都在代码的第一行里。

```
decode_header(<<2#11111111111:11,B:2,C:2,_D:1,E:4,F:2,G:1,Bits:9>>) ->
```

这个模式匹配11 bit连贯的1[①]，2 bit的B，2 bit的C，等等。注意这段代码严格遵守前面内容中所给出的MPEG头帧的格式标准。这段代码的优雅性恐怕是难以超越的，它不但极为优美，而且效率也非常高。Erlang编译器会把比特语法转换成高度优化的代码，并以极高的效率来提取字段。 86

2. 解包COFF数据

几年前，我决定写一个程序使独立的Erlang程序可以在Windows上运行。我期望可以在任何机器上生成一个可以运行Erlang的Windows可执行文件。要实现它必须要弄懂和操纵由微软COFF（Common Object File Format，通用对象文件格式）编码的文件。要发掘COFF的细节信息是一件极为麻烦的事情，不过面向C++程序员公开的文档中描述了几个API。C++程序使用DWORD、LONG、WORD、BYTE这样的类型声明（这些类型声明对于有Windows内核编程经验的程序员来说是很熟悉的）。

我们所需要的数据结构都有公开的文档，但仅仅是针对C或C++程序员的。下面是一个典型的C类型定义：

① 2#11111111111是一个二进制数。

```
typedef struct _IMAGE_RESOURCE_DIRECTORY {
   DWORD Characteristics;
   DWORD TimeDateStamp;
   WORD  MajorVersion;
   WORD  MinorVersion;
   WORD  NumberOfNamedEntries;
   WORD  NumberOfIdEntries;
} IMAGE_RESOURCE_DIRECTORY, *PIMAGE_RESOURCE_DIRECTORY;
```

为了编写Erlang代码，我首先定义了4个宏，它们必须包含在Erlang源代码文件中：

```
-define(DWORD, 32/unsigned-little-integer).
-define(LONG,  32/unsigned-little-integer).
-define(WORD,  16/unsigned-little-integer).
-define(BYTE,  8/unsigned-little-integer).
```

说明　5.4节将详细阐述宏。为了要扩展这些宏，我们使用语法?DWORD、?LONG等。例如32/unsigned-little-integer会逐字地扩展宏?DWORD。

这些宏有意地与C中对应的类型保持一致。有了这些宏，就可以很简单地编写代码把镜像数据解包到二进制中：

```
unpack_image_resource_directory(Dir) ->
    <<Characteristics      : ?DWORD,
      TimeDateStamp        : ?DWORD,
      MajorVersion         : ?WORD,
      MinorVersion         : ?WORD,
      NumberOfNamedEntries : ?WORD,
      NumberOfIdEntries    : ?WORD, _/binary>> = Dir,
      ...
```

87

对比Erlang代码和C代码，你会发现它们极为相似。因此小心地处理宏的名字和Erlang代码的布局，就可以缩小C代码和Erlang代码之间的语法鸿沟。这使得程序更加简洁易懂，减少了出错的可能性。

下一步是要从Characteristics中解析数据等。Characteristics是一个32 bit的字，其数据是一系列的标志位。用比特语法来解析显得极其简单，我们可以这样写:

```
<<ImageFileRelocsStripped:1, ImageFileExecutableImage:1, ...>> =
<<Characteristics:32>>
```

代码<<Characteristics:32>>将一个整型的Characteristics转换为一个32 bit长的二进制数据。然后下面的代码从中解析出所需的数据并存入ImageFileRelocsStripped、ImageFileExecutableImage等变量中去。

```
<<ImageFileRelocs Stripped:1, ImageFileExecutableImage:1,...>>=...
```

我们仍然采用与Windows API一致的变量名，这使得Erlang代码与规范本身的差异缩减少到了最小。

使用这些宏来解析COFF格式中的数据——唔，我不能说这么做“简单”——但至少这么做

是可能的，而且代码也相当简洁明了。

3. 从一个IPv4数据报中解析头

这个例子展示了用模式匹配操作来解析一个IPv4（Internet Protocol Version4）数据报：

```
-define(IP_VERSION, 4).
-define(IP_MIN_HDR_LEN, 5).

...
DgramSize = size(Dgram),
case Dgram of
  <<?IP_VERSION:4, HLen:4, SrvcType:8, TotLen:16,
    ID:16, Flgs:3, FragOff:13,
    TTL:8, Proto:8, HdrChkSum:16,
    SrcIP:32,
    DestIP:32, RestDgram/binary>> when HLen >= 5, 4*HLen =< DgramSize ->
      OptsLen = 4*(HLen - ?IP_MIN_HDR_LEN),
      <<Opts:OptsLen/binary,Data/binary>> = RestDgram,
      ...
```

88

这段代码在一个模式匹配表达式里匹配了IP数据报。这个模式比较复杂，分为3行。它向我们展示了在那些长度并不是字节整数倍的情况下，如何方便地提取出数据（例如Flgs和FragOff字段分别是3 bit和13 bit长）。在这个模式匹配到了IP数据报后，数据报的头和数据部分就在第二个模式匹配操作中被提取出来。

5.4 小问题集锦

至此为止，我们已经讲述了Erlang顺序型编程中的主要话题。剩下的问题都是一些需要了解但是又难以归类的零碎问题。下面的话题，其排列顺序并没有什么逻辑关系。这些话题包括以下这些。

- apply：当动态地计算函数名和模块名时，如何根据其名称和参数去计算它的值。
- 属性：Erlang模块属性的含义和语法。
- 块表达式：使用begin和end的表达式。
- 布尔表达式：所有的布尔值表达式。
- 字符集：Erlang采用哪种字符集？
- 注释：注释语法。
- epp：Erlang预处理器。
- 转义符：在字符串和原子中使用的转义符的语法。
- 表达式和表达式序列：确切地说，什么是一个表达式？
- 函数引用：如何去引用一个函数？
- 包含文件：如何在编译时去包含文件？
- 列表操作符：++和--。
- 宏：Erlang宏处理器。
- 在模式中使用匹配操作符：如何在模式中使用匹配操作符=。

- 数值：数值语法。
- 操作符优先级：Erlang操作符的优先级和结合性。
- 进程字典：每一个Erlang进程都有一个可供破坏性赋值的本地存储区域，它在某些情况下很有用。
- 引用：引用是独一无二的符号。
- 短路布尔表达式：不会完整求值的布尔表达式。
- 比较表达式：所有有关比较操作的操作符，以及比较的文法顺序。
- 下划线变量：会被编译器进行特殊处理的变量。

89

5.4.1 apply

BIF apply(Mod, Func, [Arg1, Arg2, ..., ArgN])将Mod模块中的Func函数应用到参数Arg1,Arg2...ArgN上去。它和下面这种调用方式是等价的：

```
Mod:Func(Arg1, Arg2, ..., ArgN)
```

apply的功能是让你向一个模块中的某个函数传递参数并调用此函数。它与直接函数调用的区别在于，模块名和函数名可以动态计算。

所有的Erlang BIF都能使用apply来调用。假定这些函数都属于模块erlang。那么要构建一个BIF的动态调用，我们可以这么写：

```
1> apply(erlang, atom_to_list, [hello]).
"hello"
```

警告 如果可能，要尽量避免使用apply。如果某个函数的参数个数是已知的，那么形如M:F(Arg1, Arg2, ... ArgN)的函数调用要优于apply。当通过apply进行调用时，很多分析工具将无法分析出其中的工作细节，而且编译器也肯定不能对其进行优化。因此，要尽量少用apply，除非没有其他的方法。

5.4.2 属性

模块属性的语法是-AtomTag(...)[①]。它经常被用于定义文件的属性。Erlang的模块属性分两种，预定义属性和用户定义属性。

1. 预定义模块属性

90

下面这些模块属性都有预定义的特殊含义，必须放在所有函数定义之前。

```
-module(modname).
```

这是模块声明，其中的modname必须是一个原子。这个属性必须是文件中第一个属性。按惯例modname的代码都应存储在一个名为modename.erl的文件中。如果不那么做，那么代码的自动加载就会出错。更多的信息请参见附录E.4节。

① -record(...)和-include(...)的语法很相似，但我们不把它们当作模块属性。

-import(Mod, [Name1/Arity1, Name2/Arity2,...]).

这个属性指定编译器从Mod模块中导入参数为Arity1且名为Name1的函数。

一旦你从模块中导入某个函数，那么调用这个函数时就不需要特别指明模块名。例如：

```
-module(abc).
-import(lists, [map/2]).

f(L) ->
    L1 = map(fun(X) -> 2*X end, L),
    lists:sum(L1)
```

map的调用不需要指定模块名，而对sum的调用则一定要在函数调用中包含模块名。

-export([Name1/Arity1, Name2/Arity2, ...]).

这个属性指定从当前模块中导出Name1/Arity1、Name2/Arity2等函数。注意，只有被导出的函数才能在模块外部被调用。例如：

abc.erl

```
-module(abc).
-export([a/2, b/1]).

a(X, Y) -> c(X) + a(Y).
a(X) -> 2 * X.
b(X) -> X * X.
c(X) -> 3 * X.
```

导出声明意味着只有a/2和b/1可以在模块abc之外被调用。因此，像abc:a(5)这样的调用就会导致错误，因为a/2并没有从模块中导出。

```
1> abc:a(1,2).
7
2> abc:b(12).
144
3> abc:a(5).
** exited: {undef,[{abc,a,[5]},
                   {erl_eval,do_apply,5},
                   {shell,exprs,6},
                   {shell,eval_loop,3}]} ="session">
```

-compile(Options)

向编译器选项列表中添加Options。Options是一个编译器选项或者一个编译器选项列表（这些都列在了compile模块手册中）。

说明 在调试程序时我们经常会用到编译选项“-compile(export_all).”，这会导出当前模块的所有函数而不需要显式地使用-export注解。

-vsn(Version)

指定一个模块的版本。Version可以是任何形式的文字项，Version值没有任何具体的语法形式和意义，但是它可以用来分析程序和文档。

91

2. 用户定义属性

用户定义模块属性遵循如下的语法规则：

```
-SomeTag(Value).
```

Sometag必须是一个原子，Value必须是一个文字项。模块属性的值被编译进模块并可以在运行时被提取出来。下面是一个简单的样例：

attrs.erl

```
-module(attrs).
-vsn(1234).
-author({joe,armstrong}).
-purpose("example of attributes").
-export([fac/1]).

fac(1) -> 1;
fac(N) -> N * fac(N-1).
```

```
1> attrs:module_info().
[{exports,[{fac,1},{module_info,0},{module_info,1}]},
{imports,[]},
{attributes,[{vsn,[1234]},
             {author,[{joe,armstrong}]},
             {purpose,"example of attributes"}]},
{compile,[{options,[{cwd,"/home/joe/2006/book/JAERLANG/Book/code"},
                    {outdir,"/home/joe/2006/book/JAERLANG/Book/code"}]},
          {version,"4.4.3"},
          {time,{2007,2,21,19,23,48}},
          {source,"/home/joe/2006/book/JAERLANG/Book/code/attrs.erl"}]}]
2> attrs:module_info(attributes).
[{vsn,[1234]},{author,[{joe,armstrong}]},{purpose,"example of attributes"}]
3> beam_lib:chunks("attrs.beam",[attributes]).
{ok,{attrs,[{attributes,[{author,[{joe,armstrong}]},
                         {purpose,"example of attributes"},
                         {vsn,[1234]}]}]}}
```

92

源代码文件中的用户定义属性会以一个子元组{attributes,...}的形式重现。元组{compile,...}包含了由编译器加入的编译信息，值{version,"4.4.3"}是编译器的版本，不要将它与在模块属性中定义的vsn标签混淆。在前面的样例中，attrs:module_info()返回了一个与编译模块相关的所有元数据的属性列表。attrs:module_info(attributes)[①]则返回与文件相关的特定属性的列表。

注意，函数module_info/0和module_info/1是每次模块在编译时自动创建的。

第2行和第3行的输出不是很容易读懂。为了提高可读性，我们可以写一个小函数去提取特定的属性。这个函数非常的简单：

① 其他的一些参数是exports、imports和compile。

```
4> extract:attribute("attrs.beam", author).
[{joe,armstrong}]
```

我们可以这样来调用它:

extract.erl

```
-module(extract).
-export([attribute/2]).

attribute(File, Key) ->
    case beam_lib:chunks(File,[attributes]) of
        {ok, {_Module, [{attributes,L}]}} ->
            case lookup(Key, L) of
                {ok, Val} ->
                    Val;
                error ->
                    exit(badAttribute)
            end;
        _ ->
            exit(badFile)
    end.

lookup(Key, [{Key,Val}|_]) -> {ok, Val};
lookup(Key, [_|T])         -> lookup(Key, T);
lookup(_, [])              -> error.
```

93

要运行attrs:module_info，我们必须加载模块attrs的beam代码。模块beam_lib包含了一系列无须加载代码就能分析模块的函数。样例extract.erl就是在没有为模块加载代码的情况下使用beam_lib:chunks来提取属性的。

5.4.3 块表达式

```
begin
   Expr1,
   ...,
   ExprN
end
```

可以使用块表达式来把一串表达式组织成一个类似子句的实体。begin...end块的值就是块中最后一个表达式的值。

当代码的某一处的语法只允许使用单个表达式而你却需要在这个地方使用一串表达式时，就可以使用块表达式。

5.4.4 布尔类型

Erlang中没有独立的布尔类型，原子true和false取而代之被赋予了特殊的布尔语义，它们通常作为布尔符号使用。

5.4.5　布尔表达式

下面是4种布尔表达式。

- not B1：逻辑非。
- B1 and B2：逻辑与。
- B1 or B2：逻辑或。
- B1 xor B2：逻辑异或。

在所有的这些表达式中，B1和B2必须为布尔符号或者是可以求出布尔值的布尔表达式。例如：

```
1> not true.
false.
2> true and false.
false
3> true or false.
true
4> (2 > 1) or (3 > 4).
true
```

94

强制函数返回布尔值

有时，我们要编写一些函数，它们只可能返回两个原子值的其中之一。这种情况下，最好的办法就是保证它们返回一个布尔型。同时，把函数命名为有布尔型返回也是一个好主意。

例如，假设编写一个表达文件状态的程序，我们可能会发现，其中有一个返回文件是开启还是关闭的函数file_state。当编写这个函数时，就应该考虑将其重命名，并确保它能返回一个布尔型。基于这样的考虑，我们可以重写代码，使用is_file_open()这样的名字，让其返回true或者false。

我们为何要这样做？

答案很简单，因为标准库中有大量的函数返回布尔型。因此，如果我们能够保证所有的返回结果都是两者之一的原子值，那么就能让它们与标准库函数一起很好地工作。

5.4.6　字符集

Erlang源代码文件默认以ISO-8859-1（Latin-1）字符集进行编码。这意味着Latin-1的可打印字符可以无需转义符而独立使用。

Erlang内部没有字符数据类型。字符串不是一个真正的类型，而是用一串整数列表来表示。Unicode字符串毫无疑问也可以使用一串整数列表来表示，不过从Erlang整数列表中解析和生成Unicode字符仍然存在一定的限制。

5.4.7　注释

Erlang的注释以一个百分号（%）开头，表示这一行的所有内容都是注释。Erlang没有块注释语句。

说明 你经常会在样例程序中看到双百分号（%%），双百分号在emacs erlang-mode中会被识别，从而可以启动注释行的自动缩排功能。

95

```
% This is a comment
my_function(Arg1, Arg2) ->
    case f(Arg1) of
        {yes, X} ->   % it worked
          ..
```

5.4.8 epp

在Erlang模块被编译之前，首先会被名为epp的Erlang预处理器进行自动处理。这个预处理器会扩展任何存在于源文件中的宏，并且插入任何必需的包含文件。

通常，你无须关心预处理器的输出，但在一些特殊情况下（例如，你要调试一个有错误的宏），你可能希望能够保存预处理器的输出。通过命令`compile:file(M,['p'])`可以将预处理器的输出保存在一个文件中。这个命令编译`M.erl`文件中的所有代码，然后在`M.P`文件中产生一个列表，存放所有经过扩展的宏和所有已经插入的包含文件。

5.4.9 转义符

在字符串和使用括号的原子中，你可以使用转义符来输入任何不可打印的字符。Erlang中所有的转义符序列都列在下面的表5-1中。

表5-1 转义符

转 义 符	含 义	整 数 值
\b	退格	8
\d	删除	127
\e	转义	27
\f	换页	12
\n	新行	10
\r	换行	13
\s	空格	32
\t	制表符	9
\v	纵向制表符	11
\NNN \NN \N	八进制码（N是0～7）	
\^a..\^z or \^A..\^Z	Ctrl+A到Ctrl+Z	1～26
\'	单引号	39
\"	双引号	34
\\	反斜杠	92
\C	C的ASCII码（C是一个字符）	（一个整数）

下面我们在shell中用几个样例来演示这些语法是如何工作的。（注意，在一个格式化串中，~w的意思是对一个列表不做任何修饰的打印原始结果。）

```
%% Control characters
1> io:format("~w~n", ["\b\d\e\f\n\r\s\t\v"]).
[8,127,27,12,10,13,32,9,11]
ok
%% Octal characters in a string
3> io:format("~w~n", ["\123\12\1"]).
[83,10,1]
ok
%% Quotes and escapes in a string
4> io:format("~w~n", ["\'\"\\"]).
[39,34,92]
ok
%% Character codes
5> io:format("~w~n", ["\a\z\A\Z"]).
[97,122,65,90]
ok
```

96

5.4.10　表达式和表达式序列

在Erlang中，任何可以被求出值的东西都被称作表达式。这意味着catch、if、try...catch等都是表达式。而类似记录和模块属性等，这些不能被求值的都不是表达式。

表达式序列是一系列由逗号分开的表达式。这些表达式都应该紧接放置于箭头(->)之后。表达式序列E1,E2,...,En的值被定义为序列中最后一个表达式的值。[①]这样就可以对E1、E2等表达式的计算结果绑定值。

5.4.11　函数引用

我们经常会想要引用一个当前模块或者外部模块所定义的函数，可以使用下面这些语法。

fun LocaFunc/Arity

97

这个语句用来引用一个当前模块中参数目为Arity、函数名为LocalFunc的本地函数。

fun Mod:RemoteFunc/Arity

这个表达式用来引用Mod模块中参数目为Arity、函数名为RemoteFunc的外部函数。

下面是一个在当前模块中使用函数引用的样例：

```
-module(x1).
-export([square/1, ...]).

square(X) -> X * X.
...
double(L) -> lists:map(fun square/1, L).
```

如果想要引用一个外部模块中的函数，我们可以像下面的例子这样做：

```
-module(x2).

...
double(L) -> lists:map(fun x1:square/1, L).
```

① 等同于LISP中的progn。

fun x1:square/1意味着函数sqaure/1在模块x1中。

5.4.12 包含文件

用如下的语法包含文件。

```
-include(Filename).
```

在Erlang中，这个语法可以包含扩展名为.hrl的包含文件。FileName应该包含一个绝对或者相对路径以便预处理器能够定位到相应的文件。使用下面的语法可以引入库中的包含文件：

```
-include_lib(Name).
```

例如:

```
-include_lib("kernel/include/file.hrl").
```

在这种情况下，Erlang编译器会找到对应的包含文件（在前面的样例中，引用的是kernel，即定义了这个包含文件的应用程序）。

包含文件通常包含了记录定义。如果需要在多个模块之间共享记录的定义，那么这些共用的记录就应该定义在包含文件中，而所有需要这些定义的模块都必须引入这个包含文件。 98

5.4.13 列表操作符++和--

++和--是对列表进行添加和删除的中缀操作符。

A++B意味着把A和B加起来(实际上是B附加到A)。

A--B从列表A中删除列表B。删除的意思是B中的所有元素都要从A中删除。注意，如果符号X在B中出现K次，那么在A中只会按顺序删除K个X。

下面是一些样例：

```
1> [1,2,3] ++ [4,5,6].
[1,2,3,4,5,6]
2> [a,b,c,1,d,e,1,x,y,1] -- [1].
[a,b,c,d,e,1,x,y,1]
3> [a,b,c,1,d,e,1,x,y,1] -- [1,1].
[a,b,c,d,e,x,y,1]
4> [a,b,c,1,d,e,1,x,y,1] -- [1,1,1].
[a,b,c,d,e,x,y]
5> [a,b,c,1,d,e,1,x,y,1] -- [1,1,1,1].
[a,b,c,d,e,x,y]
```

模式匹配之中的++

++也可以用在模式匹配中。当匹配字符串时，我们可以这样写模式:

```
f("begin" ++ T) -> ...
f("end" ++ T) -> ...
...
```

第一个子句的模式会被扩展为[$b,$e,$g,$i,$n|T]。

5.4.14　宏

在Erlang中，宏的语法是这样的：

```
-define(Constant, Replacement).
-define(Func(Var1, Var2,.., Var), Replacement).
```

当Erlang预处理器epp遇到一个形如?MacroName的表达式时，这个宏就会被预处理器扩展。在宏定义中出现的变量会匹配到宏调用处的完整形式。

```
-define(macro1(X, Y), {a, X, Y}).

foo(A) ->
    ?macro1(A+10, b)
```

99

这个宏扩展后成为：

```
foo(A) ->
    {a,A+10,b}.
```

除此之外，还有一批预定义宏可以提供当前模块的信息。它们是如下3个：

- ?FILE扩展为当前文件名；
- ?MODULE扩展为当前模块名；
- ?LINE扩展为当前行号。

宏的流程控制

在一个宏定义的内部，还支持流程指令。你可以使用它们去制定一个宏内部的控制流程。

`-undef(Macro).`

取消该宏定义，在这个语句后不能再调用这个宏。

`-ifdef(Macro).`

只有Macro被定义后，才对该行以下的代码进行运算。

`-ifndef(Macro).`

只有在不定义Macro的情况下，才对该行以下的代码进行运算。

`-else.`

只能在ifdef或ifndef之后出现。如果条件为false，那么该语句后的代码才会被执行。

`-endif.`

标记ifdef或ifndef语句的结束。

条件宏必须是合理嵌套的，它们通常这样出现：

```
-ifdef(debug).
-define(...).
-else.
-define(...).
-endif.
```

我们可以使用这些宏来定义一个名为TRACE的宏，如下：

m1.erl

```
-module(m1).
```

```
-export([start/0]).

-ifdef(debug).
-define(TRACE(X), io:format("TRACE ~p:~p ~p~n",[?MODULE, ?LINE, X])).
-else.
-define(TRACE(X), void).
-endif.
start() ->  loop(5).

loop(0) ->
    void;
loop(N) ->
    ?TRACE(N),
    loop(N-1).
```

100

说明 io:format(String, [Args])会根据String中的格式信息在Erlang shell中打印[Args]中的变量。格式化代码具有一个(~)前缀，~p是完整打印的缩写，~n产生一个新行。[①]

这段代码在编译时需要开启或者关闭trace宏，我们可以在c/2函数中使用一个附加参数，就像这样：

```
1> c(m1, {d, debug}).
{ok,m1}
2> m1:start().
TRACE m1:15 5
TRACE m1:15 4
TRACE m1:15 3
TRACE m1:15 2
TRACE m1:15 1
void
```

c(m1,Options)提供了一个向编译器传输选项的途径。{d,debug}把调试标志设为true以便在宏定义的-ifdef（debug）片段中能够被识别。

当关闭这个宏时，trace宏就被扩展为原子void。这里的名字选择无关紧要只是告诉我们没有人关心宏的值。

5.4.15 在模式中使用匹配操作符

假定有如下的代码：

```
Line 1   func1([{tag1, A, B}|T]) ->
             ...
             ... f(..., {tag1, A, B}, ...)
             ...
```

在第1行中，模式匹配到元组{tag1,A,B}，而在第3行又以{tag1,A,B}为参数调用函数f。当这么做时，系统会重新创建一个元组{tag1,A,B}。更加高效同时更少出错的方法是把这个模式

101

① io:format可以识别大量的格式化选项。要了解其中的详细内容，请参见13.3节。

赋值到一个临时变量Z，然后将这个临时变量传入f，如下：

```
func1([{tag1, A, B}=Z|T]) ->
    ...
    ... f(... Z, ...)
    ...
```

这个匹配操作符可以在模式的任何地方使用。如果编写下面的代码，需要重新创建两个元组：

```
func1([{tag, {one, A}, B}|T]) ->
   ...
   ...  f(..., {tag, {one,A}, B}, ...),
   ...  g(..., {one, A}), ...)
   ...
```

此时可以引入两个新变量——Z1和Z2，程序可以修改成这样：

```
func1([{tag, {one, A}=Z1, B}=Z2|T]) ->
   ...
   ...  f(..., Z2, ...),
   ...  g(..., Z1, ...),
   ...
```

5.4.16　数值类型

Erlang中的数值类型包括：整型和浮点型。

1．整型

Erlang中整型数据的计算是精确的，整型变量可代表的数据长度仅受限于可用的内存。

可以用下面3种不同的语法来表达一个整型数值。

(1) 传统语法：这种写法是我们最为熟悉的，比如12、12375、-23427都是整数。

(2) K进制整数：不以10为进制的整数可以用语法K#Digits来表示。因此我们可以写一个二进制数2#00101010或者一个16进制数16#af6bfa23。对于大于10进制的整数，用字符abc...（或ABC...）来表示数值10、11、12等。这种语法所能表示的最高进制为36进制。

102

(3) $语法：语法$C表示ASCII字符C的整数值，因此$a是97的简写，$1是49的简写，等等。紧随$之后，我们还可以使用表5-1中描述的任何转义符。因此$\n就是10，$\^c是3，等等。

下面是一些整数的例子：

```
0 -65 2#010001110 -8#377 16#fe34 16#FE34 36#wow
```

（它们的值分别是0、-65、142、-255、65076、65076和42368。）

2．浮点型

一个浮点数有5个部分：一个可选符号位、一个数值部分、一个小数点、一个小数部分以及一个可选的指数部分。

下面就是一些浮点数的例子：

```
1.0 3.14159 -2.3e+6 23.56E-27
```

经过解析，浮点数在内部会以IEEE 754的64 bit格式表示。Erlang可表示的浮点数的范围是-10^{323}～10^{308}。

5.4.17 操作符优先级

表5-2以优先级的降序和结合顺序展示了Erlang的操作符号。操作符优先级和结合率通常用于决定无括号表达式的求值顺序。

表5-2 操作符优先级

操 作 符	结 合 律
:	
#	
(unary) +, (unary) -, bnot, not	
/, *, div, rem, band, and	左结合
+, –, bor, bxor, bsl, bsr, or, xor	左结合
++, – –	右结合
==, /=, =<, <, >=, >, =:=, =/=	
andalso	
orelse	

带有高优先级的表达式（位于表5-2的上部）会先被求值，然后求值低优先级的表达式。因此，例如要求值3+4*5+6，首先运算子表达式4*5，因为(*)在表中高于(+)。然后我们运算3+20+6，由于（+）是一个左结合操作符，我们把这个算式翻译为(3+20)+6，因此先对3+20求值得到23，最后计算23+6。

用括号表达式来看，3+4*5+6就等于((3+(4*5))+6)，和其他的程序语言一样，用括号来指明优先级要比依赖于优先级规则好。

103

5.4.18 进程字典

Erlang的每一个进程都有自己的私有数据存储，叫做进程字典。进程字典是由一系列键值对组成的关联数组（在其他语言中它可能被称为映射、散列映射或者散列表），一个键对应一个值。

可以使用下面这些BIF来操纵进程字典。

```
@spec put(Key, Value) -> OldValue.
```

向进程字典加入一个Key、Value键值对。Put的返回值为OldValue，它是Key的前一个关联值。如果没有之前关联过的值，则会返回原子undefined。

```
@spec get(Key) -> Value.
```

查找Key对应的值。如果字典中存在一个关联Key，Value则返回Value，否则返回原子undefined。

```
@spec get() -> [{Key,Value}].
```

以{Key,Value}元组列表的形式返回整个字典。

```
@spec get_keys(Value) -> [Key].
```

返回字典中值为Value的键的列表。

```
@spec erase(Key) -> Value.
```

如果字典中存在键Key对应的值，那么返回对应的值，否则返回原子undefine。最后，删除
104 与键Key相关的值。

```
@spec erase() -> [{Key,Value}].
```

删除整个进程字典。返回值是由{Key,Value}元组组成的列表，其内容为进程字典被删除前的全部信息。

示例如下：

```
1> erase().
[]
2> put(x, 20).
undefined
3> get(x).
20
4> get(y).
undefined
5> put(y, 40).
undefined
6> get(y).
40
7> get().
[{y,40},{x,20}]
8> erase(x).
20
9> get().
[{y,40}]
```

如你所见，进程字典中的变量与传统命令式编程语言中的变量的行为非常相像。如果使用进程字典，那么代码将不再是没有副作用的，在2.6节中所提到的非破坏性赋值所带来的各种好处都将不复存在。因此，应该尽量避免使用进程字典。

说明　我几乎不会使用进程字典。使用进程字典会导致一些难以察觉的错误，会使调试更困难。但有一种使用情形我是认可的，那就是使用进程字典来存储“一次性写入”变量。如果一个键与一个值仅绑定一次而且不再修改，那么偶尔使用一下进程字典也是可以接受的。

5.4.19　引用

引用是全局唯一的Erlang值，使用BIF erlang:make_ref()来创建引用。引用适用于创建那些唯一标签的场合，某些数据包含了这些标签，并可以在稍后的代码中对这些数据进行等价匹配。例如，在一个bug跟踪系统中，可以给每一个新的bug报告加入一个引用，以便给该记录赋予一个
105 唯一标识。

5.4.20　短路布尔表达式

短路布尔表达式是一种仅当必要时才进行参数求值的布尔表达式。下面是两个短路布尔表达式。

```
Expr1 orelse Expr2
```

首先求值的是表达式Expr1。如果Expr1运算结果为true，那么Expr2就不会被求值（即被短路了）。如果Expr1运算结果为false，才会对Expr2进行求值。

```
Expr1 andalso Expr2
```

首先求值的是Expr1。如果Expr1运算结果为true，那么Expr2必须被求值。如果Expr1运算结果为false，则Expr2无须被求值。

说明 在相应的布尔表达式中（A or B; A and B），即便表达式的真实值可以从第一个参数中推断出来，但两个参数仍然都必须进行求值。

5.4.21 比较表达式

在表5-3中展示了Erlang中8种不同的比较表达式。

表5-3 比较表达式

操 作 符	含 义	操 作 符	含 义
X > Y	X大于Y	X == Y	X等于Y
X < Y	X小于Y	X /= Y	X不等于Y
X =< Y	X小于等于Y	X =:= Y	X全等于Y
X >= Y	X大于等于Y	X =/= Y	X不全等于Y

为进行比较，Erlang按照下面的方式为所有类型都定义了大小比较的顺序：

```
number < atom < reference < fun < port < pid < tuple < list < binary
```

这是什么意思呢？举个例子，数字（任何值）总是比原子（任何原子）小，元组总是比一个原子大，以此类推。（注意为了说明比较顺序，这里还包括了端口和PID，这些内容我们会在后面的章节论述。）

对所有类型都赋予比较顺序，意味着可以对存储了任何类型的列表进行排序，并根据这种比较顺序，编写高效的数据访问代码。

如果参数为数值，那么所有的比较操作符，除=:=、=/=之外，其行为都遵循下面的规则。

- 如果一个比较参数是整数另一个是浮点数，那么整数需要在比较前转换为浮点数。
- 如果两个比较参数都是整数或者都是浮点数，那么参数无须变换，以原来类型进行比较。

你需要非常小心地使用==操作符（尤其如果你是一个C或者Java程序员）。在99%的情况下，都应该使用=:=操作符，==仅适用于浮点数和整数的比较，=:=则用于比较两个项目是否全等。[①] 106
如果你拿不准该用那个，那么最好就用=:=。如果你看到有使用==的地方，也应该详加审查。注意，上面这些规则也适用于/=和=/=，/=意思是不等于，=/=意思是不全等。

① 全等意味着含有相同的值（就像Common Lisp中的EQUAL）。由于值是不可变的，因此也不存在任何指针相等情况。

说明　在很多库和公开代码中，你会看到很多应该使用=:=操作符的地方却使用了==。值得庆幸的是这种错误一般来说并不会造成程序错误。因为如果==的两边不包含任何浮点数的话，那么这两种操作符的行为是相同的。

此外，你还必须小心，在函数的子句中总是蕴含着模式匹配，因此如果你定义了一个函数F=fun(12)->...end，当你试图去对F(12.0)进行求值时，就会出错。

5.4.22　下划线变量

我们还需要了解一些关于下划线变量的知识。_VarName是一种特殊的语法，用于声明一个常规变量而不是匿名变量。如果一个变量在一个子句中只被使用一次，那么编译器通常都会提出警告，因为这很有可能是存在错误的征兆。如果变量仅被使用一次，但以下划线开始，那么编译器就不会产生警告消息。

由于_Var仍是一个常规变量，因此，若在编程中忘记它的存在或把它当作匿名的匹配来使用就会导致很多不明显的bug。在一个复杂的模式匹配中，这种bug更加难以察觉。比如，变量_Int
107 在不该重复的地方重复了，就会导致模式匹配失败。

下划线变量有如下两种主要用途。

- 命名一个不准备使用的变量。比如说，函数open(File,_Mode)会比open(File,_)更加具有可读性。
- 为了方便进行程序调试。例如，加入这样的代码：

```
some_func(X) ->
    {P, Q} = some_other_func(X),
    io:format("Q = ~p~n", [Q]),
    P.
```

编译器不会报告任何错误消息。

但现在如果注释掉格式化输出语句。

```
some_func(X) ->
    {P, Q} = some_other_func(X),
    %% io:format("Q = ~p~n", [Q]),
    P.
```

此时再编译这段代码，编译器会警告变量Q没有被使用过。但如果像下面这样重写这个函数：

```
some_func(X) ->
    {P, _Q} = some_other_func(X),
    io:format("_Q = ~p~n", [_Q]),
    P.
```

那么在注释掉格式化输出语句后，编译器就不会再报告警告消息。

到这里我们就真正地完成了对Erlang顺序型编程的学习。仍有一些小问题还没有涉及，不过在讲述应用程序的章节我们会再碰到它们。

108 下一章中，我们会关注一下如何用不同的方法来编译、运行程序。

第6章

编译并运行程序

在前几章中，我们仅使用了Erlang shell，而没有说要如何去编译和运行程序。Erlang shell对于运行一些小例子还是很好用的，不过当程序变得复杂时，你就希望能自动完成编译工作，降低工作负担。这就需要请出makefile了。

实际上有3种不同的方法来运行程序。通过学习本章我们对这三者有了深入的认识后，就可以根据不同的情况选择最佳方法。

编译运行时，你时不时都会碰到错误：makefile出错、环境变量错误、搜索路径设置有误等。我们会帮你处理这些情况，详细地告诉你，如果发现这些错误，应该怎么处理。

6.1 开启和停止 Erlang shell

在一个Unix系统（包括Mac OS X系统）上，你可以在命令终端下这样启动Erlang shell：

```
$ erl
Erlang (BEAM) emulator version 5.5.1 [source] [async-threads:0] [hipe]

Eshell V5.5.1  (abort with ^G)
1>
```

在Windows系统上，则可以点击erl的运行图标。

停止系统最简单的方法就是按Ctrl+C（Windows下是Ctrl+Break），然后按下A键：

```
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
a
$
```

除此之外，你还可以在shell中或在程序中运行表达式erlang:halt()。

erlang:halt()是一个BIF，可以即刻停止系统运行，在大多数情况下我都会使用这种方法来停止程序。不过，使用这个方法来停止系统有一个小小的瑕疵，如果你在运行一个大型的数据库程序，然后轻易地停止系统运行，那么在下次启动系统的时候，系统将不得不进行一次错误修复。因此，在这种情况下，你最好用一种更可控的方法来停止系统。

要执行一个可控的系统关闭操作，如果shell允许输入命令，那么你可以这样输入：

```
1> q().
ok
$
```

这种操作会刷新所有开启的文件，如果数据库开启的话则会停止数据库，按顺序关闭所有OTP应用程序。q()是init:stop()命令在shell中的别名。

如果这些手段都不管用，那么请阅读6.6节。

6.2　配置开发环境

当开始使用Erlang进行编程时，你可能会把所有的模块和文件都置于同一个目录之下，然后从这个目录启动Erlang。如果你这样做，虽然Erlang的加载器能准确无误地找到代码。但当应用程序变得更复杂时，你会希望能将代码分解，成为更加便于管理的几个部分，并把不同部分的代码归入不同的目录中。另一种情况是，你需要从其他的项目中引用代码，这些外部的代码有自己的目录结构。

6.2.1　为文件加载器设定搜索路径

Erlang运行时系统采用了一种代码自动加载机制。为让这个系统正常工作，你必须设定一系
110 列的搜索路径以便能找到正确版本的代码。

实际上，可以在Erlang中对这种代码加载机制进行定制化的编程，我们会在E.4节详细叙述这一话题。

代码加载机制按需加载代码。当系统试图调用一个模块中的函数，但它又尚未被加载时，系统会抛出一个异常，然后系统会尝试为这个未加载模块寻找一份目标代码文件。如果未加载模块名为myMissingModule，那么代码加载器会在当前设定的所有加载路径中搜索一个名为myMissingModule.beam的文件。搜索工作会在找到一个与之匹配的文件时停止，此时，这个文件中的目标代码会被加载到系统里面。

可以启动Erlang shell，然后输入命令code:get_path()来获取当前加载路径的设定值。下面是一个例子：

```
code:get_path().
[".",
"/usr/local/lib/erlang/lib/kernel-2.11.3/ebin",
"/usr/local/lib/erlang/lib/stdlib-1.14.3/ebin",
"/usr/local/lib/erlang/lib/xmerl-1.1/ebin",
"/usr/local/lib/erlang/lib/webtool-0.8.3/ebin",
"/usr/local/lib/erlang/lib/typer-0.1.0/ebin",
"/usr/local/lib/erlang/lib/tv-2.1.3/ebin",
"/usr/local/lib/erlang/lib/tools-2.5.3/ebin",
"/usr/local/lib/erlang/lib/toolbar-1.3/ebin",
"/usr/local/lib/erlang/lib/syntax_tools-1.5.2/ebin",
...]
```

下面两个命令函数在我们需要操纵加载路径时是最为常用的。

@spec code:add_patha(Dir) => true | {error, bad_directory}

增加一个新目录Dir到加载路径的开头。

@spec code:add_pathz(Dir) => true | {error, bad_directory}

增加一个新目录Dir到加载路径的末尾。

通常，随便用哪个都可以。使用这两个函数唯一需要担心的是add_patha和add_pathz产生的结果有所不同。如果你怀疑加载器加载了一个错误的模块，那么可以调用code:all_loaded()（它返回一个所有已被加载的模块列表），或者调用code:clash()来帮助你检查到底出了什么问题。

在模块code中还有几个函数也用于操纵代码加载路径，不过你可能很难碰到需要用到它们的情况，除非你要在某些罕见的操作系统上编程。

111

比较常见的做法是把这些命令集中到home目录里一个名为.erlang的文件中。或者也可以使用下面这种形式的命令来启动Erlang：

```
> erl -pa Dir1 -pa Dir2 ... -pz DirK1 -pz DirK2
```

-pa Dir参数把Dir添加到代码搜索路径的开头，-pz Dir把路径加到代码路径的末尾。

6.2.2 在系统启动时批量执行命令

刚才学到了可以在home目录中的.erlang文件里设定加载路径。但实际上，你可以把任何的Erlang代码写入这个文件中。当启动Erlang时，它会先去读取这个文件中的所有命令，然后逐条运行。假定.erlang文件是这个样子的：

```
io:format("Running Erlang~n").
code:add_patha(".").
code:add_pathz("/home/joe/2005/erl/lib/supported").
code:add_pathz("/home/joe/bin").
```

那么开启系统时，会看到下面这样的输出：

```
$ erl
Erlang (BEAM) emulator version 5.5.1 [source] [async-threads:0] [hipe]

Running Erlang
Eshell V5.5.1  (abort with ^G)
1>
```

如果当前的目录下还有一个.erlang文件，那么在启动时，它的优先级要比home目录中的.erlang文件更高。因此，可以通过在不同的位置放置不同的.erlang文件来启动行为不同的Erlang。这对于定制应用程序是非常有用的。这种情况下，在启动文件中加入一些信息打印语句会是比较好的选择。否则你可能会因为记不起本地启动文件的存在，而陷入非常困惑的情形。

提示 在一些系统中，没有明确定义home目录在哪里，或home目录可能与你想象的有出入。要确定Erlang系统所需的home目录在哪里，你可以这么做：

```
1> init:get_argument(home).
{ok,[["/home/joe"]]}
```

112 由此可见Erlang所需的home目录是/home/joe。

6.3 运行程序的几种不同方法

Erlang程序存储在模块中。因此，只要你写了一个程序，就必须在运行前编译它。除此之外，你还可以通过escript来直接运行程序，而无需对其进行编译。

下面几节将会展示编译、运行一批文件的几种不同方法。我们要处理的程序各不相同，因此启动和关闭它们的方法也不同。

第一个程序是hello.erl，仅打印“Hello world”。它不负责启动和关闭系统，当然也不需要访问任何命令行参数。与之相反，第二个程序fac就需要访问命令行参数了。

下面是我们的基本程序。它打印一个字符串内容为“Hello world”后面跟上一个换行符（~n在Erlang的io和io_lib模块中会被翻译为一个新行）。

hello.erl

```
-module(hello).
-export([start/0]).

start() ->
    io:format("Hello world~n").
```

好，现在我们以不同的方式来编译运行它。

6.3.1 在 Erlang shell 中编译运行

```
$ erl
Erlang (BEAM) emulator version 5.5.1 [source] [async-threads:0] [hipe]

Eshell V5.5.1  (abort with ^G)
1> c(hello).
{ok,hello}
2> hello:start().
Hello world
ok
```

6.3.2 在命令提示符下编译运行

```
$ erlc hello.erl
$ erl -noshell -s hello start -s init stop
Hello world
$
```

113

快速脚本

我们经常希望能在OS命令行中执行任意一个Erlang函数。-eval参数就非常适合这样一些快速脚本。

下面是一些样例：

```
erl -eval 'io:format("Memory: ~p~n", [erlang:memory(total)]).'\
    -noshell -s init stop
```

Windows用户要如上面例子一样运行，必须在下面两个方法中选择一个：在PATH变量中加入包含Erlang可执行文件的目录，或者在输入erlc或者erl时加上带引号的文件全路径。例如：

```
"C:\Program Files\erl5.5.3\bin\erlc.exe" hello.erl
..
```

第1行，erlc hello.rl编译文件hello.erl，生成名为hello.beam的目标代码文件。第2行有3个选项。

-noshell

启动Erlang，但是关闭shell。（因此你不会看到Erlang的提示信息，这些信息通常在启动系统时就会显示出来。）

-s hello start

运行函数hello:start()。

说明 当使用-s Mod...选项时，Mod模块必须已经被编译过。

-s init stop

当apply(hello,start,[])运行结束时，系统将计算函数init:stop()。

命令erl -noshell...可以被归入一个shell脚本中，因此通常我们会创建一个shell脚本来运行设置路径的程序（使用-pa目录）并启动它。

在我们的例子中，使用了两个-s命令。但其实，只要我们愿意，可以在命令行运行任意多个函数。每一个-s...命令会由一个apply语句来进行解释和运行，一旦当前函数运行完毕，则会继续执行下一个命令。

114

下面是一个启动hello.erl的简单样例：

hello.sh

```
#!/bin/sh
erl -noshell -pa /home/joe/2006/book/JAERANG/Book/code\
             -s hello start -s init stop
```

说明 这个脚本需要有一个指向包含hello.beam文件的目录的绝对路径。因此尽管这个脚本能在我的机器上工作，但不一定能在你的机器上运行，必须先编辑它以保证上述路径指向机器上的正确位置。

要运行shell脚本，我们先要对该文件进行一次chmod，然后就可以执行它了：

```
$ chmod u+x hello.sh
$ ./hello.sh
Hello world
$
```

说明　在Windows上，操作系统是无法识别#!的。在Windows环境下，需要创建批处理文件（.bat文件），必须使用全路径名来指定到Erlang可执行文件（如果PATH变量中没有设定Erlang可执行文件路径）。

一个典型的Windows批处理文件是这样的：

hello.bat

```
"C:\Program Files\erl5.5.3\bin\erl.exe" -noshell -s hello start -s init stop
```

6.3.3　把程序当作 escript 脚本运行

使用escript，你可以把程序直接当作脚本来运行——无需预先进行编译。

警告　只有Erlang R11B-4或更高的版本才支持escript。如果Erlang是一个较早的版本，那么你应升级到最新的Erlang。

要把hello程序当作escript来运行，我们首先要创建一个下面这样的文件：

hello

```
#!/usr/bin/env escript

main(_) ->
    io:format("Hello world\n").
```

115

开发时从模块导出函数

当你正在编写代码时，会遇到一件麻烦事——你得不时地添加删除函数的导出声明以便可以在shell中运行导出的函数。

有一个特别的声明-compile(export_all)，它告诉编译器把模块中的每个函数都导出。如果在编写代码的时候用上它，工作就会便利很多。

在完成编码时，应该注释掉export_all声明，然后再加入适当的导出声明。这么做的好处有两个：第一，在稍后回来阅读代码时，你会知道只有重要的函数才会被导出，所有其他的函数都不能被模块外部代码所调用，从而可以保证在导出函数的结构不变的情况下你能随心所欲地修改它们而不用担心波及其他程序；第二，如果编译器知道哪些函数会被从模块导出，那么可以针对它们产生更加优化的代码。

在Unix系统上，[①]我们可以像下面一样立刻运行这些代码而无须编译。

```
$ chmod u+x hello
$ ./hello
Hello world
$
```

① 我不知道是否可以在Windows上运行escript。如果有人知道怎么做的话，可写电子邮件来告诉我，我会把这些内容加到书里。

说明 这个文件的文件模式必须被设为“executable”(在Unix系统中，输入命令chmod u+x File)。这件事你只须做一次，而不是每次运行都需要这么做。

6.3.4 用命令行参数编程

“Hello world”没有参数。让我们再来回顾一下计算阶乘的程序，它需要一个参数。 116

首先，看一下这些代码：

fac.erl

```
-module(fac).
-export([fac/1]).

fac(0) -> 1;
fac(N) -> N*fac(N-1).
```

我们可以编译fac.erl，然后在Erlang shell中运行它：

```
$ erl
Erlang (BEAM) emulator version 5.5.1 [source] [async-threads:0] [hipe]

Eshell V5.5.1  (abort with ^G)
1> c(fac).
{ok,fac}
2> fac:fac(25).
15511210043330985984000000
```

如果想要在命令行上运行这个程序，则需要修改一下，以支持命令行参数：

fac1.erl

```
-module(fac1).
-export([main/1]).

main([A]) ->
    I = list_to_integer(atom_to_list(A)),
    F = fac(I),
    io:format("factorial ~w = ~w~n",[I, F]),
    init:stop().

fac(0) -> 1;
fac(N) -> N*fac(N-1).
```

然后，再来编译运行它：

```
$ erlc fac1.erl
$ erl -noshell -s fac1 main 25
factorial 25 = 15511210043330985984000000
```

说明 实际上，那个函数的名字并不一定要叫做main，你可以取任何你喜欢的名字。重要的是函数名和命令行中的名字要匹配。 117

最后，还可以在一个escript中运行它：

factorial

```
#!/usr/bin/env escript

main([A]) ->
    I = list_to_integer(A),
    F = fac(I),
    io:format("factorial ~w = ~w~n",[I, F]).

fac(0) -> 1;
fac(N) ->
    N * fac(N-1).
```

无须编译，直接运行：

```
$ ./factorial 25
factorial 25 = 15511210043330985984000000
$
```

6.4　使用 makefile 进行自动编译

在编写一个大项目的时候，开发过程的自动化程度越高越好。这其中有两个原因。第一，长远来看，这能节省输入量，在反复测试程序时，一次又一次地重复输入相同的命令，实在是很浪费时间，我想把我的手指解放出来。

第二，我通常会搁下手中的事情去做另外一个项目。这一耽搁可能就是好几个月，当我再回到先前搁置下来的项目时，经常会忘记如何来编译项目代码，而有了make就能救我一命！

make程序可以帮助我自动完成工作的一个工具，我用它来编译和分发Erlang代码。大多数的makefile都极为简单并且我有一个简单的模板可以满足大部分需要。

我不想在这里全面地介绍makefile，[①]在这里只会展示一些在编译Erlang程序时涉及的知识。而且，在你阅读本书时，还会学到makefile的各种知识，到最后你会对makefile有一个透彻的了解，
118 也能编写自己需要的makefile。

6.4.1　makefile 模板

下面这个例子是我从大量项目的makefile中归纳出来的模板：

Makefile.template

```
# leave these lines alone
.SUFFIXES: .erl .beam .yrl

.erl.beam:
        erlc -W $<
```

① 对于makefile的详细信息请参看http://en.wikipedia.org/wiki/Make。

```
.yrl.erl:
        erlc -W $<

ERL = erl -boot start_clean

# Here's a list of the erlang modules you want compiling
# If the modules don't fit onto one line add a \ character
# to the end of the line and continue on the next line

# Edit the lines below
MODS = module1 module2 \
       module3 ... special1 ...\
       ...
       moduleN

# The first target in any makefile is the default target.
# If you just type "make" then "make all" is assumed (because
#   "all" is the first target in this makefile)

all: compile

compile: ${MODS:%=%.beam} subdirs

## special compilation requirements are added here

special1.beam: special1.erl
        ${ERL} -Dflag1 -W0 special1.erl

## run an application from the makefile

application1: compile
        ${ERL} -pa Dir1  -s application1 start Arg1 Arg2

# the subdirs target compiles any code in
# sub-directories

subdirs:
        cd dir1; make
        cd dir2; make
        ...
# remove all the code

clean:
        rm -rf *.beam erl_crash.dump
        cd dir1; make clean
        cd dir2; make clean
```

119

makefile以一些编译Erlang模块的规则和一些扩展名为.yrl的文件开始（这些文件包含了

Erlang解析生成器程序[①]的解析定义文件）。

最重要的部分是以此为开头的部分：

```
MODS = module1 module2
```

这是一个我想编译的所有Erlang模块列表。

任何在MODS列表中出现的模块都会被Erlang命令erlc Mod.erl编译。有些模块可能需要进行特殊处理（例如模板文件中的Special1模块），因此有一个专门的规则来处理它们。

在makefile内部有一些target。一个target是在第一栏中以一个字符与数字组成的串开始，以冒号结束的字符串。在上述的makefile模板中，all、compile和special1.beam都是target。要运行makefile，需要在shell中输入命令：

```
$make [Target]
```

参数Target是可选的。如果忽略Target，那么就默认运行make文件中的第一个target。在上面的样例中，如果命令行没有指明target，那么就默认执行target all。

如果想要编译所有的代码，并运行application1，那么应该输入的命令是make application1。而如果我想将这个变成默认的行为，也就是说，在只输入make的时候就能完成上面的任务，那么就应该把定义target application1的那几行移到makefile中第一个target的位置上。

target clean会移除所有已编译完成的Erlang目标文件和erl_crash.dump文件。erl_crash.dump就是所谓的崩溃转储文件（crash dump），它包含的信息对调试程序非常有用。具体的细节可以参
120 见6.10节。

6.4.2　定制 makefile 模板

我不是一个喜欢陶醉于自己的代码的人。因此，我最常做的一件事情就是打开makefile模板，然后把那些与当前软件开发项目无关的代码一一移除。这样，makefile中剩下的内容就会变得极为精练易懂。当然还有一个办法，那就是写一个通用的makefile，它可以被所有其他的makefile，并且根据另外那些makefile中的变量对通用的makefile进行参数化配置。

经过这些步骤，最后我们得到一个非常精简的makefile，大概就像下面一样：

```
.SUFFIXES: .erl .beam

.erl.beam:
    erlc -W $<

ERL = erl -boot start_clean

MODS = module1 module2 module3

all: compile
```

① Erlang解析生成器称为yecc（yacc的Erlang版本，yacc是yet another compiler compiler的缩写），具体的教程请参见 http://www.erlang.org/contrib/parser_tutorial-1.0.tgz。

```
    ${ERL} -pa '/home/joe/.../this/dir' -s module1 start

compile: ${MODS:%=%.beam}

clean:
    rm -rf *.beam erl_crash.dump
```

6.5 在 Erlang shell 中的命令编辑

Erlang shell包含了一个内置的编辑器。它能识别通用的emacs编辑器中的行编辑命令的一个子集。你可以通过几个简单的击键来调回前一行的命令并对其继续编辑。表6-1展示了Erlang shell所支持的命令（注意^Key的意思是你应该按下Ctrl+Key）。

表 6-1

命　令	描　述
^A	一行的开始
^E	一行的结尾
^F或者右箭头	前近一个字符
^B或者左箭头	后退一个字符
^P或者上箭头	前一行
^N或者下箭头	后一行
^T	颠倒两个字母顺序
Tab键	可以帮助扩展当前的模块或者函数名

121

6.6 解决系统死锁

有时，你会发现很难停止一个运行中的Erlang，出现这种情况可能有如下几种原因。

- shell没有响应。
- Ctrl+C处理程序被禁止。
- Erlang启动时带有-detached选项，你可能难以察觉它在运行。
- Erlang启动时带有-heart Cmd选项。这个选项会启动一个操作系统监视进程来监视系统中的Erlang进程。如果发现Erlang进程死亡，监视进程就会执行Cmd。通常Cmd会重启Erlang系统。这是在构建一个容错节点时常会用到的一个技巧——如果Erlang自己死了（这是不应该发生的），这个方法就会将它重启。应付这种情况有一个诀窍，那就是在终止Erlang进程之前，先找到心跳进程（在类Unix系统下使用ps命令，Windows下则打开任务管理器），然后终止这个心跳进程。
- 可能是发生了严重的错误，导致一个Erlang僵尸进程被遗留在了操作系统中。

6.7 如何应对故障

本节列出了几个经常会碰到的问题（当然还有它们的解决方案）。

6.7.1　未定义/遗失代码

如果试图去运行一个代码加载器无法找到的代码（可能是因为代码搜索路径有误），你会碰到一个undef的错误消息，就像下面这个例子：

```
1> glurk:oops(1,23).
** exited: {undef,[{glurk,oops,[1,23]},
                  {erl_eval,do_apply,5},
                  {shell,exprs,6},
                  {shell,eval_loop,3}]} **
```

实际上，系统里并没有一个叫做glurk的模块，但这不是我们所关心的问题。你现在要注意的是这条错误消息。这个消息告诉我们系统试图用参数1和23去调用模块glurk中的oops函数。出现这条错误信息的原因肯定是下面4种情况之一。

- 系统里真的不存在glurk模块——不是因为没有正确的路径可供寻找，而是你要找的东西根本就不存在。比如说，因为拼写错误。

122

有人注意到分号了吗

如果忘记函数子句之间的分号或者错误地使用了句号，那么就会出错——是真的编译错误而不是警告。

如果你在模块bar的1234行定义了函数foo/2，但是却用句号代替了分号，那么编译器就会说：

bar.erl:1234 function foo/2 already defined.

千万别那么做，确保子句总是以分号分割。

- 系统里有一个glurk模块，但是还没有编译。系统要寻找的是位于代码搜索路径中的一个叫做glurk.beam的文件。
- 系统有一个glurk模块，它已经被编译了，但是存放blurk.beam的目录不在搜索路径的目录之中。要修正这个问题，必须修改搜索路径。稍后会告诉你怎么做。
- 在代码加载路径中有几个不同的glurk版本，而且我们选择了一个错误的版本。这个错误很少发生，但是没准你就会碰到。

  如果你怀疑是这个问题导致出错，可以运行code:clash()函数，它会报告在代码搜索路径中发现的所有重名模块。

6.7.2　makefile 不能工作

makefile会出什么错呢？实际上，不仅会出错，还会有不少。不过本书既然并不是讲述makefile的专著，因此我在这里也只讲几个比较普遍的错误。通常会遇到下面两个最常见的错误。

- makefile中的空格。makefile的格式是非常严格的。尽管你的眼睛看不到每行的缩进，但是makefile中的缩进行应该以一个tab字符开始［如果你想要在下一行中延续上一行的内容，那么上一行必须以（\）结尾］。如果某一行的开始有空格，那么makefile就会无法

分辨语义，然后你就会看到make抛出错误。 123

- Erlang文件遗失。如果在Mods中声明的模块里缺失了某一个模块文件，那么你就会得到错误信息。我们现在来模拟一下这个问题，假定在MOS中包含了一个名为glurk的模块，但是在代码目录中却没有一个叫做glurk.er的文件。在这种情况下，make就会失败并且抛出如下信息。

```
$ make
make: *** No rule to make target 'glurk.beam',
          needed by 'compile'.  Stop.
```

另外一种原因是，并没有遗失模块文件，但是在makefile中模块名称的拼写有误。

6.7.3 shell 没有响应

如果shell对命令没有响应，导致这种结果的原因有很多。可能shell进程本身崩溃了，或者输入了一个永远不会结束的命令，还有可能是你忘记了在一个命令后面输入一个“封闭引号”或者是点回车。

无论是什么原因，你都可以按下Ctrl+G来中断当前的shell，然后像下面的样例一样继续工作：

```
❶  1> receive foo -> true end.
   ^G
   User switch command
❷  --> h
   c [nn]    - connect to job
   i [nn]    - interrupt job
   k [nn]    - kill job
   j         - list all jobs
   s         - start local shell
   r [node]  - start remote shell
   q         - quit erlang
   ? | h     - this message
❸  --> j
   1* {shell,start,[init]}
❹  --> s
   --> j
   1  {shell,start,[init]}
   2* {shell,start,[]}
❺  --> c 2
   Eshell V5.5.1  (abort with ^G)
   1> init:stop().
   ok
   2> $
```

❶ 这里告诉shell去接收一个foo消息。但是因为没有人向shell发送过这个消息，此时shell就会进入无限等待。然后按下Ctrl+G。 124

❷ 系统进入“shell JCL”[①]模式。这里我可能忘记了命令，于是输入了h寻求帮助。

① JCL是Job Control Language（任务控制语言）的缩写。

❸ 输入j来列出所有的工作任务。任务号1标记上了一个星号，意思是它是默认shell。所有带有可选参数[nn]的命令都会使用默认shell，除非补充一个特殊的参数。

❹ 输入s启动了一个新shell，然后又输入了一个j。这次我看到两个shell分别标为1和2，shell 2变成了默认shell。

❺ 输入c 2，切换到新启动的shell 2，然后停止系统。

正如你看到的，你可以操纵很多shell，并且可以通过Ctrl+G在它们之间互相切换，然后在其中输入合适的命令。甚至可以使用r命令，在一个远端节点上启动一个shell。

6.8　获取帮助

在Unix系统上，可以输入下列命令：

```
$ erl -man erl
NAME
erl - The Erlang Emulator

DESCRIPTION
The erl program starts the Erlang runtime system.
The exact details (e.g. whether erl is a script
or a program and which other programs it calls) are system-dependent.
...
```

也可以用下面的方法得到单个模块的帮助信息：

```
$ erl -man lists
MODULE
lists - List Processing Functions

DESCRIPTION
This module contains functions for list processing.
The functions are organized in two groups:
...
```

说明　在Unix系统上，用户手册不是默认安装的。如果命令erl -man...不能工作，那么你就需要安装用户手册。所有的用户手册已经打了一个压缩包，可以从 http://www.erlang.org/download.html中下载。用户手册应该解压缩在Erlang的安装目录下（通常是/usr/local/erlang）。

125

所有的文档也被做成了可下载的HTML文件。在Windows上HTML文档会默认安装，你可以从开始菜单中的Erlang选项中找到它们。

6.9　调试环境

Erlang shell有一批内建的命令，你可以使用shell命令help()看到它们：

```
1> help().
** shell internal commands **
b()        -- display all variable bindings
e(N)       -- repeat the expression in query <N>
f()        -- forget all variable bindings
f(X)       -- forget the binding of variable X
h()        -- history
...
```

这些命令的定义都在模块shell_default中。

如果你想要定义自己的命令，只需创建一个叫做user_default的模块，例如：

user_default.erl

```
-module(user_default).

-compile(export_all).

hello() ->
    "Hello Joe how are you?".

away(Time) ->
    io:format("Joe is away and will be back in ~w minutes~n",
              [Time]).
```

在编译了这个模块之后，把它放到加载路径中，然后你就可以像前面那样，调用user_default模块中的任何函数而不需要给出模块名：

```
1> hello().
"Hello Joe how are you?"
2> away(10).
Joe is away and will be back in 10 minutes
ok
```

126

6.10 崩溃转储

如果Erlang崩溃了，那么它会留下一个叫做erl_crash.dump的文件。文件的内容会给你一些线索，告诉你哪里出了问题。要分析崩溃转储，Erlang中有一个基于Web的崩溃分析器。只须输入下列命令，就可以启动分析器：

```
1> webtool:start().
WebTool is available at http://localhost:8888/
Or  http://127.0.0.1:8888/
{ok,<0.34.0>}
```

用浏览器访问http://localhost:8888/，然后就可以详细研究错误日志文件了。

到目前为止，我们已经学了有关编译和运行的所有的细节问题，接下来就可以开始学习并发编程。这将是一个你从未涉足过的陌生领域，但这是乐趣的真正开始。

127

第7章 并　发

1. 关于开发

并发概念其实早已根植于我们的大脑，根本无需后天学习，它是本能的一部分。人类对于刺激的反应很快，这个过程是由大脑中一个叫做杏仁核的部分所掌控。如若没有这种应激反应，很难想象我们人类如何能够生存到今天。对于生存来说，有意识的思考速度太慢，很多时候，在我们意识到需要“踩刹车”之前，我们的身体已经先做出了反应。

在主干道上开车时，我们的大脑同时跟踪着数十辆乃至数百辆汽车的位置。这个过程是下意识的，并不需要我们有意识的参与。试想，如果没有这种能力，开车会是一件多么危险的事情。

2. 世界是并行的

如果想要编写一个程序，它能模拟现实世界中对象的行为，那么这个程序就必须具备并发结构。

这也就是为什么我们需要采用一种基于并发的编程语言来进行开发。

然而，更常见的是使用顺序型的编程语言来做这件事，这是一种毫无必要的障碍。

与之相对应，若能用一种专为并发量身定做的语言来进行开发，那么并行编程无疑就会大大简化。

3. Erlang程序模拟我们思考和反应的模式

人与人之间并不存在某种“共享的记忆”。我有我的记忆，你也有你的记忆。我们有两个大脑，每人一个。也没有证据表明它们会以某种方式纠缠在一起。如果想给你留下印象，我就向你发送消息，对你说些什么，或者向你挥挥手。

当你听到我的话，或者留意到我，你就会有印象。当然，如果既不观察你的反应也不向你提出询问，那么我也无法知道你是否接收到了我发出的消息。

这也是Erlang进程的工作方式。Erlang进程之间没有共享内存，每一个进程都有它自己的内存。想要修改其他进程的内存，你只能向它发送一个消息，然后希望它能收到而且能理解这个消息。

要想确认其他进程已经收到了你的消息并修改了自己的内存，你只能向它发问（通过发送消息）。这就是我们交互的方式。

Sue：你好Bill，我的电话号码是45678912。

Sue：你听清楚了吗？

Bill：当然，你的电话号码是45678912。

这种应答模式再熟悉不过了。从出生的那一刻起，我们就在不断地学习如何与这个世界打交道——观察它，向它发送消息，再观察它的反应。

4．人就像是一个通过发送消息来进行交流的实体

这既是我们的交流方式，也是Erlang进程的工作方式。因而，理解Erlang程序对我们来说是很简单的。

一个Erlang程序由数十数百乃至上千个小的进程组成，所有这些进程都是独立工作的。它们之间通过发送消息进行交流。每个进程都有自己私有的内存。它们的行为就像一大屋子的人，彼此还在互相交谈。

这种结构使Erlang程序在维护和扩展上天生具有优势。假如我们有10个人（进程），但是他们的工作太多以至于无法完成，该怎么办？增加更多的人手。我们又是如何管理这群人的呢？很简单，向他们喊出指令（也就是广播）。

Erlang的进程不共享内存，因此也就无须锁定内存。有锁就会丢钥匙，丢了钥匙能怎么办？发飙也没用，实际上你就是没辙。在那些用到了锁的代码中，丢了钥匙或者弄坏了锁，这两者的区别其实不大。在分布式软件系统中使用锁和钥匙更是难以避免故障发生。 130

但是Erlang既没有锁也不用钥匙。

5．如果有人死了，其他人会注意到

如果我在大厅里突然倒地死掉，可能有人会注意到（好吧，至少我希望是这样）。Erlang进程就像人一样——它们也可能意外消亡。与人类相比，所不同的是，它们在死之前会用最后一口气奋力呼喊它们因何而亡。

想象有一个挤满人的房间，突然有人倒在地上死了。临死之际，他说“我死于心脏病”或者“我死于胃穿孔”。Erlang进程的行为就是这样的。一个进程在它消亡的时候说的可能是“我死于有人要我去除0”，另一个说的可能是“我死于有人要我从一个空列表里取最后一条数据”。

此时，在这个人满为患的房间里，你可以想象其中有一小撮人他们特殊的使命就是负责清理尸体。假设两个人，Jane和John。如果Jane死了，那么John就会负责处理由Jane死亡带来的相关问题。而如果John死了，那么Jane也会做相同的事。我们可以认为在Jane和John之间有一条隐形的纽带，使得他们之中的任何一个死亡时，另一个会肩负起处理这一问题的责任。

这就是Erlang错误侦测的工作机制。进程可以互相链接，如果一个进程消亡，那么另一个进程就会得到一条消息，被告知第一个进程消亡，及其原因。

这就是最基本的原则。

这就是Erlang程序的工作机制。

本章的要点我们可以归纳为如下几点。

- Erlang程序由成百上千个进程组成，这些进程可以互发消息。
- 进程能否收到和理解这些消息是不确定的。如果你想知道一个消息是否被对方收到和理解，那么必须向这个进程发消息询问并等待回应。 131

- 两个进程可以互相链接。如果其中一个进程消亡，那么另外一个进程就会收到一条消息，指明第一个进程消亡的原因。

这种简单的编程模型，是面向并发模型编程的一部分。

下一章，我们会开始编写并发程序。我们会先学习3个新原语：spawn、send（使用!操作符）和receive。之后就可以编写一些简单的并发程序。

当进程消亡，与其相链接的进程就会收到通知。这就是第9章的话题并发编程中的错误处理。

在阅读下面两章时，你可以在脑子里想象一个挤满了人的大厅，这些人就是进程。大厅中的人只有私有的记忆，这就是进程的状态。要改变你的记忆（状态），我就对你说话，你则侧耳倾
132 听，这就是发送和接受消息。我们会传宗接代，这就是spawn，我们也会死亡，这就是进程退出。

第8章

并发编程

在本章中，我们要讨论的是进程。它们每一个都是独立运行的Erlang虚拟机，都可以对Erlang函数进行求值。

相信你之前也遇到过进程这一概念，但那只是操作系统下的进程。

在Erlang里，进程属于程序语言而非操作系统。

在Erlang里：

- 创建和销毁进程非常迅速；
- 在两个进程间收发消息非常迅捷；
- 进程在所有的操作系统上行为相同；
- 可以创建大量进程；
- 进程之间不共享任何数据，彼此完全独立；
- 进程间交互的唯一方法就是通过消息传递。

由于这些特性，Erlang有时也被称作纯粹的消息传递语言。

如果此前你还未和进程打过交道，那么至少也应该听过有关它的种种传言，说它是如何的复杂难懂。你还可能听过有关非法内存、竞态条件、共享内存污染等的恐怖传说。在Erlang中，和进程打交道可不需要这样，它只需要3个原语：spawn、send和receive。

8.1 并发原语

我们在顺序型编程中学到的所有知识在并发编程中仍然适用。所不同的是，需要引入下面这些原语。

```
Pid = spawn(Fun)
```

创建一个新的并发进程，用于对Fun求值。新进程与调用者所在的进程并发运行。spawn返回一个Pid[process identifier（进程标识符）的缩写]。你可以使用Pid向进程发送消息。

```
Pid ! Message
```

向标识符为Pid的进程发送消息。消息发送是异步的，发送者无须等待返回结果就可以继续处理它自己的事务。!被称为发送操作符。

Pid!M定义的返回值是它所发送的消息本身。因此Pid1!Pid2!...!M意味着将消息M发送到

Pid1、Pid2等所有的进程中。

receive ... end

接收一个发给当前进程的消息。它的语法是这样的：

```
receive
   Pattern1 [when Guard1] ->
       Expressions1;
   Pattern2 [when Guard2] ->
       Expressions2;
   ...
end
```

当一个消息到达进程时，系统会尝试与Pattern1匹配（也可能与Guard1匹配）。如果成功，那么对Expressions1求值。如果第一个模式不匹配，那么再尝试第二个，以此类推。如果没有匹配到任何模式，那么消息就会留给后续过程来处理，然后进程等待下一条消息。这部分的内容将在8.6节中详细描述。

134 在receive语句中的模式和断言语法与我们在定义函数时所用的相关语法一致。

8.2 一个简单的例子

还记得我们在3.1节中如何编写area/1函数吗？这里再来列一遍代码，帮助你回忆：

geometry.erl

```
area({rectangle, Width, Ht}) -> Width * Ht;
area({circle, R})            -> 3.14159 * R * R.
```

现在将这个函数进行重构，将其改造为进程：

area_server0.erl

```
-module(area_server0).
-export([loop/0]).

loop() ->
    receive
        {rectangle, Width, Ht} ->
            io:format("Area of rectangle is ~p~n",[Width * Ht]),
            loop();
        {circle, R} ->
            io:format("Area of circle is ~p~n", [3.14159 * R * R]),
            loop();
        Other ->
            io:format("I don't know what the area of a ~p is ~n",[Other]),
            loop()
    end.
```

我们能够在shell中创建一个进程对loop/0进行求值：

```
1> Pid = spawn(fun area_server0:loop/0).
<0.36.0>
```

```
2> Pid ! {rectangle, 6, 10}.
Area of rectangle is 60
{rectangle,6,10}
3> Pid ! {circle, 23}.
Area of circle is 1661.90
{circle,23}
4> Pid ! {triangle,2,4,5}.
I don't know what the area of a {triangle,2,4,5} is
{triangle,2,4,5}
```

上面的代码说明了什么？在第1行，创建了一个新的并发进程。spawn(Fun)创建了一个并发进程来对函数Fun求值，它返回Pid，这里就是打印出来的<0.36.0>。 135

第二行向这个进程发送了消息。这个消息与loop/0中的receive语句的第一个模式匹配：

```
loop() ->
    receive
        {rectangle, Width, Ht} ->
            io:format("Area of rectangle is ~p~n",[Width * Ht]),
            loop()
        ...
```

一旦接收到这个消息，进程就会打印出矩形的面积。之所以会在shell中打印{rectangle, 6, 10}，这是因为Pid! Msg的值就被定义为Msg。如果我们向进程发送一个不能识别的消息，那么它会打印警告。也就是在receive语句中通过Other->...代码进行处理。

8.3 客户/服务器介绍

客户/服务器架构是Erlang的核心内容。传统上，客户/服务器架构涉及由客户机和服务器组成的网络。大多数的情况都存在多个客户机和单个服务器。服务器这个词通常会让人联想到运行在特殊计算机上的重型软件。

但在我们的语境中，更多涉及的是轻量级的服务器机制。在客户/服务器架构下的客户机和服务器都是分离的进程，Erlang的消息传递则被用于客户机和服务器之间的通信。客户机和服务器可以运行在同一台机器上，也可以运行在不同的机器上。

客户和服务器这两个词涉及分别由两个进程扮演的不同角色。客户机总是通过向服务器发送请求来发起一个计算，服务器则在计算出一个结果之后向客户端发送一个回应。

下面我们就来写第一个客户/服务器应用程序。我们会先对前面几章中编写的程序进行一些小的修改。

在前面的例子中，我们所要做的就是向进程发送一个请求，这个进程收到后将其打印。现在，我们想做的事是向这个发送原始请求的进程发送一个回应。问题是我们还不知道消息到底是谁发送过来的。要发送一个这样的回应，客户机必须在请求中包含一个服务器可以回应的地址。这就像向某人发送一封邮件，如果你想得到回信，最好在信里写上你的地址！ 136

因此，发送者必须包含一个回复地址。那么我们就必须把代码

```
Pid ! {rectangle, 6, 10}
```

改成这样：

```
Pid ! {self(),{rectangle, 6, 10}}
```

self()是客户进程自己的Pid。处理这个请求，接收代码需要进行修改：

```
loop() ->
    receive
        {rectangle, Width, Ht} ->
            io:format("Area of rectangle is ~p~n",[Width * Ht]),
            loop()
        ...
```

改成：

```
loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! Width * Ht,
            loop();
        ...
```

注意，我们是如何向由From参数标识的进程发送计算结果的。因为客户机已经在参数中表明了自己的进程ID，所以客户机就会收到这个结果。

发送起始请求的进程被称为客户机。接收请求然后发送回应的进程就叫做服务器。

最后，我们再来加入一个小的辅助函数，叫做rpc[remote procedure call（远程过程调用）的缩写]，它用来封装发送请求和等待回应：

area_server1.erl

```
rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        Response ->
            Response
    end.
```

137

把所有这些整合到一起，我们就得到了下面这样的代码：

area_server1.erl

```
-module(area_server1).
-export([loop/0, rpc/2]).

rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        Response ->
            Response
    end.

loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
```

```
            From ! Width * Ht,
            loop();
        {From, {circle, R}} ->
            From !  3.14159 * R * R,
            loop();
        {From, Other} ->
            From ! {error,Other},
            loop()
    end.
```

我们可以在shell中做一些测试：

```
1>  Pid = spawn(fun area_server1:loop/0).
<0.36.0>
2>  area_server1:rpc(Pid, {rectangle,6,8}).
48
3>  area_server1:rpc(Pid, {circle,6}).
113.097
4>  area_server1:rpc(Pid, socks).
{error,socks}
```

这段代码中还有一点小小的瑕疵。在函数rpc/2中，我们向服务器发送一个请求，然后等待回应。但其实我们等待的不只是这个服务器发送的回应，还在等待所有的消息。如果在这个客户机等待来自服务器的回应时，其他的进程也向这个客户机发送了消息，那么它会把这个消息误认为是来自服务器的回应。我们可以通过修改receive语句的形式来纠正这个错误：

```
loop() ->
    receive
        {From, ...} ->
            From ! {self(), ...}
            loop()
        ...
```

138

同时，也要对rpc进行如下的修改：

```
rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        {Pid, Response} ->
            Response
    end.
```

这段代码是如何工作的？在进入rpc函数的时候，Pid会被绑定到某个值，因此在模式{Pid,Response}中，Pid是绑定的，而Response是自由的。这个模式仅会匹配包含二元组[①]而且二元组第一个元素是Pid的消息。所有其他的消息会被放入队列等待处理。（receive的接受机制就是前面提到的选择性接收）。

经过这样的修改，我们得到如下的代码：

area_server2.erl

```
-module(area_server2).
```

① N-tuple的意思就是一个成员数为N的元组，因此二元组就是成员数为2的元组。

```
-export([loop/0, rpc/2]).

rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        {Pid, Response} ->
            Response
    end.

loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! {self(), Width * Ht},
            loop();
        {From, {circle, R}} ->
            From !  {self(), 3.14159 * R * R},
            loop();
        {From, Other} ->
            From ! {self(), {error,Other}},
            loop()
    end.
```

这段代码会照我们期望的那样工作：

```
1> Pid = spawn(fun area_server2:loop/0).
<0.37.0>
3> area_server2:rpc(Pid, {circle, 5}).
78.5397
```

139

我们还可以对这段代码再做一些改进，把spawn和rpc的机制隐藏到模块中。这是一种良好的习惯，因为我们可以修改服务器的内部细节而不需要更改客户机的代码。最后，代码是这样的：

area_server_final.erl

```
-module(area_server_final).
-export([start/0, area/2]).

start() -> spawn(fun loop/0).

area(Pid, What) ->
    rpc(Pid, What).

rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        {Pid, Response} ->
            Response
    end.

loop() ->
```

```
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! {self(), Width * Ht},
            loop();
        {From, {circle, R}} ->
            From !  {self(), 3.14159 * R * R},
            loop();
        {From, Other} ->
            From ! {self(), {error,Other}},
            loop()
    end.
```

要运行它，我们可以调用函数start/0和area/2（也就是我们之前调用spawn和rpc的入口）。这两个名字恰到好处地描述了服务器的行为：

```
1> Pid = area_server_final:start().
<0.36.0>
2> area_server_final:area(Pid, {rectangle, 10, 8}).
80
4> area_server_final:area(Pid, {circle, 4}).
50.2654
```

8.4 创建一个进程需要花费多少时间

现在，你可能对性能心存疑虑。毕竟，如果创建成百上千个Erlang进程，我们必须付出某种代价。那么现在就让我们来看看，这么做的代价到底有多大。 140

为了探寻究竟，我们要对大量Erlang进程的创建时间进行统计。下面就是进行统计的代码：

processes.erl

```
-module(processes).

-export([max/1]).

%% max(N)
%%   Create N processes then destroy them
%%   See how much time this takes

max(N) ->
    Max = erlang:system_info(process_limit),
    io:format("Maximum allowed processes:~p~n",[Max]),
    statistics(runtime),
    statistics(wall_clock),
    L = for(1, N, fun() -> spawn(fun() -> wait() end) end),
    {_, Time1} = statistics(runtime),
    {_, Time2} = statistics(wall_clock),
    lists:foreach(fun(Pid) -> Pid ! die end, L),
    U1 = Time1 * 1000 / N,
    U2 = Time2 * 1000 / N,
    io:format("Process spawn time=~p (~p) microseconds~n",
```

```
                [U1, U2]).

wait() ->
    receive
        die -> void
    end.

for(N, N, F) -> [F()];
for(I, N, F) -> [F()|for(I+1, N, F)].
```

在我用来编写此书的电脑上（一台Intel Celeron 2.4G处理器、512M内存、运行Ubuntu Linux的电脑），运行上面的程序得到的结果是这样的：

```
1> processes:max(20000).
Maximum allowed processes:32768
Process spawn time=3.50000 (9.20000) microseconds
ok
2> processes:max(40000).
Maximum allowed processes:32768
=ERROR REPORT==== 26-Nov-2006::14:47:24 ===
Too many processes
...
```

创建20 000个进程时，每个进程平均花了3.5 μs的CPU时间，相当于9.2 μs的消逝时间（亦即
141 真实时间）。

注意，我使用BIF erlang:system_info(process_limit)来查找系统允许的进程数上限。这个值是预设的，在你的系统中，配置未必允许创建这么大的数量。当试图创建超过限制的进程数时，系统就会崩溃，然后抛出一个错误报告（命令2就是这种情况）。

我的系统对此参数的限制是32 767个进程，要超过这个限制，必须在启动Erlang虚拟机时带上+P参数，如下：

```
$ erl +P 500000
1> processes:max(50000).
Maximum allowed processes:500000
Process spawn time=4.60000 (10.8200) microseconds
ok
2> processes:max(200000).
Maximum allowed processes:500000
Process spawn time=4.10000 (10.2150) microseconds
3> processes:max(300000).
Maximum allowed processes:500000
Process spawn time=4.13333 (73.6533) microseconds
```

在这个样例中，把系统进程数量的上限提到了500 000个。我们可以看到在50 000～200 000个进程的数量区间内，创建时间实质上是常量。在达到300 000个进程之后，每个进程创建的CPU时间依然是常数时间，但真实的流逝时间增加到了之前的7倍。此时我也能听到硬盘开始嘎嘎作响。这是一个很明确的信号，告诉我系统已经开始使用交换内存页，因为物理内存已经不足以容纳300 000个进程了。

8.5 带超时的receive

有时如果某个消息迟迟不来，receive语句可能会进入无限的等待之中。造成这种情况，有几种原因。比如，是由于程序内部有逻辑错误，或者那个准备向我们发送消息的进程在发送消息之前就已经崩溃了。

为了避免这种情况，可以在receive语句里加上一个超时。这个超时设定了进程接收一个消息时等待的最长时间。它的语法是这样的：

```
receive
     Pattern1 [when Guard1] ->
         Expressions1;
     Pattern2 [when Guard2] ->
         Expressions2;
     ...
after Time ->
     Expressions
end
```

142

如果在进入receive表达式后在Time所规定的毫秒数内没有收到能够匹配的消息，那么进程就会停止等待，并对Expressions进行求值。

8.5.1 只有超时的receive

你还可以写出一个只有超时的receive语句。使用这个语句，我们可以定义一个sleep(T)函数，用它让当前进程暂停Tms。

lib_misc.erl

```
sleep(T) ->
    receive
    after T ->
       true
    end.
```

8.5.2 超时时间为0的receive

一个超时时间为0的语句会立即触发一个超时，但在此之前，系统会尝试对邮箱进行模式匹配。我们可以用这个特性来定义一个flush_buffer函数，它可以完全清空进程邮箱中的所有消息：

lib_misc.erl

```
flush_buffer() ->
    receive
        _Any ->
            flush_buffer()
    after 0 ->
        true
    end.
```

如果没有超时子句，在邮箱为空的情况下，flush_buffer会永久暂停，而不会返回。我们还可以用0超时去实现一个“优先接收”的程序，就像这样：

lib_misc.erl

```
priority_receive() ->
    receive
        {alarm, X} ->
            {alarm, X}
    after 0 ->
        receive
            Any ->
                Any
        end
    end.
```

143

如果邮箱中没有任何消息可与{alarm,X}匹配，那么priority_receive函数会接收邮箱中的第一个消息。如果邮箱中没有任何消息，那么它会在最内层的receive上暂停，然后返回它收到的第一个消息。如果有一个消息可以与{alarm,X}匹配，那么这个消息就会即刻被返回。请牢记只有在邮箱中的所有消息都进行过模式匹配之后才会检查after段是否需要进行运算。

如果没有after 0子句，那么alarm消息就不会被最先匹配。

说明　在邮箱中有大量消息的情况下，使用这种方法来优先接收消息是非常低效的，因此，如果你计划使用这种技术，那么需要确认邮箱中没有大量的消息。

8.5.3　使用一个无限等待超时进行接收

如果receive语句的超时值被设定为原子infinity，那么系统就永远都不会触发超时。在那些超时值实际上是由receive以外的其他语句来决定的程序中，这种机制会非常有用。某些情况下，外部程序希望能返回一个正常的超时值，而另一些情况则可能希望receive永远等待。

8.5.4　实现一个计时器

我们可以使用接收超时来实现一个简单的计时器。

函数stimer:start(Time, Fun)会在Time指定的毫秒数后对Fun（一个0参数的函数）进行求值。它返回一个句柄（这里就是一个PID），这个句柄可以在需要的时候取消一个计时器。

stimer.erl

```
-module(stimer).
-export([start/2, cancel/1]).

start(Time, Fun) -> spawn(fun() -> timer(Time, Fun) end).

cancel(Pid) -> Pid ! cancel.

timer(Time, Fun) ->
```

```
    receive
        cancel ->
            void
    after Time ->
            Fun()
    end.
```

144

我们可以这样来进行测试：

```
1> Pid = stimer:start(5000, fun() -> io:format("timer event~n") end).
<0.42.0>
timer event
```

这里我们等待5秒钟以便触发计时器。

下面我们会启动一个计时器，然后在超时之前将其取消：

```
2> Pid1 = stimer:start(25000, fun() -> io:format("timer event~n") end).
<0.49.0>
3> stimer:cancel(Pid1).
cancel
```

8.6 选择性接收

至此我们仍停留在send、receive原语工作机制的表面。send其实并非是把一个消息传递到一个进程去，而是把一个消息发送到这个进程的邮箱中去。同理receive则是在试图把一条消息从进程邮箱中删除。

Erlang的每一个进程都有与之对应的邮箱。当向进程发送消息时，消息就被送入邮箱之中。当系统对receive语句进行求值时，就是对进程邮箱进行检查的唯一机会。

```
receive
    Pattern1 [when Guard1] ->
        Expressions1;
    Pattern2 [when Guard1] ->
        Expressions1;
     ...
after
    Time ->
        ExpressionTimeout
end
```

receive的内部工作机制是下面这样的。

(1) 当进入一个receive语句时，我们启动一个计时器（只有在表达式中存在after段时才会计时）。

(2) 从邮箱中取出第一个消息，然后尝试对Pattern1、Pattern2等进行模式匹配。如果匹配成功，消息就从邮箱中删除，对应的模式之后的表达式就会被求值。

(3) 如果邮箱中的第一个消息不能匹配receive语句的任何一个模式，那么就会将第一个消息从邮箱中删除并送入一个“保存队列”，然后继续尝试邮箱中的第二个消息。这个过程会不断重复直到找到匹配的消息，或者邮箱中所有的消息全都检查完毕。

