栏中的地址改变为服务器选择确定的地址。

- 在一次地址转发整个过程中，只产生一次网络请求；而一次地址重写一般会产生两次请求。
- 地址转发一般发生在同一站点项目内；而地址重写没有该限制。
- 地址转发到的页面可以不用全路径名表示，而地址重写到的页面必须使用完整的路径名表示。
- 地址转发过程中，可以将客户端请求的 request 范围内属性传递给新的页面，但地址重写不可以。
- 地址转发的速度较地址重定向快。

6.2.2　Rewrite 规则

Rewrite 规则是学习和使用 Nginx 服务器 Rewrite 功能的基础，可以借助 PCRE 实现 URI 的重写，并且它还支持 Nginx 预设变量。Rewrite 规则的核心就是 PCRE。

正则表达式（Regular Expression，缩写为 regex、regexp 或 RE），是用于描述或者匹配一系列符合某个句法规则的字符串的一个字符串。大家应该都使用过 Windows/Dos 平台下用于文件查找的通配符“*”和“?”，如果要查找某个目录下所有的 Word 文档，就使用“*.doc”进行搜索，“*”会被解释成任意的字符串。和通配符类似，在很多文本编辑器或其他工具里，正则表达式通常可以被用来检索和替换符合某个句法模式的文本内容。许多程序设计语言都支持利用正则表达式进行字符串操作。PCRE 就是在 Perl 中内建的功能强大的正则表达式引擎，可以被许多工具使用的库。正则表达式有多种不同的风格，并且不同版本在对句法规则的匹配处理上也有一定的差异。

在 Nginx 服务器中，使用 ngx_http_rewrite_module 模块支持 URL 重写功能。该模块是 Nginx 服务器的标准 HTTP 模块。

由于正则表达式的知识超出了本书的范围，我们不在正文中对其详细阐述。在“附录 A”中，笔者为大家整理了 PCRE 库支持的正则表达式元字符表，并对这些元字符在正则表达式上下文中的行为进行了说明，同时也列举了大量的使用实例供大家参考。

在这里，建议大家学习正则表达式时，一定要将思维从“字符串”的概念转变为“字符”的概念，将“词句”的概念转变为“字”的概念，多结合实例进行操作，这样才能准确理解和运用正则表达式。另外，网络上有不少 PCRE 正则表达式的测试工具，大家可以搜索下载，这些工具对我们在学习和使用正则表达式的过程中很有帮助。

在 Nginx 配置中，有关 Rewrite 的配置指令不多，但是它们已经能够提供比较完整的功能了。

6.2.3　if 指令

该指令用来支持条件判断，并根据条件判断结果选择不同的 Nginx 配置，可以在 server 块或 location 块中配置该指令，其语法结构为：

```
if ( condition ) { ... }
```

其中，花括号代表一个作用域，形成一个 if 配置块，是条件为真时的 Nginx 配置。*condition* 为判断条件（true/false），它可以支持以下几种设置方法：

- 变量名。如果变量的值为空字符串或者以“0”开头的任意字符串，if 指令认为条件为 false，其他情况认为条件为 true。比如：

```
if ($slow) {
    … #Nginx 配置
}
```

- 使用“=”（等于）和“!=”（不等于）比较变量和字符串是否相等，相等时 if 指令认为条件为 true，反之为 false。

```
if ($request_method = POST) {
    return 405;
}
```

注意

这里的字符串不需要加引号。

- 使用正则表达式对变量进行匹配，匹配成功时 if 指令认为条件为 true，否则为 false。变量与正则表达式之间用“~”、“~*”、“!~”或“!~*”连接，“~”表示匹配过程中对大小写敏感，“~*”表示匹配过程中对大小写不敏感；使用“!~”和“!~*”，匹配失败时 if 指令认为条件为 true，否则为 false。在正则表达式中，可以使用小括号对变量值进行截取，在花括号中使用$1…$9 引用截取的值。比如：

```
if ($http_user_agent ~ MSIE) {
                        # $http_user_agent 的值中是否含有 MSIE 字符串，如果包含，为 true
    …
}
if ($http_cookie ~* "id=([^;]+)(?:;|$)") {
                        #Nginx 配置，可以使用$1 和$2 获取截取到的值。如：
                        # set $id $1; 将截取到的 id 赋值给$id 变量以备后用。
    …
}
```

注意

整个正则表达式字符串一般不需要加引号，但如果含有右花括号“}”或者分号“;”字符时，必须要给整个正则表达式添加引号。

- 判断请求的文件是否存在使用“-f”和“!-f”。当使用“-f”时，如果请求的文件存在，if 指令认为条件为 true，如果请求的文件不存在为 false；使用“!-f”时，如果请求的文件不存在但该文件所在的目录存在，if 指令认为条件为 true，如果该文件和它所在的目录都不存在，则为 false，如果请求的文件存在，也为 false。使用方法如下：

```
if (-f $request_filename) {
                                    #判断请求的文件是否存在
    …
}
if (!-f $request_filename) {
                                    #判断请求的文件是否不存在
    …
}
```

- 判断请求的目录是否存在使用“-d”和“!-d”。当使用“-d”时，如果请求的目录存在，if 指令认为条件为 true，如果请求的目录不存在，则为 false；当使用“!-d”时，如果请求的目录不存在但该目录的上级目录存在，if 指令认为条件为 true，如果该目录和它的上级目录都不存在，则为 false，如果请求的目录存在，也为 false。使用方法见“-f”和“!-f”的使用。
- 判断请求的目录或者文件是否存在使用“-e”和“!-e”。当使用“-e”时，如果请求的目录或者文件存在时，if 指令认为条件为 true，否则为 false。当使用“!-e”时，如果请求的文件和该文件所在路径上的目录都不存在，为 true，否则为 false。使用方法见“-f”和“!-f”的使用。
- 判断请求的文件是否可执行使用“-x”和“!-x”。当使用“-x”时，如果请求的文件可执行，if 指令认为条件为 true，否则为 false；当使用“!-x”时，如果请求的文件不可执行，为 true，否则为 false。使用方法见“-f”和“!-f”的使用。

6.2.4　break 指令

该指令用于中断当前相同作用域中的其他 Nginx 配置。与该指令处于同一作用域的 Nginx 配置中，位于它前面的指令配置生效，位于后面的指令配置无效。Nginx 服务器在根据配置处理请求的过程中遇到该指令时，回到上一层作用域继续向下读取配置。该指令可以在 server 块和 location 块以及 if 块中使用，其语法结构为：

```
break;
```

我们通过一个例子加深理解：

```
location / {
    if ($slow) {
        set $id $1                                    #处于 break 指令之前，配置有效
        break;
        limit_rate 10k;                               #处于 break 指令之后，配置无效
    }
    …                          #其他 Nginx 配置，处于 break 指令所在作用域的上一层作用域，配置有效
}
```

1. return 指令

该指令用于完成对请求的处理，直接向客户端返回响应状态代码。处于该指令后的所有 Nginx 配置都是无效的。该指令可以在 server 块和 location 块以及 if 块中使用，其语法结构有以下几种：

```
return [ text ]
return code URL;
return URL;
```

- *code*，为返回给客户端的 HTTP 状态代码。可以返回的状态代码为 0 ~ 999 的任意 HTTP 状态代码。非标准的 444 代码可以强制关闭服务器与客户端的连接而不返回任何响应信息给客户端。
- *text*，为返回给客户端的响应体内容，支持变量的使用。
- *URL*，为返回给客户端的 URL 地址。

从 Nginx 0.8.42 开始，当 *code* 使用 301（表示被请求资源永久移动到新的位置）、302（表示请求

的资源现在临时从不同的 URL 响应，要求使用 GET 方式请求）、303（表示对应当前请求的响应可以在另一个 URL 上找到，并且客户端应当采用 GET 方式访问那个资源）和 307（请求的资源临时从不同的 URL 响应）代码时，可以使用结构 2 将新的 *URL* 返回给客户端；当 *code* 使用除上面提到的其他代码时，可以使用结构 1 指定 *text* 向客户端发送指定的响应体内容。

当返回状态代码为 302 或 307 时，可以使用结构 3 对 *URL* 进行配置。返回的 URL 中应该包含“http://”、“https://”或者直接使用“$scheme”变量（Request Scheme，代表传输协议，Nginx 内置变量）指定。

注意

在 Nginx 0.7.51 之前的版本中，只支持返回 204、400、402~406、408、410、411、413、416 和 500~504 等状态代码。

6.2.5 rewrite 指令

该指令通过正则表达式的使用来改变 URI。可以同时存在一个或者多个指令，按照顺序依次对 URL 进行匹配和处理。

提示

URI 与 URL 的区别和联系。

URI（Universal Resource Identifier，通用资源标识符），用于对网络中的各种资源进行标识，由存放资源的主机名、片段标志符和相对 URI 三部分组成。存放资源的主机名一般由传输协议（Scheme）、主机和资源路径三部分组成；片段标志符指向资源内容的具体元素；相对 URI 表示资源在主机上的相对路径。一般格式为：Scheme:[//][[用户名[:密码]@]主机名[:端口号]][/资源路径]。

URL（Uniform Resource Location，统一资源定位符），是用于在 Internet 中描述资源的字符串，是 URI 的子集，主要包括传输协议（Scheme）、主机（IP、端口号或者域名）和资源具体地址（目录和文件名）等三部分。一般格式为：Scheme://主机名[:端口号][/资源路径]。

该指令可以在 server 块或者 location 块中配置，其语法结构为：

```
rewrite regex replacement [flag];
```

- *regex*，用于匹配 URI 的正则表达式。使用括号“( )”标记要截取的内容。

注意

rewrite 接收到的 URI 不包含 host 地址。因此，regex 不可能匹配到 URI 的 host 地址。我们看下面这个例子：

```
rewrite  myweb.com  http://newweb.com/permanent;
```

现在我们希望上面的 rewrite 指令重写 http://myweb.com/source 是办不到的，因为 rewrite 指令接收到的 URI 是“/source”，不包含“myweb.com”。

另外，请求 URL 中的请求指令也是不包含在 rewrtie 指令接收到的 URI 内容中的。比如：

http://myweb.com/source?agr1=value1&agr2=value2

rewrtie 指令接收到的 URI 为“/source”，不包含“?agr1=value1&agr2=value2”。

- *replacement*，匹配成功后用于替换 URI 中被截取内容的字符串。默认情况下，如果该字符

串是由“http://”或者“https://”开头的，则不会继续向下对 URI 进行其他处理，而直接将重写后的 URI 返回给客户端。

提示

刚才学习 regex 变量时我们提到，rewrite 模块接收到的 URI 不包含请求 URL 中的请求指令，但是如果我们希望将这些指令传给重写后的 URI，该怎么做呢？我们可以使用 Nginx 全局变量 $request_uri，比如：

```
rewrite    myweb.com    http://example.com$request_uri? permanent;
```

注意

在$request_uri 变量后要添加问号“？”。*replacement* 变量中支持 Nginx 全局变量的使用，常用的还有$uri 和$args 等。

- *flag*，用来设置 rewrtie 对 URI 的处理行为，可以为以下标志中的一个：
  - last，终止继续在本 location 块中处理接收到的 URI，并将此处重写的 URI 作为一个新的 URI，使用各 location 块进行处理。该标志将重写后的 URI 重新在 server 块中执行，为重写后的 URI 提供了转入到其他 location 块的机会。我们通过一个例子来加深理解：

```
location / {
rewrite  ^(/myweb/.*)/media/(.*)\..*$  $1/mp3/$2.mp3  last;
rewrite  ^(/myweb/.*)/audio/(.*)\.*$  $1/mp3/$2.ra  last;
 }
```

如果某 URI 在第 2 行被匹配成功并处理，Nginx 服务器不会继续使用第 3 行的配置匹配和处理新的 URI，而是让所有的 location 块重新匹配和处理新的 URI。

  - break，将此处重写的 URI 作为一个新的 URI，在本块中继续进行处理。该标志将重写后的地址在当前的 location 块中执行，不会将新的 URI 转向到其他 location 块。看下面的例子：

```
location /myweb/ {
  rewrite  ^(/myweb/.*)/media/(.*)\..*$  $1/mp3/$2.mp3  break;
  rewrite  ^(/myweb/.*)/audio/(.*)\.*$  $1/mp3/$2.ra  break;
}
```

如果某 URI 在第 2 行被匹配成功并处理，Nginx 服务器将新的 URI 继续在该 location 块中使用第 3 行进行匹配和处理。新的 URI 始终是在同一个 loaction 块中。

  - redirect，将重写后的 URI 返回给客户端，状态代码为 302，指明是临时重定向 URI，主要用在 *replacement* 变量不是以“http://”或者“https://”开头的情况下。
  - permanent，将重写后的 URI 返回给客户端，状态代码为 301，指明是永久重定向 URI。

在使用 flag 指令时，一定要注意各个标志之间的配合。我们再来回顾刚才学习 break 标志时的例子，对比 last 标志里的例子，如果我们将第二个例子中的 break 标志换成 last 标志，会发生什么情况呢？

细心的读者可能已经发现，在第二个例子中，location 块的 *uri* 指令是“/myweb/”，而重写后的 URI 仍然是包含“/myweb/”的，如果使用 last 标志，重写后的 URI 还可能会被该 location 块匹配到，这样就形成了无限循环。Nginx 服务器遇到这样的情况，会尝试 10 次循环之后返回错误状态代码 500。

6.2.6　rewrite_log 指令

该指令配置是否开启 URL 重写日志的输出功能，其语法结构为：

```
rewrite_log on | off
```

默认设置为 off。如果配置为开启（on），URL 重写的相关日志将以 notice 级别输出到 error_log 指令配置的日志文件中。

6.2.7　set 指令

该指令用于设置一个新的变量，其语法结构为：

```
set variable value
```

- *variable*，为变量的名称。注意要用符号“$”作为变量的第一个字符，且变量不能与 Nginx 服务器预设的全局变量同名。
- *value*，为变量的值，可以是字符串、其他变量或变量的组合等。

6.2.8　uninitialized_variable_warn 指令

该指令用于配置使用未初始化的变量时，是否记录警告日志，其语法结构为：

```
uninitialized_variable_warn on | off
```

默认设置为开启（on）状态。

6.2.9　Rewrite 常用全局变量

在表 6.1 中，笔者列出了一些在 Rewrite 功能配置过程中可能会使用到的 Nginx 全局变量，以备大家查阅。

表 6.1　Rewrite 常用全局变量举例

变量	说明
$args	变量中存放了请求 URL 中的请求指令。比如 http://www.myweb.name/server/source?arg1=value1&arg2=value2 中的“arg1=value1&arg2=value2”
$content_length	变量中存放了请求头中的 Content-length 字段
$content_type	变量中存放了请求头中的 Content-type 字段
$document_root	变量中存放了针对当前请求的根路径
$document_uri	变量中存放了请求中的当前 URI，并且不包括请求指令，比如 http://www.myweb.name/server/source?arg1=value1&arg2=value2 中的“/server/source”
$host	变量中存放了请求 URL 中的主机部分字段，比如 http://www.myweb.name/server 中的“www.myweb.name”。如果请求中的主机部分字段不可用或者为空，则存放 Nginx 配置中该 server 块中 server_name 指令的配置值
$http_user_agent	变量中存放客户端的代理信息
$http_cookie	变量中存放客户端的 cookie 信息
$limit_rate	变量中存放 Nginx 服务器对网络连接速率的限制，也就是 Nginx 配置中 limit_rate 指令的配置值

续表

变量	说明
$remote_addr	变量中存放了客户端的地址
$remote_port	变量中存放了客户端与服务器建立连接的端口号
$remote_user	变量中存放了客户端的用户名
$request_body_file	变量中存放了发给后端服务器的本地文件资源的名称
$request_method	变量中存放了客户端的请求方式，如“GET”、“POST”等
$request_filename	变量中存放了当前请求的资源文件的路径名
$request_uri	变量中存放了当前请求的 URI，并且带请求指令
$query_string	与变量$args 含义相同
$scheme	变量中存放了客户端请求使用的协议，比如 “http”、“https” 和 “ftp” 等
$server_protocol	变量中存放了客户端请求协议的版本，比如 “HTTP/1.0”、“HTTP/1.1” 等
$server_addr	变量中存放了服务器的地址
$server_name	变量中存放了客户端请求到达的服务器的名称
$server_port	变量中存放了客户端请求到达的服务器的端口号
$uri	与变量$document_uri 含义相同