145

(4) 如果邮箱中所有的消息都不能匹配，那么就挂起进程，直到下一次又有新的消息进入邮

箱时再对该进程进行重新调度执行。注意，当一个新消息到达时，只对新消息进行匹配而不会对保存队列中的消息进行再次匹配。

(5) 一个消息如果被匹配，那么存入保存队列中的所有消息就会按照它们到达进程的时间先后顺序重新放回到邮箱中。这时，如果开启了一个计时器，那么这个计时器就会被清空。

(6) 如果在我们等待一个消息时触发了计时器，那么对表达式ExpressionsTimeout求值然后把存入保存队列中的所有消息按照它们到达进程的时间先后顺序重新放回到邮箱中。

8.7 注册进程

如果想要向一个进程发送消息，那么就需要知道它的PID。不过这种方法通常很麻烦，因为这意味着系统中所有发送消息的进程都知道这个PID。另一个方面，这种做法在安全上也有问题，如果你不公开一个进程的PID，那么其他的进程就根本无法与它通信。

Erlang有一种机制可以用于发布一个进程的标识符以便其他进程可以与之通信，这种进程就叫做注册进程。Erlang中有4个BIF用于管理注册进程。

`register(AnAtom, Pid)`

将一个进程Pid注册一个名为AnAtom的原子，如果原子AnAtom已经被另一个注册进程所使用，那么注册就会失败。

`unregister(AnAtom)`

移除与AnAtom相对应进程的所有注册信息。

说明 如果一个注册进程死亡，那么它也会被自动取消注册。

`whereis(AnAtom) -> Pid | undefined`

判断AnAtom是否已经被其他进程注册。如果成功，则返回进程标识符Pid。如果AnAtom没有
146 与之相对应的进程，那么就返回原子undefined。

`registered() -> [AnAtom::atom()]`

返回一个系统中所有已经注册的名称列表。

可以使用注册器来对8.2节中的例子进行重构，也可以给创建的进程注册一个名字：

```
1> Pid = spawn(fun area_server0:loop/0).
<0.51.0>
2> register(area, Pid).
true
```

一旦进程注册了名字，我们就可以像下面这样用这个名字来向该进程发送消息：

```
3> area ! {rectangle, 4, 5}.
Area of rectangle is 20
{rectangle,4,5}
```

时钟程序

我们可以通过注册进程来实现一个时钟程序：

clock.erl

```
-module(clock).
-export([start/2, stop/0]).

start(Time, Fun) ->
    register(clock, spawn(fun() -> tick(Time, Fun) end)).

stop() -> clock ! stop.

tick(Time, Fun) ->
    receive
        stop ->
            void
    after Time ->
            Fun(),
            tick(Time, Fun)
    end.
```

这个时钟程序会不停地嘀嗒计时直到你停止它。

```
3> clock:start(5000, fun() -> io:format("TICK ~p~n",[erlang:now()]) end).
true
TICK {1164,553538,392266}
TICK {1164,553543,393084}
TICK {1164,553548,394083}
TICK {1164,553553,395064}
4> clock:stop().
stop
```

147

8.8 如何编写一个并发程序

编写一个并发程序的时候，我总是以这样的代码开始：

ctemplate.erl

```
-module(ctemplate).
-compile(export_all).

start() ->
    spawn(fun() -> loop([]) end).

rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        {Pid, Response} ->
            Response
    end.

loop(X) ->
    receive
        Any ->
```

```
            io:format("Received:~p~n",[Any]),
            loop(X)
    end.
```

接收循环仅是一个空循环，它会将接收到的所有消息都打印出来。编程时，我会先向进程发送消息。因为我启动的接收循环可以匹配所有的消息，因此能得到一些由最底层的receive语句打印出来的信息。看到这些结果之后，我们会向接收循环增加匹配模式，然后继续测试。这种技术在很大程度上决定了编码的顺序。我都会从一个小程序开始，然后慢慢地丰富它，测试它。

8.9　尾递归技术

再观察一下我们之前编写的area服务器中的接收循环：

area_server_final.erl

```
loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! {self(), Width * Ht},
            loop();
        {From, {circle, R}} ->
            From !  {self(), 3.14159 * R * R},
            loop();
    {From, Other} ->
        From ! {self(), {error,Other}},
        loop()
end.
```

148

如果你的观察足够仔细，就会发现每接收一个消息，就会处理这个消息，然后立刻再次调用loop()，这个过程就被称作*尾递归*。编译尾递归的函数可以使在一系列语句最后的一个函数调用可以被替换为一个简单的跳转指令，指向被调用函数的开头。这就意味着一个尾递归的函数可以无限循环而不需要消耗栈空间。

假定我们有如下的（错误）代码：

```
Line 1  loop() ->
        {From, {rectangle, Width, Ht}} ->
            From ! {self(), Width * Ht},
            loop(),
5           someOtherFunc();
        {From, {circle, R}} ->
            From !  {self(), 3.14159 * R * R},
            loop();
        ...
10  end
```

在第4行，我们调用了loop()，但是从编译器的角度推断，在调用loop()之后，还必须返回这里，因为在第5行还需要调用someOtherFunc()。因此，它会把someOtherFunc地址压入栈然后跳转到loop的开头。但这个代码的问题在于loop永远不会返回，而是会无限循环下去。因此，

每次执行到第4行，就会把另一个返回地址压入控制栈，最后系统就会消耗完所有的空间。

要避免这种情况也很容易，如果你在编写一个像loop()那样永不返回的函数F，那么请确保不要在调用F之后再去调用任何其他的东西，也不要在列表或者元组的构造器中使用F。

8.10 使用 MFA 启动进程

在绝大多数的程序中，我们都用spawn(Fun)来创建一个新进程。这在我们无须动态地更新代码时能够很好的工作。但有时我们还想要编写那些可以在运行时更新的代码。如果想要确保代码可以被很好地进行动态更新，那么就必须使用不同形式的spawn。 149

```
spawn(Mod, FuncName, Args)
```

它会创建一个新的进程。Args是一个形如[Arg1, Args2, ..., ArgN]的参数列表。新进程会从函数Mod:FuncName(Arg1, Arg2, ..., ArgN)开始执行。

以显式指明模块、函数名和参数列表（这称作MFA）的形式来启动一个新进程，对于软件升级来说，是一种正确的方法。它可以确保代码在编译后，处于运行状态时，仍然可以用新版本代码进行升级。而直接spawn函数而不采用这种显式MFA命名的情况，则无法获得动态代码更新的特性。关于这一问题的详情，请阅读附录E.4节。

8.11 习题

(1) 编写一个函数start(AnAtom,Fun)来把spawn(Fun)的结果注册为AnAtom。当两个并行的进程同时执行到start/2函数时，要确保代码能够正常工作。也就是说，这两个进程其中一个成功执行，而另一个必须执行失败。

(2) 编写一个环形基准测试。在一个环中创建N个进程。然后沿着环发送一条消息M次，最后总共发送N*M条消息。在N和M的不同取值下测试整个过程会消耗多长时间。

然后再用你所熟悉的其他编程语言编写一个类似的程序。整理并对比这两个结果。写一篇博客，发布到因特网上。

好了，现在你已经开始编写并发程序了！

下面我们会关注一下错误修复，还会学到如何使用链接、信号和进程退出陷阱，并通过这3种方式来编写并发编程下的容错代码。欲知详情如何，请看下章分解。 150

第9章 并发编程中的错误处理

前面我们已经看到了，在顺序编程中如何捕获异常。在本章中，我们会把这些错误处理机制拓展到并发编程的领域中来。

这是本书第二次谈到Erlang的错误处理机制，也是最后一次对这个话题进行深入探讨。要了解并发编程中处理异常的方法，我们需要引入3个新的概念：链接（link）、退出信号（exit signal）以及系统进程。

9.1 链接进程

如果一个进程在某种程度上依赖于另一个进程，那么它就需要时刻紧盯第二个进程的运行状态。使用BIF `link`是解决这类问题的方案之一。（另一种方法则是使用监视器，具体信息可以参考Erlang用户手册的相关章节）。

图9-1中的两个进程A和B，它们之间建立了链接（在图9-1中用连接AB的虚线来表示）。如果其中的一个进程调用了BIF `link(P)`（P的值是另一个进程的PID），那么这两个进程之间就建立了链接。一旦在两个进程之间建立链接，它们就会自动地互相监视。此时，若A消亡，系统就会向B发送一个叫做退出信号的东西。反之，若B消亡，则A也会收到这个信号。

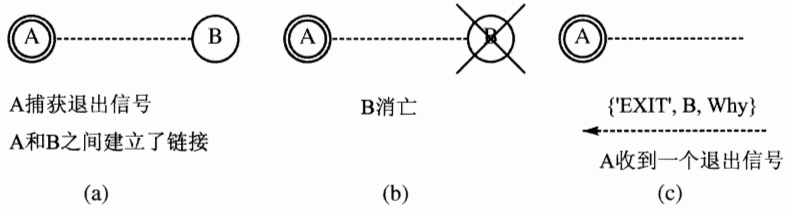


图9-1 退出信号与链接

上述这种机制在Erlang中的支持非常完备。它不仅在一个节点之内可以正常工作，在分布式的Erlang节点之间，这种机制同样能够工作。在第10章中我们会看到，在远程节点上启动一个进程与在当前节点上启动进程一样简单。本章之中，我们探讨的链接机制在分布式的计算环境下仍能正常工作。

如果一个进程接收到退出信号，会做什么事呢？若没有对这个接收进行特殊处理，那么这个

退出信号的默认处理就是让该进程也一并退出。但也可以让这个进程捕获退出信号。当进程进入这种捕获状态时，那么我们就称其为系统进程（system process）。如果一个进程链接到某个系统进程，由于某种原因而导致这个进程退出，那么链接的系统进程不会自动消亡，而是会收到一个退出信号，系统进程收到这个消息，可以再进行一些处理。

图9-1中的(a)，表示的是两个链接在一起的进程，其中的A是一个系统进程（用双圈表示）。(b)情况下，若进程B消亡，则如(c)所示，会向A发送一个退出信号。

本章的后续部分，我们会完整地介绍当进程接收到退出信号时会发生的详细情况。但在此之前，我们会用一个小的例子来演示如何使用这种机制来编写一个简单的退出处理程序。退出处理程序是一个进程，当其他进程崩溃时，它会执行一个特殊的函数。作为一种高层次的抽象结构，退出处理器本身也是一个非常有用的组件。

9.2 on_exit 处理程序

当一个进程退出时，若想执行一些动作。可以编写一个函数on_exit(Pid, Fun)，它会创建一个指向Pid进程的链接。如果Pid由于某种原因消亡（其原因在变量Why中），那么这个函数就会执行函数Fun(Why)。

152

下面是程序代码:

lib_misc.erl

```
on_exit(Pid, Fun) ->
    spawn(fun() ->
                  process_flag(trap_exit, true),
                  link(Pid),
                  receive
                      {'EXIT', Pid, Why} ->
                          Fun(Why)
                  end
          end).
```

在第3行，语句process_flag(trap_exit,true)把创建的进程变为一个系统进程。第4行的link(Pid)会把新建的进程与Pid指示的进程互相链接。最后，当这个进程消亡时，在第6行就会收到退出信号然后执行第7行的后续处理。

说明 阅读这段代码时，你会发现在每一处都使用名为Pid的变量来表示一个被链接进程的标识符。但其实我们把变量命名为LinkedPid并不恰当，因为在对link(Pid)求值之前，它还不是一个链接进程。当你接收到像{'EXIT', Pid, _}这样的消息时，它实际上就告诉你Pid是一个被链接进程，而且这个进程刚刚消亡。

要测试这段代码，我们可以定义一个函数F用于等待一个独立的消息X，然后对函数list_to_atom(X)求值：

```
1> F = fun() ->
        receive
          X -> list_to_atom(X)
        end
      end.
#Fun<erl_eval.20.69967518>
```

我们先像这样创建进程：

```
2> Pid = spawn(F).
<0.61.0>
```

然后再设定一个on_exit处理程序来监视这个进程：

```
3> lib_misc:on_exit(Pid,
                    fun(Why) ->
                         io:format(" ~p died with:~p~n",[Pid, Why])
                    end).
```

153

```
  <0.63.0>
```

若向Pid发送一个原子，那么这个进程就会崩溃（因为它是在用一个非列表参数来对list_to_atom求值），随后on_exit处理程序就会被调用：

```
4> Pid ! hello.
hello
<0.61.0> died with:{badarg,[{erlang,list_to_atom,[hello]}]}
```

进程消亡时所触发的这个函数可以执行任何你想要的运算：它可以忽略错误，记录错误日志，或者重启应用程序。具体的选择权实际上是交给了程序员。

9.3　远程错误处理

现在我们先停下来，回味一下前面这个样例。这个样例展示了Erlang哲学中极其重要的一面——远程错误处理。

由于一个Erlang系统是由大量并行进程组成的，而要在一个出错的进程中处理错误通常都会很困难。但现在有了新的选择，我们可以在一个与之无关的进程中进行这些处理。这个处理错误的进程甚至都不需要在同一台机器上。在下一章要谈到的分布式Erlang中，我们会看到这种简单的机制甚至可以跨越机器的界限正常运行。这是非常重要的，因为如果整个机器崩溃，那么处理这类错误的程序就必定不能运行在同一台机器上。

9.4　错误处理的细节

让我们来看看构成Erlang错误处理的机制的3种基础概念。

- **链接**（link）。链接定义了一种在两个进程之间的传播路径。如果两个进程被链接在一起，若其中一个进程消亡，那么系统就会向另一个进程发送一个退出信号。我们把一群与某个给定的进程进行链接的进程集合称为该进程的链接集。
- **退出信号**（exit signal）。当一个进程消亡时，它会产生一个叫做退出信号的东西。系统会

向这个濒死进程的链接集的所有进程发送退出信号。退出信号包含了一个参数来描述进程消亡的原因，这个原因可以是任何Erlang数据项。我们可以通过调用exit(Reason)原语 154
来显式地设置退出原因，或在一个错误生成时隐式地设置该值。例如，如果程序试图用一个数去除零，那么系统就会自动把退出原因设置为原子badarith。

当一个进程成功地执行完由spawn所指定的函数，那么在它消亡时就会把退出原因设为normal。

另外，你在进程Pid1中调用exit(Pid2,X)后，就可以让进程Pid1显式地向一个进程Pid2发送一个退出信号X。发送退出信号的进程并没有消亡，发送信号后它会继续执行。但是Pid2仍然会收到一个{'EXIT', Pid1, X}消息（前提是它正处于退出信号捕获状态），这与同源进程已经消亡时发生的情况完全相同。使用这种方法，Pid1可以装死（这通常是故意的）。

- **系统进程**（system process）。当进程接收到一个非正常的退出信号时它自己也会消亡，除非它是那种特殊类型的进程即系统进程。当Pid进程向一个系统进程发送一个内容为Why的退出信号时，系统会把退出信号转换为消息{'EXIT', Pid, Why}然后送入系统进程的邮箱。

  我们可以通过调用BIF process_flag(trap_exit,true)来把一个普通进程转换为一个可以捕获退出信号的系统进程。

当某个进程收到一个退出信号时，会产生数种不同的情况。具体的运行结果则依赖于接收进程的状态以及退出信号的值。它们之间的依赖关系表9-1。

表 9-1

捕获状态	退出信号	动 作
true	kill	消亡，向链接集广播退出信号killed
true	X	将{'EXIT', PID, X}加入到邮箱
false	normal	继续运行，不做任何事，屏蔽退出信号
false	kill	消亡，向链接集广播退出信号killed
false	X	消亡，向链接集广播退出信号X

如果把退出原因设为kill，那么进程就会发送一个无法捕获的退出信号。无论什么进程，哪怕是系统进程，只要它收到这个无法捕获的信号，就会被终止。OTP中的监管进程就是使用这种方法来终止僵尸进程的。当一个进程收到kill信号时，该进程自行消亡然后向它链接集中的所有进程广播killed信号。这种方法非常安全，可以避免误杀你不想终止的进程。 155

kill信号主要用于终止僵尸进程，因此请谨慎地使用它们。

9.4.1 捕获退出的编程模式

通过学习前面几节，你可能认为退出信号的捕获远不如你想象中的那么简单，但事实并非如此。确实有一些程序在生成和捕获退出信号上做得极为精巧，但大多数的程序普遍使用的都是下

面3种简单模式中的一种。

1. 模式1：我不在乎创建的进程是否崩溃

下面这个进程仅使用spawn来创建一个并行进程：

```
Pid = spawn(fun() -> ... end)
```

如果没有其他语句。那么如果这个被生成的进程崩溃，当前进程会毫无察觉地继续执行自己的其他任务。

2. 模式2：如果我创建的进程崩溃那么我也自行消亡

严格地讲，这里应改为“如果我创建的进程非正常的崩溃”。要达到这个目的，必须使用spawn_link来创建并行进程，而且一定不能在这之前将进程设为退出信号捕获状态。我们应该这么写：

```
Pid = spawn_link(fun() -> ... end)
```

此时，如果新生成的进程异常崩溃而发送非正常的退出信号，当前进程也会随之消亡。

3. 模式3：如果我创建的进程崩溃我需要处理错误

在这种情况下，就要使用spawn_link和trap_exits，代码如下：

```
...
process_flag(trap_exit, true),
Pid = spawn_link(fun() -> ... end),
...
loop(...).

loop(State) ->
    receive
        {'EXIT', SomePid, Reason} ->
              %% do something with the error
              loop(State1);
        ...
    end
```

156

如果把经过设置的执行loop函数的进程链接到另外一个进程，那么loop进程虽然会捕获到退出信号但是不会自行消亡。另一个进程即将消亡时，loop进程会捕获到所有来自濒死进程的退出信号[①]并且将其翻译为消息，然后执行任何它所期望的后续处理。

9.4.2 捕获退出信号（进阶篇）

你可以在第一次阅读本书的时候跳过这个小节。前一节中描述的3种模式应该已经足够应付编程时会遇到的大部分需要进行错误处理的情况。如果你仍然想知道并理清本章所述的全部内容，那么请继续阅读。不过需要提醒的是，要搞清楚本节所述错误处理机制的精确运作细节并不是一件容易的事情。而且，在绝大部分的情况下，你也无须了解它们，因为如果你采用前面一节所描述的那些通用编程模式或者OTP库，那么无须担心，系统自己就能准确无误地完成你所期望的工作。

① 除了由exit(Pid,kill)产生的信号。——译者注

要弄清错误处理的细节，我们需要写一个小程序来展示错误处理和链接交互的运作机制。先从下面这个小程序开始：

edemo1.erl

```
-module(edemo1).
-export([start/2]).

start(Bool, M) ->
    A = spawn(fun() -> a() end),
    B = spawn(fun() -> b(A, Bool) end),
    C = spawn(fun() -> c(B, M) end),
    sleep(1000),
    status(b, B),
    status(c, C).
```

这个程序会启动3个进程：A、B和C。然后会把A链接到B，再把B链接到C。这样A可以监视来自B的退出信号并且捕获它们。当Bool为true同时当C消亡的原因是M时，B也可以捕获退出信号。

（你可能不清楚sleep(1000)语句的具体作用。它的作用就是当C消亡时，保证在我们检查3个进程状态之前，可以把收到的任何消息都打印出来。它不会干扰程序的原有运行逻辑，不过的确影响到了打印输出的顺序。）[①] 157

A、B、C进程的代码如下：

edemo1.erl

```
a() ->
    process_flag(trap_exit, true),
    wait(a).

b(A, Bool) ->
    process_flag(trap_exit, Bool),
    link(A),
    wait(b).

c(B, M) ->
    link(B),
    case M of
        {die, Reason} ->
            exit(Reason);
        {divide, N} ->
            1/N,
            wait(c);
        normal ->
            true
    end.
```

wait/1函数只打印它接收到的所有消息：

① 使用sleep来同步进程并不是一个安全的方法。在这样的小例子中它也可以正常工作，但对于需要产品化的代码，就应该采用显式同步机制。——译者注

edemo1.erl

```
wait(Prog) ->
    receive
        Any ->
            io:format("Process ~p received ~p~n",[Prog, Any]),
            wait(Prog)
    end.
```

其余的程序代码如下：

edemo1.erl

```
sleep(T) ->
    receive
    after T -> true
    end.

status(Name, Pid) ->
    case erlang:is_process_alive(Pid) of
        true ->
            io:format("process ~p (~p) is alive~n", [Name, Pid]);
        false ->
            io:format("process ~p (~p) is dead~n", [Name,Pid])
    end.
```

158

现在运行程序，让进程C生成不同的退出信号，然后看看这些信号在B中会产生什么影响。在运行程序时，你可能要参考图9-2，它演示了从C发出的退出信号到底做了哪些工作。图9-2中的(a)框、(b)框、(c)框都告诉我们哪个程序退出了，它们是否是系统进程，以及它们是如何链接在一起的。图9-2中的(a)框、(b)框、(c)框都可分成两个部分，“接收前”部分（即每幅图的上半部分），展示了收到退出信号之前的进程状态，“接收后”部分（即(a)框、(b)框、(c)框内的图的下半部分）展示了进程收到退出信号后的状态。

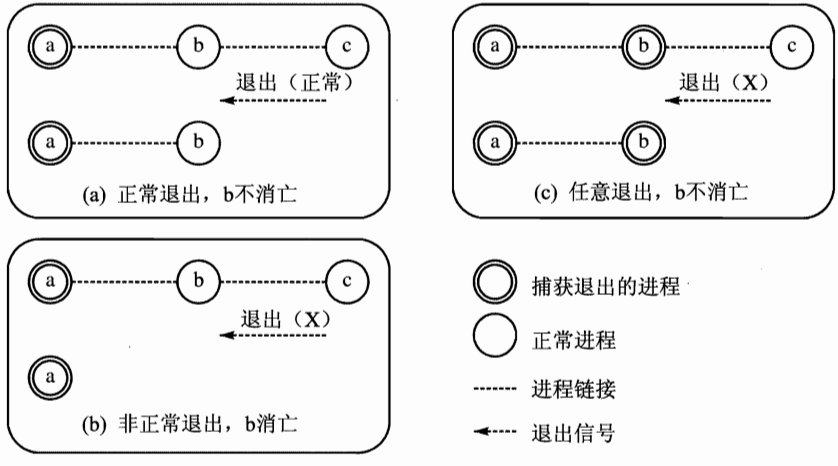


图9-2　捕获退出信号

首先假定B是一个普通进程(也就是那些没有运行过process_flag(trap_exit,true)的进程):

```
1> edemo1:start(false, {die, abc}).
Process a received {'EXIT',<0.44.0>,abc}
process b (<0.44.0>) is dead
process c (<0.45.0>) is dead
ok
```

当C执行exit(abc)时，进程B消亡（因为它没有设为捕获退出信号的状态）。当B退出时，B把收到的退出信号不加修改地重新广播给它链接集内的所有进程。A（被设置为捕获退出状态）收到退出信号，然后将其转换为错误消息{'EXIT',<0.44.0>,abc}。（注意，进程<0.44.0>就是进程B，因为此时消亡的进程是B。）

159

让我们来看另外一种情形。下面让C消亡，但此时把消亡的原因设为normal：[①]

```
2> edemo1:start(false, {die, normal}).
process b (<0.48.0>) is alive
process c (<0.49.0>) is dead
ok
```

这个时候，B没有消亡，因为它接收到的是一个normal的退出信号。

现在再让C产生一个除零错误：

```
3> edemo1:start(false, {divide,0}).
=ERROR REPORT==== 8-Dec-2006::11:12:47 ===
Error in process <0.53.0> with exit value: {badarith,[{edemo1,c,2}]}
Process a received {'EXIT',<0.52.0>,{badarith,[{edemo1,c,2}]}}
process b (<0.52.0>) is dead
process c (<0.53.0>) is dead
ok
```

当C尝试除零，就会抛出一个错误，进程会因为{badarith,...}错误而消亡。B接收到这个错误信号也会消亡，并且把退出消息广播给A。

最后，我们让C退出并且把Reason设为kill：

```
4> edemo1:start(false, {die,kill}).
Process a received {'EXIT',<0.56.0>,killed} <-- ** changed to killed **
process b (<0.56.0>) is dead
process c (<0.57.0>) is dead
ok
```

退出原因kill会导致B消亡，然后B会把退出原因设置为killed然后广播给B的链接集。图9-2的(a)框和(b)框内的图，对本例中的错误处理机制进行了非常形象地阐述。

我们再把B设置为退出捕获状态，然后重复这些测试。图9-2的(c)框中的图就详细描述了这种情况下的运作机制。

```
5> edemo1:start(true, {die, abc}).
Process b received {'EXIT',<0.61.0>,abc}
process b (<0.60.0>) is alive
process c (<0.61.0>) is dead
```

① 一个进程正常退出，其效果与调用函数exit(normal)相同。

```
ok
6> edemo1:start(true, {die, normal}).
Process b received {'EXIT',<0.65.0>,normal}
process b (<0.64.0>) is alive
process c (<0.65.0>) is dead
ok
7> edemo1:start(true, normal).
Process b received {'EXIT',<0.69.0>,normal}
process b (<0.68.0>) is alive
process c (<0.69.0>) is dead
8> edemo1:start(true, {die,kill}).
Process b received {'EXIT',<0.73.0>,kill}
process b (<0.72.0>) is alive
process c (<0.73.0>) is dead
ok
```

160

各种情况下，B都捕获到了退出消息。B的行为类似于一个“防火墙”，捕获了C进程的所有退出信息，并阻止它散播到A进程。我们可以用code/edemo2.erl来测试exit/2。这个程序和edemo1非常类似，不同之处在于它的c/2函数，在这里它调用exit/2，代码是这样的：

edemo2.erl

```
c(B, M) ->
    process_flag(trap_exit, true),
    link(B),
    exit(B, M),
    wait(c).
```

运行edemo2，观察到如下的输出：

```
1> edemo2:start(false, abc).
Process c received {'EXIT',<0.81.0>,abc}
Process a received {'EXIT',<0.81.0>,abc}
process b (<0.81.0>) is dead
process c (<0.82.0>) is alive
ok
2> edemo2:start(false, normal).
process b (<0.85.0>) is alive
process c (<0.86.0>) is alive
ok
3> edemo2:start(false, kill).
Process c received {'EXIT',<0.97.0>,killed}
Process a received {'EXIT',<0.97.0>,killed}
process b (<0.97.0>) is dead
process c (<0.98.0>) is alive
ok
4> edemo2:start(true, abc).
Process b received {'EXIT',<0.102.0>,abc}
process b (<0.101.0>) is alive
process c (<0.102.0>) is alive
ok
```

```
5> edemo2:start(true, normal).
Process b received {'EXIT',<0.106.0>,normal}
process b (<0.105.0>) is alive
process c (<0.106.0>) is alive
ok
6> edemo2:start(true, kill).
Process c received {'EXIT',<0.109.0>,killed}
Process a received {'EXIT',<0.109.0>,killed}
process b (<0.109.0>) is dead
process c (<0.110.0>) is alive
ok
```

161

9.5 错误处理原语

下面这些是用于操纵链接和发送/捕获退出信号的常用原语。

```
@spec spawn_link(Fun) -> Pid
```

这个原语的确非常像spawn(Fun)，但它除了创建进程之外还会在这两个进程之间创建链接（spawn_link是一个原子操作，它和spawn+link操作并不等价。因为在执行spawn和link的间歇之中，所创建的进程有可能就已经消亡了）。

```
@spec process_flag(trap_exit, true)
```

这个原语把当前进程转换为系统进程，系统进程可以接收和处理退出信号。

说明 在把一个进程的trap_exit标志设置为true之后，你还可以再把这个标志设为false。这个原语只应该用于把一个普通进程转换为系统进程，最好不要在其他情况下使用。

```
@spec link(Pid) -> true
```

如果Pid所指示的进程还没有链接，那么给这个进程创建一个链接。链接是双向的，如果一个进程A运行了link(B)，那么它自身也会被链接到B。这个操作的实际效果其实等同于在B中运行了link(A)。

如果Pid不存在，那么系统就会抛出noproc退出异常。

如果A已经被链接到B，你试图运行link(B)（或者相反），系统会忽略这个调用。

```
@spec unlink(Pid) -> true
```

这会移除当前进程和Pid所指进程间的链接。

```
@spec exit(Why) -> none()
```

这会导致当前的进程终止，并把终止原因设为Why。如果一个运行该语句的子句并不位于catch语句的范围之内，那么当前进程就会向当前正与它链接在一起的所有进程广播一个退出信号，其中退出原因的值就是Why。

162

Joe问答……

我们如何去构建一个容错系统

要使得一个系统具有容错性，我们至少需要两台计算机。一台计算机运行日常工作，由另

外一台计算机来监视第一台计算机并且时刻准备在它崩溃时接管工作。

这其实也是Erlang中错误恢复的工作机制。一个进程运行日常工作，另外一个进程监视第一个进程的运行，并且在错误发生时接管任务。这也就是为什么我们需要监视进程，以及为什么我们需要知道故障的原因。本章之中的例子已经告诉你要如何去处理它们了。

在分布式Erlang中，具体工作的进程和监视进程可能在物理上分布在不同的机器上。使用这种技术，我们就可以开始设计容错系统了。

这种模式非常常见，我们称之为“工人-监工”模型，整个的OTP库都构建于监控树的概念之上，而监控树正是基于这种思想来构建的。

可以说，Erlang的整个容错特性及其体系从根本上都依赖于link原语。

一旦弄清了link的工作机制，并学会如何在两台计算机之间互相访问，那么你的第一个容错系统就已经是万事俱备只欠东风了。

```
@spec exit(Pid, Why) -> true
```

这个原语向Pid所指示的进程发送一个退出信号并且把退出原因设置为Why。

```
@spec erlang:monitor(process, Item) -> MonitorRef
```

这个原语建立了一个监视器。Item为一个PID或者一个进程的注册名。关于这个原语的更多信息请查阅erlang用户手册。

163

9.6　链接进程集

假定我们有一个参与某种计算的大规模的并行进程集，在某个时刻它们之中发生了一些错误。那么我们要如何找到所有与之相关的进程并把它们终止呢？

最简单的办法是把你想要终止的所有进程归入一个互相链接的群，并且设置为不捕获退出信号状态。如果任何一个进程因为一个非正常原因而终止，那该群之中的所有进程就会消亡。

图9-3形象地描述了这种行为。在框(a)中展示了一个包含9个进程的集合，其中的2、3、4、6和7互相链接。如果其中的任何一个非正常终结，那么整个进程群都会终结，如框(b)所演示的结果。

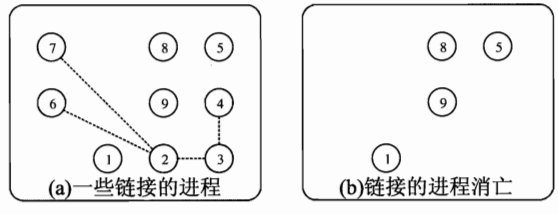


图9-3　捕获退出信号

链接进程集通常被用来构建成一个容错的软件系统。要构建这样的系统，你既可以自行设计，也可以使用18.5节中描述的库函数。

9.7　监视器

有些情况下，因为链接是对称的，你需要使用一些技巧才能处理好有关于链接的编程。如果

A消亡，B会收到一个退出信号，反之亦然。为防止一个进程的消亡，我们不得不将其变成系统进程，但有的时候我们也许并不希望这么做。这种情况相当少见，此时我们可以使用监视器。 164

监视器是一个非对称的链接。如果进程A监视进程B，B死亡，A会收到一个退出信号。但是如果A死亡，B就不会收到退出信号。我们可以在erlang用户手册里查询到创建一个监视器的具体方法。

9.8 存活进程

本章结束之前，我们要来编写一个存活进程。这个程序会创建一个注册进程，并让它永远保持存活——它无论因为什么原因消亡，都会被立刻重启。

可以用on_exit来实现它：

lib_misc.erl

```
keep_alive(Name, Fun) ->
    register(Name, Pid = spawn(Fun)),
    on_exit(Pid, fun(_Why) -> keep_alive(Name, Fun) end).
```

这个程序创建了一个叫做Name的注册进程，这个进程会运行spawn(Fun)。如果该进程因为某种原因而消亡，它就会被重启。

on_exit和keep_alive函数里其实有一个非常微妙的错误。你有没有注意到这个问题？当我们这么写的时候：

```
Pid = register(...),
on_exit(Pid, fun(X) -> ..),
```

进程有可能在两个语句之间死亡。如果在on_exit调用之前进程就已经死亡，那么就不会建立任何链接，也就是说on_exit不会如你预期的那样工作。如果两个程序使用同一个Name值，而且试图在同一个时刻运行keep_alive程序时，就会发生这种状况。这种情况就称为竞争性条件——两块代码，一块当前进程运行的代码块和on_exit内部建立链接操作的代码块互相竞争。如果这里出了错，程序就会去做一些意料之外的行为。

我不打算在这里解决这个问题，我希望你能自己思考如何处理它。当把spawn、spawn_link、register等Erlang原语组合在一起的时候，你必须仔细的考虑可能发生的竞争条件。用这种方法去编写程序，就会避免出现这类错误。

幸运的是，OTP库有许多构建服务器和监控树的代码。这些库都经过良好的测试，不会出现竞争条件的情况。最好使用这些库来构建应用程序。 165

我们已经学习了Erlang编程中所有有关错误侦测和捕获的机制。在下一章中我们会使用这些机制来构建可以从错误中恢复的软件系统。现在请把我们的眼光从单处理器系统上的编程转移到下一章，去领略一下简单的分布式编程。 166

第10章 分布式编程

在本章中，我们会介绍在编写分布式Erlang程序时会用到的库和原语。分布式编程是针对网络上仅通过消息传递来完成互相协作的计算机集群所设计的程序。

我们为何需要编写分布式的应用程序呢？因为我们有下面几种需求。

- 效率

我们把程序分成多个部分，并行地运行在不同机器上，这可以让程序执行得更快。

- 可靠性

通过把一个系统分布运行到若干台机器上，我们可以构建容错系统。如果某台计算机崩溃，那么这个系统会由其他计算机接管继续运行。

- 可伸缩性

由于我们会不断地扩展应用程序，哪怕是最强大的计算机，它的计算资源迟早都会有耗尽的一天。在这种情况下，我们必须能通过增加机器来扩充容量。增加一台新的机器应该是一项简单的工作，而不需要伤筋动骨地修改应用程序的架构。

- 天生需要分布式的应用程序

很多应用程序天生就需要分布式。如果我们编写一个网络游戏或者聊天系统，用户可能遍布全球。如果在某个特定地区，用户群足够庞大，那么我们就会倾向于在靠近用户的地方配置更多的计算资源。

- 乐趣

大多数我想要编写的有趣程序都是分布式的。这些程序很多都会涉及与世界各个角落的机器和人进行沟通的任务。

这本书里我们会讨论分布式编程的两种基本模型。

- 分布式Erlang：这种模型可以让我们在一个紧密耦合的计算机集群上编写程序。[1]在分布式Erlang中，我们要编写可运行在不同Erlang节点之上的程序。我们可以在任何一个节点创建进程，这与我们前几章中讨论单节点编程时的情形一致，所有的消息传递和错误处理原语都可以照搬套用。

① 例如，那些为了解决特定问题而运行在同一个局域网内的计算机。——译者注

- 分布式Erlang应用程序运行在一个可信任的环境中——因为任何节点都可以运行其他节点上的操作，这就需要一个高度信任的网络环境。尤其是在防火墙的保护下，Erlang应用程序可以运行在同一个局域网的不同集群之间，虽然这些集群都对外网公开。
- 基于套接字的分布式应用：使用TCP/IP套接字，我们可以编写运行在非信任环境中的分布式应用程序。这种模式的编程模型，其功能比分布式Erlang要弱一些，但是却更加可靠。在10.5节中，我们会学到如何采用一种简单的基于套接字的分布模型来构建应用程序。

如果回顾一下前几章的内容，你一定会记得我们构建程序所用的最基本元件就是进程。分布式的Erlang程序的构建是很容易的，我们所要做的只是在正确的计算机上创建进程，此后它们的运行方式就和之前一样了。因此从这个角度来说，编写一个分布式Erlang程序并不是一个复杂的任务。

然而我们更习惯于编写顺序型程序。从这一个角度来说，编写分布式程序不是一件简单的任务。在本章中，我们会运用许多新的技术来编写一些简单的分布式程序。尽管这些程序很简单，却很有用。

我们会从一系列的小例子开始。为此，需要先学习两件事情，然后就可以编写我们的第一个分布式程序。通过这个过程，我们会学到如何启动一个Erlang节点，如何在一个远程的Erlang节点上执行一个远程调用。 168

在开发一个分布式应用程序时，我总是以下面这样的步骤来编写程序。

(1) 先在一个非分布式Erlang环境中编写和测试程序。到目前为止，我们一直都在用这种方法来编写Erlang程序，因此这不会增加什么新难度。

(2) 然后会在同一台计算机的两个不同Erlang节点上测试程序。

(3) 最后才会在同一网络或者因特网上的两台互相独立的机器上开启不同的Erlang节点来测试程序。

最后一步的工作可能不会一帆风顺。尤其是当运行程序的节点分别由不同的域来控制时，我们必须确保系统防火墙和安全设置正确无误，否则机器之间的连通性问题就会接踵而来。当然，如果我们的机器是运行在同一个域内的，这样的麻烦就会大大减少。

下一节中，我们会编写一个简单的名字服务器，参照上面的方法，按部就班地按照下列步骤来完成这个任务。

- 第1步：在一个普通的无分布的Erlang系统上编写和测试名字服务。
- 第2步：在同一台机器的两个不同节点上编写和测试名字服务。
- 第3步：在同一本地网络两台不同的机器上用两个不同节点来测试名字服务。
- 第4步：在两个城市间不同域上的两台机器上测试名字服务。

10.1 名字服务

我们向一个服务提交一个名字，然后该服务器向我们返回与这个名字相关联的值，这样的服务就叫做名字服务。当然我们也能通过名字服务来修改给定名字所对应的值。

第一个名字服务极为简单。这个小例子的目的并非是要编写一个容错的名字服务而是为了让

我们开始熟悉分布式编程技术。因此它不具备容错性，也就是说在它崩溃时它存储的全部数据都
169 会丢失。

10.1.1 第一步：一个简单的名字服务

我们的名字服务kvs是一个简单的Key↦Value服务。它的接口如下。

`@spec kvs:start() -> true`

启动服务器，这个函数会创建服务并且把服务的名字注册为kvs。

`@spec kvs:store(Key, Value) -> true`

将值Value与键Key关联起来。

`@spec kvs:lookup(Key) -> {ok, Value} | undefined`

查找与键Key相关的值，如果存在与键Key关联的值那么返回{ok,Value}，否则返回undefined。

我们会使用进程字典的get和put原语来实现键值服务，代码如下：

socket_dist/kvs.erl

```
-module(kvs).
-export([start/0, store/2, lookup/1]).

start() -> register(kvs, spawn(fun() -> loop() end)).

store(Key, Value) -> rpc({store, Key, Value}).

lookup(Key) -> rpc({lookup, Key}).

rpc(Q) ->
    kvs ! {self(), Q},
    receive
        {kvs, Reply} ->
            Reply
    end.

loop() ->
    receive
        {From, {store, Key, Value}} ->
            put(Key, {ok, Value}),
            From ! {kvs, true},
            loop();
        {From, {lookup, Key}} ->
            From ! {kvs, get(Key)},
            loop()
    end.
```

我们先在本地开始测试，看看服务器能否正常工作：

```
1> kvs:start().
true
```

```
2> kvs:store({location, joe}, "Stockholm").
true
3> kvs:store(weather, raining).
true
4> kvs:lookup(weather).
{ok,raining}
5> kvs:lookup({location, joe}).
{ok,"Stockholm"}
6> kvs:lookup({location, jane}).
undefined
```

170

到目前为止我们还没有遇到什么出人意料的难题。

10.1.2 第二步：在同一台机器上，客户端运行于一个节点而服务器运行于第二个节点

现在在同一台计算机上启动两个Erlang节点。为此，需要开启两个终端窗口，在其中分别启动两个Erlang系统。

首先，我们打开一个终端shell，[①]然后在终端上启动一个名为gandalf的Erlang节点。在这个节点上启动服务：

```
$ erl -sname gandalf
(gandalf@localhost) 1> kvs:start().
true
```

Windows用户注意 在Windows中显示的节点名后缀并不一定就是localhost，如果发生这种情况，那么必须把后面出现的命令中的localhost部分用shell返回的名字替换。

参数-sname gandalf的意思是在本地主机上启动一个名为gandlf的Erlang节点。注意Erlang shell是如何在命令提示符前打印Erlang节点[②]的名称的。

然后，我们再打开第二个终端窗口，启动一个名为biblo的Erlang节点。然后使用库模块rpc来调用kvs模块中的函数。（注意rpc是一个Erlang标准库中的模块，它与我们先前编写的rpc函数没有任何关系）。

```
$ erl -sname bilbo
(bilbo@localhost) 1> rpc:call(gandalf@localhost,
                              kvs,store, [weather, fine]).
true
(bilbo@localhost) 2> rpc:call(gandalf@localhost,
                              kvs,lookup,[weather]).
{ok,fine}
```

171

现在这个程序还只是初具雏形，但这毕竟是我们从本书开始以来第一次接触分布式计算。服

① Windows用户，请阅读附录B。只要你打开一个shell窗口，系统就会运行erl-name Node命令。请确保环境变量中的PATH设置准确，以便你可以找到erl.exe（它的路径一般是这样的，C:\ProgramFiles\erl5.4.4\bin\erl.exe）。

② 节点名的形式形如Name@Host。Name和Host都是原子，因此如果其中含有非原子字符时一定要用引号引起来。

务器运行于我们第一次启动的节点，而客户机则运行在第二个节点。

我们在bilbo节点上进行一次设定weather值的操作，然后我们可以切换回gandalf节点验证weather的值：

```
(gandalf@localhost)2> kvs:lookup(weather).
{ok,fine}
```

函数rpc:call(Node, Mod, Func, [Arg1, Arg2, ..., ArgN]) 在Node上执行一个远程调用。被调用的函数是Mod:Func(Arg1, Arg2, ...,ArgN)。

正如我们看到的，这两个程序的运行情况与非分布式Erlang系统上的运行情况如出一辙。所不同的只是客户机运行在一个节点而服务器运行在另外一个节点。

下一步我们要做的就是把客户机和服务器分别放在两个不同的机器上。

10.1.3　第三步：让客户机和服务器运行于同一个局域网内的不同机器上

我们要准备使用两个节点，第一个节点是doris.myerl.example.com上的gandlf，另外一个节点是george.myerl.example.com上的bilbo。在这之前，我们分别在两个不同机器上启动两个终端窗口[①]。我们把这两个窗口分别叫做doris和george。在完成这些工作之后，我们可以很简单地在两台机器上输入下面这些命令。

步骤1：在doris上启动一个Erlang节点。

```
doris $ erl -name gandalf -setcookie abc
(gandalf@doris.myerl.example.com) 1> kvs:start().
true
```

步骤2：在george上启动一个Erlang节点并向gandalf发送一些命令。

```
george $ erl -name bilbo -setcookie abc
(bilbo@george.myerl.example.com) 1> rpc:call(gandalf@doris.myerl.example.com,
                                             kvs,store,[weather,cold]).
true
(bilbo@george.myerl.example.com) 2> rpc:call(gandalf@doris.myerl.example.com,
                                             kvs,lookup,[weather]).
{ok,cold}
```

172　这两个程序的运作方式就和同一台机器不同节点上的运行情况一模一样。

不过为了保证上面的这些步骤顺利进行，我们做了一些比在同一台机器上两个节点的情形更为复杂的设置，这里一共有4个步骤。

(1) 以-name参数启动Erlang。如果在同一台机器上启动两个节点，使用“短”名（用-sname标志指定），但是如果它们运行在不同的网络上，就需要使用-name。

如果两台机器都在同一个子网内工作，也可以使用-sname。换言之，只有在不需要DNS服务的情况下，才能使用-sname。

(2) 确保两个节点之间使用相同的cookie。这就是为何两个节点的启动都要使用命令行参数

① 使用类似ssh的工具。

-setcookie abc。（我们会在本章稍后部分介绍cookie）[①]

(3) 确保使用的节点主机名称正确无误，并且能被DNS正确解析。在这个例子中，域名myerl.example.com是直接对应本地网络，你可以在本地的/etc/hosts里加入想对应的域名入口点。

(4) 确保两个系统之中需要运行的代码版本相同[②]。在这个例子里，也就是要确保在两个系统中运行的kvs代码版本一致。我们有如下几种方式来处理这个问题。

(a) 在我家中的环境里，有两台物理独立没有共享文件的机器。这种情况下我用磁盘在两台机器之间复制kvs.erl然后编译并启动它们。

(b) 在我工作的环境中，我使用两个有共享NFS磁盘的工作站。这种情况下只要在两台不同的工作站上相同的共享目录里启动Erlang。

(c) 配置代码服务器来保证版本的一致。我不想在这里讨论设置的具体方法，请自行参阅Erlang用户手册中模块erl_prim_loader的相关文档。

(d) 使用shell 命令nl(Mod)。这个操作会在所有互相连接的节点上加载模块Mod。 173

说明 为了能让上面的例子工作，必须确保所有的节点都已经互相连接。在它们第一次去试图访问对方时，就会在节点之间建立连接。这种情况发生在你第一次去对一个远程节点上的表达式求值的时候。最简单的方法就是执行命令net_adm:ping(Node)（详情请参见net_adm的用户手册）。

10.1.4 第四步：在因特网上的不同主机上分别运行客户机和服务器

原则上，这一步与第三步没有多少差别，但我们必须更多的关注一下链接的安全问题。当我们在同一个局域网内运行两个节点时，可以不必过多的关心安全问题。因为大多数的机构都通过防火墙将局域网与因特网隔离。在防火墙的后面我们可以自由地随机地分配IP地址，而不会导致我们的计算机配置错误。

当我们向一个因特网上的Erlang集群中的某几台机器发起连接时，我们应该估计到有可能会发生由于防火墙拒绝外来链接而导致的问题。我们必须正确地配置我们的防火墙以便能够接受外来的链接。在这个问题上，并没有什么通用的方法来解决问题，因为不同的防火墙都有不同的配置。

要准备好分布式的Erlang系统，你需要按照下面几个步骤来进行配置。

(1) 确保4396端口的TCP和UDP的通信正常。Erlang系统中的epmd（Erlang Port Mapper Daemon的缩写）程序会使用这个端口。

(2) 选择分布式Erlang需要使用的一个端口或者一个端口范围，确保这些端口在防火墙上打开。如果这些端口是Min到Max（如果只有一个端口则使用Min==Max），那么使用下列命令启动Erlang：

① 当在同一台机器的两个不同节点上运行程序时，两个节点都可以访问到同一个cookie文件$HOME/.erlang.cookie，这也就是为什么我们不用在Erlang命令行中加入cookie的原因。

② 还要确保Erlang的版本一致，否则你可能会遇到很多严重而且奇怪的错误。

```
$ erl -name ... -setcookie ... -kernel inet_dist_listen_min Min \
                                       inet_dist_listen_max Max
```

10.2 分布式原语

分布式Erlang的核心概念是节点。一个节点就是一个自给自足的系统，它是一个包含了地址空间和独立进程集的完整虚拟机。

访问单个节点或者节点集都受到cookie系统的保护。每个节点都有一个cookie，而且必须保
174 证所有要和这个节点通信的其他节点都有相同的cookie。为此，在一个分布式的Erlang系统中，所有节点都必须用相同的magic cookie来启动，或者执行erlang:set_cookie来将它们的cookie改为相同的值。

具有相同cookie而且彼此互相连接的节点集称为Erlang 集群。

下面这些BIF在分布式编程中常常会用到。①

```
@spec spawn(Node, Fun) -> Pid
```

这个原语和spawn(Fun)没有多大区别，但新创建的进程位于Node节点之上。

```
@spec spawn(Node, Mod, Func, ArgList) -> Pid
```

这个原语和spawn(Mod, Func, ArgList)没有多大区别，不过新创建的进程位于Node节点上。spawn(Mod, Func, Args)原语创建一个会执行apply(Mod, Func, Args)的新进程，它返回新进程的PID。

说明 这种形式的spawn会比spawn(Node, Fun)更健壮。当分布式节点上运行的模块版本不一致时spawn(Node, Fun)会失败。

```
@spec spawn_link(Node, Fun) -> Pid
```

这个原语和spawn_link (Fun)没有多大区别，新创建的进程会在Node节点上运行。

```
@spec spawn_link(Node, Mod, Func, ArgList) -> Pid
```

这个原语和spawn(Node,Mod,Func,ArgList)没有多大区别，只是新创建的进程会与当前进程互相链接。

```
@spec disconnect_node(Node) -> bool() | ignored
```

强制断开一个节点的连接。

```
@spec monitor_node(Node, Flag) -> true
```

如果Flag为true就会打开监视，如果Flag为false就会关闭监视。在监视被打开时若有新的Node加入或者离开Erlang集群，执行这个BIF的进程就会收到{nodeup,Node}和{nodedown,Node}
175 消息。

```
@spec node() -> Node
```

这个原语会返回本地节点的名字。如果本地节点不是分布式的，就会返回nonode@nohost。

---

① 这些BIF的完整信息，请参见erlang模块的用户手册。——译者注

```
@spec node(Arg) -> Node
```

这个原语会返回Arg所指定的节点。Arg可以是一个PID、一个引用或者一个端口。如果本地节点不是分布式的，就会返回nonode@nohost。

```
@spec nodes() -> [Node]
```

这个原语会返回网络上与当前节点连接的所有其他节点列表。

```
@spec is_alive() -> bool()
```

如果本地节点状态正常而且是分布式系统的一个部分，那么这个原语就会返回true，否则返回false。

另外，我们可以使用send原语来向一个在分布式Erlang节点集中注册的进程发送消息。下面这个语法的意思就是，向节点Node上名为RegName的注册进程发送消息Msg。

```
{RegName, Node} ! Msg
```

一个启动远程进程的样例

下面这个简单的例子，会向我们展示如何在远程节点上创建一个进程。首先我们编写下面这个程序：

dist_demo.erl

```
-module(dist_demo).
-export([rpc/4, start/1]).

start(Node) ->
    spawn(Node, fun() -> loop() end).

rpc(Pid, M, F, A) ->
    Pid ! {rpc, self(), M, F, A},
    receive
        {Pid, Response} ->
            Response
    end.

loop() ->
    receive
        {rpc, Pid, M, F, A} ->
            Pid ! {self(), (catch apply(M, F, A))},
            loop()
    end.
```

然后我们启动两个节点，两个节点都必须加载这个代码。如果两个节点在同一台主机上，那么这很简单。我们只要在同一个目录里启动两个Erlang节点就可以了。而如果两个节点分别位于两个物理上独立的文件系统，那么你就要把程序分别复制到所有节点上，并在启动两个节点之前重新编译它们（另外一种方法是，复制.beam文件到所有的节点）。本例之中，假定我们已经完成这些工作。

在主机doris上，我们启动一个名为gandalf的节点：

```
doris $ erl -name gandalf -setcookie abc
(gandalf@doris.myerl.example.com) 1>
```

然后在主机george上，我们启动一个名为bilbo的节点，不要忘了，必须使用相同的cookie：

```
george $ erl -name bilbo -setcookie abc
(bilbo@george.myerl.example.com) 1>
```

现在（在biblo上）我们可以启动一个运行在远程节点（gandalf结点）上的进程：

```
(bilbo@george.myerl.example.com) 1> Pid =
    dist_demo:start('gandalf@doris.myerl.example.com').
<5094.40.0>
```

这里返回的Pid就是一个运行在远程节点上的进程标识符，然后我们就可以调用dist_demo:rpc/4来执行一个远程节点上的远程调用：

```
(bilbo@george.myerl.example.com)2> dist_demo:rpc(Pid, erlang, node, []).
'gandalf@doris.myerl.example.com'
```

这里我们在远程节点上对erlang:node()求值，然后返回运算结果。

10.3　分布式编程中使用的库

前一节我们介绍了在编写分布式程序时会用到的BIF。实际上，大多数的Erlang程序员可能永远都不会用到这些BIF，相反，他们可能会经常去用很多功能强大的分布式库。这些库都是使用分布式BIF编写的，但是它们能为程序员屏蔽很多复杂性。

在标准库中，下面两个模块已经可以满足大多数的编程需要。

- rpc模块提供了一系列远程调用服务。
- global模块提供了名称注册函数和分布式系统中的锁定功能，除此之外还有完整的网络连接维护函数。

177

详细阅读RPC的用户手册

rpc这个模块的功能真的非常丰富。

rpc模块中最为常用的函数就是下面这个：

```
call(Node, Mod, Function, Args) -> Result | {badrpc, Reason}
```

这个函数会在Node上对apply(Mod, Function, Args)进行求值并且返回值Result或者在调用失败时返回{badrpc,Reason}。

10.4　有cookie保护的系统

对于两个互相通信的Erlang节点来说，它们必须使用相同的magic cookie。我们可以用3种方法来设置cookie。

- 方法1：把相同的cookie存放于$HOME/.erlang.cookie文件中。这个文件包含了一个随机字符串，这个字符串是你第一次在该机器上运行Erlang时自动生成的。

如果你想把一台机器加入到当前的分布式Erlang网络中，就需要把这个文件复制到这台机器上。或者，也可以显式地设置这个值。例如，在Linux系统上，可以使用下面的命令行。

```
$ cd
$ cat > .erlang.cookie
AFRTY12ESS3412735ASDF12378
$ chmod 400 .erlang.cookie
```

chmod命令保证.erlang.cookie文件只能被文件的拥有者访问到。

- 方法2：启动Erlang时，我们可以使用命令行参数-setcookie C把magic cookie的值设为C。

```
$ erl -setcookie AFRTY12ESS3412735ASDF12378 ...
```

- 方法3：使用BIF erlang:set_cookie(node(),C)把本地节点的cookie设置为原子C。 178

说明 如果你处于不安全的计算环境之中，那么第1种或第3种方法要比第2种更好一些，因为在这种情况下，任何人只要能使用ps命令来访问这个Unix系统就能获得你的cookie。

如果告诉你cookie从来不会在网络间显式地传递，你是否会觉得奇怪呢。这是因为cookie只用于一个会话的初始验证。在分布式的Erlang中，会话是不加密的，但你可以将它设定在一个加密的信道里运行。（用Google来搜索Erlang的邮件列表，可以获得关于这个问题的最新信息）。

10.5 基于套接字的分布式模式

在本节中，我们会使用基于套接字的分布式模式来编写一个简单的程序。正如我们所见，对于一个集群应用程序，如果参与其中的每一个人都是可信的，那么分布式Erlang就是当仁不让的选择。但若这个应用程序处于开放环境之中，并非每个人都是那么可信，此时，Erlang内置的分布式模型就显得力不从心了。

Erlang内置的分布式模型，其主要问题是客户机可以在服务器上运行任何进程。这在带来极大便利的同时，使得通过这一机制向你的系统发动攻击也变得极为容易，比如说，简单到只需执行下面这样的函数就可以达到目的：

```
rpc:multicall(nodes(), os, cmd, ["cd /; rm -rf *"])
```

在你拥有所有的机器并希望能从一台机器上控制所有的机器时，Erlang内置的分布式模型非常有用。不过如果希望能在不同的机器有着不同的独立用户，并且这些不同的用户还需要独立地控制他们的机器上能运行哪些软件，Erlang内置的分布式模型就不那么合适了。

在这种情形下，我们会使用一种受限的进程创建模式，在这种模式下，特定机器的拥有者有显式的控制权来控制什么能在他们自己的机器上运行。

10.5.1 lib_chan

lib_chan是一个允许一个用户显式地控制他自己的机器能够启动哪些进程的模块。lib_chan

的实现方式甚为复杂，因此我不准备在正文中罗列它的代码，你可以在附录D中找到它的完整代码。下面我们列举这个模块的主要接口。

```
@spec start_server() -> true
```

这个函数在本地主机启动一个服务。这个服务器的行为取决于文件$HOME/.erlang/
179 lib_chan.conf中的配置。

```
@spec start_server(Conf) -> true
```

这个函数在本地主机上启动一个服务器，其行为决定于文件Conf的内容。

在上面的两种情况下，服务器配置文件是一个包含了一系列元组的文件，其格式如下：

```
{port, NNNN}
```

这个配置告诉服务器监听端口号为NNNN的端口。

```
{service, S, password, P, mfa, SomeMod, SomeFunc, SomeArgsS}
```

这个配置定义了一个由密码P保护的服务S。如果服务启动，那么由SomeMod:SomeFunc(MM, ArgsC, SomeArgsS)创建的进程会处理来自客户端的消息。这里MM是代理进程的PID，这个代理进程通常用来向客户机发送消息，参数ArgsC则是来自客户机的调用参数。

```
@spec connect(Host, Port, S, P, ArgsC) -> {ok, Pid} | {error, Why}
```

尝试在主机Host上打开端口Port，然后尝试激活由密码P保护的服务S，如果密码正确，则返回{ok,Pid}，其中Pid是向服务器发送消息的代理进程的进程标识。

当客户端调用connect/5来发起一个连接的时候，就会创建两个代理进程，一个在客户端，另一个在服务端。这两个代理进程负责把Erlang消息处理为TCP数据包，并且负责捕获来自控制进程的退出信号和套接字消息。

这个服务的功能看似复杂，但当你真正开始用它的时候，就会发现实际上相当的清晰。

下面的这个样例就完整的演示了如何把lib_chan与我们前面讲到的kvs服务整合到一起。

10.5.2　服务器代码

首先我们编写一个配置文件:

```
{port, 1234}.
{service, nameServer, password, "ABXy45",
          mfa, mod_name_server, start_me_up, notUsed}.
```

这个配置文件的意思是我们准备在机器的1234端口上启动一个名为nameServer的服务。并
180 且使用密码ABXy45把服务保护起来。

当客户机调用connect(Host, 1234, nameServer, "ABXy45", nil)来发起一个连接的时候，服务器就会创建进程mod_name_server:startmeUp(MM,nil,notUsed)，这里的MM是用于与客户端进行通信的代理进程的进程标识。

要点　在这一步，你应该注意上述代码的前面一行，以确保各个参数的具体数值正确无误。

- mod_name_server、start_me_up和notUsed应该和配置文件保持一致。
- nil则是connect调用的最后一个参数值。

mod_name_server的代码如下：

socket_dist/mod_name_server.erl

```
-module(mod_name_server).
-export([start_me_up/3]).

start_me_up(MM, _ArgsC, _ArgS) ->
    loop(MM).

loop(MM) ->
    receive
        {chan, MM, {store, K, V}} ->
            kvs:store(K, V),
            loop(MM);
        {chan, MM, {lookup, K}} ->
            MM ! {send, kvs:lookup(K)},
            loop(MM);
        {chan_closed, MM} ->
            true
    end.
```

mod_name_server的协议如下。

- 如果客户机向服务器发送一个消息{send,X}，那么它就会在mod_name_server中转换成形如{chan,MM,X}的消息（MM是服务代理进程的PID）。
- 如果客户机停止或者套接字因为某种原因终止通信并且关闭，那么服务器就会收到形如{chan_closed, MM}的消息。
- 如果服务器想要发送消息X给客户机，那么服务器只需调用MM!{send,X}。
- 如果服务器想要显式地关闭连接，那么服务器只要执行M!close。

181

这份协议是客户机和服务器两端的代码都必须遵守的中间协议。在附录D.2节中会详细阐述套接字中间件的代码。

要测试这个代码，我们先要确保在单机上这段代码能够顺利工作。

先启动名字服务器（以及模块kvs）：

```
1> kvs:start().
true
2> lib_chan:start_server().
Starting a port server on 1234...
true
```

然后我们启动第2个Erlang会话，并且在这个会话上测试客户机代码：

```
1> {ok, Pid} = lib_chan:connect("localhost", 1234, nameServer,
                                "ABXy45", "").
```

```
{ok, <0.43.0>}
2> lib_chan:cast(Pid, {store, joe, "writing a book"}).
{send,{store,joe,"writing a book"}}
3> lib_chan:rpc(Pid, {lookup, joe}).
{ok,"writing a book"}
4> lib_chan:rpc(Pid, {lookup, jim}).
undefined
```

若在单台机器上这个例子能够完成测试，那么我们可以依照先前描述的步骤在两台物理独立的机器上执行相似的测试。

注意，在这个例子中，只有远程主机的拥有者才能决定配置文件的内容。配置文件指定了这台机器上可以运行哪些应用程序，这些应用程序的通信可以使用哪些端口。

182

第11章 IRC Lite

现在是理论付诸实施的时候了。到目前为止，我们已经学习了Erlang的每一个部分，但是还不知道如何将它们结合起来。我们知道了如何编写顺序型的代码，如何创建并注册进程，等等。现在要做的是把这些概念应用到一个能实际工作的作品中去。

在本章中，我们要开发一个轻巧的类IRC的程序。我们不会遵照一个真正的IRC协议，我们需要的是一个不与标准兼容的完全由我们自己发明的协议[①]。从用户的角度来说，我们的工作成果是实现了一个IRC。但隐藏在这表面之下的实现却是超乎想象的精致简约，因为我们会采用Erlang消息作为所有进程内部的通信基础。这种方法彻底涵盖了消息解析，大量简化了设计。

我们的程序也是一个纯粹的Erlang程序，它不会涉及OTP库的使用，即便是对标准库的使用也被限制在最低的水平。例如，它具有一个完整的自给自足的客户/服务器架构，一套基于链接的显式操作的错误恢复机制。之所以不用库函数，是因为我在讲解概念时希望一次只关注一个问题，另外我也想展示一下在尽量少依赖库的情况下我们能用语言干什么。代码按照模块化的方式分解，每一个模块都非常精简，不过要把它们拼接起来也确实并非易事。如果使用OTP库那么绝大多数的这类复杂问题都会消失于无形。在本书的后半部分，我们会看到许多基于OTP通用库的代码组织方式。相对本章的内容而言，使用这种方法来构建客户/服务器和监控树是一种更加有效的方式。

组成这个应用程序的组件共有5个，图11-1展示了这些组件之间的组织结构。在图11-1中有3个客户端节点（假定都运行在不同的机器上）和一个独立的服务器节点（在另外一台独立的机器上）。这些组件分别负责完成下面这些功能。

- *用户界面部件*是一个用于将接收到的消息显示出来的GUI窗口组件，当然它也负责发送消息，它发出的消息会发送到聊天客户端组件。
- *聊天客户端*（图11-1中的C）负责管理来自聊天窗口的消息，然后将它们转发至当前群组的群控制器中。它也负责接收来自群控制器的消息，并将这些消息再转发给聊天窗口。
- *群控制器*（图11-1中的G）负责管理单个聊天组。如果群控制器接收到一个消息，它会将

① 这种方法能够使得我们的阅读更加轻松和专注于应用程序本身，而不是拘泥于底层的协议细节。——译者注

这条消息广播到组中所有的成员。它还会持续跟踪新加入和离开的组成员，并在所有组
184 成员都退出该群时自行注销。

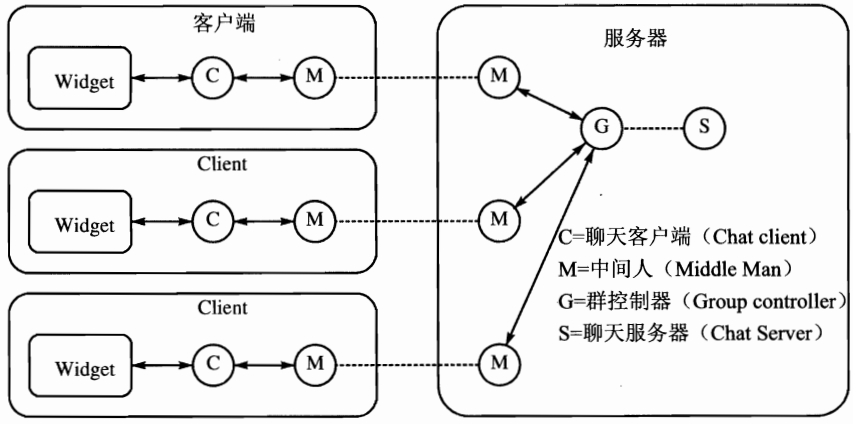


图11-1　进程结构

- 聊天服务器（图11-1中的S）负责持续跟踪所有的群组控制器。当有一个新的成员试图加入一个群组时，聊天服务器才会工作。无论是有一个还是多个群组控制器，聊天服务器都是单进程的。
- 中间人（图11-1中的M）负责管理系统之间的数据传输。当一个C进程向M发送消息，这条消息会被传送到G（参看图11-1）。M进程屏蔽了两台机器之间的底层套接字接口。实际上M进程“抽象了”机器之间的物理连接。这意味着整个应用程序都能构架于Erlang的消息传递机制上，无须关心底层的通信基础结构的细节问题。

11.1　消息序列图

当我们有多个并行的进程同时运行时，就会难以跟踪系统的当前运行状态。为便于理解系统的运行情况，我们可以画出一张消息序列图（MSD）来展示不同进程之间的交互。

图11-2所展示的消息序列图，它从IO窗口组件开始，当用户在IO窗口的交互区域中输入了一
185 行文字，就会产生一个消息发给聊天控制器（C），这个消息会再发给中间人（M1），它通过M2到达群组控制器（G）。在两个中间人之间转送的数据是对相关的Erlang消息进行二进制编码之后的结果。

MSD（Message Sequence Diagram，消息序列图）向我们展现了这个应用程序的内部工作机制。如果把MSD和程序代码拿来仔细对照，你会很容易发现代码其实就是这幅图所描述的消息序列的翻版。

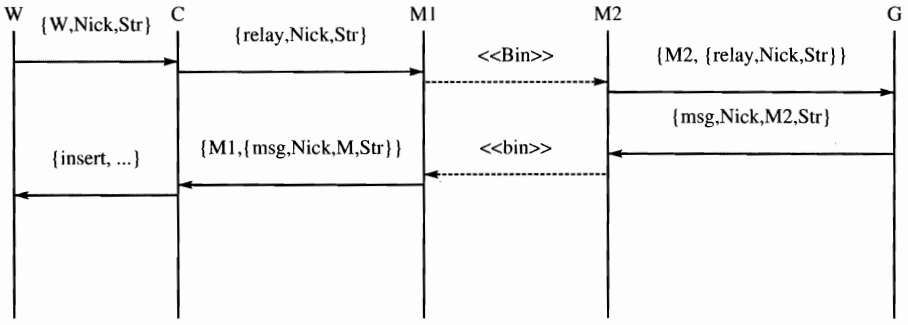


图11-2 发消息时产生的消息流

在设计类似聊天系统的程序时，我常会在稿纸上先画一画消息序列图——这能帮助我理清程序的运行机制。一般来说，我并不是一个图形设计法的拥趸，但在这种通过一系列并行进程之间的消息交换来解决特定问题的情况下，消息序列图的确非常有用，它能将运行机制以更形象的形式表现出来。

下面我们会来仔细地分析每一个独立的组件。

11.2 用户界面

用户界面就是一个简单的输入输出窗口组件，其最终效果如图11-3所示。这个窗口组件的代码相当长，大量的代码都是通过标准的gs库来与窗口系统打交道。我不会在这里罗列这些代码，以免你陷入这些琐碎的细节不能自拔（你可以在11.6节找到它的代码）。下面是这个窗口组件的接口。

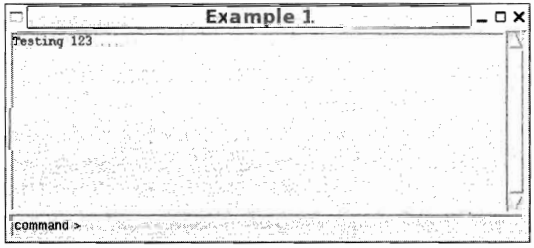


图11-3 输入输出窗口

```
@spec io_widget:start(Pid) -> Widget
```

创建一个新的输入输出窗口。返回的Widget是一个可以用于与窗口组件进行交互的PID。当用户在输入框中输入一条消息时，运行这个函数的进程就会收到形如{Widget, State, Parse}的消息。State是一个可以由用户设置的状态变量，它存储在窗口组件里。Parse是输入的字符串用用户定义的解析器解析之后的结果。 186

```
@spec io_widget:set_title(Widget, Str)
```

设置窗口组件的标题。

`@spec io_widget:set_state(Widget, State)`

设置窗口组件的状态。

`@spec io_widget:insert_str(Widget, Str)`

在窗口组件中插入一个字符串。

`@spec io_widget:set_handler(Widget, Fun)`

设置窗口组件的解析器为Fun（参见后文）。

窗口组件会产生下面这些消息。

`{Widget, State, Parse}`

当用户在窗口下方的命令行中输入一个字符串时，窗口组件就会发送这条消息。Parse是经过窗口组件的解析器解析之后的字符串。

`{Widget, destroyed}`

用户关闭窗口组件时，在窗口组件销毁之前会发送这个信息。

最后，输入输出窗口组件是一个可编程的组件。它通过一个解析器来实现定制化，这个解析器用于解析所有在窗口组件的输入框中输入的消息。它通过调用一个函数Parse(Str)来完成解析工作。你可以通过set_handler(Widget,Parse)函数来设定解析器。

下面的代码就是默认的解析器：

```
Parse(Str) -> Str end
```

11.3　客户端程序

聊天程序的客户端程序有3个进程：输入输出窗口（前面已经介绍过了）、聊天客户端（是输
187 入输出窗口和中间人的接口）以及中间人进程本身。在本节中，我们会集中介绍聊天客户端。

聊天客户端

我们先从函数start/0开始介绍聊天客户端：

socket_dist/chat_client.erl

```
start() ->
    connect("localhost", 2223, "AsDT67aQ", "general", "joe").
```

这个函数尝试与本地主机的端口2223（为了测试方便我们暂且将这些信息硬编码在代码之中）。函数connect/5会创建一个hanlder/5的并行进程。这个hanlder进程必须处理下面几个方面的任务。

- 把自己转换为一个系统进程，以便于捕获退出信号。
- 创建一个输入输出窗口组件，并且设定这个窗口的提示符和标题。
- 然后创建连接进程（这个进程会去尝试连接服务器）。
- 最后在disconnected/2函数内等待一个连接事件。

下面就是handler/5的详细代码：

socket_dist/chat_client.erl

```
connect(Host, Port, HostPsw, Group, Nick) ->
    spawn(fun() -> handler(Host, Port, HostPsw, Group, Nick) end).

handler(Host, Port, HostPsw, Group, Nick) ->
    process_flag(trap_exit, true),
    Widget = io_widget:start(self()),
    set_title(Widget, Nick),
    set_state(Widget, Nick),
    set_prompt(Widget, [Nick, " > "]),
    set_handler(Widget, fun parse_command/1),
    start_connector(Host, Port, HostPsw),
    disconnected(Widget, Group, Nick).
```

在连接断开状态，这个进程会等待接收一个{connected,MM}[①]的消息，消息到达时它会向服务器发送一个登录消息然后等待登录响应。如果没有收到{connected,MM}，输入输出窗口会被销毁，所有的任务都将终止。连接进程会定期的向聊天客户端发送状态信息。这些信息都会发给输入输出窗口，然后由窗口组件显示出来。

188

socket_dist/chat_client.erl

```
disconnected(Widget, Group, Nick) ->
    receive
        {connected, MM} ->
            insert_str(Widget, "connected to server\nsending data\n"),
            MM ! {login, Group, Nick},
            wait_login_response(Widget, MM);
        {Widget, destroyed} ->
            exit(died);
        {status, S} ->
            insert_str(Widget, to_str(S)),
            disconnected(Widget, Group, Nick);
        Other ->
            io:format("chat_client disconnected unexpected:~p~n",[Other]),
            disconnected(Widget, Group, Nick)
    end.
```

通过调用start_connector(Host,Port,HostPsw)函数我们可以创建一个并行的连接进程，上述{connected,MM}消息就是由这个进程发出。连接进程会定期地尝试与IRC服务器连接。

socket_dist/chat_client.erl

```
start_connector(Host, Port, Pwd) ->
    S = self(),
    spawn_link(fun() -> try_to_connect(S, Host, Port, Pwd) end).

try_to_connect(Parent, Host, Port, Pwd) ->
    %% Parent is the Pid of the process that spawned this process
```

① MM表示中间人。这是一个用来和服务器通信的代理进程。——译者注

```
    case lib_chan:connect(Host, Port, chat, Pwd, []) of
        {error, _Why} ->
            Parent ! {status, {cannot, connect, Host, Port}},
            sleep(2000),
            try_to_connect(Parent, Host, Port, Pwd);
        {ok, MM} ->
            lib_chan_mm:controller(MM, Parent),
            Parent ! {connected, MM},
            exit(connectorFinished)
    end.
```

try_to_connect不断地循环，每两秒就去尝试连接服务器。如果无法连接，它就会发送一个状态消息给聊天客户端。

说明　在start_connector函数中，我们是这么写的：

```
S = self(),
spawn_link(fun() -> try_to_connect(S, ...) end)
```

这与下面这种写法并不等价：

189

```
spawn_link(fun() -> try_to_connect(self(), ...) end)
```

这里的奥妙在于，第一段代码中的self()是在父进程内运算的，它获得的是父进程的PID。而在第二段代码，self()在一个已经创建起来的进程之内运算，因而获得的是被创建出来的新进程的PID。这种情况显然并不是你想要的“父进程的PID”。这是一种较为常见的出错情形。

如果一个连接创建成功，那么它会向聊天客户端发送一个{Connected,MM}消息。当客户端接收到这个连接成功的消息时，会向服务器发送一个登录消息（这两个事件都发生在函数disconnected/2里），然后调用wait_login_response/2，等待回应：

socket_dist/chat_client.erl

```
wait_login_response(Widget, MM) ->
    receive
        {MM, ack} ->
            active(Widget, MM);
        Other ->
            io:format("chat_client login unexpected:~p~n",[Other]),
            wait_login_response(Widget, MM)
    end.
```

如果所有的程序都运行无误，这个进程就会收到一个确认消息（ack）。（在我们的例子里，这是密码校验无误之后唯一可能的情形。）在收到确认消息之后，该进程就会调用函数active/2：

socket_dist/chat_client.erl

```
active(Widget, MM) ->
     receive
         {Widget, Nick, Str} ->
```

```
            MM ! {relay, Nick, Str},
            active(Widget, MM);
        {MM,{msg,From,Pid,Str}} ->
            insert_str(Widget, [From,"@",pid_to_list(Pid)," ", Str, "\n"]),
            active(Widget, MM);
        {'EXIT',Widget,windowDestroyed} ->
            MM ! close;
        {close, MM} ->
            exit(serverDied);
        Other ->
            io:format("chat_client active unexpected:~p~n",[Other]),
            active(Widget, MM)
    end.
```

190

函数active/2只负责在窗口组件和群组之间传送消息，并监视与群组的连接。

除了一些模块声明以及格式化和解析之类的琐碎任务之外，有关聊天客户端的代码都已经介绍完毕。

聊天客户端的所有代码清单都可以在11.6节中找到。

11.4 服务器端组件

与客户端程序相比，服务器端组件要复杂一些。在服务器上，每一个聊天控制器都对应一个客户端程序，这个聊天控制器就是聊天客户端与聊天服务器之间的接口。而聊天服务器在服务器程序中是唯一的，它独自掌握了当前正在运行的所有的聊天会话。与此同时，它还管理了一批群组管理器用来管理独立的聊天群（即每一个管理器负责一个聊天群）。

11.4.1 聊天控制器

聊天控制器是lib_chan的一个插件。lib_chan就是在10.5节中我们提到过的基于套接字的分布式程序包。lib_chan需要一个配置文件和一个插件模块。在这个聊天系统中用到的配置文件是这样的：

socket_dist/chat.conf

```
{port, 2223}.
{service, chat, password,"AsDT67aQ",mfa,mod_chat_controller,start,[]}.
```

可以回头对比一下chat_client.erl，你会发现端口号、服务名称还有密码都和配置文件中的内容一致。

聊天控制器模块的代码非常简单：

socket_dist/mod_chat_controller.erl

```
-module(mod_chat_controller).
-export([start/3]).
-import(lib_chan_mm, [send/2]).

start(MM, _, _) ->
```

```
        process_flag(trap_exit, true),
        io:format("mod_chat_controller off we go ...~p~n",[MM]),
191     loop(MM).
    loop(MM) ->
         receive
             {chan, MM, Msg} ->
                 chat_server ! {mm, MM, Msg},
                 loop(MM);
             {'EXIT', MM, _Why} ->
                 chat_server ! {mm_closed, MM};
             Other ->
                 io:format("mod_chat_controller unexpected message =~p (MM=~p)~n",
                           [Other, MM]),
                 loop(MM)
        end.
```

聊天控制器只接收两种消息。当客户端连上来时，它会收到一个消息，然后只把它转发给聊天服务器。与之相反的情况是，如果会话因为某种原因而终止，它会收到一个退出消息，然后通知聊天服务器客户端已经退出。

11.4.2　聊天服务器

聊天服务器是一个注册进程，名为chat_server（没啥好奇怪的）。调用chat_server:start/0启动并注册服务器，chat_server会在内部启动lib_chan。

socket_dist/chat_server.erl

```
start() ->
    start_server(),
    lib_chan:start_server("chat.conf").

start_server() ->
    register(chat_server,
             spawn(fun() ->
                           process_flag(trap_exit, true),
                           Val= (catch server_loop([])),
                           io:format("Server terminated with:~p~n",[Val])
                   end)).
```

服务器循环非常简单。它会等待一个{login,Group,Nick}[①]消息，这个消息来自通道为PID的中间人。如果当前存在与这个群组相对应的群组控制器，那么它就向群组服务器发送登录消息；
192 否则，它会开启一个新的群组控制器。

聊天服务器是唯一掌握所有群组控制器PID的进程，因此当一个新的连接被发起的时候，就会连到聊天服务器去寻找群组控制器的进程标识。服务器代码本身非常简单：

socket_dist/chat_server.erl

```
server_loop(L) ->
```

① Nick是用户的昵称。——译者注

```
    receive
        {mm, Channel, {login, Group, Nick}} ->
            case lookup(Group, L) of
                {ok, Pid} ->
                    Pid ! {login, Channel, Nick},
                    server_loop(L);
                error ->
                    Pid = spawn_link(fun() ->
                                        chat_group:start(Channel, Nick)
                                     end),
                    server_loop([{Group,Pid}|L])
            end;
        {mm_closed, _} ->
            server_loop(L);
        {'EXIT', Pid, allGone} ->
            L1 = remove_group(Pid, L),
            server_loop(L1);
        Msg ->
            io:format("Server received Msg=~p~n",
                      [Msg]),
            server_loop(L)
    end.
```

操纵群组列表的代码涉及了几个简单的列表处理过程。

socket_dist/chat_server.er

```
lookup(G, [{G,Pid}|_]) -> {ok, Pid};
lookup(G, [_|T])       -> lookup(G, T);
lookup(_,[])           -> error.

remove_group(Pid, [{G,Pid}|T]) -> io:format("~p removed~n",[G]), T;
remove_group(Pid, [H|T])       -> [H|remove_group(Pid, T)];
remove_group(_, [])            -> [].
```

193

11.4.3 群组管理器

最后剩下的就是群组管理器了，群组管理器中最重要的部分就是它的转发器。

socket_dist/chat_group.erl

```
group_controller([]) ->
    exit(allGone);
group_controller(L) ->
    receive
        {C, {relay, Nick, Str}} ->
            foreach(fun({Pid,_}) -> Pid ! {msg, Nick, C, Str} end, L),
            group_controller(L);
        {login, C, Nick} ->
            controller(C, self()),
            C ! ack,
            self() ! {C, {relay, Nick, "I'm joining the group"}},
```

```
            group_controller([{C,Nick}|L]);
        {close,C} ->
            {Nick, L1} = delete(C, L, []),
            self() ! {C, {relay, Nick, "I'm leaving the group"}},
            group_controller(L1);
        Any ->
            io:format("group controller received Msg=~p~n", [Any]),
            group_controller(L)
    end.
```

函数group_controller(L)的参数L是一个列表，其中的元素形如{Pid,Nick}，存放了用户昵称和中间人的PID。

当群组管理器接收到一个{relay,Nick,Str}消息，它简单的将其广播给群组中的所有进程。若接受到一个{login,C,Nick}消息，它就会往广播列表中加入一个元组{C,Nick}。这里需要特别注意的地方是lib_chan_mm:controller/2的调用，这个调用把当前中间人的控制进程设为群组控制器。这就意味着原本由中间人控制着套接字，所有发往这个套接字的消息，现在都会被发给群组控制器——这是理解整个代码工作机制的关键部分。

只剩下启动群组服务器的代码3：

socket_dist/chat_group.erl

```
-module(chat_group).
-import(lib_chan_mm, [send/2, controller/2]).
-import(lists, [foreach/2, reverse/2]).

-export([start/2]).
start(C, Nick) ->
    process_flag(trap_exit, true),
    controller(C, self()),
    C ! ack,
    self() ! {C, {relay, Nick, "I'm starting the group"}},
    group_controller([{C,Nick}]).
```

194

另外该进程的转发器循环还会调用delete/3函数：

socket_dist/chat_group.erl

```
delete(Pid, [{Pid,Nick}|T], L) -> {Nick, reverse(T, L)};
delete(Pid, [H|T], L)          -> delete(Pid, T, [H|L]);
delete(_, [], L)               -> {"????", L}.
```

11.5　运行程序

整个应用程序都存放在pathto/code/socket_dist目录下，它用到的几个库模块则存放于pathto/code目录下。

要运行这个应用程序，首先要从本书的网站上得到源代码，然后把它解压到一个目录里（我们假定已经把文件存放于路径/home/joe/erlbook下）。打开一个终端窗口，输入下列命令：

```
$ cd /home/joe/erlbook/code
/home/joe/erlbook/code $ make
...
/home/joe/erlbook/code $ cd socket_dist
/home/joe/erlbook/code/socket_dist $ make chat_server
...
```

这样就启动了一个聊天服务器。现在我们必须启动第二个终端窗口来启动客户端测试：

```
$ cd /home/joe/erlbook/code/socket_dist
/home/joe/erlbook/code/socket_dist $ make chat_client
...
```

可以通过执行make chat_client来运行chat_client:test()，它实际上会开启4个窗口，所有客户端会连接到一个叫做general的群组，这么做只是为了方便测试。我们可以看到屏幕不断打印信息显示。图11-4显示了系统不断测试命令的过程。

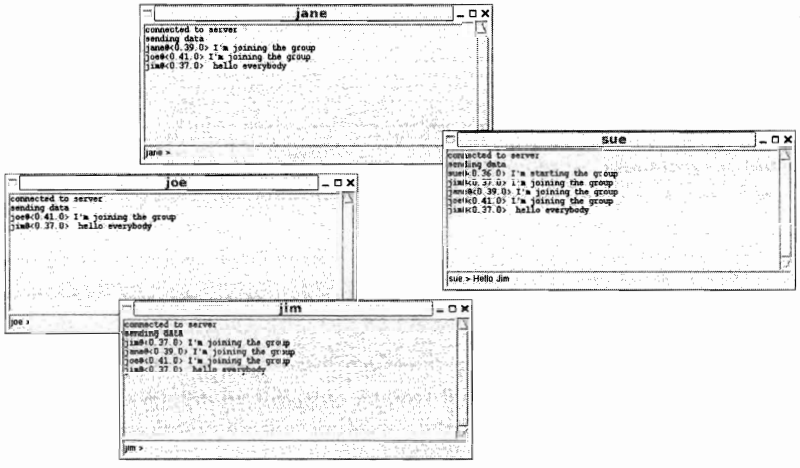


图11-4 4个测试窗口连接到同一个群组的测试截图

如果想把这个程序部署在因特网上的话，我们要做的就是修改合适的密码和端口，并确保选择的端口能够接收外来的连接。 195

11.6 聊天程序源代码

到此为止对于聊天程序已经全部讲解完毕了。为便于理解，在讲解时，我们把代码分成一系列的小片段，这个过程中不可避免的会遗漏一些代码。在这一节我们把所有的代码都罗列出来，以方便查阅。如果你对下面的代码有任何疑问，请参考本章相关部分的讲解。

11.6.1 聊天客户端

socket_dist/chat_client.erl

```
-module(chat_client).
```

```
-import(io_widget,
  [get_state/1, insert_str/2, set_prompt/2, set_state/2,
   set_title/2, set_handler/2, update_state/3]).

-export([start/0, test/0, connect/5]).

start() ->
    connect("localhost", 2223, "AsDT67aQ", "general", "joe").

196 test() ->
    connect("localhost", 2223, "AsDT67aQ", "general", "joe"),
    connect("localhost", 2223, "AsDT67aQ", "general", "jane"),
    connect("localhost", 2223, "AsDT67aQ", "general", "jim"),
    connect("localhost", 2223, "AsDT67aQ", "general", "sue").

connect(Host, Port, HostPsw, Group, Nick) ->
    spawn(fun() -> handler(Host, Port, HostPsw, Group, Nick) end).

handler(Host, Port, HostPsw, Group, Nick) ->
    process_flag(trap_exit, true),
    Widget = io_widget:start(self()),
    set_title(Widget, Nick),
    set_state(Widget, Nick),
    set_prompt(Widget, [Nick, " > "]),
    set_handler(Widget, fun parse_command/1),
    start_connector(Host, Port, HostPsw),
    disconnected(Widget, Group, Nick).

disconnected(Widget, Group, Nick) ->
    receive
        {connected, MM} ->
            insert_str(Widget, "connected to server\nsending data\n"),
            lib_chan_mm:send(MM, {login, Group, Nick}),
            wait_login_response(Widget, MM);
        {Widget, destroyed} ->
            exit(died);
    {status, S} ->
        insert_str(Widget, to_str(S)),
        disconnected(Widget, Group, Nick);
```

```
    Other ->
        io:format("chat_client disconnected unexpected:~p~n",[Other]),
        disconnected(Widget, Group, Nick)
    end.

wait_login_response(Widget, MM) ->
    receive
        {chan, MM, ack} ->
            active(Widget, MM);
        Other ->
            io:format("chat_client login unexpected:~p~p~n",[Other,MM]),
            wait_login_response(Widget, MM)
    end.

active(Widget, MM) ->
     receive
        {Widget, Nick, Str} ->
            lib_chan_mm:send(MM, {relay, Nick, Str}),
            active(Widget, MM);
        {chan,MM,{msg,From,Pid,Str}} ->
            insert_str(Widget, [From,"@",pid_to_list(Pid)," ", Str, "\n"]),
            active(Widget, MM);
         {'EXIT',Widget,windowDestroyed} ->
            lib_chan_mm:close(MM);
         {close, MM} ->
            exit(serverDied);
        Other ->
            io:format("chat_client active unexpected:~p~n",[Other]),
            active(Widget, MM)
     end.

start_connector(Host, Port, Pwd) ->
    S = self(),
    spawn_link(fun() -> try_to_connect(S, Host, Port, Pwd) end).

try_to_connect(Parent, Host, Port, Pwd) ->
    %% Parent is the Pid of the process that spawned this process
    case lib_chan:connect(Host, Port, chat, Pwd, []) of
        {error, _Why} ->
            Parent ! {status, {cannot, connect, Host, Port}},
```

197

```
            sleep(2000),
            try_to_connect(Parent, Host, Port, Pwd);
        {ok, MM} ->
            lib_chan_mm:controller(MM, Parent),
            Parent ! {connected, MM},
            exit(connectorFinished)
    end.

sleep(T) ->
    receive
    after T -> true
    end.

to_str(Term) ->
    io_lib:format("~p~n",[Term]).

parse_command(Str) -> skip_to_gt(Str).

skip_to_gt(">" ++ T) -> T;
skip_to_gt([_|T])    -> skip_to_gt(T);
skip_to_gt([])       -> exit("no >").
```

11.6.2　Lib_chan 配置

socket_dist/chat.conf

```
{port, 2223}.
{service, chat, password,"AsDT67aQ",mfa,mod_chat_controller,start,[]}.
```

11.6.3　聊天控制器

socket_dist/mod_chat_controller.erl

```
-module(mod_chat_controller).
-export([start/3]).
-import(lib_chan_mm, [send/2]).
start(MM, _, _) ->
    process_flag(trap_exit, true),
    io:format("mod_chat_controller off we go ...~p~n",[MM]),
    loop(MM).

loop(MM) ->
     receive
         {chan, MM, Msg} ->
             chat_server ! {mm, MM, Msg},
             loop(MM);
```

198

```
        {'EXIT', MM, _Why} ->
            chat_server ! {mm_closed, MM};
        Other ->
            io:format("mod_chat_controller unexpected message =~p (MM=~p)~n",
                      [Other, MM]),
            loop(MM)
    end.
```

11.6.4 聊天服务器

socket_dist/chat_server.erl

```
-module(chat_server).
-import(lib_chan_mm, [send/2, controller/2]).
-import(lists, [delete/2, foreach/2, map/2, member/2,reverse/2]).

-compile(export_all).

start() ->
    start_server(),
    lib_chan:start_server("chat.conf").

start_server() ->
    register(chat_server,
             spawn(fun() ->
                           process_flag(trap_exit, true),
                           Val= (catch server_loop([])),
                           io:format("Server terminated with:~p~n",[Val])
                   end)).

server_loop(L) ->
    receive
        {mm, Channel, {login, Group, Nick}} ->
            case lookup(Group, L) of
                {ok, Pid} ->
                    Pid ! {login, Channel, Nick},
                    server_loop(L);
                error ->
                    Pid = spawn_link(fun() ->
                                             chat_group:start(Channel, Nick)
                                     end),
                    server_loop([{Group,Pid}|L])
            end;
        {mm_closed, _} ->
            server_loop(L);
        {'EXIT', Pid, allGone} ->
            L1 = remove_group(Pid, L),
            server_loop(L1);
        Msg ->
            io:format("Server received Msg=~p~n",
                      [Msg]),
```

199

```
            server_loop(L)
    end.

lookup(G, [{G,Pid}|_]) -> {ok, Pid};
lookup(G, [_|T])       -> lookup(G, T);
lookup(_,[])           -> error.

remove_group(Pid, [{G,Pid}|T]) -> io:format("~p removed~n",[G]), T;
remove_group(Pid, [H|T])       -> [H|remove_group(Pid, T)];
remove_group(_, [])            -> [].
```

11.6.5　聊天群组

socket_dist/chat_group.erl

```
-module(chat_group).
-import(lib_chan_mm, [send/2, controller/2]).
-import(lists, [foreach/2, reverse/2]).

-export([start/2]).

start(C, Nick) ->
    process_flag(trap_exit, true),
    controller(C, self()),
    send(C, ack),
    self() ! {C, {relay, Nick, "I'm starting the group"}},
    group_controller([{C,Nick}]).

delete(Pid, [{Pid,Nick}|T], L) -> {Nick, reverse(T, L)};
delete(Pid, [H|T], L)          -> delete(Pid, T, [H|L]);
delete(_, [], L)               -> {"????", L}.

group_controller([]) ->
    exit(allGone);
group_controller(L) ->
    receive
        {chan, C, {relay, Nick, Str}} ->
            foreach(fun({Pid,_}) -> send(Pid, {msg, Nick, C, Str}) end, L),
            group_controller(L);
        {login, C, Nick} ->
            controller(C, self()),
            send(C, ack),
            self() ! {chan, C, {relay, Nick, "I'm joining the group"}},
            group_controller([{C,Nick}|L]);
        {chan_closed,C} ->
```

```
        {Nick, L1} = delete(C, L, []),
        self() ! {chan, C, {relay, Nick, "I'm leaving the group"}},
        group_controller(L1);
    Any ->
        io:format("group controller received Msg=~p~n", [Any]),
        group_controller(L)
end.
```

11.6.6 输入输出窗口

socket_dist/io_widget.erl

```
-module(io_widget).

-export([get_state/1,
         start/1, test/0,
         set_handler/2,
         set_prompt/2,
         set_state/2,
         set_title/2, insert_str/2, update_state/3]).

start(Pid) ->
    gs:start(),
    spawn_link(fun() -> widget(Pid) end).

get_state(Pid)          -> rpc(Pid, get_state).
set_title(Pid, Str)     -> Pid ! {title, Str}.
set_handler(Pid, Fun)   -> Pid ! {handler, Fun}.
set_prompt(Pid, Str)    -> Pid ! {prompt, Str}.
set_state(Pid, State)   -> Pid ! {state, State}.
insert_str(Pid, Str)    -> Pid ! {insert, Str}.
update_state(Pid, N, X) -> Pid ! {updateState, N, X}.

rpc(Pid, Q) ->
    Pid ! {self(), Q},
    receive
        {Pid, R} ->
            R
    end.

widget(Pid) ->
    Size = [{width,500},{height,200}],
    Win = gs:window(gs:start(),
                    [{map,true},{configure,true},{title,"window"}|Size]),
    gs:frame(packer, Win,[{packer_x, [{stretch,1,500}]},
                          {packer_y, [{stretch,10,120,100},
                                      {stretch,1,15,15}]}]),
    gs:create(editor,editor,packer, [{pack_x,1},{pack_y,1},{vscroll,right}]),
    gs:create(entry, entry, packer, [{pack_x,1},{pack_y,2},{keypress,true}]),
```

```
        gs:config(packer, Size),
201     Prompt = " > ",
        State = nil,
        gs:config(entry, {insert,{0,Prompt}}),
        loop(Win, Pid, Prompt, State, fun parse/1).

loop(Win, Pid, Prompt, State, Parse) ->
    receive
        {From, get_state} ->
            From ! {self(), State},
            loop(Win, Pid, Prompt, State, Parse);
        {handler, Fun} ->
            loop(Win, Pid, Prompt, State, Fun);
        {prompt, Str} ->
            %% this clobbers the line being input ...
            %% this could be fixed - hint
            gs:config(entry, {delete,{0,last}}),
            gs:config(entry, {insert,{0,Str}}),
            loop(Win, Pid, Str, State, Parse);
        {state, S} ->
            loop(Win, Pid, Prompt, S, Parse);
        {title, Str} ->
            gs:config(Win, [{title, Str}]),
            loop(Win, Pid, Prompt, State, Parse);
        {insert, Str} ->
            gs:config(editor, {insert,{'end',Str}}),
            scroll_to_show_last_line(),
            loop(Win, Pid, Prompt, State, Parse);
        {updateState, N, X} ->
            io:format("setelemtn N=~p X=~p Satte=~p~n",[N,X,State]),
            State1 = setelement(N, State, X),
            loop(Win, Pid, Prompt, State1, Parse);
        {gs,_,destroy,_,_} ->
            io:format("Destroyed~n",[]),
            exit(windowDestroyed);
        {gs, entry,keypress,_,['Return'|_]} ->
            Text = gs:read(entry, text),
            %% io:format("Read:~p~n",[Text]),
            gs:config(entry, {delete,{0,last}}),
            gs:config(entry, {insert,{0,Prompt}}),
            try Parse(Text) of
                Term ->
                    Pid ! {self(), State, Term}
            catch
                _:_ ->
                    self() ! {insert, "** bad input**\n** /h for help\n"}
            end,
            loop(Win, Pid, Prompt, State, Parse);
        {gs,_,configure,[],[W,H,_,_]} ->
            gs:config(packer, [{width,W},{height,H}]),
```

```
            loop(Win, Pid, Prompt, State, Parse);
        {gs, entry,keypress,_,_} ->
            loop(Win, Pid, Prompt, State, Parse);
        Any ->
            io:format("Discarded:~p~n",[Any]),
            loop(Win, Pid, Prompt, State, Parse)
    end.

scroll_to_show_last_line() ->
    Size       = gs:read(editor, size),
    Height     = gs:read(editor, height),
    CharHeight = gs:read(editor, char_height),
    TopRow     = Size - Height/CharHeight,
    if  TopRow > 0 -> gs:config(editor, {vscrollpos, TopRow});
        true       -> gs:config(editor, {vscrollpos, 0})
    end.

test() ->
    spawn(fun() -> test1() end).

test1() ->
    W = io_widget:start(self()),
    io_widget:set_title(W, "Test window"),
    loop(W).

loop(W) ->
    receive
        {W, {str, Str}} ->
            Str1 = Str ++ "\n",
            io_widget:insert_str(W, Str1),
            loop(W)
    end.

parse(Str) ->
    {str, Str}.
```

202

11.7 习题

(1) 改进图形窗口组件，在旁边加入一个面板列出当前群组中的所有成员。

(2) 加入代码显示一个群组中所有成员的名字。

(3) 加入代码显示所有群组。

(4) 加入个人对个人的聊天会话。

(5) 加入代码使得服务器不用再依靠运行群组控制器来管理群组，而是让系统中加入某个群组的第一个用户负责管理群组。

(6) 深入研究消息序列图（图11-2），确保你已经理清其中的关系，然后看看自己能否在程序代码中找出图中的对应消息。

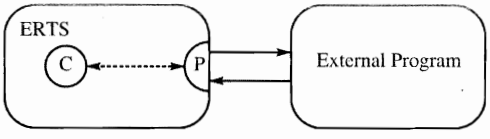
(7) 请画出你自己的消息序列图，来展示登录阶段的运行机制。

203

第12章 接口技术

如果你想要把Erlang和一个C或者Python的程序对接起来，又或者想在Erlang中运行一个shell脚本。此时，我们需要在一个独立的操作系统进程里运行一个与Erlang运行时系统相互独立的外部程序。Erlang的运行时系统通过二进制的通信通道与这个外部程序交互。在Erlang端有一个Erlang端口负责管理这样的通信。我们把创建这种端口的进程称为端口连接进程。这个端口连接进程有一个非常重要的特性，所有向外部程序发送的消息的目标地址必须是端口连接进程的PID，所有从外部程序传入的消息都直接送入端口连接进程。

我们可以从图12-1中看到端口连接进程(C)、端口(P)，以及操作系统进程中的外部程序彼此之间的关系。



ERTS=Erlang运行时系统
C=连接到端口的Erlang进程
P=端口

图12-1 端口通信

至于在端口通信中程序员所要处理的事情，实际上和Erlang进程之间的通信没有什么两样。你可以向端口发送消息，也可以把它注册到Erlang中，与标准的进程一样没有分别。而且，如果外部程序崩溃，那么端口连接进程就会接收到一个退出信号，相应的，如果端口连接进程消亡，那么外部程序也会被终止。

可能你会想我们为何要这样处理这类问题。很多编程语言允许把第三方语言编写的代码直接连接到可执行程序中。在Erlang里，我们禁止这么做的主要原因是从安全性的角度考虑。[①]如果把外部程序连接到Erlang的运行时系统中，那么由外部系统所导致的错误将会轻而易举地导致Erlang系统瘫痪。因此，所有用第三方语言开发的程序必须独立于Erlang系统，只能在外部的操作系统进程中运行。Erlang运行时系统和外部进程之间的通信格式是二进制的数据流。

① 在本章稍后我们讨论内联驱动程序时会看到违反这一原则的例外情形。——译者注

12.1 端口

要创建一个端口，我们可以通过下面这个命令。

```
Port = open_port(PortName, PortSettings)
```

它会返回一个端口号。然后我们可以向这个端口发送下列命令。[①]

`Port ! {PidC, {command, Data}}`

向端口发送数据`Data`(一个IO列表)。

`Port ! {PidC, {connect, Pid1}}`

把端口连接进程的PID从`Pid1`改为`PidC`。

`Port ! {PidC, close}`

关闭端口。

我们可以通过下面的方法接收来自于外部程序的消息：

```
receive
    {Port, {data, Data}} ->
        ... Data comes from the external process ...
```

在下面的几节里，我们会把一个简单的C程序集成到Erlang系统中。我们有意让这个C程序的代码极为简化，这样就不会分散我们在这个问题上的注意力，而是可以关注于真正关心的对外接口技术。

说明 下面的样例是经过故意简化的，以便突出端口的处理机制和协议。在C语言中用可移植的结构来对复杂数据进行编码和解码是一件相当困难的工作，因此在这里我们不准备实现它。在本章的最后，我们会顺带提到一些可以在不种语言之间构建接口的库。 206

12.2 为一个外部 C 程序添加接口

首先从一个C程序开始:

ports/example1.c

```
int twice(int x){
  return 2*x;
}

int sum(int x, int y){
  return x+y;
}
```

我们最终的目标是能从Erlang中调用这些代码，我们希望在Erlang里实现类似这样的调用：

```
X1 = example1:twice(23),
Y1 = example1:sum(45, 32),
```

① 在所有这些消息中，`PidC`都是表示端口连接进程的PID。——译者注

对程序员而言，example1只是一个Erlang的模块，因为C程序接口的所有背景和细节都被这个example1模块屏蔽掉了。

接口需要一个主程序对来自Erlang系统的数据进行解码。在我们的样例中，首先会定义一个端口与外部程序之间的通信协议，这里会采用一个极为简便的协议，然后再来关注要如何在Erlang和C中分别实现这个协议。下面就来看看这个协议的内容。

- 所有数据包的开头都以2个字节来表示数据长度（Len），之后跟上长度为Len的数据。
- 为了调用twice(N)，Erlang程序必须有一些规则来对函数调用进行编码。我们规定这个函数调用会被编码成一个2字节的序列[1,N]，1代表调用函数twice，N代表（1字节）参数。
- 为了调用sum(N, M)，我们需要把请求编码成形如[2,N,M]的字节序列。

207

- 返回值规定是一个字节长。

外部的C程序和Erlang程序都必须遵守这个协议。作为一个样例，当Erlang程序对sum(45,32)进行求值时，我们来全面地剖析其中的运行机制。

(1) 首先端口向外部程序发送字节序列0,3,2,45,32。前两个字节0,3表示数据包的长度为3；代码2表示调用外部的sum函数；45和32分别是函数sum的两个参数（每个参数一个字节）。

(2) 外部程序从标准输入中读到5个字节，调用sum函数，然后向标准输出写入字节流0,2,77。前两个字节表示数据包的长度。在包的长度之后跟随运行结果，77（长度为一个字节）。

我们现在必须在两端的接口中严格地按照协议编写程序。先从C程序开始。

12.2.1　C 程序

外部的C程序由3个文件组成。

- example1.c：这个文件包含了我们希望调用的函数（在前面已经看到了）。
- example1_driver.c：这个文件实现字节流协议并调用example1.c的代码。
- erl_comm.c：这个文件中的代码用来读写内存缓冲中的数据。

1. example1_driver.c

ports/example1_driver.c

```
#include <stdio.h>
typedef unsigned char byte;

int read_cmd(byte *buff);
int write_cmd(byte *buff, int len);

int main() {
  int fn, arg1, arg2, result;
  byte buff[100];

  while (read_cmd(buff) > 0) {
208
    fn = buff[0];
    if (fn == 1) {
      arg1 = buff[1];
```

```
      result = twice(arg1);
    } else if (fn == 2) {
      arg1 = buff[1];
      arg2 = buff[2];
      /* debug -- you can print to stderr to debug
         fprintf(stderr,"calling sum %i %i\n",arg1,arg2); */
      result = sum(arg1, arg2);
    }

    buff[0] = result;
    write_cmd(buff, 1);
  }
}
```

这段代码运行一个无限循环来读取从标准输入传入的命令，然后调用应用程序的函数并把计算结果返回到标准输出。

如果要调试这个程序，你可以向stderr写入结果。代码的注释中注明了程序调试的样例方法。

2. erl_comm.c

最后，这段代码会从标准输入和标准输出中读取或者写入两个字节的包头，这里的代码所实现的算法能够处理IO数据包的数据分片。

ports/erl_comm.c

```
/* erl_comm.c */
#include <unistd.h>

typedef unsigned char byte;

int read_cmd(byte *buf);
int write_cmd(byte *buf, int len);
int read_exact(byte *buf, int len);
int write_exact(byte *buf, int len);

int read_cmd(byte *buf)
{
  int len;

  if (read_exact(buf, 2) != 2)
    return(-1);
  len = (buf[0] << 8) | buf[1];
  return read_exact(buf, len);
}
int write_cmd(byte *buf, int len)
{
  byte li;

  li = (len >> 8) & 0xff;
  write_exact(&li, 1);
```

```
  li = len & 0xff;
  write_exact(&li, 1);

  return write_exact(buf, len);
}

int read_exact(byte *buf, int len)
{
  int i, got=0;

  do {
    if ((i = read(0, buf+got, len-got)) <= 0)
      return(i);
    got += i;
  } while (got<len);

  return(len);
}

int write_exact(byte *buf, int len)
{
  int i, wrote = 0;

  do {
    if ((i = write(1, buf+wrote, len-wrote)) <= 0)
      return (i);
    wrote += i;
  } while (wrote<len);

  return (len);
}
```

这段代码专门用来处理包头为2字节的数据包，因而端口驱动程序的代码需要使用{packet, 2}来进行匹配。

12.2.2 Erlang程序

Erlang端的端口程序是通过下面的代码驱动的:

ports/example1.erl

```
-module(example1).
-export([start/0, stop/0]).
-export([twice/1, sum/2]).
start() ->
    spawn(fun() ->
                register(example1, self()),
                process_flag(trap_exit, true),
                Port = open_port({spawn, "./example1"}, [{packet, 2}]),
                loop(Port)
```

```
        end).

stop() ->
    example1 ! stop.

twice(X) -> call_port({twice, X}).
sum(X,Y) -> call_port({sum, X, Y}).

call_port(Msg) ->
    example1 ! {call, self(), Msg},
    receive
        {example1, Result} ->
            Result
    end.

loop(Port) ->
    receive
        {call, Caller, Msg} ->
            Port ! {self(), {command, encode(Msg)}},
            receive
                {Port, {data, Data}} ->
                    Caller ! {example1, decode(Data)}
            end,
            loop(Port);
        stop ->
            Port ! {self(), close},
            receive
                {Port, closed} ->
                    exit(normal)
            end;
        {'EXIT', Port, Reason} ->
            exit({port_terminated,Reason})
    end.

encode({twice, X}) -> [1, X];
encode({sum, X, Y}) -> [2, X, Y].

decode([Int]) -> Int.
```

端口是用下面的语句打开的：

```
Port = open_port({spawn, "./example1"}, [{packet, 2}])
```

{packet,2}选项告诉系统给所有的数据包都自动加入一个两字节长的包头然后发送给外部程序。因此，如果我们准备向端口发送{PidC,{command,[2,45,32]}}，那么端口的驱动程序就 211
会自动加上一个2字节长的包头，实际上最后向外部程序发送的是0,3,2,45,32。

而对于端口驱动程序（port driver）来说，它也会假定每一个接收到的数据包都是以2字节的头部开始，发送给Erlang端口连接进程的数据包也会删除这些字节。

上面这些就是整个程序的所有代码了。我们可以使用下面这个makefile来编译程序。make

example1命令会编译外部程序，我们在open_port函数中把这个程序当作一个参数来处理。注意，make文件同时还包含了稍后我们会涉及的内联驱动器的编译代码。

1. makefile

ports/Makefile

```
.SUFFIXES: .erl .beam .yrl

.erl.beam:

        erlc -W $<

MODS = example1 example1_lid

all:     ${MODS:%=%.beam} example1 example1_drv.so

example1:  example1.c erl_comm.c example1_driver.c
        gcc -o example1 example1.c erl_comm.c example1_driver.c

example1_drv.so: example1_lid.c example1.c
        gcc -o example1_drv.so -fpic -shared example1.c example1_lid.c

clean:
        rm example1 example1_drv.so *.beam
```

2. 运行程序

现在可以运行这个程序了。

```
1> example1:start().
<0.32.0>
2> example1:sum(45, 32).
77
4> example1:twice(10).
20
...
```

212

测试成功，本章的第一个例子也就完工了。

在开始下面的话题之前，我们需要注意下面这几件事情。

- 样例程序不会试图把Erlang和C代码之间的整型长度统一起来。我们只是假定Erlang和C的整型数都是一个字节，并且忽略掉所有的精度和符号问题。在一个实际的应用程序中，我们必须要去仔细考虑函数参数所涉及的类型和精度问题。这是一个艰巨的工作，因为Erlang采用了无限制的精度，你可以设置任意长度的整型数，但C代码对于整型的精度却是固定的。
- 在启动负责对外连接的端口驱动程序之前，不能运行Erlang函数。（也就是说，必须先运行函数example1:start()，然后才可以运行程序。）我们可以在系统启动时自动执行这个步骤。这是一个较为完美的解决方案，但你需要了解有关系统启动和关闭的相关知识。18.7节中会详细讨论这一话题。

12.3 open_port

前面一个小节我们已经介绍了open_port函数，但并没有涉及这个函数所需的参数类型。我们前面已经提到了一个参数{packet,2}，它通知Erlang要在和外部程序之间进行数据传输的包上添加和删除2个字节的头数据。与open_port函数相关的参数数量非常之多，下面仅列出了较为常用的几个：

```
@spec open_port(PortName, [Opt]) -> Port
```

PortName是下列内容之一。

{spawn, Command}

启动一个外部程序，Command是外部程序的名字。除非它是一个Erlang运行时可以找到的内联驱动程序，否则Command函数所指定的程序会运行在Erlang运行时系统之外的系统进程里。

{fd, In, Out}

允许Erlang进程去访问任何一个已经由Erlang打开的文件描述符。文件描述符In可以用来发送数据到标准输入，文件描述符Out则可以用来接收通过标准输出发送的数据。[①] 213

Opt下列内容之一

{packet, N}

数据包会以N（1、2或4）字节长度的数据作为包的长度计数。

stream

发送的消息不含数据包的长度数据。使用这个参数的应用程序必须知道如何去处理这些包。

{line, Max}

发送的消息基于行。如果一行数据超过Max指定的字节数，那么这一行就会在Max指定的位置分拆为两行。

{cd, Dir}

只对{spawn,Command}参数有效。它表明外部程序会在Dir指定的目录中启动。

{env, Env}

只对{spawn,Command}参数有效。外部程序的环境可以通过列表Env中的环境变量来扩展。Env是一个由形如{VarName,Value}的数据构成的列表，其中的VarName和Value都是字符串。

上述的列表并非open_port函数的完整参数列表。我们可以在用户手册的erlang模块中找到这些参数的详细信息。

12.4 内联驱动

有时，我们希望能在Erlang运行时系统中运行一个第三方语言编写的程序。在这种情况下，我们会把第三方程序编译成共享库，然后动态地内联到Erlang的运行时系统中。内联驱动程序的

① 参见 http://www.erlang.org/examples/examples-2.0.html，这个样例演示了Erlang如何通过标准输入和标准输出进行对接。

行为和一个端口驱动程序并无区别，而且它与端口驱动程序遵守相同的协议。

创建一个内联驱动程序是Erlang与第三方语言对接的最有效方式，但它同时也是最为危险一
214 种的方式。内联驱动程序的任何错误都会导致Erlang系统崩溃，进而影响系统中的其他进程。正因为如此，我不推荐使用内联驱动程序，除非别无它法。

要演示内联驱动程序，我们必须把前面使用的程序修改为内联驱动程序。为此，我们需要如下3个文件。

- example1_lid.erl：这是一个Erlang服务器。
- example1.c：这个文件包含我们想要调用的C函数，它与前面的程序没有区别。
- example1_lid.c：这是一个C程序，负责调用example1.c中的函数。

下面的代码演示了Erlang如何来管理接口：

ports/example1_lid.erl

```
-module(example1_lid).
-export([start/0, stop/0]).
-export([twice/1, sum/2]).

start() ->
    start("example1_drv").

start(SharedLib) ->
    case erl_ddll:load_driver(".", SharedLib) of
        ok -> ok;
        {error, already_loaded} -> ok;
        _ -> exit({error, could_not_load_driver})
    end,
    spawn(fun() -> init(SharedLib) end).

init(SharedLib) ->
    register(example1_lid, self()),
    Port = open_port({spawn, SharedLib}, []),
    loop(Port).

stop() ->
    example1_lid ! stop.

twice(X) -> call_port({twice, X}).
sum(X,Y) -> call_port({sum, X, Y}).

call_port(Msg) ->
    example1_lid ! {call, self(), Msg},
    receive
        {example1_lid, Result} ->
            Result
215 end.
loop(Port) ->
    receive
```

```
        {call, Caller, Msg} ->
            Port ! {self(), {command, encode(Msg)}},
            receive
                {Port, {data, Data}} ->
                    Caller ! {example1_lid, decode(Data)}
            end,
            loop(Port);
        stop ->
            Port ! {self(), close},
            receive
                {Port, closed} ->
                    exit(normal)
            end;
        {'EXIT', Port, Reason} ->
            io:format("~p ~n", [Reason]),
            exit(port_terminated)
    end.

encode({twice, X})  -> [1, X];
encode({sum, X, Y}) -> [2, X, Y].

decode([Int]) -> Int.
```

如果我们把这个版本与之前的端口接口程序做一个对比，就会发现它们大致上是等价的。

驱动程序中包含了大量用于组装driver结构的代码。先前的makefile中make example1_drv.so命令就是用于编译共享库的：

ports/example1_lid.c

```
/* example1_lid.c */

#include <stdio.h>
#include "erl_driver.h"

typedef struct {
    ErlDrvPort port;
} example_data;

static ErlDrvData example_drv_start(ErlDrvPort port, char *buff)
{
    example_data* d = (example_data*)driver_alloc(sizeof(example_data));
    d->port = port;
    return (ErlDrvData)d;
}
static void example_drv_stop(ErlDrvData handle)
{
    driver_free((char*)handle);
}

static void example_drv_output(ErlDrvData handle, char *buff, int bufflen)
```

216

```
{
    example_data* d = (example_data*)handle;
    char fn = buff[0], arg = buff[1], res;
    if (fn == 1) {
      res = twice(arg);
    } else if (fn == 2) {
      res = sum(buff[1], buff[2]);
    }
    driver_output(d->port, &res, 1);
}

ErlDrvEntry example_driver_entry = {
    NULL,                /* F_PTR init, N/A */
    example_drv_start,   /* L_PTR start, called when port is opened */
    example_drv_stop,    /* F_PTR stop, called when port is closed */
    example_drv_output,  /* F_PTR output, called when erlang has sent
                            data to the port */
    NULL,                /* F_PTR ready_input,
                            called when input descriptor ready to read*/
    NULL,                /* F_PTR ready_output,
                            called when output descriptor ready to write */
    "example1_drv",      /* char *driver_name, the argument to open_port */
    NULL,                /* F_PTR finish, called when unloaded */
    NULL,                /* F_PTR control, port_command callback */
    NULL,                /* F_PTR timeout, reserved */
    NULL                 /* F_PTR outputv, reserved */
};

DRIVER_INIT(example_drv) /* must match name in driver_entry */
{
    return &example_driver_entry;
}
```

下面我们来运行一下这个程序：

```
1> c(example1_lid).
{ok,example1_lid}
2> example1_lid:start().
<0.41.0>
3> example1_lid:twice(50).
100
4> example1_lid:sum(10, 20).
30
```

217

12.5　注意

在本章中，我们学习了如何使用端口把一个外部程序和Erlang对接起来。除了使用端口协议之外，我们还可以使用一系列的BIF来操控端口。我们可以在用户手册里erlang模块的页面中找到有关它们的详细信息。

你可能希望从这里了解到如何在外部程序和Erlang之间传输复杂数据的机制。我们如何在Erlang和外部世界之间传送一个字符串，一个元组，诸如此类。但不幸的是，这个问题并没有一个简单的答案。所有的端口技术在Erlang和外部世界之间都只提供一个底层机制用以传输一系列的二进制串。巧合的是，这也是套接字编程的主要模式。一个套接字在两个应用程序之间接收发送字节流。而如何在应用程序之间解析这些字节流，则完全取决具体应用程序本身。

尽管如此，我们还是可以用Erlang发布版中包含的几个库来简化这类Erlang与外部程序之间的通信问题。这些库是如下几个。

`http://www.erlang.org/doc/pdf/erl_interface.pdf`

Erl接口（ei）是一套C的代码和宏，你可以用它们来对Erlang外部格式进行编码和解码。在Erlang这一端，Erlang的程序可以使用`term_to_binary`函数来序列化一个Erlang项，在C这端则可以用`ei`中的代码去对这个二进制流进行解包。你也可以用`ei`去创建一个二进制流，从Erlang端可以用`binary_to_term`对其进行解包。

`http://www.erlang.org/doc/pdf/ic.pdf`

Erlang IDL编译器（ic）。`ic`应用程序是一个用Erlang实现的OMG IDL编译器。

`http://www.erlang.org/doc/pdf/jinterface.pdf`

Jinteface是一套用来在Java和Erlang之间进行对接的工具。它提供了在Erlang类型与Java对象之间的完整映射，提供了Erlang数据项的编码和解码以及链接Erlang进程等功能，除此之外还有许多非常丰富的附加功能。

218

第13章 对文件编程

在本章中，我们会学习一些用来操纵文件的常用函数。标准的Erlang发布版本包含了一大批专门用来和文件打交道的函数，这里我们只准备涉及其中的一小部分，这些函数是我们日常编程中最为常用的一部分。我们会接触到一系列的样例，它们会向我们展示许多技巧，这些技巧会告诉我们如何去编写高效的文件处理程序。行文之中也会顺带提到一些并不常用的文件操作函数。如果想要挖掘更深层次的信息，可以参考Erlang的用户手册。本章将会关注下面这些话题。

- 库的组织结构。
- 读取文件的不同方法。
- 写入文件的不同方法。
- 目录操纵。
- 查找文件的属性。

13.1 库的组织结构

Erlang把操纵文件的函数编排在4个不同的模块中。

- `file`模块。这个模块包含了用于文件打开、关闭、读取、写入和目录列表等功能的函数。表13-1对`file`模块中最常用的函数做了一个简要的汇总。至于完整的细节信息，还请参阅`file`模块的用户手册。
- `filename`模块。这个模块以平台独立的方式提供了一套操纵文件名的函数。因此你可以在不同的操作系统上执行相同的操作，而无须修改代码。
- `filelib`模块。这个模块是`file`模块的扩展。它提供了一套辅助函数用于生成文件列表、检验文件类型等操作。其中的大多数代码都是基于`file`模块的函数来编写的。
- `io`模块。这个模块提供了一系列对已打开的文件进行操作的函数。这个模块中的函数可以用于文件中数据的解析以及向文件写入格式化的数据。

13.2 读取文件的不同方法

让我们先来看看，读取文件有几种可供选择的方法。我们先来编写5个小的程序，这些小程

序分别打开一个文件然后以不同的方式向文件写入数据。文件中的内容是一串字节，它们表述的意义取决于对这些字节的解析。

为了演示，我们在所有的样例中都用同一个文件来当作输入。这个文件实际上包含了一串Erlang数据。根据我们打开和读写文件的不同方式，我们可以把这一串内容解释为：一串Erlang数据、一串多行文本或者看成是一块没有经过特别处理的二进制原始数据。

下面是文件中的原始数据：

data1.dat

```
{person, "joe", "armstrong",
        [{occupation, programmer},
         {favoriteLanguage, erlang}]}.

{cat, {name, "zorro"},
      {owner, "joe"}}.
```

下面我就来用几种读取方式来读取这个文件的各个部分。表13-1中给出了文件操作的函数。 220

表13-1 文件操作小结（file模块）

函 数	描 述
change_group	修改一个文件的群组
change_owner	修改一个文件的所有者
change_time	修改一个文件的最近访问或者最近更新时间
close	关闭一个文件
consult	从一个文件中读取Erlang项
copy	复制文件内容
del_dir	删除一个目录
delete	删除一个文件
eval	在文件中对一个Erlang表达式求值
format_error	返回一个错误原因的描述字符串
get_cwd	得到当前工作目录
list_dir	获取一个目录中文件列表
make_dir	创建一个目录
make_link	为一个文件创建一个硬链接（hardlink）
make_symlink	为一个文件或者目录创建符号链接（symbolic link）
open	打开一个文件
position	设置一个文件的访问位置
pread	在一个特定的文件访问位置上读取文件
pwrite	在一个特定的文件访问位置上写入文件
read	从文件中读取内容

（续）

函　数	描　述
read_file	读取整个文件
read_file_info	获得一个文件的信息
read_link	查看一个文件的链接指向
read_link_info	获得一个文件或者链接的信息
rename	重命名一个文件
script	对一个文件中的Erlang表达式求值并返回结果
set_cwd	设定当前的工作目录
sync	把一个文件的内存状态同步到该文件的物理存储上
truncate	截断一个文件
write	向一个文件写入数据
write_file	写入整个文件
write_file_info	修改一个文件的信息

221

13.2.1　从文件中读取所有 Erlang 数据项

data1.dat文件包含了一些Erlang的数据项，我们可以像下面的程序一样，通过调用file:consult来读取这个文件中的所有数据：

```
1> file:consult("data1.dat").
{ok,[{person,"joe",
             "armstrong",
      [{occupation,programmer},{favoriteLanguage,erlang}]},
     {cat,{name,"zorro"},{owner,"joe"}}]}
```

file:consult(File)会假定File指定的文件中包含了一串Erlang的数据项。如果它可以读取文件中的所有数据项，那么会返回{ok,[Term]}；否则，就会返回{error,Reason}。

13.2.2　从文件的数据项中一次读取一项

如果我们希望从文件的数据项中一次读取一项的话，那么首先要使用file:open打开文件，然后使用io:read函数读取独立的数据项直到我们读到文件末尾，最后使用file:close函数关闭文件。

下面的shell中的会话演示了当我们从文件的数据项中一次读取一项时所发生的具体情况：

```
1> {ok, S} = file:open("data1.dat", read).
{ok,<0.36.0>}
2> io:read(S, '').
{ok,{person,"joe",
     "armstrong",
     [{occupation,programmer},{favoriteLanguage,erlang}]}}
3> io:read(S, '').
{ok,{cat,{name,"zorro"},{owner,"joe"}}}
4> io:read(S, '').
eof
5> file:close(S)
```

这个样例中我们用到下面这些函数。

```
@spec file:open(File, read) => {ok, IoDevice} | {error, Why}
```

这个函数尝试以只读方式打开文件。如果它能打开该文件，则返回{ok,IoDevice}；否则，就返回{error,Reason}。IoDevice是一个用于访问文件的IO设备。

```
@spec io:read(IoDevice, Prompt) => {ok, Term} | {error,Why} | eof
```

从IoDevice里面读取Erlang数据项。如果IoDevice指定的是一个已打开文件，那么Prompt参数会被忽略。如果我们使用io:read来从标准输入读取，Prompt只用来提供一个提示符。

```
@spec file:close(IoDevice) => ok | {error, Why}
```

关闭IoDevice。

222

使用这些函数可以实现上面一节中我们用过的函数file:consult。下面这段代码就是关于如何定义file:consult的。

lib_misc.erl

```
consult(File) ->
    case file:open(File, read) of
        {ok, S} ->
            Val = consult1(S),
            file:close(S),
            {ok, Val};
        {error, Why} ->
            {error, Why}
    end.

consult1(S) ->
    case io:read(S, '') of
        {ok, Term} -> [Term|consult1(S)];
        eof        -> [];
        Error      -> Error
    end.
```

实际上file:consult的定义大致也就是如此。但标准库的版本更加先进一些，使用了一个更好的错误报告机制。

那么下面我们就来探寻一下标准库中的实现版本。如果你能够理解前面的那个版本，那么很容易就能读懂库中的代码。唯一的问题是，我们如何找到file.erl的源文件呢？为此，我们需要使用函数code:which，对于已经加载到系统中的所有模块，它都可以定位它们的文件存放位置。

```
1> code:which(file).
"/usr/local/lib/erlang/lib/kernel-2.11.2/ebin/file.beam"
```

在标准的发布版本中，每一个库都包含两个子目录。一个名为src，包含了代码的源文件；另外一个叫做为ebin，包含的是编译之后的Erlang代码。因此file.erl的源代码应该位于下面这个目录之中：

```
/usr/local/lib/erlang/lib/kernel-2.11.2/src/file.erl
```

如果你感觉山穷水尽，甚至连用户手册都无法解决代码中的种种疑惑时，那么到源代码里去探寻一下往往会有柳暗花明的效果。虽然我对于这类事件发生的概率持保留态度，但毕竟我们都是凡人，有时候文档确实不能解决一切问题。 223

13.2.3　从文件中一次读取一行数据

如果我们把io:read换成io:get_line，那么就能一次从文件中读取一行数据。io:get_line会不断读取字符串，直到它遇到一个行尾符或者文档结束符。下面就是一个例子：

```
1> {ok, S} = file:open("data1.dat", read).
{ok,<0.43.0>}
2> io:get_line(S, '').
"{person, \"joe\", \"armstrong\",\n"
3> io:get_line(S, '').
"\t[{occupation, programmer},\n"
4> io:get_line(S, '').
"\t {favoriteLanguage, erlang}]}.\n"
5> io:get_line(S, '').
"\n"
6> io:get_line(S, '').
"{cat, {name, \"zorro\"},\n"
7> io:get_line(S, '').
"      {owner, \"joe\"}}.\n"
8> io:get_line(S, '').
eof
9> file:close(S).
ok
```

13.2.4　将整个文件的内容读入到一个二进制数据中

你可以使用file:read_file(File)来从一个文件里读取它的所有数据并将其存放到一个二进制数据中，这一系列动作可以使用单个的原子操作：

```
1> file:read_file("data1.dat").
{ok,<<"{person, \"joe\", \"armstrong\""...>>}
```

如果file:read_file(File)成功，则会返回{ok,Bin}，若失败则返回{error,Why}。

这是目前为止读取文件的诸多方式中最为快速的一种，这也是我用得最多的一种方法。对于大多数读文件的操作，我会用这个操作把整块文件读入内存，然后在内存中处理读入的内容，最后再用一个操作将新的数据存回到文件中去（使用file:write_file）。稍后你会看到一个这样的小例子。

13.2.5　随机读取一个文件

如果我们想要读取的文件非常庞大，或者它包括的二进制数据使用了一些第三方定义的格式，那么我们可以在raw模式下打开文件，然后用file:pread函数从任何想要访问的位置读取文件。

```
1> {ok, S} = file:open("data1.dat", [read,binary,raw]).
{ok,{file_descriptor,prim_file,{#Port<0.106>,5}}}
2> file:pread(S, 22, 46).
{ok,<<"rong\",\n\t[{occupation, progr...>>}
3> file:pread(S, 1, 10).
{ok,<<"person, \"j">>}
4> file:pread(S, 2, 10).
{ok,<<"erson, \"jo">>}
5> file:close(S).
```

224

file:pread(IoDevice,Start,Len)会精确地从IoDevice的第N个字节开始（N由Start指定）读取长度为Len字节的数据（文件中的字节数据是有编号的，它的访问顺序从1开始）。它会返回{ok,Bin}或者{error,Why}。

最后，我们会使用这些文件随机读取函数来编写一个下一章中将会用到的辅助函数。在14.7节中，我们会开发一个简单的SHOUTcast服务器（这是一种用于播放流媒体的服务器，这里想要播放的流媒体数据格式是MP3）。这个服务器中的一个组件需要能够从一个MP3文件中找到嵌入MP3文件中的艺术家和音轨名。下一节中我们就会编写这个程序。

13.2.6 读取 ID3 标记

MP3是一种用于存储压缩语音数据的二进制格式。MP3文件自己不会存储有关文件内容的信息，比如说在一个存放音乐的MP3中，创作这个音乐的艺术家的名称是不会包含在音频数据中的。这些数据（音轨名、艺术家名称，等等）都是存储在MP3文件中的一个标记格式区块内，这个区块被称为ID3。ID3标记是由程序员Eric Kemp发明的，用来存储描述音频文件内容的元数据。实际应用中ID3有着数个不同的版本，但就我们的目的而言，只需编写一段代码来访问两种格式最简单的ID3标记，分别为ID3v1和ID3v1.1标记。

ID3v1标记有一个简单的结构——文件的最后128字节包含了固定的长度标记。前3个字节包含了ASCII字符TAG，后面跟随许多固定长度的字段。整个128字节的编码，如表13-2所示。

表 13-2

长 度	内 容
3	头部包含了字符TAG
30	标题
30	艺术家
30	专辑
4	年度
30	注释
1	其他

225

在ID3v1标记中并没有预留地方来加入音轨号。Michael Mutschler在D3v1.1格式提出的一种可行的方法，这个主意的要点是把30字节的注释字段修改成下面这种格式，如表13-3所示。

表 13-3

长　度	内　容
28	注释
1	0(零)
1	音轨号

要试着在一个MP3文件中读取ID3v1标记并使用二进制数据的位匹配语法来匹配字段并不是一件很困难的事情。下面就是执行这些具体操作的代码：

id3_v1.erl

```
-module(id3_v1).
-import(lists, [filter/2, map/2, reverse/1]).
-export([test/0, dir/1, read_id3_tag/1]).

test() -> dir("/home/joe/music_keep").

dir(Dir) ->
    Files = lib_find:files(Dir, "*.mp3", true),
    L1 = map(fun(I) ->
                     {I, (catch read_id3_tag(I))}
             end,  Files),
    %% L1 = [{File, Parse}] where Parse = error | [{Tag,Val}]
    %% we now have to remove all the entries from L where
    %% Parse = error. We can do this with a filter operation
    L2 = filter(fun({_,error}) -> false;
                   (_) -> true
                end, L1),
    lib_misc:dump("mp3data", L2).

read_id3_tag(File) ->
    case file:open(File, [read,binary,raw]) of
        {ok, S} ->
            Size = filelib:file_size(File),
            {ok, B2} = file:pread(S, Size-128, 128),
            Result = parse_v1_tag(B2),
            file:close(S),
            Result;
        Error ->
            {File, Error}
    end.

parse_v1_tag(<<$T,$A,$G,
               Title:30/binary, Artist:30/binary,
               Album:30/binary, _Year:4/binary,
               _Comment:28/binary, 0:8,Track:8,_Genre:8>>) ->
    {"ID3v1.1",
     [{track,Track}, {title,trim(Title)},
```

```
      {artist,trim(Artist)}, {album, trim(Album)}]};
parse_v1_tag(<<$T,$A,$G,
              Title:30/binary, Artist:30/binary,
              Album:30/binary, _Year:4/binary,
              _Comment:30/binary,_Genre:8>>) ->
    {"ID3v1",
     [{title,trim(Title)},
      {artist,trim(Artist)}, {album, trim(Album)}]};
parse_v1_tag(_) ->
    error.

trim(Bin) ->
    list_to_binary(trim_blanks(binary_to_list(Bin))).

trim_blanks(X) -> reverse(skip_blanks_and_zero(reverse(X))).

skip_blanks_and_zero([$\s|T]) -> skip_blanks_and_zero(T);
skip_blanks_and_zero([0|T])   -> skip_blanks_and_zero(T);
skip_blanks_and_zero(X)       -> X.
```

程序的主入口点是`id3_v1:dir(Dir)`。首先我们做的事情是调用`lib_find:find(Dir, "*.mp3",true)`来搜寻所有需要的MP3文件（这个函数会在13.8节中详细讨论），它会递归地扫描`Dir`下面的目录来寻找MP3文件。一旦发现有MP3文件，就会调用`read_id3_tag`函数来解析它的标记。解析工作的步骤极为简单，因为我们可以使用位匹配语法来进行解析，然后再删除字串后用于分割字符串的空白符和0占位符来修整艺术家名称和音轨名。最后，再把结果写入一个文件以备后用（在附录的E.2节中会详细讨论函数`lib_misc:dump`）。

相当多的音乐文件都采用ID3v1标记，还有一些采用了ID3v2、V3、V4标记——这些后续版本的标记标准在文件头部加入了不同格式的标记（甚至还有的在文件当中加入标记，当然，这比较少见）。标记程序通常既添加ID3v1标记又在文件头中添加附加标记（比较难以识别）。单就我们的例子而言，只需关心含有合法ID3v1和ID3v1.1标记的文件。前面我们已经学到了如何读取文件，现在可以继续另外一个不同的话题——如何写入文件。 227

13.3 写入文件的不同方法

写入一个文件的方式与读取文件相当类似。下面就来看看这些方法。

13.3.1 向一个文件中写入一串 Erlang 数据项

如果我们希望创建一个通过`file:consult`函数来读取的文件，标准库中并没有与之对应的函数，因此我们必须要自己动手写一个。我将这个函数命名为`unconsult`。[①]

lib_misc.erl

```
unconsult(File, L) ->
```

① 写书的一大乐趣就是可以任意选择我喜欢的模块名和函数名，只要我喜欢，没什么不可以。

```
    {ok, S} = file:open(File, write),
    lists:foreach(fun(X) -> io:format(S, "~p.~n",[X]) end, L),
    file:close(S).
```

我们可以在shell中用如下的命令来创建test1.dat文件：

```
1> lib_misc:unconsult("test1.dat",
                      [{cats,["zorrow","daisy"]},
                      {weather,snowing}]).
ok
```

测试一下，看它是否能够正常工作：

```
2> file:consult("test1.dat").
{ok,[{cats,["zorrow","daisy"]},{weather,snowing}]}
```

要实现unconsult，我们需要用write模式打开一个文件，然后使用io:format(S,"~p.~n",[X])来将Erlang数据写入其中。

io:format函数是进行格式化输出的不二之选。为进行格式化输出，我们一般会这么调用这个函数。

```
@spec io:format(IoDevice, Format, Args) -> ok
```

ioDevice是一个IO设备（这个设备必须是以write模式打开的），Format是一个包含了格式化代码的字符串，而Args是一串需要输出的数据项。

对于Args中的每一个数据项，必须在格式化串中对应一个格式化命令。格式化命令是一个波浪号开头的字符串，下面这些是我们最常用到的格式化命令。

~n写入一个换行符。用~n写入一个换行符是一种独立于平台的方法。在Unix系统上，~n会写
228 ASCII（10）到输出流，在Windows系统上则会写一个回车换行符ASCII(13,10)到输出流。

~p完整打印参数。

~s参数是一个字符串。

~w以标准语法写入数据。这是输出Erlang数据项常会用到的命令。

字符串的格式化参数可以说是浩繁复杂，恐怕没有谁能够将它们记住。但你可以在用户手册找到关于io模块的完整列表。我所能记得也就是~p、~s、~n这3个参数。如果你能够掌握这3个参数，那么基本也就够用了。

题外话

我刚才说的只是安慰你一下罢了，你没有当真吧。事实上除了~p、~s、~n这3个常用的参数之外，实际应用中要用到的参数比这3个多得多，如表13-4所示。

表　13-4

格　式	结　果
io:format("|~10s|~n", ["abc"])	|␣␣␣␣␣␣␣abc|
io:format("|~-10s|~n", ["abc"])	|abc␣␣␣␣␣␣␣|
io:format("|~10.3.+s|~n",["abc"])	|+++++++abc|
io:format("|~10.10.+s|~n",["abc"])	|abc+++++++|
io:format("|~10.7.+s|~n",["abc"])	|+++abc++++|

13.3.2 向文件中写入一行

这和前面的样例非常相似，这里只是用一个不太一样的格式化命令而已：

```
1> {ok, S} = file:open("test2.dat", write).
{ok,<0.62.0>}
2> io:format(S, "~s~n", ["Hello readers"]).
ok
3> io:format(S, "~w~n", [123]).
ok
4> io:format(S, "~s~n", ["that's it"]).
ok
5> file:close(S).
```

上面的代码创建了一个名为test2.dat的文件，带有下面这些内容：

```
Hello readers
123
that's it
```

229

13.3.3 一步操作写入整个文件

在所有的写文件操作中，这是最为高效的方法。file:write_file(File,IO)把数据写入IO，这里的IO指的是文件File的IO列表。(所谓的IO列表指的就是一个列表，它的元素是二进制数据、0～255的整型或者IO列表本身。如果输出一个IO列表，它会自动平坦化，也就是说列表之中所有的方括号都会被删除。)这个方法极为高效，也是我经常会用到的函数。下一节中的代码会演示这个函数的用法。

从一个文件中生成URL列表

我们先来编写一个简单的名为urls2htmlFile(L,File)的函数。这个函数会接受一个URL列表L，然后创建一个HTML文件，这个HTML文件所在的URL可以看作一个可以点击的链接。这个程序可以让我们学到如何在单个IO操作中完成整个文件创建的技巧。

我们把程序放在模块scavenge_urls中。首先编写头部：

scavenge_urls.erl

```
-module(scavenge_urls).
-export([urls2htmlFile/2, bin2urls/1]).
-import(lists, [reverse/1, reverse/2, map/2]).

urls2htmlFile(Urls, File) ->
    file:write_file(File, urls2html(Urls)).

bin2urls(Bin) ->  gather_urls(binary_to_list(Bin), []).
```

这个程序有两个入口点。urls2htmlFile(Urls,File)接收一个URL列表，然后创建一个HTML文件，它包含每一个URL的可点击链接。bin2ulrs(Bin)对一个二进制数据进行搜索，并返回二进制数据中所有URL的列表。下面是urls2htmlFile的代码：

scavenge_urls.erl

```
urls2html(Urls) -> [h1("Urls"),make_list(Urls)].

h1(Title) -> ["<h1>", Title, "</h1>\n"].

make_list(L) ->
    ["<ul>\n",
     map(fun(I) -> ["<li>",I,"</li>\n"] end, L),
     "</ul>\n"].
```

这个代码返回一个字符串的嵌套列表。值得注意的是我们并没有去试图对列表平坦化处理（如果这么做将会非常低效），在我们创建了这个字符串嵌套列表之后，只是把它扔到一个输出函数中去。在我们使用file:write_file来把一个嵌套文件写入文件时，IO系统会自动地平坦化列
230 表（也就是说，它仅输出嵌在列表中的字符，而不会把列表的方括号输出）。最后，我们来看看下面这段代码，它将URL列表精确地转换成二进制数据：

scavenge_urls.erl

```
gather_urls("<a href" ++ T, L) ->
    {Url, T1} = collect_url_body(T, reverse("<a href")),
    gather_urls(T1, [Url|L]);
gather_urls([_|T], L) ->
    gather_urls(T, L);
gather_urls([], L) ->
    L.

collect_url_body("</a>" ++ T, L) -> {reverse(L, "</a>"), T};
collect_url_body([H|T], L)       -> collect_url_body(T, [H|L]);
collect_url_body([], _)          -> {[],[]}.
```

执行整个过程，我们还需要找到一些数据来进行解析。我们输入的数据（一个二进制数据）是一个HTML页面中的内容，因此如果我们要传给scavenge一个HTML页面，这需要使用函数socket_examples:nano_get_url（参见14.1节）。

下面我们在shell中一步一步地做这个测试：

```
1> B = socket_examples:nano_get_url("www.erlang.org"),
   L = scavenge_urls:bin2urls(B),
   scavenge_urls:urls2htmlFile(L, "gathered.html").
ok
```

这段代码生成了文件gathered.html：

gathered.html

```
<h1>Urls</h1>
<ul>
<li><a href="old_news.html">Older news.....</a></li>
<li><a href="http://www.erlang-consulting.com/training_fs.html">here</a></li>
<li><a href="project/megaco/">Megaco home</a></li>
<li><a href="EPLICENSE">Erlang Public License (EPL)</a></li>
```

```
<li><a href="user.html#smtp_client-1.0">smtp_client-1.0</a></li>
<li><a href="download-stats/">download statistics graphs</a></li>
<li><a href="project/test_server">Erlang/OTP Test
Server</a></li>
<li><a href="http://www.erlang.se/euc/06/">proceedings</a></li>
<li><a href="/doc/doc-5.5.2/doc/highlights.html">
      Read more in the release highlights.
</a></li>
<li><a href="index.html"><img src="images/erlang.gif"
    border="0" alt="Home"></a></li>
</ul>
```

231

13.3.4 在随机访问模式下写入文件

与在随机访问模式下的读文件类似，要在这个模式下工作，首先需要以write模式来打开这个文件。然后，需要用file:pwrite(Position,Bin)来写文件。

下面是一个小样例：

```
1> {ok, S} = file:open("...", [raw,write,binary])
{ok, ...}
2> file:pwrite(S, 10, <<"new">>)
ok
3> file:close(S)
ok
```

这个例子在文件的第10个字节处写入一串新的字符串，这会覆盖掉原先的内容。

13.4 目录操作

File模块有3个函数用来操纵目录。list_dir(Dir)函数用于生成Dir目录下的文件列表，make_dir(Dir)用来创建一个新的目录，还有del_dir(Dir)用来删除目录。

我在编写此书的同时也为你们提供了一套代码目录，如果你在代码目录下运行函数list_dir的话，就会看到如下的结果：

```
1> cd("/home/joe/book/erlang/Book/code").
/home/joe/book/erlang/Book/code
ok
2> file:list_dir(".").
{ok,["id3_v1.erl~",
     "update_binary_file.beam",
     "benchmark_assoc.beam",
     "id3_v1.erl",
     "scavenge_urls.beam",
     "benchmark_mk_assoc.beam",
     "benchmark_mk_assoc.erl",
     "id3_v1.beam",
     "assoc_bench.beam",
     "lib_misc.beam",
     "benchmark_assoc.erl",
```

```
"update_binary_file.erl",
"foo.dets",
"big.tmp",
..
```

显而易见的是，这个结果并不关心文件的排列顺序，也没有给出目录中每一个文件的属性：
232 到底哪个是目录，哪个是文件，每个文件的大小如何，诸如此类。

如果想要知道目录列表中每个文件的属性，可以用**file:read_file_info**函数。我们将在下一节着重讨论这个函数。

13.5　查询文件的属性

要查询一个文件F的属性，可以调用函数**file:read_file_info(F)**。如果F是一个合法的文件名或者目录名，那么这个函数会返回**{ok,Info}**。其中的**Info**是一个类型为**#file_info**的记录，这个记录的定义是这样的：

```
-record(file_info,
        {size,          % Size of file in bytes.
         type,          % Atom: device, directory, regular,
                        % or other.
         access,        % Atom: read, write, read_write, or none.
         atime,         % The local time the file was last read:
                        % {{Year, Mon, Day}, {Hour, Min, Sec}}.
         mtime,         % The local time the file was last written.
         ctime,         % The interpretation of this time field
                        % is dependent on operating system.
                        % On Unix it is the last time the file or
                        % or the inode was changed.  On Windows,
                        % it is the creation time.
         mode,          % Integer: File permissions.  On Windows,
                        % the owner permissions will be duplicated
                        % for group and user.
         links,         % Number of links to the file (1 if the
                        % filesystem doesn't support links).
         major_device,  % Integer: Identifies the file system (Unix),
                        % or the drive number (A: = 0, B: = 1) (Windows).
```

说明　mode和access字段的功能是重复的。你可以在一个操作中使用mode字段来设置多个文件的属性，而使用access的话操作则更简单。

要查询一个文件的类型和长度，我们可以像下面的例子一样调用**read_file_info**（注意，我们必须包含**file.hrl**文件，因为这个文件包含了记录**#file_info**的定义。）：

lib_misc.erl

```
-include_lib("kernel/include/file.hrl").
file_size_and_type(File) ->
    case file:read_file_info(File) of
```

```
        {ok, Facts} ->
            {Facts#file_info.type, Facts#file_info.size};
        _ ->
            error
    end.
```

233

好了，我们现在可以来扩充目录列表的信息。下面我们还会看到函数ls()会在list_file所返回的目录列表中加上文件属性。

lib_misc.erl

```
ls(Dir) ->
    {ok, L} = file:list_dir(Dir),
    map(fun(I) -> {I, file_size_and_type(I)} end, sort(L)).
```

好，现在如果我们再打印出文件列表，那么它们一定是经过排序并且附加了有用的信息：

```
1> lib_misc:ls(".").
[{"Makefile",{regular,1244}},
 {"README",{regular,1583}},
 {"abc.erl",{regular,105}},
 {"alloc_test.erl",{regular,303}},
 ...
 {"socket_dist",{directory,4096}},
 ...
```

为方便起见，filelib模块导出了一些日常使用的函数，诸如file_size(File)、is_dir(X)。这些函数只是对file:read_file_info的封装。如果我们只是想要得到文件长度的话，那么直接调用filelib:file_size毫无疑问要比先调用file:read_file_info再从#file_info记录中进行解析要方便得多。

13.6 复制和删除文件

函数file:copy(Source, Destination)将源文件Source复制到目标文件Destination。

函数file:delete(File)删除文件。

13.7 小知识

到目前为止，我们提及的大多文件操纵函数在我的编程工作中都是属于日常设施的范畴。需要去翻阅用户手册查询函数说明的情况并不常见。所以我讲解的内容，难免就会挂一漏万。那么还有哪些是你们必须知道却又被我忽略了的呢？这里就不全盘收录那些冗长的说明了，我给你列出一张简明的纲要。在你想要知道具体的信息时，就去查阅用户手册的相关部分吧。

- 文件模式。当我们使用file:open来打开一个文件时，其实是用了某一个特定或者复合的模式打开了这个文件。这些打开模式的实际数目远比我们想象的要多，比如，我们可以用compressed 模式来读写一个使用gzip压缩的文件，等等。你可以在用户手册里查到完整的信息。

234

- 修改文件的时间、群组、系统链接。你可以在file模块中找到与这些功能相关的函数。
- 错误代码。我可以很高兴的告诉你，所有的错误都是一个形如{error,Why}的元组。实际上，Why是一个原子（比如说，enonet的意思就是一个文件不存在，等等），错误代码种类很多，你可以在用户手册里找到它们的详细信息。
- filename模块。filename模块有几个相当有用的函数，你可以用它们来把文件名从一个完整的路径中提取出来，可以找到文件的扩展名，等等，当然还能帮助你拼出完整的文件名。更妙的是，所有这些操作都是平台独立的。
- filelib模块。filelib模块能够大大简化我们日常工作的函数并不是太多。filelib:ensure_dir(Name)函数就是这为数不多的其中之一，它会判断给定文件的目录及其所有父目录是否都存在，如若不存在，就会尝试创建它们。

13.8　一个搜索小程序

作为最后一个样例，我们将要使用file:list_dir和file:read_file_info两个函数来编写一个通用的“搜索”小工具。

指向这个模块的入口函数是这样的。

```
lib_find:files(Dir, RegExp, Recursive, Fun, Acc0)
```

这些参数的意义如下。

Dir

开始执行文件搜索的目录名。

RegExp。

一个用来测试文件名是否匹配的正则表达式[①]。如果遇到与正则表达式相匹配的文件，就会调用Fun(File,Acc)。这里的File是指与正则表达式匹配的文件。

Recursive = true | false。

235　这是一个标志，决定是否要对搜索路径的子目录进行递归搜索。

Fun(File, AccIn) -> AccOut。

若regExp能与File匹配，那么程序就会调用该函数。Acc是一个累加器，它的初始值为Acc0。程序每次调用时，都会向Fun传送Acc，并且在Fun运行完毕后返回新值作为下次调用的参数。累加器最终的结果也就是函数lib_find: files/5的最终结果。

我们可以向lib_find:files/5传入任何想要的函数。比如说，我们可以使用下面这个函数来构建一个文件列表，初始传入一个空的列表：

```
fun(File, Acc) -> [File|Acc] end
```

模块的入口函数lib_find:files(Dir,ShellRegExp,Flag)提供了一个简化的接口，以保持程序的可扩展性。这里的ShellRegExp是一种正则表达式的简化版本，比起完整版本的正则表达式，它更容易编写。

① 要想了解正则表达式的具体信息，请参看regexp模块的用户手册。

下面这行代码就是这个函数的精简版调用：

```
lib_find:files(Dir, "*.erl" , true)
```

这个调用会递归搜索Dir目录（及其子目录）下所有的Erlang文件。若将最后一个参数改为false，那就只会搜索Dir目录下的文件而不会继续深入它的子目录进行搜索。

最后，我们这里给出lib_find的完整代码：

lib_find.erl

```
-module(lib_find).
-export([files/3, files/5]).
-import(lists, [reverse/1]).

-include_lib("kernel/include/file.hrl").

files(Dir, Re, Flag) ->
    Re1 = regexp:sh_to_awk(Re),
    reverse(files(Dir, Re1, Flag, fun(File, Acc) ->[File|Acc] end, [])).

files(Dir, Reg, Recursive, Fun, Acc) ->
    case file:list_dir(Dir) of
        {ok, Files} -> find_files(Files, Dir, Reg, Recursive, Fun, Acc);
        {error, _}  -> Acc
    end.
find_files([File|T], Dir, Reg, Recursive, Fun, Acc0) ->
    FullName = filename:join([Dir,File]),
    case file_type(FullName) of
        regular ->
            case regexp:match(FullName, Reg) of
                {match, _, _}  ->
                    Acc = Fun(FullName, Acc0),
                    find_files(T, Dir, Reg, Recursive, Fun, Acc);
                _ ->
                    find_files(T, Dir, Reg, Recursive, Fun, Acc0)
            end;
        directory ->
            case Recursive of
                true ->
                    Acc1 = files(FullName, Reg, Recursive, Fun, Acc0),
                    find_files(T, Dir, Reg, Recursive, Fun, Acc1);
                false ->
                    find_files(T, Dir, Reg, Recursive, Fun, Acc0)
            end;
        error ->
            find_files(T, Dir, Reg, Recursive, Fun, Acc0)
    end;
find_files([], _, _, _, _, A) ->
    A.
```

236

```
file_type(File) ->
    case file:read_file_info(File) of
        {ok, Facts} ->
            case Facts#file_info.type of
                regular   -> regular;
                directory -> directory;
                _         -> error
            end;
        _ ->
            error
    end.
```

237

第14章

套接字编程

在我编写的程序中，那些涉及套接字的程序往往要比其他程序更加有趣。套接字是一个通信终端，它允许计算机在因特网上使用IP协议进行通信。在本章中，我们会关注于因特网的两个核心协议：TCP（传输控制协议）和UDP（用户数据报协议）。

UDP使得应用程序之间互相传递短消息（也就是报文）成为可能。不过它的消息传输不是可靠的，传输甚至还可能打乱数据的顺序。与之相对，TCP则提供了一种可靠的字节流，连接一旦建立，这些字节流就会按顺序传输。

为什么套接字编程很有趣呢？这是因为它允许应用程序彼此之间在因特网上进行交互，这种交互显然要比纯粹的本地代码具有更多的可能性。

Erlang中针对套接字的编程主要涉及两个库：gen_tcp用于TCP，gen_udp则用于UDP。

本章之中，我们会看到如何使用TCP和UDP套接字来编写客户机和服务器。我们会学到各种不同工作模式下的服务器：并行的、顺序的、阻塞的、非阻塞的。与此同时，我们也会学到如何去编写具备流控制（traffic-shapping）能力的应用程序，这些程序通常都用来控制应用程序之间的数据流向。

14.1 使用 TCP

我们将从一个TCP的简单例子开始我们的套接字编程学习。这个例子只是简单的从一个服务器获取数据。在此之后我们会编写一个简单的顺序型TCP服务器，然后看看我们如何将其并行化以便能够控制多个并行的会话。

14.1.1 从服务器上获取数据

我们先从编写一个小的函数开始[①]。这个小函数会使用TCP套接字从http://www.google.com上抓取一个HTML页面：

socket_examples.erl

```
nano_get_url() ->
    nano_get_url("www.google.com").
```

① Erlang发布版中的标准库函数http:request(Uri)，实现了类似的功能，但这里我们的目的是要弄清楚如何使用gen_tcp中的库函数来实现这个功能。

```
nano_get_url(Host) ->
❶    {ok,Socket} = gen_tcp:connect(Host,80,[binary, {packet, 0}]),
❷    ok = gen_tcp:send(Socket, "GET / HTTP/1.0\r\n\r\n"),
     receive_data(Socket, []).

receive_data(Socket, SoFar) ->
     receive
❸        {tcp,Socket,Bin} ->
             receive_data(Socket, [Bin|SoFar]);
❹        {tcp_closed,Socket} ->
❺            list_to_binary(reverse(SoFar))
     end.
```

这段代码是如何工作的?

❶ 程序通过调用gen_tcp:connect打开一个TCP套接字，它会连到http://www.google.com的80端口上。连接调用中的参数binary告诉系统以binary模式打开套接字，在这个模式下应用程序之间的数据传输都被认为是二进制格式的。{packet,0}意味着Erlang系统会把TCP数据原封不动地直接传送给应用程序。

❷ 程序调用gen_tcp:send函数将消息GET/HTTP/1.0\r\n\r\n发送到套接字。然后等待回应。回应消息通常不会在一个数据包内全部返回，而是会一帧一帧地返回，每帧一部分数据。程
240 序以顺序型消息的方式收到这些数据帧并把它转发给打开套接字（或者控制套接字）的进程。

❸ 程序会收到一个{tcp, Socket, Bin}消息。这个元组中的第3个参数是二进制类型。这就是我们为何以二进制模型打开套接字的原因所在。这个消息只是Web服务器发给程序的众多的数据帧之一。程序会把它添加到一个列表之中，这个列表汇聚了程序目前接收到的所有数据，然后程序继续等待接收下一帧。

❹ 当程序收到一个{tcp_closed, Socket}消息的时候，这就表明Web服务器已经停止向程序发送数据。[①]

❺ 因为在接收时我们是以错误的顺序来存储这些数据帧的，因此在所有的数据帧都接收完毕之后，我们需要调整顺序，并拼接所有数据帧。

简单的测试一下这个例子:

```
1> B = socket_examples:nano_get_url().
<<"HTTP/1.0 302 Found\r\nLocation: http://www.google.se/\r\n
   Cache-Control: private\r\nSet-Cookie: PREF=ID=b57a2c:TM"...>>
```

说明　在你运行nano_get_url函数时，它的返回结果是一个二进制数据，因此我们看到的只是它在Erlang shell中的打印格式。如果将二进制数据完整打印出来，那么所有的控制字符都会按照转义符的格式来显示。而如果二进制数据太长也会被截断，在输出的末尾被截断的部分会用3个点来表示(...>>)。如果你想观察所有的二进制数据，那么可以使用io:format来打印或者使用string:tokens来将它们切成小的片段。

① 只有在HTTP/1.0协议中才是这样。对于那些高版本的HTTP协议来说，它们会采取不同的处理方案。

```
2> io:format("~p~n",[B]).
<<"HTTP/1.0 302 Found\r\nLocation: http://www.google.se/\r\n
   Cache-Control: private\r\nSet-Cookie: PREF=ID=b57a2c:TM"
   TM=176575171639526:LM=1175441639526:S=gkfTrK6AFkybT3;
   expires=Sun, 17-Jan-2038 19:14:07
   ... several lines omitted ...
>>
3>string:tokens(binary_to_list(B),"\r\n").
["HTTP/1.0 302 Found",
"Location: http://www.google.se/",
"Cache-Control: private",
"Set-Cookie: PREF=ID=ec7f0c7234b852dece4:TM=11713424639526:
 LM=1171234639526:S=gsdertTrK6AEybT3;
 expires=Sun, 17-Jan-2038 19:14:07 GMT; path=/; domain=.google.com",
"Content-Type: text/html",
"Server: GWS/2.1",
"Content-Length: 218",
"Date: Fri, 16 Feb 2007 15:25:26 GMT",
"Connection: Keep-Alive",
... lines omitted ...
```

241

这个程序或多或少地向我们展示了一个Web客户端的工作机制。（实际上，我们还有大量工作没有完成——这涉及如何在一个Web浏览器上显示我们接收到的结果。）不过，前面这个样例为我们后面的学习开了一个好头。你可能会去尝试把这段代码修改一下，让它能够抓取一个网站的所有内容并且保存下来，或者自动地去检索和收取你的电子邮件。只要你愿意，没有什么做不到的。

注意，重整数据帧的那部分代码实际上是这样的：

```
receive_data(Socket, SoFar) ->
    receive
        {tcp,Socket,Bin} ->
            receive_data(Socket, [Bin|SoFar]);
        {tcp_closed,Socket} ->
            list_to_binary(reverse(SoFar))
    end.
```

在数据帧到达时，程序只是简单的把数据帧加入列表SoFar的头部。当程序接收完所有的帧并且套接字也被关掉之后，程序会反转列表，然后把它们拼接起来。

你可能会以为下面这种写法来完成数据帧的收集工作要更加简洁：

```
receive_data(Socket, SoFar) ->
    receive
        {tcp,Socket,Bin} ->
            receive_data(Socket, list_to_binary([SoFar,Bin]));
        {tcp_closed,Socket} ->
            SoFar
    end.
```

这段代码的确可以正常工作，但是它要比之前的那个版本低效得多。这是因为后一个版本中我们要持续地向一个缓冲区中追加新的二进制数据，这种行为会产生许多无用的数据副本。因此

最好的方法是在一个列表中收集所有的数据帧（此时，在所有的数据帧都添加完时的顺序是逆序），然后翻转整个列表并把所有的数据在一个操作内拼接起来。

14.1.2　一个简单的TCP服务器

上一节里，我们尝试编写了一个简单的客户端，现在我们再来编写一个服务器。

这个服务器打开2345端口然后等待一条消息。这条消息是一个二进制的数据，其中包含了一个Erlang数据项。这个Erlang的数据项是一个字符串，内容则是一个表达式。服务器对这个表达
242 式进行求值，然后通过套接字把运算结果发送给客户端。

如何编写一个Web服务器

编写Web客户端或者服务器的程序，这是一件非常有趣的工作。的确，有很多人已经写过这些东西了，但如果我们想要真的理解其中的工作机制，那么挖掘这些软件背后的结构，分析出它们真正的工作原理是非常有帮助的。谁又能保证——我们自己的Web服务器不能超过那些现存的最好产品呢？那么，我们如何来着手这件事呢？

要创建一个Web服务器，或者写一个软件来实现某个标准的因特网协议，我们首先需要知道这个协议的具体内容是什么，其次，工欲善其事，必先利其器，好的工具软件也必不可少。

在抓取网页的例子代码中，我们如何知道必须要打开80端口，我们又是如何知道必须向服务器发送一个GET/HTTP/1.0\r\n\r\n这样的命令？答案很简单，所有针对因特网服务的主要协议都被定义在请求注解RFC（requests for comment）当中。RFC1945就定义了HTTP/1.0协议。RFC的官方网站是http://www.ietf.org（因特网工程任务组的主页）。

一个数据包嗅探器也是我们需要的好帮手。有了数据包嗅探器我们就能抓取和分析所有进出应用程序的IP数据包。大多数的数据包嗅探器都包括了可以用来对数据包进行解码和分析的工具，这些工具还能把这些数据还原成有意义的格式呈现给我们。其中Wireshark是这些软件之中最负盛名的一个（以前叫做Ethereal），你可以从http://www.wireshark.org下载它。

243 有了数据包嗅探器和相应的RFC文档，我们就可以着手开发我们自己的杀手级应用程序了。

要编写这种程序（实际上任何一种基于TCP/IP的程序都是如此），我们必须问自己一些简单的问题。

- 我们要如何组织数据？如何知道一次请求或响应的消息之中含有多少数据？
- 在请求或响应的过程中，数据该如何编码和解码[对数据编码的工作有时也被称为整编（marshaling），而对数据解码的工作则被称为反整编（demarshaling）]。

TCP套接字的数据是一种无差异的字节流。数据在传输中，可以被切分成任意尺寸的数据帧，因此我们需要做一些转换才能让程序知道单个请求或响应到底含有多少数据。

在Erlang中我们使用的转换方法很简单，每一个逻辑请求或响应前都会带上一个N（1、2或4）字节长的长度计数。这就是gen_tcp:connect函数和gen_tcp:listen函数中{packet,N}[①]参数的

① 这里我们所说的打包（pakcet）指的是应用程序中一个请求和响应的长度，而不是在物理网络上的数据包。

含义。注意，服务器和客户端使用的packet参数必须一致。如果服务器套接字以{packet,2}方式打开，而客户端以{packet,4}打开，那么程序就不能收到任何有用的消息。

在使用{packet,N}选项打开一个套接字后，我们就无须担心数据帧了。Erlang的内置驱动器会确保在把数据发送到应用程序之前，所有数据消息的数据帧都会按照其长度正确接收。

下一个需要关心的问题是对数据的编码和解码。我们使用最为简单可行的方案，用term_to_binary函数来对Erlang数据项进行编码，然后用它的反函数binary_to_term来对数据进行解码。

值得注意的是，我们仅需两行代码就能实现客户端和服务器通信所需的数据包规则和编码规则。首先是使用{packet,4}选项来打开一个套接字，其次是使用term_to_binary和它的反函数来对数据进行编码和解码。

Erlang系统为Erlang数据项的打包和编码解码提供了非常便利的设施，这在处理基于文本的协议如HTTP或者XML之类的领域中给予我们极大的优势。在小规模数据的处理上，使用Erlang内置的term_to_binary函数及其反函数binary_to_term比使用XML数据的操作，速度要快出一个数量级。 244

socket_examples.er

```
start_nano_server() ->
❶       {ok, Listen} = gen_tcp:listen(2345, [binary, {packet, 4},
                                             {reuseaddr, true},
                                             {active, true}]),
❷       {ok, Socket} = gen_tcp:accept(Listen),
❸       gen_tcp:close(Listen),
        loop(Socket).

loop(Socket) ->
    receive
        {tcp, Socket, Bin} ->
            io:format("Server received binary = ~p~n",[Bin]),
❹           Str = binary_to_term(Bin),
            io:format("Server (unpacked)  ~p~n",[Str]),
❺           Reply = lib_misc:string2value(Str),
            io:format("Server replying = ~p~n",[Reply]),
❻           gen_tcp:send(Socket, term_to_binary(Reply)),
            loop(Socket);
        {tcp_closed, Socket} ->
            io:format("Server socket closed~n")
    end.
```

这个例子又是如何工作的？

❶ 首先程序调用gen_tcp:listen监听来自端口2345上的连接，然后设置消息打包规则。{packet, 4}的意思是每一个应用程序消息都是从一个4字节长的头部开始的。

然后函数gen_tcp:listen(..)返回{ok, Socket}或者{error,Why}，但我们这里关心的只是可以返回给我们一个已打开套接字的情况。因此这里程序是这样编写的：

```
{ok, Listen} = gen_tcp:listen(...),
```

如果gen_tcp:listen返回{error,...}的话，那么这句就会导致一个模式匹配的异常。而在正常的流程里，这句话会把变量Listen与监听到的套接字绑定起来。接下来我们对套接字所作的处理只有一个，那就是把套接字作为一个参数送入函数gen_tcp:accept。

❷ 在我们调用函数gen_tcp:accept(Listen)时，程序会在这里暂停并等待一个连接。当一个新的连接建立起来时，这个函数会返回变量Socket，该变量绑定到新建连接的套接字上，通过
245 这个套接字，服务器就可以与那个发起连接的客户机进行通信。

❸ 当accept函数返回之后，我们立刻调用gen_tcp:close(Listen)。这个函数关闭监听套接字，此后服务器不会继续处于监听状态，也无法建立任何新的连接。这个操作并不影响已经建立起来的连接，它只是阻止新连接的建立。

❹ 程序对输入数据进行解码（反整编）。

❺ 对字符串进行求值。

❻ 对回应消息进行编码（整编）然后把它发回套接字。

值得注意的是，这个程序只会接收单个请求，一旦程序运行完成，那就不会再继续接收新的连接。

这就是一个最简单的服务器，它向我们展示了如何对应用程序的数据进行打包和编码。它接收一个请求，计算出一个回应，把回应送回，然后自行关闭。

要测试这个服务器，我们需要一个对应的客户端:

socket_examples.erl

```
nano_client_eval(Str) ->
    {ok, Socket} =
        gen_tcp:connect("localhost", 2345,
                        [binary, {packet, 4}]),
    ok = gen_tcp:send(Socket, term_to_binary(Str)),
    receive
        {tcp,Socket,Bin} ->
            io:format("Client received binary = ~p~n",[Bin]),
            Val = binary_to_term(Bin),
            io:format("Client result = ~p~n",[Val]),
            gen_tcp:close(Socket)
    end.
```

为了测试这些代码，我们可以在同一台机器上同时运行客户端和服务器，因此在gen_tcp:connect函数中主机名参数就被硬编码为localhost。

注意，客户端程序是如何调用term_to_binary函数来对消息编码的，它又如何用binary_to_term来重组消息。

246 要运行这个测试，我们需要打开两个终端窗口，然后在每个窗口上启动一个Erlang shell。

首先我们启动服务器：

```
1> socket_examples:start_nano_server().
```

在服务器窗口里不会看到任何的输出，因为我们还没有接收到任何的数据。然后切换到客户端窗口，输入：

```
1> socket_examples:nano_client_eval("list_to_tuple([2+3*4,10+20])").
```

在服务器窗口我们应该会看到下面这些消息：

```
Server received binary = <<131,107,0,28,108,105,115,116,95,116,
                           111,95,116,117,112,108,101,40,91,50,
                           43,51,42,52,44,49,48,43,50,48,93,41>>
Server (unpacked)  "list_to_tuple([2+3*4,10+20])"
Server replying = {14,30}
```

在客户端窗口我们则会看到下面这些消息：

```
Client received binary = <<131,104,2,97,14,97,30>>
Client result = {14,30}
ok
```

最后，在服务器窗口，我们还会看到这么一行字：

```
Server socket closed
```

14.1.3 改进服务器

上面一节里我们编写了一个服务器，它只接收一个连接然后自行关闭。我们只要对这段代码做一点小改动，就能得到如下两种截然不同的服务器。

(1) 一种是顺序型服务器——一个服务器一次只接收一个连接。

(2) 一种是并行服务器——一个服务器同时可以接收多个并行的连接。

上一节中的例子代码是这样的：

```
start_nano_server() ->
    {ok, Listen} = gen_tcp:listen(...),
    {ok, Socket} = gen_tcp:accept(Listen),
    loop(Socket).
 ...
```

要通过修改这段代码来产生这两种服务器。 247

1. 顺序型服务器

要编写一个顺序型服务器，我们只需要把代码改成这样：

```
start_seq_server() ->
    {ok, Listen} = gen_tcp:listen(...),
    seq_loop(Listen).

seq_loop(Listen) ->
    {ok, Socket} = gen_tcp:accept(Listen),
    loop(Socket),
    seq_loop(Listen).

loop(..) -> %% as before
```

这与前一节中的例子极为相似，但我们希望服务器不仅仅只服务一个请求，这就需要让监听端口继续打开而不是调用gen_tcp:close(Listen)。另外一个不同之处就在于当loop(Socket)运行完成之后，我们会再次调用seq_loop(listen)，它会继续等待下一个客户端的连接。

当服务器忙于服务一个现存的连接时，如果又有新的客户机尝试连接服务器，那么这个连接就会在服务器上排队，直到服务器完成对现有连接的服务为止。如果等待队列中的连接数目超过了监听套接字的能力，那么这个连接就会被拒绝。

这里的程序仅仅涉及了如何启动服务器。关闭服务器的代码也是非常简单的（并发服务器的关闭也是如此），只要终止已经启动的服务器进程或者服务器进程组就可以了。gen_tcp会在自己与控制进程之间建立连接，如果控制进程消亡，那么它也会关闭对应的套接字。

2. 并行服务器

编写一个并行服务器的技巧在于每次当gen_tcp:accept函数接收到一个新的连接时就去启动一个新进程：

```
start_parallel_server() ->
    {ok, Listen} = gen_tcp:listen(...),
    spawn(fun() -> par_connect(Listen) end).

par_connect(Listen) ->
    {ok, Socket} = gen_tcp:accept(Listen),
    spawn(fun() -> par_connect(Listen) end),
    loop(Socket).

loop(..) -> %% as before
```

这个例子与前面我们看到的顺序型服务器大同小异。最主要的不同之处在于这个例子中添加了启动进程的代码，它会确保程序为每一个新建的套接字连接创建一个新的进程。现在正好可以对比一下这两个例子。你应该留意到spawn语句所处的位置，也应该注意到我们如何把一个顺序型服务器修改为一个并行服务器。

248

这3个服务器都调用gen_tcp:listen和gen_tcp:accept，不同之处在于程序是在一个并行的代码之中还是在一个顺序型的代码之中调用这些函数。

14.1.4　注意

请注意下列事项。

- 创建一个套接字的进程（通过调用gen_tcp:accept或gen_tcp:connect）也就是所说的该套接字的控制进程。这个套接字所收到的任何消息都会转发给这个控制进程，如果控制进程消亡，那么该套接字也会自行关闭。我们可以通过调用gen_tcp:controlling_process(Socket, NewPid)函数来把一个套接字的控制进程改为新的控制进程NewPid。
- 我们的并行服务器具有可以支持几千个连接的并发潜力。但我们仍然会希望能限制同时处理连接的数量。这可以通过维护一个计数器来实现，这个计数器统计在某一时刻共有多少连接存活。每当程序建立一个新的连接就对这个计数器加1，每当一个连接终止时就

将这个计数器减1。通过这个计数器就可以限制系统之中同时运行的连接数量。

- 在接收到一个连接的时候，就显式地设置请求套接字的属性，这是一个非常好的主意：

```
{ok, Socket} = gen_tcp:accept(Listen),
inet:setopts(Socket, [{packet,4},binary,
                      {nodelay,true},{active, true}]),
loop(Socket)
```