6.3　Rewrite 的使用

ngx_http_rewrite_module 是 Nginx 服务器的重要模块之一，它一方面实现了 URL 的重写功能，另一方面为 Nginx 服务器提供反向代理服务提供了支持，同时，我们可以利用 URL 重写功能完成一些其他工作，达到特殊的效果。

前面我们已经学习了 Rewrite 功能的基本配置和使用方法，在这一节中，笔者梳理了一些 Rewrite 功能的其他一些用法，供大家学习和在实际应用中参考。

6.3.1　域名跳转

通过 Rewrite 功能可以实现一级域名跳转，也可以实现多级域名跳转。在 server 块中配置 Rewrite 功能即可。笔者准备了几个例子供大家参考：

```
# 例 1
...
server
{
  listen  80;
  server_name  jump.myweb.name;
  rewrite  ^/  http://www.myweb.info/;                    #域名跳转
  ...
}
...
```

```
# 例 2
...
server
{
  listen  80;
  server_name  jump.myweb.name  jump.myweb.info;
   if ($host  ~  myweb\.info)                    #注意正则表达式中对点号 "." 要用 "\" 进行转义
  {
       rewrite  ^(.*)  http://jump.myweb.name$1  permanent;          #多域名跳转
  }
  ...
}
...
# 例 3
...
server
{
  listen  80;
  server_name  jump1.myweb.name  jump2.myweb.name;
  if ($http_host  ~*  ^(.*)\.myweb\.name$)
  {
    rewrite  ^(.*)  http://jump.myweb.name$1;                          #三级域名跳转
    break;
  }
  ···  #其他配置
}
···  #其他配置
```

在上面的例子中展示了通过 Rewrite 功能完成域名跳转的相关配置。

在例 1 中，客户端访问 http://jump.myweb.name 时，URL 将被 Nginx 服务器重写为 http://jump.myweb.info，客户得到的数据其实是由 http://jump.myweb.info 响应的。在例 2 中，客户端访问 http://jump.myweb.info/reqsource 时，URL 将被 Nginx 服务器重写为 http://jump.myweb.name/reqsource，客户端得到的数据实际上是由 http://jump.myweb.name 响应的。在例 3 中，客户端访问 http://jump1.myweb.name/reqsource 或者 http:// jump2.myweb.name/reqsource，URL 都将被 Nginx 服务器重写为 http:// jump.myweb.name/reqsource，实现了三级域名的跳转。

6.3.2　域名镜像

镜像网站是指将一个完全相同的网站分别放置到几个服务器上，并分别使用独立的 URL，其中一个服务器上的网站叫主站，其他的为镜像网站。镜像网站和主站没有太大区别，或者可算是主站的后备。镜像网站可以保存网页信息、历史性数据，以防止丢失。可以通过镜像网站提高网站在不同地区的响应速度。镜像网站可以平衡网站的流量负载，可以解决网络带宽限制、封锁等。

使用 Nginx 服务器的 Rewrite 功能可以轻松地实现域名镜像的跳转。其实原理很简单，就是在 server 块中配置 Rewrite 功能，将不同的镜像 URL 重写到指定的 URL 就可以了。以下是一个供大家参考的

配置示例：

```
...
server
{
  ...
  listen  80;
  server_name  mirror1.myweb.name;
  rewrite  ^(.*)  http://jump1.myweb.name$1  last;
}
server
{
  ...
  listen  81;
  server_name  mirror2.myweb.name;
  rewrite  ^(.*)  http://jump2.myweb.name$1  last;
}
...
```

如果我们不想将整个网站做镜像，只想为某一个子目录下的资源做镜像，我们可以在 location 块中配置 Rewrite 功能，原理和上面是一样的。比如：

```
server
{
  listen  80;
  server_name  jump.myweb.name;
  location  ^~  /source1
  {
    ...
    rewrite  ^/source1(.*)  http://jump.myweb.name/websrc2$1  last;
    break;
  }
  location  ^~  /source2
  {
    ...
    rewrite  ^/source2(.*)  http://jump.myweb.name/websrc2$1  last;
    break;
  }
  ...
}
...  #其他配置
```

6.3.3　独立域名

当一个网站包含多个板块时，可以为其中的某些板块设置独立域名。其原理和设置某个子目录镜像的原理是相同的。比如：

```
server
{
```

```
  ...
  listen  80;
  server_name  bbs.myweb.name;
  rewrite  ^(.*)  http://www.myweb.name/bbs$1  last;
  break;
}
server
{
  ...
  listen  81;
  server_name  home.myweb.name;
  rewrite  ^(.*)  http://www.myweb.name/home$1  last;
  break;
}
...
```

6.3.4 目录自动添加“/”

如果网站设定了默认资源文件，那么当客户端使用 URL 访问时可以不加具体的资源文件名称。比如，在访问 www.myweb.name 站点时，应该在浏览器地址栏中输入“http://www.myweb.name/index.htm”这样的 URL，如果我们设置了 www.myweb.name 站点的首页为 index.htm，那么直接在地址栏中输入“http://www.myweb.name”即可访问成功，“/index.htm”可以省略不写，现在大家基本上也都习惯了这种访问方式。

但是如果请求的资源文件在二级目录下，这样的习惯可能会造成无法正常访问资源。比如，在访问 http://www.myweb.name/bbs/index.htm 时，如果将 URL 省略为“http://www.myweb.name/bbs/”可以进行正常访问，但是如果将 URL 写为“http://www.myweb.name/bbs”，将末尾的斜杠“/”也省略了，访问就会出问题。也就是说，Nginx 服务器访问二级目录时不加斜杠“/”无法访问。但是我们不可能要求客户端始终按照我们的要求在末尾添加斜杠“/”，那么如何来解决这个问题呢？我们可以使用 Rewrite 功能为末尾没有斜杠“/”的 URL 自动添加一个斜杠“/”：

```
server
{
  ...
  listen  81;
  server_name  www.myweb.name;
  location  ^~  /bbs
  {
    ...
    if (-d $request_filename)
    {
          rewrite ^/(.*)([^/])$  http://$host/$1$2/  permanent;
    }
  }
}
```

在该示例中，我们使用 if 指令判断请求的“/bbs”是目录以后，匹配接收到的 URI 串，并将各部

分的值截取出来重新组装，并在末尾添加斜杠“/”。注意应该使用 permanent 标志指明是永久重定向该 URI。

6.3.5　目录合并

搜索引擎优化（Search Engine Optimization，SEO）是一种利用搜索引擎的搜索规则来提高目的网站在有关搜索引擎内排名的方式。我们在创建自己的站点时，可以通过一些措施有效提高搜索引擎优化的程度，比如为网页添加包含有有效关键字的标题，在正文中多使用有效关键字，制作网站导航时注意通用规则，尽量避免大量的动态网页，等等。目录合并也是增强 SEO 的一个方法，它主要将多级目录下的资源文件请求转化为看上去是对目录级数很少的资源的访问。我们来看一个例子，比如在笔者的网站中有这样一个网页，它的路径如下所示：

```
[root]/server/12/34/56/78/9.htm
```

网页 9.htm 存在于第 5 级目录下，如果要访问这个资源文件，客户端的 URL 要写成 http://www.myweb.name/server/12/34/56/78/9.htm，这非常不利于搜索引擎的搜索，同时也给客户端的输入带来负担。那么，我们有没有办法让 URL 的目录级数“看上去”少一些呢？通过 Rewrite 功能，我们很容易就可以办到。比如我们进行以下的配置：

```
server
{
  ...
  listen  80;
  server_name  www.myweb.name;
  location  ^~  /server
  {
    ...
    rewrite  ^/server-([0-9]+)-([0-9]+)-([0-9]+)-([0-9]+)-([0-9]+)\.htm$
/server/$1/$2/$3/$4/$5.html  last;
    break;
  }
}
...
```

那么，客户端只要输入“http://www.myweb.name/server-12-34-56-78-9.htm”即可访问到 9.htm 这个资源文件了。这个 URL 是不是“看上去”简单多了呢？这其实是充分利用了 rewrite 指令支持正则表达式的特性。

6.3.6　防盗链

盗链是一种损害原有网站合法利益，给原网站所在服务器造成额外负担的非法行为。要采取防盗链的措施，首先需要了解盗链的实现原理。

客户端向服务器请求资源时，为了减少网络带宽，提高响应时间，服务器一般不会一次将所有资源完整地传回给客户端。比如在请求一个网页时，首先会传回该网页的文本内容，当客户端浏览器在解析文本的过程中发现有图片存在时，会再次向服务器发起对该图片资源的请求，服务器将存储的图片资源再发送给客户端。在这个过程中，如果该服务器上只包含了网页的文本内容，并没有存储相关

的图片资源，而是将图片资源链接到其他站点的服务器上去了，这就形成了盗链行为。在这种情况下，客户端请求的图片资源实际上是来自于其他的服务器。很明显，盗链行为是一种对其他服务器不公平的行为。我们在搭建自己的站点时应当有意识地采取防盗链措施。

要实现防盗链，需要了解 HTTP 协议中的请求头部的 Referer 头域和采用 URL 的格式表示访问当前网页或者文件的源地址。通过该头域的值，我们可以检测到访问目标资源的源地址。这样，如果我们检测到 Referer 头域中的值并不是自己站点内的 URL，就采取阻止措施，实现防盗链。但是，需要提醒大家的是，由于 Referer 头域中的值是可以被更改的，因此该方法不能够完全阻止所有的盗链行为。

知道了盗链行为的原理和防盗链的实现原理，我们就可以利用 Nginx 服务器的 Rewrite 功能实现防盗链了。

Nginx 配置中有一个指令 valid_referers，用来获取 Referer 头域中的值，并且根据该值的情况给 Nginx 全局变量$invalid_referer 赋值。如果 Referer 头域中没有符合 valid_referers 指令配置的值，$invalid_referer 变量将会被赋值为 1。valid_referers 指令的语法结构为：

```
valid_referers none | blocked | server_names | string ...;
```

- *none*，检测 Referer 头域不存在的情况。
- *blocked*，检测 Referer 头域的值被防火墙或者代理服务器删除或伪装的情况。这种情况下，该头域的值不以“http://”或者“https://”开头。
- *server_names*，设置一个或多个 URL，检测 Referer 头域的值是否是这些 URL 中的某个。从 Nginx 0.5.33 以后支持使用通配符“*”。

有了 valid_referers 指令和$invalid_referer 变量，就能通过 Rewrite 功能来实现防盗链。有两种实现方案：一种是根据请求资源的资源类型，一种是根据请求目录。

根据文件类型实现防盗链的一个配置实例如下：

```
server
{
  ...
  listen  80;
  server_name  www.myweb.name;
  location ~* ^.+\.(gif|jpg|png|swf|flv|rar|zip)$
  {
    ...
    valid_referers  none  blocked  server_names  *. myweb.name;
    if ($invalid_referer)
    {
      rewrite  ^/  http://www.myweb.com/images/forbidden.png;
    }
  }
}
...
```

在配置中，当有网络连接对以 gif、jpg、png 为后缀的图片资源、以 swf、flv 为后缀的媒体资源、以 rar、zip 为后缀的压缩存档资源发起请求时，如果检测到 Referer 头域中没有符合 valid_referers 指令

配置的值，就将客户端请求的 URL 重写为 http://www.myweb.com/images/forbidden.png，这防止了非法盗链行为。当然我们也可以不重写 URL，直接返回 HTTP 错误状态，如 403 状态。

根据请求目录实现防盗链的一个配置实例如下：

```
server
{
  ...
  listen  80;
  server_name  www.myweb.name;
  location /file/
  {
    ...
    root /server/file/;
    valid_referers  none  blocked  server_names  *. myweb.name;
    if ($invalid_referer)
    {
      rewrite  ^/  http://www.myweb.com/images/forbidden.png;
    }
  }
}
...
```

其原理其实和根据文件类型实现防盗链的原理是一样的，只是在 location 块的 *uri* 变量上做了改变，对于请求服务器上[root]/server/file/目录下的资源采取了防盗链措施。

6.4　本章小结

本章重点介绍了 Nginx 服务器的 Rewrite 功能。Rewrite 压缩功能依赖于 ngx_http_upstream_module 模块，是 Nginx 服务器提供重定向服务的重要功能之一。然后还介绍了模块支持的 Nginx 配置指令，并对常见的应用需求进行了分析，提供了配置实例。相信大家通过本章的学习，就能够比较熟练地使用 Nginx 服务器的 Rewrite 功能进行服务器的重定向配置。

第7章

Nginx 服务器的代理服务

在本章中，我们将学习 Nginx 服务器的重要功能——代理服务。我们将要在本书后面学习的其他 Nginx 服务器功能有许多是从代理服务衍生出来的。几乎在所有 Nginx 服务器的应用场合中，都会涉及本章的内容。

在具体学习 Nginx 服务器的 HTTP 代理和反向代理功能之前，我们首先应该明确代理服务与反向代理服务的概念和区别。

在本章中将要学习以下知识：

- 正向代理与反向代理的基本概念
- Nginx 正向代理服务的配置指令
- Nginx 反向代理服务的配置指令
- Nginx 反向代理服务器的应用——负载均衡

7.1 正向代理与反向代理的概念

代理（Proxy）服务，通常也称为正向代理服务，可以使用图 7.1 示意，其中箭头的方向指示访问的方向。如果把局域网外 Internet 想象成一个巨大的资源库，那么资源就分布在 Internet 的各个站点上，局域网内的客户端要访问这个库里的资源必须统一通过代理服务器才能对各个站点进行访问。

局域网内的机器借助代理服务访问局域网外的网站，这主要是为了增强局域网内部网络的安全性，使得网外的威胁因素不容易影响到网内，这里代理服务器起到了一部分防火墙的功能。同时，利用代理服务器也可以对局域网对外网的访问进行必要的监控和管理。正向代理服务器不支持外部对内部网络的访问请求，即图 7.2 中的箭头方向不能反过来。

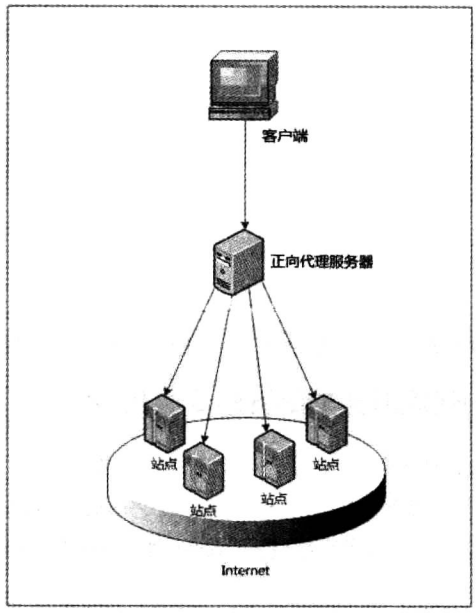

图 7.1　正向代理服务器示意图

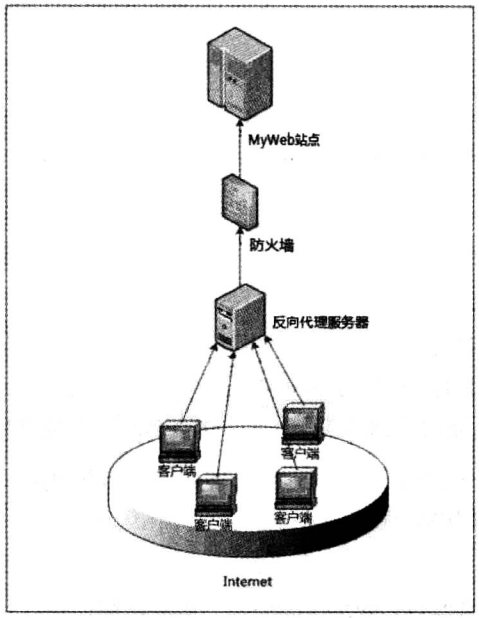

图 7.2　反向代理服务器示意图

与正向代理服务相反，如果局域网向 Internet 提供资源，让 Internet 上的其他用户可以访问局域网内的资源，也可以设置使用一个代理服务器，它提供的服务就叫做反向代理（Reverse Proxy）服务。可以看到，反向代理服务与代理服务在功能逻辑上刚好是相反的。箭头方向代表访问的方向。

正向代理服务器与反向代理服务器的概念很简单，归纳起来就是，正向代理服务器用来让局域网客户机接入外网以访问外网资源，反向代理服务器用来让外网的客户端接入局域网中的站点以访问站