- 从R11B-3版本开始，Erlang可以允许若干个进程对同一个监听套接字调用函数gen_tcp:accept/1。这能简化并行服务器的开发，因为你可以有一个预先生成的进程池，它们都在等待gen_tcp:accept/1。

14.2 控制逻辑

Erlang的套接字可以以3种模式打开：active、active once或passive。我们可以在调用gen_tcp:connect(Address, Port, Options)或者gen_tcp:listen(Port, Options)时通过选项{active, true | false | once}设置套接字的参数来实现。 249

如果指定了{active, true}，那么程序会创建一个主动套接字；而{active false}则会创建一个被动套接字；{active, once}也会创建一个主动套接字，但这个套接字仅接收一条消息，当它接收完这条消息之后，如果你想让它接收下一条消息，那么就必须再次激活它。

在下面的几节里我们会用到所有这些不同类型的套接字。

主动套接字和被动套接字的区别在于当套接字收到消息之后的工作机制：

- 建立主动套接字之后，当数据到达时系统会向控制进程发送{tcp, Socket, Data}消息。而控制进程无法控制这些消息流。一个独立的客户机有可能会向系统发送成千上万条消息，而这些消息都会被转送到控制进程。但控制进程却无法停掉这个消息流。
- 如果套接字以被动模式打开，那么控制进程必须调用gen_tcp:recv(Socket, N)来接收来自于套接字的数据。它会尝试从套接字接收N字节的数据。如果N为0，那么所有可用的字节数据都会返回。在这种情况下，服务器可以通过选择调用gen_tcp:recv的时机来控制来自客户机的消息流。

被动套接字用来控制向服务器发送的数据流。为了演示这3个模式，我们可以用如下3种方式来编写一个服务器的消息接收循环。

- 主动型消息接收（非阻塞）。
- 被动型消息接收（阻塞）。
- 混合型消息接收（半阻塞）。

14.2.1 主动型消息接收(非阻塞)

我们的第一个例子会用主动模式打开一个套接字，然后从套接字接收消息：

```
{ok, Listen} = gen_tcp:listen(Port, [..,{active, true}...]),
{ok, Socket} = gen_tcp:accept(Listen),
loop(Socket).
```

250

```
loop(Socket) ->
    receive
        {tcp, Socket, Data} ->
            ... do something with the data ...
        {tcp_closed, Socket} ->
            ...
    end.
```

这个进程无法控制服务器循环中的消息流。如果客户端发送的数据快过服务器可以处理的速度，那么系统就会被消息淹没——消息缓冲区会被塞满，系统可能会莫名其妙的崩溃。

这种类型的服务器被称为异步服务器，因为它不会阻塞客户端。只有在我们可以确信服务器的性能能够跟上客户机的需求时，才应该选择使用异步服务器。

14.2.2　被动型消息接收(阻塞)

在这一节中，我们会编写一个阻塞服务器。这个服务器通过设置{active, false}选项来以被动模式打开一个套接字，它不会因为一个过于活跃的客户机通过发送大量数据的攻击而崩溃。

这段代码之中的服务器循环在每次程序想要接收数据的地方都会调用gen_tcp:recv。调用recv函数时客户端会被阻塞。需要注意的是，操作系统还会做一些缓存允许客户机继续发送少量的数据，然后才会将其阻塞，此时Erlang还没有调用到recv函数。

```
{ok, Listen} = gen_tcp:listen(Port, [..,{active, false}...]),
{ok, Socket} = gen_tcp:accept(Listen),
loop(Socket).

loop(Socket) ->
    case gen_tcp:recv(Socket, N) of
        {ok, B} ->
            ... do something with the data ...
            loop(Socket);
        {error, closed}
            ...
    end.
```

14.2.3　混合型模式(半阻塞)

你可能会认为对任何一种类型的服务器使用被动模式都是正确的。但糟糕的是，在程序处于
251 被动模式的时候，我们只能等待来自一个套接字的数据。这在必须同时等待多个套接字的数据的程序时，就显得无能为力了。

幸运的是，我们还可以采用一种混合策略，即不是阻塞的也不是非阻塞的。我们可以用{active,once}选项打开套接字。在这个模式中，套接字是主动的但是仅仅针对一个消息。在控制进程发过一个消息后，必须显式地调用函数inet:setopts来把它重新激活以便接收下一个消息。在此之前，系统会处于阻塞状态。这种方法兼具了上面两种方式的优点。下面我们就来看看混合型模式是如何工作的：

```
{ok, Listen} = gen_tcp:listen(Port, [..,{active, once}...]),
```

```
    {ok, Socket} = gen_tcp:accept(Listen),
    loop(Socket).

    loop(Socket) ->
        receive
            {tcp, Socket, Data} ->
                ... do something with the data ...
                %% when you're ready enable the next message
                inet:setopts(Sock, [{active, once}]),
                loop(Socket);
             {tcp_closed, Socket} ->
                ...
        end.
```

使用{active,once}选项，用户可以实现高级的控制流[有时我们称之为流量控制（traffic-shapping）]，它可以有效的防止服务器被过多的消息淹没。

14.3 连接从何而来

假设我们编写了某种在线服务，某一天我们发现有人向服务器发送垃圾信息，我们该如何处理呢？首先我们要搞清楚这些连接是从哪里来的。为此，我们可以调用函数inet:peername(Socket)。

```
@spec inet:peername(Socket) -> {ok, {IP_Address, Port}} | {error, Why}
```

这个函数返回该连接另外一段的IP地址和端口号，因此服务器就可以发现是谁发起了这些连接。IP_Address使用一个整型元组{N1,N2,N3,N3}表示IPV4的IP地址，使用{K1,K2,K3,K4,K5,K6,K7,K8}表示IPV6的地址。这里的Ni和Ki表示的都是从0~255的整型数。 252

14.4 套接字的出错处理

套接字的出错处理是极为简单的——基本上你不需要做任何事情。正如前文所述，每一个套接字都有一个控制进程（也就是创建这个套接字的进程）。如果控制进程消亡，那么套接字也会自动关闭。

例如，如果我们有一个客户端和一个服务器，如果服务器因为程序的逻辑错误而崩溃，那么服务器的套接字就会被自动关闭，同时客户机也会收到一个{tcp_closed, Socket}的消息。

我们可以用下面这个小程序来测试这种机制：

socket_examples.erl

```
error_test() ->
    spawn(fun() -> error_test_server() end),
    lib_misc:sleep(2000),
    {ok,Socket} = gen_tcp:connect("localhost",4321,[binary, {packet, 2}]),
    io:format("connected to:~p~n",[Socket]),
    gen_tcp:send(Socket, <<"123">>),
    receive
        Any ->
```

```
            io:format("Any=~p~n",[Any])
    end.

error_test_server() ->
    {ok, Listen} = gen_tcp:listen(4321, [binary,{packet,2}]),
    {ok, Socket} = gen_tcp:accept(Listen),
    error_test_server_loop(Socket).

error_test_server_loop(Socket) ->
    receive
        {tcp, Socket, Data} ->
            io:format("received:~p~n",[Data]),
            atom_to_list(Data),
            error_test_server_loop(Socket)
    end.
```

运行它，我们会看到这样的信息：

```
1> socket_examples:error_test().
connected to:#Port<0.152>
received:<<"123">>
=ERROR REPORT==== 9-Feb-2007::15:18:15 ===
Error in process <0.77.0> with exit value:
 {badarg,[{erlang,atom_to_list,[<<3 bytes>>]},
 {socket_examples,error_test_server_loop,1}]}
Any={tcp_closed,#Port<0.152>}
ok
```

253

我们开启一个服务器，等待两秒让它开启，然后向它发送一个消息，其内容为一个二进制数据<<"123">>。当服务器收到这个消息后，会尝试对函数atom_to_list(Data)进行求值。但由于Data是二进制类型的，所以函数会立刻抛出异常。这时服务器上的套接字控制进程崩溃，[①]而与之对应的套接字（服务器端）也会自动关闭。客户端会收到一个{tcp_closed,Socket}消息。

14.5 UDP

现在让我们看看UDP（User Datagram Protocal，用户数据报协议）。在因特网上，任意的两台计算机之间都可以使用UDP协议来发送被称做报文的短消息。UDP报文并不可靠，这也就意味着如果客户端向服务器发送一串UDP报文，那么这些数据报有可能是乱序到达的，当然有的时候也完全不会出错，此外，有的时候报文还会非常频繁。但如果单个数据报能够到达目的地，它必定是完整。大型的数据报会被切分成小块的数据帧，底层的IP协议会在这些数据报发送之前对其进行重组。

UDP是一个无连接的协议，这就意味着客户机在发送消息之前不需要向服务器发起一个连接，这同时也意味着UDP非常适合于那些有大量客户端向服务器发送短消息的应用程序。

在Erlang中编写UDP的客户端和服务器比TCP还要简单得多，因为我们无须费心去关注如何维护与服务器的连接。

① 系统的监视器会打印出你在shell之中看到的诊断信息。

14.5.1 最简单的 UDP 服务器和客户机

下面让我们来看看UDP服务器。UDP服务器常见的形式是这样的：

```
server(Port) ->
{ok, Socket} = gen_udp:open(Port, [binary]),
loop(Socket).

loop(Socket) ->
    receive
       {udp, Socket, Host, Port, Bin} ->
           BinReply = ... ,
           gen_udp:send(Socket, Host, Port, BinReply),
           loop(Socket)
    end.
```

254

这个例子比对应的TCP程序稍微简单一些，这是因为我们不用关心进程是否会收到“socket closed”消息。值得注意的是在我们打开一个套接字时使用的是二进制模式，这个模式意味着系统驱动程序向控制进程发送的消息都是二进制的数据。

现在再来看看客户机。下面是一个非常简单的客户机代码，它只是打开了一个UDP套接字，向服务器发送消息，等待一个回应（或者超时），然后关闭套接字返回由服务器传回的值。

```
client(Request) ->
    {ok, Socket} = gen_udp:open(0, [binary]),
    ok = gen_udp:send(Socket, "localhost", 4000, Request),
    Value = receive
               {udp, Socket, _, _, Bin} ->
                   {ok, Bin}
            after 2000 ->
               error
            end,
    gen_udp:close(Socket),
    Value
```

我们必须在这个程序之中设置一个超时，这是因为UDP协议的传输是不可靠的，我们有可能得不到来自服务器的回应。

14.5.2 一个计算阶乘 UDP 的服务器

我们可以非常容易地编写一个UDP服务器，这个服务器可以对任何接收到的数值进行阶乘运算。这个例子只对前面的那个程序进行了稍许修改。

udp_test.erl

```
-module(udp_test).
-export([start_server/0, client/1]).

start_server() ->
    spawn(fun() -> server(4000) end).
```

```
%% The server
server(Port) ->
    {ok, Socket} = gen_udp:open(Port, [binary]),
    io:format("server opened socket:~p~n",[Socket]),
    loop(Socket).

loop(Socket) ->
    receive
        {udp, Socket, Host, Port, Bin} = Msg ->
            io:format("server received:~p~n",[Msg]),
            N = binary_to_term(Bin),
            Fac = fac(N),
            gen_udp:send(Socket, Host, Port, term_to_binary(Fac)),
            loop(Socket)
    end.
fac(0) -> 1;
fac(N) -> N * fac(N-1).

%% The client

client(N) ->
    {ok, Socket} = gen_udp:open(0, [binary]),
    io:format("client opened socket=~p~n",[Socket]),
    ok = gen_udp:send(Socket, "localhost", 4000,
                      term_to_binary(N)),
    Value = receive
                {udp, Socket, _, _, Bin} = Msg ->
                    io:format("client received:~p~n",[Msg]),
                    binary_to_term(Bin)
            after 2000 ->
                    0
            end,
    gen_udp:close(Socket),
    Value.
```

255

注意，我在代码之中添加了一些打印语句，以便可以观察程序的具体执行情况。在开发一个程序时，我总是会加入一些打印语句，而在程序调试完毕之后再做修改或者干脆注释掉。

好了，一切就绪，现在可以运行这个程序。首先我们启动服务器：

```
1> udp_test:start_server().
server opened socket:#Port<0.106>
<0.34.0>
```

它会在后台运行，我们可以发起一个客户端调用：

```
2> udp_test:client(40).
client opened socket=#Port<0.105>
server received:{udp,#Port<0.106>,{127,0,0,1},32785,<<131,97,40>>}
client received:{udp,#Port<0.105>,
                 {127,0,0,1}, 4000,
                 <<131,110,20,0,0,0,0,0,0,64,37,5,255,
```

```
                    100,222,15,8,126,242,199,132,27,
                    232,234,142>>}
81591528324789773434561126959611589427200000000
```

14.5.3 关于 UDP 协议的其他注意事项

我们应该注意到UDP是一个无连接的协议，因此服务器也就没有机会阻塞客户端，即拒绝从客户端接收数据——因为服务器无从得知谁是客户端。 256

大型的UDP数据报通过网络传输时，它们可能会被切分成多个帧。UDP数据报在网络的传输过程中，如果UDP数据报大小超过它所经过路由器设定的最大传输单元（MTU）限制时，就会出现分帧的情形。在优化UDP的网络应用时，最为常见的解决方案就是从一个比较小的数据报长度开始（比如说，500字节左右），然后一边测试吞吐量一边逐渐地增大数据报的长度。如果在某一时刻吞吐量明显下降，那么你就知道此时的数据报可能是过大了。

一个UDP数据报可能被传输两次（有些人会被这种特性搞得手足无措），因此你必须很小心的编写远程过程调用的代码。有的时候可能会发生这样的情形，在你回应第二条消息的时候，实际上只是回应了第一条消息的复制品。为了避免这个问题，我们应该修改客户端的代码，为其加入一个唯一的引用（reference），然后在服务器返回时去校验这个引用。为了保证引用的唯一性，我们可以调用Erlang的BIF `make_ref`，这个函数保证会返回一个全球唯一的引用。增加这些逻辑之后，这个远程过程调用的代码看起来会是这样的：

```
client(Request) ->
    {ok, Socket} = gen_udp:open(0, [binary]),
    Ref = make_ref(), %% make a unique reference
    B1 = term_to_binary({Ref, Request}),
    ok = gen_udp:send(Socket, "localhost", 4000, B1),
    wait_for_ref(Socket, Ref).

wait_for_ref(Socket, Ref) ->
    receive
        {udp, Socket, _, _, Bin} ->
            case binary_to_term(Bin) of
                {Ref, Val} ->
                    %% got the correct value
                    Val;
                {_SomeOtherRef, _} ->
                    %% some other value throw it away
                    wait_for_ref(Socket, Ref)
            end;
    after 1000 ->
        ...
    end.
```

14.6 向多台机器广播消息

为了完整地介绍UDP，我还会向你介绍要如何来设置一个广播通道。使用这种代码的情况非常少，但也许有一天你会需要用到它们。 257

broadcast.erl

```
-module(broadcast).
-compile(export_all).

send(IoList) ->
    case inet:ifget("eth0", [broadaddr]) of
        {ok, [{broadaddr, Ip}]} ->
            {ok, S} =  gen_udp:open(5010, [{broadcast, true}]),
            gen_udp:send(S, Ip, 6000, IoList),
            gen_udp:close(S);
        _ ->
            io:format("Bad interface name, or\n"
                      "broadcasting not supported\n")
    end.

listen() ->
    {ok, S} = gen_udp:open(6000),
    loop(S).

loop(S) ->
    receive
        Any ->
            io:format("received:~p~n", [Any]),
            loop(S)
    end.
```

这里我们需要用到两个端口，一个是发送广播的端口，另一个则是接收广播消息的端口。我们选择了端口5010来发送广播数据，端口6000则用来监听广播的数据（这两个具体的数值是随便选取的，选取这两个值并没有什么特别的原因，我只是在自己的系统上找了两个尚未使用的端口）。

只发送广播包的进程需要打开5010端口，而该网络上的所有其他机器都调用`broadcast:listen()`，该函数会打开机器上的6000端口，然后监听广播消息。

`broadcast:send(IoList)`函数把数据`IoList`向本地网络上的所有机器进行广播发送。

说明　要让这段代码可以工作，网卡的名字必须拼写正确，而且必须支持广播。例如在我的iMac机器上，我需要使用名字“en0”来替代“eth0”。另外一个需要注意的地方是，如果运行UDP监听进程的机器和发送UDP广播的机器分别处于不同子网之上，那么它也不可能接收到UDP广播，因为路由器会默认屏蔽掉UDP广播包。

258

14.7　SHOUTcast服务器

作为本章的结束，我们要使用这一章刚学到的套接字编程技术来编写一个SHOUTcast服务器。SHOUTcast是一种由Nullsoft公司的程序员开发的协议，主要用于传输音频数据流。[①]SHOUTcast使用HTTP协议来发送MP3或者AAC编码的音频数据。

① http://www.shoutcast.com/。

要了解它的工作原理，我们先要来看看SHOUTcast协议本身，然后对整个服务器的结构做一个概览，最后以整个服务器的代码来结束本章。

14.7.1 SHOUTcast 协议

SHOUTcast协议很简单。

(1) 首先是客户机（可能是类似XMMS、Winamp或者iTunes之类的软件）向SHOUTcast服务器发送一个HTTP请求。下面的就是当我在家运行自己的SHOUTcast服务器，由XMMS发出的请求：

```
GET / HTTP/1.1
Host: localhost
User-Agent: xmms/1.2.10
Icy-MetaData:1
```

(2) 我的SHOUTcast服务器对这个请求回应了下面这条消息：

```
ICY 200 OK
icy-notice1: <BR>This stream requires
  <a href=http://www.winamp.com/>;Winamp</a><BR>
icy-notice2: Erlang Shoutcast server<BR>
icy-name: Erlang mix
icy-genre: Pop Top 40 Dance Rock
icy-url: http://localhost:3000
content-type: audio/mpeg
icy-pub: 1
icy-metaint: 24576
icy-br: 96
... data ...
```

(3) 然后SHOUTcast服务器会开始发送一个连续的数据流，这个数据具有下面这样的结构：

```
F H F H F H F ...
```

259

F是一个MP3音频数据块，这个数据块的长度固定为24 576字节长（这个值由icy-metaint参数给出）。H是一个头数据块。头数据块包含了一个单字节K后面跟着16*K字节的数据。因此，最小的头块以二进制数据表示就是<<0>>。下一个头块就应该是这样：

```
<<1,B1,B2,…,B16>>
```

这个头块的数据内容是一个形如StreamTitle=' ...';StreamUrl='http:// ...';这样的字符串，程序会用0来补足整个数据块后面长度不足的部分。

14.7.2 SHOUTcast 服务器的工作机制

要编写这个服务器，我们必须要注意下面这些细节。

(1) 产生一个播放列表。我们的服务器使用一个文件来保存在13.2节中的代码生成的歌曲列表。音频文件则从这个列表之中随机选取。

(2) 生成一个并行服务器，以便可以并行地提供数路音频数据流。我们用14.1节中展示的技术来实现这个服务器。

(3) 对每一个音频文件，我们希望只向客户端发送音频数据而不发送内嵌的ID3标记。[①]

要移除这些标记，我们可以使用id3_tag_lengths之中的代码，这段代码使用了13.2节和5.3节中开发的代码，这儿就不显示这些代码了。

14.7.3 SHOUTcast 服务器的伪代码

在我们涉及最终的程序代码之前，先来看看省去若干细节之后的代码主干：

```
start_parallel_server(Port) ->
   {ok, Listen} = gen_tcp:listen(Port, ..),
   %% create a song server -- this just knows about all our music
   PidSongServer = spawn(fun() -> songs() end),
260 spawn(fun() -> par_connect(Listen, PidSongServer) end).
 %% spawn one of these processes per connection
 par_connect(Listen, PidSongServer) ->
     {ok, Socket} = gen_tcp:accept(Listen),
     %% when accept returns spawn a new process to
     %% wait for the next connection
     spawn(fun() -> par_connect(Listen, PidSongServer) end),
     inet:setopts(Socket, [{packet,0},binary, {nodelay,true},
                           {active, true}]),
     %% deal with the request
     get_request(Socket, PidSongServer, []).

%% wait for the TCP request
get_request(Socket, PidSongServer, L) ->
     receive
        {tcp, Socket, Bin} ->
           ... Bin contains the request from the client
           ... if the request is fragmented we call loop again ...
           ... otherwise we call
           .... got_request(Data, Socket, PidSongServer)
        {tcp_closed, Socket} ->
           ... this happens if the client aborts
           ... before it has sent a request (very unlikely)
 end.

%% we got the request -- send a reply
 got_request(Data, Socket, PidSongServer) ->
      .. data is the request from the client ...
      .. analayse it ...
      .. we'll always allow the request ..
```

① 我不清楚这种策略是否正确。音频编码器也许可以跳过错误数据，因此，在原则上我们也可以不去掉ID3标记让它们随着音频数据一起发过去。但实际试验下来，似乎还是把ID3标记去掉之后的效果更好些。

```
        gen_tcp:send(Socket, [response()]),
        play_songs(Socket, PidSongServer).

%% play songs forever or until the client quits
play_songs(Socket, PidSongServer) ->
      ... PidSongServer keeps a list of all our MP3 files
      Song = rpc(PidSongServer, random_song),
      ... Song is a random song ...
      Header = make_header(Song),
      ... make the header ...
      {ok, S} = file:open(File, [read,binary,raw]),
      send_file(1, S, Header, 1, Socket),
      file:close(S),
      play_songs(Socket, PidSongServer).

send_file(K, S, Header, OffSet, Socket) ->
  ... send the file in chunks to the client ...
  ... returns when the entire file is sent ...
  ... but exits if we get an error when writing to
      the socket -- this happens if the client quits
```

261

查看实际的代码，你会发现虽然实现的细节略有差别，但原理是一致的。这里是完整的代码清单：

shout.erl

```
-module(shout).

%% In one window > shout:start()
%% in another window xmms http://localhost:3000/stream

-export([start/0]).
-import(lists, [map/2, reverse/1]).

-define(CHUNKSIZE, 24576).

start() ->
    spawn(fun() ->
                  start_parallel_server(3000),
                  %% now go to sleep - otherwise the
                  %% listening socket will be closed
                  lib_misc:sleep(infinity)
          end).

start_parallel_server(Port) ->
    {ok, Listen} = gen_tcp:listen(Port, [binary, {packet, 0},
                                         {reuseaddr, true},
                                         {active, true}]),
```

```
    PidSongServer = spawn(fun() -> songs() end),
    spawn(fun() -> par_connect(Listen, PidSongServer) end).

par_connect(Listen, PidSongServer) ->
    {ok, Socket} = gen_tcp:accept(Listen),
    spawn(fun() -> par_connect(Listen, PidSongServer) end),
    inet:setopts(Socket, [{packet,0},binary, {nodelay,true},{active, true}]),
    get_request(Socket, PidSongServer, []).

get_request(Socket, PidSongServer, L) ->
    receive
        {tcp, Socket, Bin} ->
            L1 = L ++ binary_to_list(Bin),
            %% split checks if the header is complete
            case split(L1, []) of
                more ->
                    %% the header is incomplete we need more data
                    get_request(Socket, PidSongServer, L1);
                {Request, _Rest} ->
                    %% header is complete
                    got_request_from_client(Request, Socket, PidSongServer)
            end;
        {tcp_closed, Socket} ->
262         void;
        _Any  ->
            %% skip this
            get_request(Socket, PidSongServer, L)
    end.

split("\r\n\r\n" ++ T, L) -> {reverse(L), T};
split([H|T], L)           -> split(T, [H|L]);
split([], _)              -> more.

got_request_from_client(Request, Socket, PidSongServer) ->
    Cmds = string:tokens(Request, "\r\n"),
    Cmds1 = map(fun(I) -> string:tokens(I, " ") end, Cmds),
    is_request_for_stream(Cmds1),
    gen_tcp:send(Socket, [response()]),
    play_songs(Socket, PidSongServer, <<>>).


play_songs(Socket, PidSongServer, SoFar) ->
    Song = rpc(PidSongServer, random_song),
    {File,PrintStr,Header} = unpack_song_descriptor(Song),
    case id3_tag_lengths:file(File) of
        error ->
            play_songs(Socket, PidSongServer, SoFar);
        {Start, Stop} ->
            io:format("Playing:~p~n",[PrintStr]),
```

```
            {ok, S} = file:open(File, [read,binary,raw]),
            SoFar1 = send_file(S, {0,Header}, Start, Stop, Socket, SoFar),
            file:close(S),
            play_songs(Socket, PidSongServer, SoFar1)
    end.


send_file(S, Header, OffSet, Stop, Socket, SoFar) ->
    %% OffSet = first byte to play
    %% Stop   = The last byte we can play
    Need = ?CHUNKSIZE - size(SoFar),
    Last = OffSet + Need,
    if
        Last >= Stop ->
            %% not enough data so read as much as possible and return
            Max = Stop - OffSet,
            {ok, Bin} = file:pread(S, OffSet, Max),
            list_to_binary([SoFar, Bin]);
        true ->
            {ok, Bin} = file:pread(S, OffSet, Need),
            write_data(Socket, SoFar, Bin, Header),
            send_file(S, bump(Header),
                      OffSet + Need,  Stop, Socket, <<>>)
    end.                                                          263
write_data(Socket, B0, B1, Header) ->
    %% Check that we really have got a block of the right size
    %% this is a very useful check that our program logic is
    %% correct
    case size(B0) + size(B1) of
        ?CHUNKSIZE ->
            case gen_tcp:send(Socket, [B0, B1, the_header(Header)]) of
                ok -> true;
                {error, closed} ->
                    %% this happens if the player
                    %% terminates the connection
                    exit(playerClosed)
            end;
        _Other ->
            %% don't send the block - report an error
            io:format("Block length Error: B0 = ~p b1=~p~n",
                      [size(B0), size(B1)])
    end.

bump({K, H})     -> {K+1, H}.

the_header({K, H}) ->
    case K rem 5 of
        0 -> H;
        _ -> <<0>>
```

```
    end.


is_request_for_stream(_) -> true.


response() ->
    ["ICY 200 OK\r\n",
     "icy-notice1: <BR>This stream requires",
     "<a href=\"http://www.winamp.com/\">Winamp</a><BR>\r\n",
     "icy-notice2: Erlang Shoutcast server<BR>\r\n",
     "icy-name: Erlang mix\r\n",
     "icy-genre: Pop Top 40 Dance Rock\r\n",
     "icy-url: http://localhost:3000\r\n",
     "content-type: audio/mpeg\r\n",
     "icy-pub: 1\r\n",
     "icy-metaint: ",integer_to_list(?CHUNKSIZE),"\r\n",
     "icy-br: 96\r\n\r\n"].


songs() ->
    {ok,[SongList]} = file:consult("mp3data"),
    lib_misc:random_seed(),
    songs_loop(SongList).
songs_loop(SongList) ->
    receive
        {From, random_song} ->
            I = random:uniform(length(SongList)),
            Song = lists:nth(I, SongList),
            From ! {self(), Song},
            songs_loop(SongList)
    end.

rpc(Pid, Q) ->
    Pid ! {self(), Q},
    receive
        {Pid, Reply} ->
            Reply
    end.

unpack_song_descriptor({File, {_Tag,Info}}) ->
    PrintStr = list_to_binary(make_header1(Info)),
    L1 = ["StreamTitle='",PrintStr,
          "';StreamUrl='http://localhost:3000';"],
    %% io:format("L1=~p~n",[L1]),
    Bin = list_to_binary(L1),
    Nblocks = ((size(Bin) - 1) div 16) + 1,
    NPad = Nblocks*16 - size(Bin),
    Extra = lists:duplicate(NPad, 0),
```

264

```
    Header = list_to_binary([Nblocks, Bin, Extra]),
    %% Header is the Shoutcast header
    {File, PrintStr, Header}.

make_header1([{track,_}|T]) ->
    make_header1(T);
make_header1([{Tag,X}|T]) ->
    [atom_to_list(Tag),": ",X," "|make_header1(T)];
make_header1([]) ->
    [].
```

14.7.4 运行 SHOUTcast 服务器

要运行并测试服务器，我们需要完成下面3个步骤。

(1) 创建一个播放列表。

(2) 启动服务器。

(3) 把一个客户端连接到服务器。

1. 创建播放列表

要创建播放列表，需要按照下面几个步骤来执行程序。

(1) 进入代码目录。

265

(2) 编辑mp3_manager.erl文件中函数start1中的路径，把它指定到含有你要播放的音频文件的根目录。

(3) 编译mp3_manager然后输入命令mp3_manager:start1()，此时你应该会看到下面这些输出：

```
1> c(mp3_manager).
{ok,mp3_manager}
2> mp3_manager:start1().
Dumping term to mp3data
ok
```

如果有兴趣，你现在就可以去文件mp3data中查看一下分析结果。

2. 启动SHOUTcast服务器

你可以输入下面命令来启动服务器：

```
1> shout:start().
...
```

3. 测试服务器

(1) 切换到另外一个窗口来启动一个音频播放器，然后把它指向名为http://localhost:3000的音频流。

在我的系统中我使用XMMS然后输入下面这条命令：

```
xmms http://localhost:3000
```

说明 如果你希望访问运行在另外一台机器上的服务器，那么必须给定该服务器的IP地址。例

如，要在Windows机器上使用Winamp，我会选择Winamp中的Play→URL菜单，然后在URL对话框中输入http://192.168.1.168:3000。

而要在我的IMac机器上使用ITunes，则需要选择Advanced→OpenStream菜单，然后输入前面的这串URL来访问服务器。

(2) 你会在运行服务器的窗口中看到一些诊断输出信息。

(3) 好好享受自己的劳动成果吧！

14.8　进一步深入

在本章中，我们学习到的只是操纵套接字的最常用函数。你可以在gen_tcp、gen_udp、inet
266 的用户手册中发掘出更多有用的信息。

第15章

ETS和DETS：大量数据的存储机制[①]

ets和dets是Erlang用于高效存储大量Erlang数据条目的两个系统模块。ETS是Erlang Term Storage的缩写，而DETS是disk ETS的缩写。

ETS和DETS基本上是在做同一件事：它们提供大型的“键-值”搜索表。所不同的是，ETS驻留在内存，而DETS驻留在磁盘。ETS非常高效，在ETS中，无论你存储多少数据，查询速度都与之无关（在某些特定情况下，与之呈对数相关），当然了，这里的前提是你要拥有足够大的内存。DETS和ETS提供的接口几乎一样，所不同的只是它会将数据存储在磁盘上。正是因为这样，DETS要比ETS慢得多，而这么做的回报是，它也比ETS更加节省内存。此外，ETS和DETS的表可以被多个进程共享，这意味着可以通过这两个模块实现进程间高效的数据交换。

ETS和DETS表的数据结构是“键-值”对。我们在表上最常用的操作是插入和查找。一个ETS或DETS的表其实就是一系列Erlang元组。

ETS中的数据存储是临时的，这也就意味着当ETS表被释放的时候，相应的数据也会全部被丢弃。而DETS则不然，DETS中的数据存储是持久的，它们写在磁盘上，就算整个系统崩溃，它们也会毫发无损。只是当我们再次打开这个DETS表时，会首先检查数据的一致性。如果发现数据损坏，会试图进行修复（这会是一个耗时的过程，因为修复过程会遍历整个表，检查所有数据）。

这个过程一般都可以恢复表中的所有数据，但如果最后的一笔数据，恰好是在系统崩溃的那个时刻创建的话，那么它可能会丢失。

ETS表在那些需要高效操纵海量数据的场合都可以大显身手。而你也已经了解Erlang语言有“非破坏性赋值”的特性。当“非破坏性赋值”和“纯Erlang数据结构”使得编程变得难以进行时，ETS表也是一种不错的替代方案。

ETS表看起来像是用Erlang本身实现的，其实不然，它实际上是由底层的运行时系统实现的。相比一般的Erlang对象，它的行为很不相同。比如说，ETS表不会被垃圾回收，这意味着，在使

① Erlang目前版本（到R12b为止）的ETS和DETS在32bit的机器上有4 G数据条目数的限制，在现有的技术背景下，这个数量级的存储能力不足以用“海量”来形容，因而，这里使用了别扭但更准确“大量”这一形容词。

——译者注

用极大的ETS表时，也无须顾虑垃圾回收造成的影响。但凡事都有两面，相比一般的Erlang对象，ETS对象的创建和访问性能则会有一些轻微的降低。

15.1　表的基本操作

ETS和DETS的表有如下4种基本操作。

- 创建一个新表或打开一个已经存在的表。我们可以用ets:new或dets:open_file来进行这种操作。
- 将一个或者多个元组插入表。这时我们可以用insert(Tablename, X)，这里X可以是一个元组或者元组的列表。在ETS和DETS中，insert的参数和行为都一样。
- 在表中查找元组。我们可以用lookup(Tablename, Key)，其返回结果是匹配Key的元组列表。ETS和DETS都定义了lookup函数。（这里为什么会返回元组的列表呢？如果表的类型是bag，那么在这个表中，可以有多个元组使用相同的键。下一节我们会关注表的类型。）如果给出的键没有与之匹配的元组，则会返回一个空的列表。
- 释放表。用完一个表，可以调用dets:close(TableId)或ets:delete(TableId)来告诉系统释放这个表。

268

15.2　表的类型

ETS和DETS表存储元组。元组中的一个元素（默认为第一个元素）称作表的键。当我们向表中插入元组或者从中取出元组的时候，正是基于键来进行操作的。在向表中插入元组时，其行为是由两个因素所决定的：表的类型和这个键的值。类型为set的表，要求所有元组的键值各不相同，而类型为bags的表，则允许多个元组有相同的键值。

根据应用程序的行为，选择正确的表类型，这至关重要。

set和bag两种表类型各自都有两个变种，总共有4种类型：set、ordered set、bag和duplicate bag。

在set类型下，表中每一个元组的键值都不能相同。在ordered set下，元组会进行排序。在bag类型下，多个元组可以有相同的键值，但不能有两个完全相同的元组。在duplicate bag类型下，不仅多个元组可以有相同的键值，同一个元组也可以在表中出现多次。

我们可以用下面的小测试程序来演示它们是怎样工作的。

ets_test.erl

```
-module(ets_test).
-export([start/0]).

start() ->
    lists:foreach(fun test_ets/1,
                  [set, ordered_set, bag, duplicate_bag]).

test_ets(Mode) ->
    TableId = ets:new(test, [Mode]),
```

```
    ets:insert(TableId, {a,1}),
    ets:insert(TableId, {b,2}),
    ets:insert(TableId, {a,1}),
    ets:insert(TableId, {a,3}),
    List = ets:tab2list(TableId),
    io:format("~-13w  => ~p~n", [Mode, List]),
    ets:delete(TableId).
```

这个程序依次以上述4种类型中的一种打开一个ETS表，向它插入元组{a,1}、{b,2}、{a,1}以及{a,3}，然后调用tab2list，这个函数会将整个表转换为列表，并打印出来。 269

运行的时候，我们得到这样的输出：

```
1> ets_test:start().
set             => [{b,2},{a,3}]
ordered_set     => [{a,3},{b,2}]
bag             => [{b,2},{a,1},{a,3}]
duplicate_bag   => [{b,2},{a,1},{a,1},{a,3}]
```

对于set类型的表，每个键值只允许出现一次。比如，先插入{a,1}再插入{a,3}，那么最终的结果会是{a,3}。set和ordered set的唯一区别是ordered set中的成员是按照键值来排序的。在上面的例子中，我们从tab2list的输出结果中可以观察到这一点。

同一个键值在bag类型的表中可以出现多次。比如，先插入{a,1}再插入{a,3}，最终结果会包括这两个元组，而不是像set一样，只有最后一个；而如果先插入{a,1}再插入{a,1}，标准的bag表只会包含一个{a,1}，因为两次插入的{a,1}是相同的；而在duplicate bag表中则会同时包含两个{a,1}。

15.3 ETS 表的效率考虑

在内部，ETS表是用散列表来表示的（除了ordered set，它是用平衡二叉树来表示的）。这意味着，使用set类型有点空间浪费，而ordered set则有点时间浪费。向一个set表插入数据所耗费的时间是一个常量，而向一个ordered set插入数据所耗费的时间则与表的数据量有对数相关性。

在set和ordered set之间选择的时候，你需要考虑在它建立之后，你想用它来做什么——如果你需要一个有序的表，那就使用ordered set好了。

相比之下bag比duplicate bag的使用代价还高，因为每次插入一个数据，都要比较是否存在同样的键值。如果有大量的元组都有相同的键值，将会非常低效。

ETS表与正常的进程存储空间是分离开来的，其存储区域与普通进程无关。一个ETS表隶属于创建它的进程——当这个进程死掉或者调用了ets:delete，这个表就被删掉了。ETS表不会进行垃圾回收，这就意味着我们可以放心地把大量数据放在其中，无须担心垃圾回收的影响。 270

当我们向一个ETS表插入元组的时候，所有表示这个元组的数据结构会从进程的栈（或堆）中复制到ETS表。对ETS表进行查找操作时，结果元组又会从ETS表复制到进程的栈（或堆）中。

除了大的二进制数据外，其他数据结构都是这么运作的。大的二进制数据存储在它自己的存储空间中。这个空间可以被多个进程和ETS表共享，系统中有一个采用引用计数的方式垃圾回收

器来管理和跟踪引用某个二进制数据的进程或ETS数目。当某个特定二进制数据的引用计数降到0的时候，这块存储区域就可以回收。

这些听起来似乎很复杂，但从总体上说，在进程之间发送包含大量二进制数据的消息，或者向ETS表插入包含二进制数据的元组，代价都很低。因此，尽可能地用二进制数据来表示字符串或大块的无类型内存是一种高效的编程模式。

15.4　创建 ETS 表

调用ets:new可以创建ETS表。创建表的这个进程就是表的所有者。表创建之后，它的一系列属性设置不能再更改。如果所有者进程死掉，或者调用ets:delete时，就会自动释放表的内存空间。

ets:new的参数是下面这样的。

```
@spec ets:new(Name, [Opt]) -> TableId
```

Name是一个原子，[Opt]是一个选项列表，取值范围如下。

- set | ordered_set | bag | duplicate_bag。以某种特定类型创建一个表（我们之前已经提过了这些类型）。
- private。创建私有表，只有所有者进程可以读写这个表。

271

ETS表就像一个黑板

受保护的表提供了某种“黑板系统”[①]。你可以把受保护的ETS表当作一种有名字的黑板。任何人，只要他知道这个黑板的名字，就可以阅读这个黑板——但只有这个黑板的所有者可以往上面写东西。

说明　以公开方式打开的ETS表可以被任何一个知道它名字的进程读写。这种情况下，使用者必须确保对这个表的读写操作是一致的。

- public。创建公开表，所有知道这个表标识的进程都可以对这个表进行读写操作。
- protected。创建受保护的表，所有知道这个表标识的进程都可以对这个表进行读操作，但只有这个表的所有者进程可以对这个表进行写操作。
- named_table。命名表，如果存在这个选项，则可以在后续操作中使用Name来操作表。
- {keypos, K}。使用K作为键的位置，通常情况下使用的是第一个位置。可能只有一种情况才需要用到这个选项，那就是当我们需要存储Erlang的记录时（记录实际上是变相的元组），它的第一元素包含的是这个记录的名字（对每个记录来说，它的值都

① 黑板系统是微软公司参与开发的、流行于全世界的一种在线学习管理系统软件，可以认为是某种CMS，在概念上与wiki或blog有某种程度的类似。——译者注

是一样的)。

说明 打开一个ETS表时不带任何选项，等同于使用了这样的默认选项[set,protected,{keypos,1}]。

本章的所有代码使用的都是受保护的ETS表。所有知道表标识的进程都读取数据，但只有一个进程可以写数据，利用这种类型的表来共享数据，其成本接近于零，因而非常有用。 272

15.5 ETS 程序示例

本节的例子用来生成文字索引（这里提到的是trigram[①]）。这是一个漂亮的“炫技”程序，可以展现ETS表的强大功能。

我们的目标是建立一个启发式的程序，它试图预测一个给定的字符串是不是一个英语单词。在20.4节，我们有一个例子是建立在全文检索技术之上的搜索引擎，在那里我们会再次用到这个程序。

怎样才能预测一个随机产生的字母序列是不是英语单词？使用三字索引是方法之一，三字索引是由3个字母组成的一个序列（由更多个字母组成的序列称作n-gram）。就目前来说，并非所有的3个字母的组合都是合法的英语单词，比如说，英语当中并不存在akj或者rwb这样的词。这样的话，如果要测试一个字符串是不是英语单词，我们要做的就是，先由大量的英语单词生成三字索引当作基准，然后，对字符串中所有的连续3字符序列进行对比。

我们的程序要做的第一件事就是从大量的英语单词中计算所有出现的三字索引。在对set表、ordered set表以及由sets模块提供的“纯Erlang”的集合进行性能评估之后，我们决定使用ETS的set来完成这项工作。

我们在下面几节中要做的事情如下。

(1) 做一个迭代器，它遍历英语中的所有三字索引。这非常简单，只需要写一段向不同类型的表插入三字索引的代码就行了。

(2) 创建set和ordered set类型的ETS表，用它们来表示所有的三字索引。此外，还需要建立一个set来包含所有这些索引。

(3) 评估创建这3种不同的表所花费的时间。

(4) 评估访问这3种不同的表所花费的时间。

(5) 在这些评估的基础上，筛选出最优的方法，并以此为基准来编写代码。

这些代码都在lib_trigrams中。我们下面的章节会展示这些代码，这里会跳过部分细节，不过不用担心，这一章的结尾有完整的代码。这就是我们的计划，马上开始。 273

① trigram即三字索引，文字索引中的一种，这是搜索领域的概念。文字索引主要用于分词，搜索引擎依赖文字索引对文本资料进行分词，进而可以建立关键词索引，并提供搜索功能。——译者注

15.5.1　三字索引迭代器

我们会定义一个函数，叫做for_each_trigram_in_the_english_language(F, A)。这个函数会以每一个英语的三字索引作为参数对函数F求值。F是一个fun(Str, A) -> A类型的函数，Str的取值范围是语言中的所有三字索引，A是一个累加器。

写出这个迭代器[①]，需要大量的单词。我使用了一个包含354 984个英文单词的集合[②]来生成三字索引。根据这个单词列表，我们可以这样来定义迭代器：

lib_trigrams.erl

```
for_each_trigram_in_the_english_language(F, A0) ->
    {ok, Bin0} = file:read_file("354984si.ngl.gz"),
    Bin = zlib:gunzip(Bin0),
    scan_word_list(binary_to_list(Bin), F, A0).

scan_word_list([], _, A) ->
    A;
scan_word_list(L, F, A) ->
    {Word, L1} = get_next_word(L, []),
    A1 = scan_trigrams([$\s|Word], F, A),
    scan_word_list(L1, F, A1).

%% scan the word looking for \r\n
%% the second argument is the word (reversed) so it
%% has to be reversed when we find \r\n or run out of characters

get_next_word([$\r,$\n|T], L) -> {reverse([$\s|L]), T};
get_next_word([H|T], L)       -> get_next_word(T, [H|L]);
get_next_word([], L)          -> {reverse([$\s|L]), []}.

scan_trigrams([X,Y,Z], F, A) ->
    F([X,Y,Z], A);
scan_trigrams([X,Y,Z|T], F, A) ->
    A1 = F([X,Y,Z], A),
    scan_trigrams([Y,Z|T], F, A1);
scan_trigrams(_, _, A) ->
    A.
```

这里有两点值得注意。第一，我们使用zlib:gunzip(Bin)来解压原文件中的二进制数据。这个列表太长了，我宁愿在磁盘上保留这个文件的压缩文件，而不是原始的ASCII文件。第二，我们在每个单词的头尾都加了一个空格，在三字索引分析过程中，我们把空格当作一个标准的字符。

274

① 这里把它称作迭代器，严格意义上说，它实际上是一个fold（折叠）操作，就像lists:foldl那样。

② 这个字符列表来自于http://www.dcs.shef.ac.uk/research/ilash/Moby/。

15.5.2 构造表

我们像这样来构造这个ETS表：

lib_trigrams.erl

```
make_ets_ordered_set() -> make_a_set(ordered_set, "trigramsOS.tab").
make_ets_set()         -> make_a_set(set, "trigramsS.tab").

make_a_set(Type, FileName) ->
    Tab = ets:new(table, [Type]),
    F = fun(Str, _) -> ets:insert(Tab, {list_to_binary(Str)}) end,
    for_each_trigram_in_the_english_language(F, 0),
    ets:tab2file(Tab, FileName),
    Size = ets:info(Tab, size),
    ets:delete(Tab),
    Size.
```

值得注意的是表示三字索引的方式，每个三字索引由3个字母组成，比如ABC，在向ETS表插入的时候，实际上是用一个{<<"ABC">>}这样的元组来表示这个三字索引。是的，只有一个元素的元组，这看起来是有点诡异，我们都知道元组是用来容纳多个元素的容器，只有一个元素的元组显然是没什么意义的。那么，这里单元素的元组，又是什么意思呢？回忆一下，ETS表要求它的每个记录都是元组，并默认元组的第一个元素作为元组的键。在这里，元组{key}实际上就是一个没有值的键。

接下来是构造容纳所有三字索引的集合（这里使用的是Erlang的模块sets，而不是ETS）：

lib_trigrams.erl

```
make_mod_set() ->
    D = sets:new(),
    F = fun(Str, Set) -> sets:add_element(list_to_binary(Str),Set) end,
    D1 = for_each_trigram_in_the_english_language(F, D),
    file:write_file("trigrams.set", [term_to_binary(D1)]).
```

15.5.3 构造表有多快

本章末尾列出的lib_trigrams:make_tables()函数负责构造所有的表。它包含了一些测试代码，我们可以得到表的大小和构造它所花费的时间。

```
1> lib_trigrams:make_tables().
Counting - No of trigrams=3357707 time/trigram=0.577938
Ets ordered Set size=19.0200 time/trigram=2.98026
Ets set size=19.0193 time/trigram=1.53711
Module Set size=9.43407 time/trigram=9.32234
ok
```

275

首先，这个输出告诉我们的是：单词列表中总共有3 300 000个三字索引，平均下来，处理每个三字索引花费的时间是0.5μs。

在ordered set类型的ETS表中，平均插入一个三字索引要花费2.9μs，这个数值在set类型的ETS表上是1.5μs，而对应的，在Erlang的集合模块上花费的时间是9.3μs。在存储上，每个三字索引在

set和ordered set的ETS表中占19字节，在Erlang集合中占9字节。

15.5.4　访问表有多快

构造表要花一些时间，但这并不是我们这个例子最关键的问题。最关键的问题是，访问表有多快？要回答这个问题，先要写一段代码来评估访问时间。我们会把表中的每个三字索引都查找一次，然后得到每次查找的平均时间。下面是代码：

lib_trigrams.erl

```
timer_tests() ->
    time_lookup_ets_set("Ets ordered Set", "trigramsOS.tab"),
    time_lookup_ets_set("Ets set", "trigramsS.tab"),
    time_lookup_module_sets().

time_lookup_ets_set(Type, File) ->
    {ok, Tab} = ets:file2tab(File),
    L = ets:tab2list(Tab),
    Size = length(L),
    {M, _} = timer:tc(?MODULE, lookup_all_ets, [Tab, L]),
    io:format("~s lookup=~p micro seconds~n",[Type, M/Size]),
    ets:delete(Tab).

lookup_all_ets(Tab, L) ->
    lists:foreach(fun({K}) -> ets:lookup(Tab, K) end, L).

time_lookup_module_sets() ->
    {ok, Bin} = file:read_file("trigrams.set"),
    Set = binary_to_term(Bin),
    Keys = sets:to_list(Set),
    Size = length(Keys),
    {M, _} = timer:tc(?MODULE, lookup_all_set, [Set, Keys]),
    io:format("Module set lookup=~p micro seconds~n",[M/Size]).

lookup_all_set(Set, L) ->
    lists:foreach(fun(Key) -> sets:is_element(Key, Set) end, L).
```

运行的结果是：

```
1> lib_trigrams:timer_tests().
Ets ordered Set lookup=1.79964 micro seconds
Ets set lookup=0.719279 micro seconds
Module sets lookup=1.35268 micro seconds
ok
```

276

这里统计的是每次查找耗费的平均微秒数。

15.5.5　胜出的是……

毫无疑问，set类型的ETS表明显胜出。在我的机器上，在set类型的ETS表上每次查找只

要0.5 μs，性能很出色。

说明 像我们前面做的那样，写代码对某个操作的性能进行实际评估，其实是很好的编程习惯。我们当然没有必要极端到对每件事都进行这样的评估，但是很显然，对程序中最为耗时的操作进行评估是很有必要的。对那些不那么耗时的操作，应该是怎么写着清爽就怎么去写，务必追求优雅。如果是因为效率而不得不用一些难看方式来写代码，那么，至少我们需要有充足的注释（或文档），告诉其他人我们这么做的理由。

剩下的事情就好办了，现在我们可以着手编写一个程序来判断一个字符串是不是英文单词。

要检查一个字符串是不是一个英文单词，我们可以把这个字符串中的所有三字索引与到前面得到的三字索引表进行比对，看看它们是否出现。**is_word**函数是这样的。

lib_trigrams.erl

```
is_word(Tab, Str) -> is_word1(Tab, "\s" ++ Str ++ "\s").

is_word1(Tab, [_,_,_]=X) -> is_this_a_trigram(Tab, X);
is_word1(Tab, [A,B,C|D]) ->
    case is_this_a_trigram(Tab, [A,B,C]) of
        true  -> is_word1(Tab, [B,C|D]);
        false -> false
    end;
is_word1(_, _) ->
    false.

is_this_a_trigram(Tab, X) ->
    case ets:lookup(Tab, list_to_binary(X)) of
        [] -> false;
        _  -> true
    end.

open() ->
    {ok, I} = ets:file2tab(filename:dirname(code:which(?MODULE))
                           ++ "/trigramsS.tab"),
    I.

close(Tab) -> ets:delete(Tab).
```

277

open和**close**函数负责打开和关闭之前我们算出的ETS表，且任何对**is_word**函数的调用必须在这两个方法的包围之中。

在定位包含三字索引表的外部文件的位置时，我们用了一个小技巧。这个文件和当前模块的代码放在了同一个目录之中，**code:which(?MODULE)**返回的是当前模块的目标文件名。这样我们就很容易找到同一个目录下的三字索引文件了。

15.6 DETS

DETS将Erlang元组存在磁盘上，文件大小不能超过2G。使用之前要先打开DETS文件，用完后要妥善关闭。如果没有关闭，那么下次打开这个文件时，会自动进行修复。修复可能会耗费大量时间，所以，使用完后，请记得关闭DETS文件，这很重要。

DETS表和ETS表有不同的共享特性。打开一个DETS表时，要给出一个全局性的名字。如果两个或者多个本地进程用相同的名字和属性打开一个DETS表，那么它们会共享这个表。这个表会一直保持打开状态，直到所有的进程都关闭了这个表（或者这些进程死掉）。

例子：文件名索引

现在开始另外一个例子，这个例子也是全文检索引擎中还会用到的另一个工具，这是20.4节的内容。

我们需要在磁盘上建立一个映射表，将文件名映射为整数，反过来也是一样。我们需要定义函数filename2index以及它的反函数index2filename。

为此，我们要建立一个DETS表来实现这个功能。它处理这样3种元组。

```
{free, N}
```

N是表中第一个可用的索引值。当我们将一个新的文件名放到表中时，要把它的索引赋为此值。

```
{FileNameBin, K}
```

278

FileNameBin（二进制）索引值为K。

```
{K, FileNameBin}
```

K（整数）表示文件FileNameBin。

有一点值得注意，每次向表中增加一个新文件我们要增加两条记录，一个File↦Index记录和一个反向的记录Index↦Filename。这是从效率角度考虑的。在构造ETS或DETS表的时候，元组中只有一个元素作为键。虽说我们也能匹配非键的元素，但这是一个效率低下的过程，因为它要遍历整个表来搜索记录。尤其是当这整张表都在磁盘上时，毫无疑问会是一个代价非常高的操作。

闲话休提，我们直入主题，首先要做的是打开和关闭存储所有文件名的DETS表。

lib_filenames_dets.erl

```
-module(lib_filenames_dets).
-export([open/1, close/0, test/0, filename2index/1, index2filename/1]).

open(File) ->
    io:format("dets opened:~p~n", [File]),
    Bool = filelib:is_file(File),
    case dets:open_file(?MODULE, [{file, File}]) of
        {ok, ?MODULE} ->
            case Bool of
```

```
                true  -> void;
                false -> ok = dets:insert(?MODULE, {free,1})
            end,
            true;
        {error,_Reason} ->
            io:format("cannot open dets table~n"),
            exit(eDetsOpen)
    end.

close() -> dets:close(?MODULE).
```

open函数的代码会自动初始化DETS表，新建一个表的时候，它会插入一个{free, 1}这样的元组。当文件存在的时候，filelib:is_file(File)会返回true，否则会返回false。注意，dets:open_file既用来创建一个新文件，也用来打开一个已经存在的文件。这就是我们为什么要在调用dets:open_file之前先检查文件是否存在的原因。

这段代码中，大量使用了?MODULE这个宏，它会展开当前的模块名称（也就是lib_filenames_dets）。使用DETS时，很多方法都需要唯一的原子参数作为表名。而系统中不可能有两个同名的模块，用模块名作为表名，我们就可以轻易的获得这种唯一性。如果在系统中处处都遵循这种约定，我们就可以确保所有用到的表名是唯一的。 279

使用?MODULE而不是在每一处都明确的写出模块名称，这使得我可以继续保持更改模块名称的自由，而不会担心影响到代码的正常运行。

打开文件以后，向表中加入新的文件名就很简单了。这个过程是通过调用filename2index顺带完成的。如果表中已经有了这个文件名，那么就返回它的索引；如果没有，就生成一个新的索引，并更新表，这次有3个元组。

lib_filenames_dets.erl

```
filename2index(FileName) when is_binary(FileName) ->
    case dets:lookup(?MODULE, FileName) of
        [] ->
            [{_,Free}] = dets:lookup(?MODULE, free),
            ok = dets:insert(?MODULE,
                     [{Free,FileName},{FileName,Free},{free,Free+1}]),
            Free;
        [{_,N}] ->
            N
    end.
```

注意这个程序是如何一次存储3个元组的。dets:insert的第二个参数既可以是一个元组，也可以是元组的列表。另外一个值得注意的是文件名使用二进制数据来表达，这也是基于效率的考量。在使用ETS和DETS表时，最好习惯于使用二进制数据来表达字符串。

细心的读者可能会发现filename2index函数里有一个潜在的资源竞争。如果有两个并发的进程在调用dets:insert之前调用了dets:lookup，那么filename2index就会返回错误的值。因

此，在执行这个操作的时候，必须要保证每次只有一个进程进行调用。

从索引转为文件名更容易：

lib_filenames_dets.erl

```
index2filename(Index) when is_integer(Index) ->
    case dets:lookup(?MODULE, Index) of
        []        -> error;
        [{_,Bin}] -> Bin
    end.
```

这里有一个小的设计约定。如果我们调用索引index2filename（Index）没有对应的文件名，
280 这个时候怎么办呢？我们可以调用exit(ebadIndex)来退出，不过在这里我们选择了一个更平常的方式，我们返回原子类型的error。因为合法的文件名是二进制类型的，所以，调用者能轻易地区分出返回的是合法文件名还是异常情况。

另外一个需要留意的是filename2index和index2filename函数上的保护测试。这些测试保证我们得到是正确类型的参数。这些测试很有必要，如果向DETS表插入错误类型的数据可能导致非常难调试的情况。设想一下，在保存数据一个月以后再来读取，当我们发现数据类型不对时，无论做什么恐怕都已经为时太晚。所以，最好在数据加到表中之前检查数据是否正确。

15.7　我们没有提及的部分

ETS和DETS表还支持一些我们这一章没有提及的操作。这些操作大致可以归到下面这些类别中。

- 根据模式匹配来获取和删除对象。
- 在ETS和DETS之间转换，或者在ETS表和磁盘文件之间转换。
- 了解表的资源使用状况。
- 在表的所有记录中移动。
- 修复损坏的DETS表。
- 可视化表。

可以在ETS和DETS的在线手册中找到更多的资讯，通过 http://www.erlang.org/doc/man/ets.html和http://www.erlang.org/doc/man/dets.html可以访问。

作为本章的结尾，有必要提及Mnesia。ETS和DETS最初是设计用来实现Mnesia的，我们还没有谈到Mnesia，这是第17章的内容。Mnesia是一个用Erlang实现的实时数据库。Mnesia的内部使用了ETS和DETS，在ETS和DETS中的很多接口，就是为了便于在Mnesia的内部使用才开放的。Mnesia可以提供很多单靠ETS和DETS无法完成的功能。比如说，我们可以在非键值的元素上做索引。这样我们就没有必要使用前面例子中用到的那种两次插入技巧。实际上，Mnesia也使用多
281 个ETS或DETS来实现这些功能，但对于使用者来说，这些已经被很好的隐藏起来了。

15.8 代码清单

lib_trigrams_complete.erl

```
-module(lib_trigrams).
-export([for_each_trigram_in_the_english_language/2,
         make_tables/0, timer_tests/0,
         open/0, close/1, is_word/2,
         how_many_trigrams/0,
         make_ets_set/0, make_ets_ordered_set/0, make_mod_set/0,
         lookup_all_ets/2, lookup_all_set/2
        ]).
-import(lists, [reverse/1]).

make_tables() ->
    {Micro1, N} = timer:tc(?MODULE, how_many_trigrams, []),
    io:format("Counting - No of trigrams=~p time/trigram=~p~n",[N,Micro1/N]),
    {Micro2, Ntri} = timer:tc(?MODULE, make_ets_ordered_set, []),
    FileSize1 = filelib:file_size("trigramsOS.tab"),
    io:format("Ets ordered Set size=~p time/trigram=~p~n",[FileSize1/Ntri,
                                                           Micro2/N]),
    {Micro3, _} = timer:tc(?MODULE, make_ets_set, []),
    FileSize2 = filelib:file_size("trigramsS.tab"),
    io:format("Ets set size=~p time/trigram=~p~n",[FileSize2/Ntri, Micro3/N]),
    {Micro4, _} = timer:tc(?MODULE, make_mod_set, []),
    FileSize3 = filelib:file_size("trigrams.set"),
    io:format("Module sets size=~p time/trigram=~p~n",[FileSize3/Ntri, Micro4/N]).

make_ets_ordered_set() -> make_a_set(ordered_set, "trigramsOS.tab").
make_ets_set()         -> make_a_set(set, "trigramsS.tab").

make_a_set(Type, FileName) ->
    Tab = ets:new(table, [Type]),
    F = fun(Str, _) -> ets:insert(Tab, {list_to_binary(Str)}) end,
    for_each_trigram_in_the_english_language(F, 0),
    ets:tab2file(Tab, FileName),
    Size = ets:info(Tab, size),
    ets:delete(Tab),
    Size.

make_mod_set() ->
    D = sets:new(),
    F = fun(Str, Set) -> sets:add_element(list_to_binary(Str),Set) end,
    D1 = for_each_trigram_in_the_english_language(F, D),
    file:write_file("trigrams.set", [term_to_binary(D1)]).

timer_tests() ->
    time_lookup_ets_set("Ets ordered Set", "trigramsOS.tab"),
    time_lookup_ets_set("Ets set", "trigramsS.tab"),
    time_lookup_module_sets().
```

```
time_lookup_ets_set(Type, File) ->
    {ok, Tab} = ets:file2tab(File),
    L = ets:tab2list(Tab),
    Size = length(L),
    {M, _} = timer:tc(?MODULE, lookup_all_ets, [Tab, L]),
    io:format("~s lookup=~p micro seconds~n",[Type, M/Size]),
    ets:delete(Tab).

lookup_all_ets(Tab, L) ->
    lists:foreach(fun({K}) -> ets:lookup(Tab, K) end, L).

time_lookup_module_sets() ->
    {ok, Bin} = file:read_file("trigrams.set"),
    Set = binary_to_term(Bin),
    Keys = sets:to_list(Set),
    Size = length(Keys),
    {M, _} = timer:tc(?MODULE, lookup_all_set, [Set, Keys]),
    io:format("Module set lookup=~p micro seconds~n",[M/Size]).

lookup_all_set(Set, L) ->
    lists:foreach(fun(Key) -> sets:is_element(Key, Set) end, L).

how_many_trigrams() ->
    F = fun(_, N) -> 1 + N  end,
    for_each_trigram_in_the_english_language(F, 0).

%% An iterator that iterates through all trigrams in the language
for_each_trigram_in_the_english_language(F, A0) ->
    {ok, Bin0} = file:read_file("354984si.ngl.gz"),
    Bin = zlib:gunzip(Bin0),
    scan_word_list(binary_to_list(Bin), F, A0).

scan_word_list([], _, A) ->
    A;
scan_word_list(L, F, A) ->
    {Word, L1} = get_next_word(L, []),
    A1 = scan_trigrams([$\s|Word], F, A),
    scan_word_list(L1, F, A1).

%% scan the word looking for \r\n
%% the second argument is the word (reversed) so it
%% has to be reversed when we find \r\n or run out of characters

get_next_word([$\r,$\n|T], L) -> {reverse([$\s|L]), T};
get_next_word([H|T], L)       -> get_next_word(T, [H|L]);
get_next_word([], L)          -> {reverse([$\s|L]), []}.

scan_trigrams([X,Y,Z], F, A) ->
    F([X,Y,Z], A);
```

283

```
scan_trigrams([X,Y,Z|T], F, A) ->
    A1 = F([X,Y,Z], A),
    scan_trigrams([Y,Z|T], F, A1);
scan_trigrams(_, _, A) ->
    A.

%% access routines
%%   open() -> Table
%%   close(Table)
%%   is_word(Table, String) -> Bool

is_word(Tab, Str) -> is_word1(Tab, "\s" ++ Str ++ "\s").

is_word1(Tab, [_,_,_]=X) -> is_this_a_trigram(Tab, X);
is_word1(Tab, [A,B,C|D]) ->
    case is_this_a_trigram(Tab, [A,B,C]) of
        true  -> is_word1(Tab, [B,C|D]);
        false -> false
    end;
is_word1(_, _) ->
    false.

is_this_a_trigram(Tab, X) ->
    case ets:lookup(Tab, list_to_binary(X)) of
        [] -> false;
        _  -> true
    end.

open() ->
    {ok, I} = ets:file2tab(filename:dirname(code:which(?MODULE))
                           ++ "/trigramsS.tab"),
    I.

close(Tab) -> ets:delete(Tab).
```

284

第16章 OTP概述

OTP即Open Telecom Platform（开放电信平台）。不过，这实在是一个充满了误解的糟糕名字，实际上它远比这个名字听起来的要通用得多。它是一个应用程序操作系统，还包括大量库和程序用来构建大规模的分布式容错系统。OTP最初是由瑞典的Ericsson公司开发，它的设计目标就是用来构造容错系统。[①]

OTP包含了很多强大的工具，比如说，一个完备的Web服务器、FTP服务器、CORBA ORB等，它们全部都是用Erlang写成的。OTP还包含构建电信应用程序所需要的高级工具，比如，实现了H248、SNMP以及一个ASN.1到Erlang的交叉编译器。限于篇幅，这里就不准备对这些话题展开讨论了。有兴趣的读者可以参考附录C.1节中的链接。

用OTP来写程序，最为核心的概念是OTP中的行为（behavior）。一个行为封装了某种常见的行为模式，你也可以把这些行为理解为某种应用程序框架，可以通过回调模块来定制这些框架。

OTP的强大之处在于，它依靠行为引入了容错、扩容和动态代码升级等许多重要的特性。换句话说，在行为基础上编程，程序员不再需要考虑容错之类的事务，行为已经帮你搞定了这些问题，放心地写回调模块就行。以Java的世界作为对照，行为就好比是J2EE的容器。

简单来说，行为解决问题的非功能性部分，功能性的部分留给程序员自己写的回调模块来解决。这么做的好处是显而易见的——因为对于所有的系统来说，非功能性的部分都是一样的（比如，怎么做代码的在线升级，等等），而功能性的部分则五花八门，各有各的不同。

在本章中，我们要来仔细研究这些行为中的一种——gen_server模块。在一头扎进gen_server一大堆零零碎碎的问题之前，让我们先从一个简单的服务器程序开始（我实在想不出有什么比这更简单的服务器程序了）。每次一小步，对它进行持续的改进，直到将其打造成为一个具有完整特性的gen_server模块。相信通过这种学习方式，你终将能在底层的细枝末节中游刃有余，因为在这一过程中你已经真正领会到了gen_server的精微要义。

下面是本章的学习计划。

(1) 用Erlang写一个小的客户/服务器程序。

(2) 慢慢抽离出这个程序的通用部分，并增加一些特性。

---

① Ericsson已经在Erlang公共协议（EPL）下开源发布了OTP的源代码。EPL是与Mozilla公共协议（MPL）相似的开源协议。

(3) 进入实际的代码。

16.1 通用服务器程序的进化路线

本节是本书的精华之所在，如果读一遍不能理解，那就多读两遍，有必要的话，读100遍也在所不惜。不求倒背如流，但求运用自如。

我们下面会写4个小的服务器程序，它们以server1、server2、……这样的方式来命名。每一个都与前一个略有区别。整个过程的目标是要把问题的非功能性部分和功能性部分完全分离开来。目前，你可能还不能完全体会上面这句话的内涵，不过没关系，读完本节，你就会懂了。好了，让我们深吸一口气，准备开始吧。

16.1.1 server 1：原始服务器程序

这是我们最初的服务器程序，它可以通过回调模块来进行定制。

server1.erl

```
-module(server1).
-export([start/2, rpc/2]).

start(Name, Mod) ->
    register(Name, spawn(fun() -> loop(Name, Mod, Mod:init()) end)).
rpc(Name, Request) ->
    Name ! {self(), Request},
    receive
        {Name, Response} -> Response
    end.

loop(Name, Mod, State) ->
    receive
        {From, Request} ->
            {Response, State1} = Mod:handle(Request, State),
            From ! {Name, Response},
            loop(Name, Mod, State1)
    end.
```

286

这一小段代码准确地体现了一个服务器程序的精髓。接下来给server1写一个回调程序，比如，下面是一个名称服务器的回调：

name_server.erl

```
-module(name_server).
-export([init/0, add/2, whereis/1, handle/2]).
-import(server1, [rpc/2]).

%% client routines
add(Name, Place) -> rpc(name_server, {add, Name, Place}).
whereis(Name)    -> rpc(name_server, {whereis, Name}).
```

```
%% callback routines
init() -> dict:new().

handle({add, Name, Place}, Dict) -> {ok, dict:store(Name, Place, Dict)};
handle({whereis, Name}, Dict)    -> {dict:find(Name, Dict), Dict}.
```

这一段代码做了两件事。首先，它是服务器的回调程序，负责处理框架发过来的调用请求；其次，它定义了客户端调用的常规接口。将两者放在同一个模块中，在其他的语言看来或许会觉得有些怪异，但在OTP中，这是一种极常见的做法。①

要测试这个程序，我们可以这么做：

```
1> server1:start(name_server, name_server).
true
2> name_server:add(joe, "at home").
ok
3> name_server:whereis(joe).
{ok,"at home"}
```

现在停下来，想一想。在回调模块中，没有关于并发的代码，也没有spawn、send、receive
287 和register等。它只是纯粹的顺序代码，没别的了，这有什么深意吗？

这意味着我们可以在无须理解底层并发模型的前提下，照样编写“客户/服务器”模型的程序。

这是所有服务器的基本模式。一旦领会了这个基本结构，依葫芦画瓢就是很简单的事了。

16.1.2　server 2：支持事务的服务器程序

下面的这个服务器程序，在请求导致服务器程序出现异常时，会让客户端代码异常退出。

server2.erl

```
-module(server2).
-export([start/2, rpc/2]).

start(Name, Mod) ->
    register(Name, spawn(fun() -> loop(Name,Mod,Mod:init()) end)).

rpc(Name, Request) ->
    Name ! {self(), Request},
    receive
        {Name, crash} -> exit(rpc);
        {Name, ok, Response} -> Response
    end.

loop(Name, Mod, OldState) ->
    receive
```

① 其他的语言讲求“接口和实现分离”，所以，这两部分通常会分开，放到两个文件当中，比如：name_server.h和name_server.c，或interface和implement。

```
        {From, Request} ->
            try Mod:handle(Request, OldState) of
                {Response, NewState} ->
                    From ! {Name, ok, Response},
                    loop(Name, Mod, NewState)
            catch
                _:Why ->
                    log_the_error(Name, Request, Why),
                    %% send a message to cause the client to crash
                    From ! {Name, crash},
                    %% loop with the *original* state
                    loop(Name, Mod, OldState)
            end
    end.

log_the_error(Name, Request, Why) ->
    io:format("Server ~p request ~p ~n"
              "caused exception ~p~n",
              [Name, Request, Why]).
```

这个版本的服务器程序，实际上提供了一种“事务机制”——如果在handler函数中发生了异常，它会用之前的状态进行循环；只有当handler函数正常返回时，才会用handler函数返回的新状态进行循环。

288

这里为什么需要保留前一个状态呢？当handler函数失效时，发起调用的客户端会收到一个让它立刻退出的消息。因为请求导致了handler函数失败，所以客户端的调用是无法成功的。但受到影响的也只有这一个客户端而已，连到同一服务器上的其他客户端不会受到任何影响。另外，当handler函数中发生错误的时候，服务器的状态也没有因此而发生任何改变。

注意，相比上一个版本，这里的回调模块并没有任何改动，只是更改了服务器代码本身。换句话说，我们可以独立地对非功能性的行为部分进行更改，而功能性的部分不受影响。

说明 从严格意义上说，上面这句话并不是很严谨。我们实际上还是需要稍微更改回调模块，也就是说，在-import声明中，需要把名字从server1改成server2。除此之外，再没其他的更改了。

16.1.3 server 3：支持热代码替换的服务器程序

下面，我们来增加热代码替换的特性：

server3.erl

```
-module(server3).
-export([start/2, rpc/2, swap_code/2]).

start(Name, Mod) ->
    register(Name,
             spawn(fun() -> loop(Name,Mod,Mod:init()) end)).
```

```
swap_code(Name, Mod) -> rpc(Name, {swap_code, Mod}).

rpc(Name, Request) ->
    Name ! {self(), Request},
    receive
        {Name, Response} -> Response
    end.

loop(Name, Mod, OldState) ->
    receive
        {From, {swap_code, NewCallBackMod}} ->
            From ! {Name, ack},
            loop(Name, NewCallBackMod, OldState);
        {From, Request} ->
            {Response, NewState} = Mod:handle(Request, OldState),
            From ! {Name, Response},
            loop(Name, Mod, NewState)
    end.
```

289

它是怎么工作的？

如果向服务器程序发送一个替换代码的消息，服务器程序就会把回调模块换成消息中的新模块。

我们可以用例子来演示这一点。先用一个回调模块启动server3，然后再动态地替换一个回调模块。这里，我们不能再继续使用name_server了，因为它硬编码了服务器的名字。我们来做一个新的name_server1，改改服务器的名字（也就是把server1改成了server3）。

name_server1.erl

```
-module(name_server1).
-export([init/0, add/2, whereis/1, handle/2]).
-import(server3, [rpc/2]).

%% client routines
add(Name, Place) -> rpc(name_server, {add, Name, Place}).
whereis(Name)    -> rpc(name_server, {whereis, Name}).

%% callback routines
init() -> dict:new().

handle({add, Name, Place}, Dict) -> {ok, dict:store(Name, Place, Dict)};
handle({whereis, Name}, Dict)    -> {dict:find(Name, Dict), Dict}.
```

首先，我们使用name_server1回调模块启动server3：

```
1> server3:start(name_server, name_server1).
true
2> name_server:add(joe, "at home").
ok
3> name_server:add(helen, "at work").
ok
```

假设我们要从名称服务器中列出所有的名字，但是API中没有支持这个需求的方法，name_server只支持add和lookup访问。

我们飞快地打开文本编辑器，写一个新的回调模块：

new_name_server.erl

```
-module(new_name_server).
-export([init/0, add/2, all_names/0, delete/1, whereis/1, handle/2]).
-import(server3, [rpc/2]).
%% interface
all_names()      -> rpc(name_server, allNames).
add(Name, Place) -> rpc(name_server, {add, Name, Place}).
delete(Name)     -> rpc(name_server, {delete, Name}).
whereis(Name)    -> rpc(name_server, {whereis, Name}).

%% callback routines
init() -> dict:new().

handle({add, Name, Place}, Dict) -> {ok, dict:store(Name, Place, Dict)};
handle(allNames, Dict)           -> {dict:fetch_keys(Dict), Dict};
handle({delete, Name}, Dict)     -> {ok, dict:erase(Name, Dict)};
handle({whereis, Name}, Dict)    -> {dict:find(Name, Dict), Dict}.
```

290

编译程序，然后让服务器程序替换回调模块：

```
4> c(new_name_server).
{ok,new_name_server}
5> server3:swap_code(name_server, new_name_server).
ack
```

现在我们可以使用服务器程序中新的函数了：

```
6> new_name_server:all_names().
[joe,helen]
```

正如你所见，回调模块的热代码替换就是这么实现的，这个特性又被称作动态代码升级。听起来挺唬人的吧，可是，这些步骤就在你眼前发生，简简单单、一清二楚。

再停下来想想。刚才增加的两个特性，在我们以往的经验里绝对是艰巨的挑战。支持事务的服务器程序？恐怕很棘手吧。支持动态代码升级？恐怕你想都没想过。

这项强大的技术颠覆了我们的传统想法。一直以来，服务器程序都被看作是有状态的程序，我们向它发送消息，它的状态就会发生改变。但服务器程序正在运行的时候，它的代码是不可改变的。想改变服务器代码，必须要停机，更改代码，然后再重新启动。但正如我们在刚才的例子中看到的，改变服务器程序中正在运行的代码也可以变得和改变服务器的状态一样容易。[①]

291

16.1.4 server 4：同时支持事务和热代码替换

上面的两个例子，代码升级和事务机制是两个分离的特性。我们现在来把这两者合并到一个

① 在一些不允许因为软件升级维护而停机的系统中，我们已经大量应用了这种技术。

服务器程序中。看清楚，别眨眼。

server4.erl

```
-module(server4).
-export([start/2, rpc/2, swap_code/2]).

start(Name, Mod) ->
    register(Name, spawn(fun() -> loop(Name,Mod,Mod:init()) end)).

swap_code(Name, Mod) -> rpc(Name, {swap_code, Mod}).

rpc(Name, Request) ->
    Name ! {self(), Request},
    receive
        {Name, crash} -> exit(rpc);
        {Name, ok, Response} -> Response
    end.

loop(Name, Mod, OldState) ->
    receive
        {From, {swap_code, NewCallbackMod}} ->
            From ! {Name, ok, ack},
            loop(Name, NewCallbackMod, OldState);
        {From, Request} ->
            try Mod:handle(Request, OldState) of
                {Response, NewState} ->
                    From ! {Name, ok, Response},
                    loop(Name, Mod, NewState)
            catch
                _: Why ->
                    log_the_error(Name, Request, Why),
                    From ! {Name, crash},
                    loop(Name, Mod, OldState)
            end
    end.

log_the_error(Name, Request, Why) ->
    io:format("Server ~p request ~p ~n"
              "caused exception ~p~n",
              [Name, Request, Why]).
```

是的，这段简单的代码就是一个同时支持热代码替换和事务机制的服务器框架。怎么样，还
292 不错吧。

16.1.5 server 5：压轴好戏

我们已经有了动态升级代码的特性，还可以更进一步，来点更有意思的。下面这个服务器程序什么服务也不提供，直到你给它下达指令，让它成为某种服务器程序：

server5.erl

```
-module(server5).
-export([start/0, rpc/2]).

start() -> spawn(fun() -> wait() end).

wait() ->
    receive
        {become, F} -> F()
    end.

rpc(Pid, Q) ->
    Pid ! {self(), Q},
    receive
        {Pid, Reply} -> Reply
    end.
```

如果我们启动这个服务器程序，然后向它发送{become, F}消息，它就会通过计算F()函数把自己变成一个F服务器程序。

```
1>  Pid = server5:start().
<0.57.0>
```

此时，我们的服务器程序什么也不做，它就等着become消息。

我们现在来定义一个服务器函数，下面是一个简单的阶乘计算：

my_fac_server.erl

```
-module(my_fac_server).
-export([loop/0]).

loop() ->
    receive
        {From, {fac, N}} ->
            From ! {self(), fac(N)},
            loop();
        {become, Something} ->
            Something()
    end.

fac(0) -> 1;
fac(N) -> N * fac(N-1).
```

293

PlanetLab①的Erlang

几年前，我以研究人员的身份参与了PlanetLab的工作。我有了PlanetLab网络的访问权限，于是在PlanetLab的所有机器上安装了上面这样的“空”Erlang服务器程序（大概有450台左右）。

① PlanetLab是一个全球范围的研究网络（http://www.planet-lab.org）。

我当时也不是很清楚可以用这些机器来做点什么，所以，我先把服务器程序的基础架构搭建起来，留待日后扩展。

这个架构一旦运行起来，只要很轻松地向“空”服务器程序发出一条消息，就可以让它成为一个真正的服务器程序。

比如说，平时我们的工作顺序是先启动一个Web服务器程序，然后再给它安装一些插件。而我的方法是退后一步，先装一个“空”服务器程序，然后采用这种机制把空服务器程序变成一个Web服务器程序。在使用Web服务器程序之后，我们还可以让它变成其他的服务器程序。

先确定已经编译，然后让进程<0.57.0>变成我们想要的阶乘服务器程序。

```
2> c(my_fac_server).
{ok,my_fac_server}
3>  Pid ! {become, fun my_fac_server:loop/0}.
{become,#Fun<my_fac_server.loop.0>}
```

现在，这个进程已经是一个阶乘服务器程序了，我们可以这样来调用：

```
4> server5:rpc(Pid, {fac,30}).
265252859812191058636308480000000
```

直到我们下次再给它发送{become, Something}消息，让它变成其他的服务器程序，否则它会继续提供阶乘服务。

正如你前面所见到的，我们可以制造出各种不同类型的服务器，它们有着各不相同的机制以及让人惊奇的特性。这个技术强大得甚至有些超越我们的驾驭能力。借助它无穷的潜力，我们可以让一段小程序变得非常强大，而依然保持优美。但是当我们面对的是工业规模的项目，牵涉到成百上千的程序员时，可能并不是很希望有太多动态的东西。我们需要在通用与强大之间保持平衡，专注于为商用产品提供有用的特性。能在保持运行的前提下对产品的代码进行升级，这是很
294 棒的特性，但如果发现代码有问题，需要调试，这个时候如果因为对代码做了太多的动态改动而难以确定具体出错的地方，则有可能会让人抓狂。

这一节的例子代码，其实并不是完全正确的。这样写代码的方式主要是为了演示我们这里提到的思路，代码中其实存在着小的隐含问题。这里我们并不会立即指出这些问题，但在本章结尾会给出一些提示。

Erlang的gen_server模块是上面这个思路的结果，它提供了一个成熟的服务器框架（与本章中前面的程序一脉相承）。

从1998年起，它就已经广泛应用于工业产品之中。一个产品可能由几百个服务器程序组成。这些服务器程序由程序员的常规顺序代码构成。而所有的错误处理和非功能性的行为，已经作为服务器程序的一个通用部分被提取出来了。

下面，让我们从想象跳回现实，看看实际工作中的gen_server是怎样的。

16.2　gen_server 起步

现在我们来到了最细节的部分。下面是写gen_server回调模块的3个要点，很简单。

(1) 确定一个回调模块的名称。
(2) 写接口函数。
(3) 在回调模块中写需要的6个回调函数。
这确实很简单，无须思考——只管照着做就行了。

16.2.1 第一步：确定回调模块的名称

下面要做一个非常简单的支付系统，我们把这个模块命名为my_bank。[①] 295

16.2.2 第二步：写接口函数

我们会定义下面5个接口函数，它们都在my_bank模块之中。
- start()打开银行。
- stop()关闭银行。
- new_account(Who)开一个新账户。
- deposit(Who, Amount)存钱。
- withdraw(Who, Amount)取钱，当然，这里需要判断余额是否足够。

每一个函数都意味着对gen_server中的例程的一次调用，如下：

my_bank.erl

```
start() -> gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).
stop()  -> gen_server:call(?MODULE, stop).

new_account(Who)      -> gen_server:call(?MODULE, {new, Who}).
deposit(Who, Amount)  -> gen_server:call(?MODULE, {add, Who, Amount}).
withdraw(Who, Amount) -> gen_server:call(?MODULE, {remove, Who, Amount}).
```

gen_server:start_link({local, Name}, Mod, ...) 会启动一个本地服务器。[②]?MODULE宏展开的时候会对应模块的名称，也就是这里的my_bank。Mod是回调模块的名称。现在我们暂时忽略gen_server:start的其他参数。

gen_server:call(?MODULE, Term)用来发起对服务器的远程调用。

16.2.3 第三步：编写回调函数

回调模块必须开放6个回调函数：init/1、handle_call/3、handle_cast/2、handle_info/2、terminate/2和code_change/3。 296

为了更加省事，我们可以用模板来写gen_server程序，这是最简单的模板：

① 和你想的一样，确实有几个在线的金融服务是用Erlang写的（比如：http://kreditor.se/）。他们没有发布他们的代码，但如果他们发布出来，你很可能会发现与我们这里的代码差不多。
② 使用global参数，将启动一个能在由Erlang节点组成的集群中全局访问的服务器。

gen_server_template.mini

```
-module().
%% gen_server_mini_template

-behaviour(gen_server).
-export([start_link/0]).
%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

start_link() -> gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).

init([]) -> {ok, State}.

handle_call(_Request, _From, State) -> {reply, Reply, State}.
handle_cast(_Msg, State) -> {noreply, State}.
handle_info(_Info, State) -> {noreply, State}.
terminate(_Reason, _State) -> ok.
code_change(_OldVsn, State, Extra) -> {ok, State}.
```

这个模板包含了简单的骨架，我们可以填充模板，来创建自己的服务器程序。加了-behaviour关键字后，当我们忘了定义需要的回调函数时，编译器会产生警告或者错误消息。

提示　如果你用emacs，那么可以把gen_server模板绑定到几个快捷键上。如果在Erlang模式下编辑，Erlang > Skeletons菜单提供了创建gen_server模板的选项。如果你没使用emacs，也无所谓，本章末尾也提供了这些模板。

从这个模板开始，我们只需要稍微编辑一下，要做的是将模板中的参数调整为接口函数中的参数。

最主要的部分是handle_call/3函数。我们要把自定义的接口和这3个参数进行匹配，也就是说，我们要填充下面省略掉的部分：

```
handle_call({new, Who}, From, State} ->
    Reply  = ...
    State1 = ...
    {reply, Reply, State1};
handle_call({add, Who, Amount}, From, State} ->
    Reply  = ...
    State1 = ...
    {reply, Reply, State1};
handle_call({remove, Who, Amount}, From, State} ->
    Reply  = ...
    State1 = ...
    {reply, Reply, State1};
```

297

上面代码中的Reply值会作为远程过程调用的返回值发回给发起调用的客户端。

State是表示服务器全局状态的变量，它会在后面的循环中继续传递。在我们的银行模块中，

它是ETS表的标识，这个状态没有发生变化（尽管，表的标识没有发生变化，但表中的内容会发生变化）。

我们继续填充这个模板，不断编辑，最终得到的代码会是下面这样：

my_bank.erl

```
init([]) -> {ok, ets:new(?MODULE,[])}.

handle_call({new,Who}, _From, Tab) ->
    Reply = case ets:lookup(Tab, Who) of
                []  -> ets:insert(Tab, {Who,0}),
                       {welcome, Who};
                [_] -> {Who, you_already_are_a_customer}
            end,
    {reply, Reply, Tab};
handle_call({add,Who,X}, _From, Tab) ->
    Reply = case ets:lookup(Tab, Who) of
                []  -> not_a_customer;
                [{Who,Balance}] ->
                    NewBalance = Balance + X,
                    ets:insert(Tab, {Who, NewBalance}),
                    {thanks, Who, your_balance_is,  NewBalance}
            end,
    {reply, Reply, Tab};
handle_call({remove,Who, X}, _From, Tab) ->
    Reply = case ets:lookup(Tab, Who) of
                []  -> not_a_customer;
                [{Who,Balance}] when X =< Balance ->
                    NewBalance = Balance - X,
                    ets:insert(Tab, {Who, NewBalance}),
                    {thanks, Who, your_balance_is,  NewBalance};
                [{Who,Balance}] ->
                    {sorry,Who,you_only_have,Balance,in_the_bank}
            end,
    {reply, Reply, Tab};                                          298
handle_call(stop, _From, Tab) ->
    {stop, normal, stopped, Tab}.

handle_cast(_Msg, State) -> {noreply, State}.
handle_info(_Info, State) -> {noreply, State}.
terminate(_Reason, _State) -> ok.
code_change(_OldVsn, State, Extra) -> {ok, State}.
```

通过调用gen_server:start_link(Name, CallBackMod, StartArgs, Opts)启动服务器程序时，回调模块中第一个被调用的函数是Mod:init(StartArgs)，它必须返回{ok, State}。这里的State值会作为handle_call函数的第3个参数再次出现。

注意终止服务器程序的方法。handle_call(Stop, From, Tab)返回{stop, normal, stopped, Tab}，这就是终止服务器程序的方法。第二个参数（normal）会作为my_bank:terminate/2的第

一个参数；而第三个参数（stopped）则会成为my_bank:stop()的返回值。

代码就这么多了，我们来访问这个银行吧。

```
1> my_bank:start().
{ok,<0.33.0>}
2> my_bank:deposit("joe", 10).
not_a_customer
3> my_bank:new_account("joe").
{welcome,"joe"}
4> my_bank:deposit("joe", 10).
{thanks,"joe",your_balance_is,10}
5> my_bank:deposit("joe", 30).
{thanks,"joe",your_balance_is,40}
6> my_bank:withdraw("joe", 15).
{thanks,"joe",your_balance_is,25}
7> my_bank:withdraw("joe", 45).
{sorry,"joe",you_only_have,25,in_the_bank}
```

16.3 gen_server 回调的结构

有了对gen_server的感性的认识，现在我们再来看看gen_server回调模块的更多细节。

16.3.1 启动服务器程序时发生了什么

gen_server:start_link(Name, Mod, InitArgs, Opts)调用负责启动所有这一切。它创建了一个名为Name的通用服务器程序，它的回调模块是Mod，Opts则控制了这个通用服务器程序的行为。在这里我们可以指定日志信息、调试函数等。然后就是调用Mod:init(InitArgs)来启动服务器程序。

299

在模板中，初始化方法的入口是这样的：

```
%%--------------------------------------------------------------------
%% Function: init(Args) -> {ok, State}          |
%%                         {ok, State, Timeout} |
%%                         ignore               |
%%                         {stop, Reason}
%% Description: Initiates the server
%%--------------------------------------------------------------------
init([]) ->
    {ok, #state{}}.
```

通常情况下，我们返回的是{ok, State}。如果想知道其他参数的含义，可以查阅文档中关于gen_server的部分。

返回值是{ok, State}，这意味着已经成功启动，服务器的初始状态是State。

16.3.2 调用服务器程序时发生了什么

为了调用服务器程序，客户端程序会调用函数gen_server:call(Name, Request)。这会导

致调用回调模块中的handle_call/3函数。

handle_call/3在模板中的入口是这样定义的：

```
%%--------------------------------------------------------------------
%% Function:
%% handle_call(Request, From, State) -> {reply, Reply, State}          |
%%                                      {reply, Reply, State, Timeout} |
%%                                      {noreply, State}               |
%%                                      {noreply, State, Timeout}      |
%%                                      {stop, Reason, Reply, State}   |
%%                                      {stop, Reason, State}
%% Description: Handling call messages
%%--------------------------------------------------------------------
handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.
```

gen_server:call/2的第二个参数Request会作为第一个参数再次出现在handle_call/3中。From是发起调用的客户端进程的PID，State是当前的客户端状态。

如果一切正常的话，我们返回{replay, Reply, NewState}。这个时候，Reply会作为gen_server:call的返回值提交给客户端，而NewState是服务器的新状态。

其他的返回值包括{noreply,...}和{stop,...}，它们用得相对较少。noreply会使得服务器继续运行，但客户端会等待回应，回调程序要从服务框架接管向其他进程返回数据的职责。而stop则会导致服务器程序终止运行。 300

16.3.3 调用和通知

我们已经看到了在gen_server:call和handle_call之间的交互动作。它们主要用来实现远程过程调用（RPC）。而gen_server:cast(Name, Name)则实现了通知（cast），就是一个无返回值的调用（实际上，也就是发送一个消息，但传统上将这种情况叫做通知，用来和远程调用区别开来）。

相应的回调函数是handle_cast，在模板中的入口是这样的：

```
%%--------------------------------------------------------------------
%% Function: handle_cast(Msg, State) -> {noreply, NewState}          |
%%                                      {noreply, NewState, Timeout} |
%%                                      {stop, Reason, NewState}
%% Description: Handling cast messages
%%--------------------------------------------------------------------
handle_cast(_Msg, State) ->
    {noreply, NewState}.
```

处理程序通常只返回{noreply, NewState}，这会改变服务器的状态；它也可以返回{stop,...}，这会终止服务器。

16.3.4 发给服务器的原生消息

回调函数handle_info(Info, State)用来处理发给服务器的原生消息。那么，到底什么样

的消息才是原生消息呢？如果服务器和其他的进程建立了连接，并且正在捕捉退出事件，那么它有可能会突然收到意外的{'EXIT', Pid, What}这样的消息。又或者，系统中的其他进程获得了通用服务器程序的PID，那么它可能会给服务器发送消息。诸如此类的消息最终都会作为Info的值传给回调程序。

handle_info在模板中的入口是这样的：

```
%%--------------------------------------------------------------------
%% Function: handle_info(Info, State) -> {noreply, State}          |
%%                                       {noreply, State, Timeout} |
%%                                       {stop, Reason, State}
%% Description: Handling all non-call/cast messages
%%--------------------------------------------------------------------
handle_info(_Info, State) ->
    {noreply, State}.
```

301 它的返回值和handle_cast的一样。

16.3.5 Hasta la Vista, Baby（服务器的终止）

很多原因会导致服务器终止。比如，在handle_Something函数中返回一个{stop, Reason, NewState}，或者直接终止服务器，返回一个{'EXIT', reason}。所有这些情况，无论是怎么引起的，无一例外，都会调用回调模块的terminate(Reason, NewState)函数。

模板是这样的：

```
%%--------------------------------------------------------------------
%% Function: terminate(Reason, State) -> void()
%% Description: This function is called by a gen_server when it is
%% about to terminate. It should be the opposite of Module:init/1 and
%% do any necessary
%% cleaning up. When it returns, the gen_server terminates with Reason.
%% The return value is ignored.
%%--------------------------------------------------------------------
terminate(_Reason, State) ->
    ok.
```

因为我们要终止了，所以这个代码没有必要再返回一个新的状态了。那么，要如何处理参数中的这个State呢？其实还是有很多事情可以做的。主要取决于你的应用程序，比如，可以把这个State写到磁盘，以消息形式发给另一个进程，或者直接丢掉。如果将来某个时候你还要再次启动这个服务器，可以用terminate/2来触发 "I'll be back" 函数。[①]

16.3.6 热代码替换

你可以在运行中动态地改变服务器的状态。版本管理子系统会在系统对软件进行升级时调用这个回调函数。

① "Hasta La Vista, Baby" 和 "I'll be back" 是电影《终结者2》中的对白，入选全球100大经典台词。前者的意思是 "再见，宝贝"，后者则是 "我会回来的"。——译者注

如果想了解这个话题的更多细节，可以参阅OTP设计原则文档中的版本管理一节。[①]

```
%%--------------------------------------------------------------------
%% Func: code_change(OldVsn, State, Extra) -> {ok, NewState} %%
%% Description: Convert process state when code is changed
%%--------------------------------------------------------------------
code_change(_OldVsn, State, _Extra) -> {ok, State}.
```

302

16.4 代码和模板

这个模板已经在emacs-mode中内置：

16.4.1 gen_server 模板

gen_server_template.full

```
%%%-------------------------------------------------------------------
%%% File    : gen_server_template.full
%%% Author  : my name <yourname@localhost.localdomain>
%%% Description :
%%%
%%% Created :  2 Mar 2007 by my name <yourname@localhost.localdomain>
%%%-------------------------------------------------------------------
-module().

-behaviour(gen_server).

%% API
-export([start_link/0]).

%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

-record(state, {}).

%%====================================================================
%% API
%%====================================================================
%%--------------------------------------------------------------------
%% Function: start_link() -> {ok,Pid} | ignore | {error,Error}
%% Description: Starts the server
%%--------------------------------------------------------------------
start_link() ->
    gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).

%%====================================================================
%% gen_server callbacks
```

① OTP的设计原则，这个文档可以从http://www.erlang.org/doc/pdf/design_principles.pdf 下载。

```
%%====================================================================

%%--------------------------------------------------------------------
%% Function: init(Args) -> {ok, State} |
%%                         {ok, State, Timeout} |
%%                         ignore               |
%%                         {stop, Reason}
%% Description: Initiates the server
%%--------------------------------------------------------------------
init([]) ->
    {ok, #state{}}.
%%--------------------------------------------------------------------
%% Function: %% handle_call(Request, From, State) -> {reply, Reply, State} |
%%                                      {reply, Reply, State, Timeout} |
%%                                      {noreply, State} |
%%                                      {noreply, State, Timeout} |
%%                                      {stop, Reason, Reply, State} |
%%                                      {stop, Reason, State}
%% Description: Handling call messages
%%--------------------------------------------------------------------
handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.

%%--------------------------------------------------------------------
%% Function: handle_cast(Msg, State) -> {noreply, State} |
%%                                      {noreply, State, Timeout} |
%%                                      {stop, Reason, State}
%% Description: Handling cast messages
%%--------------------------------------------------------------------
handle_cast(_Msg, State) ->
    {noreply, State}.

%%--------------------------------------------------------------------
%% Function: handle_info(Info, State) -> {noreply, State} |
%%                                       {noreply, State, Timeout} |
%%                                       {stop, Reason, State}
%% Description: Handling all non call/cast messages
%%--------------------------------------------------------------------
handle_info(_Info, State) ->
    {noreply, State}.

%%--------------------------------------------------------------------
%% Function: terminate(Reason, State) -> void()
%% Description: This function is called by a gen_server when it is about to
%% terminate. It should be the opposite of Module:init/1 and do any necessary
%% cleaning up. When it returns, the gen_server terminates with Reason.
%% The return value is ignored.
%%--------------------------------------------------------------------
terminate(_Reason, _State) ->
```

303

```
    ok.

%%--------------------------------------------------------------------
%% Func: code_change(OldVsn, State, Extra) -> {ok, NewState}
%% Description: Convert process state when code is changed
%%--------------------------------------------------------------------
code_change(_OldVsn, State, _Extra) ->
    {ok, State}.

%%--------------------------------------------------------------------
%%% Internal functions
%%--------------------------------------------------------------------
```

304

16.4.2 my_bank

my_bank.erl

```
-module(my_bank).

-behaviour(gen_server).
-export([start/0]).
%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).
-compile(export_all).

start() -> gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).
stop()  -> gen_server:call(?MODULE, stop).

new_account(Who)      -> gen_server:call(?MODULE, {new, Who}).
deposit(Who, Amount)  -> gen_server:call(?MODULE, {add, Who, Amount}).
withdraw(Who, Amount) -> gen_server:call(?MODULE, {remove, Who, Amount}).

init([]) -> {ok, ets:new(?MODULE,[])}.

handle_call({new,Who}, _From, Tab) ->
    Reply = case ets:lookup(Tab, Who) of
                []  -> ets:insert(Tab, {Who,0}),
                       {welcome, Who};
                [_] -> {Who, you_already_are_a_customer}
            end,
    {reply, Reply, Tab};
handle_call({add,Who,X}, _From, Tab) ->
    Reply = case ets:lookup(Tab, Who) of
                []  -> not_a_customer;
                [{Who,Balance}] ->
                    NewBalance = Balance + X,
                    ets:insert(Tab, {Who, NewBalance}),
                    {thanks, Who, your_balance_is,  NewBalance}
            end,
    {reply, Reply, Tab};
```

```
handle_call({remove,Who, X}, _From, Tab) ->
    Reply = case ets:lookup(Tab, Who) of
                []  -> not_a_customer;
                [{Who,Balance}] when X =< Balance ->
                    NewBalance = Balance - X,
                    ets:insert(Tab, {Who, NewBalance}),
                    {thanks, Who, your_balance_is,  NewBalance};
                [{Who,Balance}] ->
                    {sorry,Who,you_only_have,Balance,in_the_bank}
            end,
305 {reply, Reply, Tab};
handle_call(stop, _From, Tab) ->
    {stop, normal, stopped, Tab}.
handle_cast(_Msg, State) -> {noreply, State}.
handle_info(_Info, State) -> {noreply, State}.
terminate(_Reason, _State) -> ok.
code_change(_OldVsn, State, Extra) -> {ok, State}.
```

16.5 进一步深入

gen_server其实相当简单。我们这里并没有完全覆盖它所有的接口函数，也没有讨论所有接口函数的所有参数。重要的是，一旦你理解了它的基本概念，就可以从手册中了解gen_server的更多细节。

我们这一章只考察了使用gen_server的最简单的情况，这已经能够满足大多数情况下的需要。更复杂的应用程序通常会让gen_server返回noreply的返回值，然后自己来接管对其他进程返回数据的职责。可以阅读“设计原则”文档以及sys和proc_lib模块的文档[①]，了解更多这方
306 面的信息。

① http://www.erlang.org/doc/pdf/design_principles.pdf。

第17章

Mnesia：Erlang数据库

无论是做一个多用户的游戏，搭建一个新的网站，还是架设一个在线支付系统，如果不用数据库管理系统（DBMS），恐怕都是难以想象的。

实际上，在下载和安装Erlang后，在磁盘上的数千个文件之中，已经内置了一个相当完备的数据库管理系统，它的名字叫做Mnesia。它非常快，而且，更妙的是可以直接存储任意的Erlang数据结构（也就是说，你不用费劲地做什么持久化的动作）。

Mnesia可以根据需要灵活配置。比如说，既可以把数据表存储在内存中（如果你着重考虑数据的访问速度），也可以存储在磁盘上（如果你更关注数据的持久性），甚至还可以在不同的机器上建立同一份数据的多个副本（如果你的数据无比珍贵，不容有失）。

有意思吧，我们来深入了解一下。

17.1 数据库查询

我们从Mnesia的查询功能开始。我预见到，你很可能会惊讶于Mnesia的查询语法与SQL[①]或列表解析之间的相似性。这种相似性意味着我们并不需要学太多的新东西就能迅速上手。[②]

下面的例子，都假设已经建好了数据库，这个库中有两张表（如表17-1和表17-2所示），分别是shop和cost。它们分别包含下面这样的数据。

表17-1 shop表

Item	Quantity	Cost
apple	20	2.3
orange	100	3.8
pear	200	3.6
banana	420	4.5
potato	2456	1.2

① 一种流行的关系数据库查询语言。

② 实际上，列表解析和SQL语句看起来也很相似。对此，你实在没有必要感到惊讶，因为它们都源自数学中的集合论。

表17-2　cost表

Name	Price
apple	1.5
orange	2.4
pear	2.2
banana	1.5
potato	0.6

在Mnesia中表示这样的表，我们还需要定义表示每列记录的结构。这些记录的定义是这样的：

test_mnesia.erl

```
-record(shop, {item, quantity, cost}).
-record(cost, {name, price}).
```

在这里我尽量讲的通俗易懂，但恐怕很难做到，我要展现查询是怎么回事，让你找到其中的窍门。要做到这点，必须要创建并填充数据库。为了不分散你的注意力，无奈之下，只能来玩点小把戏了。请相信我，并照我说的做。我已经把这些初始化的代码写好了，它们在test_mnesia.erl文件中，我们用erl来运行它。

```
1> c(test_mnesia).
{ok,test_mnesia}
2> test_mnesia:do_this_once().
=INFO REPORT==== 29-Mar-2007::20:33:12 ===
    application: mnesia
    exited: stopped
    type: temporary
stopped
```

308 好了，现在让我们从霍格沃兹（影片《哈利波特》中的魔法学院）回来，继续我们的例子。

17.1.1　选取表中所有的数据

下面是选取shop表中所有数据的代码。（我为那些熟悉SQL的读者在每一小段代码前面的注释中给出了与之等价的SQL语句）

test_mnesia.erl

```
%% SQL equivalent
%%  SELECT * FROM shop;

demo(select_shop) ->
    do(qlc:q([X || X <- mnesia:table(shop)]));
```

这里的重点在于对qlc:q的调用，它负责把参数中的查询语句编译为用来执行数据库查询的内部形式。我们把得到的编译结果交给一个名为do()的函数，这个函数也是在test_mnesia中定义的。它的职责是运行这个查询，并返回结果。为了便于在erl中执行，我们将整个过程都包含在demo(select_shop)函数中。（mnesia_test完整的代码清单就在本章末尾。）

我们可以这样运行：

```
1> test_mnesia:start().
ok
2> test_mnesia:reset_tables().
{atomic, ok}
3> test_mnesia:demo(select_shop).
[{shop,orange,100,3.80000},
 {shop,pear,200,3.60000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,20,2.30000}]
```

说明 在这里，数据行是以不确定的顺序出现的，因为我们并没有做任何排序处理。

在这里，查询是在语句qlc:q([X||X<-mnesia:table(shop)])中设置的。

你注意到了吗，这个语法和我们在3.6节看到的非常相似。是的，实际上，qlc就是query list comprehensions（查询列表解析）的缩写，要访问Mnesia数据库中的数据，qlc是一个很重要的工具。

[X||X<-mnesia:table(shop)]意思是“从Mnesia的shop表中获取X（每一行作为一个X），用这些X构成一个列表”。这里的X，实际上其类型就是我们上面定义的shop记录。 309

说明 qlc:q/1的参数必须是列表解析语法的语句本身，而不能是任何其他结果相同的表达式。比如，下面的代码与我们的例子可不是等价的：

```
Var=[X||X<-mnesia:table(shop)],qlc:q(Var)
```

17.1.2 选取表中的数据

这个查询从shop表中选取品名和数量这两列的数据。

test_mnesia.erl

```
%% SQL equivalent
%%  SELECT item, quantity FROM shop;

demo(select_some) ->
    do(qlc:q([{X#shop.item, X#shop.quantity} || X <- mnesia:table(shop)])).
```

```
4> test_mnesia:demo(select_some).
[{orange,100},{pear,200},{banana,420},{potato,2456},{apple,20}]
```

上面这个查询中，X的类型是之前定义的shop记录。回忆一下我们在3.9节中描述过的语法，X#shop.item这个语法引用了shop记录的item字段。因此，{X#shop.item, X#shop.quantity}就是由X的item和quantity这两个字段构成的元组。

17.1.3 按条件选取表中的数据

下面的查询从shop表中选取所有库存量小于250的条目。也许我们可以执行这个查询，看看哪些商品需要补货了。

test_mnesia.erl

```
%% SQL equivalent
%%   SELECT shop.item FROM shop
%%   WHERE  shop.quantity < 250;

demo(reorder) ->
    do(qlc:q([X#shop.item || X <- mnesia:table(shop),
                             X#shop.quantity < 250
                                ])).
```

```
5> test_mnesia:demo(reorder).
[orange,pear,apple]
```

310 看看，把查询条件写到列表解析的语句中，是一种多么流畅自然的表达方式。

17.1.4　从两个表选取数据（关联查询）

假设我们补货的限制条件是库存少于250而且单价低于2.0个货币单位。

要完成这个查询，我们必须访问两张表，查询的代码是这样的：

test_mnesia.erl

```
%% SQL equivalent
%%   SELECT shop.item, shop.quantity, cost.name, cost.price
%%   FROM shop, cost
%%   WHERE shop.item = cost.name
%%     AND cost.price < 2
%%     AND shop.quantity < 250

demo(join) ->
    do(qlc:q([X#shop.item || X <- mnesia:table(shop),
                             X#shop.quantity < 250,
                             Y <- mnesia:table(cost),
                             X#shop.item =:= Y#cost.name,
                             Y#cost.price < 2
                                ])).
```

```
6> test_mnesia:demo(join).
[apple]
```

这里的关键是要用商品名称来做关联，也就是说，要在shop表的item字段和cost表的name字段之间进行关联，语句就是这样：

```
X#shop.item =:= Y#cost.name
```

17.2　增删表中的数据

这里我们要假设已经建好了数据库，而且已经定义好了一个shop表。现在我们要向这个表增

加或删除一些数据。

17.2.1 增加一行

我们可以用这样的语法来向shop表插入数据：

test_mnesia.erl

```
add_shop_item(Name, Quantity, Cost) ->
    Row = #shop{item=Name, quantity=Quantity, cost=Cost},
    F = fun() ->
                mnesia:write(Row)
        end,
    mnesia:transaction(F).
```

上面的代码会创建一个shop记录，并把它插到表中：

```
1> test_mnesia:start().
ok
2> test_mnesia:reset_tables().
{atomic, ok}
%% list the shop table
3> test_mnesia:demo(select_shop).
[{shop,orange,100,3.80000},
 {shop,pear,200,3.60000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,20,2.30000}]
%% add a new row
4> test_mnesia:add_shop_item(orange, 236, 2.8).
 {atomic,ok}
%% list the shop table again so we can see the change
5> test_mnesia:demo(select_shop).
[{shop,orange,236,2.80000},
 {shop,pear,200,3.60000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,20,2.30000}]
```

311

说明 shop表的第一列也是表的主键，也就是shop记录的item字段。这个表的类型是“set”（参考15.2节中讨论的内容）。如果新建的记录和表中的记录主键相同，那么老数据就会被覆盖，如若不然，就会作为一行新的数据插入。

17.2.2 删除一行

要想删除一行数据，需要知道这一行数据的目标ID（OID）。这个OID由表的名称和主键的键值构成：

test_mnesia.erl

```
remove_shop_item(Item) ->
    Oid = {shop, Item},
    F = fun() ->
                mnesia:delete(Oid)
        end,
    mnesia:transaction(F).
```

```
6> test_mnesia:remove_shop_item(pear).
{atomic,ok}
%% list the table -- the pear has gone
7> test_mnesia:demo(select_shop).
[{shop,orange,236,2.80000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,20,2.30000}]
 [{shop,orange,236,2.80000},
8> mnesia:stop().
ok
```

312

17.3　Mnesia 事务

当增加或删除记录或者执行一个查询的时候，我们会写这样的代码：

```
do_something(...) ->
    F = fun() ->
            % ...
            mnesia:write(Row)
            % ... or ...
            mnesia:delete(Oid)
            % ... or ...
            qlc:e(Q)
        end,
    mnesia:transaction(F)
```

其中，F是一个没有参数的函数。在F函数的内部，是mnesia:write/1、mnesia:delete/1或qlc:e(Q)（这里Q是用qlc:q/1编译过的查询语句）这些操作的组合。有了这个函数，我们可以调用mnesia:transaction(F)来执行这个函数中的表达式序列。

写一个函数，然后再用另一函数来执行这个函数，似乎有点复杂，我们为什么要这么做呢？这里的交易又意味着什么？要说明这个问题，我们可以假设这样一种“两个进程同时访问一个数据”的情况。比如，我的银行账户有10块钱。现在有两个人要同时从这个账户中取8块钱的现金。这种情况下，毫无疑问，要有某种机制来保证其中一个人的操作会成功，另一个人的操作会失败。

这种机制，正是由mnesia:transaction/1来提供的。一次事务内的所有读写操作，要么就都成功，要么就全失败。如果事务失败，数据库中的数据不会发生任何变化。

Mnesia解决这个问题时候，采用的是“悲观锁”策略。每次Mnesia交易访问一个表的时候，它会试图进行锁定，具体是该锁定单个记录还是整张表，它会根据查询语句进行判断。如果它判

断出这个查询可能会导致死锁，就会立即取消事务，并恢复这个过程中发生的任何更改。

如果锁定时，因为有其他的进程正在访问数据，因而事务无法成功初始化，那么，系统会稍等一会儿再来重试这个事务。很明显，事务函数中代码很有可能会被执行很多次。 313

正因为此，交易方法内的代码应该避免任何的副作用。比如，如果我们写了这样的代码：

```
F = fun() ->
        ...
        io:format("reading ..."), %% don't do this
        ...
    end,
mnesia:transaction(F),
```

这可能产生大量的输出，因为这个函数可能会被多次重试。

说明1 mnesia:write/1和mnesia:delete/1应该仅在mnesia:transaction/1处理的交易方法中调用。

说明2 你不应自己写代码显式地捕捉和处理Mnesia访问函数（如mnesia:write/1、mnesia:delete/1等）的执行异常。实际上Mnesia的事务机制本身要依赖于这些方法抛出的异常，以检测失败的情况。如果你捕捉这些异常，然后自己处理它们，实际上就破坏了Mnesia的事务机制。

17.3.1 取消一个事务

还是那个商店的例子，我们的商店附近有个农场，出产苹果。农场主很喜欢橙子，他愿意用苹果来换橙子。交互的比例是两个苹果换一个橙子。那么，要买N个橙子，农场主要拿2*N个苹果来。

下面这个方法在农场主换橙子的时候用来更新数据库：

test_mnesia.erl

```
farmer(Nwant) ->
    %% Nwant = Number of oranges the farmer wants to buy
    F = fun() ->
            %% find the number of apples
            [Apple] = mnesia:read({shop,apple}),
            Napples = Apple#shop.quantity,
            Apple1  = Apple#shop{quantity = Napples + 2*Nwant},
            %% update the database
            mnesia:write(Apple1),
            %% find the number of oranges
            [Orange] = mnesia:read({shop,orange}),
            NOranges = Orange#shop.quantity,
            if
                NOranges >= Nwant ->
                    N1 =  NOranges - Nwant,
                    Orange1 = Orange#shop{quantity=N1},
                    %% update the database
```

314

```
                mnesia:write(Orange1);
            true ->
                %% Oops -- not enough oranges
                mnesia:abort(oranges)
        end
    end,
mnesia:transaction(F).
```

上面这段代码，首先就更新了苹果的数量，实际上，应该是在检查橙子的数量足够之后再做这件事的。好吧，我承认这是一个愚蠢的例子，先这么着吧，毕竟我的主要目的是为了展示事务机制是怎么工作的。绕了这么大的一个圈子就是为了展示这个更新在事务失败的时候会发生回滚。通常，我倾向于先确认有足够的橙子，然后再把橙子和苹果的数据写回数据库。

行了，我们来看看，一大早，当农场主跑过来要买50个橙子的时候会怎样：

```
1> test_mnesia:start().
ok
2> test_mnesia:reset_tables().
{atomic, ok}
%% List the shop table
3> test_mnesia:demo(select_shop).
[{shop,orange,100,3.80000},
 {shop,pear,200,3.60000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,20,2.30000}]
%% The farmer buys 50 oranges
%% paying with 100 apples
4> test_mnesia:farmer(50).
{atomic,ok}
%% Print the shop table again
5> test_mnesia:demo(select_shop).
[{shop,orange,50,3.80000},
 {shop,pear,200,3.60000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,120,2.30000}]
```

下午农场主想再多买100个橙子（这个人确实很喜欢橙子）：

```
6> test_mnesia:farmer(100).
{aborted,oranges}
7> test_mnesia:demo(select_shop).
[{shop,orange,50,3.80000},
 {shop,pear,200,3.60000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,120,2.30000}]
```

这个数据库为什么叫Mnesia呢

最初想到的名字是Amnesia（健忘症）。我们的一个老板不喜欢这个名字，他说不能叫Amnesia——一个数据库怎么能忘记事情呢。于是我们去掉了A，有了这个糟糕的名字。

我们调用mnesia:abort(Reson)的时候，事务就失败了，mnesia:write所做的更新操作也被回滚。数据库恢复到进入事务之前的状态。

17.3.2 加载测试数据

现在我们知道事务是怎么回事了，可以回头看看加载测试数据的那些代码。

test_mnesia:example_tables/0函数用来初始化数据表，填充测试数据。元组的第一个元素是表名，后面跟着的数据与原始的记录顺序保持一致。

test_mnesia.erl

```
example_tables() ->
    [%% The shop table
     {shop, apple,   20,   2.3},
     {shop, orange,  100,  3.8},
     {shop, pear,    200,  3.6},
     {shop, banana,  420,  4.5},
     {shop, potato,  2456, 1.2},
     %% The cost table
     {cost, apple,   1.5},
     {cost, orange,  2.4},
     {cost, pear,    2.2},
     {cost, banana,  1.5},
     {cost, potato,  0.6}
    ].
```

这是例子向Mnesia插入数据的代码：

test_mnesia.erl

```
reset_tables() ->
    mnesia:clear_table(shop),
    mnesia:clear_table(cost),
    F = fun() ->
                foreach(fun mnesia:write/1, example_tables())
        end,
    mnesia:transaction(F).
```

316

实际上，通过example_tables/1函数返回的列表，只是对其中的每一个元组调用mnesia:write。

17.3.3 do()函数

demo/1调用的do函数稍微要复杂一点：

test_mnesia.erl

```
do(Q) ->
    F = fun() -> qlc:e(Q) end,
    {atomic, Val} = mnesia:transaction(F),
    Val.
```

它是在Mnesia的事务中调用`qlc:e(Q)`。Q是一个编译过的QLC查询，qlc:e(Q)执行这个查询，并将查询结果都返回到一个列表之中。返回值`{atomic, Val}`的意思是事务成功，返回值是`Val`。这里的`Val`是交易的执行结果。

17.4　在表中保存复杂数据

如果你是一个C程序员，你是怎样把一个C的结构保存到SQL数据库中的呢？又或者你是一个Java程序员，在保存一个对象到数据库中时又要做些什么？实际上，这个步骤相当烦琐。

通常的数据库管理系统有一个缺点，那就是它的每一列都只支持为数不多的几种数据类型。比如说，你可以保存一个整数、一个字符串、一个浮点数等，但如果你想保存的是一个复杂对象，那恐怕就要头疼了。

Mnesia是被设计用来保存Erlang数据结构的。更明确地说，你可以把任意类型的Erlang数据结构保存到Mnesia表中。

比如说，我们要把一些建筑的设计方案放到Mnesia的数据库中。首先，我们需要一个记录，用来表示这些设计：

test_mnesia.erl

```
-record(design, {id, plan}).
```

然后定义一个函数，它向数据库中添加一些设计：

test_mnesia.erl

```
add_plans() ->
    D1 = #design{id   = {joe,1},
                 plan = {circle,10}},
317 D2 = #design{id   = fred,
                 plan = {rectangle,10,5}},
    D3 = #design{id   = {jane,{house,23}},
                 plan = {house,
                         [{floor,1,
                           [{doors,3},
                            {windows,12},
                            {rooms,5}]},
                          {floor,2,
                           [{doors,2},
                            {rooms,4},
                            {windows,15}]}]}},
    F = fun() ->
                mnesia:write(D1),
```

```
                mnesia:write(D2),
                mnesia:write(D3)
        end,
    mnesia:transaction(F).
```

现在我们向数据库添加一些设计：

```
1> test_mnesia:start().
ok
2> test_mnesia:add_plans().
{atomic,ok}
```

现在我们的数据库有了一些设计，可以这样访问函数来获取这些设计：

test_mnesia.erl

```
get_plan(PlanId) ->
    F = fun() -> mnesia:read({design, PlanId}) end,
    mnesia:transaction(F).
```

```
3> test_mnesia:get_plan(fred).
{atomic,[{design,fred,{rectangle,10,5}}]}
4> test_mnesia:get_plan({jane, {house,23}}).
{atomic,[{design,{jane,{house,23}},
                 {house,[{floor,1,[{doors,3},
                                   {windows,12},
                                   {rooms,5}]},
                         {floor,2,[{doors,2},
                                   {rooms,4},
                                    {windows,15}]}]}}]}
```

正如你所见，无论是主键还是被抽取出来的记录，都可以是任意的Erlang表达式。

套用一句技术术语我们可以说，数据库中的数据结构与编程语言的数据结构之间并不存在“匹配阻抗”问题。这意味着向数据库增加或者删除复杂的数据结构是非常迅速的。 318

表格切分

Mnesia支持“表格切分”技术（在数据表上进行横向分割）。这个特性主要用来实现极大量数据的存储。数据表被切分后，会被存储在不同的机器上，它每一个片段都是一个Mnesia的数据表。被切分的表一样可以进行备份、索引等，与其他的表没有什么区别。

参考Mnesia用户指南，可以了解这一特性的更多详情。

17.5 表的类型和位置

Mnesia数据表可以进行很多种不同的配置。首先我们可以选择将表存储在内存或者磁盘上（或者同时保存在内存和磁盘上），其次，可以选择将数据表保存在一台机器上，或者在多台机器上备份。

在设计阶段，我们就必须要认真思考我们想要存储的数据有什么样的特性，然后根据这些特

性来选择。下面是Mnesia数据表的属性。

- *内存表*

非常快，但其中数据是瞬态的，所以当系统崩溃或者停掉数据库的时候，这些数据都会丢失。

- *磁盘表*

系统崩溃不会导致其中的数据丢失（当然了，前提是没有发生磁盘物理损坏）。

对于磁盘表来说，每次成功提交一个Mnesia事务的时候，其底层的实现方式是这样的。它会先将数据写到一个磁盘日志当中，这个日志会持续保持增长，每隔一段时间，会把日志中的数据同步到数据表中，并清除掉日志中的相应条目。如果系统崩溃了，在下次重启时，会检查这个磁盘日志，先将尚未写入的数据同步到数据库中，然后才会开放数据库服务。好，梳理一下整个过程，也就是说，只要一个事务成功，那么其中的数据就会被写到磁盘日志中，即便此时发生系统崩溃，也不会影响这个交易的数据，因为下次重启时，会自动将这些数据同步到数据表中。

对于磁盘表来说，只有一种情况下，会发生数据丢失的情况，那就是系统在提交的过程之中
319 崩溃，此时，交易还没有成功提交，数据还没有写入磁盘日志，因而，重启时也无法进行同步。

在使用内存表之前，有一件事必须要做，那就是检查当前系统的物理内存能否完整装下这张表的全部数据。如果物理内存不够大，那么操作系统就会使用虚拟内存，这意味着大量的内存–磁盘的数据换页操作（也就是所谓的swap或者page操作），这会导致性能的急剧恶化。

内存表是瞬态的，你需要搞清楚的另外一个重要问题是，如果内存表中的数据丢失，有没有关系？如果这个问题的答案是很有关系，那么就需要给这个表在本机建立一个磁盘备份或者在另外的机器上做个备份（既可以是内存的，也可是磁盘的，还可以是两者混合型的）。

17.5.1　创建表

创建一个表的时候，我们调用`mnesia:create_table(Name, Args)`函数。这里的`Args`是由`{Key, Val}`这样的元组组成的列表。如果表创建成功，会得到`{atomic, ok}`返回值，否则会返回`{aborted, Reason}`。

下面是最常见的`create_table`参数。

`Name`

这是表的名字（一个原子）。通常来说它的取值就是Erlang记录的名字，这个记录就是我们之前提到过的表示每行数据的记录。

`{type, Type}`

这个参数表明表的类型。`Type`是`set`、`ordered_set`或者`bag`。其意义与我们在15.2节中提到的一样。

`{disc_copies, NodeList}`

`NodeList`是Erlang节点的列表，这个表的备份会存储在这些节点上。使用这个选项时，系统会同时创建内存表和磁盘表。

一个节点上有一个`disc_copies`类型的备份表，同时另一个节点上可能有以不同类型存储的

同一个表。这个配置提供了如下梦幻般的特性。

(1) 读操作在内存中执行，速度飞快。

(2) 写操作时，数据被写到持久介质上。

```
{ram_copies, NodeList}
```

NodeList中的每个Erlang节点上都会有这个表的内存备份。 320

```
{disc_only_copies, NodeList}
```

NodeList中的每个Erlang节点上都会有一个这个表的磁盘备份。因为这个表不是在内存中的，所以，访问起来会慢一些。

```
{attributes, AtomList}
```

AtomList的列表表明了表中每列的名称。需要留意的是，当我们为此建立了xxx这样的Erlang记录时，我们可以简单的使用这样的语法{attribute, record_info(fields, xxx)}，而不需要显式的逐个字段的指定列名。

说明 create_table还有更多很多我们这里没有介绍到的选项。想要完整的了解这些选项，可以参考mnesia的手册。

17.5.2 表属性的常见组合

下面的例子，都用Atts来指代{attributes, ...}元组。

下面是一些常见的选项配置，覆盖了我们经常会用的各种情况。

```
mnesia:create_table(shop, [Attrs])
```

- 只在一个节点的内存中存储。
- 如果这个节点失效，表中的数据都会丢失。
- 在所有的方法中，这是最快的组合。
- 机器的物理内存要能放下整张表。

```
mnesia:create_table(shop,[Attrs,{disc_copies,[node()]}])
```

- 在一个节点上做内存+磁盘备份。
- 如果节点失效，表中的数据可以从磁盘上恢复。
- 读操作很快，写操作稍慢。
- 机器的物理内存要能放下整张表。

```
mnesia:create_table(shop, [Attrs,{disc_only_copies,[node()]}])
```

- 单个节点的磁盘备份。
- 无须考虑内存大小，可以存储大量数据。
- 比用内存备份的方式要慢一些。

```
mnesia:create_table(shop,[Attrs,{ram_copies,[node(),someOtherNode()]})
```

- 这个表在两个节点上有内存备份。
- 如果两个节点都失效了，数据会丢失。
- 机器的物理内存要能放下整张表。
- 在两个节点上都能访问这个表。

321

```
mnesia:create_table(shop,[Attrs, {disc_copies, [node(),someOtherNode()]}])
```

- 两个节点上都做磁盘备份。
- 每个节点上都可以进行恢复。
- 两个节点都失效，仍然可以恢复。

17.5.3　表的行为

当一个表在多个Erlang节点上备份时，它会尽可能的进行同步。如果某个节点失效，系统会继续保持运作，但是少了一个备份。当失效的节点再次上线时，它会与其他节点再次进行同步，备份数量得以保持。

说明　如果运行Mnesia的节点停止工作，可能会导致Mnesia的过载。比如说在笔记本电脑上运行，当笔记本休眠后重启时，Mnesia可能会有临时过载的情况发生，这会产生一些警告信息。可以直接忽略掉这些信息。

17.6　创建和初始化数据库

创建一个Mnesia数据库时，需要执行一些操作，这些操作，你只用做一次。

```
$ erl
1> mnesia:create_schema([node()]).
ok
2> init:stop().
ok
$ ls
Mnesia.nonode@nohost
```

`mnesia:create_schema(NodeList)`在`NodeList`（列表中的每个项都要求是个合法的Erlang节点）中的所有节点上执行一个新Mnesia数据库的初始化动作（图17-1就是表查看器的初始化屏幕）。在我们的例子中，我们给出的节点列表是`[node()]`，也就是说当前节点。`Mnesia`初始化之后会创建一个名为`Mnesia.nonode@nohost`的目录结构。我们可以退出Erlang的交互环境，操作系统的`ls`命令来验证这一点。

如果我们在一个名为`joe`的分布式节点上重复这些动作，我们得到的输出是这样的。

```
$ erl -name joe
(joe@doris.myerl.example.com) 1> mnesia:create_schema([node()]).
mnesia:create_schema([node()]).
ok
```

322

```
2> init:stop().
ok
$ ls
Mnesia.joe@doris.myerl.example.com
```

我们还可以在Erlang启动的时候指向一个特定的数据库：

```
$ erl  -mnesia dir '"/home/joe/some/path/to/Mnesia.company"'
1> mnesia:create_schema([node()]).
ok
2> init:stop().
ok
```

这里/home/joe/some/path/to/Mnesia.company是存放数据库的目录的名字。

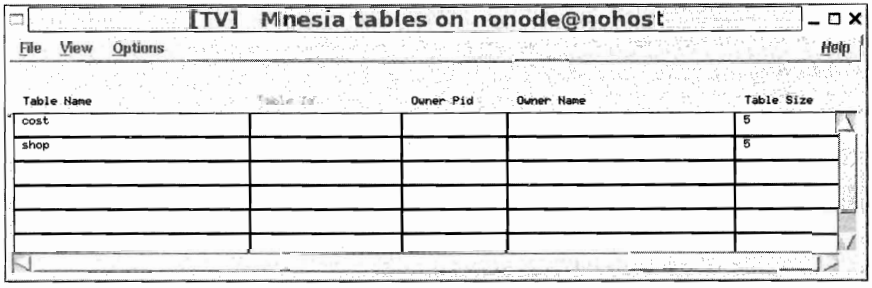


图17-1 表查看器的初始化屏幕

17.7 表查看器

Erlang自带一个用来查看Mnesia和ETS表的GUI工具，叫做表查看器。可以用tv:start()来启动这个工具。你会看到类似图17-1这样的初始化屏幕。要查看Mnesia中的表，你可以从菜单中选择View→Tab项。图17-2中的界面，就是shop表显示在表查看器中的样子。

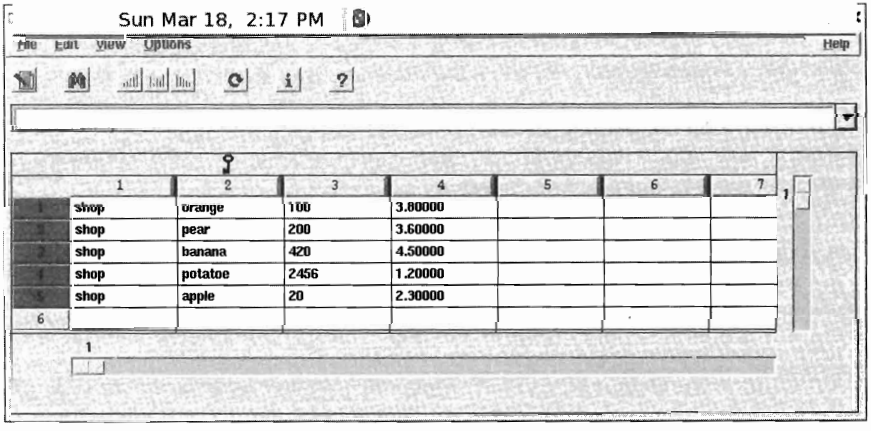


图17-2 表查看器

17.8 进一步深入

不知道我上面的文字是不是已经吊起了你对Mnesia的兴趣。Mnesia是非常强大的数据库管理
323 系统。从1998年起，它就和Ericsson的一些电信核心系统一起投入实际运营。

我们这里只是给出一些使用Mnesia中最常见的例子，关于Mnesia，可以再写一本书。因为这毕竟还是一本关于Erlang的书，那么，关于这一话题，我们也就只能言尽于此了。需要说明的是，这章展示的只是我自己用到的一些技术，实际上，还有很多我没有实际使用过（甚至还不理解）的Mnesia技术没有展现在这里。不过我想单是这些就已经够你玩上半天的了，要搭建一个成熟的应用系统，也已经完全没问题。

我没有提及的主要是这么几个方面。

- 备份和恢复：Mnesia有一大堆的配置是和备份相关的，这些配置让你能从各种类型的灾难中恢复数据。
- 脏操作：Mnesia运行进行一系列的脏操作（`dirty_read`、`dirty_write`等）。这些操作在事务之外进行。这些操作比较危险，但如果你很确信自己在干什么的话，也可以使用。比如，你的应用程序是单线程的或者什么其他的特殊情况。使用脏操作通常是从效率角度考虑。
- SNMP表：Mnesia有内建的SNMP表类型，这使得实现SNMP管理系统变成一件异乎寻常的简单任务。

Mnesia的权威参考是Erlang官方站点的《Mnesia用户指南》（参考附录C）。除此之外，在Erlang的发行版本中，也有一些Mnesia的使用例子位于examples子目录之下（在我的机器上，这个目录
324 是`/usr/local/lib/erlang/lib/mnesia-X.Y.Z/examples`）。

17.9 代码清单

test_mnesia.erl

```
-module(test_mnesia).
-import(lists, [foreach/2]).
-compile(export_all).

%% IMPORTANT: The next line must be included
%%            if we want to call qlc:q(...)

-include_lib("stdlib/include/qlc.hrl").

-record(shop, {item, quantity, cost}).
-record(cost, {name, price}).
-record(design, {id, plan}).

do_this_once() ->
    mnesia:create_schema([node()]),
    mnesia:start(),
```

```
    mnesia:create_table(shop,   [{attributes, record_info(fields, shop)}]),
    mnesia:create_table(cost,   [{attributes, record_info(fields, cost)}]),
    mnesia:create_table(design, [{attributes, record_info(fields, design)}]),
    mnesia:stop().

start() ->
    mnesia:start(),
    mnesia:wait_for_tables([shop,cost,design], 20000).


%% SQL equivalent
%%  SELECT * FROM shop;

demo(select_shop) ->
    do(qlc:q([X || X <- mnesia:table(shop)])).


%% SQL equivalent
%%  SELECT item, quantity FROM shop;

demo(select_some) ->
    do(qlc:q([{X#shop.item, X#shop.quantity} || X <- mnesia:table(shop)])).


%% SQL equivalent
%%   SELECT shop.item FROM shop
%%   WHERE  shop.quantity < 250;

demo(reorder) ->
    do(qlc:q([X#shop.item || X <- mnesia:table(shop),
                             X#shop.quantity < 250
                                ])).
%% SQL equivalent
%%   SELECT shop.item, shop.quantity, cost.name, cost.price
%%   FROM shop, cost
%%   WHERE shop.item = cost.name
%%     AND cost.price < 2
%%     AND shop.quantity < 250

demo(join) ->
    do(qlc:q([X#shop.item || X <- mnesia:table(shop),
                             X#shop.quantity < 250,
                             Y <- mnesia:table(cost),
                             X#shop.item =:= Y#cost.name,
                             Y#cost.price < 2
                                ])).

do(Q) ->
    F = fun() -> qlc:e(Q) end,
```

325

```
    {atomic, Val} = mnesia:transaction(F),
    Val.


example_tables() ->
    [%% The shop table
     {shop, apple,   20,   2.3},
     {shop, orange,  100,  3.8},
     {shop, pear,    200,  3.6},
     {shop, banana,  420,  4.5},
     {shop, potato,  2456, 1.2},
     %% The cost table
     {cost, apple,   1.5},
     {cost, orange,  2.4},
     {cost, pear,    2.2},
     {cost, banana,  1.5},
     {cost, potato,  0.6}
    ].

add_shop_item(Name, Quantity, Cost) ->
    Row = #shop{item=Name, quantity=Quantity, cost=Cost},
    F = fun() ->
            mnesia:write(Row)
        end,
    mnesia:transaction(F).

remove_shop_item(Item) ->
    Oid = {shop, Item},
    F = fun() ->
            mnesia:delete(Oid)
        end,
326 mnesia:transaction(F).
farmer(Nwant) ->
    %% Nwant = Number of oranges the farmer wants to buy
    F = fun() ->
            %% find the number of apples
            [Apple] = mnesia:read({shop,apple}),
            Napples = Apple#shop.quantity,
            Apple1  = Apple#shop{quantity = Napples + 2*Nwant},
            %% update the database
            mnesia:write(Apple1),
            %% find the number of oranges
            [Orange] = mnesia:read({shop,orange}),
            NOranges = Orange#shop.quantity,
            if
                NOranges >= Nwant ->
                    N1 =  NOranges - Nwant,
                    Orange1 = Orange#shop{quantity=N1},
                    %% update the database
```

```
                    mnesia:write(Orange1);
                true ->
                    %% Oops -- not enough oranges
                    mnesia:abort(oranges)
            end
        end,
    mnesia:transaction(F).


reset_tables() ->
    mnesia:clear_table(shop),
    mnesia:clear_table(cost),
    F = fun() ->
            foreach(fun mnesia:write/1, example_tables())
        end,
    mnesia:transaction(F).



add_plans() ->
    D1 = #design{id   = {joe,1},
                 plan = {circle,10}},
    D2 = #design{id   = fred,
                 plan = {rectangle,10,5}},
    D3 = #design{id   = {jane,{house,23}},
                 plan = {house,
                         [{floor,1,
                           [{doors,3},
                            {windows,12},
                            {rooms,5}]},
                          {floor,2,
                           [{doors,2},
                            {rooms,4},
                            {windows,15}]}]}},
    F = fun() ->
                mnesia:write(D1),
                mnesia:write(D2),
                mnesia:write(D3)
        end,
    mnesia:transaction(F).

get_plan(PlanId) ->
    F = fun() -> mnesia:read({design, PlanId}) end,
    mnesia:transaction(F).
```

327

328

第18章 构造基于OTP的系统

在本章中，我们要为一个Web公司构造一个后端系统。这间公司卖两样东西：素数以及面积计算。客户可以从我们这里买素数，或者让我们帮他计算几何物体的面积。我们假设这两项核心业务有着不可限量的发展潜力。

我们要构建两个服务器，一个用来产生素数，另一个用来计算面积。这里我们会用到gen_server框架，在16.2节已经讨论过它了。

构建这样的系统，我们必须认真对待错误。即便是最彻底的测试，也无法保证找出了所有的bug。需要考虑，万一某个服务器出现了致命的错误而导致崩溃，这个时候应该如何处理？是的，在下面的例子中，为了模拟这种情况，我们会在代码中刻意置入一个错误，来确保服务器一定会崩溃。

毫无疑问，需要有一个机制来保证，当服务器进程崩溃时，能够检测到这种情况，并且重启服务器进程。这正是我们引入监控树（supervision tree）概念的意义所在。我们会建立一个监管进程守护我们的服务器进程，如果发现这些进程崩溃，就立即重启它们。

当然，如果一个服务器进程崩溃了，我们很显然也需要知道它发生的原因，以便于日后对这一问题进行改进。要记录所有错误，我们将使用OTP的错误日志记录器。这就需要我们了解如何配置错误日志，以及如何从错误日志产生报告。

计算素数的时候，尤其是计算非常大的素数时，我们的CPU很有可能会过热。要防止这点，我们需要打开风扇，对着它猛吹，这也就意味着我们需要考虑警报。我们会通过OTP的事件处理框架来生成和处理警报。

这些问题（创建服务器进程、监控服务器进程、错误日志以及警报检测）是任何一个生产系统都必须要面对和解决的典型问题。尤其是考虑到这间公司有着不可限量的未来，我们也可以把这里的架构在更多的系统中重用。实际情况也是如此，这一架构已经被大量商业公司成功应用于自己的产品之中。

最终，当一切都准备妥当的时候，我们可以把代码打包到一个OTP的应用程序中去。通过这种特殊的方式，我们可以把与某个特定问题相关的所有东西全部都组织在一起，以便于通过OTP系统本身来启动、停止以及进行管理。

这里呈现出来的概念确实有点让人眼花缭乱，而且它们彼此之间的关系也有些微妙。这里存

在所谓的循环依赖问题。错误日志是事件管理的一种特例，警报本身就是事件，而错误日志本身是一个受到监控的进程，而监控进程本身又可以调用错误日志。

情况如此的错综复杂，这里我也只能生硬地按照自己的理解来定义一些顺序，使得这些概念的讲解更有条理一些。后面的内容中，我们会做下面这些事情。

(1) 了解一下通用事件处理程序的基本概念。
(2) 了解错误日志的工作方式。
(3) 加入警报管理。
(4) 写两个应用服务器进程。
(5) 写一个监控树，把服务器进程加进去。
(6) 把所有这些东西打包到一个应用程序中去。

18.1 通用的事件处理

一个事件表明发生了某件事——通常来说，这样的事情是程序员认为很值得注意的事情，也意味着我们要对此做点什么。

在我们编码的时候，当这类值得注意的事件发生时，我们只需要发送一条event消息给某个已注册的进程，比如：

```
RegProcName ! {event, E}
```

E就是这个事件（可以是任意的Erlang语句），RegProcName是某个已注册进程的名字。 330

我们并不知道（也不关心）这个消息发出去之后会发生什么事情。我们已经做完了自己的本职工作，而且告诉其他人某件事发生了。

现在，我们来关注这个收到事件消息的进程，它被称作是事件处理程序。最简单的事件处理程序是一个“啥也不干”的处理程序，当它接到一个{event, X}这样的事件时，它不对事件进行任何处理，而是直接丢掉。

下面是我们的第一个通用事件处理程序：

event_handler.erl

```
-module(event_handler).
-export([make/1, add_handler/2, event/2]).

%% make a new event handler called Name
%%  the handler function is noOp -- so we do nothing with the event
make(Name) ->
    register(Name, spawn(fun() -> my_handler(fun no_op/1) end)).

add_handler(Name, Fun) -> Name ! {add, Fun}.

%% generate an event
event(Name, X) -> Name ! {event, X}.

my_handler(Fun) ->
```

```
    receive
        {add, Fun1} ->
            my_handler(Fun1);
        {event, Any} ->
            (catch Fun(Any)),
            my_handler(Fun)
    end.

no_op(_) -> void.
```

事件处理程序API是这样的。

`event_handler:make(Name)`

创建了一个“啥也不干”的事件处理程序，它的名字是Name（一个原子）。这提供了一个可以发送事件的位置。

`event_handler:event(Name, X)`

向Name事件处理程序发送一个X事件。

`event_handler:add_handler(Name, Fun)`

给Name事件处理程序增加一个事件处理方法Fun。现在，当事件X发生的时候，事件处理程
331 序会执行Fun(X)。

现在我们就来建立一个事件处理程序，并产生一个错误事件：

```
1> event_handler:make(errors).
true
2> event_handler:event(errors, hi).
{event,hi}
```

是的，啥也没发生，因为我们没有在事件处理程序中安装回调模块。

要让事件处理程序做点事，我们需要写一个回调模块，然后把它安装到事件处理程序中去。下面的代码就是一个事件处理程序回调模块：

motor_controller.erl

```
-module(motor_controller).

-export([add_event_handler/0]).

add_event_handler() ->
    event_handler:add_handler(errors, fun controller/1).

controller(too_hot) ->
    io:format("Turn off the motor~n");
controller(X) ->
    io:format("~w ignored event: ~p~n",[?MODULE, X]).
```

编译，然后安装它：

```
3> c(motor_controller).
{ok,motor_controller}
```

```
4> motor_controller:add_event_handler().
{add,#Fun<motor_controller.0.99476749>}
```

现在，我们来给这个处理程序发送一个事件，它应该被motor_controller:controller/1函数处理。

```
5> event_handler:event(errors, cool).
motor_controller ignored event: cool
{event,cool}
6> event_handler:event(errors, too_hot).
Turn off the motor
{event,too_hot}
```

这个练习的要点是什么呢？首先，程序提供了一个可以发送事件的名字。这里它是注册为errors的一个进程。然后我们定义了向这个进程发送消息时所使用的协议。值得留意的一点是，我们这里并没有明确当事件到达的时候要做什么动作。实际上，这个时候做的动作就是noOp(X)，它就是“什么也不做”的动作。之后，在下一步，程序安装了一个自定义的事件处理程序。 332

“非常延迟的绑定”与“需求变更”

假设写一个函数，把event_handler:event例程对程序员隐藏起来。比如说，看起来像这样的代码：

lib_misc.erl

```
too_hot() ->
    event_handler:event(errors, too_hot).
```

然后告诉程序员，当错误发生的时候，让他们去调用lib_misc:too_hot()。对于too_hot的调用，在大多数程序设计语言中，这个过程要么是静态的，要么是从调用代码动态链接到这个函数上。一旦链接建立起来，每次调用这个函数都会执行你的代码所描述的固定任务。如果之后我们改主意了，想在这个事件发生时做些别的，这个时候我们会发现，要想改变系统的行为并没有很容易的方法（尤其是当它正在运行时）。

Erlang处理事件的方法全然不同。它允许我们解耦“事件的产生”和“对事件的处理”这两个过程，将其完全分离。我们可以在任何需要的时候改变事件的处理，这个改变的过程已经简化到了“只需要给事件处理程序发送一个新的事件处理函数”的程度。在这种处理方式下，没有什么东西是静态链接的，随时可以换用别的事件处理。

使用这个机制，我们可以构造一个可以随着时间推移不断演化的系统，而演化的过程无需停机，甚至也不需要升级代码。

说明 这不是所谓的“延迟绑定”，而是“非常延迟的绑定”，而且在此之后你仍然可以继续“变更需求”。

到这儿，你恐怕会有点疑惑。为什么我们要在这里讨论事件处理程序呢？其实，所谓的事件处理程序，最为核心的概念就在于它提供了一个基础架构，我们可以安装自定义的处理程序。

错误日志的基础架构也遵循这一模式。这也就意味着我们同样可以在出错日志中安装各种处理程序，以便在事件发生的时候做不同的事情。除此之外，警报处理系统也遵循着相同的架构模式。

18.2　错误日志

OTP系统已经内置了一个可自定义的错误日志模块。我们可以从3种不同的视角来看错误日志。
333 程序员的视角关注代码中要记录一个错误日志的函数调用；配置的视角关注错误日志如何存储以及被保存在哪里；报告的视角则关心错误发生之后，如何进行分析。我们将逐条讲述这些内容。

18.2.1　记录一个错误

在程序员的视角，错误日志的API很简单，下面是这些API的一部分。

```
@spec error_logger:error_msg(String) -> ok
```

向错误日志发送一个错误消息。

```
1> error_logger:error_msg("An error has occurred\n").
=ERROR REPORT==== 28-Mar-2007::10:46:28 ===
An error has occurred
ok
```

```
@spec error_logger:error_msg(Format, Data) -> ok
```

向错误日志发送一个错误消息它的参数与io:format(Format, Data)函数的参数一样。

```
2> error_logger:error_msg("~s, an error has occurred\n", ["Joe"]).
=ERROR REPORT==== 28-Mar-2007::10:47:09 ===
Joe, an error has occurred
ok
```

```
@spec error_logger:error_report(Report) -> ok
```

向错误日志发送一个标准错误报告。

- `@type Report = [{Tag, Data} | term()] | string() | term()]`
- `@type Tag = term()`
- `@type Data = term()`

```
3> error_logger:error_report([{tag1,data1},a_term,{tag2,data}]).
=ERROR REPORT==== 28-Mar-2007::10:51:51 ===
tag1: data1
a_term
tag2: data
```

需要说明的是，这只是可用错误日志API中的一小部分。详细讨论这些API没啥意思，下面的例子中，我们也只会用到error_msg。完整的细节可以参考手册中error_logger的部分。

18.2.2　配置错误日志

可以对错误日志进行多种配置。在Erlang shell中我们可以看到所有的错误信息（如果我们什

么都不配，这就是默认的配置）。我们可以把在shell中报告的所有错误写到一个格式化的文本文件中。此外，我们还能创建一个循环日志。你可以把循环日志想象成一个由错误日志产生的巨大环形缓存区域。新的消息进来时，会把它加到日志的尾部，如果日志被装满了，最早的条目就会被清除。 334

循环日志是一个极有用的特性。你可以决定总共有多少个日志文件，以及单个日志文件的大小。系统负责帮你删除旧文件以及创建新文件，以维持整个巨大的环形数据区域。你可以调整日志大小，用来记录最近几天的操作，这在大多数情况下都绰绰有余了。

1. 标准错误日志

启动Erlang的时候，我们可以给系统设置如下一些启动参数。

```
$ erl -boot start_clean
```

这会创建一个适合程序开发的环境，只会提供错误日志的简单形式（不带启动参数的erl命令效果等同于erl -boot start_clean）。

```
$erl -boot start_sasl
```

这会创建一个适合产品化系统的环境。SASL是System Architecture Support Libraries的缩写，它负责错误日志、过载保护等。

谁也没法记住日志记录器的所有参数（也没这个必要），所以，日志文件的配置最好从配置文件中完成。下面的小节，我们会看一下默认配置下系统中的错误日志是如何工作的，然后了解在4种典型的配置方式下，错误日志不同的工作方式。

2. 不进行配置的SASL

这是在启动SASL时，不进行配置的情况：

```
$ erl -boot start_sasl
Erlang (BEAM) emulator version 5.5.3 [async-threads:0]  ...

=PROGRESS REPORT==== 27-Mar-2007::11:49:12 ===
supervisor: {local,sasl_safe_sup}
   started: [{pid,<0.32.0>},
             {name,alarm_handler},
             {mfa,{alarm_handler,start_link,[]}},
             {restart_type,permanent},
             {shutdown,2000},
             {child_type,worker}]

... many lines removed ...
Eshell V5.5.3  (abort with ^G)
```

335

现在我们调用error_logger的方法来报告错误：

```
1> error_logger:error_msg("This is an error\n").
=ERROR REPORT==== 27-Mar-2007::11:53:08 ===
This is an error
ok
```

注意，错误是在Erlang shell中报告出来的，错误报告取决于错误日志记录器的配置。

3. 控制记录何种日志

错误日志记录器会产生下面几种类型的报告。

- 监管报告。在监管进程启动或者停止被监管的进程时，会产生这个报告（我们在18.5节中会讨论它）。
- 进程报告。每次OTP监管进程启动或者停止的时候会产生这个报告。
- 崩溃报告。当被监管的进程退出时，如果它的退出原因不是normal或者shutdown，就会产生这个报告。

这3种报告是自动产生的，程序员无需关心。除此之外，当程序显式调用error_handler的方法时，也会产生3种日志报告。通过这3种报告，程序可以记录错误、警报以及提示信息。在这里，这3个术语并没有什么特别的语意，仅仅是程序员可以使用的3种标签，用来标明错误日志条目的自然属性（也就是说，想怎么用，随你）。

在日后对错误日志进行分析的时候，我们可以用这3种标签协助我们来检查问题，过滤日志条目。在对日志进行配置的时候，我们也可以指定，比如，只保存错误，其他的信息不予保存。下面我们就来写一个这样的配置文件。

elog1.config

```
%% no tty
[{sasl, [
         {sasl_error_logger, false}
        ]}].
```

336

如果用这个配置文件启动系统，只有错误报告会被记录，进程报告之类的全部被忽略掉。而且，所有的错误报告都在shell当中。

```
$ erl -boot start_sasl -config elog1
1> error_logger:error_msg("This is an error\n").
=ERROR REPORT==== 27-Mar-2007::11:53:08 ===
This is an error
ok
```

4. 文本文件和shell

下一个配置文件列出了shell中的错误报告，同时在shell中报告的所有东西的副本也生成一个文件：

elog2.config

```
%% single text file - minimal tty
[{sasl, [
         %% All reports go to this file
         {sasl_error_logger, {file, "/home/joe/error_logs/THELOG"}}
        ]}].
```

我们启动Erlang，生成一些错误信息，然后看看日志文件以检查这个配置文件的效果。

```
$ erl -boot start_sasl -config elog2
1> error_logger:error_msg("This is an error\n").
=ERROR REPORT==== 27-Mar-2007::11:53:08 ===
This is an error ok
```

我们可以查看/home/joe/error_logs/THELOG文件的内容，应该是这样的：

```
=PROGRESS REPORT==== 28-Mar-2007::11:30:55 ===
supervisor: {local,sasl_safe_sup}
   started: [{pid,<0.34.0>},
             {name,alarm_handler},
             {mfa,{alarm_handler,start_link,[]}},
             {restart_type,permanent},
             {shutdown,2000},
             {child_type,worker}]
...
```

5. 循环日志和shell

这个配置下，日志会在shell以及一个循环日志中同时输出，这是一个很有用的配置。 337

elog3.config

```
%% rotating log and minimal tty
[{sasl, [
         {sasl_error_logger, false},
         %% define the parameters of the rotating log
         %% the log file directory
         {error_logger_mf_dir,"/home/joe/error_logs"},
         %% # bytes per logfile
         {error_logger_mf_maxbytes,10485760}, % 10 MB
         %% maximum number of logfiles
         {error_logger_mf_maxfiles, 10}
        ]}].
```

```
$erl -boot start_sasl -config elog3
1> error_logger:error_msg("This is an error\n").
=ERROR REPORT==== 28-Mar-2007::11:36:19 ===
This is an error
false
```

当用这个配置运行系统的时候，所有的错误也会输出到这个循环日志中。本章后面的章节，我们会看到从日志中提取错误信息的方法。

6. 产品化环境

在产品化环境中，我们只关注错误，而对进程和信息日志不感兴趣。所有我们在配置中让日志记录器仅仅记录错误，以免这些信息淹没在大量的进程和信息日志中。

elog4.config

```
%% rotating log and errors
[{sasl, [
         %% minimise shell error logging
         {sasl_error_logger, false},
```

```
        %% only report errors
        {errlog_type, error},
        %% define the parameters of the rotating log
        %% the log file directory
        {error_logger_mf_dir,"/home/joe/error_logs"},
        %% # bytes per logfile
        {error_logger_mf_maxbytes,10485760}, % 10 MB
        %% maximum number of
        {error_logger_mf_maxfiles, 10}
       ]}].
```

运行的时候，这个配置与前一个配置的效果看起来很相似，区别在于，只有错误会被记录到日志中。

338

18.2.3　分析错误

读取错误日志是rb模块的职责，它的接口相当简单。

```
1> rb:help().
Report Browser Tool - usage
===========================
rb:start()         - start the rb_server with default options
rb:start(Options)  - where Options is a list of:
                     {start_log, FileName}
                       - default: standard_io
                     {max, MaxNoOfReports}
                       - MaxNoOfReports should be an integer or 'all'
                       - default: all
    ...
    ... many lines omitted ...
    ...
```

我们启动报告浏览器的时候可以指定它需要读取多少行（这个例子中，是最后的20行）。

```
2> rb:start([{max,20}]).
rb: reading report...done.
3> rb:list().
No            Type      Process  Date       Time
==            ====      =======  ====       ====
11            progress  <0.29.0> 2007-03-28 11:34:31
10            progress  <0.29.0> 2007-03-28 11:34:31
9             progress  <0.29.0> 2007-03-28 11:34:31
8             progress  <0.29.0> 2007-03-28 11:34:31
7             progress  <0.22.0> 2007-03-28 11:34:31
6             progress  <0.29.0> 2007-03-28 11:35:53
5             progress  <0.29.0> 2007-03-28 11:35:53
4             progress  <0.29.0> 2007-03-28 11:35:53
3             progress  <0.29.0> 2007-03-28 11:35:53
2             progress  <0.22.0> 2007-03-28 11:35:53
1             error     <0.23.0> 2007-03-28 11:36:19
ok
```

```
> rb:show(1).

ERROR REPORT  <0.40.0>                    2007-03-28 11:36:19
==========================================================

This is an error
ok
```

要隔离一个特定的错误，我们可以使用rb:grep(RegExp)命令。它会找出所有匹配这个正则表达式的记录。这里我就不深入分析错误日志的细节问题了，学习它的最好方法是花一点时间，自己来和rb进行一些交互，看看都能做点什么。注意，你根本不需要自己来删日志，循环机制会 339
删除旧的日志。

如果你想要保留所有的日志，那么你就需要每隔一段特定的时间过来抓取，并自己进行清理。

18.3 警报管理

在我们的应用程序中，我们只需要一个警报，在CPU融化之前，必须要发出这个警报。因为CPU要进行高强度的大素数计算（还记得吗？我们公司的业务之一就是计算素数，然后卖出去）。这一次我们要动真格的了，使用真正的OTP警报处理程序（而不是本章开头那样简单的东西）。

警报处理程序是一个遵循gen_event行为准则的回调模块，代码是这样的：

my_alarm_handler.erl

```
-module(my_alarm_handler).
-behaviour(gen_event).

%% gen_event callbacks
-export([init/1, handle_event/2, handle_call/2,
         handle_info/2, terminate/2]).

%% init(Args) must return {ok, State}
init(Args) ->
    io:format("*** my_alarm_handler init:~p~n",[Args]),
    {ok, 0}.

handle_event({set_alarm, tooHot}, N) ->
    error_logger:error_msg("*** Tell the Engineer to turn on the fan~n"),
    {ok, N+1};
handle_event({clear_alarm, tooHot}, N) ->
    error_logger:error_msg("*** Danger over. Turn off the fan~n"),
    {ok, N};
handle_event(Event, N) ->
    io:format("*** unmatched event:~p~n",[Event]),
    {ok, N}.

handle_call(_Request, N) -> Reply = N, {ok, N,  N}.
```

```
handle_info(_Info, N)    -> {ok, N}.

terminate(_Reason, _N)   -> ok.
```

340

这些代码看起来和我们在16.3节中看到的gen_server回调函数非常相似。handle_event(Event, State)是这里有意思的部分，它应该返回{ok, NewState}。这里的Event是一个{EventType, EventArg}这样的元组，其中，EventType可以是set_event或clear_event，EventArg是用户提供的参数。我们稍后会看到这些消息是怎么产生的。

现在可以来点有趣的了。我们会启动系统、生成一个警报、安装一个警报处理程序、生成一个新警报等：

```
$ erl -boot start_sasl -config elog3
1> alarm_handler:set_alarm(tooHot).
ok
=INFO REPORT==== 28-Mar-2007::14:20:06 ===
alarm_handler: {set,tooHot}

2> gen_event:swap_handler(alarm_handler,
                          {alarm_handler, swap},
                          {my_alarm_handler, xyz}).

*** my_alarm_handler init:{xyz,{alarm_handler,[tooHot]}}
3> alarm_handler:set_alarm(tooHot).
ok
=ERROR REPORT==== 28-Mar-2007::14:22:19 ===
*** Tell the Engineer to turn on the fan
4> alarm_handler:clear_alarm(tooHot).
ok
=ERROR REPORT==== 28-Mar-2007::14:22:39 ===
*** Danger over. Turn off the fan
```

这里都做了些什么呢？

(1) 我们使用-boot start_sasl启动了Erlang，这么做会有一个标准的警报处理程序。当我们设置或者清除警报的时候，什么事也不会发生。这和我们之前讨论过的“啥也不干”的事件处理程序很类似。

(2) 当设置一个警报（第1行），我们得到了一个信息日志。此时，我们并没有对警报做什么特别的处理。

(3) 我们安装了一个自定义的警报处理程序（第2行）。在这里，传给my_alarm_handler的参数（xyz）并没有什么特别的意义。这个语法需要一些值，但我们并没有用值而只是用了一些原子xyz。从输出的日志中，可以认出它来。

341

*** my_alarm_handler_init: ... 是回调模块输出的。

(4) 我们设置然后又清除了tooHot的警报（第3行和第4行）。这一次由我们自定义的警报处理程序来处理。只需要观察shell输出就能确认这一点。

读取日志

让我们回到错误日志，看看发生了什么：

```
1> rb:start([{max,20}]).
rb: reading report...done.
2> rb:list().
No                Type    Process  Date       Time
==                ====    =======  ====       ====
...
3         info_report    <0.29.0> 2007-03-28 14:20:06
2               error    <0.29.0> 2007-03-28 14:22:19
1               error    <0.29.0> 2007-03-28 14:22:39
3> rb:show(1).

ERROR REPORT  <0.33.0>            2007-03-28 14:22:39
==================================================
*** Danger over. Turn off the fan
ok
4> rb:show(2).
ERROR REPORT  <0.33.0>            2007-03-28 14:22:19
==================================================
*** Tell the Engineer to turn on the fan
```

这下清楚了吧，错误日志机制就是这么工作的。

实际操作中，我们需要确定设定的日志大小足够容纳几天或者几个星期的操作日志。每隔几天（或者几个星期）我们都需要检查日志，调查每一个错误。

说明 rb模块中，有选择特定错误类型，并将其输出到一个文件的函数。有了这样的工具，你可以将分析错误日志的工作完全自动化。

18.4 应用服务

我们的应用程序有两个服务，一个是素数服务，另一个是面积服务。下面看到的代码就是素数服务，它是遵循gen_server行为来编写的（参考16.2节相关内容）。注意其中是如何整合我们上一节提到的警报处理的。

342

18.4.1 素数服务

prime_server.erl

```
-module(prime_server).
-behaviour(gen_server).

-export([new_prime/1, start_link/0]).

%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).
```

```
start_link() ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).

new_prime(N) ->
    %% 20000 is a timeout (ms)
    gen_server:call(?MODULE, {prime, N}, 20000).

init([]) ->
    %% Note we must set trap_exit = true if we
    %% want terminate/2 to be called when the application
    %% is stopped
    process_flag(trap_exit, true),
    io:format("~p starting~n",[?MODULE]),
    {ok, 0}.

handle_call({prime, K}, _From, N) ->
    {reply, make_new_prime(K), N+1}.

handle_cast(_Msg, N)  -> {noreply, N}.

handle_info(_Info, N)  -> {noreply, N}.

terminate(_Reason, _N) ->
    io:format("~p stopping~n",[?MODULE]),
    ok.

code_change(_OldVsn, N, _Extra) -> {ok, N}.

make_new_prime(K) ->
    if
        K > 100 ->
            alarm_handler:set_alarm(tooHot),
            N = lib_primes:make_prime(K),
            alarm_handler:clear_alarm(tooHot),
            N;
        true ->
            lib_primes:make_prime(K)
    end.
```

343

18.4.2　面积服务

接着是面积服务，它也是遵照gen_server行为模式来写的。值得留意的是，以这种方式来写一个服务是非常快的。在写这个例子的时候，我实际上就是剪切的素数服务的代码，将其改为一个面积服务，只花了几分钟就完成了。

面积服务中规中矩没啥技巧的。而它还包含一个故意留下的错误（你发现了吗？），我的如意算盘是要让它崩溃在你眼前，然后又通过监控进程在你眼前神奇般的重启。更进一步，我们还会得到一个包含了所有这些情况的错误日志。

area_server.erl

```
-module(area_server).
-behaviour(gen_server).

-export([area/1, start_link/0]).

%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

start_link() ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).

area(Thing) ->
    gen_server:call(?MODULE, {area, Thing}).

init([]) ->
    %% Note we must set trap_exit = true if we
    %% want terminate/2 to be called when the application
    %% is stopped
    process_flag(trap_exit, true),
    io:format("~p starting~n",[?MODULE]),
    {ok, 0}.

handle_call({area, Thing}, _From, N) -> {reply, compute_area(Thing), N+1}.

handle_cast(_Msg, N)  -> {noreply, N}.

handle_info(_Info, N)  -> {noreply, N}.

terminate(_Reason, _N) ->
    io:format("~p stopping~n",[?MODULE]),
    ok.

code_change(_OldVsn, N, _Extra) -> {ok, N}.

compute_area({square, X})      -> X*X;
compute_area({rectonge, X, Y}) -> X*Y.
```

344

18.5 监控树

监控树是由进程形成的一棵树。树的顶端是监控进程，它负责监控下方的工作进程，如果工作进程崩溃了，它就会执行重启操作。监控树有两种类型，你可以在图18-1中看到。

- one-for-one监控树。在one-for-one监控方式下，如果一个进程失效，它的监控进程就会重启它。
- all-for-one监控树。在all-for-one监控方式下，如果一个进程失效，所有的工作进程都会被终止（通过调用回调模块的terminate/2函数）。然后所有的工作进程又重新启动。

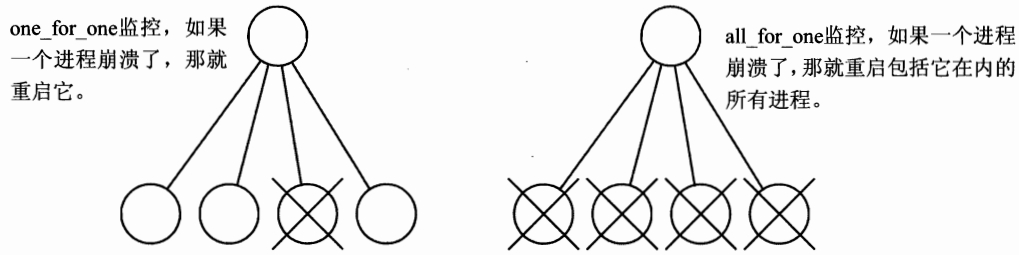


图18-1　两种类型的监控树

使用OTP的监控行为模式可以创建一个监控进程。可以通过一个回调模块来制定监控的策略，与此同时，也要指定如何启动每一个工作进程的方法。监控树通过下面这种形式的函数来描述：

```
init(...) ->
    {ok, {RestartStrategy, MaxRestarts, Time},
         [Worker1, Worker2, ...]}.
```

345

这里的RestartStrategy可以是one_for_one或者all_for_one。MaxRestarts和Time设定了一个“重启频率”。如果一个监控进程在Time时间内执行了超过MaxRestarts次重启，那么监控进程会终止所有的工作进程，然后退出自己。这种机制是为了避免出现这种情况，一个进程崩溃，然后重启，然后因为相同的原因又发生崩溃，如此不断反复，形成一个死循环。

Worker1、Worker2等是描述如何启动每一个工作进程的元组。我们马上就会看到它们长什么样子。

好了，有了这些知识，我们可以回到我们的公司，构建一个监控树。

我们要做的第一件事就是要为监控进程选一个名字。比如说，叫“sellaprime”，sellaprime监控进程的工作就是要保证这两个服务器进程一直处于运行状态。要达成这一目标，我们还需要写一个回调模块。这一次要遵循gen_supervisor行为模式。这就是我们的回调模块：

sellaprime_supervisor.erl

```
-module(sellaprime_supervisor).
-behaviour(supervisor).          % see erl -man supervisor

-export([start/0, start_in_shell_for_testing/0, start_link/1, init/1]).

start() ->
    spawn(fun() ->
                supervisor:start_link({local,?MODULE}, ?MODULE, _Arg = [])
          end).

start_in_shell_for_testing() ->
    {ok, Pid} = supervisor:start_link({local,?MODULE}, ?MODULE, _Arg = []),
    unlink(Pid).
```

```
start_link(Args) ->
    supervisor:start_link({local,?MODULE}, ?MODULE, Args).

init([]) ->
    %% Install my personal error handler
     gen_event:swap_handler(alarm_handler,
                                {alarm_handler, swap},
                                {my_alarm_handler, xyz}),
    {ok, {{one_for_one, 3, 10},
          [{tag1,
            {area_server, start_link, []},
            permanent,
            10000,
            worker,
            [area_server]},
 {tag2,
  {prime_server, start_link, []},
  permanent,
  10000,
  worker,
  [prime_server]}
]}}.
```

346

这里最关键的部分是init/1函数返回的数据结构：

sellaprime_supervisor.erl

```
{ok, {{one_for_one, 3, 10},
      [{tag1,
        {area_server, start_link, []},
        permanent,
        10000,
        worker,
        [area_server]},
       {tag2,
        {prime_server, start_link, []},
        permanent,
        10000,
        worker,
        [prime_server]}
      ]}}.
```

这个数据结构定义了一个监控策略。前面我们已经讨论过监控策略和重启频率的问题了，现在剩下来的也就是素数和面积服务器进程的启动描述。

工作进程的描述是这样的元组：

```
{Tag, {Mod, Func, ArgList},
       Restart,
       Shutdown,
       Type,
       [Mod1]}
```

这样参数的含义如下所示。

Tag

这是一个原子，用来标记之后要如何来表示这个工作进程（如果有必要的话）。

{Mod, Func, ArgList}

这定义了监控进程是如何启动工作进程的。它会继续传递给apply(Mod, Fun, ArgList)。

Restart = permanent | transient | temporary

347 一个永久进程每次崩溃都会进行重启，一个瞬态进程只有在它非正常退出的情况下进行重启（退出值不是一个合法的正常退出值），一个临时进程则不会进行重启。

Shutdown

这是一个终止时间。这是一个工作进程从它启动到它终止之间允许运行的时间。如果超过这个时间仍在运行，就会将其终止。（可以有其他的值，具体参考supervisor的用户手册）。

Type = worker | supervisor

这是被监控进程的类型。我们可以将一个监控进程置于另外一个监控进程的控制之下，并由此构建一个更复杂的监控树。

[Mod1]

如果子进程是一个监控进程或gen_server行为模式的回调函数，这个名字指定了回调模块的名称（可以有其他的值，具体参考supervisor的用户手册）

这些参数看起来挺吓人的，实则不然。实际应用中，你大可以从前面的代码中复制粘贴这些值到代码中，然后稍作修改，以符合你的模块名称，这在大多数情况下都够用了。

18.6　启动整个系统

现在好了，我们已经准备就绪，期待已久的好戏就要上演。我们的公司“闪亮登场”了，谁要买第一个素数？

我们把系统启动起来：

```
$ erl -boot start_sasl -config elog3
1> sellaprime_supervisor:start_in_shell_for_testing().
*** my_alarm_handler init:{xyz,{alarm_handler,[]}}
area_server starting
prime_server starting
```

然后做一个合法的查询：

```
2> area_server:area({square,10}).
100
```

再来一个不合法的查询：

```
3> area_server:area({rectangle,10,20}).
area_server stopping

=ERROR REPORT==== 28-Mar-2007::15:15:54 ===
** Generic server area_server terminating
```

```
** Last message in was {area,{rectangle,10,20}}
** When Server state == 1
** Reason for termination ==
** {function_clause,[{area_server,compute_area,[{rectangle,10,20}]},
                     {area_server,handle_call,3},
                     {gen_server,handle_msg,6},
                     {proc_lib,init_p,5}]}
  area_server starting
  ** exited: {{function_clause,
                [{area_server,compute_area,[{rectangle,10,20}]},
                 {area_server,handle_call,3},
                 {gen_server,handle_msg,6},
                 {proc_lib,init_p,5}]},
                 {gen_server,call,
                   [area_server,{area,{rectangle,10,20}}]}} **
```

348

监控策略管用吗

Erlang是设计用来构造容错系统的。它最初是由瑞典电信公司Ericsson的计算机实验室开发出来的。此后，Ericsson的OTP项目组继续开发这一技术，以帮助很多内部用户。通过使用gen_server、gen_supervisor之类的技术，用Erlang实现过达到99.9999999%（9个9）可用性的系统。合理使用错误处理机制，可以让你的程序永久运行（好吧，近乎于永久运行）。在实际的产品中，我们之前描述过的错误日志运行了很多年。

喔哦，发生了什么事？面积服务器进程崩溃了，我们碰到了故意埋下的错误。监控进程已经检测到了这个崩溃，并重启了面积服务器进程。这些在日志中都有记录。

崩溃之后，一切都恢复了正常，就像什么都没发生过一样。我们再来试试一个合法的查询：

```
4> area_server:area({square,25}).
625
```

一切运转良好！现在，我们再来生成一个小素数：

```
5> prime_server:new_prime(20).
Generating a 20 digit prime ........
37864328602551726491
```

我们再来生成一个大素数：

349

```
6> prime_server:new_prime(120).
Generating a 120 digit prime
=ERROR REPORT==== 28-Mar-2007::15:22:17 ===
*** Tell the Engineer to turn on the fan
....................................

=ERROR REPORT==== 28-Mar-2007::15:22:20 ===
*** Danger over. Turn off the fan
765525474077993399589034417231006593110007130279318737419683
288059079481951097205184294443332300308877493399942800723107
```

现在我们有了一个可用的系统。如果服务器进程崩溃，会自动进行重启，而错误的情况会记

录在日志当中。

我们来看看错误日志：

```
1> rb:start([{max,20}]).
rb: reading report...done.
rb: reading report...done.
{ok,<0.53.0>}
2> rb:list().
No                Type      Process Date       Time
==                ====      ======= ====       ====
20            progress     <0.29.0> 2007-03-28 15:05:15
19            progress     <0.22.0> 2007-03-28 15:05:15
18            progress     <0.23.0> 2007-03-28 15:05:21
17   supervisor_report     <0.23.0> 2007-03-28 15:05:21
16               error     <0.23.0> 2007-03-28 15:07:07
15               error     <0.23.0> 2007-03-28 15:07:23
14               error     <0.23.0> 2007-03-28 15:07:41
13            progress     <0.29.0> 2007-03-28 15:15:07
12            progress     <0.29.0> 2007-03-28 15:15:07
11            progress     <0.29.0> 2007-03-28 15:15:07
10            progress     <0.29.0> 2007-03-28 15:15:07
9             progress     <0.22.0> 2007-03-28 15:15:07
8             progress     <0.23.0> 2007-03-28 15:15:13
7             progress     <0.23.0> 2007-03-28 15:15:13
6                error     <0.23.0> 2007-03-28 15:15:54
5         crash_report  area_server 2007-03-28 15:15:54
4    supervisor_report     <0.23.0> 2007-03-28 15:15:54
3             progress     <0.23.0> 2007-03-28 15:15:54
2                error     <0.29.0> 2007-03-28 15:22:17
1                error     <0.29.0> 2007-03-28 15:22:20
```

350 这里有问题，我们有一个面积服务器进程的崩溃报告。发生了什么事呢？（假装还不知道啦。）

```
9> rb:show(5).

CRASH REPORT  <0.43.0>                          2007-03-28 15:15:54
==============================================================
Crashing process
pid                                                      <0.43.0>
registered_name                                       area_server
error_info
{function_clause,[{area_server,compute_area,[{rectangle,10,20}]},
                  {area_server,handle_call,3},
                     {gen_server,handle_msg,6},
                     {proc_lib,init_p,5}]}
initial_call
    {gen,init_it,
        [gen_server,
         <0.42.0>,
         <0.42.0>,
         {local,area_server},
```

```
        area_server,
        [],
        []]}
ancestors                    [sellaprime_supervisor,<0.40.0>]
messages                                                  []
links                                              [<0.42.0>]
dictionary                                                []
trap_exit                                              false
status                                               running
heap_size                                                233
stack_size                                                21
reductions                                               199
ok
```