点中的资源。理解这两个概念的关键是要明白我们当前的角色和目的是什么，在正向代理服务器中，我们的角色是客户端，目的是要访问外网的资源；在反向代理服务器中，我们的角色是站点，目的是把站点的资源发布出去让其他客户端能够访问。

我们知道了这两个概念，就可以学习如何让 Nginx 服务器来提供代理和反向代理服务了。

7.2　Nginx 服务器的正向代理服务

我们首先来学习 Nginx 服务器的正向代理服务的相关配置指令和正向代理服务的使用。

7.2.1　Nginx 服务器正向代理服务的配置的 3 个指令

在实际应用中，使用 Nginx 服务器代理服务功能的情况相对少一些，Nginx 代理服务本身也相对简单，涉及的主要指令不多，我们接下来分别介绍。这些指令原则上可以出现在 Nginx 配置文件的 http 块、server 块或者 location 块中，但一般是在搭建的 Nginx 服务器中单独配置一个 server 块用来设置代理服务。

1. resolver 指令

该指令用于指定 DNS 服务器的 IP 地址。DNS 服务器的主要工作是进行域名解析，将域名映射为对应的 IP 地址。该指令的语法结构为：

```
resolver address ... [valid=time];
```

- *address*，DNS 服务器的 IP 地址。如果不指定端口号，默认使用 53 端口。
- *time*，设置数据包在网络中的有效时间。出现该指令的主要原因是，在访问站点时，有很多情况使得数据包在一定时间内不能被传递到目的地，但是又不能让该数据包无期限地存在，于是就需要设定一段时间，当数据包在这段时间内没有到达目的地，就会被丢弃，然后发送者会接收到一个消息，并决定是否要重发该数据包。

使用该指令的一个例子如下：

```
resolver 127.0.0.1 [::1]:5353 valid=30s;
```

在实际应用中，一般不需要设置这么复杂，只要将 DNS 服务器的 IP 地址设置给该指令即可。

从 Nginx 1.1.7 版本开始，该指令支持设置多个 IP 地址，从 Nginx 1.3.1 开发版本和 Nginx 1.2.2 稳定版本开始，该指令支持设置 IPV6 地址。

2. resolver_timeout 指令

该指令用于设置 DNS 服务器域名解析超时时间，语法结构为：

```
resolver_timeout time;
```

3. proxy_pass 指令

该指令用于设置代理服务器的协议和地址，它不仅仅用于 Nginx 服务器的代理服务，更主要的是应用于反向代理服务，我们马上就会谈及。该指令的语法结构为：

```
proxy_pass URL;
```

其中，*URL* 即为设置的代理服务器协议和地址。

在代理服务配置中，该指令的设置相对固定，因此在这里就不介绍其他细节了，具体内容在学习 Nginx 服务器的反向代理服务时再重点阐述。在代理服务配置中，该指令配置为：

```
proxy_pass http://$http_host$request_uri;
```

其中，代理服务器协议设置为 HTTP，$http_host 和$request_uri 两个变量是 Nginx 配置支持的用于自动获取主机和 URI 的变量。配置代理服务时，一般不要改变该指令的配置。

7.2.2　Nginx 服务器正向代理服务的使用

这里，给大家准备了一个 Nginx 服务器基本代理服务的配置实例片段，帮助大家在实际应用中更容易地实施部署。

```
...
server
{
  resolver 8.8.8.8;
  listen 82;
   location /
   {
       proxy_pass http://$http_host$request_uri;
    }
}
...
```

实例片段很简单，设置 DNS 服务器地址为 8.8.8.8，使用默认的 53 端口作为 DNS 服务器的服务端口，代理服务的监听端口设置为 82 端口，Nginx 服务器接收到的所有请求都由第 5 行的 location 块进行过滤处理。

Nginx 服务器代理服务使用的场合不多，从上一节的配置指令来看，功能也相对简单。在使用过程中，有一些需要注意的事项笔者在这里说明一下。

首先，我们在上一节中提到过，设置 Nginx 服务器的代理服务器，一般是配置到一个 server 块中。注意，在该 server 块中，不要出现 server_name 指令，即不要设置虚拟主机的名称或 IP。而 resolver 指令是必需的，如果没有该指令，Nginx 服务器无法处理接收到的域名。

其次，Nginx 服务器的代理服务器不支持正向代理 HTTPS 站点。

以上学习的三个指令是设置代理服务的主要指令，在实际应用过程中还可以结合 Nginx 服务器的缓存功能配置提供 Nginx 服务器的服务性能，我们在后面的相关章节中对 Nginx 服务器的缓存功能有详细介绍。

7.3　Nginx 服务器的反向代理服务

Nginx 服务器的反向代理服务是其最常用的重要功能之一，在实际工作中应用广泛，涉及的配置指令也比较多，各类指令完成的功能也不尽相同。本节首先按照功能分类向大家介绍配置该服务需要掌握的配置指令。由反向代理服务又可以衍生出多种与此相关的 Nginx 服务器的重要功能，随后将逐步梳理这些功能，并提供配置实例供大家参考。

Nginx 服务器提供的反向代理服务也是比较高效的。它能够同时接收的客户端连接由 worker_processes 指令和 worker_connections 指令决定，计算方法为：worker_processes * worker_connections / 4。

配置 Nginx 服务器反向代理用到的指令如果没有特别说明，原则上可以出现在 Nginx 配置文件的 http 块、server 块或者 location 块中，但同正向代理服务的设置一样，一般是在搭建的 Nginx 服务器中单独配置一个 server 块用来设置反向代理服务。这些指令主要由 ngx_http_proxy_module 模块进行解析和处理。该模块是 Nginx 服务器的标准 HTTP 模块。

7.3.1　反向代理的基本设置的 21 个指令

学习 Nginx 服务器的反向代理服务，要涉及与后端代理服务器相关的配置，详细内容参考 6.1 节。本小节涉及的指令比较重要，是为客户端提供正常 Web 服务的基础，大家应该熟练掌握，尤其是 proxy_pass 指令，在实际应用过程中需要注意一些配置细节，需要小心使用。

1. proxy_pass 指令

该指令用来设置被代理服务器的地址，可以是主机名称、IP 地址加端口号等形式。其语法结构为：

```
proxy_pass URL;
```

其中，URL 为要设置的被代理服务器的地址，包含传输协议、主机名称或 IP 地址加端口号、URI 等要素。传输协议通常是“http”或者“https://”。指令同时还接受以“unix”开始的 UNIX-domain 套接字路径。例如：

```
proxy_pass http://www.myweb.name/uri;
proxy_pass http://localhost:8000/uri/;
proxy_pass http://unix:/tmp/backend.socket:/uri/;
```

如果被代理服务器是一组服务器的话，可以使用 upstream 指令配置后端服务器组。例如：

```
...
upstream proxy_svrs                                          #配置后端服务器组
{
  server http://192.168.1.1:8001/uri/;
  server http://192.168.1.2:8001/uri/;
  server http://192.168.1.3:8001/uri/;
}
server
{
  ...
  listen 80;
  server_name  www.myweb.name;
  location /
  {
    proxy_pass  proxy_svrs;                                  #使用服务器组的名称
  }
}
```

这里首先需要提醒大家 proxy_pass 指令在使用服务器组名称时应该注意的一个细节。在上例中，在组内的各个服务器中都指明了传输协议“http://”，而在 proxy_pass 指令中就不需要指明了。如果现在将 upstream 指令的配置改为：

```
upstream proxy_svrs
{
  server 192.168.1.1:8001/uri/;
  server 192.168.1.2:8001/uri/;
  server 192.168.1.3:8001/uri/;
}
```

我们就需要在 proxy_pass 指令中指明传输协议“http://”：

```
proxy_pass  http://proxy_svrs;
```

在使用该指令的过程中还要注意，*URL* 中是否包含有 URI，Nginx 服务器的处理方式是不同的。如果 *URL* 中不包含 URI，Nginx 服务器不会改变原地址的 URI；但是如果包含了 URI，Nginx 服务器将会使用新的 URI 替代原来的 URI。我们举例来说明。

请看下面的 Nginx 配置片段：

```
...
server
{
  ...
  listen 80;
  server_name  www.myweb.name;
  location  /server/
  {
    ...
    proxy_pass  http://192.168.1.1;
  }
}
```

如果客户端使用“http://www.myweb.name/server”发起请求，该请求被配置中显示的 location 块进行处理，由于 proxy_pass 指令的 *URL* 变量不含有 URI，所以转向的地址为“http://192.168.1.1/server”。

我们再来看下面的 Nginx 配置片段：

```
...
server
{
  ...
  listen 80;
  server_name  www.myweb.name;
  location /server/
  {
    ...
    proxy_pass  http://192.168.1.1/loc/;
  }
}
```

在该配置实例中，proxy_pass 指令的 *URL* 包含了 URI“/loc/”。如果客户端仍然使用“http://www.myweb.name/server/”发起请求，Nginx 服务器将会把地址转向“http://192.168.1.1/loc/”。

通过上面的实例，我们可以总结出，在使用 proxy_pass 指令时，如果不想改变原地址中的 URI，就不要在 *URL* 变量中配置 URI。

明白了上面这两个例子的用法，我们来解释大家经常讨论的一个问题，就是 proxy_pass 指令的 *URL* 变量末尾是否加斜杠“/”的问题。

请看这两个配置示例：

```
#配置 1:   proxy_pass  http://192.168.1.1;
#配置 2:   proxy_pass  http://192.168.1.1/;
```

配置 1 和配置 2 的区别在于，配置 2 中 proxy_pass 指令的 URL 变量末尾添加了斜杠“/”，这意味着配置 2 中 proxy_pass 指令的 *URL* 变量包含了 URI “/”，而配置 1 中 proxy_pass 指令的 URL 变量不包含 URI。理解了这一点，我们就可以解释下面的实例和现象了。大家注意各例子之间的对比。

实例 1：

```
...
server
{
  ...
  listen 80;
  server_name  www.myweb.name;
  location  /                                          #注意 location 的 uri 变量
  {
    ...
    #配置 1:   proxy_pass  http://192.168.1.1;
    #配置 2:   proxy_pass  http://192.168.1.1/;
  }
}
```

在该配置中，location 块使用“/”作为 *uri* 变量的值来匹配不包含 URI 的请求 URL。由于请求 URL 中不包含 URI，因此配置 1 和配置 2 的效果是一样的。比如，客户端的请求 *URL* 为“http://www.myweb.name/index.htm”，其将会被实例 1 中的 location 块匹配成功并进行处理。不管使用配置 1 还是配置 2，转向的 *URL* 都为：“http://192.168.1.1/index.htm”。

实例 2：

```
...
server
{
  ...
  listen 80;
  server_name  www.myweb.name;
  location  /server/                                   #注意 location 的 uri 变量
  {
    ...
    #配置 1:   proxy_pass  http://192.168.1.1;
    #配置 2:   proxy_pass  http://192.168.1.1/;
  }
}
```

在该配置中，location 块使用“/server/”作为 *uri* 变量的值来匹配包含 URI “/server/”的请求 URL。这时，使用配置 1 和配置 2 的转向结果就不相同了。使用配置 1 的时候，proxy_pass 指令中的 *URL* 变

量不包含 URI，Nginx 服务器将不改变原地址的 URI；使用配置 2 的时候，proxy_pass 指令中的 *URL* 变量包含 URI　“/”，Nginx 服务器会将原地址的 URI 替换为“/”。

比如，客户端的请求 URL 为“http://www.myweb.name/server/index.htm”，将会被实例 2 中的 location 块匹配成功并进行处理。使用配置 1 的时候，转向的 URL 为“http://192.168.1.1/server/index.htm”，原地址的 URI　“/server/”未被改变；但使用配置 2 时，转向的 URL 为“http://192.168.1.1/index.htm”，可以看到，原地址的 URI　“/server/”被替换为“/”。

大家在应用过程中，一定要注意到该指令在配置上的细节问题，分清楚 URL 和 URI 的区别与联系，并能够正确使用它们配置出符合需求的 Nginx 服务器。

2. proxy_hide_header 指令

该指令用于设置 Nginx 服务器在发送 HTTP 响应时，隐藏一些头域信息。其语法结构为：

```
proxy_hide_header field;
```

其中，*field* 为需要隐藏的头域。该指令可以在 http 块、server 块或者 location 块中进行配置。

3. proxy_pass_header 指令

默认情况下，Nginx 服务器在发送响应报文时，报文头中不包含“Date”、“Server”、“X-Accel”等来自被代理服务器的头域信息。该指令可以设置这些头域信息以被发送，其语法结构为：

```
proxy_pass_header field;
```

其中，field 为需要发送的头域。该指令可以在 http 块、server 块或者 location 块中进行配置。

4. proxy_pass_request_body 指令

该指令用于配置是否将客户端请求的请求体发送给代理服务器，其语法结构为：

```
proxy_pass_request_body on | off;
```

默认设置为开启（on），开关可以在 http 块、server 块或者 location 块中进行配置。

5. proxy_pass_request_headers 指令

该指令用于配置是否将客户端请求的请求头发送给代理服务器，其语法结构为：

```
proxy_pass_request_body on | off;
```

默认设置为开启（on），开关可以在 http 块、server 块或者 location 块中进行配置。

6. proxy_set_header 指令

该指令可以更改 Nginx 服务器接收到的客户端请求的请求头信息，然后将新的请求头发送给被代理的服务器。其语法结构为：

```
proxy_set_header field value;
```

- *field*，要更改的信息所在的头域。
- *value*，更改的值，支持使用文本、变量或者变量的组合。

默认情况下，该指令的设置为：

```
proxy_set_header Host  $proxy_host;
proxy_set_header Connection  close;
```

请看一些设置实例：

```
proxy_set_header Host  $http_host;          #将目前 Host 头域的值填充成客户端的地址。
```

```
proxy_set_header Host  $host;                    #将当前 location 块的 server_name 指令值填充到
Host 头域。
proxy_set_header Host  $host:$proxy_port;        #将当前 location 块的 server_name 指令值和
                                                 # listener 指令值一起填充到 Host 头域。
```

7. proxy_set_body 指令

该指令可以更改 Nginx 服务器接收到的客户端请求的请求体信息，然后将新的请求体发送给被代理的服务器。其语法结构为：

```
proxy_set_body value;
```

其中，*value* 为更改的信息，支持使用文本、变量或者变量的组合。

8. proxy_bind 指令

官方文档中对该指令的解释是，强制将与代理主机的连接绑定到指定的 IP 地址。通俗来讲就是，在配置了多个基于名称或者基于 IP 的主机的情况下，如果我们希望代理连接由指定的主机处理，就可以使用该指令进行配置，其语法结构为：

```
proxy_bind address;
```

其中，*address* 为指定主机的 IP 地址。

注意

Nginx 0.8.22 及以上版本支持该指令。

9. proxy_connect_timeout 指令

该指令配置 Nginx 服务器与后端被代理服务器尝试建立连接的超时时间，其语法结构为：

```
proxy_connect_timeout time;
```

其中，*time* 为设置的超时时间，默认为 60 s。

10. proxy_read_timeout 指令

该指令配置 Nginx 服务器向后端被代理服务器（组）发出 read 请求后，等待响应的超时时间，其语法结构为：

```
proxy_read_timeout time;
```

其中，*time* 为设置的超时时间，默认为 60 s。

11. proxy_send_timeout 指令

该指令配置 Nginx 服务器向后端被代理服务器（组）发出 write 请求后，等待响应的超时时间，其语法结构为：

```
proxy_read_timeout time;
```

其中，*time* 为设置的超时时间，默认为 60 s。

12. proxy_http_version 指令

该指令用于设置用于 Nginx 服务器提供代理服务的 HTTP 协议版本，其语法结构为：

```
proxy_http_version 1.0 | 1.1;
```

默认设置为 1.0 版本。1.1 版本支持 upsteam 服务器组设置中的 keepalive 指令。

13. proxy_method 指令

该指令用于设置 Nginx 服务器请求被代理服务器时使用的请求方法，一般为 POST 或者 GET。设

置了该指令，客户端的请求方法将被忽略。其语法结构为：

```
proxy_method method;
```

其中，*method* 的值可以设置为 POST 或者 GET，注意不加引号。

14. proxy_ignore_client_abort 指令

该指令用于设置在客户端中断网络请求时，Nginx 服务器是否中断对被代理服务器的请求，其语法结构为：

```
proxy_ignore_client_abort on | off;
```

默认设置为 off，当客户端中断网络请求时，Nginx 服务器中断对被代理服务器的请求。

15. proxy_ignore_headers 指令

该指令用于设置一些 HTTP 响应头中的头域，Nginx 服务器接收到被代理服务器的响应数据后，不会处理被设置的头域。其语法结构为：

```
proxy_ignore_headers field ...;
```

其中，*field* 为要设置的 HTTP 响应头的头域，例如“X-Accel-Redirect”、“X-Accel-Expires”、“Expires”、“Cache-Control”或“Set-Cookie”等。

16. proxy_redirect 指令

该指令用于修改被代理服务器返回的响应头中的 Location 头域和“Refresh”头域，与 proxy_pass 指令配合使用。比如，Nginx 服务器通过 proxy_pass 指令将客户端的请求地址重写为被代理服务器的地址，那么 Nginx 服务器返回给客户端的响应头中“Location”头域显示的地址就应该和客户端发起请求的地址相对应，而不是代理服务器直接返回的地址信息，否则就会出问题。该指令解决了这个问题，可以把代理服务器返回的地址信息更改为需要的地址信息。其语法结构为：

```
1 proxy_redirect redirect replacement;
2 proxy_redirect default;
3 proxy_redirect off;
```

- *redirect*，匹配“Location”头域值的字符串，支持变量的使用和正则表达式。
- *replacement*，用于替换 *redirect* 变量内容的字符串，支持变量的使用。

该指令的用法我们通过几个配置实例来解释。

对于第 1 个结构，假设被代理服务器返回的响应头中“Location”头域为：

```
Location: http://localhost:8081/proxy/some/uri/
```

该指令设置为：

```
proxy_redirect  http://localhost:8081/proxy/  http://myweb/frontend/;
```

Nginx 服务器会将“Location”头域的信息更改为：

```
Location: http://myweb/frontend//some/uri/
```

这样，客户端收到的响应信息头部中的“Location”头域也就被更改了。

结构 2 使用 default，代表使用 location 块的 *uri* 变量作为 *replacement*，并使用 proxy_pass 变量作为 *redirect*。请看下面两段配置，它们的配置效果是等同的。

```
#配置 1
location /server/
```

```
{
  proxy_pass  http://proxyserver/source/;
  proxy_redirect  default;
}
#配置 2
location /server/
{
  proxy_pass  http://proxyserver/source/;
  proxy_redirect  http://proxyserver/source/  /server/;
}
```

使用结构 3 可以将当前作用域下所有的 proxy_redirect 指令配置全部设置为无效。

17. proxy_intercept_errors 指令

该指令用于配置一个状态是开启还是关闭。在开启该状态时，如果被代理的服务器返回的 HTTP 状态代码为 400 或者大于 400，则 Nginx 服务器使用自己定义的错误页（使用 error_page 指令）；如果是关闭该状态，Nginx 服务器直接将被代理服务器返回的 HTTP 状态返回给客户端。其语法结构为：

```
proxy_intercept_errors on | off;
```

18. proxy_headers_hash_max_size 指令

该指令用于配置存放 HTTP 报文头的哈希表的容量，其语法结构为：

```
proxy_headers_hash_ max _size size;
```

其中，size 为 HTTP 报文头哈希表的容量上限，默认为 512 个字符，即：

```
proxy_headers_hash_ max _size 512;
```

Nginx 服务器为了能够快速检索 HTTP 报文头中的各项信息，比如服务器名称、MIME 类型、请求头名称等，使用哈希表存储这些信息。Nginx 服务器在申请存放 HTTP 报文头的空间时，通常以固定大小为单位申请，该大小由 proxy_headers_hash_bucket_size 指令配置。

在 Nginx 配置中，不仅能够配置整个哈希表的大小上限，对大部分的内容项，也可以配置其大小上限，比如 server_names_hash_max_size 指令和 server_names_hash_bucket_size 指令用来设置服务器名称的字符数长度。

19. proxy_headers_hash_bucket_size 指令

该指令用于设置 Nginx 服务器申请存放 HTTP 报文头的哈希表容量的单位大小。该指令的具体作用在上面 proxy_headers_hash_max_size 指令的使用中已经说明。其语法结构为：

```
proxy_headers_hash_bucket_size size;
```

其中，size 为设置的容量，默认为 64 个字符。

20. proxy_next_upstream 指令

在配置 Nginx 服务器反向代理功能时，如果使用 upstream 指令配置了一组服务器作为被代理服务器，服务器组中各服务器的访问规则遵循 upstream 指令配置的轮询规则，同时可以使用该指令配置在发生哪些异常情况时，将请求顺次交由下一个组内服务器处理。该指令的语法结构为：

```
proxy_next_upstream status …;
```

其中，*status* 为设置的服务器返回状态，可以是一个或者多个。这些状态包括：

- error，在建立连接、向被代理的服务器发送请求或者读取响应头时服务器发生连接错误。

- timeout，在建立连接、向被代理的服务器发送请求或者读取响应头时服务器发生连接超时。
- invalid_header，被代理的服务器返回的响应头为空或者无效。
- http_500 | http_502 | http_503 | http_504 | http_404，被代理的服务器返回 500、502、503、504 或者 404 状态代码。
- off，无法将请求发送给被代理的服务器。

注意

与被代理的服务器进行数据传输的过程中发送错误的请求，不包含在该指令支持的状态之内。

21. proxy_ssl_session_reuse 指令

该指令用于配置是否使用基于 SSL 安全协议的会话连接（“https://”）被代理的服务器，其语法结构为：

```
proxy_ssl_session_reuse on | off;
```

默认设置为开启(on)状态。如果我们在错误日志中发现“SSL3_GET_FINISHED:digest check failed”的情况，可以将该指令配置为关闭（off）状态。

7.3.2　Proxy Buffer 的配置的 7 个指令

我们简单介绍一下 Proxy Buffer 的工作原理，这对理解和该功能相关的几个指令是有帮助的。

我们首先要明确的是，接下来介绍的与 Proxy Buffer 相关的指令 proxy_buffer_size、proxy_buffers、proxy_busy_buffers_size 等，它们的配置都是针对每一个请求起作用的，而不是全局概念，即每个请求都会按照这些指令的配置来配置各自的 Buffer，Nginx 服务器不会生成一个公共的 Proxy Buffer 供代理请求使用。

Proxy Buffer 启用以后，Nginx 服务器会异步地将被代理服务器的响应数据传递给客户端。

Nginx 服务器首先尽可能地从被代理服务器那里接收响应数据，放置在 Proxy Buffer 中，Buffer 的大小由 proxy_buffer_size 指令和 proxy_buffers 指令决定。如果在接收过程中，发现 Buffer 没有足够大小来接收一次响应的数据，Nginx 服务器会将部分接收到的数据临时存放在磁盘的临时文件中，磁盘上的临时文件路径可以通过 proxy_temp_path 指令进行设置，临时文件的大小由 proxy_max_temp_file_size 指令和 proxy_temp_file_write_size 指令决定。一次响应数据被接收完成或者 Buffer 已经装满后，Nginx 服务器开始向客户端传输数据。

每个 Proxy Buffer 装满数据后，在从开始向客户端发送一直到 Proxy Buffer 中的数据全部传输给客户端的整个过程中，它都处于 BUSY 状态，期间对它进行的其他操作都会失败。同时处于 BUSY 状态的 Proxy Buffer 总大小由 proxy_busy_buffers_size 指令限制，不能超过该指令设置的大小。

当 Proxy Buffer 关闭时，Nginx 服务器只要接收到响应数据就会同步地传递给客户端，它本身不会读取完整的响应数据。

1. proxy_buffering 指令

该指令用于配置是否启用或者关闭 Proxy Buffer，其语法结构为：

```
proxy_buffering on | off;
```

默认设置为开启状态。

开启和关闭 Proxy Buffer 还可以通过在 HTTP 响应头部的“X-Accel-Buffering”头域设置“yes”或者“no”来实现，但 Nginx 配置中 proxy_ignore_headers 指令的设置可能导致该头域设置失效。

2. proxy_buffers 指令

该指令用于配置接收一次被代理服务器响应数据的 Proxy Buffer 个数和每个 Buffer 的大小，其语法结构为：

```
proxy_buffers number size;
```

- *number*，Proxy Buffer 的个数。
- *size*，每个 Buffer 的大小，一般设置为内存页的大小。根据平台的不同，可能为 4KB 或者 8KB。

由这个指令可以得到接收一次被代理服务器响应数据的 Proxy Buffer 总大小为 *number* * *size*。该指令的默认设置为：

```
proxy_buffers 8 4k|8k;
```

3. proxy_buffer_size 指令

该指令用于配置从被代理服务器获取的第一部分响应数据的大小，该数据中一般包含了 HTTP 响应头，Nginx 服务器通过它来获取响应数据和被代理服务器的一些必要信息。该指令的语法结构为：

```
proxy_buffer_size size;
```

其中，*size* 为设置的缓存大小，默认设置为 4KB 或者 8KB，保持与 proxy_buffers 指令中的 *size* 变量相同，当然也可以设置得更小。

注意该指令不要和 proxy_buffers 指令混淆。

4. proxy_busy_buffers_size 指令

该指令用于限制同时处于 BUSY 状态的 Proxy Buffer 的总大小，通过上面对 Proxy Buffer 机制的阐述，相信大家很容易理解该指令的功能。该指令的语法结构为：

```
proxy_busy_buffers_size size;
```

其中，*size* 为设置的处于 BUSY 状态的缓存区总大小。默认设置为 8KB 或者 16KB。

5. proxy_temp_path 指令

该指令用于配置磁盘上的一个文件路径，该文件用于临时存放代理服务器的大体积响应数据。如果 Proxy Buffer 被装满后，响应数据仍然没有被 Nginx 服务器完全接收，响应数据就会被临时存放在该文件中。在上面对 Proxy Buffer 机制的阐述中已经提到过这一点。该指令的语法结构为：

```
proxy_temp_path path [level1 [level2 [level3]]];
```

- *path*，设置磁盘上存放临时文件的路径。
- *levelN*，设置在 *path* 变量设置的路径下第几级 hash 目录中存放临时文件。

看一个例子：

```
proxy_temp_path /nginx/proxy_web/spool/proxy_temp 1 2;
```

在该例子中，配置临时文件存放在/nginx/proxy_web/spool/proxy_temp 路径下的第二级目录中。基于该配置的一个临时文件的路径可以是：

```
/nginx/proxy_web/spool/proxy_temp/1/10/00000100101
```

6. proxy_max_temp_file_size 指令

该指令用于配置所有临时文件的总体积大小，存放在磁盘上的临时文件大小不能超过该配置值，这避免了响应数据过大造成磁盘空间不足的问题，其语法结构为：

```
proxy_max_temp_file_size size;
```

其中，*size* 为设置的临时文件总体积上限值，默认设置为 1024 MB。

7. proxy_temp_file_write_size 指令

该指令用于配置同时写入临时文件的数据量的总大小，合理的设置可以避免磁盘 IO 负载过重导致系统性能下降的问题，其语法结构为：

```
proxy_temp_file_write_size size;
```

其中，*size* 为设置的数据量总大小上限值，默认设置根据平台的不同，可以为 8 KB 或 16 KB，一般与平台的内存页大小相同。

7.3.3　Proxy Cache 的配置的 12 个指令

Proxy Cache 机制与缓存数据的产生和使用有很大关系。Proxy Cache 机制实际上是 Nginx 服务器提供的 Web 缓存机制的一部分，我们在后面章节中还会详细介绍 Web 缓存机制的其他方面。到这里有人可能产生疑问了，刚才学习的 Porxy Buffer 和这里的 Proxy Cache 有什么区别和联系吗？这里笔者需要解释一下，大家可以从含义和功能两方面区分 Nginx 服务器的这两类机制。

Buffer 和 Cache 虽然都是用于提供 IO 吞吐效率的，但是它们是一对不同的概念，翻译成中文分别是“缓冲”和“缓存”两个词。Buffer，主要用于传输效率不同步或者优先级别不相同的设备之间传递数据，一般通过对一方数据进行临时存放，再统一发送的办法传递给另一方，以降低进程之间的等待时间，保证速度较快的进程不发生间断，临时存放的数据一旦传送给另一方，这些数据本身也就没有用处了；Cache，主要用于将硬盘上已有的数据在内存中建立缓存数据，提高数据的访问效率，对于过期不用的缓存可以随时销毁，但不会销毁硬盘上的数据。

在 Nginx 服务器中，Porxy Buffer 和 Proxy Cache 都与代理服务相关，主要用来提供客户端与被代理服务器之间的交互效率。Porxy Buffer 实现了被代理服务器响应数据的异步传输，Proxy Cache 则主要实现 Nginx 服务器对客户端数据请求的快速响应。Nginx 服务器在接收到被代理服务器的响应数据之后，一方面通过 Porxy Buffer 机制将数据传递给客户端，另一方面根据 Proxy Cache 的配置将这些数据缓存到本地硬盘上。当客户端下次要访问相同的数据时，Nginx 服务器直接从硬盘检索到相应的数据返回给用户，从而减少与被代理服务器交互的时间。

特别需要说明的是，Proxy Cache 机制依赖于 Porxy Buffer 机制，只有在 Porxy Buffer 机制开启的情况下 Proxy Cache 的配置才发挥作用。

在 Nginx 服务器中还提供了另一种将被代理服务器数据缓存到本地的方法 Proxy Store，与 Proxy Cache 的区别是，它对来自被代理服务器的响应数据，尤其是静态数据只进行简单的缓存，不支持缓存过期更新、内存索引建立等功能，但支持设置用户或用户组对缓存数据的访问权限。

1. proxy_cache 指令

该指令用于配置一块公用的内存区域的名称，该区域可以存放缓存的索引数据。这些数据在 Nginx 服务器启动时由缓存索引重建进程负责建立，在 Nginx 服务器的整个运行过程中由缓存管理进程负责

定时检查过期数据、检索等管理工作。该指令的语法结构为：

```
proxy_cache zone | off;
```

- *zone*，设置的用于存放缓存索引的内存区域的名称。
- *off*，关闭 proxy_cache 功能，是默认的设置。

从 Nginx 0.7.66 开始，Proxy Cache 机制开启后会检查被代理服务器响应数据 HTTP 头中的“Cache-Control”头域、“Expires”头域。当“Cache-Control”头域中的值为“no-cache”、“no-store”、“private”或者“max-age”赋值为 0 或无意义时，当“Expires”头域包含一个过期的时间时，该响应数据不被 Nginx 服务器缓存。这样做的主要目的是为了避免私有的数据被其他客户端得到。

2. proxy_cache_bypass 指令

该指令用于配置 Nginx 服务器向客户端发送响应数据时，不从缓存中获取的条件。这些条件支持使用 Nginx 配置的常用变量。其语法结构为：

```
proxy_cache_bypass string ...;
```

其中，*string* 为条件变量，支持设置多个，当至少有一个字符串指令不为空或者不等于 0 时，响应数据不从缓存中获取。

看一个例子：

```
proxy_cache_bypass $cookie_nocache $arg_nocache $arg_comment $http_pragma $http_
authorization;
```

其中，$cookie_nocache、$arg_nocache、$arg_comment、$http_pragma 和$http_authorization 都是 Nginx 配置文件的变量，有关 Nginx 配置常用变量的使用，请查询附录 A 的相关内容，这里就不再赘述了。

3. proxy_cache_key 指令

该指令用于设置 Nginx 服务器在内存中为缓存数据建立索引时使用的关键字，其语法结构为：

```
proxy_cache_key string;
```

其中，*string* 为设置的关键字，支持变量。在 Nginx 0.7.48 之前的版本中默认的设置为：

```
proxy_cache_key $scheme$proxy_host$request_uri;
```

如果我们希望缓存数据包含服务器主机名称等关键字，则可以将该指令设置为：

```
proxy_cache_key "$scheme$host$request_uri";
```

在 Nginx 0.7.48 之后的版本中，通常使用以下配置：

```
proxy_cache_key $scheme$proxy_host$uri$is_args$args;
```

4. proxy_cache_lock 指令

该指令用于设置是否开启缓存的锁功能。在缓存中，某些数据项可以同时被多个请求返回的响应数据填充。开启该功能后，Nginx 服务器同时只能有一个请求填充缓存中的某一数据项，这相当于给该数据项上锁，不允许其他请求操作。其他的请求如果也想填充该项，必须等待该数据项的锁被释放。这个等待时间由 proxy_cache_lock_timeout 指令配置。