{function_clause, compute_area, ...}这一句输出告诉了我们程序崩溃发生的确切位置，定位问题并进行改正并不复杂。我们继续往下看。

```
10> rb:show(2).

ERROR REPORT  <0.33.0>                        2007-03-28 15:22:17
==============================================================
*** Tell the Engineer to turn on the fan
```

以及

```
10> rb:show(1).

ERROR REPORT  <0.33.0>                        2007-03-28 15:22:20
==============================================================
*** Danger over. Turn off the fan
```

这是计算大素数时产生的风扇警报！

351

18.7 应用程序

我们已经非常接近于完成状态了，现在要做的是写一个.app文件，用来包含应用程序的信息：

sellaprime.app

```
%% This is the application resource file (.app file) for the 'base'
%% application.
{application, sellaprime,
 [{description, "The Prime Number Shop"},
  {vsn, "1.0"},
  {modules, [sellaprime_app, sellaprime_supervisor, area_server,
             prime_server, lib_primes, my_alarm_handler]},
  {registered,[area_server, prime_server, sellaprime_super]},
  {applications, [kernel,stdlib]},
  {mod, {sellaprime_app,[]}},
  {start_phases, []}
 ]}.
```

然后我们要写一个上述文件已经提到的回调模块（sellaprime_app）：

sellaprime_app.erl

```
-module(sellaprime_app).
-behaviour(application).
-export([start/2, stop/1]).

%%-----------------------------------------------------------------
%% Function: start(Type, StartArgs) -> {ok, Pid} |
%%                                     {ok, Pid, State} |
%%                                     {error, Reason}
%% Description: This function is called whenever an application
%% is started using application:start/1,2, and should start the processes
%% of the application. If the application is structured according to the
%% OTP design principles as a supervision tree, this means starting the
%% top supervisor of the tree.
%%-----------------------------------------------------------------

start(_Type, StartArgs) ->
    sellaprime_supervisor:start_link(StartArgs).

%%-----------------------------------------------------------------
%% Function: stop(State) -> void()
%% Description: This function is called whenever an application
%% has stopped. It is intended to be the opposite of Module:start/2 and
%% should do any necessary cleaning up. The return value is ignored.
%%-----------------------------------------------------------------

stop(_State) ->
    ok.
```

352

我们需要公开start/2和stop/1函数。做完这些，我们可以从shell中这样启动和停止应用程序：

```
$ erl -boot start_sasl -config elog3
1> application:loaded_applications().
[{kernel,"ERTS  CXC 138 10","2.11.3"},
 {stdlib,"ERTS  CXC 138 10","1.14.3"},
 {sasl,"SASL  CXC 138 11","2.1.4"}]
2> application:load(sellaprime).
ok
3> application:loaded_applications().
[{sellaprime,"The Prime Number Shop","1.0"},
 {kernel,"ERTS  CXC 138 10","2.11.3"},
 {stdlib,"ERTS  CXC 138 10","1.14.3"},
 {sasl,"SASL  CXC 138 11","2.1.4"}]
4> application:start(sellaprime).
*** my_alarm_handler init:{xyz,{alarm_handler,[]}}
area_server starting
```

```
prime_server starting
ok
5> application:stop(sellaprime).
prime_server stopping
area_server stopping

=INFO REPORT==== 2-Apr-2007::19:34:44 ===
application: sellaprime
exited: stopped
type: temporary
ok
6> application:unload(sellaprime).
ok
7> application:loaded_applications().
[{kernel,"ERTS  CXC 138 10","2.11.4"},
 {stdlib,"ERTS  CXC 138 10","1.14.4"},
 {sasl,"SASL  CXC 138 11","2.1.5"}]
```

这就是一个完备的OTP应用程序。第2行，我们加载了应用程序，加载动作会载入所有的代码，但不会启动应用程序。第4行启动了应用程序，第5行将其停止。注意，我们启动和终止应用程序的时候，那些输出信息表明，在面积服务器进程和素数服务器进程之中的相关回调函数都被正确的调用了。第6行，我们卸载了应用程序，应用程序中所有模块的代码都被清除。

在用OTP构造复杂系统的时候，我们通常将其打包为应用程序。这使得我们可以统一的启动、终止和进行管理。

注意，在我们使用`init:stop()`来关闭系统的时候，所有正在运行的应用程序都会以一定的顺序关闭。 353

```
$ erl -boot start_sasl -config elog3
1>  application:start(sellaprime).
*** my_alarm_handler init:{xyz,{alarm_handler,[]}}
area_server starting
prime_server starting
ok
2> init:stop().
ok
prime_server stopping
area_server stopping
$
```

第2个命令后面的两行输出分别来自面积服务器进程和素数服务器进程，这表明`gen_server`回调模块中的`terminator/2`函数被调用了。[①]

18.8 文件系统的组织

我还没有提及任何关于文件组织的内容，这是有意为之，我的意图是尽量让你每次只需要面

① 它确实回来了！（还记得“终结者”那章吗？）。

对一个问题。

结构良好的OTP应用程序，它各个部分的文件通常会分开放在各个约定好的地方。但这并不是必须的，因为相关的文件也可以在运行时定位，这与我们这里讨论的文件组织没有关系。

在这本书中，我把几乎所有的例子文件都放在同一个目录中。这简化了例子，也避免了在不同的程序间切换时因为搜索路径而造成麻烦。

在sellaprime公司的例子中主要用到的文件如表18-1所示。

表　18-1

文　件	内　容
area_server.erl	面积服务——gen_server回调
prime_server.erl	素数服务——gen_server回调
sellaprim_supervisor.erl	监控回调
sellaprim_app.erl	应用程序回调
my_alam_handler.erl	gen_event的事件回调
sellaprime.app	应用程序规范
elog4.config	错误日志配置文件

354 要了解这些文件和模块是如何使用的，我们可以考察启动应用程序的时候事件的发生顺序。

(1) 我们用这样的命令启动系统：

```
$ erl -boot start_sasl -config elog4.config
1> application:start(sellaprime).
...
```

sellaprime.app文件必须在Erlang的根目录或者其中的一个子目录中。

然后，会查找sellaprime.app中的{mod, ...}声明，这包含了应用程序控制器的名称。在我们的例子中，就是sellaprime_app模块。

(2) sellaprime_app:start/2回调函数被调用。

(3) sellaprime_app:start/2 调用了sellaprime_supervisor:start_link/2，这启动了sellaprime的监控进程。

(4) 监控回调sellaprime_supervisor:init/1被调用。这安装了一个错误处理程序，并返回了一个监控规范，这个规范描述了如何启动面积服务器进程和素数服务器进程。

(5) sellaprime监控进程启动面积服务器进程和素数服务器进程，它们都是gen_server的回调模块。

停止这些非常简单，我们只需要调用application:stop(sellaprime)或者init:stop()就可以了。

18.9　应用程序监视器

应用程序监视器是一个图形界面的查看程序，使用appmon:start()可以启动它。输入这个命令后，你会看到一个类似于图18-2的窗口。点击，就可以查看其中一个应用的监控树。图18-3

就是显示sellaprime应用程序时的界面。

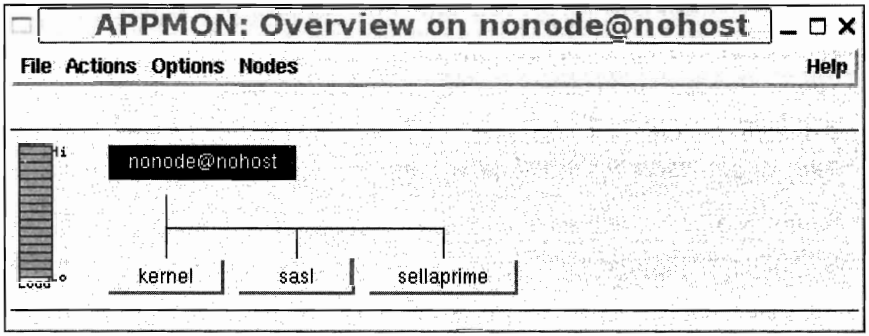


图18-2 应用监视器初始窗口

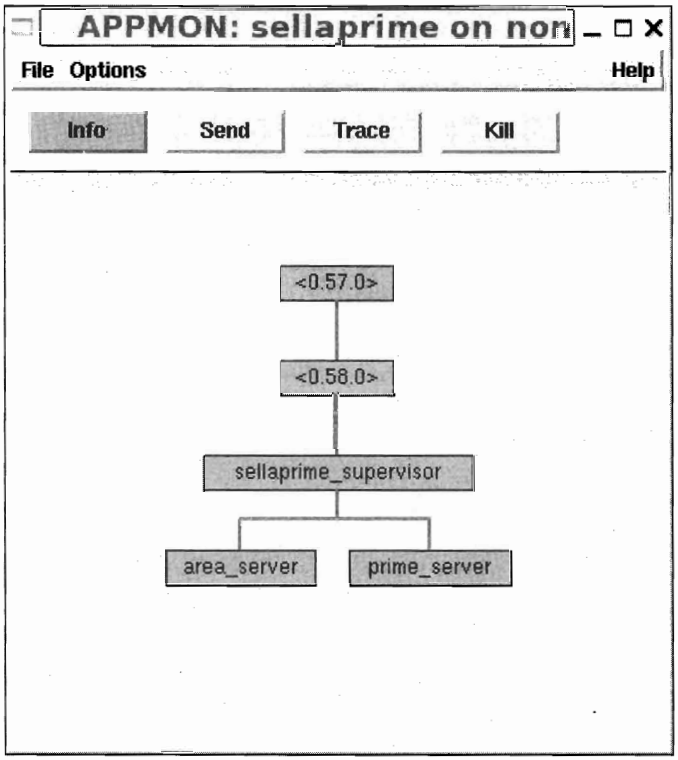


图18-3 sellaprime应用程序

18.10 进一步深入

我们这里跳过了很多细节，仅提及了我们涉及的一小部分。你可以从gen_event、error_logger、supervisor以及application的手册中获取更详细的介绍。

下面这些文件也包含更多关于如何使用OTP行为模式的细节。

http://www.erlang.org/doc/pdf/design_principles.pdf

(97页) 涉及Gen servers、gen event、supervisors。

http://www.erlang.org/doc/pdf/system_principles.pdf

(19页) 如何创建一个启动文件。

http://www.erlang.org/doc/pdf/appmon.pdf

(16页) 应用程序监视器。

18.11　我们如何创建素数

很简单。

lib_primes.erl

```
%% make a prime with at least K decimal digits.
%% Here we use 'Bertrand's postulate.
%% Bertrands postulate is that for every N > 3,
%% there is a prime P satisfying N < P < 2N - 2
%% This was proved by Tchebychef in 1850
%% (Erdos improved this proof in 1932)


make_prime(1) ->
    lists:nth(random:uniform(5), [1,2,3,5,7]);
make_prime(K) when K > 0 ->
    new_seed(),
    N = make_random_int(K),
    if N > 3 ->
            io:format("Generating a ~w digit prime ",[K]),
            MaxTries = N - 3,
            P1 = make_prime(MaxTries, N+1),
            io:format("~n",[]),
            P1;
        true ->
            make_prime(K)
    end.


make_prime(0, _) ->
    exit(impossible);
make_prime(K, P) ->
    io:format(".",[]),
    case is_prime(P) of
        true  -> P;
        false -> make_prime(K-1, P+1)
    end.
```

```
%% Fermat's little theorem says that if
%% N is a prime and if A < N then
%% A^N mod N = A

is_prime(D) ->
    new_seed(),
    is_prime(D, 100).

is_prime(D, Ntests) ->
    N = length(integer_to_list(D)) -1,
    is_prime(Ntests, D, N).

is_prime(0, _, _) -> true;
is_prime(Ntest, N, Len) ->
    K = random:uniform(Len),
    %% A is a random number less than N
    A = make_random_int(K),
    if
        A < N ->
            case lib_lin:pow(A,N,N) of
                A -> is_prime(Ntest-1,N,Len);
                _ -> false
            end;
        true ->
            is_prime(Ntest, N, Len)
    end.
```

```
1> lib_primes:make_prime(500).
Generating a 500 digit prime ..............................
791015726987201027909055597115096126908592921342508297266243912
125926314028552834613243970133079247710947860309449739469644043
439969675871437494053122242294696670762292613938500209657830906
062534166780603261012226023459181325555764028306928844115181391
107802007557066746476035515105154017421267382367314941956505578
474497545252666718280976890401503018406521440650857349061213980
678938094352667372672691906693169783133618111423622890401868042
872198074546193740053777668271056036892838181730070340565057841
53
```

第19章

多核小引

如何写出能在多核CPU上运行得更快的程序？这完全取决于可变状态以及并发特性。多年以前（20多年前）机器世界就已经有了下面两种并发模型。

- 基于状态共享的并发。
- 基于消息传递的并发。

主流的程序界走了其中的一条路（基于状态共享的并发路线），而Erlang社区走了另外一条（还有为数不多的另外几种语言也走了这条“基于消息传递的并发”路线，这些语言是Oz和Occam）。

在基于消息传递的并发中，并不存在被共享的状态，所有的计算都通过进程来完成。交换数据的唯一方法就是通过异步消息来传递。

优势何在？

状态共享并发与“可变状态”的概念密切相关（从语义上说，这意味着可以改变内存中的数据）——所有的语言，诸如C、Java、C++等，在它们之中都有被称作“状态”的概念，我们的程序要负责改变状态。

在只有一个进程来对“状态”进行改动的时候，这没有什么问题。

如果有多个进程共享同一个状态，而且这些进程都需要对同一块内存进行改动的时候，麻烦就来了。

要防止对共享内存的并发修改，常常会用到锁机制。无论把这一技术叫做信号量、同步方法，

还是什么别的名字，它的实质始终还是锁。

如果程序在关键区域崩溃（当它们掌握锁的时候），麻烦就大了，其他的所有程序都无法再获得锁。如果程序破坏了共享内存，同样也会是一场灾难，其他的程序也不可能知道该怎么处理。

程序员要如何解决这类问题？非常困难。连在单核处理器上能够正常工作的程序，直接放在多核环境下运行，都可能要出问题（除非禁用多核）。

针对这一问题有多种方案（事务内存可能是其中最好的）但它们充其量也只算是一种补救措施。最糟糕的情况下，它们会让事情变得更加棘手。

Erlang没有可变的数据结构：[①]

- 没有可变的数据结构意味着没有锁；

① 这个说法并不十分确切，但大致如此。

- 没有可变的数据结构也意味着更容易并行化。

我们怎样来做并行？很简单，程序员将整个方案拆分成若干个并发进程。

这种编程风格有自己的术语，叫做“面向并发的编程”（缩写为COP，与OOP相对应）。

现在我们进入本书的最后一章，在这里我们会看到，多核CPU上的程序是如何运作的。 360

第20章 多核编程

对于Erlang的程序员来说，有一个不同寻常的好消息，那就是，无需任何修改，你的程序就能在*n*核的CPU上跑得快*n*倍。

不过前提是你必须遵循以下这些简单的规则。

你的程序要由多个进程构成，这些进程之间彼此不会互相冲突，而且程序逻辑也不存在顺序瓶颈。是的，只要遵循这些规则，你的程序就能在多核CPU上跑得更快。

若非如此，比如，你写了一堆庞杂的单进程顺序代码，压根没用上任何的spawn语句来创建并发进程，那么就不要期望它能在多核的环境下跑得更快。

不过，天无绝人之路，就算你的代码不幸就是如此，那也不用失落，因为只要稍做修改，它也能迅速变成轻灵快捷的并发版本。

在本章中，我们会涉及下面这些话题。

- 要让程序在多核CPU上运行，需要做些什么？
- 如何并行化顺序代码。
- 顺序瓶颈问题。
- 如何避免副作用。

做完上面这些，再来考虑更复杂的情况下，如何设计。我们会实现一个名为mapreduce（映射-归并算法）的高阶函数，并把它应用到全文检索搜索引擎中去。mapreduce是Google为在大量元素上执行并发计算而开发的一种抽象算法。

为什么要关心多核CPU

你可能很疑惑，多核CPU，这又有什么大不了的？难道说不并行化我们的程序，在多核系统上就不能运行了不成？答案是肯定的。现如今，双核的CPU已经是标配，在我工作的实验室，4核的CPU也已经屡见不鲜，时不时的也会用到32核的机器。

如果一个程序在双核的系统上能快上一倍，这好像不怎么吸引人（好吧，如果你坚持的话，我承认，还是有那么一点点吸引人）。不过我们也不得不承认，双核CPU的主频比单核的要慢，所以，性能的增长也许不会那么明显。

Intel公司有一个Keifer项目，它的目标是要在2009/2010年面向市场推出自己的32核处理器。而Sun公司则早已向市场推出了自己的Niagra计算机，它有8个核，每个核有4个硬件超线程。

如果两倍的性能提升不过瘾，那么10倍，甚至100倍的性能提升呢？应该足够让人欢呼雀跃的了吧。现代处理器的运算速度已经极其迅猛，单核上已足以运行4个超线程。换句话说，一个32核的CPU可以提供等同于128线程的运算能力。是的，100倍的速度增长就在眼前。

100倍！这确实让人兴奋不已。

需要我们做的，就是写好代码。

20.1 如何在多核的CPU上更有效率地运行

想要运行得更高效，我们必须努力达到下列标准。

(1) 使用大量进程。

(2) 避免副作用。

(3) 避免顺序瓶颈。

(4) 以“少量消息、大量运算”的方式写代码。

如果达到了上面这些标准，我们的Erlang程序就可以在多核CPU上运行得更高效。 362

20.1.1 使用大量进程

这很重要——别让CPU闲着，要让所有的CPU在所有的时间都在忙碌地工作。达到这一目标最简便易行的办法就是使用大量的进程。

这里所说的“大量进程”，是相对于CPU的数量的。如果我们有大量的进程，那么就无须担心会有CPU闲着。这看起来是一个纯粹统计学上的结论。如果我们只是少量地使用进程，那么，可能碰巧在单核CPU上会略有优势，但单核CPU的这一微弱优势在成千上百的进程面前就会荡然无存。虽说现在的主流机器中只有为数不多的几个核心，但如果希望程序能够面向未来，那么就需要考虑仅在一个芯片上就有几千个核的工作场景。

进程之间的工作量应该差别不大。如果一个进程要做大量的工作，而另一个进程只做一点点工作，这肯定不是个好主意。

在很多应用程序中，无须精心设计，我们就会有大量的进程。如果应用程序本身就是“内在并发”的，那么我们就无须考虑如何对代码进行并发化的处理。比如说，假如我们在写一个消息系统，它要管理几万个并发连接，那么，从这几万个并发的连接上，我们可以获得天然的并发特性。至少，处理每一个连接的代码是无须考虑并发问题的。

20.1.2 避免副作用

副作用是并发的大敌。本书一开始，我们就讨论了“不发生变化的变量”。这是理解为什么在多核CPU上Erlang程序会比用有“破坏性赋值”的语言写出来的程序运行得更快的关键。

在一个有共享内存和线程的语言中，当两个线程同时向同一块内存进行写入时，就会发生非常棘手的情况。在一个进程进行写入操作时，系统会锁定共享内存，这会阻止另一个进程并发的写操作。这些锁的概念通常会对程序员隐藏起来，而是以互斥或者同步方法的方式出现在编程语言之中。共享内存的主要问题是一个线程有可能破坏另一个线程要用到的内存。也就是说，即便
363 程序是正确的，另外一个线程仍然有可能破坏数据结构，从而导致程序崩溃。

Erlang没有共享内存，所以不存在这个问题。实际上这个说法并非完全确切，Erlang中只有两种形式的共享内存，都与共享的ETS或DETS表有关，而且，就算有问题也很容易避免。

共享的ETS或DETS表

ETS表可以被多个进程共享，在15.4节中，我们讨论了集中创建ETS表的方法。`ets:new`的一个选项允许我们创建一个`public`表。回忆一下。

创建一个public表，所有知道这个表标识的进程都可以读写这个表。

这很危险，除非是下面的情况。

- 我们可以保证每次只有一个进程会对这个表进行写操作，其他的所有进程只进行读操作。
- 向ETS表做写入操作的进程是正确的，而且不会写入非法的数据。

这些属性通常是无法由系统保证的，而必须要依赖于程序的逻辑。

说明1　在ETS表上的每一个操作都是原子的。不可能在ETS像一个原子一样执行一系列操作。虽然我们不破坏ETS表中的数据，但如果多个进程未经协调地并发更新一个共享的表，仍然有可能导致表中的数据在逻辑上的不一致。

说明2　`protected`类型的ETS表则要安全得多。只有一个进程（它的创建者）可以写入数据，但有多个进程可以读数据。这个特性是由系统保证的。但是请记住，即便只有一个进程可以向ETS表写入数据，仍然有可能出问题，比如，当这个进程破坏表中的数据时，所有需要读取数据的进程都会受到影响。

如果要使用ETS表，那么`private`类型会是最安全的。显而易见，DETS也与此类似。我们可能会创建一个共享的DETS表，并允许多个不同的进程执行写操作。应该避免这样使用。

说明　ETS和DETS是为了实现Mnesia而创建的，在一开始就不是为了单独使用而准备的。如果
364 应用程序要在多个进程之间并发地使用共享内存，应该使用Mnesia的事务机制。

20.1.3　顺序瓶颈

做完了程序的并发化改写，获得了大量的进程，而且也没有共享内存的操作，那么下一个需要思考的问题就是顺序瓶颈。总有一些东西具有内在的顺序性。实际上，如果顺序性是问题的本质，那么我们也没法把它去掉。特定的事件以特定的顺序发生，如果实际的情况就是如此，那么无论我们怎么努力，也没法改变这一顺序，正如我们的生老病死，这是一个无法改变的顺序，我们没法把这类事情也并发化。

顺序瓶颈是当多个并发的进程需要访问一个顺序的资源时，自然产生的瓶颈，典型的例子就是IO。通常我们只有一个磁盘，所有对这个磁盘的输出最终都是顺序的。要知道，磁盘上只装了一组磁头，我们没法改变这一事实。

每次我们创建并注册一个进程时，我们实际上也创建了一个潜在的顺序瓶颈，所以，尽量避免使用注册进程。如果我们必须要创建这样的注册进程并把它当作服务器来使用，那么就尽量确保它对所有的请求能尽可能快地进行反馈。

通常来说，解决顺序瓶颈的唯一方法是从算法层面着手改进。这通常都不是一件轻松的事，而且往往代价很高。我们必须将一个不是分布式的算法改成一个分布式的算法。关于这个话题（分布算法）学术界已经进行了深入的研究，有大量的研究成果，但在通常的程序设计语言库中却鲜有实际应用。造成这一状况的主要原因是，对于这类算法的需求目前还没有浮现出来。这一情形目前已经发生改变，因为现在我们的编程越来越多涉及网络计算和多核计算环境。

因特网和多核CPU在计算机编程中如影随形，这迫使我们必须去深入了解这些论文，并着手去实现其中的一些令人叹为观止的算法。

一个分布式的订票系统

假设我们有个单一的资源——Strolling Bones乐队下次演唱会的门票。要保证买票的时候最终能够得到一张门票，我们通常会使用一个票务代理进程来掌握所有的门票。但这引入了一个顺序瓶颈，要怎么避免呢？

很简单，假设你有两个票务代理。开始售票的时候，第一个代理负责所有单号的门票，第二个则负责所有双号的门票。这就能保证代理不会把同一张票卖出去两次。 365

如果其中一个代理已经卖完了所有的票，那么他就从另外一个代理那里拿一堆票。

我并没有说这是一个完美的方案，比如，看演唱会的时候，你可能希望和你的朋友们坐在一起，这意味着六个连续的座位。但这解决了顺序瓶颈的问题——使用两个售票窗口，而不是一个。

同理，我们可以把单一的票务代理变成*n*个，进一步，*n*也可以随时间变化，每一个代理可以加入和离开整个网络，还可以随时崩溃。实际上，这是分布式计算领域一个非常活跃的研究领域，“分布式散列表”（distributed hash table）是对这一研究领域的称谓。如果你到google上查询这个术语，能够找到这一领域内的大量研究论文。

20.2 并行化顺序代码

记得我们之前着重讲述过的列表操作吗？尤其是`lists:map`这个函数，它的定义是这样的：

```
map(_, [])     -> [];
map(F, [H|T]) -> [F(H)|map(F, T)].
```

可以用一个简单的策略来加快这个顺序化程序的处理速度。我们把这个程序的新版本称作**pmap**，它对所有参数进行并发求值：

lib_misc.erl

```
pmap(F, L) ->
    S = self(),
```

```
    %% make_ref() returns a unique reference
    %%   we'll match on this later
    Ref = erlang:make_ref(),
    Pids = map(fun(I) ->
                    spawn(fun() -> do_f(S, Ref, F, I) end)
               end, L),
    %% gather the results
    gather(Pids, Ref).

do_f(Parent, Ref, F, I) ->
    Parent ! {self(), Ref, (catch F(I))}.

gather([Pid|T], Ref) ->
    receive
        {Pid, Ref, Ret} -> [Ret|gather(T, Ref)]
    end;
gather([], _) ->
    [].
```

366

pmap能像map一样工作，所不同的是，当我们调用pmap(F, L)的时候，它对L中的每一个参数创建了一个独立的进程进行求值。值得注意的是每一个对L中参数的求值进程，都可能会以任意的顺序结束，很显然它与这个参数在L中的顺序并没有保持对应关系。

实际上，是由在gather函数中的选择性receive语句来保证返回结果的顺序与原始参数的列表顺序一致。

在map和pmap这两个函数之间，还有一点细微的语义差别。在pmap中，我们在遍历列表的时候使用了(catch F(H))。这是为了确保在计算pmap的过程中，即便是在F(H)的计算中发生了异常也会被正确地终止。在不发生异常的情况下，这两个函数的外部行为完全是一样的。

要点　上面这句话在严格意义上讲并不正确。如果这两个函数当中会发生副作用，那么它们的行为并不会完全一样。假设F(H)有对进程字典数据的更新动作。当我们调用map的时候，更新的是调用进程的字典。

而用pmap的时候，每一个F(H)都会在它自己的进程中运算，因此，如果我们仍然做相同的动作，那么我们所做的更新会发生在这个新的进程字典上，不会影响到调用pmap这个进程。所以，请小心：有副作用的代码无法简单地通过将map替换为pmap来得到并发化。

什么时候可以用 pmap

“把代码中的map替换为pmap”这并不是提高性能时包治百病的大力丸。下面的这些问题仍然需要你消耗大量的脑细胞。

1. 并发粒度

不要在计算量很小的函数上使用pmap。比如这样的情形：

```
map(fun(I) -> 2*I end, L)
```

这里fun当中的计算量实在是太微不足道了。相比并发方式的“开启进程、等待反馈”这整个过程中的开销，我们从并发化中获得的性能优势，已经被完全抵消了。 367

2. 不要创建太多的进程

请记住，pmap(F, L)要创建length(L)这么多的并发进程。如果L巨大无比，那么，你也要创建出同等数量的进程。那么到底该用多少个进程呢？瑞典语[①]有一个单词非常适合用在这里，我们应该创建lagom[②]数量的进程。

3. 在恰当的抽象层次上思考

将一个列表映射到一个函数，可以有很多不同的方法，也许pmap并不是最合适的抽象。下面是这些可能的方法中最简单的一种。

在这个版本的pmap中，我们关心返回值的顺序（我们用选择性的receive语句来实现这一点）。如果我们不关心返回值的顺序，可以这么写：

lib_misc.erl

```
pmap1(F, L) ->
    S = self(),
    Ref = erlang:make_ref(),
    foreach(fun(I) ->
                  spawn(fun() -> do_f1(S, Ref, F, I) end)
            end, L),
    %% gather the results
    gather1(length(L), Ref, []).

do_f1(Parent, Ref, F, I) ->
    Parent ! {Ref, (catch F(I))}.

gather1(0, _, L) -> L;
gather1(N, Ref, L) ->
    receive
        {Ref, Ret} -> gather1(N-1, Ref, [Ret|L])
    end.
```

稍加改动上面的代码可以变成一个并行版的foreach。代码和上面的可能非常相似，我们只需要去掉构造返回值的语句就可以了。我们只关心程序是否运行完毕。

另外一个实现pmap的方法是限制并行的进程数，最多允许使用K个进程，K可以是一个固定的常量。当pmap需要处理非常巨大的列表时，这个特性会很有用。 368

另一个版本的pmap不仅可以把计算任务分布到处理器的其他核中去，还可以把计算任务分布到一个Erlang网络的其他节点上。

限于篇幅，这里就不展示这个版本pmap的代码了，不复杂，你完全可以卷起袖子自己做一个出来。

---

① 瑞典语lagom这个单词的意思类似于中文中的“增之一分则太长，减之一分则太短”，意为恰到好处。——译者注

② Erlang来自瑞典，在概括瑞典人的特征时，“lagom är bäst”是最常用的句子（可以大致译为“够用最好”）。

本节的目的是为了指出，通过spawn、send和receive这些基本语句的组合，可以构造出无数种并发计算抽象模型。你可以通过这些语句来为你自己的程序创建出自己的并发抽象模型。

再次强调——避免副作用是提高并发性的关键，要牢牢记住这一点。

20.3　小消息、大计算

我们已经讨论了足够多的理论，可不能光说不练，现在就来动点真格的，实际评估，怎么样？在本小节，我们要做两个实验。我们要把一个有100个元素的列表分别映射到两个函数，然后比较顺序映射和并发映射的执行时间。

我们要利用两个不同的问题集，第一个是这样的：

```
L = [L1, L2, ..., L100],
map(fun lists:sort/1, L)
```

L当中的每一个元素都是一个包含1000个随机数的列表（我们对它进行排序）。

第二个是这样：

```
L = [27,27,..., 27],
map(fun ptests:fib/1, L)
```

这里L是一个包含100个27的列表，我们来计算[fib(27), fib(27), ...] 100次（fib是斐波那契函数）。

我们会评估这两个函数的运行时间，然后把map换成pmap，并重新计时。

在第一个计算（排序）中，我们使用pmap时会通过消息在进程间发送大量的数据（一个包含1000个随机数的列表），而排序运算实际上是非常迅速的。第二个计算中，我们对每一个进程只
369 需要发送一个很小的请求（计算fib(27)），但计算fib(27)意味着很大的计算量。

在第二个问题中，由于进程之间只有少量的数据复制，却有着相当大的计算量，因此，我们期望第二个问题在多核CPU上会比第一个问题有更明显的性能提升。

实际操作层面，我们还需要一个脚本来自动运行我们的测试。不过，在此之前，我们先来看看如何启动SMP Erlang。

运行 SMP Erlang[①]

SMP[②] Erlang可以运行在多种不同的架构和操作系统之上。目前的系统可以在支持一到两个处理器的主板上和Intel的双核或4核CPU一起运行，还可以在Sun和Cavium处理器上运行。这个领域正处于极速发展之中，Erlang的每一个发行版所支持的操作系统和处理器的数量都会增加。你可以在Erlang的版本发布备注中找到最新的信息。（在Erlang下载页面 http://www.erlang.org/download.html上点击最新版的标题。）

① 目前的Erlang虚拟机实际上有3个版本：beam、beam.smp和beam.hybrid，启动时，会根据系统平台和启动参数来决定具体启动哪一个虚拟机。——译者注

② SMP是symmetric multiprocessing（对称多核处理器）的缩写，一个SMP机器中有两个或者多个独立的CPU，它们连到一个共享的内存。这些CPU可以在一个多核的芯片上，也可以是多个芯片，或者是这两种形式的组合。

说明 从R11B-0版开始，在所有已经支持的平台下，SMP Erlang都是默认打开的（也就是说，默认会建立一个带有SMP支持的虚拟机）。在其他的平台下强制打开SMP Erlang，需要在配置中加上-enable-smp-support标志。

SMP Erlang有两个命令行标志，用来决定它在多核CPU下如何运行。

```
$erl -smp +S N
```

-smp

启动 SMP Erlang。

+S N

使用N个调度器来运行Erlang。每一个Erlang调度器都是一个完整的虚拟机，它拥有所有其他虚拟机的信息。如果忽略这个参数，它就默认为SMP机器中逻辑处理器的数量。

我们为什么要改变这个参数呢？有如下几个原因。 370

- 我们希望评估不同的调度器数目（意味着不同的CPU数量）会对程序性能造成怎样的影响。
- 我们可以通过改变N的数目，从而在单核CPU上模拟多核CPU下的运行情况。
- 我们可以有比物理CPU数目更多的调度器。在一些情况下，这能显著提高系统吞吐能力，从而让系统运行状况更优。具体是何种原因导致这些效果，目前还没有全部弄清楚，这也是另外一个研究的热点领域。

要执行我们的测试，还需要一个运行脚本：

runtests

```
#!/bin/sh
echo "" >results
for i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16\
         17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
do
   echo $i
   erl -boot start_clean -noshell -smp +S $i \
       -s ptests tests $i >> results
done
```

这段脚本只是简单地用1～32的不同调度器数目来启动Erlang进行测试，它会将所有的计时结果写到一个名为results的文件之中，以供分析。

下面是我们的测试程序：

ptests.erl

```
-module(ptests).
-export([tests/1, fib/1]).
-import(lists, [map/2]).
-import(lib_misc, [pmap/2]).

tests([N]) ->
```

```
    Nsched = list_to_integer(atom_to_list(N)),
    run_tests(1, Nsched).

run_tests(N, Nsched) ->
    case test(N) of
        stop ->
            init:stop();
        Val ->
            io:format("~p.~n",[{Nsched, Val}]),
            run_tests(N+1, Nsched)
    end.
test(1) ->
    %% Make 100 lists
    %%   Each list contains 1000 random integers
    seed(),
    S = lists:seq(1,100),
    L = map(fun(_) -> mkList(1000) end, S),
    {Time1, S1} = timer:tc(lists,    map,  [fun lists:sort/1, L]),
    {Time2, S2} = timer:tc(lib_misc, pmap, [fun lists:sort/1, L]),
    {sort, Time1, Time2, equal(S1, S2)};
test(2) ->
    %% L = [27,27,27,..] 100 times
    L = lists:duplicate(100, 27),
    {Time1, S1} = timer:tc(lists,    map,  [fun ptests:fib/1, L]),
    {Time2, S2} = timer:tc(lib_misc, pmap, [fun ptests:fib/1, L]),
    {fib, Time1, Time2, equal(S1, S2)};
test(3) ->
    stop.

%% Equal is used to test that map and pmap compute the same thing
equal(S,S)   -> true;
equal(S1,S2) ->  {differ, S1, S2}.

%% recursive (inefficent) fibonacci
fib(0) -> 1;
fib(1) -> 1;
fib(N) -> fib(N-1) + fib(N-2).

%% Reset the random number generator. This is so we
%% get the same sequence of random numbers each time we run
%% the program

seed() -> random:seed(44,55,66).

%% Make a list of K random numbers
%%    Each random number in the range 1..1000000
mkList(K) -> mkList(K, []).

mkList(0, L) -> L;
mkList(N, L) -> mkList(N-1, [random:uniform(1000000)|L]).
```

371

上面的代码分别用map和pmap来运行两个不同的测试用例。我们可以在图20-1中看到结果。根据测试的时间，按比例绘制了从map和pmap分别测试到的运行时间。图中我们可以看到，小消息大计算的CPU密集型运算，随着CPU数量的增长，其性能大致呈线性趋势增长。而大消息小计算的应用随着CPU的数量增长，对性能的影响则不明显。

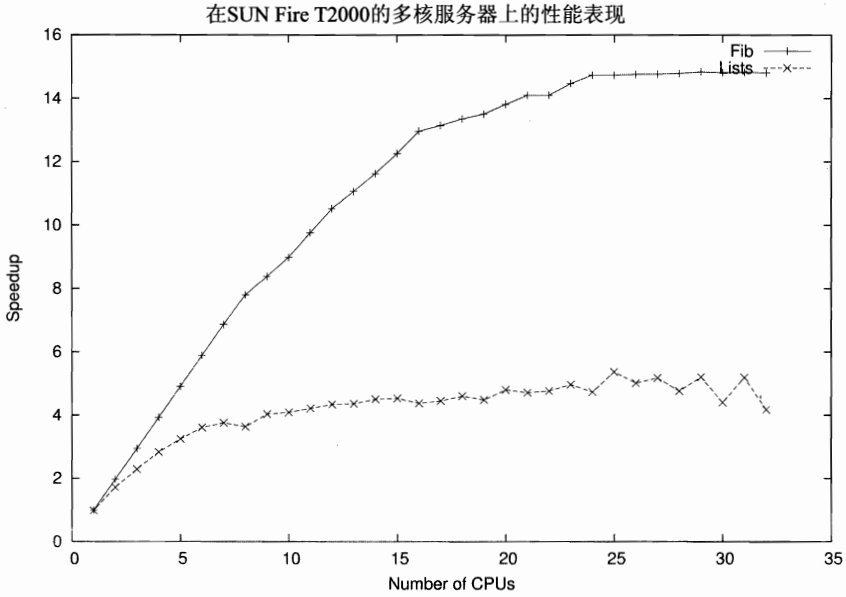


图20-1 在多核CPU上的性能提升

一图胜千言，对于这个图形我们就说这么多了。Ericsson有一款基于Erlang的商业产品，它在双核的CPU上获得了接近一倍的性能提升，对于这样的优化结果，我们都觉得深受鼓舞。最后说一句，SMP Erlang的变化日新月异，今天正确的做法，在明天可能就会变得不再正确。 372

20.4 映射-归并算法和磁盘索引程序

现在我们要从理论转向实践。首先，我们要来看看高阶函数mapreduce，然后我们会在一个简单的索引引擎中使用这种技术。在这里，我们的目标并不是要做一个世上最快最好的索引引擎，而是要通过这一技术来解决相关应用场景下真实面对的设计问题。

20.4.1 映射-归并算法

在图20-2中，向我们展示了映射-归并（map-reduce）算法的基本思想。开启一定数量的映射进程，让它们负责产生一系列的{Key, Value}这样的键-值对。映射进程把这些键-值对发送给一个归并进程，它负责合并这些键-值对，合并的方式就是把有相同键的值组合起来。

警告 这里提到的map这个词，尤其是在mapreduce的上下文中，与本书其他部分中提到的map函数全然不同，切忌混淆。

mapreduce（映射-归并算法）是由Google公司的Jeffrey Dean和Sanjay Ghemawat提出的高阶
373 并行函数，据说Google的集群中每天都要大量使用这个算法。

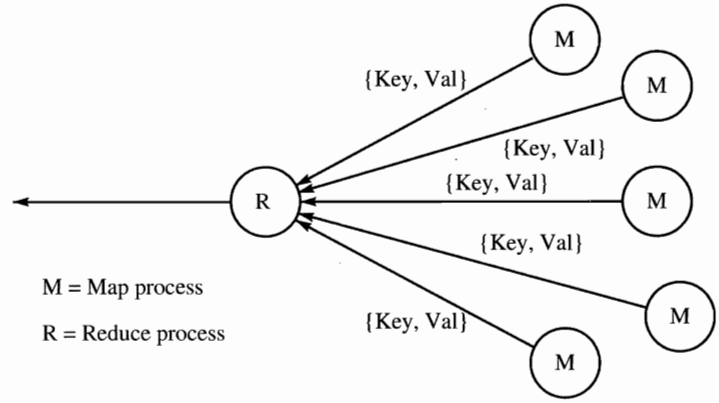


图20-2　映射-归并算法

我们可以用多种不同的方式来实现多种不同语意的映射-归并算法。这个算法与其说是一个特定的算法，还不如说是一个算法族。

mapreduce是这么定义的：

```
@spec mapreduce(F1, F2, Acc0, L) -> Acc
    F1 = fun(Pid, X) -> void,
    F2 = fun(Key, [Value], Acc0) -> Acc
    L  = [X]
    Acc = X = term()
```

- F1(Pid, X)是映射函数。F1的任务是发送一组{Key, Value}数据给Pid，然后退出。mapreduce每次会给列表中的每个X创建一个新的进程。
- F2(Key, [Value], Acc0) -> Acc是归并函数。当所有的映射函数都退出的时候，归并函数要负责针对每一个键，将它对应的所有的值合并到一起。此时，它会对每一个它收集到的{Key, [Value]}调用F2(Key, [Value], Acc)函数。Acc是一个累加器，它的初始值是Acc0。F2会返回一个新的累加器（另外一种描述方式是这样，F2在所有它收集到的
374 {Key, [Value]}对上执行一个折叠操作）。
- Acc0是累加器的初始值，当调用F2时会被使用。
- L是一个X的列表。F1(Pid, X)会对列表L中的每一个X进行运算，Pid是由mapreduce创建的归并进程的进程标识符。

mapreduce定义在phofs（parallel higher-order function的缩写）模块中：

phofs.erl

```
-module(phofs).
-export([mapreduce/4]).
```

```
-import(lists, [foreach/2]).

%% F1(Pid, X) -> sends {Key,Val} messages to Pid
%% F2(Key, [Val], AccIn) -> AccOut

mapreduce(F1, F2, Acc0, L) ->
    S = self(),
    Pid = spawn(fun() -> reduce(S, F1, F2, Acc0, L) end),
    receive
        {Pid, Result} ->
            Result
    end.

reduce(Parent, F1, F2, Acc0, L) ->
    process_flag(trap_exit, true),
    ReducePid = self(),
    %% Create the Map processes
    %%   One for each element X in L
    foreach(fun(X) ->
                    spawn_link(fun() -> do_job(ReducePid, F1, X) end)
            end, L),
    N = length(L),
    %% make a dictionary to store the Keys
    Dict0 = dict:new(),
    %% Wait for N Map processes to terminate
    Dict1 = collect_replies(N, Dict0),
    Acc = dict:fold(F2, Acc0, Dict1),
    Parent ! {self(), Acc}.

%% collect_replies(N, Dict)
%%     collect and merge {Key, Value} messages from N processes.
%%     When N processes have terminated return a dictionary
%%     of {Key, [Value]} pairs

collect_replies(0, Dict) ->
    Dict;
collect_replies(N, Dict) ->
    receive
        {Key, Val} ->
            case dict:is_key(Key, Dict) of
                true ->
                    Dict1 = dict:append(Key, Val, Dict),
                    collect_replies(N, Dict1);
                false ->
                    Dict1 = dict:store(Key,[Val], Dict),
                    collect_replies(N, Dict1)
            end;
        {'EXIT', _,  Why} ->
```

375

```
            collect_replies(N-1, Dict)
    end.

%% Call F(Pid, X)
%%   F must send {Key, Value} messsages to Pid
%%     and then terminate

do_job(ReducePid, F, X) ->
    F(ReducePid, X).
```

进一步深入之前，先来对这个**mapreduce**函数做一下测试，这样我们能更加理解它的工作机制。

我们要写一个小程序，以统计这本书所附的代码中所有单词的出现频率，代码是这样的：

test_mapreduce.erl

```
-module(test_mapreduce).
-compile(export_all).
-import(lists, [reverse/1, sort/1]).

test() ->
    wc_dir(".").

wc_dir(Dir) ->
    F1 = fun generate_words/2,
    F2 = fun count_words/3,
    Files = lib_find:files(Dir, "*.erl", false),
    L1 = phofs:mapreduce(F1, F2, [], Files),
    reverse(sort(L1)).

generate_words(Pid, File) ->
    F = fun(Word) -> Pid ! {Word, 1} end,
    lib_misc:foreachWordInFile(File, F).

count_words(Key, Vals, A) ->
    [{length(Vals), Key}|A].
```

376

```
5> test_mapreduce:test().
[{341,"1"},
 {330,"end"},
 {318,"0"},
 {265,"N"},
 {235,"X"},
 {214,"T"},
 {213,"2"},
 {205,"start"},
 {196,"L"},
 {194,"is"},
 {185,"file"},
 {177,"Pid"},
 ...
```

运行的时候，代码目录里有102个Erlang模块，因此mapreduce也就创建了102个并发进程，它们每一个都向归并进程发送由键值对组成的数据流。这在100个核心处理上应该运行得很好（如果硬盘跟得上的话）。

现在我们已经明白mapreduce是怎么回事了，可以回到索引引擎了。

20.4.2 全文检索

建立索引时，一件必须要做的事情就是找出一个文件中出现的所有单词。我们在“映射-归并”算法的“映射”阶段会用到这一点。

在此之前，我们先来看看在全文检索当中会用到的数据结构。

1. 反向索引

我们的全文检索通过反向索引来实现，在本小节中，我们需要回顾反向索引的概念，并了解它是如何存储在文件系统当中的。

为了向你展现这个进化过程，我们先从一个简单的例子开始。假设文件系统中有3个文件，每个文件都包含一些单词。

我们的文件及其内容如表20-1所示。

表 20-1

文 件 名	内 容
/home/dogs	rover jack buster winston
/home/animals/cats	zorro daisy jaguar
/home/cars	rover jaguar ford

377

为了建立这样的反向索引，首先给这些文件编号，如表20-2所示。

表 20-2

索 引	文 件 名
1	/home/dogs
2	/home/animals/cats
3	/home/cars

然后我们在此基础之上构造一个词表，将单词与其出现的文件的索引对照起来，如表20-3所示。

表 20-3

单 词	文 件 索 引
rover	1, 3
jack	1
buster	1
winston	1
zorro	2
daisy	2
jaguar	2, 3
ford	3

2. 反向索引的查询

一旦建立了反向索引，查询就是一件相当简单的事情了。比如，我们想找buster这个词，它出现在编号为1的文件中，名为/home/dogs，而查询rover AND jaguar，我们可以先查出rover（结果为文件1和文件3），再查出jaguar（结果为文件2和文件3），然后对这两个结果取交集（结果是文件3），也就是/home/cars这个文件。

3. 反向索引的数据结构

我们需要两个持久的数据结构。

- 文件名-索引对应表。在反向索引中，文件名是用整数来表示的。比如说，一个常见的词可能会在成千上万的文件中出现，使用整数来表示文件名，简化了表示形式，能大大节约空间。我们会用一个DETS表来存储这些信息。前面的15.6节中，我们已经写了一个程序来做这件事。
- 单词-文件索引表。对于每一个出现在文件之中的单词，都需要记录这个文件的索引号。这里使用文件系统来实现这个数据结构。在我们的例子中，可以创建名为rover、buster等这样的文件。索引器程序把这些单词保存在某个索引目录中。例如，如果索引目录是/user/index，我们可以在这个索引目录下找到名为buster的文件，这个索引目录包含这些文件的索引。

378

20.4.3　索引器的操作

我们通过调用indexer:start()来开始所有的操作，它的定义是这样的：

indexer-1.1/indexer.erl

```
start() ->
    indexer_server:start(output_dir()),
    spawn_link(fun() -> worker() end).
```

它做了两件事，其一，它启动一个名为indexer_server的服务器进程（这是一个用gen_server写成的服务器进程），其二，它启动了一个worker进程来执行索引动作。

indexer-1.1/indexer.erl

```
worker() ->
    possibly_stop(),
    case indexer_server:next_dir() of
        {ok, Dir} ->
            Files = indexer_misc:files_in_dir(Dir),
            index_these_files(Files),
            indexer_server:checkpoint(),
            possibly_stop(),
            sleep(10000),
            worker();
        done ->
            true
    end.
```

woker进程做了下面这些事情。

(1) 调用indexer_server:next_dir()，它会返回下一个需要索引的目录。

(2) 调用index_misc:fires_in_dir来查找目录下需要进行索引的文件。

(3) 调用index_these_files(Files)来对这些文件进行索引。

(4) 调用indexer_server:checkpoint()，这与异常恢复有关。每次索引完成一个新的目录，程序都会告诉服务器进程我们已经做完了对于这个目录的索引。如果这个程序异常退出或被停止重启，下一次调用indexer_server:next_dir()时会从上次调用的目录处恢复，继续进行。

每一个索引的周期结束，worker都会调用possibly_stop()以检查是否需要停止。如若没有，它会休眠一段时间，然后进入下一个周期。

379

实际的索引操作在index_these_files之中，这里就是我们要使用“映射－归并”算法来实现并行的地方。

indexer-1.1/indexer.erl

```
index_these_files(Files) ->
    Ets = indexer_server:ets_table(),
    OutDir = filename:join(indexer_server:outdir(), "index"),
    F1 = fun(Pid, File) -> indexer_words:words_in_file(Pid, File, Ets) end,
    F2 = fun(Key, Val, Acc) -> handle_result(Key, Val, OutDir, Acc) end,
    indexer_misc:mapreduce(F1, F2, 0, Files).

handle_result(Key, Vals, OutDir, Acc) ->
    add_to_file(OutDir, Key, Vals),
    Acc + 1.
```

add_to_file(OutDir, Word, Is)将Is中的索引数组添加到OutDir目录中的Word文件中。

indexer-1.1/indexer.erl

```
add_to_file(OutDir, Word, Is) ->
    L1 = map(fun(I) -> <<I:32>> end, Is),
    OutFile = filename:join(OutDir, Word),
    case file:open(OutFile, [write,binary,raw,append]) of
        {ok, S} ->
            file:pwrite(S, 0, L1),
            file:close(S);
        {error, E} ->
              exit({ebadFileOp, OutFile, E})
    end.
```

20.4.4 运行索引器

```
1> indexer:cold_start().
2> indexer:start().
...
N> indexer:stop().
Scheduling a stop
ack
Stopping
```

20.4.5　评论

这是一个相当复杂的程序（是整本书里最为复杂的程序），当然还谈不上完备。它虽复杂，但其结构却相当简洁明了，因而很容易理解。

它有完整的启动和停止策略，并且能够从错误中恢复。这些特性得益于indexer_checkpoint
380 模块。它用“映射-归并”算法采用并发方式来处理分词，效果让人非常满意。

这个搜索引擎目前还只是一个玩具，要变成一个功能完善的产品，我们还需要进行下面一些改进。

- 改进单词抽取。这可是一个说来容易做来难的领域，可能会耗费掉你大量的精力。问题本身并不是很复杂，但这一问题并不存在什么通用的方案。每一种类型的文件（Erlang、PDF、TXT、C、Java等）都需要用一个单独的（而且是不同的）分析技术来抽取相关的单词。针对所有的语言（以及拼写）也都需要再做这个工作。
- 映射-归并算法要进行改进，以便支持处理极大量的数据集。我们上面例子之中的键-值合并方式当数据量非常大的时候就会耗尽内存，而且也没有在磁盘上进行备份的措施。我们必须要仔细的思考在数据量极大的情况下，应该如何表示数据集。
- 反向索引的数据结构只使用了文件系统来存储数据，而文件名－索引对照表也只是使用了一个DETS表来存储，这在实际应用中毫无疑问是不够的。比如，我们要处理这个星球上所有机器的所有文件，要满足这样的要求，至少分布式的散列表是要用到的。

别气馁，google也不是一天就能写得出来的，我们把这些问题摆在一边，退一步，回顾我们的整个方案。最起码，我们学到了一些在多核CPU上编程的方法，简单地应用一些高阶函数，就避免了令人头痛的副作用。

20.4.6　索引器的代码

索引器的所有代码都在代码下载的indexer目录当中。总共有9个文件，大约1 200行代码。出于节约纸张的考虑，我这里就不再把代码列出来了。

indexer.erl

主程序，它导出了start()、stop()以及cold_start()函数。对于索引器的使用者来说，这就是需要他了解的全部操作。

indexer_porter.erl

词干分析算法，用来减少索引的单词数量。我们会把一个单词的各种变体归并为它的词干或基本形式。比如说，fishing、fished和fisher有相同的词根fish，这一过程被称作词干分析。我们这里用到的词干分析算法是Porter算法（由Martin Porter发明），indexer_porter模块实现了这一算
381 法。

indexer_server.erl

这是一个用gen_server建立起来的服务器进程。它掌管DETS表，这个表indexer_

filenames_dets也用到了。它还跟踪全局数据，比如需要索引的目录名称以及索引进度等。

indexer_filenames_dets.erl

这个模块负责从文件名到索引号的映射，它与我们之前看到的lib_filenames_dets是一样的。

indexer_checkpoint.erl

这个模块提供检查点机制。它会保存一个数据文件到磁盘，如果应用程序发生崩溃，可以从这个检查点恢复，然后继续运行。

indexer_trigrams.erl

这个模块于之前的lib_trigrams.erl类似，它负责对单词进行三元分析（参考15.5小节的例子程序）。

indexer_misc.erl

常用的各种杂项操作，包含一个“映射-归并”算法。

indexer_words.erl

从文件中抽取单词，然后调用其他函数完成三元分析，并对单词进行词干分析操作。

indexer_dir_crawler.erl

indexer_server会调用这个模块来获取目录列表。

你一定已经留意到了这些文件名。有一个文件名为indexer.erl，其他的一些文件则都取了indexer_XXX.erl这样的名字。这实际上遵循的是发布复杂Erlang应用程序常用的一种命名方式。在用Erlang构建一个应用程序的时候，通常会选择一个应用程序名（这里就是indexer），然后会构建一个“主模块”（也就是indexer.erl）和一些“子模块”（indexer_XXXX）。

通过这种方式，可以从其他的模块自由的复制、重命名乃至更改一些代码，而不用担心会发生命名冲突。这个方式既有优点也有缺点，主要的优点是在这种命名空间习惯下我们可以独立的开发代码，无须担心彼此之间的共享问题。而其缺点主要在于通用库的代码可能最终会被大量的名字和版本号弄得乱糟糟的，当需要进行合并的时候，就会比较困难了。

Erlang自己的代码发布也遵循这一约定。比如，/usr/local/lib/erlang/lib/mnesia-4.3.4/src这个目录就是根据这一约定用于存放Mnesia代码的目录。这一规则的主要例外是kernel和stdlib当中的模块，它们不遵循命名约定，而是尽量使用更为直观的短名字。 382

20.5 面向未来的成长

计算机领域的版图已经改变。庞大的单核处理器其增长已经日趋乏力，就像恐龙一样，它也会渐渐变成化石进入博物馆。取而代之的多核时代，在分布式的处理器上，我可以像使用电能一样，按需获取我们想要的处理能力。

与之相应，我们的编程技术也正朝着新的时代奋力迈进，Erlang正是一个这样的技术方案。这本书中，已经向你展示了一些基本的Erlang概念，循着这个路径，你会得到更可靠、更好维护而且在现有（和未来）的架构上更具扩展性的代码。 383

祝你在新的风格里编程愉快。

附录A 给我们的程序写文档

Erlang在设计时严格地考虑过类型问题，但Erlang的类型大都是隐含的。在我们进行交流的时候，很有必要清晰化地标示这些类型，为此Erlang社区开发出了一种标注方式来表达类型。这些标注并不是程序代码本身，而是作为注释的一部分，这是描述代码的一种很好的方式。通过这种方式，我们可以明确描述一个函数接受和返回的都是什么类型的数据，消除不必要的混淆。

最近，这一标注方式已经被一些处理Erlang程序源码的工具所支持（我们后面会谈到这些工具）。

需要再强调一次，类型标注仅仅用于文档，它们并不是Erlang代码，在shell中也不能使用这些标注。Erlang模块中的类型声明也只是注释的一部分，编译器会完全地忽略它。比如，我们可以这样写：

```
-module(math).
-export([fac/1]).

%% @spec fac(int()) -> int().

fac(0) -> 1;
fac(N) -> N * fac(N-1).
```

在Erlang的手册和API描述中，就使用了这样的类型标注系统。在本书中，你能随处见到这样的用法。这些标注只是一些信息，而非函数中类型的完整表述，但这已经足够，通过这些标注，我们就已经能够捕捉到函数的本质。

A.1 Erlang 类型标注

在标注中，我们需要明确的是两种东西：类型和函数规范。

A.1.1 定义一个类型

类型名称“typeName”以typeName()的方式表示。类型可以是预定义的，也可以是用户定义的。预定义的类型有any()、atom()、binary()、bool()、char()、cons()、deep_string()、float()、function()、integer()、iolist()、list()、nil()、none()、number()、pid()、port()、reference()、string()、term()以及tuple()。

它们的含义如下。

- any()意味着“任意的Erlang数据类型”，term()也表示相同的含义。
- atom()、binary()、float()、function()、integer()、pid()、port()以及reference()都是Erlang语言的基本数据类型。
- bool()是原子，其取值要么是true要么是false。

- char()是integer()的一个子集，用来表示一个字符。
- iolist()可以递归定义为[char() | binary() | iolist()]。一块二进制数据可以包含在这个列表的尾部，这通常是一种高效产生字符输出的做法。可以参看13.3节，那里就有一个产生IO列表的样例函数。
- tuple()是一个元组。
- list(L)是[L]的另外一种表示方式。
- nil()表示一个空的列表[]。
- string()与list(char())表示相同的意思。
- deep_string()等同于[char()|deep_string()]这样的递归定义方式。
- none()意味着“无数据类型”，这种类型用于表达一个函数永远不会返回（比如，一个无限等待的接收循环）。严格意义上说，这不是一种类型，但作为文档中的一种表达方式，它能很好地表示“函数永不返回”这个意思。

我们可以用下面的形式来构造新的类型（用户定义类型）。

```
@type newType() = TypeExpression
```

386

先来看看这个用法的一些例子，然后再来讨论定义一个类型表达式的正式规则。

```
@type  onOff() = on | off.
@type  person() = {person, name(), age()}.
@type  people() = [person()].
@type  name() = {firstname, string()}.
@type  age() = integer().
...
```

根据上述的定义规则，我们可以有一些例子，比如：{firstname, "dave"}就是name()类型的，[{person, {firstname,"john"}, 35}, {person, {firstname,"mary"}, 26}]就符合people()类型，比较简单，我们可以依次类推。

类型表达式的定义归纳如下。

- {T1, T2, ..., Tn}是一个元组类型表达式，其中的T1、T2、...、Tn也都是类型表达式。如果X1是T1类型，X2是T2类型，……，Xn是Tn类型，我们就可以说{X1, X2, ..., Xn}是{T1, T2, ..., Tn}类型的。
- [T]是一个列表类型表达式，其中的T也是一个类型表达式。对于一个[X1, X2, ..., Xn]，如果所有的X都是T类型的，那么我们也可以说这个列表是[T]类型的。
- T1|T2是一个选择类型的类型表达式，这里的T1和T2都是某种类型。如果X的类型是T1或者T2，那么我们可以说X是 T1|T2 类型的。
- fun(T1, T2, ..., Tn) -> T是一个函数类型表达式，其中所有的T都是类型表达式。如果X1是T1类型的，X2是T2类型，……，那么我们可以说fun(X1, X2, ..., Xn) -> X是fun(T1, T2, ..., Tn) -> T类型的。
- 预定义类型、用户定义类型或者预定义类型的一个实例，也都是类型表达式。

现在我们了解如何定义类型了，下面我们来看看函数规范。

A.1.2 规范化函数的输入和输出类型

函数规范描述一个函数的参数以及其返回值的类型。函数规范看上去是下面这样的：

```
@spec functionName(T1, T2, ..., Tn) -> Tret
```

387 这里的T1，T2，…，Tn描述函数的参数，而Tret则描述返回值。

每一个Ti都可以是下面3种形式中的一种。

- TypeVar：一个类型变量。类型变量是用一个类型声明的变量，它用来描述一个未知的类型（顺便说一句，这和Erlang的变量毫无关系）。如果我们在一个类型规范中的几个不同地方都使用相同的类型变量，毫无疑问，这个类型变量的多个实例都是相同类型的。
- TypeVar::Type：一个类型变量后面跟着一个类型。这意味着TypeVar是Type类型的（Type是一个类型表达式）。
- Type：一个类型表达式。

先来看例子，然后我再做详细解释：

```
@spec file:open(FileName, Mode) -> {ok, Handle} | {error, Why}.
@spec file:read_line(Handle) -> {ok, Line} | eof.
```

这里file:open/2的规范表明，如果我们打开FileName文件，则获得的返回值会是{ok, Handle}或者{error, Why}。这里的竖线“|”表示“或”。

说明　在上下文中，我们通常都省略函数的模块前缀。比如，当我们只是在谈论file模块的时候，会写@spec open(...)而不是@spec file:open()。

这里的FileName和Mode都是类型变量，但它们是什么类型的呢？FileName到底是原子、字符串或者什么别的类型？只通过这个定义，我们无法确定——这也是为什么我们会说这个定义是“非正式”的原因。有些时候我们并不需要知道一个参数的数据类型，比如说，我们并不需要知道Handle到底是什么类型的（它是file:open的其中一个返回值），因为只需要将它原样传递给file:read_line/1就行了。

我们也可以像下面的例子一样使用函数类型：

```
@spec lists:map(fun(A) -> B, [A]) -> [B].
@spec lists:filter(fun(X) -> bool(), [X]) -> [X].
```

通常我们并不需要涉及函数类型的过多细节，它们都是用描述性的名字来表示的，通过上下文我们能很容易地猜出含义。如果想要避免猜测，我们可以用几种不同的方式来优化它，它们的含义相同。

这里是一个例子：

```
@spec file:open(FileName::string(), [mode()]) ->
    {ok, Handle::file_handle()} | {error, Why::string()}
@type mode() = read  |  write  |  compressed  |  raw  |  binary | ...
```

388

另一个例子是这样的：

```
@spec file:open(string(), Modes) -> {ok, Handle} | {error, string()}
    Handle = file_handle(),
    Modes = [Mode],
    Mode = read  |  write  |  compressed  |  raw  |  binary | ...
```

还可以用下面这种形式：

```
@spec file:open(string(), [mode()]) -> {ok, file_handle()} | error().
@type error() = {error, string()}.
@type mode() = read  |  write  |  compressed  |  raw  |  binary | ...
```

A.1.3 API 中的类型定义

在Erlang手册以及本书当中，定义API时我们常用描述性的列表来对函数定义进行说明。这种情况下，我们一般都会去掉@spec关键字，直接进行定义。通常我们只会用到类型变量，然后跟一段文字来描述它们。

这里有一个例子，是从手册中file部分的页面中摘录下来的。

```
file:open(File, [Mode]) -> {ok, Handle} | {error, Why}
```

用Mode模式打开File文件（字符串）。Mode的取值为read、write等。如果打开成功，则返回{ok, Handle}，否则，返回{error, Why}，这里的Why是错误信息。打开成功之后，可以用Handle来访问这个文件。

```
file:read_line(Handle) -> {ok, Line} | eof
```

从已经打开的Handle文件中读取一行。返回值Line是一个字符串，或者当到达文件尾部时，返回eof。等等。

A.2 使用类型的工具

下面这些工具会使用类型。

- Edoc。EDoc是Erlang程序文档生成器。它的功能基本与Java编程语言中的Javadoc工具相当。
  EDoc允许你在程序代码中用在注释中插入标注的方式来写文档。在EDoc中可以使用的标注标签包括：@name、@doc、@type和@author等。
  edoc模块提供了一大堆函数，通过这些函数，可以操作那些包含了EDoc文档标注的Erlang源程序。
- Dialyzer。Dialyzer是一个静态分析工具，它可以识别程序的细微差异，比如，类型错误、不可达的代码、无必要的测试等。它既可以分析一个单独的Erlang模块，也可以分析整个应用程序（一组Erlang模块）。

这两个工具都已经打包在Erlang的标准发行包中了，随时可用。

389

390

附录B

Microsoft Windows环境下的Erlang环境

我在很多不同的平台下运行Erlang，我个人喜欢在各种不同的平台下都保持开发环境的一致。在Windows环境下，我发现下面的配置非常有用（在你的机器上，可能需要改变一些目录名称，以符合你自己的系统）。B.1节中的步骤对于Windows用户来说是最关键的，如果你喜欢类Unix的开发环境，可以执行从B.2节到B.4节中的步骤。如果你喜欢Emacs，那么可以执行B.5节中的步骤（这很有用，因为在Emacs下有一个单独的Erlang高级模式，能带来不少方便）。

B.1 Erlang

到http://www.erlang.org/download.html下载最新的Windows安装文件（37M），文件名为otp_win32-R11B-3.exe[①]。点击安装，Erlang就被安装在默认目录。在我的机器上，Erlang的默认安装目录是C:\Prgram Files\erl.5.4.12\。

B.2 下载安装 MinGW

MinGW是Minimalist GNU for Windows 的缩写。

(1) 下载MinGW安装程序，这是一个小程序，大约只有130KB，安装过程中，它会询问你具体想要安装的部分，并负责下载安装。我用的是从 http://www.mingw.org/download.shtml下载的MinGW-5.0.2.exe。

(2) 运行安装程序，确保选中了“MinGW base tools”和“MinGW make”项目。

(3) 随后就是一路回答Yes，它会安装大约44 MB的东西到C:\MinGW目录中。

B.3 下载安装 MSYS

(1) 到http://www.mingw.org/download.shtml 下载最新的 MSYS。在标有MSYS的部分找到符合你的系统的版本，我下载的是MSYS-1.0.10.exe（2742 KB）。

(2) 到http://sourceforge.net/projects/mingw下载。

(3) 点击安装这个文件。

(4) 回答问题，你只需要回答一个问题——MinGW安装在哪里，回答C:\MinGW。之后，你就有了一个可用的shell。选择Start→Programs→MinGW→MSYS→msys启动shell。你也可以通过点击桌面上的蓝色图标，

① 这个文件的名字随着版本升级而变化，选最新的文件就对了。——译者注

进入shell。

如果你点击有一个大M的图标，你会得到一个shell的窗口，在这个shell中，可以使用很多你喜欢的Unix命令。

B.4 安装 MSYS 开发工具（可选）

装了这个，就可以有SSH之类的东西，这不是必须的，但有了当然更好。

(1) 找到MsysDTK-1.0.1.exe。

(2) 点击安装。

B.5 emacs

(1) 到 ftp://ftp.gnu.org/gnu/emacs/windows/下载emacs-21.3-fullbin-i386.tar.gz。

(2) 用 Winzip之类的压缩工具将其解压到一个目录中去。

(3) 去到emacs的bin目录，点击addpm完成安装。（它会在开始菜单的“Start→Programs”下安装emacs的快捷方式），我自己也在桌面上放了一个runemacs的快捷方式。 392

定制emacs

把下面emacs.setup中的内容复制home目录下的.emacs文件中。在我的机器上，这个文件的路径是C:/.emacs。[①]

emacs.setup

```
(setq default-frame-alist
      '((top . 10) (left . 10)
        (width . 80) (height . 43)
        (cursor-color . "blue")
        (cursor-type . box)
        (foreground-color . "black")
        (background-color . "white")
        (font . "-*-Courier New-bold-r-*-*-18-108-120-120-c-*-iso8859-8")))

(show-paren-mode)

(global-font-lock-mode t)
(setq font-lock-maximum-decoration t)


;; Erlang stuff this is the path to erlang
;; windows path below -- change to match your environment
(setq load-path (cons  "c:/Program Files/erl5.5.3/lib/tools-2.5.3/emacs"
                       load-path))

(require 'erlang-start)

;; (if window-system
;;    (add-hook 'erlang-mode-hook 'erlang-font-lock-level-3))

 (add-hook 'erlang-mode-hook 'erlang-font-lock-level-3)
```

393

① 对于中文用户，这个配置文件中的字体可能需要略加改进。——译者注

附录C

资　源

C.1　在线文档

Erlang官方文档

http://www.erlang.org/doc/

Erlang官方站点完整的Erlang文档。

Erlang手册

http://www.erlang.org/doc/man/erlang.html

所有本书未曾涉及的细节都可以在这份文档之中找到。

Erlang编程风格指南

http://www.erlang.se/doc/programming_rules.shtml

怎样才是好的Erlang编程风格？这里定义了一些原则。在数个商业项目的编程之中实际应用了这些风格。[①]

Erlang常见问题解答

http://www.erlang.org/faq/t1.html

每一个好项目都得有一个常见问题解答。

手册

http://www.erlang.org/doc/man/index.html

所有Erlang模块的最新文档（如果它有文档的话）以及操作命令(如`erl`、`erlc`、`escript`等)。每个模块都有自己独立的文档页面，如`erlang.org/doc/man/lists.html`。

PDF版手册

http://www.erlang.org/doc/pdf/index.html

Erlang系统所有模块最新PDF版手册的顶层目录。每个模块有自己单独的手册文件，如`erlang.org/doc/pdf/mnesia.pdf`。

应用程序文档

http://www.erlang.org/doc/apps/Name

① 还可以从这里找到PDF版：http://www.erlang.se/doc/programming_rules.pdf。

每个应用程序都有自己的PDF文档，如Mnesia的PDF文档在这里可以找到：http://www.erlang.org/doc/apps/mnesia。 395

C.2 书籍和论文

Concurrent Programming in Erlang[VWWA96]

第一本Erlang书籍，包含两个部分，第一部分讲述Erlang语言本身，第二部分讲述应用程序。这本书的第一部分有免费PDF版可以下载。①

Erlang Programmation[Rém03] （法语）

Mickaël Rémond 的这本书是为法语读者准备的。

“Making Reliable Systems in the Presence of Software Errors”

这是我的博士论文②，包含了Erlang的部分历史，以及其理论体系。相比这本书，它为更严肃的目标而准备。你会找到OTP行为的更多描述，以及一些Erlang的实践练习。③

Erlang 4.7规范

新手慎入，这篇文档试图精确描述Erlang 4.7④版规范的细节。虽然有点过时，但从语言规范的角度来说，仍是这一领域最好的文档。而从另一个角度来说，Erlang从4.7到5.5在这些方面的变化也不大，对于目前的5.5版Erlang仍然适用。

C.3 相关链接

http://www.it.uu.se/research/group/hipe/publications.shtml

高性能Erlang（HIPE）是瑞典Uppsala大学的一个研究小组，他们在多年以前就已经开始参与Erlang的开发工作。这页包含了他们大量Erlang论文和研究报告的链接。

http://dmoz.org/Computers/Programming/Languages/Erlang/

Erlang的开放目录。

C.4 博客

http://armstrongonsoftware.blogspot.com/ 396

我对软件的遐想。

http://yarivsblog.com/

Yariv Sadan 关于Erlang和ErlyWeb的博客，ErlyWeb是一个Erlang写的Web开发框架。

http://www.process-one.net/en/blogs/

Mickaël Rémond和他的朋友们的博客。

http://erlang-china.org/

Erlang中文社区，是一个面向中文Erlang用户的开放式博客，报道Erlang的最新资讯，发布Erlang的中文版文档，以及一个小的开放式Wiki（Erlang文档计划）。

---

① http://www.erlang.org/download/erlang-book-part1.pdf。

② http://www.erlang.org/download/armstrong_thesis_2003.pdf。

③ 这篇论文已由Erlang中文社区的网友段先德译为中文，并在Erlang中文社区erlang-china.org发表。——译者注

④ http://www.erlang.org/download/erl_spec47.ps.gz。

C.5 论坛、社区以及交流网站

http://www.erlang.org/mailman/listinfo

Erlang的邮件列表，在很多网站都有镜像。

http://www.trapexit.org/

Erlang的大社区站点，包含论坛、Wiki以及How-to文档。其中的how-to[①]和cookbook[②]部分非常有用。

Erlounges

Erlounges是由Erlang用户自发组织的聚会，它们一般都在酒吧、餐馆等地点开展，以非正式的方式举行。其主要的话题也围绕Erlang编程展开，通常事先会在Erlang的邮件列表中广而告之。

IRC频道

`irc.freenode.net`上有一个`#erlang`频道。

C.6 会议

ACM SIGPLAN Workshop

每年一次，为期一天。

Erlang User Conference

每年一次[③]，为期两天。

C.7 项目

http://jungerl.sourceforge.net/

397 一个SourceForge上的Erlang大代码仓库。

http://cean.process-one.net/

cean是Comprehensive Erlang Archive Network的缩写，一个试图合并所有Erlang正在进行的项目的项目。

http://yaws.hyber.org/

yaws是Yet Another Web Server的缩写。这是一个为多个商业产品所采用的Web服务器，它是用Erlang写成的。

http://ejabberd.jabber.ru/

基于Jabber协议的即时消息服务器，也是用Erlang写成的。

C.8 参考文献

[Rém03]Mickaël Rémond. Erlang Programmation. Eyrolles, Paris,2003.

[VWWA96]Robert Virding, Claes Wikstrom, Mike Williams, and Joe Armstrong.Concurrent Programming
398 in Erlang.Prentice Hall, Englewood Cliffs, NJ, second edition, 1996.