该指令的语法结构为：

```
proxy_cache_lock on | off;
```

该指令在 Nginx 1.1.2 及之后的版本中可以使用，默认情况下，设置为关闭状态。

5. proxy_cache_lock_timeout 指令

该指令用于设置缓存的锁功能开启以后锁的超时时间。具体细节参见 proxy_cache_lock 指令的相关内容。该指令的语法结构为：

```
proxy_cache_lock_timeout time;
```

其中，*time* 为设置的时间，默认为 5 s。

6. proxy_cache_min_uses 指令

该指令用于设置客户端请求发送的次数，当客户端向被代理服务器发送相同请求达到该指令设定的次数后，Nginx 服务器才对该请求的响应数据做缓存。合理设置该值可以有效地降低硬盘上缓存数据的数量，并提高缓存的命中率。该指令的语法结构为：

```
proxy_cache_min_uses number;
```

其中，*number* 为设置的次数。默认设置为 1。

7. proxy_cache_path 指令

该指令用于设置 Nginx 服务器存储缓存数据的路径以及和缓存索引相关的内容，其语法结构为：

```
proxy_cache_path path [levels=levels] keys_zone=name:size1 [inactive=time1]
[max_size=size2] [loader_files=number] [loader_sleep=time2]
[loader_threshold=time3];
```

- *path*，设置缓存数据存放的根路径，该路径应该是预先存在于磁盘上的。
- *levels*，设置在相对于 *path* 指定目录的第几级 hash 目录中缓存数据。levels=1，表示一级 hash 目录；levels=1:2，表示两级，依次类推。目录的名称是基于请求 URL 通过哈希算法获取到的。
- *name:size1*，Nginx 服务器的缓存索引重建进程在内存中为缓存数据建立索引，这一对变量用来设置存放缓存索引的内存区域的名称和大小。
- *time1*，设置强制更新缓存数据的时间，当硬盘上的缓存数据在设定的时间内没有被访问时，Nginx 服务器就强制从硬盘上将其删除，下次客户端访问该数据时重新缓存。该指令默认设置为 10 s。
- *size2*，设置硬盘中缓存数据的大小限制。我们知道，硬盘中的缓存源数据由 Nginx 服务器的缓存管理进程进行管理，当缓存的大小超过该变量的设置时，缓存管理进程将根据最近最少被访问的策略删除缓存。
- *number*，设置缓存索引重建进程每次加载的数据元素的数量上限。在重建缓存索引的过程中，进程通过一系列的递归遍历读取硬盘上的缓存数据目录及缓存数据文件，对每个数据文件中的缓存数据在内存中建立对应的索引，我们称每建立一个索引为加载一个数据元素。进程在每次遍历过程中可以同时加载多个数据元素，该值限制了每次遍历中同时加载的数据元素的数量。默认设置为 100。
- *time2*，设置缓存索引重建进程在一次遍历结束、下次遍历开始之间的暂停时长。默认设置为 50 ms。
- *time3*，设置遍历一次磁盘缓存源数据的时间上限。默认设置为 200 ms。

该指令设置比较复杂，一般需要设置前面三个指令的情形比较多，后面的几个变量与 Nginx 服务器缓存索引重建进程及管理进程的性能相关，一般情况下保持默认设置就可以了。我们来看几个简单的配置实例：

```
proxy_cache_path /nginx/cache/a levels=1 keys_zone=a:10m;
proxy_cache_path /nginx/cache/b levels=2:2 keys_zone=b:100m;
proxy_cache_path /nginx/cache/c levels=1:1:2 keys_zone=c:1000m;
```

注意

该指令和其他指令不同，只能放在 http 块中。

8. proxy_cache_use_stale 指令

如果 Nginx 在访问被代理服务器过程中出现被代理的服务器无法访问或者访问错误等现象时，Nginx 服务器可以使用历史缓存响应客户端的请求，这些数据不一定和被代理服务器上最新的数据相一致，但对于更新频率不高的后端服务器来说，Nginx 服务器的该功能在一定程度上能够为客户端提供不间断访问。该指令用来设置一些状态，当后端被代理的服务器处于这些状态时，Nginx 服务器启用该功能。该指令的语法结构为：

```
proxy_cache_use_stale error | timeout | invalid_header | updating | http_500 | http_502
| http_503 | http_504 | http_404 | off ...;
```

该指令可以支持的状态如语法结构中所示。大部分状态和后文中要学习的 proxy_next_upstream 指令是对应的。各个状态的含义我们在学习 proxy_next_upstream 指令时详细讲解。

需要注意其中的 updating 状态的含义。该状态并不是指被代理服务器在 updating 状态，而是指客户端请求的数据在 Nginx 服务器中正好处于更新状态。这一点需要大家留意。

该指令的默认设置为 off。

9. proxy_cache_valid 指令

该指令可以针对不同的 HTTP 响应状态设置不同的缓存时间，其语法结构为：

```
proxy_cache_valid [ code ...] time;
```

- *code*，设置 HTTP 响应的状态代码。该指令可选，如果不设置，Nginx 服务器只为 HTTP 状态代码为 200、301 和 302 的响应数据做缓存。可以使用“any”表示缓存所有该指令中未设置的其他响应数据。
- *time*，设置缓存时间。

看几个例子：

```
proxy_cache_valid 200 302 10m;
proxy_cache_valid 301 1h;
proxy_cache_valid any 1m;
```

该例子中，对返回状态为 200 和 302 的响应数据缓存 10 分钟，对返回状态为 301 的响应数据缓存 1 小时，对返回状态为非 200、302 和 301 的响应数据缓存 1 分钟。

10. proxy_no_cache 指令

该指令用于配置在什么情况下不使用 cache 功能，其语法结构为：

```
proxy_no_cache string ...
```

其中，*string* 可以是一个或者多个变量。当 *string* 的值不为空或者不为“0”时，不启用 cache 功能。

11. proxy_store 指令

该指令配置是否在本地磁盘缓存来自被代理服务器的响应数据。这是 Nginx 服务器提供的另一种缓存数据的方法，但是该功能相对 Proxy Cache 简单一些，它不提供缓存过期更新、内存索引建立等功能，不占用内存空间，对静态数据的效果比较好。该指令的语法结构为：

```
proxy_store on | off | string;
```

- *on* | *off*，设置是否开启 Proxy Store 功能。如果使用变量 *on*，功能开启，缓存文件会存放到 alias 指令或 root 指令设置的本地路径下。默认设置为 *off*。
- *string*，自定义缓存文件的存放路径。如果使用变量 string，Proxy Store 功能开启，缓存文件会存放到指定的本地路径下。

Proxy Store 方法多使用在被代理服务器端发生错误的情况下，用来缓存被代理服务器的响应数据。

12. proxy_store_access 指令

该指令用于设置用户或用户组对 Proxy Store 缓存的数据的访问权限，其语法结构为：

```
proxy_store_access users:permissions ...;
```

- *users*，可以设置为 user、group 或者 all。
- *permissions*，设置权限。

有关 Proxy Store 方法的使用，我们通过官方给出的实例加深理解，在该实例中笔者通过注释对配置做了说明：

```
location /images/
{
  root  /data/www;
  error_page  404 = /fetch$uri;                          #定义了 404 错误的请求页面
}
location  /fetch/                                        #匹配 404 错误时的请求
{
  proxy_pass  http://backend;
  proxy_store  on;                                       #开启 Proxy Store 方法
  proxy_store_access  user:rw  group:rw  all:r;
  root  /data/www;                                       #缓存数据的路径
}
```

7.4　Nginx 服务器的负载均衡

Nginx 服务器反向代理服务的一个重要用途是实现负载均衡。随着信息数量的不断增长，目前网络的业务量急剧升高，访问量和数据流量也在飞速增长，因而对网络本身的处理能力和计算强度的要求也越来越高。现有的网络硬件条件显然不能满足日益增长的需求，但是完全抛弃已有的硬件环境又是不现实的，于是，就出现了“网络负载均衡”（Load Balancing）这样的技术，该技术一出现就迅速在网络建设中得到普及。

7.4.1　什么是负载均衡

负载均衡技术不是本书的重点，因此笔者在本节主要介绍相关的基本概念。

网络负载均衡技术的大致原理是利用一定的分配策略将网络负载平衡地分摊到网络集群的各个操作单元上，使得单个重负载任务能够分担到多个单元上并行处理，或者使得大量并发访问或数据流量分担到多个单元上分别处理，从而减少用户的等待响应时间。

在实际应用中，负载均衡会根据网络的不同层次（一般按照 OSI 的七层参考模型）进行划分。现代的负载均衡技术主要实现和作用于网络的第四层或第七层，完全独立于网络基础硬件设备，成为单独的技术设备。Nginx 服务器实现的负载均衡一般认为是七层负载均衡。

负载均衡主要通过专门的硬件设备实现或者通过软件算法实现。通过硬件设备实现的负载均衡效果好、效率高、性能稳定，但是成本比较高。通过软件实现的负载均衡主要依赖于均衡算法的选择和程序的健壮性。均衡算法也是多种多样的，常见的有两大类：即静态负载均衡算法和动态负载均衡算法。静态算法实现比较简单，在一般网络环境下也能达到比较好的效果，主要有一般轮询算法、基于比率的加权轮询算法以及基于优先级的加权轮询算法等。动态负载均衡算法在较为复杂的网络环境中适应性更强，效果更好，主要有基于任务量的最少连接优先算法、基于性能的最快响应优先算法、预测算法及动态性能分配算法等。

7.4.2　Nginx 服务器负载均衡配置

理解了“负载均衡”的概念，就可以利用 Nginx 服务器实现负载均衡的配置了。Nginx 服务器实现了静态的基于优先级的加权轮询算法，主要使用的配置是 proxy_pass 指令和 upsteam 指令，这些内容实际上很容易理解，关键点在于 Nginx 服务器的配置灵活多样，如何在配置负载均衡的同时合理地融合其他功能，形成一套可以满足实际需求的配置方案。

在本小节中，笔者为大家准备了一些配置 Nginx 负载均衡的基础实例片段，当然不可能将所有的配置情况包括在内，希望能够起到抛砖引玉的效果，同时也需要大家在实际应用过程中多总结多积累。在配置中需要注意的地方笔者将以注释的形式添加。

7.4.3　配置实例一：对所有请求实现一般轮询规则的负载均衡

在以下实例片段中，backend 服务器组中所有服务器的优先级全部配置为默认的 weight=1，这样它们会按照一般轮询策略依次接收请求任务。该配置是一个最简单的实现 Nginx 服务器负载均衡的配置。所有访问 www.myweb.name 的请求都会在 backend 服务器组中实现负载均衡。实例代码如下：

```
...
upstream backend                                    #配置后端服务器组
{
  server 192.168.1.2:80;
  server 192.168.1.3:80;
  server 192.168.1.4:80;                            #默认 weight=1
}
server
{
```

```
  listen 80;
  server_name  www.myweb.name;
  index  index.html  index.htm;
  location  /  {
    proxy_pass  http://backend;
    proxy_set_header  Host  $host;
    ...
  }
  ...
}
```

7.4.4　配置实例二：对所有请求实现加权轮询规则的负载均衡

与“配置实例一”相比，在该实例片段中，backend 服务器组中的服务器被赋予了不同的优先级别，weight 变量的值就是轮询策略中的“权值”。其中，192.168.1.2:80 的级别最高，优先接收和处理客户端请求；192.168.1.4:80 的级别最低，是接收和处理客户端请求最少的服务器，192.168.1.3:80 将介于以上两者之间。所有访问 www.myweb.name 的请求都会在 backend 服务器组中实现加权负载均衡。实例代码如下：

```
...
upstream backend                                          #配置后端服务器组
{
  server 192.168.1.2:80  weight=5;
  server 192.168.1.3:80  weight=2;
  server 192.168.1.4:80;                                  #默认 weight=1
}
server
{
  listen 80;
  server_name  www.myweb.name;
  index  index.html  index.htm;
  location  /  {
    proxy_pass  http://backend;
    proxy_set_header  Host  $host;
    ...
  }
  ...
}
```

7.4.5　配置实例三：对特定资源实现负载均衡

在该实例片段中，我们设置了两组被代理的服务器组，名为“videobackend”的一组用于对请求 video 资源的客户端请求进行负载均衡，另一组用于对请求 file 资源的客户端请求进行负载均衡。通过对 location 块 *uri* 的不同配置，我们就很轻易地实现了对特定资源的负载均衡。所有对“http://www.myweb.name/video/*”的请求都会在 videobackend 服务器组中获得均衡效果，所有对“http://www.myweb.name/file/*”的请求都会在 filebackend 服务器组中获得均衡效果。在该实例中展示的是实现一

般负载均衡的配置，对于加权负载均衡的配置可以参考“配置实例二”。

在 location /file/ {……}块中，我们将客户端的真实信息分别填充到了请求头中的“Host”、“X-Real-IP”和“X-Forwarded-For”头域，这样后端服务器组收到的请求中就保留了客户端的真实信息，而不是 Nginx 服务器的信息。实例代码如下：

```
… #其他配置
upstream videobackend                                          #配置后端服务器组 1
{
  server 192.168.1.2:80;
  server 192.168.1.3:80;
  server 192.168.1.4:80;
}
upstream filebackend                                           #配置后端服务器组 2
{
  server 192.168.1.5:80;
  server 192.168.1.6:80;
  server 192.168.1.7:80;
}
server
{
  listen 80;
  server_name  www.myweb.name;
  index  index.html  index.htm;
  location  /video/  {
    proxy_pass  http://videobackend;                           #使用后端服务器组 1
    proxy_set_header  Host  $host;
    …
  }
  location  /file/  {
    proxy_pass  http://filebackend;                            #使用后端服务器组 2
                                                               #保留客户端的真实信息
    proxy_set_header  Host  $host;
    proxy_set_header  X-Real-IP  $remote_addr;
    proxy_set_header  X-Forwarded-For  $proxy_add_x_forwarded_for;
    …
  }
  …
}
```

7.4.6　配置实例四：对不同域名实现负载均衡

在该实例片段中，我们设置了两个虚拟服务器和两组后端被代理的服务器组，分别用来接收不同的域名请求和对这些请求进行负载均衡处理。如果客户端请求域名为“home.myweb.name”，则由服务器 server 1 接收并转向 homebackend 服务器组进行负载均衡处理；如果客户端请求域名为“bbs.myweb.name”，则由服务器 server 2 接收并转向 bbsbackend 服务器组进行负载均衡处理。这样就实现了对不同域名的负载均衡。

需要注意两组后端服务器组中有一台服务器 server 192.168.1.4:80 是公用的。在该服务器上需要部署两个域名下的所有资源才能保证客户端请求不会出现问题。实例代码如下：

```
...
upstream bbsbackend
{
  server 192.168.1.2:80  weight=2;
  server 192.168.1.3:80  weight=2;
  server 192.168.1.4:80;
}
upstream homebackend
{
  server 192.168.1.4:80;
  server 192.168.1.5:80;
  server 192.168.1.6:80;
}
                                                    # 开始配置 server 1
server
{
  listen 80;
  server_name  home.myweb.name;
  index  index.html  index.htm;
  location  /  {
    proxy_pass  http://homebackend;
    proxy_set_header  Host  $host;
    ...
  }
  ...
}
                                                    # 开始配置 server 2
server
{
  listen 81;
  server_name  bbs.myweb.name;
  index  index.html  index.htm;
  location  /  {
    proxy_pass  http://bbsbackend;
    proxy_set_header  Host  $host;
    ...
  }
  ...
}
```

7.4.7　配置实例五：实现带有 URL 重写的负载均衡

首先，我们来看具体的源码，这是在实例一的基础上做的修改：

```
...
upstream backend
{
```

```
  server 192.168.1.2:80;
  server 192.168.1.3:80;
  server 192.168.1.4:80;
}
server
{
  listen 80;
  server_name  www.myweb.name;
  index  index.html  index.htm;
  location  /file/  {
    rewrite  ^(/file/.*)/media/(.*)\.*$  $1/mp3/$2.mp3  last;
}

  location  /  {
    proxy_pass  http://backend;
    proxy_set_header  Host  $host;
    ...
  }
  ...
}
```

该实例片段与“配置实例一”相比，增加了对 URI 包含“/file/”的 URL 重写功能。例如客户端的请求 URL 为“http://www.myweb.name/file/download/media/1.mp3”时，该虚拟服务器首先使用 location file/ {……}块将该 URL 进行重写为 http://www.myweb.name/file/download/mp3/1.mp3，然后新的 URL 再由 location / {……}块转发到后端的 backend 服务器组中实现负载均衡。这样，就轻而易举地实现了带有 URL 重写功能的负载均衡。在该配置方案中，一定要掌握清楚 rewrite 指令中 last 标记和 break 标记的区别，才能达到预计的效果。

以上 5 个配置实例展示了 Nginx 服务器实现不同情况下负载均衡配置的基本方法。由于 Nginx 服务器的功能在结构上是增量式的，因此，我们可以在这些配置的基础上继续添加更多功能，比如本书后面章节中要介绍的 Web 缓存等功能，以及前面介绍过的 Gzip 压缩技术、身份认证、权限管理等。同时在使用 upsteam 指令配置服务器组时，可以充分发挥各个指令的功能，配置出满足需求、高效稳定、功能丰富的 Nginx 服务器。

7.5　本章小结

本章重点介绍了 Nginx 服务器的代理服务，包括正向代理服务和反向代理服务。Nginx 服务器的代理服务是它的特色服务，使用十分广泛，尤其是反向代理服务，在功能和性能上都有出色的表现，受到广大用户的青睐。本章首先对正向代理服务和反向代理服务的基本知识进行了简单介绍，之后分别展示了正向代理服务和反向代理服务的配置指令，同时，提供了大量的配置实例来说明代理服务的使用，尤其是对反向代理的经典应用——负载均衡服务提供了较为详尽的说明。相信大家通过本章的学习，对 Nginx 服务器的代理服务有了明确的认识和应用能力。

第 8 章

Nginx 服务器的缓存机制

Nginx 服务器在该方面主要提供了两大方面的方案：一是负载均衡；二是 Web 缓存。在上一章中我们学习了 Nginx 服务器负载均衡的实现方法，本章我们重点学习如何配置和使用 Nginx 服务器的缓存机制。

本章将要学习到的主要内容有：

- Web 缓存技术的基础知识
- Nginx 服务器基于 Proxy Store 的缓存机制
- Nginx 服务器基于 memcached 的缓存机制
- Nginx 服务器基于 Proxy Cache 的缓存机制
- Nginx 与 Squid 服务器组合的配置

8.1 Web 缓存技术简述

传统观点认为，影响网络访问速度的主要因素有网络带宽、访问距离和服务器的处理能力。随着网络接入速度的不断提升，主干带宽的不断扩容，目前的网络环境已经得到了极大的改善，影响网络访问速度的主要瓶颈出现在服务器的承载能力和处理能力方面。在实际使用 Web 服务器的过程中，我们能够看到绝大多数的产品在提高自身负载能力的方面提供了各式各样有效的办法，比如使用镜像服务器、使用缓存服务器、实施负载均衡等。

响应速度历来是衡量 Web 应用和服务性能优劣的重要指标之一，尤其在动态网站在网络上泛滥的今天，除了优化发布内容本身以外，另一个主要的办法就是把不需要实时更新的动态页面输出结果转化成静态网页形成缓存，进而按照静态网页来访问。这样的机制使得动态网站的响应速度显著提升。

Web 缓存技术被认为是减轻服务器负载、降低网络拥塞、增强网络可扩展性的有效途径之一，其基本思想是利用客户访问的时间局部性原理，将客户访问过的内容建立一个副本，在一段时间内存放在本地，当该数据下次被访问时，不必连接到后端服务器，而是由本地保留的副本数据响应。

具体来说，该技术主要在 Web 服务器和客户端之间实现，Web 服务器首先根据客户端的请求从后端服务器获取响应数据，并传回给客户端，同时，Web 服务器将该响应数据在本地建立副本保存。当下次有相同的客户端请求时，Web 服务器直接使用本地的副本响应访问请求，而不是向后端的服务器再次发送请求。

保存在本地的这些副本具有一个过期时间（也叫新鲜度），超过该时间将会更新。判断一个副本数据是否为过期数据的办法有很多，可以使用保留时间来判断，也可以使用数据完整度来判断。许多 Web 服务器还具有校验功能，就是当某些副本数据过期以后，先向后端服务器发送校验请求，后端服务器对这些数据进行校验，如果发现原数据和副本没有差别，则将过期副本重新置为可用副本。

Web 缓存技术的优点是很明显的。由于客户端的部分请求内容直接从 Web 服务器处获取，该技术减轻了后端服务器的负载，同时也减少了 Web 服务器与后端服务器之间的网络流量，从而减轻了网络拥塞，同时还能减小数据传输延迟，有效降低客户访问延迟。该技术还能实现另一个很实用的功能，如果由于后端服务器故障或网络故障造成后端服务器无法响应客户请求，客户端可以从 Web 服务器中获取缓存的内容副本，这增强了数据的可用性，使得后端服务器的鲁棒性得到了加强。

Nginx 服务器作为高效的、备受推崇的 Web 服务器，其实现 Web 缓存技术的方法是多种多样的。在接下来的几个小节中笔者将对常用的五种配置方案进行详细的阐述。

8.2　404 错误驱动 Web 缓存

Nginx 服务器的这一种实现 Web 缓存的原理其实很简单，主要还是依靠自身的 Proxy Store 功能对 404 错误进行重定向来实现。当 Nginx 服务器在处理客户端请求时，发现请求的资源数据不存在，会产生 404 错误，然后服务器通过捕获该错误，进一步转向后端服务器请求数据，最后将后端服务器的响应数据传回给客户端，同时在本地进行缓存。从实现原理上来看，Nginx 服务器向后端服务器发起数据请求并完成 Web 缓存，主要是由产生的 404 错误驱动的。

请大家看一个简单的实现 404 错误驱动 Web 缓存的配置方案片段：

```
...
location  /  {
  ...
  root  /myweb/server/;                                    #主目录
  error_page 404  =200  /errpage$request_uri;              #404 定向到/errpage 目录下
}
                                                           # 捕获 404 错误的重定向
location  /errpage/  {
  ...
  internal;                                                #该目录不能通过外部链接直接访问
  alias /home/html/;
  proxy_pass  http://backend/;                             #后端 upstream 地址或者源地址
```

```
 proxy_set_header  Accept-Encoding  "";          #后端不返回压缩（gzip/deflate）数据
 proxy_store on;                                 #指定 nginx 将代理返回的文件保存
 proxy_store_access  user:rw  group:rw  all:r;   #配置缓存数据的访问权限
 proxy_temp_path  /myweb/server/tmp;             #配置临时目录，该目录要和
                                                 #/myweb/server/在同一个硬盘分区内
}
```

配置将 404 错误响应进行重定向，然后使用 location 块捕获重定向请求，向后端服务器发起请求获取响应数据，然后将数据转发给客户端的同时缓存到本地。

8.3　资源不存在驱动 Web 缓存

该方法同“404 错误驱动 Web 缓存”的方法在原理上大同小异，不同之处是，该方法是通过 location 块的 location if 条件判断直接驱动 Nginx 服务器与后端服务器的通信和 Web 缓存，而后者是对资源不存在引发的 404 错误进行捕获，进而驱动 Nginx 服务器与后端服务器的通信和 Web 缓存。

请大家再看一个简单的通过判断资源不存在驱动 Web 缓存的配置方案片段：

```
…#其他配置
location  /  {
  …#其他配置
  root /home/html/;
  internal;  #配置该目录不能通过外部链接直接访问
  alias /myweb/server/;
  proxy_set_header  Accept-Encoding  "";  #配置后端不返回压缩（gzip 或 deflate）数据
  proxy_store on;  #指定 nginx 将代理返回的文件保存
  proxy_store_access  user:rw  group:rw  all:r;  #配置缓存数据的访问权限
  proxy_temp_path  /myweb/server/tmp;  #配置临时目录，该目录要和/myweb/server/在同一个硬
盘分区内
 if  ( !-f $request_filename )  #判断请求资源是否存在
  {
    proxy_pass  http://backend/;  #配置后端 upstream 地址或者源地址
  }
}
```

在配置实例中使用 location if 条件判断支持的“!-f”判断请求的资源在 Nginx 服务器上是否存在，如果不存在就通过后端服务器获取数据，然后回传给客户端，同时使用 Proxy Store 进行缓存。

以上两种缓存机制在原理上是相近的，在实际的应用中，我们通常可以将 Proxy Store 的缓存目录配置到/dev/shm 中提高缓存数据的处理速度。如果不是在内存中保存缓存数据，这两种缓存机制不支持缓存数据的清理机制，缓存文件会一直保存在本地占用硬盘空间。

这两种缓存机制还有需要注意的地方是它们只能缓存 200 状态代码下的响应数据，这就是为什么我们在介绍“404 错误驱动 Web 缓存”机制时的配置实例中将 404 错误重新改写为 200 状态的原因。

两种缓存机制也不支持动态链接请求。比如 getsource?id=1 和 getsource?id=2 这两个请求，这两种缓存机制会忽略 id=1 参数，从而造成返回的资源不正确等问题。这是这两种缓存机制的缺点，但在实际应用中有时也有一定的使用价值。我们在后面的章节中会学习 Nginx 搭配 Squid 的服务器架构实现

方案，在那里我们会看到，对于带有参数的链接，Squid 服务器是支持的，因此带有不同参数的相同域名请求 Squid 服务器是被区别对待的，而 Nginx 服务器将不会区分这样的请求，这在一定的情况下对后端服务器能够起到一定的保护作用。我们在学习相关章节时再进行详细的分析。

8.4 基于 memcached 的缓存机制的 6 个指令

memcached 是一套高性能的基于分布式的缓存系统，用于动态 Web 应用以减轻后台数据服务器的负载。它可以独立于任何程序单独作为后台程序运行，通过在内存中的缓存数据来减少对后台数据服务器的请求，从而提高对客户端的响应。

memcached 可以处理并发的网络连接。它在内存中开辟一块空间，然后建立一个 Hash 表，将缓存数据通过键/值存储在 Hash 表中进行管理。memcached 由服务端和客户端两个核心组件组成，服务端先通过计算“键”的 Hash 值来确定键/值对在服务端所处的位置。当确定键/值对的位置后，客户端就会发送一个查询请求给对应的服务端，让它来查找并返回确切的数据。

在 Nginx 服务器的标准 HTTP 模块中有一个 ngx_http_memcached_module 模块，专门用于处理和 memcached 相关的配置和功能实现，虽然在目前的版本中还没有支持完整的功能，但是其性能很好，对于一般的应用场景是比较好的选择方案。我们来学习一下该模块支持的一些配置指令。

1. memcached_pass 指令

该指令用于配置 memcached 服务器的地址，其语法结构为：

```
memcached_pass address;
```

其中，*address* 为 memcached 服务器的地址，支持 IP+端口的地址或者是域名地址。也可以使用 upstream 指令配置一个 memcached 服务器组，然后将 *address* 配置为 upstream 的名称。

2.memcached_connect_timeout 指令

该指令用于配置连接 memcached 服务器的超时时间，其语法结构为：

```
memcached_connect_timeout time;
```

其中，*time* 为设置的超时时间，默认为 60 s。建议该时间不要超过 75 s。

3. memcached_read_timeout 指令

该指令配置 Nginx 服务器向 memcached 服务器发出两次 read 请求之间的等待超时时间，如果在该时间内没有进行数据传输，连接将会被关闭。该语法结构为：

```
memcached_read_timeout time;
```

其中，*time* 为设置的超时时间，默认为 60 s。

4. memcached_send_timeout 指令

该指令配置 Nginx 服务器向 memcached 服务器发出两次 write 请求之间的等待超时时间，如果在该时间内没有进行数据传输，连接将会被关闭。该语法结构为：

```
memcached_send_timeout time;
```

其中，*time* 为设置的超时时间，默认为 60 s。

5. memcached_buffer_size 指令

该指令用于配置 Nginx 服务器用于接收 memcached 服务器响应数据的缓存区大小，其语法结构为：

```
memcached_buffer_size size;
```

其中，*size* 为设置的缓存区大小，一般是所在平台的内存页大小的倍数。默认设置为：

```
memcached_buffer_size 4k|8k;
```

6. memcached_next_upstream 指令

该指令在配置了一组 memcached 服务器的情况下使用。服务器组中各 memcached 服务器的访问规则遵循 upstream 指令配置的轮询规则，同时可以使用该指令配置在发生哪些异常情况时，将请求顺次交由下一个组内服务器处理。该指令的语法结构为：

```
memcached_next_upstream status …;
```

其中，*status* 为设置的 memcached 服务器返回状态，可以是一个或者多个。这些状态包括：

- error，在建立连接、向 memcached 服务器发送请求或者读取响应头时服务器发生连接错误。
- timeout，在建立连接、向 memcached 服务器发送请求或者读取响应头时服务器发生连接超时。
- invalid_header，memcached 服务器返回的响应头为空或者无效。
- not_found，memcached 服务器未找到对应的键/值对。
- off，无法将请求发送给 memcached 服务器。

注意

与 memcached 服务器进行数据传输的过程中的发送错误，不包含在该指令支持的状态之内。

以上是使用 memcached 缓存经常用到的一些指令。在实际配置 Nginx 服务器使用 memcached 时，我们需要对 Nginx 配置的全局变量$memcached_key 进行设置。请看下面的配置实例片段，该片段是 Nginx 官方文档提供的：

```
… #其他配置
server {
    location / {
        …#其他配置
        set $memcached_key "$uri?$args";
        memcached_pass 192.168.1.4: 8080;
        error_page 404 502 504 = @fallback;
    }
    location @fallback {
        proxy_pass http://backend;
    }
    …#其他配置
}
```

在该配置中，我们设置$memcached_key 变量的值为"$uri?$args"，Nginx 服务器会根据该值调用 Hash 算法向 memcached 服务器发送查询请求。如果在请求缓存数据的时候返回的状态代码为 404、502 或者 504 时，则将错误进行重定位，重定位后的请求被 location @fallback {…}捕获并转向后台服务器请求实际数据。

8.5 Proxy Cache 缓存机制

我们在第 7 章学习 Nginx 服务器的反向代理服务时，已经对 Proxy Cache 缓存机制的原理和配置指令进行了详细介绍。该机制是 Nginx 服务器自己实现的类似于 Squid 的缓存机制，它使用 md5 算法将请求链接 hash 后生成文件系统目录保存响应数据。

Nginx 服务器在启动后，会生成专门的进程对磁盘上的缓存文件进行扫描，在内存中建立缓存索引，提高访问效率，并且还会生成专门的管理进程对磁盘上的缓存文件进行过期判定、更新等方面的管理。Proxy Cache 缓存机制支持对任意链接响应数据的缓存，不仅限于 200 状态时的数据。从这几点来看，Proxy Cache 缓存机制不管在性能还是在数据管理上要远远优于 Proxy Cache 缓存机制。

Proxy Cache 缓存机制的一个缺陷是，它没有实现自动清理磁盘上缓存源数据的功能，因此在长时间使用过程中会对服务器存储造成一定的压力。

在下面这个配置实例中，实现了 Nginx 服务器 Proxy Cache 缓存机制的一般配置：

```
…#其他配置
http {
  …#其他配置
proxy_cache_path /myweb/server/proxycache levels=1:2 max_size=2m inactive=5m
loader_sleep=1m; keys_zone=MYPROXYCACHE:10m  #配置了缓存数据存放路径和 Proxy Cache 使用
的内存 Cache 空间
proxy_temp_path /myweb/server/tmp;  #配置响应数据的临时存放目录
server {
…… #其他配置
location  /  {
…… #其他配置
proxy_pass  http://www.myweb.name/;
proxy_cache  MYPROXYCACHE;  #配置使用 MYPROXYCACHE 这个 keys_zone
proxy_cache_valid 200 302 1h;  #配置 200 状态和 302 状态的响应缓存 1 小时
proxy_cache_valid 301 1d;  #配置 301 状态的响应缓存 1 天
proxy_cache_valid any 1m;  #配置其他状态的响应数据缓存 1 分钟
  }
 }
}
```

在该实例中，我们首先在 http 块中配置了缓存数据存放路径和 Proxy Cache 使用的内存 Cache 空间。缓存数据存放在磁盘上/myweb/server/proxycache 目录下，它包含两级 hash 目录，缓存数据的总量不能超过 20MB；如果缓存在 5 分钟内没有被访问，则强制更新。内存 Cache 空间的名字为 MYPROXYCACHE，大小不能超过 10 MB，每隔 1 分钟遍历一次磁盘缓存源数据，更新内存 Cache 中的缓存索引。

之后，我们在 server 块中，配置使用上面设置好的 MYPROXYCACHE 内存空间进行 Proxy Cache 工作，对不同响应状态的数据缓存时间进行了配置。