① http://wiki.trapexit.org/index.php/Category:HowTo。

② http://wiki.trapexit.org/index.php/Category:CookBook。

③ http://www.erlang.se/euc。

附录 D

套接字应用程序

在本附录中，我们将会关注于lib_chan库的实现细节。在10.5节，我们已经介绍过这个库。在紧随其后的第11章中，我们就用到了这个库。

lib_chan的代码在TCP/IP之上实现了一个完整的网络层，提供了包括认证以及Erlang数据流传输在内的完整功能。一旦理解了lib_chan的内在原则，我们就可以依葫芦画瓢，加以裁剪，做出符合我们自己需要的基础通信架构。

另一方面，就其本身而言，在构造分布式的系统时，lib_chan也会是一个相当有用的组件。

为保持附录本身的内容完整，部分内容与第10.5节有重复。

本附录中的代码是迄今为止我们介绍过的所有代码中最为复杂的。如果第一遍看不懂，是再正常没有的事了。而如果你只是想用lib_chan而不关心它是怎么实现的，那么，只需要读第一小节，跳过其他的部分就行了。

D.1 例子

作为开始，我们要用一个简单的例子来向你展现如何使用lib_chan。我们将会创建一个简单的服务器进程，它的功能是要计算阶乘和斐波那契数，我们还要用一个密码来保护它。

这个服务器会在2233端口工作。

我们通过下面4个简单的步骤来创建这个服务器。

(1) 写一个配置文件。

(2) 编写服务代码。

(3) 启动服务器进程。

(4) 通过网络访问这个服务器。

D.1.1 第一步：写一个配置文件

下面就是我们这个例子的配置文件：

socket_dist/config1

```
{port, 2233}.
{service, math, password, "qwerty", mfa, mod_math, run, []}.
```

这个配置文件包含了几个service元组，形式如下：

```
{service, <Name>, password, <P>, mfa, <Mod>, <Func>, <ArgList>}
```

这些参数以service、password和mfa这些原子来分隔。mfa是module、function、args的缩写，这表示后面跟着的3个参数意为模块名、函数名以及调用方法所用的参数列表。

在我们的例子中，这个配置文件说明了一个名为math的服务器，会在2233端口提供服务。这个服务器由密码qwerty保护。它在mod_math中定义，并通过调用mod_math:run/3来启动，而run/3的第3个参数是[]。

D.1.2　第二步：编写服务器代码

math服务器的代码是这样的：

socket_dist/mod_math.erl

```
-module(mod_math).
-export([run/3]).

run(MM, ArgC, ArgS) ->
    io:format("mod_math:run starting~n"
              "ArgC = ~p ArgS=~p~n",[ArgC, ArgS]),
    loop(MM).

loop(MM) ->
    receive
        {chan, MM, {factorial, N}} ->
            MM ! {send, fac(N)},
            loop(MM);
        {chan, MM, {fibonacci, N}} ->
            MM ! {send, fib(N)},
            loop(MM);
        {chan_closed, MM} ->
            io:format("mod_math stopping~n"),
            exit(normal)
    end.

fac(0) -> 1;
fac(N) -> N*fac(N-1).

fib(1) -> 1;
fib(2) -> 1;
fib(N) -> fib(N-1) + fib(N-2).
```


当一个客户端连接到2233端口并请求名为math的服务时，lib_auth会对客户端进行认证。若密码正确无误，会通过调用mod_math:run(MM, ArgC,ArgS)来新开一个处理进程。MM是中间人进程[①]的PID，ArgC来自客户端，ArgS则来自于配置文件。

当客户端发起一个X消息给服务器进程时，该进程会收到{chan,MM,X}形式的消息。如果客户端死掉或者网络连接出现问题，它也会收到一个{chan_closed,MM}的消息。需要给客户端发送Y消息时，服务器进程只需简单的使用MM ! {send, Y}，而要关闭一个通信频道，只需MM ! close。

math服务器非常简单，它只需等待一个{chan, MM, {factorial, N}}消息，然后用MM ! {send, fac(N)}将结果发给客户端。

D.1.3　第三步：启动服务器

启动服务器的步骤如下：

① middle man process，即中间人进程。在Erlang代码中常常都能见到它的影子，一般而言，这种进程负责处理一些边边角角的外围事务，如等待后续数据、编码解码、数据格式化等，并将规整好的格式良好的数据交给核心业务进程。——译者注

```
1> lib_chan:start_server("./config1").
lib_chan starting:"./config1"
Terms=[{port,2233},
       {service,math,password,"qwerty",
        mfa,mod_math,run,[]}]
true
```

D.1.4 第四步：通过网络访问服务器

我们可以在同一台机器上通过下面的代码来进行测试：

```
2> {ok, S} = lib_chan:connect("localhost",2233,math,
                              "qwerty",{yes,go}).
```

401

```
{ok,<0.47.0>}
3> lib_chan:rpc(S, {factorial,20}).
2432902008176640000
4> lib_chan:rpc(S, {fibonacci,15}).
610
4> lib_chan:disconnect(S).
close
mod_math stopping
```

D.2 lib_chan 如何工作

lib_chan使用了如下4个模块的代码。

- lib_chan扮演“主模块”的角色。对外调用的程序员而言，它是唯一需要了解的模块。掌握它提供的函数，就能完成相关任务。另外3个模块（下面将会讲到）用于在内部实现lib_chan的细节。
- lib_chan_mm负责编码解码Erlang的消息，并管理一个套接字上的通信。
- lib_chan_cs设置服务器并管理客户端连接，限制最大的并发连接数也是它的主要工作之一。
- lib_chan_auth的代码用于实现一个简单的“暗号/回应”认证。

D.2.1 lib_chan

lib_chan的结构如下：

```
-module(lib_chan).

start_server(ConfigFile) ->
    %% read configuration file - check syntax
    %% call start_port_server(Port, ConfigData)
    %% where Port is the required Port and ConfigData
    %% contines the configuration data

start_port_server(Port, ConfigData) ->
   lib_chan_cs:start_raw_server( ..
          fun(Socket) ->
              start_port_instance(Socket, ConfigData),
          end, ... )
   %% lib_chan_cs manages the connection
   %% when a new connection comes the fun which is an
   %% argument to start_raw_server will be called
```

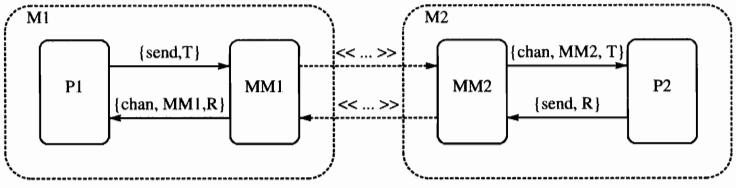
402

```
start_port_instance(Socket, ConfigData) ->
   %% this is spawned when the client connects
   %% to the server. Here we setup a middle man,
   %% then perform authentication. If everything works call
   %% really_start(MM, ArgC, {Mod, Func, ArgS})
   %% (the last three arguments come from the configuration file

really_start(MM, ArgC, {Mod, Func, ArgS}) ->
   apply(Mod, Func, [MM, ArgC, ArgS]).

connect(Host, Port, Service, Password, ArgC) ->
   %% client side code
```

D.2.2 lib_chan_mm：中间人

lib_chan_mm实现了一个中间人模式。对于应用程序而言，只需要理解消息。而对于网络而言，被传输的只是一些字节流。这两者之间的中间人则隐藏了通过套接字进行通信的实现细节，它的职责是组装消息（在网络上进行传输时这些数据有时需要先被分片、传输然后再进行组装）和对Erlang数据进行编码解码，与字节流进行转换。

这个时候请仔细看看图D-1，它展现了中间人的架构模式。当M1机器上的P1进程想要给M2机器上的P2进程发送一个T消息时，它执行MM1!{sent,T}语句。在这里，MM1可以看作是P2的代理。所有发到MM1的消息最终都经过编码，然后写到一个套接字当中，发给MM2。MM2对从套接字中收到的数据进行解码，然后向P2发送一条内容为{chan, MM2, T}的消息。



图D-1　通过中间人进行网络通信

403

在M1这台机器上，MM1进程扮演了P2代理的角色，而在M2机器上，MM2进程则扮演了P1代理的角色。MM1和MM2都是中间人进程的PID。中间人进程的代码看起来是这样的：

```
loop(Socket, Pid) ->
    receive
        {tcp, Socket, Bin} ->
            Pid ! {chan, self(), binary_to_term(Bin)},
            loop(Socket, Pid);
        {tcp_closed, Socket} ->
            Pid ! {chan_closed, self()};
        close ->
            gen_tcp:close(Socket);
        {send, T} ->
            gen_tcp:send(Socket, [term_to_binary(T)]),
            loop(Socket, Pid)
    end.
```

这个循环可以看作是套接字的数据世界与Erlang的消息世界之间的通道。你在D.3节能找到lib_chan_mm的完整代码。完整的代码比我们这里展示的简单版本要稍微复杂一点，但究其本质仍是一样的。主要的区别在于，那个版本中，我们增加了一些代码来跟踪消息，并加入了一些接口函数。

D.2.3 lib_chan_cs

lib_chan_cs负责建立客户端与服务器之间的通信。它最重要的两个接口函数（也就是它导出的函数）是这样的：

```
start_raw_server(Port, Max, Fun, PacketLength)
```

这个函数会在某个端口上启动一个连接监听器,它最多会允许Max个并发的会话，Fun是一个接受单个参数的函数。当一个连接建立时，就会运行Fun(Socket)函数，套接字通信的数据包的长度是PacketLength。

```
start:raw_client(Host, Port, PacketLength) => {ok, Socket} | {error, Why}
```

这个函数试图去连接一个由start_raw_server打开的端口。

lib_chan_cs的代码遵循14.1节中描述过的模式。所不同的是，它还会跟踪并控制最大并发连接数，以避免服务器过载。这个概念相当简单，但实现起来却比较琐碎，源代码中，有二十多行看起来有点奇怪的代码（捕获退出消息等），就是实现这个功能的。这样的方式来编写代码似乎看起来有点纠缠不清，但不用担心，它能很好的完成自己的工作，而且，对于这个模块的使用者来说，也无须关心这些烦琐的细节。 404

D.2.4 lib_chan_auth

这个模块实现了一个简单的“暗号 / 应答”验证机制。“暗号 / 应答”验证机制相当简单，即对每一个服务名称来说，都有一个相应的“口令”，客户端和服务器端围绕这个“口令”来进行验证。用一个例子来展示，你就知道这比它听起来的要简单多了。假设我们的服务名为math，它的“口令”是qwerty。

如果一个客户端想使用math服务，那么就需要向服务器证明自己也知道这个口令。这个过程是下面这样的。

(1) 客户端向服务器发送请求，表明自己想使用math服务。

(2) 服务器产生一个随即字符串C，将其发送给客户端，这就是一个“暗号”。这个字符串由函数lib_chan_auth:make_challenge()产生的。我们可以在shell中交互式的看看它怎么运作：

```
1> C = lib_chan_auth:make_challenge().
"qnyrgzqefvnjdombanrsmxikc"
```

(3) 客户端收到C这个字符串，并计算一个“反馈”R，这里的R=MD5(C++Secret)。R用lib_chan_auth:make_response函数产生。如：

```
2> R = lib_chan_auth:make_response(Challenge, "qwerty").
"e759ef3778228beae988d91a67253873"
```

(4) 反馈返回到服务器。服务器接收到这个反馈，然后检查它是否正确。检查通过lib_chan_auth:is_response_correct这个函数来完成。

```
3> lib_chan_auth:is_response_correct(C, R, "qwerty").
true
```

D.3 lib_chan 的代码

下面列出代码。

D.3.1 lib_chan

socket_dist/lib_chan.erl

```
-module(lib_chan).
```

```
-export([cast/2, start_server/0, start_server/1,
         connect/5, disconnect/1, rpc/2]).
-import(lists, [map/2, member/2, foreach/2]).
-import(lib_chan_mm, [send/2, close/1]).
%%----------------------------------------------------------------------
%% Server code

start_server() ->
    case os:getenv("HOME") of
        false ->
            exit({ebadEnv, "HOME"});
        Home ->
            start_server(Home ++ "/.erlang_config/lib_chan.conf")
    end.

start_server(ConfigFile) ->
    io:format("lib_chan starting:~p~n",[ConfigFile]),
    case file:consult(ConfigFile) of
        {ok, ConfigData} ->
            io:format("ConfigData=~p~n",[ConfigData]),
            case check_terms(ConfigData) of
                [] ->
                    start_server1(ConfigData);
                Errors ->
                    exit({eDeaemonConfig, Errors})
            end;
        {error, Why} ->
            exit({eDaemonConfig, Why})
    end.

%% check_terms() -> [Error]

check_terms(ConfigData) ->
    L = map(fun check_term/1, ConfigData),
    [X || {error, X} <- L].

check_term({port, P}) when is_integer(P)      -> ok;
check_term({service,_,password,_,mfa,_,_,_}) -> ok;
check_term(X) -> {error, {badTerm, X}}.

start_server1(ConfigData) ->
    register(lib_chan, spawn(fun() -> start_server2(ConfigData) end)).

start_server2(ConfigData) ->
    [Port] = [ P || {port,P} <- ConfigData],
    start_port_server(Port, ConfigData).

start_port_server(Port, ConfigData) ->
    lib_chan_cs:start_raw_server(Port,
                                 fun(Socket) ->
                                         start_port_instance(Socket,
                                                             ConfigData) end,
                                 100,
                                 4).
```

```
start_port_instance(Socket, ConfigData) ->
    %% This is where the low-level connection is handled
    %% We must become a middle man
    %% But first we spawn a connection handler
    S = self(),
    Controller = spawn_link(fun() -> start_erl_port_server(S, ConfigData) end),
    lib_chan_mm:loop(Socket, Controller).

start_erl_port_server(MM, ConfigData) ->
    receive
        {chan, MM, {startService, Mod, ArgC}} ->
            case get_service_definition(Mod, ConfigData) of
                {yes, Pwd, MFA} ->
                    case Pwd of
                        none ->
                            send(MM, ack),
                            really_start(MM, ArgC, MFA);
                        _ ->
                            do_authentication(Pwd, MM, ArgC, MFA)
                    end;
                no ->
                    io:format("sending bad service~n"),
                    send(MM, badService),
                    close(MM)
            end;
        Any ->
            io:format("*** ErL port server got:~p ~p~n",[MM, Any]),
            exit({protocolViolation, Any})
    end.

do_authentication(Pwd, MM, ArgC, MFA) ->
    C = lib_chan_auth:make_challenge(),
    send(MM, {challenge, C}),
    receive
        {chan, MM, {response, R}} ->
            case lib_chan_auth:is_response_correct(C, R, Pwd) of
                true ->
                    send(MM, ack),
                    really_start(MM, ArgC, MFA);
                false ->
                    send(MM, authFail),
                    close(MM)
            end
    end.


%% MM is the middle man
%% Mod is the Module we want to execute ArgC and ArgS come from the client and
%% server respectively
really_start(MM, ArgC, {Mod, Func, ArgS}) ->
    %% authentication worked so now we're off
    case (catch apply(Mod,Func,[MM,ArgC,ArgS])) of
        {'EXIT', normal} ->
            true;
        {'EXIT', Why} ->
```

```
            io:format("server error:~p~n",[Why]);
        Why ->
            io:format("server error should die with exit(normal) was:~p~n",
                      [Why])
    end.

%% get_service_definition(Name, ConfigData)

get_service_definition(Mod, [{service, Mod, password, Pwd, mfa, M, F, A}|_]) ->
    {yes, Pwd, {M, F, A}};
get_service_definition(Name, [_|T]) ->
    get_service_definition(Name, T);
get_service_definition(_, []) ->
    no.

%%----------------------------------------------------------------------
%% Client connection code
%% connect(...) -> {ok, MM} | Error

connect(Host, Port, Service, Secret, ArgC) ->
    S = self(),
    MM = spawn(fun() -> connect(S, Host, Port) end),
    receive
        {MM, ok} ->
            case authenticate(MM, Service, Secret, ArgC) of
                ok    -> {ok, MM};
                Error -> Error
            end;
        {MM, Error} ->
            Error
    end.

connect(Parent, Host, Port) ->
    case lib_chan_cs:start_raw_client(Host, Port, 4) of
        {ok, Socket} ->
            Parent ! {self(), ok},
            lib_chan_mm:loop(Socket, Parent);
        Error ->
            Parent ! {self(),  Error}
    end.

authenticate(MM, Service, Secret, ArgC) ->
    send(MM, {startService, Service, ArgC}),
    %% we should get back a challenge or a ack or closed socket
    receive
        {chan, MM, ack} ->
            ok;
        {chan, MM, {challenge, C}} ->
            R = lib_chan_auth:make_response(C, Secret),
            send(MM, {response, R}),
            receive
                {chan, MM, ack} ->
                    ok;
                {chan, MM, authFail} ->
                    wait_close(MM),
```

```
                    {error, authFail};
                Other ->
                    {error, Other}
            end;
        {chan, MM, badService} ->
            wait_close(MM),
            {error, badService};
        Other ->
            {error, Other}
    end.

wait_close(MM) ->
    receive
        {chan_closed, MM} ->
            true
    after 5000 ->
            io:format("**errror lib_chan~n"),
            true
    end.

disconnect(MM) -> close(MM).

rpc(MM, Q) ->
    send(MM, Q),
    receive
        {chan, MM, Reply} ->
            Reply
    end.

cast(MM, Q) ->
    send(MM, Q).
```

D.3.2 lib_chan_cs

socket_dist/lib_chan_cs.erl

```
-module(lib_chan_cs).
%% cs stands for client_server

-export([start_raw_server/4, start_raw_client/3]).
-export([stop/1]).
-export([children/1]).
%% start_raw_server(Port, Fun, Max)
%%   This server accepts up to Max connections on Port
%%   The *first* time a connection is made to Port
%%   Then Fun(Socket) is called.
%%   Thereafter messages to the socket result in messages to the handler.

%% tcp_is typically used as follows:
%% To setup a listener
%%   start_agent(Port) ->
%%     process_flag(trap_exit, true),
%%     lib_chan_server:start_raw_server(Port,
%%                                      fun(Socket) -> input_handler(Socket) end,
%%                                      15,
%%                                      0).
```

409

```
start_raw_client(Host, Port, PacketLength) ->
    gen_tcp:connect(Host, Port,
                    [binary, {active, true}, {packet, PacketLength}]).

%% Note when start_raw_server returns it should be ready to
%% Immediately accept connections

start_raw_server(Port, Fun, Max, PacketLength) ->
    Name = port_name(Port),
    case whereis(Name) of
        undefined ->
            Self = self(),
            Pid = spawn_link(fun() ->
                                 cold_start(Self,Port,Fun,Max,PacketLength)
                             end),
            receive
                {Pid, ok} ->
                    register(Name, Pid),
                    {ok, self()};
                {Pid, Error} ->
                    Error
            end;
        _Pid ->
            {error, already_started}
    end.

stop(Port) when integer(Port) ->
    Name = port_name(Port),
    case whereis(Name) of
        undefined ->
            not_started;
        Pid ->
            exit(Pid, kill),
            (catch unregister(Name)),
            stopped
    end.
children(Port) when integer(Port) ->
    port_name(Port) ! {children, self()},
    receive
        {session_server, Reply} -> Reply
    end.


port_name(Port) when integer(Port) ->
    list_to_atom("portServer" ++ integer_to_list(Port)).


cold_start(Master, Port, Fun, Max, PacketLength) ->
    process_flag(trap_exit, true),
    %% io:format("Starting a port server on ~p...~n",[Port]),
    case gen_tcp:listen(Port, [binary,
                               %% {dontroute, true},
                               {nodelay,true},
                               {packet, PacketLength},
                               {reuseaddr, true},
```

410

```
                                {active, true}]) of
        {ok, Listen} ->
            %% io:format("Listening to:~p~n",[Listen]),
            Master ! {self(), ok},
            New = start_accept(Listen, Fun),
            %% Now we're ready to run
            socket_loop(Listen, New, [], Fun, Max);
        Error ->
            Master ! {self(), Error}
    end.


socket_loop(Listen, New, Active, Fun, Max) ->
    receive
        {istarted, New} ->
            Active1 = [New|Active],
            possibly_start_another(false,Listen,Active1,Fun,Max);
        {'EXIT', New, _Why} ->
            %% io:format("Child exit=~p~n",[Why]),
            possibly_start_another(false,Listen,Active,Fun,Max);
        {'EXIT', Pid, _Why} ->
            %% io:format("Child exit=~p~n",[Why]),
            Active1 = lists:delete(Pid, Active),
            possibly_start_another(New,Listen,Active1,Fun,Max);
        {children, From} ->
            From ! {session_server, Active},
            socket_loop(Listen,New,Active,Fun,Max);
        _Other ->
            socket_loop(Listen,New,Active,Fun,Max)
    end.

possibly_start_another(New, Listen, Active, Fun, Max)
  when pid(New) ->
    socket_loop(Listen, New, Active, Fun, Max);
possibly_start_another(false, Listen, Active, Fun, Max) ->
    case length(Active) of
        N when N < Max ->
            New = start_accept(Listen, Fun),
            socket_loop(Listen, New, Active, Fun,Max);
        _ ->
            socket_loop(Listen, false, Active, Fun, Max)
    end.

start_accept(Listen, Fun) ->
    S = self(),
    spawn_link(fun() -> start_child(S, Listen, Fun) end).

start_child(Parent, Listen, Fun) ->
    case gen_tcp:accept(Listen) of
        {ok, Socket} ->
            Parent ! {istarted,self()},              % tell the controller
            inet:setopts(Socket, [{packet,4},
                                  binary,
                                  {nodelay,true},
                                  {active, true}]),
```

411

```
            %% before we activate socket
            %% io:format("running the child:~p Fun=~p~n", [Socket, Fun]),
            process_flag(trap_exit, true),
            case (catch Fun(Socket)) of
                {'EXIT', normal} ->
                    true;
                {'EXIT', Why} ->
                    io:format("Port process dies with exit:~p~n",[Why]),
                    true;
                _ ->
                    %% not an exit so everything's ok
                    true
            end
    end.
```

D.3.3 lib_chan_mm

socket_dist/lib_chan_mm.erl

```
%% Protocol
%%   To the controlling process
%%      {chan, MM, Term}
%%      {chan_closed, MM}
%%   From any process
%%      {send, Term}
%%      close
```

412

```
-module(lib_chan_mm).

%% TCP Middle man
%%   Models the interface to gen_tcp

-export([loop/2, send/2, close/1, controller/2, set_trace/2, trace_with_tag/2]).

send(Pid, Term)       -> Pid ! {send, Term}.
close(Pid)            -> Pid ! close.
controller(Pid, Pid1) -> Pid ! {setController, Pid1}.
set_trace(Pid, X)     -> Pid ! {trace, X}.

trace_with_tag(Pid, Tag) ->
    set_trace(Pid, {true,
                    fun(Msg) ->
                            io:format("MM:~p ~p~n",[Tag, Msg])
                    end}).

loop(Socket, Pid) ->
    %% trace_with_tag(self(), trace),
    process_flag(trap_exit, true),
    loop1(Socket, Pid, false).

loop1(Socket, Pid, Trace) ->
    receive
        {tcp, Socket, Bin} ->
            Term = binary_to_term(Bin),
            trace_it(Trace,{socketReceived, Term}),
            Pid ! {chan, self(), Term},
```

```
            loop1(Socket, Pid, Trace);
        {tcp_closed, Socket} ->
            trace_it(Trace, socketClosed),
            Pid ! {chan_closed, self()};
        {'EXIT', Pid, Why} ->
            trace_it(Trace,{controllingProcessExit, Why}),
            gen_tcp:close(Socket);
        {setController, Pid1} ->
            trace_it(Trace, {changedController, Pid}),
            loop1(Socket, Pid1, Trace);
        {trace, Trace1} ->
            trace_it(Trace, {setTrace, Trace1}),
            loop1(Socket, Pid, Trace1);
        close ->
            trace_it(Trace, closedByClient),
            gen_tcp:close(Socket);
        {send, Term}  ->
            trace_it(Trace, {sendingMessage, Term}),
            gen_tcp:send(Socket, term_to_binary(Term)),
            loop1(Socket, Pid, Trace);
        UUg ->
            io:format("lib_chan_mm: protocol error:~p~n",[UUg]),
            loop1(Socket, Pid, Trace)
    end.

trace_it(false, _)     -> void;
trace_it({true, F}, M) -> F(M).
```

D.3.4 lib_chan_auth

socket_dist/lib_chan_auth.erl

```
-module(lib_chan_auth).

-export([make_challenge/0, make_response/2, is_response_correct/3]).

make_challenge() ->
    random_string(25).

make_response(Challenge, Secret) ->
    lib_md5:string(Challenge ++ Secret).

is_response_correct(Challenge, Response, Secret) ->
    case lib_md5:string(Challenge ++ Secret) of
        Response -> true;
        _        -> false
    end.

%% random_string(N) -> a random string with N characters.

random_string(N) -> random_seed(), random_string(N, []).

random_string(0, D) -> D;
random_string(N, D) ->
    random_string(N-1, [random:uniform(26)-1+$a|D]).
```

413

```
random_seed() ->
    {_,_,X} = erlang:now(),
    {H,M,S} = time(),
    H1 = H * X rem 32767,
    M1 = M * X rem 32767,
    S1 = S * X rem 32767,
    put(random_seed, {H1,M1,S1}).
```

414

附录 E 其　他

本附录为你展现如何分析和调试代码，它还详细地描述了动态代码加载是如何工作的。

E.1 分析和评估工具

如何考察一个正在运行中的程序，查找程序中可能存在的缺陷、定位性能瓶颈、识别无用代码、统计对于不推荐方法的调用？这些都是让人挠头的问题，本节专门就来解决这些问题。

E.1.1 代码覆盖

在我们对代码进行测试的时候，不仅希望能了解哪些代码会被执行，更希望了解到哪些代码没有包括在我们的测试当中。这些没有被执行到的代码，可能隐含潜在的错误，找出它们无疑是很有帮助的事情。这就是代码覆盖分析程序所做的事情。

这是一个例子：

```
1> cover:start().          %% start the coverage analyser
{ok,<0.34.0>}
2> cover:compile(shout).   %% compile shout.erl for coverage
{ok,shout}
3> shout:start().          %% run the program
<0.41.0>
Playing:<<"title: track018 performer: .. ">>
4> %% let the program run for a bit
4>  cover:analyse_to_file(shout).  %% analyse the results
{ok,"shout.COVER.out"}             %% this is the results file
```

分析结果在一个文件中，它的内容如下：

```
     ...
     |  send_file(S, Header, OffSet, Stop, Socket, SoFar) ->
     |      %% OffSet = first byte to play
     |      %% Stop   = The last byte we can play
131..|      Need = ?CHUNKSIZE - size(SoFar),
131..|      Last = OffSet + Need,
131..|      if
     |       Last >= Stop ->
     |           %% not enough data so read as much as possible and return
  0..|           Max = Stop - OffSet,
  0..|           {ok, Bin} = file:pread(S, OffSet, Max),
  0..|           list_to_binary([SoFar, Bin]);
     |       true ->
131..|           {ok, Bin} = file:pread(S, OffSet, Need),
```

```
131..|          write_data(Socket, SoFar, Bin, Header),
131..|          send_file(S, bump(Header),
     |                    OffSet + Need,  Stop, Socket, <<>>)
     |     end.
     ...
```

留意这个文件最左边的列，我们看到的数字是每一行代码被执行的次数。标记为0的那些行要特别留意。这些代码没有被执行过，因此，我们也无法保证程序到底是不是正确的。

这意味着我们需要修改测试用例，以保证所有的代码行都被覆盖。这是系统化的测试中很有价值的方法。它能帮助我们发现可能的隐含缺陷，提高测试质量。

最佳测试方法

进行代码的覆盖分析能够回答这个问题。哪些行代码一直没有在测试中被执行到？一旦我们知道了这个问题的答案，我们就能改进测试用例，确保这些行的代码被执行。

对付那些难于重现或者在你意料之外的bug，这是一个屡试不爽的方法。每一行没有测到的代码都有可能包含错误。保证每一样代码都被测到，这是我所知道的最好的测试方法。

我在最早的Erlang JAM[①]编译器上采用了这种方法。两年之中，只收到了3个bug报告，而在这之后，再也没有收到报告了。

416

E.1.2　性能评估

标准的Erlang发布版本已经自带了3个性能评估工具。

- cprof测试每个函数被调用了多少次。这个工具较为轻量在运行系统上使用这个工具会给系统带来5%到10%的额外负载。
- fprof显示函数调用和被调用的时间，并将结果输出到一个文件中。这个工具比较适合于在实验环境或模拟环境中对系统进行大规模的性能评估。它会带来非常显著的系统负载。
- eprof会分析一个Erlang程序中计算时间主要消耗在哪里。它实际上是fprof的前任，所不同的是它比较适合规模小一些的性能评估。

这里是一个使用cprof的例子：

```
1> cprof:start().          %% start the profiler
4501
2> shout:start().          %% run the application
<0.35.0>
3> cprof:pause().          %% pause the profiler
4844
4> cprof:analyse(shout). %% analyse function calls
{shout,232,
   [{{shout,split,2},73},
    {{shout,write_data,4},33},
    {{shout,the_header,1},33},
    {{shout,send_file,6},33},
    {{shout,bump,1},32},
    {{shout,make_header1,1},5},
    {{shout,'-got_request_from_client/3-fun-0-',1},4},
```

① JAM是Joe's Abstract Machine的缩写，它是Erlang的第一个编译器。

```
    {{shout,songs_loop,1},2},
    {{shout,par_connect,2},2},
    {{shout,unpack_song_descriptor,1},1},
    ...
5> cprof:stop().          %% stop the profiler
4865
```

此外，cprof:analyse()还能分析所有模块的函数调用，只要是在测量期间收集到了相关的信息。

若想了解cprof工具的详细信息，请参看http://www.erlang.org/doc/man/cprof.html。

E.1.3 xref

xref模块可以生成交叉引用。它需要在代码在编译时启用了debug_info的编译开关。

在开发中，时不时的对代码进行交叉引用检查，这是一个不错的点子。不过，在这里我没法向你展现对于这本书中代码的xref输出结果，因为，一方面整个开发都已经结束，而另一方面，这些代码之中也不存在函数缺失的情况。我这里会展示对另一个项目的代码进行交叉引用检查的结果。 417

vsg是我的业余项目之一，它是一个简单的图形程序，或许某天我会把它发布出来。我在vsg的开发目录里做一次xref分析，就像这样：

```
$ cd /home/joe/2007/vsg-1.6
$ rm *.beam
$ erlc +debug_info *.erl
$ erl
1> xref:d('.')
[{deprecated,[]},
 {undefined,[{{new,win1,0},{wish_manager,on_destroy,2}},
             {{vsg,alpha_tag,0},{wish_manager,new_index,0}},
             {{vsg,call,1},{wish,cmd,1}},
             {{vsg,cast,1},{wish,cast,1}},
             {{vsg,mkWindow,7},{wish,start,0}},
             {{vsg,new_tag,0},{wish_manager,new_index,0}},
             {{vsg,new_win_name,0},{wish_manager,new_index,0}},
             {{vsg,on_click,2},{wish_manager,bind_event,2}},
             {{vsg,on_move,2},{wish_manager,bind_event,2}},
             {{vsg,on_move,2},{wish_manager,bind_tag,2}},
             {{vsg,on_move,2},{wish_manager,new_index,0}}]},
 {unused,[{vsg,new_tag,0},
          {vsg_indicator_box,theValue,1},
          {vsg_indicator_box,theValue,1}]}]
```

xref:d('.')对当前目录下的所有代码执行交叉引用分析，这些代码必须都是使用了调试标志编译出来的。它会生成一个列表，显示depreciated（不建议使用的）、undefined（未定义的）和unused（未使用的）的函数。

和其他大多数的工具一样，xref也有一大堆的选项。如果你想掌握好这个特性强大的工具，那就多去读一读它营养丰富的手册。

E.2 调试

调试Erlang程序非常容易，以至于容易到超乎你的想象，这在很大程度上要归功于Erlang的“单次变量绑定”特性。因为Erlang中并没有指针或者可变的状态（除了ETS表和进程字典），所以，要想知道是哪里出了问题时，会变得异常简单。而一旦我们观察到了某个变量不正常的值，找到相应故障发生的时间和地 418

点，也就变得易如反掌。

在我写C程序的时候，我觉得调试器是相当有用的工具。因为，我可以让它监控变量，当变量发生改变的时候，我就会知道，这很重要，因为在C语言中，内存不一定是直接被改变的，比如说，还能通过指针来进行间接的改动。这使得你很难来判断某段内存中的数据究竟是被哪里的代码改掉的。而在Erlang的调试器中，并不需要提供这样的功能，因为Erlang中根本没有“通过指针来改变状态”这种事。

下面的小节，我们会看到Erlang编译器的代码检查，以及调试一个程序的不同方法。它们按照难易程度排列。利用日志输出来调试是调试程序最为简单的方法，跟踪一个进程是最复杂的。我们先从简单的开始。

E.2.1　编译检查

编译时，如果我们的程序有语法错误，编译器会给出明确的错误提示。大部分的提示都是明确易懂的，比如，我们丢了括号、逗号或者是语言的关键字，此时编译器会对出问题的语句给出精确到文件名和行号的错误提示。下面就是一些我们可能会看到的错误。

1．函数头不匹配

如果一个函数的子句，其定义的名称或者参数个数没有与前面的定义保持一致，就会产生这个错误：

bad.erl

```
Line 1  foo(1,2) ->
            a;
        foo(2,3,a) ->
            b.

        1> c(bad).
        ./bad.erl:3: head mismatch
```

2．未绑定的变量

下面的代码中包含未绑定的变量：

bad.erl

```
Line 1  foo(A, B) ->
            bar(A, dothis(X), B),
            baz(Y, X).

        1> c(bad).
        ./bad.erl:2: variable 'X' is unbound
        ./bad.erl:3: variable 'Y' is unbound
```

419

上面的错误提示表明，在第2行之中的变量X没有值。这个错误并非真正是发生在第2行的，只是在第2行才被检测出来。第3行也用到了X这个变量，但编译器只会在这个错误第一次发生时报告。

3．未终结的字符串

未终结的字符串以"..."开始。

如果我们忘记了一个字符串或者原子的引号，我们就会得到这个错误消息。有些时候，要费好半天的功夫才能找出哪里丢了引号。如果因为这个错误无法编译通过，但又实在找不出哪里丢了引号，你可以试着在代码中的任意位置（或者在你认为有问题的地方加引号，这更好）加上一个引号。这样，再次编译时，给出的错误信息可能更精确，而这会帮助你最终找到问题的根源。

4．不安全的变量

如果我们编译如下的代码：

bad.erl

```
Line 1  foo() ->
            case bar() of
                1 ->
                    X = 1,
                    Y = 2;
                2 ->
                    X = 3
            end,
            b(X).
```

我们会得到一个这样的警告：

```
1> c(bad).
./bad.erl:5: Warning: variable 'Y' is unused
{ok,bad}
```

这只是一个警告，程序中定义到的Y并没有被用到。如果我们把代码改成这样：

bad.erl

```
Line 1  foo() ->
            case bar() of
                1 ->
                    X = 1,
                    Y = 2;
                2 ->
                    X = 3
            end,
            b(X, Y).
```

我们则会得到一个错误信息：

420

```
> c(bad).
./bad.erl:9: variable 'Y' unsafe in 'case' (line 2)
{ok,bad}
```

此时，编译器会认为程序有可能会走case表达式的第二个分支(在这种情况下，变量Y就没有被定义过)，这时它会报告一个“不安全的变量”错误。

5. 影子变量

bad.erl

```
Line 1  foo(X, L) ->
            lists:map(fun(X) -> 2*X end, L).
```

```
1> c(bad).
./bad.erl:1: Warning: variable 'X' is unused
./bad.erl:2: Warning: variable 'X' shadowed in 'fun'
{ok,bad}
```

这种情况下，编译器认为我们的代码很可能隐含了一些模棱两可的意图，它用这个警告信息提醒我们注意这些代码。上面的代码，在fun的内部我们计算了2*X，但是这个X究竟是fun的参数还是foo函数的参数呢？

如果遇到这种情况，最好的办法是将这两处X中的一个改成其他的名字。我们可以这样改写：

bad.erl

```
foo(X, L) ->
    lists:map(fun(Z) -> 2*Z end, L).
```

现在就没问题了，我们可以在fun的定义当中使用X这个变量，而不会引起混淆。

6. 运行时的检查

当一个Erlang进程崩溃的时候，我们会得到一个错误消息。要看到这个错误消息，需要由另外一个进程来监控这个会崩溃的进程，并在崩溃发生的时候把错误信息显示出来。如果我们只是简单的通过spawn建立一个进程，当它崩溃的时候，我们是不会得到任何错误信息的。如果我们想看到所有这些错误信息，最好的办法是用spawn_link来创建进程。

7. 栈跟踪

421 对于那些由shell监控的进程来说，每个崩溃都会打印出崩溃时的栈跟踪信息。

为了展示栈跟踪都有些什么信息，我们会先写一个简单的函数，它有意留了一个错误，然后我们在shell中调用这个函数，就像这样：

lib_misc.erl

```
deliberate_error(A) ->
    bad_function(A, 12),
    lists:reverse(A).

bad_function(A, _) ->
    {ok, Bin} = file:open({abc,123}, A),
    binary_to_list(Bin).
```

```
1> lib_misc:deliberate_error("file.erl").
** exited: {{badmatch,{error,einval}},
            [{lib_misc,bad_function,2},
             {lib_misc,deliberate_error,1},
             {erl_eval,do_apply,5},
             {shell,exprs,6},
             {shell,eval_loop,3}]} **
```

当调用lib_misc:deliberate_error("file.erl")的时候，果然发生了错误，我们得到了想要的栈跟踪。在栈的顶部，我们看到了这么一行：

```
{badmatch, {error, einval}}
```

它来自于这个代码：

```
{ok, Bin} = file:open({abc,123}, A)
```

调用file:open/2返回了{error, einval}。[①]得到这个错误，是因为{abc, 123}并不是file:open的一个合法输入参数。那么，在我们试图把这个结果与{ok, Bin}进行匹配的时候，我们就会触发badmatch错误。运行时系统会输出{badmatch, {error, einval}}，这是在进行匹配时产生了错误的值。接下来的内容就是栈跟踪了，这些信息的第一条是错误所在的函数名，之后是这个函数执行完毕后依次将会返回的函数名。也就是说，错误发生在lib_misc:bad_funtion/2中，函数本应返回lib_misc:deliberate_error/1，等等。

注意，这里的栈跟踪信息只有最顶上的一条信息值得关注。如果对于问题函数的调用来自于尾递归的调用，这样的调用是不会出现在栈跟踪当中的。这种情况可以通过对deliberate_error函数进行重命名并做一些小的修改来予以呈现：

lib_misc.erl

```
deliberate_error1(A) ->
    bad_function(A, 12).
```

422

① einval是一个POSIX规范的错误代码，它是invalid value的缩写。

当我们调用这段代码发生错误时，deliberate_error1不会出现在栈跟踪信息中。

```
2> lib_misc:deliberate_error1("file.erl").
** exited: {{badmatch,{error,einval}},
            [{lib_misc,bad_function,2},
             {erl_eval,do_apply,5},
             {shell,exprs,6},
             {shell,eval_loop,3}]} **
```

deliberate_error1的调用不在栈跟踪信息中是因为，对bad_function的调用是deliber- ate_error1的最后一条语句，而当它结束时，也不需要返回deliberate_error1，而是会返回给它的调用者。

（这是因为Erlang进行了所谓的“尾递归优化”。如果某个函数的最后一个语句是调用另外一个函数，那么这个调用会被替换为更有效率的跳转。如果没有这种优化，诸如等待接收信息的代码之中出现过的那种无限循环的编码风格将无法工作。这样的优化方式下，这个函数调用者也被替换成了更外层的调用者，因而，它在栈跟踪里面也就变成不可见的了。）

E.2.2 调试技术

Erlang程序员可以使用多种方法来调试他们的程序。最为常用的技术是在不正常的程序中添加输出语句。当你所关心的数据结构变得异常庞大时，这种方法就会变得不敷应用，这个时候，还可以将信息导到文件当中，以备日后检查。

一些人用错误日志来保存错误信息，另一些人则喜欢转存到文件当中。如果这些都不管用，我们还可以使用Erlang的调试器或者跟踪程序的执行。下面我们来看这些方法。

1. io:format调试

在程序中加入输出语句是最为常见的调试方法。你只需简单的把io:format(...)语句加到程序中的合适位置，在特定的情况下，将你感兴趣的东西输出到屏幕上就可以了。

在调试并发程序时，发送消息之前（或刚刚接到消息的时候）将它们即时输出到屏幕上，这常常也是一种很不错的调试手段。

423

在我写一个并发程序时，我通常是从一个接收消息的循环开始，代码就像这样：

```
loop(...) ->
    receive
       Any ->
           io:format("*** warning unexpected message:~p~n",[Any])
           loop(...)
    end
```

之后，随着我不断的向这个接收循环增加新的匹配模式，如果这个进程收到无法理解的消息，我总能接到警告信息。我也会使用spawn_link来代替spawn，以保证一旦进程非正常的结束，我总能获得错误信息。

我常常会使用一个名为NYI（Not Yet Implemented）的宏。它的定义是这样的：

lib_misc.erl

```
-define(NYI(X),(begin
                    io:format("*** NYI ~p ~p ~p~n",[?MODULE, ?LINE, X]),
                    exit(nyi)
                end)).
```

这样，我可以在需要的地方这么用它：

lib_misc.erl

```
glurk(X, Y) ->
    ?NYI({glurk, X, Y}).
```

我还没有写glurk函数的内部代码，所以，如果此时我调用glurk函数，程序就会崩溃：

```
> lib_misc:glurk(1,2).
*** NYI lib_misc 83 {glurk,1,2}
** exited: nyi *
```

这会导致进程退出，并显示一个错误信息，这样我就知道必须要先完成这个函数。

2. 转存到文件

如果我们关注的数据结构比较大，我们可以把它写到一个文件中，这可以通过一个简单的dump/2来实现：

lib_misc.erl

```
dump(File, Term) ->
    Out = File ++ ".tmp",
    io:format("** dumping to ~s~n",[Out]),
    {ok, S} = file:open(Out, [write]),
    io:format(S, "~p.~n",[Term]),
    file:close(S).
```

424

它会显示一个警告信息，提醒我们一个新的文件已经生成。它会在文件名后增加.tmp扩展名（这样，日后我们就可以很容易的删掉这些临时文件）。此外，它还会将我们感兴趣的变量以良好的格式输出到文件中。我们可以稍后用一个文本编辑器来慢慢检查这个文件。这是一个非常有效的方法，而且相当简单，尤其是在我们需要检查比较大的数据结构时，这个方法能帮上大忙。

3. 使用错误日志

我们可以利用错误日志并创建一个文本文件来保存这些调试输出。我们可以通过一个配置文件来达到这个目的：

elog5.config

```
%% text erorr log
[ {kernel,
  [{error_logger,
    {file, "/home/joe/error_logs/debug.log"}}]}].
```

然后，我们通过下面的命令来启动Erlang：

```
erl -config elog5.config
```

所有调用error_logger模块中的函数所产生的错误信息，以及所有在shell中输出的错误信息都会汇集到配置文件所指定的文件当中。

E.2.3　调试器

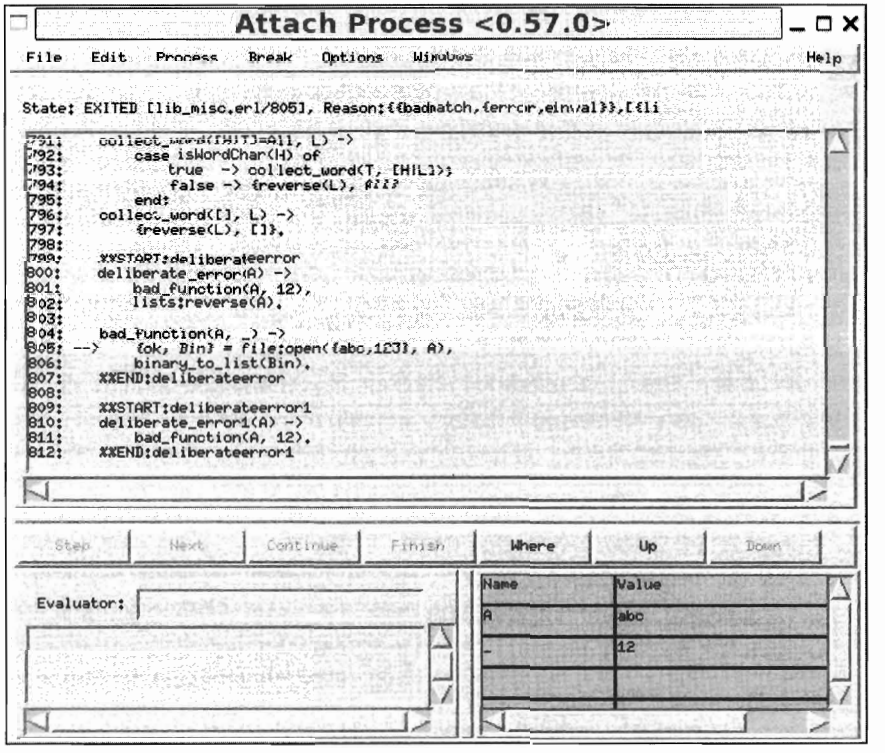
Erlang的标准发布版本包含一个调试器。这里我并不准备对它进行更深入的探讨，只会告诉你如何启动它，然后给你指出文档的位置。启动之后，它的使用是相当简单的。你可以在其中检查变量、执行单步跟踪和设置断点等。我们常常都需要同时调试多个进程，调试器还可以启动自己的多个实例，这样我们就可以有多个调试窗口，每个需要调试的进程都可以有自己的窗口。

只是启动调试器本身稍微有点烦琐：

```
1> %% recompile lib_misc so we can debug it
1>  c(lib_misc, [debug_info]).
{ok, lib_misc}
2> im().  %% A window will pop up. Ignore it for now
```

```
<0.42.0>
3> ii(lib_misc).
{module,lib_misc}
4> iaa([init]).
true.
5> lib_misc:
...
```

425



图E-1 调试器[①]初始屏幕

运行上面的代码会出现一个图E-1中那样的窗口。

上述那些没有模块前缀的命令（ii/1、iaa/1之类的）都是i模块公开的函数。这个i模块是调试器/解释器的接口模块。在shell中使用时，可以不带前缀直接使用。

我们可以调用函数让调试器运行：

```
im()
```

启动一个图形监控器。这是调试器的主窗口，它会显示调试器监控到的所有进程的状态。

426

```
ii(Mod)
```

解释Mod模块的代码。

```
iaa([init])
```

① Table Viewer是表查看器，怀疑这里作者写错，应该是调试器才对。

任何被解释过的代码，当用这些代码来启动一个进程时，将调试器附加到此进程之上。

更进一步

想了解调试的更多信息，可以参考“调试器参考手册”http://www.erlang.org/doc/pdf/debugger. pdf（46页的PDF文件）。它包含截屏图像、API文档以及其他更多的内容。对于每一个想要深入使用调试器的用户，这是一份必读文档。在http://www.erlang.org/doc/man/i.html当中则可以找到更多可以在shell中使用的调试命令。

E.3　跟踪

即便没有特别的编译你的代码，你仍然可以跟踪一个进程的运行。对一个进程（或者一堆进程）进行跟踪，是一种强有力的调试手段。通过跟踪你可以了解你的系统具体的行为是怎样的，在调试复杂的系统时，尤其是当你又不能去更改源码的情况下，这个手段尤其有用。比如说，要在嵌入式环境下进行调试，或者调试没有源码的程序时，这个方法能够派上大用场。

在比较低的层面上，我们可以通过调用几个Erlang的BIF（内建函数）来设置跟踪。而当要用这些BIF来处理比较复杂的跟踪情况时，则会变得非常困难。为了解决这个问题，已经开发了好几个库来简化这一过程。

下面，我们先来看看如何使用这些底层的BIF来设置跟踪，然后再来评估那些提供了高层跟踪接口的库。

对于底层的跟踪来说，两个内建的函数最为重要。`erlang:trace/3`意味着“我想要监控这个进程，如
427 果我感兴趣的事情发生，请给我发消息”，`erlang:trace_pattern`则定义了什么样的事情“我感兴趣”。

```
erlang:trace(PidSpec, How, FlagList)
```

启动跟踪。`PidSpec`告诉系统要跟踪哪个进程，`How`是一个布尔值，它可以打开或者关闭跟踪，`FlagList`决定要跟踪些什么（比如，可以跟踪所有的函数调用，所有发送的消息，何时进行垃圾回收，等等）。

一旦调用了`erlang:trace/3`，调用者进程就会在跟踪事件发生时收到跟踪消息。跟踪事件本身会调用`erlang:trace_pattern/3`来进行检查。

```
erlang:trace_pattern(MFA, MatchSpec, FlagList)
```

这个函数用来设置一个跟踪模式。如果匹配了模式，相应的动作就会被执行。这里的`MFA`是一个`{Module, Function, Args}`元组，它表明了跟踪模式要匹配什么代码，`MatchSpec`是一个模式，每次进入由`MFA`标明的函数，当模式所定义的条件被满足时都会被测试，`FlagList`意味着将要执行的动作。

`MatchSpec`就是匹配模式，手工来编写`MatchSpec`是相当复杂的，而花费大量的时间来理解它也无助于增进我们对这一过程的理解。幸好有几个库[①]让我们大大地简化了这个步骤。

使用上面提到的两个BIF，我们就可以写出简单的跟踪。`trace_module(Mod, Fun)`在`Mod`模块上设置了跟踪，然后执行`Fun()`。我们希望跟踪到在`Mod`模块中所有函数的调用及其返回值。

tracer_test.erl

```
trace_module(Mod, StartFun) ->
    %% We'll spawn a process to do the tracing
    spawn(fun() -> trace_module1(Mod, StartFun) end).

trace_module1(Mod, StartFun) ->
    %% The next line says: trace all function calls and return
```

① http://www.erlang.org/doc/man/ms_transform.html。

```
    %%                      values in Mod
    erlang:trace_pattern({Mod, '_','_'},
                         [{'_',[],[{return_trace}]}],
                         [local]),
    %% spawn a function to do the tracing
    S = self(),
    Pid = spawn(fun() -> do_trace(S, StartFun) end),
    %% setup the trace. Tell the system to start tracing
    %% the process Pid
    erlang:trace(Pid, true, [call,procs]),
    %% Now tell Pid to start
    Pid ! {self(), start},
    trace_loop().

%% do_trace evaluates StartFun()
%%    when it is told to do so by Parent
do_trace(Parent, StartFun) ->
    receive
        {Parent, start} ->
            StartFun()
    end.

%% trace_loop displays the function call and return values
trace_loop() ->
    receive
        {trace,_,call, X} ->
            io:format("Call: ~p~n",[X]),
            trace_loop();
        {trace,_,return_from, Call, Ret} ->
            io:format("Return From: ~p => ~p~n",[Call, Ret]),
            trace_loop();
        Other ->
            %% we get some other message - print them
            io:format("Other = ~p~n",[Other]),
            trace_loop()
    end.
```

428

现在我们再来定义一个测试程序：

tracer_test.erl

```
test2() ->
    trace_module(tracer_test, fun() -> fib(4) end).

fib(0) -> 1;
fib(1) -> 1;
fib(N) -> fib(N-1) + fib(N-2).
```

然后，我们就可以进行跟踪了：

```
1> c(tracer_test).
{ok,tracer_test}
2> tracer_test:test2().
<0.42.0>Call: {tracer_test,'-trace_module1/2-fun-0-',
               [<0.42.0>,#Fun<tracer_test.0.36786085>]}
Call: {tracer_test,do_trace,[<0.42.0>,#Fun<tracer_test.0.36786085>]}
Call: {tracer_test,'-test2/0-fun-0-',[]}
```

```
Call: {tracer_test,fib,[4]}
Call: {tracer_test,fib,[3]}
Call: {tracer_test,fib,[2]}
Call: {tracer_test,fib,[1]}
Return From: {tracer_test,fib,1} => 1
Call: {tracer_test,fib,[0]}
Return From: {tracer_test,fib,1} => 1
Return From: {tracer_test,fib,1} => 2
Call: {tracer_test,fib,[1]}
Return From: {tracer_test,fib,1} => 1
Return From: {tracer_test,fib,1} => 3
Call: {tracer_test,fib,[2]}
Call: {tracer_test,fib,[1]}
Return From: {tracer_test,fib,1} => 1
Call: {tracer_test,fib,[0]}
Return From: {tracer_test,fib,1} => 1
Return From: {tracer_test,fib,1} => 2
Return From: {tracer_test,fib,1} => 5
Return From: {tracer_test,'-test2/0-fun-0-',0} => 5
Return From: {tracer_test,do_trace,2} => 5
Return From: {tracer_test,'-trace_module1/2-fun-0-',2} => 5
Other = {trace,<0.43.0>,exit,normal}
```

E.3.1 使用库

使用**dbg**这个库模块，我们可以达到与上面的例子相同的效果。它隐藏了调用低层Erlang BIF所带来的烦琐细节。

tracer_test.erl

```
test1() ->
    dbg:tracer(),
    dbg:tpl(tracer_test,fib,'_',
            dbg:fun2ms(fun(_) -> return_trace() end)),
    dbg:p(all,[c]),
    tracer_test:fib(4).
```

运行这个程序，我们会获得如下的输出：

```
1> tracer_test:test1().
(<0.34.0>) call tracer_test:fib(4)
(<0.34.0>) call tracer_test:fib(3)
(<0.34.0>) call tracer_test:fib(2)
(<0.34.0>) call tracer_test:fib(1)
(<0.34.0>) returned from tracer_test:fib/1 -> 1
(<0.34.0>) call tracer_test:fib(0)
(<0.34.0>) returned from tracer_test:fib/1 -> 1
(<0.34.0>) returned from tracer_test:fib/1 -> 2
(<0.34.0>) call tracer_test:fib(1)
(<0.34.0>) returned from tracer_test:fib/1 -> 1
(<0.34.0>) returned from tracer_test:fib/1 -> 3
(<0.34.0>) call tracer_test:fib(2)
(<0.34.0>) call tracer_test:fib(1)
(<0.34.0>) returned from tracer_test:fib/1 -> 1
(<0.34.0>) call tracer_test:fib(0)
(<0.34.0>) returned from tracer_test:fib/1 -> 1
```

```
(<0.34.0>) returned from tracer_test:fib/1 -> 2
(<0.34.0>) returned from tracer_test:fib/1 -> 5
```

E.3.2 更进一步

想了解更多的跟踪机制，你需要阅读如下3个模块的手册。

- dbg提供了Erlang内建跟踪函数的一个简化接口。
- ttb是另外一个跟踪BIF的接口，它比dbg所在的层次还要再高一些。
- ms_transform构造匹配模式，可以在你自己的跟踪软件中使用它。

E.4 动态代码加载

动态代码加载可能是Erlang核心之中最让人惊讶的特性之一。更妙的是，你可以只是使用它，而无须关心这一切在后台到底是怎么发生的。

概念本身非常简单，就算在程序的运行过程之中重新编译了代码，我们都遵循这一简单的原则——每次调用某模块的某函数时，都去调用它的最新版本。

如果b此时正在循环之中，而我们又重新编译了b，那么在下一次调用b时，也会自动的去调用那个最新版本的b。

如果很多不同的进程都在运行，而它们都调用了b，那么如果b被重新编译，这些进程都会调用新版本的b。要了解这是怎么工作的，我们需要写两个小的模块，a和b。b非常简单：

b.erl

```
-module(b).
-export([x/0]).

x() -> 1.
```

然后是a：

a.erl

```
-module(a).
-compile(export_all).

 start(Tag) ->
    spawn(fun() -> loop(Tag) end).

loop(Tag) ->
    sleep(),
    Val = b:x(),
    io:format("Vsn1 (~p) b:x() = ~p~n",[Tag, Val]),
    loop(Tag).

sleep() ->
    receive
        after 3000 -> true
    end.
```

现在我们编译a，并启动几个a的进程：

```
1> c(b).
{ok, b}
2> c(a).
```

```
{ok, a}
3> a:start(one).
<0.41.0>
Vsn1 (one) b:x() = 1
4> a:start(two).
<0.43.0>
Vsn1 (one) b:x() = 1
Vsn1 (two) b:x() = 1
Vsn1 (one) b:x() = 1
Vsn1 (two) b:x() = 1
```

这个进程会休眠3秒、醒来、调用b:x()，把结果输出到屏幕，然后再次休眠。现在我们打开编辑器，把b改成下面这样：

```
-module(b).
-export([x/0]).

x() -> 2.
```

然后在shell中重新编译b，就会看到这样的输出：

```
4> c(b).
{ok,b}
Vsn1 (one) b:x() = 2
Vsn1 (two) b:x() = 2
Vsn1 (one) b:x() = 2
Vsn1 (two) b:x() = 2
...
```

两个最初版本的a仍在运行，但此时它们调用的是新版本的b。所以，当我们在a模块中调用b:x()的时候，我们实际上是在“调用最新版本的b”。我们可以多次的修改和重新编译b，所有调用它的模块都会自动的调用最新版本的b，这一切都是自动发生的，我们不需要做什么特别的事。

现在我们重新编译了b，但如果我们此时修改和重新编译a，会发生什么情况？我们下面就做这个实验，把a的代码这样修改：

```
-module(a).
-compile(export_all).
start(Tag) ->
    spawn(fun() -> loop(Tag) end).

loop(Tag) ->
    sleep(),
    Val = b:x(),
    io:format("Vsn2 (~p) b:x() = ~p~n",[Tag, Val]),
    loop(Tag).

sleep() ->
    receive
        after 3000 -> true
    end.
```

432

现在我们编译并启动a：

```
5> c(a).
{ok,a}
Vsn1 (one) b:x() = 2
Vsn1 (two) b:x() = 2
...
6> a:start(three).
```

```
<0.53.0>
Vsn1 (one) b:x() = 2
Vsn1 (two) b:x() = 2
Vsn2 (three) b:x() = 2
Vsn1 (one) b:x() = 2
Vsn1 (two) b:x() = 2
Vsn2 (three) b:x() = 2
...
```

有趣的事情发生了。当我们启动一个新版本的a时，我们看到新版本的a运行正常。而与此同时，之前启动的旧版本的a也仍然运转正常。

现在我们再来改变b的代码：

```
-module(b).
-export([x/0]).

x() -> 3.
```

然后，我们在shell中重新编译b，看看会发生什么：

```
7> c(b).
{ok,b}
Vsn1 (one) b:x() = 3
Vsn1 (two) b:x() = 3
Vsn2 (three) b:x() = 3
...
```

现在新版本和旧版本的a都在使用最新版本的b。

433

最后，我们再次改变a的代码（对于a来说，这是第3次改动）：

```
-module(a).
-compile(export_all).

start(Tag) ->
    spawn(fun() -> loop(Tag) end).

loop(Tag) ->
    sleep(),
    Val = b:x(),
    io:format("Vsn3 (~p) b:x() = ~p~n",[Tag, Val]),
    loop(Tag).

sleep() ->
    receive
        after 3000 -> true
    end.
```

现在我们重新编译a，然后启动一个新版本的a，我们会看到下面的输出：

```
8> c(a).
{ok,a}
Vsn2 (three) b:x() = 3
...
9> a:start(four).
<0.106.0>
Vsn2 (three) b:x() = 3
Vsn3 (four) b:x() = 3
Vsn2 (three) b:x() = 3
```

```
Vsn3 (four) b:x() = 3
...
```

输出包含了最后两个版本的a（版本2和版本3），运行版本1的a进程死了。

任何时刻Erlang只允许两个版本的a模块同时运行，即当前版本和一个旧版本。当我们重新编译一个模块时，任何运行着旧版本代码的进程会被终止，之前的当前版本会变成旧版本，而最新编译的版本会成为当前版本。你可以把这“两个版本代码的更替”想象成为“寄存器的移位动作”。增加一个新版本代码的时候，最老的版本就作废了。进程可以同时运行新版本和旧版本的代码。

434 阅读purge_module的文档[①]，可以了解更多信息。

① http://www.erlang.org/doc/man/erlang.html#purge_module/1。

附录F

模块和函数参考

本附录包括了对标准Erlang发行版的核心库和标准库中大部分模块（及其函数）的一句话评论（我这里没有包括那些比较“生僻”的模块，砍掉了这些篇幅，本书的重量总算可以保持在可以接受的范围内）。

说明 ordsets和orddict模块没有包括在这个附录当中。ordsets提供的函数和sets一样，唯一的区别是它用一个有序的列表来表示集合当中的元素。orddict则和dict相似，所不同在于它使用一个键值对的列表来表示字典，而这个列表按键值来排序。

F.1 application模块

通用OTP应用程序的方函数。

```
Module:config_change(Changed, New, Removed) -> ok
```

配置参数发生变化的回调函数。

```
Module:prep_stop(State) -> NewState
```

应用程序准备停止的回调函数。

```
Module:start(StartType, StartArgs) -> {ok, Pid} | {ok, Pid, State} | {error, Reason}
```

应用程序被启动的回调函数。

```
Module:start_phase(Phase, StartType, PhaseArgs) -> ok | {error, Reason}
```

应用程序在扩展方式下启动的回调函数。

```
Module:stop(State)
```

应用程序被停止，执行清除工作的回调函数。

```
get_all_env(Application) -> Env
```

获取Application应用程序的所有配置。


```
get_all_key(Application) -> {ok, Keys} | undefined
```

获取Application应用程序所有配置的键值。

```
get_application(Pid | Module) -> {ok, Application} | undefined
```

获取Pid进程（或Module模块）所属应用程序的名称。

```
get_env(Application, Par) -> {ok, Val} | undefined
```

获取Application应用程序Par配置参数的值。

```
get_key(Application, Key) -> {ok, Val} | undefined
```

获取Application应用程序Key参数的值。

```
load(AppDescr, Distributed) -> ok | {error, Reason}
```

以Distrubuted方式加载AppDescr应用程序。

```
loaded_applications() -> [{Application, Description, Vsn}]
```

获取当前已经加载应用程序的列表。

```
permit(Application, Bool) -> ok | {error, Reason}
```

将当前结点上Application应用程序的运行权限改为Bool。

```
set_env(Application, Par, Val, Timeout) -> ok
```

将Appliction应用程序的Par配置参数设置为Val。

```
start(Application, Type) -> ok | {error, Reason}
```

加载并以Type方式启动Application应用程序。

```
start_type() -> StartType | local | undefined
```

获取当前进程所属应用程序的启动类型。

```
stop(Application) -> ok | {error, Reason}
```

停止Application应用程序。

```
takeover(Application, Type) -> ok | {error, Reason}
```

以Type方式接管Appliction应用程序。

```
unload(Application) -> ok | {error, Reason}
```

卸载Appliction应用程序。

```
unset_env(Application, Par, Timeout) -> ok
```

清除Application应用程序的Par配置参数。

```
which_applications(Timeout) -> [{Application, Description, Vsn}]
```

获取当前正在运行的应用程序的列表。

F.2　base64模块

实现base 64的编码解码算法，具体细节参考RFC 2045。

```
encode_to_string(Data) -> Base64String
```

对Data数据进行base 64编码。

```
mime_decode_string(Base64) -> DataString
```

436 对以base 64编码过的Base64数据进行解码。

F.3　beam_lib模块

操作BEAM文件格式的接口函数。

```
chunks(Beam, [ChunkRef]) -> {ok, {Module, [ChunkData]}} | {error, beam_lib, Reason}
```

从BEAM文件或二进制数据中读取由[ChunkRef]指定的块。

```
chunks(Beam, [ChunkRef], [Option]) -> {ok, {Module, [ChunkResult]}} | {error,
   beam_lib,Reason}
```

从BEAM文件或二进制数据中读取由[ChunkRef]指定的块（带[Option]属性读取）。

```
clear_crypto_key_fun() -> {ok, Result}
```

注销目前的加密key函数。

```
cmp(Beam1, Beam2) -> ok | {error, beam_lib, Reason}
```

比对Beam1数据和Beam2数据是否相同。

```
cmp_dirs(Dir1, Dir2) -> {Only1, Only2, Different} | {error, beam_lib, Reason1}
```

比对Dir1目录和Dir2目录中的BEAM文件是否相同。

```
crypto_key_fun(CryptoKeyFun) -> ok | {error, Reason}
```

将CryptoKeyFun注册为当前的加密key函数。

```
diff_dirs(Dir1, Dir2) -> ok | {error, beam_lib, Reason1}
```

比对Dir1目录和Dir2目录中的BEAM文件，寻找不同之处。

```
format_error(Reason) -> Chars
```

获取与Reason错误相对应的英文描述。

```
info(Beam) -> [{Item, Info}] | {error, beam_lib, Reason1}
```

获取Beam数据的信息。

```
md5(Beam) -> {ok, {Module, MD5}} | {error, beam_lib, Reason}
```

读取BEAM文件的模块版本。

```
strip(Beam1) -> {ok, {Module, Beam2}} | {error, beam_lib, Reason1}
```

去除Beam1数据中非必需的部分。

```
strip_files(Files) -> {ok, [{Module, Beam2}]} | {error, beam_lib, Reason1}
```

去除Files列表的 BEAM文件中非必需的部分。

```
strip_release(Dir) -> {ok, [{Module, Filename]}} | {error, beam_lib, Reason1}
```

去除Dir版本目录下，BEAM文件中非必需的部分。

```
version(Beam) -> {ok, {Module, [Version]}} | {error, beam_lib, Reason}
```

获取Beam数据的模块版本。

F.4 c模块

命令接口模块。

```
bt(Pid) -> void()
```

显示Pid进程的栈回溯信息。

```
c(File, Options) -> {ok, Module} | error
```

使用Options选项编译并加载File文件。

437

```
cd(Dir) -> void()
```

设置当前目录为Dir。

```
flush() -> void()
```

强制刷新所有发给shell的消息。

```
help() -> void()
```

显示帮助信息。

```
i(X, Y, Z) -> void()
```

显示Pid为<X.Y.Z>的进程信息。

```
l(Module) -> void()
```

加载或者重新加载Module模块。

```
lc(Files) -> ok
```

编译Files列表中的文件。

```
ls() -> void()
```

列出当前目录中的文件。

```
ls(Dir) -> void()
```

列出Dir目录中的文件。

```
m() -> void()
```

显示当前系统中已加载的模块。

```
m(Module) -> void()
```

显示Module模块的信息。

```
memory() -> [{Type, Size}]
```

显示当前系统的内存分配情况。

```
memory([Type]) -> [{Type, Size}]
```

显示[Type]列表中各种内存的分配情况。

```
nc(File, Options) -> {ok, Module} | error
```

在当前系统的所有的节点上编译并加载File文件。

```
ni() -> void()
```

显示当前系统的信息。

```
nl(Module) -> void()
```

在所有的节点上加载Module模块。

```
nregs() -> void()
```

显示当前系统中已经注册的进程。

```
pid(X, Y, Z) -> pid()
```

将X、Y、Z转换为PID（进程标识符）。

```
pwd() -> void()
```

输出当前工作目录。

```
q() -> void()
```

退出，等同于init:stop()。

```
xm(ModSpec) -> void()
```

438 对ModSpec指定的模块进行交叉引用检查，等同于xref:m/1。

```
y(File) -> YeccRet
```

根据File语法文件生成相应的LALR-1分析器。

```
y(File, Options) -> YeccRet
```

根据File语法文件带Options参数生成相应的LALR-1分析器。

F.5　calendar模块

在当地时间和UTC时间之间进行转换，以及其他的日期、星期、时间的转换。

```
date_to_gregorian_days(Year, Month, Day) -> Days
```

计算从公元0年到Year年Month月Day日之间的天数。

```
datetime_to_gregorian_seconds({Date, Time}) -> Seconds
```

计算从公元0年到Date日Time时之间的秒数。

```
day_of_the_week(Year,Month, Day) -> DayNumber
```

计算Year年Month月Day日是星期几。

```
gregorian_days_to_date(Days) -> Date
```

将Days天数（从公元0年开始计算）转换为日期。

```
gregorian_seconds_to_datetime(Seconds) -> {Date, Time}
```

将Seconds秒数（从公元0年开始计算）转换为日期和时间。

```
is_leap_year(Year) -> bool()
```

计算Year年是否为闰年。

```
last_day_of_the_month(Year, Month) -> int()
```

计算Year年的Month月有多少天。

```
local_time() -> {Date, Time}
```

获取当前的当地时间。

```
local_time_to_universal_time({Date1, Time1}) -> {Date2, Time2}
```

将当地时间转换为UTC时间（不建议使用）。

```
local_time_to_universal_time_dst({Date1, Time1}) -> [{Date, Time}]
```

将当地时间转换为UTC时间（上述函数的替代）。

```
now_to_datetime(Now) -> {Date, Time}
```

将Now时间值转换为（UTC时间的）日期和时间。

```
now_to_local_time(Now) -> {Date, Time}
```

将Now时间值转换为（当地时间的）日期和时间。

```
seconds_to_daystime(Seconds) -> {Days, Time}
```

将Seconds秒转换为日期和时间。

```
seconds_to_time(Seconds) -> Time
```

将Seconds秒转换为时间。

```
time_difference(T1, T2) -> {Days, Time}
```

比较T1和T2两个时间元组之间相差多少时间（不建议使用）。

```
time_to_seconds(Time) -> Seconds
```

计算从午夜起到Time时间的秒数。

439

```
universal_time() -> {Date, Time}
```

获取当前的UTC时间。

```
universal_time_to_local_time({Date1, Time1}) -> {Date2, Time2}
```

将UTC时间转换为当地时间。

```
valid_date(Year, Month, Day) -> bool()
```

检查Year年Month月Day日是不是合法的日期。

F.6 code模块

Erlang的代码服务器。

```
add_patha(Dir) -> true | {error, What}
```

将Dir目录添加到代码目录列表的前端。

```
add_pathsa(Dirs) -> ok
```

将Dirs列表中的目录添加到代码目录列表的前端。

```
add_pathsz(Dirs) -> ok
```

将Dirs列表中的目录添加到代码目录列表的末端。

```
add_pathz(Dir) -> true | {error, What}
```

将Dir目录添加到代码目录列表的末端。

```
all_loaded() -> [{Module, Loaded}]
```

获取所有已加载模块的列表。

```
clash() -> ok
```

显示命名冲突检查报告。

```
compiler_dir() -> string()
```

获取编译器的库文件目录。

```
del_path(Name | Dir) -> true | false | {error, What}
```

从代码目录列表中删除Dir目录（或Name名称）。

```
delete(Module) -> true | false
```

删除Module模块目前的代码。

```
ensure_loaded(Module) -> {module, Module} | {error, What}
```

确保Module模块已经被加载。

```
get_object_code(Module) -> {Module, Binary, Filename} | error
```

获取Module模块的BEAM代码。

```
get_path() -> Path
```

获取代码服务器当前的搜索路径。

```
is_loaded(Module) -> {file, Loaded} | false
```

检查Module模块是否已经加载。

```
lib_dir() -> string()
```

获取Erlang/OTP的库文件目录。

```
lib_dir(Name) -> string() | {error, bad_name}
```

440

获取Name应用程序的库文件目录。

```
load_abs(Filename) -> {module, Module} | {error, What}
```

从Filename绝对路径加载BEAM文件。

```
load_binary(Module, Filename, Binary) -> {module, Module} | {error, What}
```

将Binary的BEAM数据载入，Filename为其文件名，Module为模块名。

```
load_file(Module) -> {module, Module} | {error, What}
```

从代码目录中加载Module模块。

```
objfile_extension() -> ".beam"
```

获取目标文件的扩展名。

```
priv_dir(Name) -> string() | {error, bad_name}
```

获取Name应用程序的Priv目录。

```
purge(Module) -> true | false
```

清除Module模块老版本的代码。

```
rehash() -> ok
```

创建或者重新建立代码目录的缓存。

```
replace_path(Name, Dir) -> true | {error, What}
```

将代码目录列表中名为Name的目录替换为Dir。

```
root_dir() -> string()
```

获取Erlang/OTP的根目录。

```
set_path(Path) -> true | {error, What}
```

设置代码服务器的搜索路径。

```
soft_purge(Module) -> true | false
```

在没有进程使用的情况下清除Module模块老版本的代码。

```
stick_dir(Dir) -> ok | {error, What}
```

将Dir目录标记为“黏性”。

```
unstick_dir(Dir) -> ok | {error, What}
```

去除Dir目录的“黏性”标记。

```
where_is_file(Filename) -> Absname | non_existing
```

搜索代码目录，获取Filename文件名的完整路径。

```
which(Module) -> Which
```

获取Module模块目标代码文件的位置。

F.7 dets模块

基于磁盘的数据存储。

```
all() -> [Name]
```

获取当前节点上所有已打开DETS表的名称。

```
bchunk(Name, Continuation) -> {Continuation2, Data} | '$end_of_table' | {error, Reason}
```

返回Name表第Continuation块的数据。

```
close(Name) -> ok | {error, Reason}
```

关闭Name表。

441

```
delete(Name, Key) -> ok | {error, Reason}
```

删除Name表中所有键为Key的数据。

```
delete_all_objects(Name) -> ok | {error, Reason}
```

删除Name表中的所有数据。

```
delete_object(Name, Object) -> ok | {error, Reason}
```

删除Name表中的Object数据。

```
first(Name) -> Key | '$end_of_table'
```

获取Name表的第一个键。

```
foldl(Function, Acc0, Name) -> Acc1 | {error, Reason}
```

对Name表中的所有数据以foldl方式执行Function函数。

```
foldr(Function, Acc0, Name) -> Acc1 | {error, Reason}
```

对Name表中的所有数据以foldr方式执行Function函数。

```
from_ets(Name, EtsTab) -> ok | {error, Reason}
```

用EtsTab的数据来替换Name表中的数据。

```
info(Name) -> InfoList | undefined
```

获取Name表的（所有）信息。

```
info(Name, Item) -> Value | undefined
```

获取Name表关于Item的信息。

```
init_table(Name, InitFun [, Options]) -> ok | {error, Reason}
```

使用InitFun来初始化Name表。

```
insert(Name, Objects) -> ok | {error, Reason}
```

向Name表插入（一个或者多个）Object数据。

```
insert_new(Name, Objects) -> Bool
```

向Name表插入（一个或者多个）Object数据（只插入新数据）。

```
is_compatible_bchunk_format(Name, BchunkFormat) -> Bool
```

测试BchunkFormat数据块是否与Name表兼容。

```
is_dets_file(FileName) -> Bool | {error, Reason}
```

测试FileName文件是否为DETS数据文件。

```
lookup(Name, Key) -> [Object] | {error, Reason}
```

获取Name表中所有主键为Key的数据。

```
match(Continuation) -> {[Match], Continuation2} | '$end_of_table' | {error, Reason}
```

对DETS表的第Continuation块数据，进行比对，返回相匹配的变量。

```
match(Name, Pattern) -> [Match] | {error, Reason}
```

用Pattern比对Name表中的所有数据，返回相匹配的变量。

```
match(Name, Pattern, N) -> {[Match], Continuation} | '$end_of_table' | {error, Reason}
```

用Pattern比对Name表的第1块数据，返回相匹配的变量。

```
match_delete(Name, Pattern) -> N | {error, Reason}
```

442 用Pattern比对Name表中的数据，删除相匹配的数据。

```
match_object(Continuation) -> {[Object], Continuation2} | '$end_of_table' | {error,
   Reason}
```

对DETS表的第Continuation块数据进行比对，返回相匹配的记录。

```
match_object(Name, Pattern) -> [Object] | {error, Reason}
```

用Pattern比对Name表中的所有数据，返回相匹配的记录。

```
match_object(Name, Pattern, N) -> {[Object], Continuation} | '$end_of_table' | {error,
   Reason}
```

用Pattern比对Name表的第1块数据，返回相匹配的记录。

```
member(Name, Key) -> Bool | {error, Reason}
```

测试Name表中是否存在主键为Key的记录。

```
next(Name, Key1) -> Key2 | '$end_of_table'
```

获取Name表中Key1主键之后的下一个主键。

```
open_file(Filename) -> {ok, Reference} | {error, Reason}
```

打开Filename的DETS数据文件。

```
open_file(Name, Args) -> {ok, Name} | {error, Reason}
```

打开Name表的数据文件。

```
pid2name(Pid) -> {ok, Name} | undefined
```

获取PID进程管理的DETS表名。

```
repair_continuation(Continuation,MatchSpec) -> Continuation2
```

通过select/1或select/3修复第Continuation块数据。

```
safe_fixtable(Name, Fix)
```

修复Name表的导航信息。

```
select(Continuation) -> {Selection, Continuation2} | '$end_of_table' | {error, Reason}
```

对DETS表中的第Continuation块数据进行比对。

```
select(Name, MatchSpec) -> Selection | {error, Reason}
```

对DETS表中的所有数据进行比对。

```
select(Name,MatchSpec, N) -> {Selection,Continuation} | '$end_of_table' | {error,
   Reason}
```

对DETS表中的第1块数据进行比对。

```
select_delete(Name, MatchSpec) -> N | {error, Reason}
```

删除Name表中所有匹配MatchSpec条件的数据。

```
slot(Name, I) -> '$end_of_table' | [Object] | {error, Reason}
```

获取Name表中第I组的数据。

```
sync(Name) -> ok | {error, Reason}
```

同步Name表，确保所有的改动都写入了磁盘。

```
table(Name [, Options]) -> QueryHandle
```

获取Name表的QLC查询句柄。

```
to_ets(Name, EtsTab) -> EtsTab | {error, Reason}
```

将Name表中的所有数据插入到EtsTab表中。

```
traverse(Name, Fun) -> Return | {error, Reason}
```

对Name表中的所有数据遍历执行Fun函数。

443

```
update_counter(Name, Key, Increment) -> Result
```

更新Name表中的计数器。

F.8 dict模块

存储键值对数据的字典结构。

```
append(Key, Value, Dict1) -> Dict2
```

向字典Dict1增加键为Key值为Value的记录。

```
append_list(Key, ValList, Dict1) -> Dict2
```

对字典Dict1中键Key所对应的值列表增加ValList数据。

```
erase(Key, Dict1) -> Dict2
```

删除字典Dict1中键Key所对应的记录。

```
fetch(Key, Dict) -> Value
```

获取字典Dict中键Key所对应的值。

```
fetch_keys(Dict) -> Keys
```

获取字典Dict中所有的键。

```
filter(Pred, Dict1) -> Dict2
```

用Pred函数对字典Dict1中的所有数据进行过滤。

```
find(Key, Dict) -> {ok, Value} | error
```

查找字典Dict中键Key所对应的值。

```
fold(Fun, Acc0, Dict) -> Acc1
```

对字典Dict中的所有数据以fold方式执行Fun函数。

```
from_list(List) -> Dict
```

将List列表中的键值对转换为字典。

```
is_key(Key, Dict) -> bool()
```

检查键Key是否存在于字典Dict中。

```
map(Fun, Dict1) -> Dict2
```

对字典Dict1中的数据以map方式执行Fun函数。

```
merge(Fun, Dict1, Dict2) -> Dict3
```

对字典Dict1和字典Dict2用Fun函数进行合并。

```
new() -> dictionary()
```

新建字典。

```
store(Key, Value, Dict1) -> Dict2
```

将键为Key值为Value的记录存入字典Dict1。

```
to_list(Dict) -> List
```

将字典Dict中的数据转换为键值对数据的列表。

```
update(Key, Fun, Dict1) -> Dict2
```

用Fun函数来更新字典Dict1中键Key所对应的值。

```
update(Key, Fun, Initial, Dict1) -> Dict2
```

444

用Fun函数来更新字典Dict1中键Key所对应的值，初始值为Initial。

```
update_counter(Key, Increment, Dict1) -> Dict2
```

增加字典Dict1中键Key对应的计数器。

F.9　digraph模块

有向图（图论、拓扑学）。

```
add_edge(G, V1, V2) -> edge() | {error, Reason}
```

在图G的顶点V1和V2之间增加一个边。

```
add_vertex(G) -> vertex()
```

在图G中增加一个顶点。

```
del_edge(G, E) -> true
```

从图G中删除边E。

```
del_edges(G, Edges) -> true
```

从图G中删除Edges列表中的边。

```
del_path(G, V1, V2) -> true
```

从图G中删除顶点V1和V2之间的路径。

```
del_vertex(G, V) -> true
```

从图G中删除顶点V。

```
del_vertices(G, Vertices) -> true
```

从图G中删除Vertices列表中的顶点。

```
delete(G) -> true
```

删除图G。

```
edge(G, E) -> {E, V1, V2, Label} | false
```

获取图G中边E的信息。

```
edges(G) -> Edges
```

获取图G所有的边。

```
edges(G, V) -> Edges
```

获取图G中顶点V的所有边。

```
get_cycle(G, V) -> Vertices | false
```

在图G中的顶点V上寻找循环。

```
get_path(G, V1, V2) -> Vertices | false
```

在图G中的顶点V1和V2之间寻找一条路径。

```
get_short_cycle(G, V) -> Vertices | false
```

在图G中的顶点V上寻找最短的循环。

```
get_short_path(G, V1, V2) -> Vertices | false
```

在图G中的顶点V1和V2之间寻找一条最短路径。

```
in_degree(G, V) -> integer()
```

计算图G中顶点V的in degree（入边的数量）。

```
in_edges(G, V) -> Edges
```

获取图G中顶点V的所有的入边。

445

```
in_neighbours(G, V) -> Vertices
```

获取图G中所有与顶点V有入边相连的顶点。

```
info(G) -> InfoList
```

获取图G的信息。

```
new() -> digraph()
```

新建一个受保护的空的图，允许循环。

```
new(Type) -> digraph() | {error, Reason}
```

用Type类型新建一个空的图。

```
no_edges(G) -> integer() >= 0
```

获取图G中边的数量。

```
no_vertices(G) -> integer() >= 0
```

获取图G中顶点的数量。

```
out_degree(G, V) -> integer()
```

获取图G中顶点V的out degree（出边的数量）。

```
out_edges(G, V) -> Edges
```

获取图G中顶点V所有的出边。

```
out_neighbours(G, V) -> Vertices
```

获取图G中所有与顶点V有出边相连的顶点。

```
vertex(G, V) -> {V, Label} | false
```

获取图G中顶点V的标签。

```
vertices(G) -> Vertices
```

获取图G中所有的顶点。

F.10 digraph_utils模块

有向图上的算法。

```
components(Digraph) -> [Component]
```

获取图Digraph的component（部分）。

```
condensation(Digraph) -> CondensedDigraph
```

获取图Digraph的condensation（总结）。

```
cyclic_strong_components(Digraph) -> [StrongComponent]
```

获取图Digraph的cyclic strong component（循环的有强连接的部分）。

```
is_acyclic(Digraph) -> bool()
```

判断图Digraph是否是循环图。

```
loop_vertices(Digraph) -> Vertices
```

获取图Digraph中有循环关系的节点。

```
postorder(Digraph) -> Vertices
```

获取图Digraph顶点的后序排列。

```
preorder(Digraph) -> Vertices
```

446 获取图Digraph顶点的前序排列。

```
reachable(Vertices, Digraph) -> Vertices
```

获取图Digraph中Vertices顶点（列表）可以去到的顶点。

```
reachable_neighbours(Vertices, Digraph) -> Vertices
```

获取图Digraph中Vertices顶点（列表）可以去到的相邻顶点。

```
reaching(Vertices, Digraph) -> Vertices
```

获取图Digraph中所有可以到达Vertices顶点（列表）的顶点。

```
reaching_neighbours(Vertices, Digraph) -> Vertices
```

获取图Digraph中所有可以到达Vertices顶点（列表）的相邻顶点。

```
strong_components(Digraph) -> [StrongComponent]
```

获取图Digraph的 strong components（有强连接的部分）。

```
subgraph(Digraph, Vertices [, Options]) -> Subgraph | {error, Reason}
```

获取图Digraph中包含Vertices顶点（列表）的子图。

```
topsort(Digraph) -> Vertices | false
```

获取图Digraph中顶点的拓扑顺序。

F.11　disk_log模块

基于磁盘的日志机制。

```
accessible_logs() -> {[LocalLog], [DistributedLog]}
```

获取当前节点可以访问的磁盘日志（列表）。

```
balog(Log, Bytes) -> ok | {error, Reason}
```

将Bytes中的信息异步记入Log日志。

```
balog_terms(Log, BytesList) -> ok | {error, Reason}
```

将BytesList列表中的信息异步记入Log日志。

```
bchunk(Log, Continuation, N) -> {Continuation2, Binaries} | {Continuation2, Binaries,
   Badbytes} | eof | {error, Reason}
```

从之前写入的日志Log中读取一块连续的数据。

```
block(Log, QueueLogRecords) -> ok | {error, Reason}
```

阻塞Log日志。

```
blog(Log, Bytes) -> ok | {error, Reason}
```

将Bytes中的信息（同步）记入Log日志。

```
blog_terms(Log, BytesList) -> ok | {error, Reason}
```

将BytesList列表中的信息（同步）记入Log日志。

```
breopen(Log, File, BHead) -> ok | {error, Reason}
```

重新打开Log日志，另行保存旧日志。

```
btruncate(Log, BHead) -> ok | {error, Reason}
```

截断Log日志。

```
change_header(Log, Header) -> ok | {error, Reason}
```

Log日志的拥有进程可以通过这个函数更改日志头部或头函数。

```
change_notify(Log, Owner, Notify) -> ok | {error, Reason}
```

对Log日志的拥有进程Owner，更改其通知选项。

447

```
change_size(Log, Size) -> ok | {error, Reason}
```

将Log日志的尺寸设置为Size。

```
chunk_info(Continuation) -> InfoList | {error, Reason}
```

获取磁盘日志连续块Continuation的信息。

```
chunk_step(Log, Continuation, Step) -> {ok, Continuation2} | {error, Reason}
```

在一个回滚日志Log中前进或后退。

```
close(Log) -> ok | {error, Reason}
```

关闭日志Log。

```
format_error(Error) -> Chars
```

返回错误Error的英文描述。

```
inc_wrap_file(Log) -> ok | {error, Reason}
```

改为使用Log日志的下一个日志文件。

```
info(Log) -> InfoList | {error, no_such_log}
```

获取Log日志的信息。

```
lclose(Log, Node) -> ok | {error, Reason}
```

关闭Node节点上的Log日志。

```
open(ArgL) -> OpenRet | DistOpenRet
```

打开ArgL所描述的日志文件。

```
pid2name(Pid) -> {ok, Log} | undefined
```

获取Pid进程管理的日志名称。

```
sync(Log) -> ok | {error, Reason}
```

刷新Log日志，确保所有信息写入磁盘。

```
unblock(Log) -> ok | {error, Reason}
```

解除Log日志的阻塞。

F.12 epp模块

Erlang代码预处理器。

```
close(Epp) -> ok
```

关闭对与Epp相关文件的预处理过程。

```
open(FileName, IncludePath, PredefMacros) -> {ok,Epp} | {error, ErrorDescriptor}
```

打开FileName文件，准备进行预处理。

```
parse_erl_form(Epp) -> {ok, AbsForm} | {eof, Line} | {error, ErrorInfo}
```

从打开的Erlang源文件中获取下一个Erlang语句。

```
parse_file(FileName,IncludePath,PredefMacro) -> {ok,[Form]} | {error,OpenError}
```

对FileName文件进行解析和预处理。

448

F.13　erl_eval模块

Erlang原解释器。

```
add_binding(Name, Value, Bindings) -> BindingStruct
```

增加绑定。

```
binding(Name, BindingStruct) -> Binding
```

返回绑定。

```
bindings(BindingStruct) -> Bindings
```

返回绑定。

```
del_binding(Name, Bindings) -> BindingStruct
```

删除绑定。

```
expr(Expression, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) -> {value,
   Value, NewBindings}
```

演算表达式。

```
expr_list(ExpressionList, Bindings, LocalFunctionHandler, NonlocalFunctionHandler)
   -> {ValueList, NewBindings}
```

演算表达式（列表）。

```
exprs(Expressions, Bindings, LocalFunctionHandler, NonlocalFunctionHandler) ->
   {value, Value, NewBindings}
```

演算表达式。

```
new_bindings() -> BindingStruct
```

返回绑定结构。

F.14　erl_parse模块

Erlang解析器。

```
abstract(Data) -> AbsTerm
```

将Erlang语句转换为抽象语句。

```
format_error(ErrorDescriptor) -> Chars
```

格式化一个错误描述。

```
normalise(AbsTerm) -> Data
```

将抽象语句转换为Erlang语句。

```
parse_exprs(Tokens) -> {ok, Expr_list} | {error, ErrorInfo}
```

解析Erlang表达式。

```
parse_form(Tokens) -> {ok, AbsForm} | {error, ErrorInfo}
```

解析Erlang语句。

```
parse_term(Tokens) -> {ok, Term} | {error, ErrorInfo}
```

解析Erlang变量。

```
tokens(AbsTerm, MoreTokens) -> Tokens
```

449　从表达式生成一组标记（列表）。

F.15　erl_pp模块

Erlang输出美化模块。

attribute(Attribute, HookFunction) -> DeepCharList
输出一个属性。
expr(Expression, Indent, Precedence, HookFunction) ->-> DeepCharList
输出一个表达式。
exprs(Expressions, Indent, HookFunction) -> DeepCharList
输出几个表达式。
form(Form, HookFunction) -> DeepCharList
输出一个语句。
function(Function, HookFunction) -> DeepCharList
输出一个函数。
guard(Guard, HookFunction) -> DeepCharList
输出一个防护语句。

F.16 erl_scan模块

Erlang标记扫描器。
format_error(ErrorDescriptor) -> string()
格式化错误描述。
reserved_word(Atom) -> bool()
判断是不是保留字。
string(CharList) -> {ok, Tokens, EndLine} | Error
扫描字符串，返回标记。
tokens(Continuation, CharList, StartLine) ->Return
可重入的扫描器。

F.17 erl_tar模块

读写TAR打包文件的工具。
add(TarDescriptor, Filename, Options) -> RetValue
向打开的TAR包添加文件。
add(TarDescriptor, Filename, NameInArchive, Options) -> RetValue
向打开的TAR包添加文件。
close(TarDescriptor)
关闭之前打开的TAR包。
create(Name, FileList) ->RetValue
创建TAR包。
create(Name, FileList, OptionList)
带选项的创建TAR包。
extract(Name) -> RetValue
展开TAR包内的所有内容。
extract(Name, OptionList)
带参数的展开TAR包内的内容。
format_error(Reason) -> string()

将错误变量转换为可以阅读的信息。

```
open(Name, OpenModeList) -> RetValue
```

打开TAR包。

```
t(Name)
```

显示TAR包内所有文件的名称。

```
table(Name) -> RetValue
```

获取TAR包内所有文件的名称。

```
table(Name, Options)
```

获取TAR包内所有文件的名称和信息。

```
tt(Name)
```

显示TAR包内所有文件的名称和信息。

F.18　erlang模块

Erlang的BIF。

```
abs(Number) -> int() | float()
```

取Number的绝对值。

```
apply(Fun, Args) -> term() | empty()
```

用Args参数调用Fun函数。

```
apply(Module, Function, Args) -> term() | empty()
```

用Args参数调用Module模块的Fun函数。

```
atom_to_list(Atom) -> string()
```

获取Atom原子的文本表示形式。

```
binary_to_list(Binary) -> [char()]
```

将二进制数据Binary转换为列表。

```
binary_to_list(Binary, Start, Stop) -> [char()]
```

将二进制数据Binary从Start到Stop的部分转换为列表。

```
binary_to_term(Binary) -> term()
```

从二进制数据Binary中解码Erlang值。

```
check_process_code(Pid,Module) -> bool()
```

检查Pid进程的运行代码是否为旧版本。

```
concat_binary(ListOfBinaries)
```

合并二进制数据的列表（不建议使用）。

```
date() -> {Year, Month, Day}
```

451　获取当前日期。

```
delete_module(Module) -> true | undefined
```

将Module当前的代码标记为旧版。

```
disconnect_node(Node) -> bool() | ignored
```

强制断开与Node节点的连接。

```
element(N, Tuple) -> term()
```

获取Tuple元组的第N个元素。

```
erase() -> [{Key, Val}]
```

获取并清除进程字典的所有内容。

```
erase(Key) -> Val | undefined
```

获取并清除进程字典中键Key所对应的值。

```
erlang:append_element(Tuple1, Term) -> Tuple2
```

向元组Tuple1增加额外的值Term。

```
erlang:bump_reductions(Reductions) -> void()
```

增加缩减计数器（减计数器是BEAM的内建机制）。

```
erlang:cancel_timer(TimerRef) -> Time | false
```

取消TimerRef计时器。

```
erlang:demonitor(MonitorRef) -> true
```

停止MonitorRef监视。

```
erlang:demonitor(MonitorRef, OptionList) -> true
```

用OptionList列表中的参数停止MonitorRef监视。

```
erlang:display(Term) -> true
```

在标准输出上输出Term变量。

```
erlang:error(Reason)
```

当前进程出错，原因为Reason。

```
erlang:error(Reason, Args)
```

当前进程出错，原因为Reason，参数为Args。

```
erlang:fault(Reason)
```

当前进程失效，原因为Reason。

```
erlang:fault(Reason, Args)
```

当前进程失效，原因为Reason，参数为Args。

```
erlang:fun_info(Fun) -> [{Item, Info}]
```

获取Fun函数的信息。

```
erlang:fun_info(Fun, Item) -> {Item, Info}
```

获取Fun函数Item项目的信息。

```
erlang:fun_to_list(Fun) -> string()
```

获取Fun函数的文字表现形式。

```
erlang:function_exported(Module, Function, Arity) -> bool()
```

检查Module模块的Function函数是否公开并被加载。

```
erlang:get_cookie() -> Cookie | nocookie
```

获取当前节点的安全cookie。

452

```
erlang:get_stacktrace() -> [{Module, Function, Arity | Args}]
```

获取上一个异常的栈跟踪。

```
erlang:hash(Term, Range) -> Hash
```

散列函数（不建议使用）。

```
erlang:hibernate(Module, Function, Args)
```

休眠当前进程，直到再有消息发送给它。

```
erlang:info(Type) -> Res
```

获取系统信息（不建议使用）。

```
erlang:integer_to_list(Integer, Base) -> string()
```

将整数Interger转换为Base进制的文字表示形式。

```
erlang:is_builtin(Module, Function, Arity) -> bool()
```

检查Module模块的Function函数是否是用C实现的BIF。

```
erlang:list_to_integer(String, Base) -> int()
```

将Base进制以文字形式表示的整数转换为整型数。

```
erlang:loaded() -> [Module]
```

获取当前已加载模块的列表。

```
erlang:localtime() -> {Date, Time}
```

获取现在的当地时间。

```
erlang:localtime_to_universaltime({Date1, Time1}) -> {Date2, Time2}
```

将当地时间Date1日Time1时转换为UTC时间。

```
erlang:localtime_to_universaltime({Date1, Time1}, IsDst) -> {Date2, Time2}
```

将当地时间Date1日Time1时转换为UTC时间。

```
erlang:make_tuple(Arity, InitialValue) -> tuple()
```

创建元素数量为Arity的元组。

```
erlang:md5(Data) -> Digest
```

计算数据Data的MD5摘要。

```
erlang:md5_final(Context) -> Digest
```

结束MD5计算过程Context，获取最终计算的MD5摘要。

```
erlang:md5_init() -> Context
```

新建MD5计算过程。

```
erlang:md5_update(Context, Data) -> NewContext
```

更新MD5计算过程，返回一个新的上下文。

```
erlang:memory() -> [{Type, Size}]
```

获取动态内存分配情况。

```
erlang:memory(Type | [Type]) -> Size | [{Type, Size}]
```

获取动态内存分配Type项（或Type列表）的情况。

```
erlang:monitor(Type, Item) -> MonitorRef
```

开始一个监视过程。

```
erlang:monitor_node(Node, Flag, Options) -> true
```

453 监视Node节点的状态。

```
erlang:phash(Term, Range) -> Hash
```

可移植的散列函数（上述散列函数的替代版本）。

```
erlang:phash2(Term [, Range]) -> Hash
```

可移植的散列函数。

```
erlang:port_call(Port, Operation, Data) -> term()
```

对Port端口进行同步调用。

```
erlang:port_info(Port) -> [{Item, Info}] | undefined
```

获取Port端口的信息。

```
erlang:port_info(Port, Item) -> {Item, Info} | undefined | []
```