Proxy Cache 机制是 Nginx 服务器自身实现的一个功能比较完整、性能也不错的缓存机制，在实际应用过程中使用比较广泛。下面笔者为大家展示一个更加完整的 Nginx 服务器使用 Proxy Cache 机制

的例子。

在该例中添加了对过期缓存数据的半自动请求功能。该功能使用到了第三方模块 ngx_cache_purge，我们可以从站点 http://labs.frickle.com/files/ngx_cache_purge-1.0.tar.gz 下载到该模块的源码。该模块启用后，通过访问特定的 URL 可以实现对过期缓存的清理。在配置编译 Nginx 服务器源码时，需要使用--add-module 参数将该第三方模块添加到 Nginx 服务器中进行编译。有关第三方模块的编译和安装请大家参阅第 16 章的相关章节。Nginx 服务器的详细配置如下，笔者通过注释的形式添加了相关说明：

```
user  nobody  nobody;
worker_processes  2;
error_log /usr/local/webserver/nginx/logs/nginx_error.log crit;
pid /usr/local/webserver/nginx/nginx.pid;
worker_rlimit_nofile 65535;
events
{
    use epoll;
    worker_connections 65535;
}
http
{
    include mime.types;
    default_type application/octet-stream;
    charset utf-8;
    sendfile on;
    tcp_nopush on;
    keepalive_timeout 60;
    tcp_nodelay on;
    client_body_buffer_size 512k;
    proxy_connect_timeout 5;
    proxy_read_timeout 60;
    proxy_send_timeout 5;
    proxy_buffer_size 16k;
    proxy_buffers 4 64k;
    proxy_busy_buffers_size 128k;
    proxy_temp_file_write_size 128k;
    gzip on;
    gzip_min_length 1k;
    gzip_buffers 4 16k;
    gzip_http_version 1.1;
    gzip_comp_level 2;
    gzip_types text/plain application/x-javascript text/css application/xml;
    gzip_vary on;
#设置 Web 缓存区名称为 cache_one，内存缓存空间大小为 200MB，1 天清理一次缓存，硬盘缓存空间大小为
30GB。
    proxy_temp_path /data0/proxy_temp_dir;
```

```
proxy_cache_path /data0/proxy_cache_dir levels=1:2 keys_zone=cache_one:200m
inactive=1d max_size=30g;
    upstream backend {
        server 192.168.1.3:80 weight=1 max_fails=2 fail_timeout=30s;
        server 192.168.1.4:80 weight=1 max_fails=2 fail_timeout=30s;
        server 192.168.1.5:80 weight=1 max_fails=2 fail_timeout=30s;
    }
    server
    {
        listen  80;
        server_name  myweb;
        index index.html index.htm;
        root /data0/htdocs/www;
        location /
        {
            #如果后端的服务器返回 502、504、执行超时等错误，将请求转发到另一台服务器。
            proxy_next_upstream http_502 http_504 error timeout invalid_header;
            proxy_cache cache_one;
            #针对不同的 HTTP 状态码设置不同的缓存时间
             proxy_cache_valid 200 304 12h;
            # Web 缓存的 Key 值以域名、URI、参数组成
            proxy_cache_key  $host$uri$is_args$args;
            proxy_set_header Host  $host;
            proxy_set_header X-Forwarded-For  $remote_addr;
            proxy_pass  http://backend_server;
            expires 1d;
        }
#配置缓存清除功能
        location ~ /purge(/.*)
        {
            #设置只允许指定的 IP 或 IP 段才可以清除 URL 缓存。
            allow 127.0.0.1;
            allow 192.168.0.0/16;
            deny all;
            proxy_cache_purge cache_one $host$1$is_args$args;
        }
        #配置数据不缓存
        location ~ .*\.(php|jsp|cgi)?$
        {
            proxy_set_header Host    $host;
            proxy_set_header X-Forwarded-For   $remote_addr;
            proxy_pass http://backend;
    }
}
```

在该配置中，我们启用了 ngx_cache_purge 模块的功能，根据配置内容，如果想要清除已有的缓存内容，在浏览器的地址栏中输入 URL：http:// myweb/purge/eppic.jpg 就可以清除 eppic.jpg 缓存了。

Nginx 服务器本身的 Web 缓存机制功能是比较完整的，但在实际的应用中，除了使用 Nginx 服务器自身提供的 Web 缓存机制外，还可以使用专门的 Web 缓存服务器来完成缓存的工作，而 Nginx 服务器专门用作反向代理服务器和提供负载均衡等功能。

8.6　Nginx 与 Squid 组合

Squid Cache（简称为 Squid）是目前在大访问量的网站建设中应用非常广泛的 Web 缓存服务器。它可以作为网页服务器的前置缓存服务器来缓存相关的请求数据，也可以缓存公网资源为局域网内用户提供共享资源。但是，Squid 服务本身不支持在单台服务器同一端口（例如要反向代理 Web 必须指定 80 端口）下运行多个进程，这样的话就需要给每一个 Squid 服务分配一台服务器设备，这样非常浪费资源。

Nginx 具有反向代理服务功能，支持服务器组的配置和对组间服务器的轮询，我们可以运用 Nginx 的这一功能，实现在同一台服务器中运行多个 Squid 服务的目的。

8.6.1　Squid 服务器的配置

将多个 Squid 服务安装到同一台服务器上的不同目录下，并配置不同的监听端口，比如：

```
/squid0 监听在 squid_server_ip:10010
/squid1 监听在 squid_server_ip:10011
/squid2 监听在 squid_server_ip:10012
```

其中，*squid_server_ip* 是部署 Squid 服务的服务器 IP 地址。

有关 Squid 服务器的具体配置内容我们不做过多介绍，请大家参阅其官方文档。

8.6.2　Nginx 服务器的配置

Nginx 服务器的配置代码如下：

```
user  www www;
worker_processes  10;
error_log  /usr/local/nginx/logs/nginx_error.log;
pid  /usr/local/nginx/nginx.pid;
worker_rlimit_nofile 51200;
events {
   use epoll;
   worker_connections  51200;
}
   http {
   include          mime.types;
   default_type     application/octet-stream;
   sendfile         on;
   tcp_nopush       on;
   tcp_nodelay      on;
   keepalive_timeout  65;
   #配置 Squid 服务器组
```

```
    upstream squid_server {
        server  squid_server_ip:10010;
        server  squid_server_ip:10011;
        server  squid_server_ip:10012;
    }
    #gzip  on;
    server {
        listen        80;
        server_name  myweb;
        #将客户端请求转向 Squid 服务器组处理
        location  /  {
              proxy_pass       http://squid_server;
              proxy_redirect     off;
              proxy_set_header   Host           $host:80;
              proxy_set_header   X-Real-IP       $remote_addr;
              proxy_set_header   X-Forwarded-For  $proxy_add_x_forwarded_for;
        }
    }
}
```

8.7　基于第三方模块 ncache 的缓存机制

ncache 是新浪公司（http://www.sina.com.cn/）的开源产品，作为 Nginx 服务器的一个 HTTP 模块进行开发，以实现更好的兼容性和可扩展性。

ncache 完全是一套定制化的产品，可以满足快速部署、网络高并发量、数据海量存储量的需求，它使用先进的技术进行组合。我们可以从 http://code.google.com/p/ncache/站点了解更多相关的信息以及源码下载链接和使用文档链接。不在这里详细介绍 ncache 模块的使用方法，有关 Nginx 服务器第三方模块的编译和安装将在专门的章节中讲解，请大家参阅第 16 章。

8.8　本章小结

缓存机制在 Web 服务器提高自身负载能力、提高用户请求响应速率等方面非常重要，绝大多数的 Web 服务器都支持缓存机制，甚至有些服务器专门用来完成服务器的缓存功能，比如 Squid 服务器。在本章中，我们介绍了 Nginx 服务器实现缓存机制的多种方法，包括其自身提供的两种 Web 缓存机制 Proxy Store 和 Proxy Cache、基于 memcached 的缓存机制、与 Squid 服务器联合实现的缓存机制以及基于第三方模块 ncache 的缓存机制。我们重点对前面三种机制的配置和实现原理进行了介绍。大家在实际应用中应该根据具体环境和具体需求采用最适当的方式实现 Web 缓存。

第 9 章

Nginx 服务器的邮件服务

邮件服务是 Nginx 服务器的基础功能之一，也是最初开发的主要需求，不过其在实际中应用得似乎不多。本章主要介绍配置 Nginx 邮件服务的基本指令，并且提供了比较完整的配置实例供大家在实际应用中进行参考。首先对邮件服务的实现原理和相关概念、涉及的协议进行简单介绍，这样有利于大家更好地理解 Nginx 邮件服务器的相关配置。

我们在本章将要学习的主要内容有：

- 邮件服务的基本知识
- Nginx 配置邮件服务的指令
- 邮件服务的配置实例

9.1 邮件服务

Nginx 邮件服务器在配置上并不是单纯地提供邮件服务，它还能够在此基础上同时实现负载均衡、数据压缩、代理等多项功能，并且各种功能的配置方法简单，部署方便。相信大家在日常工作和生活中经常使用各种邮件服务

邮件服务，俗称“电子邮件”，是网络上最为流行的应用之一。邮件服务属于典型的异步通信，其成本低廉、响应迅速、易于操作。现代邮件服务支持的传递内容十分丰富，如支持超链接、HTML 文本、图像、视频、音频等多种类型的数据。

一套完整的邮件服务主要有三类主要部件构成：分别是用户代理、邮件服务器和用于实现传输的简单邮件传送协议（Simple Mail Transfer Protocol，SMTP）。

- “用户代理”部分相当于我们平时用于收发电子邮件的“信箱”，也就是通常所说的“邮箱”

（Mailbox），它主要用于阅读、回复、转发、保存和编写邮件消息，比如 mail.google.com、mail.163.com、mail.yahoo.com 等公共站点以及各单位内部提供的邮件站点。20 世纪末也出现了可以安装在个人 PC 端本地的基于图形用户界面的电子邮件用户代理客户端，不过其需要其他协议的支持。它们在原理和功能上都是一样的，都属于“用户代理”的范畴。

- “邮件服务器”实现了部分邮件数据存储和维护功能，相当于用户邮箱的载体。每个使用邮件服务的用户都有一个位于某邮件服务器上的邮箱。该用户收到的邮件将会全部放在他对应的邮箱中以便对其进行维护和管理。现代的邮箱也支持保存用户发出的邮件数据。除此之外，该部分还有一个重要的功能就是支持“简单邮件传送协议”的实现。
- “简单邮件传送协议 SMTP”是邮件服务器在提供服务的过程中应该遵守的规范，它定义了如何实现把邮件消息从发信人的邮件服务器传送到收信人的邮件服务器。该协议属于 TCP/IP 协议簇，它帮助发送或中转邮件时查找下一个目的地。该协议支持认证，简单地说就是要求用户必须提供账户和密码才可以登录邮件服务器上自己对应的邮箱，这在很大程度上能避免用户受到垃圾邮件的侵扰。实现了 SMTP 协议的服务器也通常被称为“SMTP 服务器”。

在现代的邮件服务中，还有两类重要协议。邮局协议（Post Office Protocol，POP）规定了如何实现将个人 PC 接入到网络中的邮件服务器和下载邮件的协议，目前使用的是第三版本，因此通常称为 POP3 协议。该协议是网络电子邮件的第一个离线协议标准，它允许用户从邮件服务器上将邮件存储到本地，同时删除保存在服务器上的邮件数据。实现了 POP3 协议的服务器通常被称为“POP3 服务器”。

另一类协议是交互式邮件存取协议（Internet Mail Access Protocol，IMAP），它是和 POP3 类似的邮件访问标准协议之一，不同之处主要在于邮件服务器上的邮件不会在用户下载后被删除，在本地对邮件进行的任何操作也都会反馈到邮件服务器上。也就是说，使用 IMAP 协议的话，邮件服务器上邮箱中的邮件和客户端本地的邮件是一致的。实现了 IMAP 协议的服务器通常被称为“IMAP 服务器”。

使用 Nginx 服务器提供的邮件服务，对以上三种协议都有良好的支持，并且可以对邮件代理、用户认证进行配置，同时支持基于 SSL/TLS 协议的邮件服务的配置。

9.2　Nginx 邮件服务的配置的 12 个指令

用于支持 Nginx 邮件服务及相关配置的标准模块包括 ngx_mail_core_module、ngx_mail_pop3_module、ngx_mail_imap_module、ngx_mail_smtp_module、ngx_mail_auth_http_module、ngx_mail_proxy_module 和 ngx_mail_ssl_module 等。从模块的命名上我们就可以看出各个模块支持的主要功能。第一个实现了 Nginx 邮件服务的核心功能，第二、三、四三个模块实现了上一节学习过的三类用于邮件传输的协议，后面的三个模块分别实现了用户认证、邮件代理、支持基于 SSL/TLS 协议等功能。其中，ngx_mail_ssl_module 模块要求 OpenSSL 库的支持。

注意

在目前的 Nginx 服务器发行包中，这些用于提供邮件服务的模块均不会默认安装。因此在编译 Nginx 服务器时，需要在 configuration 时添加--with-mail（安装前 6 种模块）或者--with-mail_ssl_module（最后一种模块）参数。

Nginx 服务器使用 mail 块来形成一个包含有 Nginx 邮件服务各项配置的配置块。也就是说，所有和邮件服务相关的配置都应该包含在 mail 块中进行配置。mail 块的级别相当于我们在配置 Nginx Web 服务器时使用到的 http 块。同样，在 mail 块中，也可以形成一个或多个 server 块来指定不同的虚拟主机。在本章中，如果没有特殊说明，我们提到的 server 块均是指 mail 块中的 server 块。下面我们来学习配置 Nginx 邮件服务应该掌握的一些配置指令。

1. listen 指令

该指令用于配置邮件服务器服务监听的 IP 地址和端口，其语法结构为：

```
listen address:port;
```

- *address*，为邮件服务器服务监听的 IP 地址，支持通配符“*”、主机名称。
- *port*，为邮件服务器服务监听端口。

请大家看几个配置实例：

```
listen 127.0.0.1:110;
listen *:110;
listen 110;                                          # 与*:110 的配置含义相同
listen localhost:110;
```

从 Nginx 0.7.58 开始，该指令也支持 IPv6 地址，例如：

```
listen [::1]:120;
listen [::]:120;
```

从 Nginx 1.3.5 开始，该指令也支持 UNIX-domain sockets，例如：

```
listen unix:/var/run/nginx.sock;
```

该指令只能在 server 块中进行配置。

2. server_name 指令

该指令用于为每个 server 块构成的虚拟主机配置的域名，其语法结构为：

```
server_name name;
```

其中，name 为配置的服务器域名。

如果 mail 块中配置了多个虚拟主机，该指令只能在 server 块中配置；如果只有一个虚拟主机，该指令可以在 mail 块中配置。

3. protocol 指令

该指令用于配置当前虚拟主机支持的协议，其语法结构为：

```
protocol imap | pop3 | smtp;
```

对 IMAP、POP3、SMTP 三种协议的配置我们在后面会学习到。如果不明确指定虚拟主机支持的协议，在 Nginx 邮件服务有能力提供三种协议的处理时，则使用三种协议的默认接听端口来接收并处理客户端请求。

该指令只能在 server 块中进行配置。

4. so_keepalive 指令

该指令用于配置后端代理服务器是否启用“TCP keepalive”模式来处理 Nginx 邮件服务器转发的

客户端连接，其语法结构为：

```
so_keepalive on | off;
```

默认情况下，该指令配置为 off。

该指令可以在 mail 块或者 server 块中配置。

5. 配置 POP3 协议

用于配置 POP3 协议的指令有两个：分别是 pop3_auth 指令和 pop3_capabilities 指令。

pop3_auth 指令用于配置 POP3 认证用户的方式，其语法结构为：

```
pop3_auth method ...;
```

其中，*method* 支持以下配置：

- plain，使用 USER/PASS、AUTH PLAIN、AUTH LOGIN 方法认证。这也是邮件服务提供 POP3 协议支持时最基本的认证方式，也是 Nginx 邮件服务的默认设置。
- apop，使用 APOP 方法认证。该方法需要客户端提供的密码是非加密密码。
- cram-md5，使用 AUTH CRAM-MD5 方法认证。该方法也需要客户端提供的密码是非加密密码。

该指令可以在 mail 块中或者 server 块中配置。

pop3_capabilities 指令用于配置 POP3 协议的扩展功能，其语法结构为：

```
pop3_capabilities extension ...;
```