获取Port端口Item项的信息。

```
erlang:port_to_list(Port) -> string()
```

获取Port端口的文字表达形式。

```
erlang:ports() -> [port()]
```

获取当前所有已经打开的端口。

```
erlang:process_display(Pid, Type) -> void()
```

在标准错误输出上显示Pid进程的信息。

```
erlang:raise(Class, Reason, Stacktrace)
```

终止当前进程的执行，抛出Class类型的异常。

```
erlang:read_timer(TimerRef) -> int() | false
```

获取计时器TimerRef的剩余时间。

```
erlang:ref_to_list(Ref) -> string()
```

获取Ref的文字表达形式。

```
erlang:resume_process(Pid) -> true
```

恢复暂停的Pid进程。

```
erlang:send(Dest, Msg) -> Msg
```

向Dest进程发送Msg消息。

```
erlang:send(Dest, Msg, [Option]) -> Res
```

以Option选项向Dest进程发送Msg消息。

```
erlang:send_after(Time, Dest, Msg) -> TimerRef
```

新建一个计时器。

```
erlang:send_nosuspend(Dest, Msg) -> bool()
```

以非阻塞的方式发送消息。

```
erlang:send_nosuspend(Dest, Msg, Options) -> bool()
```

以非阻塞的方式发送消息。

```
erlang:set_cookie(Node, Cookie) -> true
```

设置当前节点的安全cookie为Cookie。

```
erlang:spawn_monitor(Fun) -> {pid(),reference()}
```

新建一个进程并对其进行监视。

```
erlang:spawn_monitor(Module, Function, Args) -> {pid(),reference()}
```

新建一个进程并对其进行监视。

454

```
erlang:start_timer(Time, Dest, Msg) -> TimerRef
```

新建一个计时器。

```
erlang:suspend_process(Pid) -> true
```

暂停Pid进程。

```
erlang:system_flag(Flag, Value) -> OldValue
```

将系统标志Flag的值设为Value。

```
erlang:system_info(Type) -> Res
```

获取系统信息的Type项。

```
erlang:system_monitor() -> MonSettings
```

获取系统监视设置。

```
erlang:system_monitor(undefined | {MonitorPid, Options}) -> MonSettings
```

设置或清除系统监视属性。

```
erlang:system_monitor(MonitorPid, [Option]) -> MonSettings
```

设置系统监视属性。

```
erlang:trace(PidSpec, How, FlagList) -> int()
```

对PidSpec指定的进程设置跟踪属性。

```
erlang:trace_delivered(Tracee) -> Ref
```

设置当前进程以便能收到跟踪投递的消息。

```
erlang:trace_info(PidOrFunc, Item) -> Res
```

获取Pid进程（或Func函数）的跟踪信息。

```
erlang:trace_pattern(MFA, MatchSpec) -> int()
```

设置全局跟踪匹配模式。

```
erlang:trace_pattern(MFA, MatchSpec, FlagList) -> int()
```

设置全局跟踪匹配模式。

```
erlang:universaltime() -> {Date, Time}
```

获取当前的UTC时间。

```
erlang:universaltime_to_localtime({Date1, Time1}) -> {Date2, Time2}
```

将UTC时间的Date1日Time1时转换为当地时间。

```
erlang:yield() -> true
```

让出当前进程的执行。

```
exit(Reason)
```

停止当前进程，原因为Reason。

```
exit(Pid, Reason) -> true
```

向Pid进程发出停止信号，原因为Reason。

```
float(Number) -> float()
```

将数值Number转换为浮点数。

```
float_to_list(Float) -> string()
```

获取浮点数Float的文字表达形式。

```
garbage_collect() -> true
```

强制调用进程立即执行垃圾回收。

```
garbage_collect(Pid) -> bool()
```

455　强制Pid进程立即执行垃圾回收。

```
get() -> [{Key, Val}]
```

获取进程字典。

```
get(Key) -> Val | undefined
```

从进程字典中获取键Key对应的值。

```
get_keys(Val) -> [Key]
```

从进程字典中获取对应值Val的所有键。

```
group_leader() -> GroupLeader
```

获取当前进程的GroupLeader。

```
group_leader(GroupLeader, Pid) -> true
```

设置Pid进程的GroupLeader。

```
halt()
```

停止Erlang的运行时系统，正常退出。

```
halt(Status)
```

停止Erlang的运行时系统，退出状态为Status。

```
hd(List) -> term()
```

获取列表List的头。

```
integer_to_list(Integer) -> string()
```

获取整数Integer的字符表示形式。

```
iolist_size(Item) -> int()
```

获取Item这个iolist的大小。

```
iolist_to_binary(IoListOrBinary) -> binary()
```

将iolist转换为二进制数据。

```
is_alive() -> bool()
```

检查当前节点是否存活（可以成为分布式系统的一部分）。

```
is_atom(Term) -> bool()
```

检查变量Term是否为原子类型。

```
is_binary(Term) -> bool()
```

检查变量Term是否为二进制类型。

```
is_boolean(Term) -> bool()
```

检查变量Term是否为布尔类型。

```
is_float(Term) -> bool()
```

检查变量Term是否为浮点类型。

```
is_function(Term) -> bool()
```

检查变量Term是否为函数类型。

```
is_function(Term, Arity) -> bool()
```

检查变量Term是否为带Arity个参数的函数。

```
is_integer(Term) -> bool()
```

检查变量Term是否为整数类型。

456

```
is_list(Term) -> bool()
```

检查变量Term是否为列表类型。

```
is_number(Term) -> bool()
```

检查变量Term是否为数值。

```
is_pid(Term) -> bool()
```

检查变量Term是否为PID类型。

```
is_port(Term) -> bool()
```

检查变量Term是否为Port类型。

```
is_process_alive(Pid) -> bool()
```

检查进程Pid是否存活（尚未退出）。

```
is_record(Term, RecordTag) -> bool()
```

检查变量Term是否为RecordTag标记的记录。

```
is_record(Term, RecordTag, Size) -> bool()
```

检查变量Term是否为RecordTag标记的记录。

```
is_reference(Term) -> bool()
```

检查变量Term是否为引用（Reference）类型。

```
is_tuple(Term) -> bool()
```

检查变量Term是否为元组类型。

```
length(List) -> int()
```

获取List列表的长度。

```
link(Pid) -> true
```

在当前进程与Pid进程（或端口）之间建立链接。

```
list_to_atom(String) -> atom()
```

将字符串String转换为原子。

```
list_to_binary(IoList) -> binary()
```

将字符串String转换为二进制数据。

```
list_to_existing_atom(String) -> atom()
```

将字符串String转换为已经存在的原子。

```
list_to_float(String) -> float()
```

将字符串String转换为浮点数。

```
list_to_integer(String) -> int()
```

将字符串String转换为整型数。

```
list_to_pid(String) -> pid()
```

将字符串String转换为PID。

```
list_to_tuple(List) -> tuple()
```

将字符串String转换为元组。

```
load_module(Module, Binary) -> {module, Module} | {error, Reason}
```

从Binary中加载Module模块。

```
make_ref() -> ref()
```

457 获取一个引用（通常用作近乎唯一的标识）。

```
module_loaded(Module) -> bool()
```

检查Module模块是否已经加载成功。

```
monitor_node(Node, Flag) -> true
```

监控Node节点的状态。

```
node() -> Node
```

获取当前节点的名字。

```
node(Arg) -> Node
```

获取Arg（可以是Pid、Port或Ref）所在的节点。

```
nodes() -> Nodes
```

获取当前系统所有的节点。

```
nodes(Arg | [Arg]) -> Nodes
```

获取当前系统所有Arg类型的节点。

```
now() -> {MegaSecs, Secs, MicroSecs}
```

获取从GMT0点以来经过的时间。

```
open_port(PortName, PortSettings) -> port()
```

打开一个端口。

```
pid_to_list(Pid) -> string()
```

将Pid转换为字符串。

```
port_close(Port) -> true
```

关闭一个Port。

```
port_command(Port, Data) -> true
```

向Port发送一个Data数据。

```
port_connect(Port, Pid) -> true
```

设置Port的所有者为Pid进程。

```
port_control(Port, Operation, Data) -> Res
```

在Port上同步的执行Operation操作。

```
pre_loaded() -> [Module]
```

获取预加载的模块列表。

```
process_flag(Flag, Value) -> OldValue
```

设置当前进程的进程标志。

```
process_flag(Pid, Flag, Value) -> OldValue
```

设置Pid进程的进程标志。

```
process_info(Pid) -> [{Item, Info}] | undefined
```

获取Pid进程的信息。

```
process_info(Pid, Item) -> {Item, Info} | undefined | []
```

获取Pid进程的Item项信息。

```
processes() -> [pid()]
```

获取所有进程的列表。

```
purge_module(Module) -> void()
```

去掉Module模块的旧代码。

458

```
put(Key, Val) -> OldVal | undefined
```

向进程字典增加键为Key值为Val的记录。

```
register(RegName, Pid | Port) -> true
```

用RegName名字注册Pid进程（或Port端口）。

```
registered() -> [RegName]
```

获取所有已注册的名字。

```
round(Number) -> int()
```

对Number进行四舍五入。

```
self() -> pid()
```

获取当前进程的Pid。

```
setelement(Index, Tuple1, Value) -> Tuple2
```

设置元组Tuple1第Index个元素的值为Value。

```
size(Item) -> int()
```

获取Item（元组或二进制）的大小。

```
spawn(Fun) -> pid()
```

新建进程，其入口点为Fun。

```
spawn(Node, Fun) -> pid()
```

在Node节点上新建进程。

```
spawn(Module, Function, Args) -> pid()
```

新建进程，其入口点为Module:Fun。

```
spawn(Node, Module, Function, ArgumentList) -> pid()
```

在Node节点上新建进程，其入口点为Module:Fun。

```
spawn_link(Fun) -> pid()
```

新建进程，并建立链接。

```
spawn_link(Node, Fun) ->
```

在Node节点上新建进程，并建立链接。

```
spawn_link(Module, Function, Args) -> pid()
```

新建进程，并建立链接。

```
spawn_link(Node, Module, Function, Args) -> pid()
```

在Node节点上新建进程，并建立链接。

```
spawn_opt(Fun, [Option]) -> pid() | {pid(),reference()}
```

带参数新建进程。

```
spawn_opt(Node, Fun, [Option]) -> pid()
```

在Node节点上带参数新建进程。

```
spawn_opt(Module, Function, Args, [Option]) -> pid() | {pid(),reference()}
```

带参数新建进程。

```
spawn_opt(Node, Module, Function, Args, [Option]) -> pid()
```

459 在Node节点上带参数新建进程。

```
split_binary(Bin, Pos) -> {Bin1, Bin2}
```

在Pos处将Bin截为两段。

```
statistics(Type) -> Res
```

获取系统统计信息。

```
term_to_binary(Term) -> ext_binary()
```

将Term变量编码为二进制表示。

```
term_to_binary(Term, [Option]) -> ext_binary()
```

将Term变量编码为二进制表示。

```
throw(Any)
```

抛出异常。

```
time() -> {Hour, Minute, Second}
```

获取当前时间。

```
tl(List1) -> List2
```

获取列表List1的尾部。

```
trunc(Number) -> int()
```

对数值Number进行截断。

```
tuple_to_list(Tuple) -> [term()]
```

将Tuple元组转换为列表。

```
unlink(Id) -> true
```

去掉当前进程与其他进程（或端口）之间的链接。

```
unregister(RegName) -> true
```

注销注册的RegName名称。

```
whereis(RegName) -> pid() | port() | undefined
```

获取RegName所对应的PID或者端口。

F.19 error_handler模块

默认的错误处理程序。

```
undefined_function(Module, Function, Args) -> term()
```

调用一个未定义的函数时，触发此函数。

```
undefined_lambda(Module, Fun, Args) -> term()
```

调用一个未定义的lambda函数时，触发此函数。

F.20 error_logger模块

Erlang错误日志模块。

```
add_report_handler(Handler, Args) -> Result
```

给错误日志增加一个事件处理程序。

```
delete_report_handler(Handler) -> Result
```

460

从错误日志中删除一个事件处理程序。

```
error_report(Report) -> ok
```

给错误日志发送一个标准的错误报告。

```
error_report(Type, Report) -> ok
```

给错误日志发送一个用户定义的错误报告。

```
format(Format, Data) -> ok
```

给错误日志发送一个标准的错误事件。

```
info_msg(Format, Data) -> ok
```

给错误日志发送一个标准的信息事件。

```
info_report(Report) -> ok
```

给错误日志发送一个标准的信息报告。

```
info_report(Type, Report) -> ok
```

给错误日志发送一个用户定义的信息报告。

```
logfile(Request) -> ok | Filename | {error, What}
```

允许或者禁止错误输出到文件。

```
tty(Flag) -> ok
```

允许或者禁止错误输出到控制台。

```
warning_map() -> Tag
```

获取当前的警告事件映射。

```
warning_msg(Format, Data) -> ok
```

给错误日志发送一个标准的警告事件。

```
warning_report(Report) -> ok
```

给错误日志发送一个标准的警告报告。

```
warning_report(Type, Report) -> ok
```

给错误日志发送一个用户自定义的警告报告。

F.21 ets模块

内建的变量存储机制。

```
all() -> [Tab]
```
获取所有的ETS表。
```
delete(Tab) -> true
```
删除Tab表。
```
delete(Tab, Key) -> true
```
删除Tab表中键为Key的所有记录。
```
delete_all_objects(Tab) -> true
```
删除Tab表中的所有记录。
```
delete_object(Tab,Object) -> true
```
删除Tab表中的特定记录。
```
file2tab(Filename) -> {ok,Tab} | {error,Reason}
```
461 从Filename文件中读取表的数据。
```
first(Tab) -> Key | '$end_of_table'
```
获取Tab表中的第一个键。
```
fixtable(Tab, true|false) -> true | false
```
修复Tab表以便进行遍历（已经废弃）。
```
foldl(Function, Acc0, Tab) -> Acc1
```
以foldl方式对Tab表中的所有数据执行Function函数。
```
foldr(Function, Acc0, Tab) -> Acc1
```
以foldr方式对Tab表中的所有数据执行Function函数。
```
from_dets(Tab, DetsTab) -> Tab
```
从DETS表中读取表的数据。
```
fun2ms(LiteralFun) -> MatchSpec
```
伪函数，将fun语法转换为匹配模式。
```
i() -> void()
```
将所有ETS表的信息输出到控制台上。
```
i(Tab) -> void()
```
在控制台上浏览Tab表。
```
info(Tab) -> [{Item, Value}] | undefined
```
获取Tab表的信息。
```
info(Tab, Item) -> Value | undefined
```
获取Tab表Item项的信息。
```
init_table(Name, InitFun) -> true
```
用InitFun函数初始化Name表。
```
insert(Tab, ObjectOrObjects) -> true
```
向Tab表插入一个Object数据。
```
insert_new(Tab, ObjectOrObjects) -> bool()
```
向Tab表插入一个Object数据（只插入新数据）。
```
is_compiled_ms(Term) -> bool()
```
检查一个Erlang值是否为ets: match-spec-compile的结果。
```
last(Tab) -> Key | '$end_of_table'
```
获取Tab表（表的类型为ordered_set）的最后一个键Key。

```
lookup(Tab, Key) -> [Object]
```
在Tab表中查找键为Key的记录。
```
lookup_element(Tab, Key, Pos) -> Elem
```
在Tab表中查找键为Key的记录，返回其第Pos个元素。
```
match(Continuation) -> {[Match],Continuation} | '$end_of_table'
```
对ETS表的第Continuation块数据，进行比对，返回相匹配的变量。
```
match(Tab, Pattern) -> [Match]
```
用Pattern比对Tab表中的所有数据，返回相匹配的变量。
```
match(Tab, Pattern, Limit) -> {[Match],Continuation} | '$end_of_table'
```
用Pattern比对Tab表的第1块数据，返回相匹配的变量。 462
```
match_delete(Tab, Pattern) -> true
```
用Pattern比对Tab表中的所有数据，删除相匹配的记录。
```
match_object(Continuation) -> {[Match],Continuation} | '$end_of_table'
```
对ETS表的第Continuation块数据进行比对，返回相匹配的记录。
```
match_object(Tab, Pattern) -> [Object]
```
用Pattern比对Tab表中的所有数据，返回相匹配的记录。
```
match_object(Tab, Pattern, Limit) -> {[Match],Continuation} | '$end_of_table'
```
用Pattern比对Tab表的第1块数据，返回相匹配的数据。
```
match_spec_compile(MatchSpec) -> CompiledMatchSpec
```
将MatchSpec匹配编译为内在的表示形式。
```
match_spec_run(List,CompiledMatchSpec) -> list()
```
在列表List上执行CompiledMatchSpec的匹配操作。
```
member(Tab, Key) -> true | false
```
测试Tab表中是否存在主键为Key的记录。
```
new(Name, Options) -> tid()
```
新建一个名为Name的表。
```
next(Tab, Key1) -> Key2 | '$end_of_table'
```
获取Tab表中Key1主键之后的主键。
```
prev(Tab, Key1) -> Key2 | '$end_of_table'
```
获取Tab表（表类型为ordered_set）中Key1主键之前的主键。
```
rename(Tab, Name) -> Name
```
将Tab表改名为Name。
```
repair_continuation(Continuation,MatchSpec) -> Continuation
```
通过select/1或select/3修复第Continuation块数据。
```
safe_fixtable(Tab, true|false) -> true
```
修复Name表的导航信息。
```
select(Continuation) -> {[Match],Continuation} | '$end_of_table'
```
对DETS表中的第Continuation块数据进行比对。
```
select(Tab, MatchSpec) -> [Match]
```
对DETS表中的所有数据进行比对。
```
select(Tab, MatchSpec, Limit) -> {[Match],Continuation} | '$end_of_table'
```
对DETS表中的第1块数据进行比对。

select_count(Tab, MatchSpec) -> NumMatched
对DETS表中的所有数据进行比对，返回匹配数据的条数。
select_delete(Tab, MatchSpec) -> NumDeleted
删除Name表中所有匹配MatchSpec条件的数据。
slot(Tab, I) -> [Object] | '$end_of_table'
463 获取Name表中第I组的数据。
tab2file(Tab, Filename) -> ok | {error,Reason}
将Tab表转存到Filename文件中。
tab2list(Tab) -> [Object]
将Tab表中的所有数据转换为列表。
table(Tab [, Options]) -> QueryHandle
获取Name表的QLC查询句柄。
test_ms(Tuple, MatchSpec) -> {ok, Result} | {error, Errors}
在Tuple元组上测试MatchSpec匹配。
to_dets(Tab, DetsTab) -> Tab
把Tab表的数据写入DETS表DetsTab中去。
update_counter(Tab, Key, Incr) -> Result
更新Tab表中的计数器。

F.22 file模块

文件接口模块。
change_group(Filename, Gid) -> ok | {error, Reason}
更高Filname文件所属的组。
change_owner(Filename, Uid) -> ok | {error, Reason}
更改Filname的所有者。
change_owner(Filename, Uid, Gid) -> ok | {error, Reason}
更改Filname的所有者。
change_time(Filename, Mtime) -> ok | {error, Reason}
更改Filname的文件时间。
change_time(Filename, Mtime, Atime) -> ok | {error, Reason}
更改Filname的文件时间。
close(IoDevice) -> ok | {error, Reason}
关闭文件IoDevice。
consult(Filename) -> {ok, Terms} | {error, Reason}
从文件Filename中读取Erlang值。
copy(Source, Destination, ByteCount) -> {ok, BytesCopied} | {error, Reason}
复制文件内容。
del_dir(Dir) -> ok | {error, Reason}
删除目录Dir。
delete(Filename) -> ok | {error, Reason}
删除文件Filename。

```
eval(Filename) -> ok | {error, Reason}
```
执行文件Filename中的Erlang表达式。
```
eval(Filename, Bindings) -> ok | {error, Reason}
```
执行文件Filename中的Erlang表达式。 464
```
file_info(Filename) -> {ok, FileInfo} | {error, Reason}
```
获取Filename文件的信息（不建议使用）。
```
format_error(Reason) -> Chars
```
获取错误Reason的文字描述。
```
get_cwd() -> {ok, Dir} | {error, Reason}
```
获取当前目录。
```
get_cwd(Drive) -> {ok, Dir} | {error, Reason}
```
获取驱动器Drive的当前目录。
```
list_dir(Dir) -> {ok, Filenames} | {error, Reason}
```
获取Dir目录的文件列表。
```
make_dir(Dir) -> ok | {error, Reason}
```
新建Dir目录。
```
make_link(Existing, New) -> ok | {error, Reason}
```
创建到文件Existing的硬连接。
```
make_symlink(Name1, Name2) -> ok | {error, Reason}
```
创建到目录Name1的符号连接。
```
open(Filename, Modes) -> {ok, IoDevice} | {error, Reason}
```
打开Filename文件。
```
path_consult(Path, Filename) -> {ok, Terms, FullName} | {error, Reason}
```
从文件Filename中读取Erlang值。
```
path_eval(Path, Filename) -> {ok, FullName} | {error, Reason}
```
执行文件Filename中的Erlang表达式。
```
path_open(Path, Filename, Modes) -> {ok, IoDevice, FullName} | {error, Reason}
```
打开Filename文件。
```
path_script(Path, Filename) -> {ok, Value, FullName} | {error, Reason}
```
执行文件Filename中的Erlang表达式。
```
path_script(Path, Filename, Bindings) -> {ok, Value, FullName} | {error, Reason}
```
执行文件Filename中的Erlang表达式。
```
pid2name(Pid) -> string() | undefined
```
获取Pid进程控制的文件名称。
```
position(IoDevice, Location) -> {ok, NewPosition} | {error, Reason}
```
设置IoDevice文件中的当前位置。
```
pread(IoDevice, LocNums) -> {ok, DataL} | {error, Reason}
```
从IoDevice文件的当前位置读取LocNums字节。
```
pread(IoDevice, Location, Number) -> {ok, Data} | {error, Reason}
```
从IoDevice文件的Location位置读取Number字节。
```
pwrite(IoDevice, LocBytes) -> ok | {error, {N, Reason}}
```
从IoDevice文件的当前位置写入LocBytes。

pwrite(IoDevice, Location, Bytes) -> ok | {error, Reason}

465 从IoDevice文件的Location位置写入LocBytes。

read(IoDevice, Number) -> {ok, Data} | eof | {error, Reason}

从IoDevice文件读取Number字节。

read_file(Filename) -> {ok, Binary} | {error, Reason}

读取Filename文件。

read_file_info(Filename) -> {ok, FileInfo} | {error, Reason}

获取Filename文件的信息。

read_link(Name) -> {ok, Filename} | {error, Reason}

获取Name连接到的文件名。

read_link_info(Name) -> {ok, FileInfo} | {error, Reason}

获取Name连接的信息。

rename(Source, Destination) -> ok | {error, Reason}

将Source文件改名为Destination文件。

script(Filename) -> {ok, Value} | {error, Reason}

执行文件Filename中的Erlang表达式。

script(Filename, Bindings) -> {ok, Value} | {error, Reason}

执行文件Filename中的Erlang表达式。

set_cwd(Dir) -> ok | {error,Reason}

设置当前目录为Dir。

sync(IoDevice) -> ok | {error, Reason}

同步IoDevice文件，确保所有的改动都已写入磁盘。

truncate(IoDevice) -> ok | {error, Reason}

截断IoDevice文件。

write(IoDevice, Bytes) -> ok | {error, Reason}

向IoDevice文件写入Bytes数据。

write_file(Filename, Binary) -> ok | {error, Reason}

向Filename文件写入Binary数据。

write_file(Filename, Binary, Modes) -> ok | {error, Reason}

向Filename文件写入Binary数据。

write_file_info(Filename, FileInfo) -> ok | {error, Reason}

更改Filename文件的文件信息。

F.23　file_sorter模块

文件排序器。

check(FileNames, Options) -> Reply

检查FileNames列表中的文件是否有序。

keycheck(KeyPos, FileNames, Options) -> Reply

检查FileNames列表中的文件是否按Key进行了排序。

keymerge(KeyPos, FileNames, Output, Options) -> Reply

466 用Key对FileNames列表中的文件进行合并。

```
keysort(KeyPos, Input, Output, Options) -> Reply
```
使用Key来对文件进行排序。
```
merge(FileNames, Output, Options) -> Reply
```
合并FileNames中的值。
```
sort(Input, Output, Options) -> Reply
```
对文件进行排序。

F.24 filelib模块

操作文件的常用工具函数，诸如文件名通配符等。
```
ensure_dir(Name) -> ok | {error, Reason}
```
确定Name文件（或目录）的所有父目录都已经存在。
```
file_size(Filename) -> integer()
```
获取Filename文件的大小。
```
fold_files(Dir, RegExp, Recursive, Fun, AccIn) -> AccOut
```
对Dir目录下的文件按RegExp表达式进行匹配，对符合要求的文件执行Fun函数。
```
is_dir(Name) -> true | false
```
测试Name是否为目录。
```
is_file(Name) -> true | false
```
测试Name是否为文件。
```
is_regular(Name) -> true | false
```
测试Name是否为一个标准文件。
```
last_modified(Name) -> {{Year,Month,Day},{Hour,Min,Sec}}
```
获取Name文件的最后更新时间。
```
wildcard(Wildcard) -> list()
```
使用Unix风格的通配符匹配文件。
```
wildcard(Wildcard, Cwd) -> list()
```
在Cwd目录下，使用Unix风格的通配符匹配文件。

F.25 filename模块

文件名操作函数。
```
absname(Filename) -> string()
```
获取现对于当前目录下的Filename所对应的路径。
```
absname(Filename, Dir) -> string()
```
获取对于Dir目录下的Filename所对应的路径。
```
absname_join(Dir, Filename) -> string()
```
组合目录Dir下的文件名Filename。
```
basename(Filename) -> string()
```
获取Filename路径的最后一个部分。
```
basename(Filename, Ext) -> string()
```
获取Filename路径的最后一个部分，去掉扩展名。
```
dirname(Filename) -> string()
```

获取Filename路径的目录部分。

```
extension(Filename) -> string()
```

获取Filename路径的扩展名部分。

```
find_src(Beam, Rules) -> {SourceFile, Options}
```

获取Beam文件的源文件名和编译选项。

```
flatten(Filename) -> string()
```

将Filename路径转换为一个扁平字符串。

```
join(Components) -> string()
```

将Components列表中的元素组合成一个路径。

```
join(Name1, Name2) -> string()
```

将Name1和Name2组合成一个路径。

```
nativename(Path) -> string()
```

获取路径Path在当前操作系统下的目录表示格式。

```
pathtype(Path) -> absolute | relative | volumerelative
```

获取路径Path的类型。

```
rootname(Filename, Ext) -> string()
```

去掉Filename的扩展名。

```
split(Filename) -> Components
```

将Filename按照路径进行拆分。

F.26　gb_sets模块

通用平衡树。

```
add_element(Element, Set1) -> Set2
```

将Element元素增加到树Set1中。

```
balance(Set1) -> Set2
```

重新平衡树Set1。

```
del_element(Element, Set1) -> Set2
```

从树Set1中删除Element元素。

```
delete(Element, Set1) -> Set2
```

从树Set1中删除Element元素。

```
filter(Pred, Set1) -> Set2
```

用Pred函数过滤树Set1。

```
fold(Function, Acc0, Set) -> Acc1
```

对树Set1的所有元素以fold方式执行Function函数。

```
from_list(List) -> Set
```

468

将列表List转换为树。

```
from_ordset(List) -> Set
```

将有序列表List转换为树。

```
insert(Element, Set1) -> Set2
```

将Element元素增加到树Set1中。

```
intersection(SetList) -> Set
```

获取SetList列表中树的交集。

```
intersection(Set1, Set2) -> Set3
```

获取树Set1和Set2的交集。

```
is_element(Element, Set) -> bool()
```

判断Element元素是否在树Set当中。

```
is_empty(Set) -> bool()
```

判断树Set是否为空。

```
is_set(Set) -> bool()
```

判断树Set是否为平衡树。

```
is_subset(Set1, Set2) -> bool()
```

判断树Set1是否为树Set2的子集。

```
iterator(Set) -> Iter
```

获得树Set的遍历器。

```
largest(Set) -> term()
```

获得树Set最大的元素。

```
new() -> Set
```

创建一个新的空的树。

```
next(Iter1) -> {Element, Iter2 | none}
```

用遍历器访问树的下一个元素。

```
singleton(Element) -> gb_set()
```

生成只有一个元素Element的树。

```
size(Set) -> int()
```

获取树Set的元素数量。

```
smallest(Set) -> term()
```

获取树Set中最小的元素。

```
subtract(Set1, Set2) -> Set3
```

对树Set1和Set2进行逻辑减。

```
take_largest(Set1) -> {Element, Set2}
```

从树Set1中抽取最大的元素。

```
take_smallest(Set1) -> {Element, Set2}
```

从树Set1中抽取最小的元素。

```
to_list(Set) -> List
```

将树Set转换为列表。

```
union(SetList) -> Set
```

对SetList列表中的树进行逻辑加。

469

```
union(Set1, Set2) -> Set3
```

对树Set1和Set2进行逻辑加。

F.27 gb_trees模块

通用平衡树。

```
balance(Tree1) -> Tree2
```

对树Tree1进行平衡。

```
delete(Key, Tree1) -> Tree2
```

从树Tree1中删除一个节点。

```
delete_any(Key, Tree1) -> Tree2
```

从树Tree1中删除一个节点（允许该节点不存在）。

```
empty() -> Tree
```

获得一个空的树。

```
enter(Key, Val, Tree1) -> Tree2
```

向树Tree1中插入（或更新）一个节点。

```
from_orddict(List) -> Tree
```

用orddict来构造一个树。

```
get(Key, Tree) -> Val
```

从树Tree中查找Key所对应的值。

```
insert(Key, Val, Tree1) -> Tree2
```

向树Tree1插入一个新的节点。

```
is_defined(Key, Tree) -> bool()
```

测试树Tree中是否已经包含Key节点。

```
is_empty(Tree) -> bool()
```

测试树Tree是否为空。

```
iterator(Tree) -> Iter
```

获取树Tree的遍历器。

```
keys(Tree) -> [Key]
```

获取树Tree中所有的Key。

```
largest(Tree) -> {Key, Val}
```

获取树Tree最大的节点。

```
lookup(Key, Tree) -> {value, Val} | none
```

从树Tree中查找Key所对应的值。

```
next(Iter1) -> {Key, Val, Iter2
```

使用遍历器获取树Tree的下一个节点。

```
size(Tree) -> int()
```

获取树Tree的节点数量。

```
smallest(Tree) -> {Key, Val}
```

470

获取树Tree最小的节点。

```
take_largest(Tree1) -> {Key, Val, Tree2}
```

从树Tree1中抽取最大的元素。

```
take_smallest(Tree1) -> {Key, Val, Tree2}
```

从树Tree1中抽取最小的元素。

```
to_list(Tree) -> [{Key, Val}]
```

将树Tree转换为列表。

```
update(Key, Val, Tree1) -> Tree2
```

更新树Tree1中键为Key记录的值。

```
values(Tree) -> [Val]
```

获取树Tree的所有值。

F.28 gen_event模块

通用的消息处理行为。

```
Module:code_change(OldVsn, State, Extra) -> {ok, NewState}
```

代码升级或降级时执行的回调函数。

```
Module:handle_call(Request, State) -> Result
```

处理同步请求的回调函数。

```
Module:handle_event(Event, State) -> Result
```

处理事件的回调函数。

```
Module:handle_info(Info, State) -> Result
```

处理其他消息的回调函数。

```
Module:init(InitArgs) -> {ok,State}
```

初始化事件处理程序的回调函数。

```
Module:terminate(Arg, State) -> term()
```

终止时的回调函数。

```
add_handler(EventMgrRef, Handler, Args) -> Result
```

增加一个事件处理程序。

```
add_sup_handler(EventMgrRef, Handler, Args) -> Result
```

增加一个受监控的事件处理程序。

```
call(EventMgrRef, Handler, Request, Timeout) -> Result
```

同步调用一个事件处理进程。

```
delete_handler(EventMgrRef, Handler, Args) -> Result
```

删除一个事件处理程序。

```
start(EventMgrName) -> Result
```

启动一个独立的事件管理进程。

```
start_link(EventMgrName) -> Result
```

启动一个受监控的事件管理进程。

```
stop(EventMgrRef) -> ok
```

终止一个事件管理进程。

471

```
swap_handler(EventMgrRef, {Handler1,Args1}, {Handler2,Args2}) -> Result
```

替换事件处理程序。

```
swap_sup_handler(EventMgrRef, {Handler1,Args1}, {Handler2,Args2}) -> Result
```

替换事件处理程序。

```
sync_notify(EventMgrRef, Event) -> ok
```

向事件处理进程发起一个同步的事件通知。

```
which_handlers(EventMgrRef) -> [Handler]
```

获取事件管理器的事件处理程序。

F.29 gen_fsm模块

通用的有限状态机行为。

```
Module:StateName(Event, StateData) -> Result
```
处理异步事件的回调函数。
```
Module:StateName(Event, From, StateData) -> Result
```
处理同步事件的回调函数。
```
Module:code_change(OldVsn, StateName, StateData, Extra) ->
{ok, NextStateName, NewStateData}
```
执行代码升级/降级操作时的回调函数。
```
Module:handle_event(Event, StateName, StateData) -> Result
```
处理异步消息的回调函数。
```
Module:handle_info(Info, StateName, StateData) -> Result
```
处理其他消息的回调函数。
```
Module:handle_sync_event(Event, From, StateName, StateData) -> Result
```
处理同步消息的回调函数。
```
Module:init(Args) -> Result
```
初始化的回调函数。
```
Module:terminate(Reason, StateName, StateData)
```
结束时的回调函数。
```
cancel_timer(Ref) -> RemainingTime | false
```
取消状态机内部的计时器。
```
enter_loop(Module, Options, StateName, StateData, FsmName, Timeout)
```
进入状态已的消息接收循环。
```
reply(Caller, Reply) -> true
```
给调用者发送反馈。
```
send_all_state_event(FsmRef, Event) -> ok
```
向状态机发出异步事件。
```
send_event(FsmRef, Event) -> ok
```
向状态机发出异步事件。
```
send_event_after(Time, Event) -> Ref
```
472 在状态机内发送延迟事件。
```
start(FsmName, Module, Args, Options) -> Result
```
创建一个独立的状态机进程。
```
start_link(FsmName, Module, Args, Options) -> Result
```
创建一个受监控的状态机进程。
```
start_timer(Time, Msg) -> Ref
```
在状态机内开始一个内部的计时器。
```
sync_send_all_state_event(FsmRef, Event, Timeout) -> Reply
```
向状态机发出同步事件。
```
sync_send_event(FsmRef, Event, Timeout) -> Reply
```
向状态机发出同步事件。

F.30　gen_sctp模块

gen_sctp模块提供了在套接字上使用SCTP（流控制传输协议）进行通信所需的函数。

```
abort(sctp_socket(), Assoc) -> ok | {error, posix()}
```
非正常的终止Assoc关联，不刷新未发送数据。
```
close(sctp_socket()) -> ok | {error, posix()}
```
完全关闭sctp_socket端口及其所有的关联。
```
connect(Socket, IP, Port, Opts) -> {ok,Assoc} | {error, posix()}
```
在Socket上建立新的关联。
```
connect(Socket, IP, Port, [Opt], Timeout) -> {ok, Assoc} | {error, posix()}
```
在Socket上建立新的关联。
```
controlling_process(sctp_socket(), pid()) -> ok
```
将PID进程设定为端口新的控制进程。
```
eof(Socket, Assoc) -> ok | {error, Reason}
```
妥善关闭Assoc关联，刷新未发送的数据。
```
error_string(integer()) -> ok | string() | undefined
```
将SCTP错误代码转换为提示信息。
```
listen(Socket, IsServer) -> ok | {error, Reason}
```
设置一个监听端口。
```
open([Opt]) -> {ok, Socket} | {error, posix()}
```
创建一个SCTP端口，绑定到本地地址。
```
recv(sctp_socket(), timeout()) -> {ok, {FromIP, FromPort, AncData, Data}} | {error, Reason}
```
从端口接收消息。
```
send(Socket, SndRcvInfo, Data) -> ok | {error, Reason}
```
通过端口发送消息。
```
send(Socket, Assoc, Stream, Data) -> ok | {error, Reason}
```
通过端口上已建立的关联和流来发送消息。

473

F.31 gen_server模块

通用的服务器行为。
```
Module:code_change(OldVsn, State, Extra) -> {ok, NewState}
```
代码升级/降级的回调函数。
```
Module:handle_call(Request, From, State) -> Result
```
处理同步调用的回调函数。
```
Module:handle_cast(Request, State) -> Result
```
处理异步调用的回调函数。
```
Module:handle_info(Info, State) -> Result
```
处理其他消息的回调函数。
```
Module:init(Args) -> Result
```
初始化的回调函数。
```
Module:terminate(Reason, State)
```
结束时的回调函数。
```
abcast(Nodes, Name, Request) -> abcast
```
向多个服务器进程发送异步请求。

```
call(ServerRef, Request, Timeout) -> Reply
```
向服务器进程发送同步请求。
```
cast(ServerRef, Request) -> ok
```
向服务器进程发送异步请求。
```
enter_loop(Module, Options, State, ServerName, Timeout)
```
进入服务器进程的消息接收循环。
```
multi_call(Nodes, Name, Request, Timeout) -> Result
```
向多个服务器进程发起异步调用。
```
reply(Client, Reply) -> true
```
向调用者发送反馈。
```
start(ServerName, Module, Args, Options) -> Result
```
启动一个独立的服务器进程。
```
start_link(ServerName, Module, Args, Options) -> Result
```
启动一个受监控的服务器进程。

F.32　gen_tcp模块

TCP/IP通信接口。
```
accept(ListenSocket, Timeout) -> {ok, Socket} | {error, Reason}
```
监听端口接受一个连接请求。
```
close(Socket) -> ok | {error, Reason}
```
关闭一个端口。
```
connect(Address, Port, Options, Timeout) -> {ok, Socket} | {error, Reason}
```
474 连接到一个TCP端口。
```
controlling_process(Socket, Pid) -> ok | {error, eperm}
```
将Pid进程设置为端口新的控制进程。
```
listen(Port, Options) -> {ok, ListenSocket} | {error, Reason}
```
设置一个监听端口。
```
recv(Socket, Length, Timeout) -> {ok, Packet} | {error, Reason}
```
通过被动模式的端口接收数据包。
```
send(Socket, Packet) -> ok | {error, Reason}
```
通过端口发送数据包。
```
shutdown(Socket, How) -> ok | {error, Reason}
```
立即关闭一个端口。

F.33　gen_udp模块

UDP通信接口。
```
close(Socket) -> ok | {error, Reason}
```
关闭一个UDP端口。
```
controlling_process(Socket, Pid) -> ok
```
设置Pid进程为端口新的控制进程。
```
open(Port, Options) -> {ok, Socket} | {error, Reason}
```

打开指定的UDP端口。

```
recv(Socket, Length, Timeout) -> {ok, {Address, Port, Packet}} | {error, Reason}
```

通过被动模式的端口接收数据包。

```
send(Socket, Address, Port, Packet) -> ok | {error, Reason}
```

通过端口发送数据包。

F.34 global模块

全局名称注册机制。

```
del_lock(Id, Nodes) -> void()
```

删除一个锁定。

```
notify_all_name(Name, Pid1, Pid2) -> none
```

注册和重新注册时，用此函数通知相关进程。

```
random_exit_name(Name, Pid1, Pid2) -> Pid1 | Pid2
```

注册和重新注册时，用此函数终止相关进程。

```
random_notify_name(Name, Pid1, Pid2) -> Pid1 | Pid2
```

注册和重新注册时，用此函数通知相关进程。

```
re_register_name(Name, Pid, Resolve) -> void()
```

（原子化地）重新注册一个名称。

```
register_name(Name, Pid, Resolve) -> yes | no
```

将Pid进程注册到全局名称Name。

475

```
registered_names() -> [Name]
```

获取所有已注册的全局名称。

```
send(Name, Msg) -> Pid
```

向已注册的全局名称进程发送消息。

```
set_lock(Id, Nodes, Retries) -> boolean()
```

对Nodes列表中的节点设置锁定。

```
sync() -> void()
```

同步全局的名称服务。

```
trans(Id, Fun, Nodes, Retries) -> Res | aborted
```

在Id锁定上以交易方式执行Fun。

```
unregister_name(Name) -> void()
```

注销全局名称Name。

```
whereis_name(Name) -> pid() | undefined
```

获取全局名称Name对应的Pid。

F.35 inet模块

TCP/IP协议相关函数。

```
close(Socket) -> ok
```

关闭端口。

```
format_error(Posix) -> string()
```

获取Posix错误的文字描述。

```
get_rc() -> [{Par, Val}]
```
获取IP配置的参数。
```
getaddr(Host, Family) -> {ok, Address} | {error, posix()}
```
获取Host的IP地址。
```
getaddrs(Host, Family) -> {ok, Addresses} | {error, posix()}
```
获取Host的IP地址列表。
```
gethostbyaddr(Address) -> {ok, Hostent} | {error, posix()}
```
获取Address地址的Host条目。
```
gethostbyname(Name) -> {ok, Hostent} | {error, posix()}
```
获取Name主机的Host条目。
```
gethostbyname(Name, Family) -> {ok, Hostent} | {error, posix()}
```
获取Name主机的Host条目。
```
gethostname() -> {ok, Hostname} | {error, posix()}
```
获取本地主机的名称。
```
getopts(Socket, Options) -> OptionValues | {error, posix()}
```
获取Socket的选项。
```
peername(Socket) -> {ok, {Address, Port}} | {error, posix()}
```
476 获取Socket远端的地址和端口号。
```
port(Socket) -> {ok, Port}
```
获取Socket的本地端口号。
```
setopts(Socket, Options) -> ok | {error, posix()}
```
给Socket设置属性。
```
sockname(Socket) -> {ok, {Address, Port}} | {error, posix()}
```
获取Socket本地的地址和端口号。

F.36　init模块

系统启动调度。
```
boot(BootArgs) -> void()
```
启动Erlang的运行时系统。
```
get_args() -> [Arg]
```
获取非标志的命令行参数。
```
get_argument(Flag) -> {ok, Arg} | error
```
获取命令行Flag标志的参数。
```
get_arguments() -> Flags
```
获取命令行所有标志的参数。
```
get_plain_arguments() -> [Arg]
```
获取非标志的命令行参数。
```
get_status() -> {InternalStatus, ProvidedStatus}
```
获取系统状态信息。
```
reboot() -> void()
```
顺畅的终止Erlang节点。

```
restart() -> void()
```
重启运行中的Erlang节点。
```
script_id() -> Id
```
获取用户启动脚本的标识。
```
stop() -> void()
```
顺畅的终止Erlang节点。

F.37 io模块

标准IO服务接口函数。
```
format([IoDevice,] Format, Data) -> ok
```
输出格式化的数据。
```
fread([IoDevice,] Prompt, Format) -> Result
```
读取格式化的输入。
```
  get_chars([IoDevice,] Prompt, Count) -> string() | eof
```
读取指定长度的字符。
```
get_line([IoDevice,] Prompt) -> string() | eof
```
读取一行。
```
nl([IoDevice]) -> ok
```
输出一个换行符号。
```
parse_erl_exprs([IoDevice,] Prompt, StartLine) -> Result
```
读取并解析Erlang表达式。
```
parse_erl_form([IoDevice,] Prompt, StartLine) -> Result
```
读取并解析Erlang表达式。
```
put_chars([IoDevice,] IoData) -> ok
```
输出字符串。
```
read([IoDevice,] Prompt) -> Result
```
读取值。
```
read(IoDevice, Prompt, StartLine) -> Result
```
读取值。
```
scan_erl_exprs([IoDevice,] Prompt, StartLine) -> Result
```
读取并扫描Erlang表达式。
```
scan_erl_form([IoDevice,] Prompt, StartLine) -> Result
```
读取并扫描Erlang表达式。
```
setopts([IoDevice,] Opts) -> ok | {error, Reason}
```
设置属性。
```
write([IoDevice,] Term) -> ok
```
输出值。

F.38 io_lib模块

IO库函数。
```
char_list(Term) -> bool()
```

477

检查Term是否为字符列表。

```
deep_char_list(Term) -> bool()
```

检查Term是否为深层字符列表。

```
format(Format, Data) -> chars()
```

格式化数据。

```
fread(Format, String) -> Result
```

读取格式化输入。

```
fread(Continuation, String, Format) -> Return
```

继续读取格式化输入。

```
indentation(String, StartIndent) -> int()
```

输出字符串后进行缩进。

```
nl() -> chars()
```

478 输出换行符号。

```
print(Term, Column, LineLength, Depth) -> chars()
```

美化打印Term值。

```
printable_list(Term) -> bool()
```

测试Term是否为可打印字符。

```
write(Term, Depth) -> chars()
```

输出Term。

```
write_atom(Atom) -> chars()
```

输出原子值Atom。

```
write_char(Integer) -> chars()
```

输出字符值Integer。

```
write_string(String) -> chars()
```

输出字符串String。

F.39　lib模块

一些有用的库函数。

```
error_message(Format, Args) -> ok
```

输出错误信息。

```
flush_receive() -> void()
```

刷新消息。

```
nonl(String1) -> String2
```

去掉最后一个换行符号。

```
progname() -> atom()
```

返回Erlang启动脚本的名称。

```
send(To, Msg)
```

发送消息。

```
sendw(To, Msg)
```

发送消息，并等待回应。

F.40 lists模块

列表处理函数。

all(Pred, List) -> bool()

List中是否所有的元素都满足Pred条件。

any(Pred, List) -> bool()

List中是否有能满足Pred条件的元素。

append(ListOfLists) -> List1

合并ListOfLists中的列表。

append(List1, List2) -> List3

合并两个列表。

concat(Things) -> string()

合并原子列表。

479

delete(Elem, List1) -> List2

从列表List1中删除Elem元素。

dropwhile(Pred, List1) -> List2

从列表List1中删除符合Pred条件的元素。

duplicate(N, Elem) -> List

构造一个由N个Elem组成的列表。

filter(Pred, List1) -> List2

用Pred条件过滤列表List1。

flatlength(DeepList) -> int()

获取深层列表DeepList扁平化之后的列表长度。

flatmap(Fun, List1) -> List2

对List1列表进行扁平化并通过Fun函数映射生成新列表。

flatten(DeepList) -> List

对深层列表DeepList进行扁平化处理。

flatten(DeepList, Tail) -> List

对深层列表DeepList进行扁平化处理。

foldl(Fun, Acc0, List) -> Acc1

在列表List上以foldl方式执行Fun函数。

foldr(Fun, Acc0, List) -> Acc1

在列表List上以foldr方式执行Fun函数。

foreach(Fun, List) -> void()

对列表List的每一个元素执行Fun函数。

keydelete(Key, N, TupleList1) -> TupleList2

从元组列表TupleList1中删除第N个字段为Key的记录。

keymap(Fun, N, TupleList1) -> TupleList2

对元组列表TupleList1通过Fun函数映射生成新列表。

keymember(Key, N, TupleList) -> bool()

检查元组列表TupleList中是否存在第N个字段为Key的数据。

keymerge(N, TupleList1, TupleList2) -> TupleList3

将两个以第N个字段为键的元组列表TupleList1和TupleList2合并。

```
keyreplace(Key, N, TupleList1, NewTuple) -> TupleList2
```

替换元组列表TupleList1当中的元素。

```
keysearch(Key, N, TupleList) -> {value, Tuple} | false
```

在元组列表TupleList1中搜索特定元素。

```
keysort(N, TupleList1) -> TupleList2
```

对元组列表TupleList1进行排序。

```
last(List) -> Last
```

480 获取列表List的最后一个元素。

```
map(Fun, List1) -> List2
```

用Fun将列表List1映射为新的列表。

```
mapfoldl(Fun, Acc0, List1) -> {List2, Acc1}
```

同时进行map和fold操作。

```
mapfoldr(Fun, Acc0, List1) -> {List2, Acc1}
```

同时进行map和fold操作。

```
max(List) -> Max
```

返回列表List中最大的元素。

```
member(Elem, List) -> bool()
```

检查Elem是否为列表List中的元素。

```
merge(ListOfLists) -> List1
```

合并列表ListOfLists中的有序列表。

```
merge(List1, List2) -> List3
```

合并有序列表List1和List2。

```
merge(Fun, List1, List2) -> List3
```

用Fun函数合并有序列表List1和List2。

```
merge3(List1, List2, List3) -> List4
```

合并3个有序列表。

```
min(List) -> Min
```

获取列表List中最小的元素。

```
nth(N, List) -> Elem
```

获取列表List第N个元素。

```
nthtail(N, List1) -> Tail
```

获取列表List第N个元素之后的尾部。

```
partition(Pred, List) -> {Satisfying, NonSatisfying}
```

用Pred函数将列表List分为两个部分。

```
prefix(List1, List2) -> bool()
```

判断List1是否是List2的前缀。

```
reverse(List1) -> List2
```

反转列表List1的顺序。

```
reverse(List1, Tail) -> List2
```

反转列表List1，然后将Tail加在后面。

```
seq(From, To, Incr) -> Seq
```

生成由整数序列构成的列表。

sort(List1) -> List2

对List1进行排序。

sort(Fun, List1) -> List2

用Fun函数对List1进行排序。

split(N, List1) -> {List2, List3}

从第N个元素起，将列表List1分为两段。

481

splitwith(Pred, List) -> {List1, List2}

按Pred条件将List分为两段。

sublist(List1, Len) -> List2

获取列表List1长度为Len的部分。

sublist(List1, Start, Len) -> List2

获取列表List1从Start起长度为Len的部分。

subtract(List1, List2) -> List3

对列表List1和List2做逻辑减。

suffix(List1, List2) -> bool()

判断列表List1是否为列表List2的后缀。

sum(List) -> number()

获取列表List中所有元素的和。

takewhile(Pred, List1) -> List2

从列表List1中去除满足Pred条件的元素。

ukeymerge(N, TupleList1, TupleList2) -> TupleList3

将两个元组列表TupleList1和TupleList2合并，去除重复的元素。

ukeysort(N, TupleList1) -> TupleList2

对元组列表TupleList1进行排序，去除重复的元素。

umerge(ListOfLists) -> List1

合并列表ListOfLists中的有序列表，去除重复的元素。

umerge(List1, List2) -> List3

合并有序列表List1和List2，去除重复的元素。

umerge(Fun, List1, List2) -> List3

用Fun函数合并有序列表List1和List2，去除重复的元素。

umerge3(List1, List2, List3) -> List4

合并3个有序列表，去除重复的元素。

unzip(List1) -> {List2, List3}

将由两个元组组成的列表List1分解成为两个列表。

unzip3(List1) -> {List2, List3, List4}

将由3个元组组成的列表List1分解成为3个列表。

usort(List1) -> List2

对列表List1排序，去除重复的元素。

usort(Fun, List1) -> List2

用Fun函数对列表List1排序，去除重复的元素。

zip(List1, List2) -> List3

将列表List1和List2合成为一个列表。

```
zip3(List1, List2, List3) -> List4
```

将3个列表合成为一个列表。

```
zipwith(Combine, List1, List2) -> List3
```

482 用Combine函数将列表List1和List2合成为一个列表。

```
zipwith3(Combine, List1, List2, List3) -> List4
```

用Combine函数将3个列表合成为一个列表。

F.41　math模块

数学函数。

```
erf(X) -> float()
```

误差函数。

```
erfc(X) -> float()
```

另外一个误差函数。

```
pi() -> float()
```

获取圆周率。

```
sqrt(X)
```

对X开平方。

F.42　ms_transform模块

将函数语法转换为匹配规则的转换函数。

```
format_error(Errcode) -> ErrMessage
```

获取错误代码Errcode的文字描述。

```
parse_transform(Forms,_Options) -> Forms
```

将Erlang的抽象格式转换为匹配规则文法。

```
transform_from_shell(Dialect,Clauses,BoundEnvironment) -> term()
```

在shell中执行转换时，会用到这个函数。

F.43　net_adm模块

Erlang的网络管理功能。

```
dns_hostname(Host) -> {ok, Name} | {error, Host}
```

获取Host主机的正式名称。

```
host_file() -> Hosts | {error, Reason}
```

读取hosts.erlang文件。

```
localhost() -> Name
```

获取当前主机的名字。

```
names(Host) -> {ok, [{Name, Port}]} | {error, Reason}
```

获取Host主机上的Erlang节点。

```
ping(Node) -> pong | pang
```

建立到Node节点的连接。

```
world(Arg) -> [node()]
```
查找并连接hosts.erlang文件中的所有节点。 483
```
world_list(Hosts, Arg) -> [node()]
```
查找并连接hosts列表中的所有节点。

F.44 net_kernel模块

Erlang网络核心。
```
allow(Nodes) -> ok | error
```
允许Nodes列表中的节点访问。
```
connect_node(Node) -> true | false | ignored
```
建立到Node节点的连接。
```
get_net_ticktime() -> Res
```
获取网络检测时间。
```
monitor_nodes(Flag, Options) -> ok | Error
```
订阅节点的状态变化消息。
```
set_net_ticktime(NetTicktime, TransitionPeriod) -> Res
```
设置网络检测时间。
```
start([Name, NameType, Ticktime]) -> {ok, pid()} | {error, Reason}
```
将当前节点连入分布式系统。
```
stop() -> ok | {error, not_allowed | not_found}
```
将当前节点从分布式系统中断开。

F.45 os模块

与特定操作系统相关的函数。
```
cmd(Command) -> string()
```
在目标操作系统的shell中执行Command命令。
```
find_executable(Name, Path) -> Filename | false
```
获取可执行程序的绝对地址。
```
getenv() -> [string()]
```
获取环境变量列表。
```
getenv(VarName) -> Value | false
```
获取环境变量VarName的值。
```
getpid() -> Value
```
获取Erlang虚拟机的进程标识（操作系统的进程标识）。
```
putenv(VarName, Value) -> true
```
设置环境变量。
```
type() -> {Osfamily, Osname} | Osfamily
```
获取当前操作系统的家族/名称。
```
version() -> {Major, Minor, Release} | VersionString
```
获取当前操作系统的版本号。 484

F.46　proc_lib 模块

遵循OTP设计原则的同步和异步进程启动函数。

`format(CrashReport) -> string()`

格式化崩溃报告。

`hibernate(Module, Function, Args)`

休眠进程，直到有消息发送给它。

`init_ack(Ret) -> void()`

进程启动时会用到这个函数。

`initial_call(Process) -> {Module,Function,Args} | Fun | false`

从进程（由proc_lib启动的进程）中提取初始化信息。

`spawn(Node, Module, Function, Args) -> pid()`

启动一个新进程。

`spawn_link(Node, Module, Function, Args) -> pid()`

启动一个新进程，并建立连接。

`spawn_opt(Node, Module, Func, Args, SpawnOpts) -> pid()`

用特定的属性启动一个新进程。

`start_link(Module, Function, Args, Time, SpawnOpts) -> Ret`

同步启动一个新进程。

`translate_initial_call(Process) -> {Module,Function,Arity} | Fun`

从进程（由proc_lib启动的进程）中提取初始化信息，并进行转换。

F.47　qlc模块

针对Mnesia、ETS、DETS等的查询接口。

`append(QHL) -> QH`

获取查询句柄。

`append(QH1, QH2) -> QH3`

获取查询句柄。

`cursor(QueryHandleOrList [, Options]) -> QueryCursor`

创建查询游标。

`delete_cursor(QueryCursor) -> ok`

删除查询游标。

`e(QueryHandleOrList [, Options]) -> Answers`

获取查询的所有结果。

`fold(Function, Acc0, QueryHandleOrList [, Options]) -> Acc1 | Error`

在查询结果上以fold方式执行Function函数。

`format_error(Error) -> Chars`

485 获取错误的描述信息。

`info(QueryHandleOrList [, Options]) -> Info`

获取查询句柄的信息。

`keysort(KeyPos, QH1 [, SortOptions]) -> QH2`

获取查询句柄。

```
next_answers(QueryCursor [, NumberOfAnswers]) -> Answers | Error
```

获取查询的部分或全部结果。

```
q(QueryListComprehension [, Options]) -> QueryHandle
```

获取列表查询语法的句柄。

```
sort(QH1 [, SortOptions]) -> QH2
```

获取查询句柄。

```
string_to_handle(QueryString [, Options [, Bindings]]) -> QueryHandle | Error
```

获取列表查询语法的句柄。

```
table(TraverseFun, Options) -> QueryHandle
```

获取表的查询句柄。

F.48 queue模块

先进先出的队列抽象数据结构。

```
cons(Item, Q1) -> Q2
```

在队列Q1的头部插入元素。

```
daeh(Q) -> Item
```

获取队列Q的最后一个元素。

```
from_list(L) -> queue()
```

将列表L转换为队列。

```
head(Q) -> Item
```

获取队列Q中位于头部的元素。

```
in(Item, Q1) -> Q2
```

在队列Q1尾部插入Item元素。

```
in_r(Item, Q1) -> Q2
```

在队列Q1头部插入Item元素。

```
init(Q1) -> Q2
```

去掉队列Q1的最后一个元素。

```
is_empty(Q) -> true | false
```

判断队列Q是否为空。

```
join(Q1, Q2) -> Q3
```

合并队列Q1和Q2。

```
lait(Q1) -> Q2
```

去掉队列Q1的最后一个元素。

```
last(Q) -> Item
```

获取队列Q的最后一个元素。

486

```
len(Q) -> N
```

获取队列Q的长度。

```
new() -> Q
```

创建一个队列。

```
out(Q1) -> Result
```

去掉队列Q1的第一个元素。

```
out_r(Q1) -> Result
```

去掉队列Q1的最后一个元素。

```
reverse(Q1) -> Q2
```

反转队列Q1。

```
snoc(Q1, Item) -> Q2
```

在队列Q1的末尾插入Item元素。

```
split(N, Q1) -> {Q2,Q3}
```

将队列Q1在第N个元素处截为两个队列。

```
tail(Q1) -> Q2
```

去掉队列Q1的第一个元素。

```
to_list(Q) -> list()
```

将队列Q转换为列表。

F.49　random模块

产生伪随机数。

```
seed() -> ran()
```

用默认值产生随机数种子。

```
seed(A1, A2, A3) -> ran()
```

用数字产生随机数种子。

```
seed0() -> ran()
```

获取产生随机数的默认状态。

```
uniform()->float()
```

产生一个随机的浮点数。

```
uniform(N) -> int()
```

产生一个随机的整数。

```
uniform_s(State0) -> {float(), State1}
```

产生一个随机的浮点数。

```
uniform_s(N, State0) -> {int(), State1}
```

487　产生一个随机的整数。

F.50　regexp模块

处理字符串的正则表达式函数。

```
first_match(String, RegExp) -> MatchRes
```

匹配正则表达式。

```
format_error(ErrorDescriptor) -> Chars
```

获取错误的文字描述。

```
gsub(String, RegExp, New) -> SubRes
```

用正则表达式进行全局替换。

```
match(String, RegExp) -> MatchRes
```

匹配正则表达式。

matches(String, RegExp) -> MatchRes
匹配正则表达式。
parse(RegExp) -> ParseRes
解析正则表达式。
sh_to_awk(ShRegExp) -> AwkRegExp
将sh风格的正则表达式转换为awk风格。
split(String, RegExp) -> SplitRes
用正则表达式切分字符串。
sub(String, RegExp, New) -> SubRes
用正则表达式进行替换（只替换第一个匹配的结果）。

F.51 rpc模块

远程函数调用服务。
abcast(Name, Msg) -> void()
异步的向所有节点上的Name进程发送Msg消息。
abcast(Nodes, Name, Msg) -> void()
异步的向Node节点上的Name进程发送Msg消息。
async_call(Node, Module, Function, Args) -> Key
异步调用Node节点上的函数。
block_call(Node, Module, Function, Args) -> Res | {badrpc, Reason}
使用当前进程，同步调用Node节点上的函数。
block_call(Node, Module, Function, Args, Timeout) -> Res | {badrpc, Reason}
使用当前进程，同步调用Node节点上的函数。
call(Node, Module, Function, Args) -> Res | {badrpc, Reason}
同步调用Node节点上的函数。
call(Node, Module, Function, Args, Timeout) -> Res | {badrpc, Reason}
同步调用Node节点上的函数。
488
cast(Node, Module, Function, Args) -> void()
异步调用Node节点上的函数，忽略结果。
eval_everywhere(Module, Funtion, Args) -> void()
异步调用所有节点上的函数，忽略结果。
eval_everywhere(Nodes,Module, Function, Args) -> void()
异步调用所有节点上的函数，忽略结果。
multi_server_call(Name, Msg) -> {Replies, BadNodes}
同步调用多个节点上的函数。
multi_server_call(Nodes, Name, Msg) -> {Replies, BadNodes}
同步调用多个节点上的函数。
multicall(Module, Function, Args) -> {ResL, BadNodes}
同步调用多个节点上的函数。
multicall(Module, Function, Args, Timeout) -> {ResL, BadNodes}
同步调用多个节点上的函数。

multicall(Nodes, Module, Function, Args) -> {ResL, BadNodes}
同步调用多个节点上的函数。
multicall(Nodes, Module, Function, Args, Timeout) -> {ResL, BadNodes}
同步调用多个节点上的函数。
nb_yield(Key) -> {value, Val} | timeout
非阻塞的投递函数的调用结果。
nb_yield(Key, Timeout) -> {value, Val} | timeout
非阻塞的投递函数的调用结果。
parallel_eval(FuncCalls) -> ResL
并行的在所有节点上执行函数调用。
pinfo(Pid) -> [{Item, Info}] | undefined
获取Pid进程的信息。
pinfo(Pid, Item) -> {Item, Info} | undefined | []
获取Pid进程的Item项信息。
pmap({Module, Function}, ExtraArgs, List2) -> List1
并行的以map方式对列表执行Function函数。
safe_multi_server_call(Nodes, Name, Msg) -> {Replies, BadNodes}
同步调用多个节点上的函数（不建议使用）。
sbcast(Name, Msg) -> {GoodNodes, BadNodes}
同步的向所有节点上的Name进程发送Msg消息。
sbcast(Nodes, Name, Msg) -> {GoodNodes, BadNodes}
同步的向Nodes节点上的Name进程发送Msg消息。
server_call(Node, Name, ReplyWrapper, Msg) -> Reply | {error, Reason}
同步调用Node节点上的函数。
yield(Key) -> Res | {badrpc, Reason}
489 （阻塞的）投递函数的调用结果。

F.52　seq_trace模块

消息的顺序跟踪。
get_system_tracer() -> Tracer
获取当前系统跟踪器的标识。
get_token() -> TraceToken
获取跟踪令牌。
get_token(Component) -> {Component, Val}
获取跟踪令牌的属性。
print(TraceInfo) -> void()
在顺序跟踪输出上打印跟踪信息。
print(Label, TraceInfo) -> void()
在顺序跟踪输出上打印跟踪信息。
reset_trace() -> void()
停止当前节点的顺序跟踪。

set_system_tracer(Tracer) -> OldTracer
设置系统跟踪器。
set_token(Token) -> PreviousToken
设置跟踪令牌。
set_token(Component, Val) -> {Component, OldVal}
设置跟踪令牌的属性。

F.53 sets模块

集合操作函数。
add_element(Element, Set1) -> Set2
向Set1集合加入Element元素。
del_element(Element, Set1) -> Set2
从Set1集合中删除Element元素。
filter(Pred, Set1) -> Set2
用Pred函数过滤Set1集合。
fold(Function, Acc0, Set) -> Acc1
对Set集合以fold方式执行Function函数。
from_list(List) -> Set
将List列表转换为集合。
intersection(SetList) -> Set
获取SetList列表中所有集合的交集。
intersection(Set1, Set2) -> Set3
获取集合Set1和Set2的交集。
is_element(Element, Set) -> bool()
判断Element是否在Set集合之中。
is_set(Set) -> bool()
判断Set是否为集合。
is_subset(Set1, Set2) -> bool()
判断Set1是否为Set2的子集。
new() -> Set
创建一个空的集合。
size(Set) -> int()
获取集合Set的元素数量。
subtract(Set1, Set2) -> Set3
对集合Set1和Set2执行逻辑减。
to_list(Set) -> List
将Set转换为列表。
union(SetList) -> Set
获取SetList列表中所有集合的并集。
union(Set1, Set2) -> Set3
取集合Set1和Set2的并集。

490

F.54　shell模块

Erlang的shell。

`history(N) -> integer()`

设置保存前N个命令。

`results(N) -> integer()`

设置保存前N个命令。

`start_restricted(Module) -> ok`

退出正常的shell，开始受限的shell。

`stop_restricted() -> ok`

退出受限的shell，返回正常的shell。

F.55　slave模块

启动和控制从属节点的函数。

`pseudo([Master | ServerList]) -> ok`

启动若干个伪服务器。

`pseudo(Master, ServerList) -> ok`

启动若干个伪服务器。

`relay(Pid)`

运行伪服务器。

`start(Host, Name, Args) -> {ok, Node} | {error, Reason}`

491

在Host上启动从节点。

`start_link(Host, Name, Args) -> {ok, Node} | {error, Reason}`

在Host上启动从节点，并建立连接。

`stop(Node) -> ok`

停止从节点。

F.56　sofs模块

操纵集族[①]的模块。

`a_function(Tuples [, Type]) -> Function`

创建函数。

`canonical_relation(SetOfSets) -> BinRel`

返回规范射。

`composite(Function1, Function2) -> Function3`

返回两个函数的复合。

`constant_function(Set, AnySet) -> Function`

创建一个将一个集合中的每个元素映射到另一个集合的函数。

`converse(BinRel1) -> BinRel2`

返回一个二元关系的逆关系。

① 所谓的集族就是类似{{a,1},{b,2},{c,3}...}这样集合的集合。——译者注

difference(Set1, Set2) -> Set3
返回两个集合的差。
digraph_to_family(Graph [, Type]) -> Family
创建一个有向图的族。
domain(BinRel) -> Set
返回一个二元关系的域。
drestriction(BinRel1, Set) -> BinRel2
返回一个二元关系的限制。
drestriction(SetFun, Set1, Set2) -> Set3
返回一个关系的限制。
empty_set() -> Set
返回无类型的空集。
extension(BinRel1, Set, AnySet) -> BinRel2
扩展一个二元关系的域。
family(Tuples [, Type]) -> Family
创建一个子集合族。
family_difference(Family1, Family2) -> Family3
返回两个集族的差。
family_domain(Family1) -> Family2
返回集族的定义域。
family_field(Family1) -> Family2
返回域的族。
family_intersection(Family1) -> Family2
返回集族中各集合的交集。
family_intersection(Family1, Family2) -> Family3
返回两个集族的交集。
family_projection(SetFun, Family1) -> Family2
返回子集集族的投影。
family_range(Family1) -> Family2
返回集族的值域。
family_specification(Fun, Family1) -> Family2
使用一个断言来选择一个集族中的某个子集。
family_to_digraph(Family [, GraphType]) -> Graph
从一个集族中创建有向图。
family_to_relation(Family) -> BinRel
从一个集族中创建一个关系。
family_union(Family1) -> Family2
返回一个集族中各个集合的并集。
family_union(Family1, Family2) -> Family3
返回两个集族的并集。
field(BinRel) -> Set
返回一个二元关系的域。

492

```
from_external(ExternalSet, Type) -> AnySet
```
创建一个集合。
```
from_sets(ListOfSets) -> Set
```
从一个集合列表中创建集合。
```
from_sets(TupleOfSets) -> Ordset
```
从一个集合元组中创建一个有序集。
```
from_term(Term [, Type]) -> AnySet
```
创建一个集合。
```
image(BinRel, Set1) -> Set2
```
返回一个集合在二元关系BinRel上的映射。
```
intersection(SetOfSets) -> Set
```
返回一组集合的交集。
```
intersection(Set1, Set2) -> Set3
```
返回两个集合的交集。
```
intersection_of_family(Family) -> Set
```
返回一个集族的交集。
```
inverse(Function1) -> Function2
```
返回一个函数的反函数。
```
inverse_image(BinRel, Set1) -> Set2
```
493 返回一个集合在二元关系BinRel上的逆映射。
```
is_a_function(BinRel) -> Bool
```
测试一个二元关系是否是一个函数。
```
is_disjoint(Set1, Set2) -> Bool
```
测试两个函数是否是不相交并集。
```
is_empty_set(AnySet) -> Bool
```
测试集合是否为空集。
```
is_equal(AnySet1, AnySet2) -> Bool
```
测试两个集合是否等价。
```
is_set(AnySet) -> Bool
```
测试一个集合是否为无序集合。
```
is_sofs_set(Term) -> Bool
```
测试一个元组是否为无序集合。
```
is_subset(Set1, Set2) -> Bool
```
测试两个集合是否包含交集。
```
is_type(Term) -> Bool
```
测试一个元组的类型。
```
join(Relation1, I, Relation2, J) -> Relation3
```
返回两个关系的复合。
```
multiple_relative_product(TupleOfBinRels, BinRel1) -> BinRel2
```
返回一个二元关系元组和一个关系之间的多重相对积。
```
no_elements(ASet) -> NoElements
```
返回集合元素的数目。

partition(SetOfSets) -> Partition
返回一个集合的集合的粗粒度划分。
partition(SetFun, Set) -> Partition
返回一个集合的划分。
partition(SetFun, Set1, Set2) -> {Set3, Set4}
返回一个集合的划分。
partition_family(SetFun, Set) -> Family
返回一个划分的族索引。
product(TupleOfSets) -> Relation
返回一个集合元组的笛卡儿积。
product(Set1, Set2) -> BinRel
返回两个集合的笛卡儿积。
projection(SetFun, Set1) -> Set2
返回一个集合的替代元素。
range(BinRel) -> Set
返回一个二元关系的值域。
relation(Tuples [, Type]) -> Relation
创建一个关系。
relation_to_family(BinRel) -> Family
从一个二元关系中创建集族。
relative_product(TupleOfBinRels [, BinRel1]) -> BinRel2
返回一个二元关系元组和一个二元关系的相对积。
relative_product(BinRel1, BinRel2) -> BinRel3
返回两个二元关系的相对积。
relative_product1(BinRel1, BinRel2) -> BinRel3
返回两个二元关系的相对积。
restriction(BinRel1, Set) -> BinRel2
返回一个二元关系的限制。
restriction(SetFun, Set1, Set2) -> Set3
返回一个集合的限制。
set(Terms [, Type]) -> Set
创建一个原子的集合或者任何一个类型的集合。
specification(Fun, Set1) -> Set2
使用一个断言选取某个子集。
strict_relation(BinRel1) -> BinRel2
用给定的关系求出严格关系。
substitution(SetFun, Set1) -> Set2
用一个给定的定义域返回一个函数。
symdiff(Set1, Set2) -> Set3
返回两个集合的对称差。
symmetric_partition(Set1, Set2) -> {Set3, Set4, Set5}
返回两个集合的划分。

494

to_external(AnySet) -> ExternalSet
返回一个集合的元素。
to_sets(ASet) -> Sets
返回一个集合元素的列表或者元组。
type(AnySet) -> Type
返回一个集合的类型。
union(SetOfSets) -> Set
返回一个集合的集合的并集。
union(Set1, Set2) -> Set3
返回两个集合的并集。
union_of_family(Family) -> Set
返回一个集族的并集。
weak_relation(BinRel1) -> BinRel2
495 返回一个二元关系的弱关系。

F.57 string模块

字符串处理函数。
centre(String, Number, Character) -> Centered
让String中间对齐。
chars(Character, Number, Tail) -> String
获取由Number个Character字符组成的字符串。
concat(String1, String2) -> String3
连接两个字符串。
copies(String, Number) -> Copies
复制字符串。
cspan(String, Chars) -> Length
从字符串String的开头获取Chars出现的跨度。
equal(String1, String2) -> bool()
判断String1和String2是否相等。
left(String, Number, Character) -> Left
让String左对齐。
len(String) -> Length
获取String的长度。
rchr(String, Character) -> Index
获取String中第一次/最后一次出现Character字符的位置。
right(String, Number, Character) -> Right
让String右对齐。
rstr(String, SubString) -> Index
获取SubString在String中出现的位置。
strip(String, Direction, Character) -> Stripped
截去String中开始/结束的Character字符。

```
sub_string(String, Start, Stop) -> SubString
```
获取String的一部分。
```
sub_word(String, Number, Character) -> Word
```
获取String中单词的一部分。
```
substr(String, Start, Length) -> Substring
```
获取String的一部分。
```
to_float(String) -> {Float,Rest} | {error,Reason}
```
解析字符中的浮点数。
```
to_integer(String) -> {Int,Rest} | {error,Reason}
```
解析字符中的整数。
```
to_upper(Char) -> CharResult
```
将Char转换为大写。
```
tokens(String, SeparatorList) -> Tokens
```
496
将String按分割字符打散。
```
words(String, Character) -> Count
```
统计以空格分隔的单词数目。

F.58 supervisor模块

通用监控行为。
```
Module:init(Args) -> Result
```
初始化回调函数，返回监控规范。
```
check_childspecs([ChildSpec]) -> Result
```
检查规范的语法是否正确。
```
delete_child(SupRef, Id) -> Result
```
从监控树中删除一个节点。
```
restart_child(SupRef, Id) -> Result
```
重新启动监控树中的子进程。
```
start_child(SupRef, ChildSpec) -> Result
```
向监控树动态增加一个节点。
```
start_link(SupName, Module, Args) -> Result
```
创建一个监控进程。
```
terminate_child(SupRef, Id) -> Result
```
终止监控树中的子进程。
```
which_children(SupRef) -> [{Id,Child,Type,Modules}]
```
获取某监控树下所有子进程的监控规范和进程。

F.59 sys模块

系统消息接口函数。
```
Mod:system_code_change(Misc,Module, OldVsn, Extra) -> {ok, NMisc}
```
系统代码更新时的回调函数。
```
Mod:system_continue(Parent, Debug, Misc)
```

系统恢复运行时的回调函数。

```
Mod:system_terminate(Reason, Parent, Debug, Misc)
```

系统终止时的回调函数。

```
change_code(Name, Module, OldVsn, Extra, Timeout) -> ok | {error, Reason}
```

向进程发送代码变更的系统消息。

```
debug_options(Options) -> [dbg_opt()]
```

将调试属性列表转换为调试数据结构。

```
get_debug(Item,Debug,Default) -> term()
```

497 获取调试属性。

```
get_status(Name,Timeout) -> {status, Pid, {module, Mod}, [PDict, SysState, Parent, Dbg,
 Misc]}
```

获取进程状态。

```
handle_debug([dbg_opt()],FormFunc,Extra,Event) -> [dbg_opt()]
```

产生一个系统消息。

```
handle_system_msg(Msg,From,Parent,Module,Debug,Misc)
```

处理系统消息。

```
install(Name,{Func,FuncState},Timeout)
```

在进程中安装调试函数。

```
log(Name,Flag,Timeout) -> ok | {ok, [system_event()]}
```

在内存中记录系统消息。

```
log_to_file(Name,Flag,Timeout) -> ok | {error, open_file}
```

在文件中记录系统消息。

```
no_debug(Name,Timeout) -> void()
```

关闭调试。

```
print_log(Debug) -> void()
```

打印调试数据结构中记载的消息。

```
remove(Name,Func,Timeout) -> void()
```

从进程中去掉调试函数。

```
resume(Name,Timeout) -> void()
```

恢复一个暂停的进程。

```
statistics(Name,Flag,Timeout) -> ok | {ok, Statistics}
```

允许或禁止收集统计数据。

```
suspend(Name,Timeout) -> void()
```

暂停一个进程。

```
trace(Name,Flag,Timeout) -> void()
```

在标准输出上打印系统消息。

F.60　timer模块

计时器模块。

```
apply_after(Time, Module, Function, Arguments) -> {ok, Tref} | {error, Reason}
```

在指定的时间只有执行特定函数。

```
apply_interval(Time, Module, Function, Arguments) -> {ok, TRef} | {error, Reason}
```

每间隔指定的时间执行特定函数。

```
cancel(TRef) -> {ok, cancel} | {error, Reason}
```

取消某个定时器。

```
hms(Hours, Minutes, Seconds) -> Milliseconds
```

将时分秒数据转换为毫秒。

```
hours(Hours) -> Milliseconds
```

将小时转换为毫秒。

498

```
kill_after(Time) -> {ok, TRef} | {error,Reason2}
```

指定时间之后终止某个进程。

```
minutes(Minutes) -> Milliseconds
```

将分钟转换为毫秒。

```
now_diff(T2, T1) -> Tdiff
```

比较两个时间之间的间隔。

```
seconds(Seconds) -> Milliseconds
```

将秒转换为毫秒。

```
send_after(Time, Message) -> {ok, TRef} | {error,Reason}
```

指定时间之后发送消息。

```
send_interval(Time, Message) -> {ok, TRef} | {error, Reason}
```

每间隔指定的时间发送消息。

```
sleep(Time) -> ok
```

让当前进程休眠指定的时间。

```
start() -> ok
```

启动一个全局的计时服务器。

```
tc(Module, Function, Arguments) -> {Time, Value}
```

测量某函数运行的确切时间。

F.61 win32reg模块

提供对Windows注册表的访问函数。

```
change_key(RegHandle, Key) -> ReturnValue
```

移动到注册表的指定项目。

```
change_key_create(RegHandle, Key) -> ReturnValue
```

移动到注册表的指定项目，如若不存在则创建它。

```
close(RegHandle)-> ReturnValue
```

关闭注册表。

```
current_key(RegHandle) -> ReturnValue
```

获取当前在注册表中的位置。

```
delete_key(RegHandle) -> ReturnValue
```

删除当前位置的注册表记录。

```
delete_value(RegHandle, Name) -> ReturnValue
```

删除当前位置指定名称的值。

```
expand(String) -> ExpandedString
```

展开一个包含环境变量的字符串。

```
format_error(ErrorId) -> ErrorString
```

获取一个错误的文字描述。

```
open(OpenModeList)-> ReturnValue
```

499 打开注册表。

```
set_value(RegHandle, Name, Value) -> ReturnValue
```

在当前位置设置注册表项。

```
sub_keys(RegHandle) -> ReturnValue
```

获取当前位置的子项目。

```
value(RegHandle, Name) -> ReturnValue
```

获取当前位置下某个名称的值。

```
values(RegHandle) -> ReturnValue
```

获取当前位置下所有的值。

F.62　zip模块

读取和创建ZIP压缩包的工具函数。

```
create(Name, FileList, Options) -> RetValue
```

创建一个ZIP压缩包。

```
extract(Archive, Options) -> RetValue
```

从ZIP压缩包中解开文件。

```
t(Archive)
```

打印ZIP压缩包中的所有文件名。

```
table(Archive, Options)
```

获取ZIP压缩包中所有文件的名称。

```
tt(Archive)
```

答应ZIP压缩包中每个文件的名称和信息。

```
zip_close(ZipHandle) -> ok | {error, einval}
```

关闭一个ZIP压缩包。

```
zip_get(FileName, ZipHandle) -> {ok, Result} | {error, Reason}
```

从打开的ZIP压缩包中提取文件。

```
zip_list_dir(ZipHandle) -> Result | {error, Reason}
```

从打开的ZIP压缩包中获取文件列表。

```
zip_open(Archive, Options) -> {ok, ZipHandle} | {error, Reason}
```

打开一个ZIP压缩包。

F.63　zlib模块

Zlib压缩接口。

```
adler32(Z, Binary) -> Checksum
```

计算校验和。

```
adler32(Z, PrevAdler, Binary) -> Checksum
```

计算校验和。

```
close(Z) -> ok
```
关闭流。

500

```
compress(Binary) -> Compressed
```
对数据用标准zlib函数进行压缩。
```
crc32(Z) -> CRC
```
获取当前的CRC。
```
crc32(Z, Binary) -> CRC
```
计算CRC。
```
crc32(Z, PrevCRC, Binary) -> CRC
```
计算CRC。
```
deflate(Z, Data) -> Compressed
```
对数据进行压缩。
```
deflate(Z, Data, Flush) ->
```
对数据进行压缩。
```
deflateEnd(Z) -> ok
```
结束压缩会话。
```
deflateInit(Z) -> ok
```
初始化压缩会话。
```
deflateInit(Z, Level) -> ok
```
初始化压缩会话。
```
deflateInit(Z, Level, Method, WindowBits, MemLevel, Strategy) -> ok
```
初始化压缩会话。
```
deflateParams(Z, Level, Strategy) -> ok
```
动态更新压缩参数。
```
deflateReset(Z) -> ok
```
重置压缩会话。
```
deflateSetDictionary(Z, Dictionary) -> Adler32
```
初始化压缩字典。
```
getBufSize(Z) -> Size
```
获取缓冲区大小。
```
gunzip(Bin) -> Decompressed
```
对带有gz头的数据进行解压。
```
gzip(Data) -> Compressed
```
压缩数据，带gz头。
```
infate(Z, Data) -> DeCompressed
```
对数据进行解压缩。
```
infateEnd(Z) -> ok
```
结束解压缩会话。
```
infateInit(Z) -> ok
```
初始化解压缩会话。
```
infateInit(Z, WindowBits) -> ok
```
初始化解压缩会话。

501

```
infateReset(Z) -> ok
```
重置解压缩会话。
```
infateSetDictionary(Z, Dictionary) -> ok
```
初始化解压缩字典。
```
open() -> Z
```
打开流，获得对于流的引用。
```
setBufSize(Z, Size) -> ok
```
设置缓冲区大小。
```
uncompress(Binary) -> Decompressed
```
使用标准zlib函数解压数据。
```
unzip(Binary) -> Decompressed
```
解压不含zlib头的数据。
```
zip(Binary) -> Compressed
```
502
压缩数据，不带zlib头。

索　引

索引中页码为英文原书页码，与书中边栏页码一致。注意，索引中页码后面“f”和“n”含义，“f”代表词条是在图中出现的；“n”代表词条是在脚注中出现的。

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

P

Q

R

S

T

U

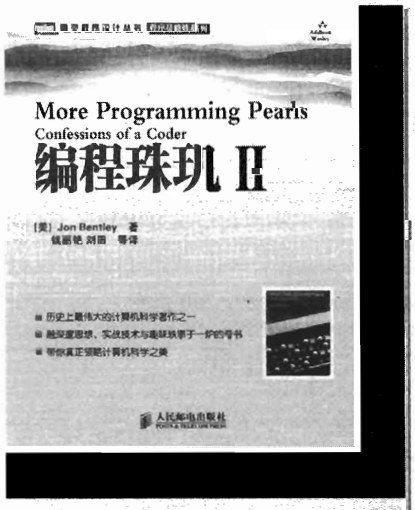
V

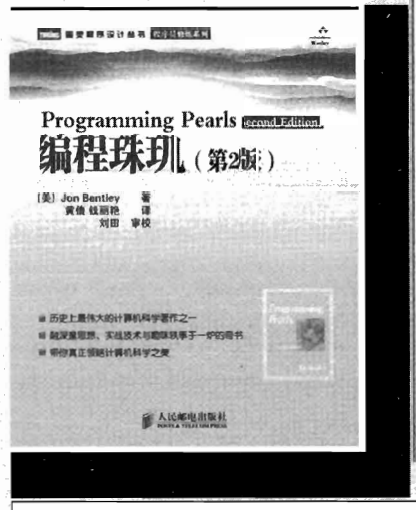
W

X

Y

一部20余年畅销不衰的不朽经典





本书赞誉

“《编程珠玑》第1版是对我职业生涯早期影响最大的书之一，其中的许多真知灼见多年之后仍然使我受益匪浅。Jon在第2版中对素材进行了大量更新，许多新内容让我耳目一新。”

——Steve McConnell，软件工程大师，IEEE Software前主编，《代码大全》作者

“对每一位遇到的程序员，我都会毫不迟疑地建议他阅读并不断重读这部经典之作。”

——Slashdot

内容简介

多年以来，当程序员们推选出最心爱的计算机图书时，《编程珠玑》总是位于前列。正如自然界里珍珠出自细沙对牡蛎的磨砺，计算机科学大师Jon Bentley以其独有的洞察力和创造力，从磨砺程序员的实际问题中凝结出一篇篇不朽的编程“珠玑”，成为世界计算机界名刊《ACM通讯》历史上最受欢迎的专栏，最终结集为两部不朽的计算机科学经典名著，影响和激励着一代又一代程序员和计算机科学工作者。

作者简介

Jon Bentley世界著名计算机科学家，被誉为影响算法发展的十位大师之一。他先后任职于卡内基－梅隆大学（1976~1982）、贝尔实验室（1982~2001）和Avaya实验室（2001年至今）。在卡内基－梅隆大学担任教授期间，他培养了包括Tcl语言设计者John Ousterhout、Java语言设计者James Gosling、《算法导论》作者之一Charles Leiserson在内的许多计算机科学大家。2004年荣获Dr. Dobb's程序设计卓越奖。

- 历史上最伟大的计算机科学著作之一
- 融深邃思想、实战技术与趣味轶事于一炉的奇书
- 带你真正领略计算机科学之美