其中，*extension* 为要加入 POP3 协议的扩展。有关 POP3 协议的标准扩展大家可以参阅 http://www.iana.org/assignments/pop3-extension-mechanism 上的相关内容，这部分超出了本书的讨论范围，笔者不多赘述。

该指令可以在 mail 块中或者 server 块中配置，默认配置为：

```
pop3_capabilities TOP USER UIDL;
```

6. 配置 IMAP 协议

用于配置 IMAP 协议的指令包括 imap_auth 指令、imap_capabilities 指令和 imap_client_buffer 指令三个。前两个指令和配置 POP3 协议时使用的两个用法是相同的。

imap_auth 指令，用于配置 POP3 认证用户的方式，其语法结构为：

```
imap_auth method ...;
```

其中，*method* 支持以下配置：

- plain，使用 AUTH=PLAIN 方法认证。仍然是 Nginx 邮件服务提供 IMAP 协议的默认设置。
- login，使用 AUTH=LOGIN 方法进行认证。
- cram-md5，使用 AUTH CRAM-MD5 方法认证。该方法也需要客户端提供的密码是非加密密码。

该指令可以在 mail 块中或者 server 块中配置。

smtp_auth 指令，用于配置 IMAP 协议的扩展功能，其语法结构为：

```
imap_capabilities extension ...;
```

其中，*extension* 为要加入 IMAP 协议的扩展。有关 IMAP 协议的标准扩展大家可以参阅 http://www.iana.org/assignments/imap4-capabilities 上的相关内容，这部分超出了本书的讨论范围，笔者也不多赘述。

该指令可以在 mail 块中或者 server 块中配置，默认配置为：

```
imap_capabilities IMAP4 IMAP4rev1 UIDPLUS;
```

imap_client_buffer 指令，配置用于 IMAP 协议读数据缓存的大小，其语法结构为：

```
imap_client_buffer size;
```

其中，size 为配置的读缓存大小，一般为平台的一个内存页大小。默认配置为：

```
imap_client_buffer 4k|8k;
```

7. 配置 SMTP 协议

用于配置 SMTP 协议的指令包括 smtp_auth 指令和 smtp_capabilities 指令，它们的用法也和前面两个协议中的基本相同。

smtp_auth 指令用于配置 SMTP 认证用户的方式，其语法结构为：

```
smtp_auth method ...;
```

其中，*method* 支持以下配置：

- plain，使用 AUTH=PLAIN 方法认证。
- login，使用 AUTH=LOGIN 方法进行认证。
- cram-md5，使用 AUTH CRAM-MD5 方法认证。该方法也需要客户端提供的密码是非加密密码。

该指令可以在 mail 块中或者 server 块中配置，默认配置为：

```
smtp_auth login plain;
```

smtp _capabilities 指令用于配置 SMTP 协议的扩展功能，其语法结构为：

```
smtp _capabilities extension ...;
```

其中，extension 为要加入 SMTP 协议的扩展。有关 SMTP 协议的标准扩展大家可以参阅 http://www.iana.org/assignments/mail-parameters 上的相关内容，这部分超出了本书的讨论范围，笔者也不多赘述。

该指令可以在 mail 块中或者 server 块中配置。

8. auth_http 指令

该指令用于配置 Nginx 提供邮件服务时的用于 HTTP 认证的服务器地址，其语法结构为：

```
auth_http URL;
```

其中，*URL* 为 HTTP 认证服务器的地址。

该指令可以在 mail 块或者 server 块中进行配置。

9. auth_http_header 指令

通过该指令可以在 Nginx 服务器向 HTTP 认证服务器发起认证请求时，向请求头添加指定的头域。例如：

```
auth_http_header X-Auth-Key "secret_string";
```

该指令可以在 mail 块或者 server 块中进行配置。

10. auth_http_timeout 指令

该指令用于配置 Nginx 服务器向 HTTP 认证服务器发起认证请求后等待响应的超时时间，其语法结构为：

```
auth_http_timeout time;
```

其中，*time* 为超时时间，默认设置为 60 s。一般该时间设置不超过 75 s。

11. proxy_buffer 指令

该指令用于配置了后端代理服务器（组）的情况，用来配置 Nginx 服务器代理缓存的大小，一般为平台的一个内存页大小。默认配置为：

```
proxy_buffer 4k|8k;
```

该指令可以在 mail 块或者 server 块中进行配置。

12. proxy_pass_error_message 指令

该指令用于配置了后端代理服务器（组）的情况，用来配置是否将后端服务器上邮件服务认证过程中产生的错误信息发送给客户端，其语法结构为：

```
proxy_pass_error_message on | off;
```

通常情况下，如果 Nginx 邮件服务通过 HTTP 认证成功后，后端被代理服务器（组）不会再产生错误信息。但在有些 POP3 服务器中，会出现 HTTP 认证成功，但后端被代理服务器产生认证错误的情况。使用该指令在这些情况下是有用的。默认情况下，该指令设置为 off。

9.3　Nginx 邮件服务配置实例

以上是我们在使用 Nginx 服务器提供邮件服务功能时应该掌握的一些配置指令。在本节中向大家展示一个比较完整的邮件服务配置实例，帮助大家加深理解。

```
...
mail {
    server_name  mail.myweb.name;
    auth_http  mail.postfix.cn:80/auth.php;    #配置了 HTTP 认证的地址
    imap_capabilities  IMAP4rev1  UIDPLUS  IDLE LITERAL+  QUOTA;
    pop3_auth    plain  apop  cram-md5;
    pop3_capabilities  LAST  TOP  USER  PIPELINING  UIDL;
    smtp_auth    login  plain  cram-md5;
    smtp_capabilities  "SIZE 10485760"  ENHANCEDSTATUSCODES  8BITMIME  DSN;
    xclient      off;
server {
    listen   25;
    protocol  smtp;
}
server {
    listen   110;
    protocol  pop3;
```

```
    proxy_pass_error_message  on;
}
server {
    listen   143;
    protocol  imap;
}
24 }
```

在上面的配置实例中，邮件服务器的域名为 mail.myweb.name（加粗部分），配置了三台虚拟服务器分别用于提供 SMTP 协议、POP3 协议和 IMAP 协议。斜体加粗行是对三种协议的配置。listen 25 关闭了对 xClient 的支持。auth_http mail.postfix.cn:80/auth.php 配置了 HTTP 认证的地址，我们可以根据具体的平台环境选择合适的认证实现方式。这里使用 PHP 实现了一个基本的认证机制，代码示例如下：

```
<?php
    if (!isset($_SERVER["HTTP_AUTH_USER"] ) || !isset($_SERVER["HTTP_AUTH_PASS"] )){
    fail();
  }
    $uname=$_SERVER["HTTP_AUTH_USER"] ;
    $upass=$_SERVER["HTTP_AUTH_PASS"] ;
    $protocol=$_SERVER["HTTP_AUTH_PROTOCOL"] ;
                                                          // 配置后端服务器端口
    $backend_port=88;
      if ($protocol=="imap") {
        $backend_port=143;
  }
      if ($protocol=="smtp") {
        $backend_port=25;
  }
                                                          //认证
      if($username == "someone@mail.myweb.name ") {
        $server_ip = "192.168.1.4";
  }
 else {
       fail();
  }
      pass($server_ip, $backend_port);
      function pass($server,$port)  {
    header("Auth-Status: OK");
    header("Auth-Server: $server");
    header("Auth-Port: $port");
    exit;
  }
  function fail(){
    header("Auth-Status: Invalid login or password");
    exit;
  }
?>
```

9.4 本章小结

从历史上来看，Nginx 服务器最初产生的主要目的就是支持邮件服务，虽然现在使用这一服务的场合不多了，但它仍然是 Nginx 服务器的重要功能之一。本章我们对 Nginx 服务器的邮件服务进行了简单的介绍，梳理了相关的配置指令，并且提供了一个配置实例供大家参考。在实际应用中，如果大家对邮件服务器的功能需求不是很高，可以考虑使用 Nginx 服务器，毕竟它的灵活性和稳定性在众多 Web 服务器产品中是首屈一指的。

第 10 章

Nginx 源码结构

学习 Nginx 源码，首先要从宏观上把握源码的模块结构和各个模块之间的联系，在思路上对源码的总体实现要有一定的认识，这样在深入到具体的程序编码设计时就能够比较容易地找到程序设计者的编程思路，并能够按照这个思路按图索骥，理清源码结构。

本章从两个方面向大家介绍 Nginx 源码的结构，以期大家能够在宏观上对 Nginx 源码的实现有所了解和感悟。本章我们将要学习以下内容：

- Nginx 服务器源码目录结构。
- 从源码结构看 Nginx 的模块化结构。

10.1 Nginx 源码的 3 个目录结构

在第 2 章中我们提到解压 Linux 版本的 Nginx 压缩包后，有一个 src 目录，其中存放了 Nginx 软件的所有源代码。进入该目录以后，能够看到整个 Nginx 源码的组织情况：

```
# ls
core  event  http  mail  misc  os
```

我们在本章不打算深入到源码的具体实现，只从源码的目录结构来看看 Nginx 服务器的设计结构。

mail 目录中存放了实现 Nginx 服务器邮件服务的源码，主要实现对邮件服务依赖的数据结构的定义和初始化，对 SMTP 协议、POP3 协议和 IMAP 协议的实现，以及对 SSL 的支持等。

misc 目录中存放了两个文件。ngx_cpp_test_module.cpp 文件实现的功能是测试程序中引用的头文件是否与 C++兼容。ngx_google_perftools_module.c 文件是用来支持 Google PerfTools 的使用的。Google Perftools 包含四个工具，用于优化内存分配的效率和速度，帮助在高并发的情况下很好地控制内存的

使用，对 Nginx 服务器的运行做出进一步优化。

os 目录，默认只包含一个 unix 目录，其中存放的源代码是针对“类 Unix 系统”，如 Solaris、FreeBSD 等的特殊情况，进行了实现。

core 目录、event 目录和 http 目录，是我们学习的重点，需要详细介绍一下。

10.1.1　core 目录

该目录中存放了 Nginx 使用到的关键数据结构和 Nginx 内核实现的源码。我们大致浏览一下该目录中的重要文件。

- nginx.*文件，包含 Nginx 程序入口函数 main()的文件，实现了对 Nginx 各模块的整体控制。
- ngx_connection.*文件，实现与网络连接管理相关的功能。
- ngx_inet.*文件，实现与 Socket 网络套接字相关的功能。
- ngx_cycle.*文件，实现对系统整个运行过程中参数、资源的统一管理和调配。
- ngx_log.*文件，实现日志输出、管理的相关功能。
- ngx_file.*文件，实现文件读写相关功能。
- ngx_regex.*文件，实现 Nginx 服务器对正则表达式的支持。
- ngx_string.*文件，实现对字符串处理的基本功能。
- ngx_times.文件，实现对时间的获取和处理操作。

该目录中，还包含了一些重要数据结构的定义和操作，比如 ngx_array.*、ngx_list.*、ngx_queue.*、ngx_hash.*、ngx_*tree.*和 ngx_output_chain.*等；一些与内存管理相关的源码，比如 ngx_palloc.*、ngx_shmtx.*和 ngx_open_file_cache.*等。

可以说，该目录中的源码定义了 Nginx 服务器赖以运行的最基础的数据结构，实现了对它们的基本操作，也实现了用户各模块公共调用的基本功能。

10.1.2　event 目录

该目录中的源码不多，但是实现了 Nginx 服务器的事件驱动模型，实现了 Nginx 服务器的消息机制。这一部分的源码质量的好坏，对 Nginx 服务器的承载能力是有巨大影响的。该目录下主要有这些文件：

```
# ls
modules  ngx_event_accept.c   ngx_event_connect.c  ngx_event_openssl.c
ngx_event_pipe.h  ngx_event_timer.cngx_event.c  ngx_event_busy_lock.c
ngx_event_connect.h  ngx_event_openssl.h  ngx_event_posted.c  ngx_event_timer.h
ngx_event.h  ngx_event_busy_lock.h  ngx_event_mutex.c  ngx_event_pipe.c
ngx_event_posted.h
```

modules 目录中存放的源码实现了 Nginx 服务器支持的各类事件驱动模型：

```
# ls modules/
ngx_aio_module.c  ngx_epoll_module.c  ngx_kqueue_module.c
ngx_rtsig_module.c   ngx_win32_select_module.c  ngx_devpoll_module.c
```

```
ngx_eventport_module.c  ngx_poll_module.c    ngx_select_module.c
```

包括 AIO、epoll、kqueue、rtsig、Windows 32 位平台和 linux 平台的 select、/dev/poll、poll 等事件驱动模型。

modules 目录之外的其他源码主要实现了和事件驱动机制相关的数据结构的定义、初始化功能，完成事件接收、传递、管理的功能，以及事件驱动模型调用的功能等。

10.1.3　http 目录

该目录下的源码为 Nginx 服务器提供 Web 服务提供了主要的支持。该目录的结构安排和 event 目录相同：

```
modules                          ngx_http_file_cache.c       ngx_http_special_response.c
ngx_http.c                       ngx_http_header_filter_module.c     ngx_http_upstream.c
ngx_http.h                       ngx_http_parse.c            ngx_http_upstream.h
ngx_http_busy_lock.c             ngx_http_parse_time.c       ngx_http_upstream_round_robin.c
ngx_http_busy_lock.h             ngx_http_postpone_filter_module.c ngx_http_upstream_round_robin.h
ngx_http_cache.h                 ngx_http_request.c          ngx_http_variables.c
ngx_http_config.h                ngx_http_request.h          ngx_http_variables.h
ngx_http_copy_filter_module.c    ngx_http_request_body.c  ngx_http_write_filter_module.c
ngx_http_core_module.c    ngx_http_script.c    ngx_http_core_module.h  ngx_http_script.h
```

modules 目录中存放 HTTP 模块的实现源码，鉴于篇幅所限，笔者不在此详细列出该目录下的所有文件，大家可以自行浏览。在该目录下，我们从源码文件的命名上很容易分辨出它们实现的模块。几乎所有的 HTTP 标准模块和可选模块都在这里有对应的源码实现。它们的命名都遵循第 3 章中我们提到过的 Nginx 模块命名规则：以“ngx_http_”作为前缀、“_module”作为后缀，中间描述模块功能。

在该目录中的其他源码，实现了 Nginx 服务器提供 HTTP 服务需要依赖的数据结构定义、初始化功能，完成网络连接建立、管理、断开的功能，数据报文解析、服务器组管理的功能等。这一部分是我们在后面要学习的重点。

在本节中，我们学习了 Nginx 服务器源码解压目录的结构，对各目录下源码要实现的功能有了大致的了解。那么，从功能的角度来对 Nginx 源码进行划分和学习，又将是怎样的呢?

10.2　Nginx 源码的模块化结构

Nginx 服务器的源码是模块化设计的典范，我们在第 3 章中已经对此有所了解。我们可以人为地将 Nginx 的源码按照实现功能划分为 8 个模块。如图 10.1 所示。

10.2.1　公共功能

该部分的源码实现了 Nginx 各个模块依赖的公共基础，包括字符串处理、时间管理、脚本执行、文件读写、消息输出、锁机制等。如图 10.2 所示。

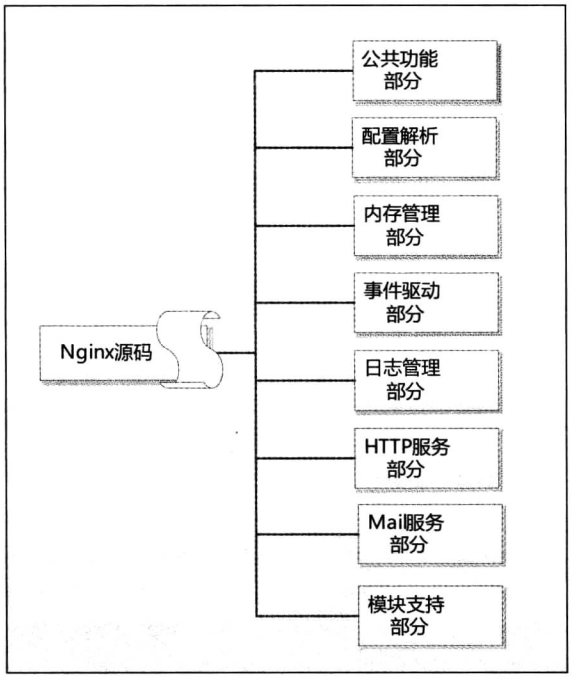

图 10.1　Nginx 源码功能结构示意图

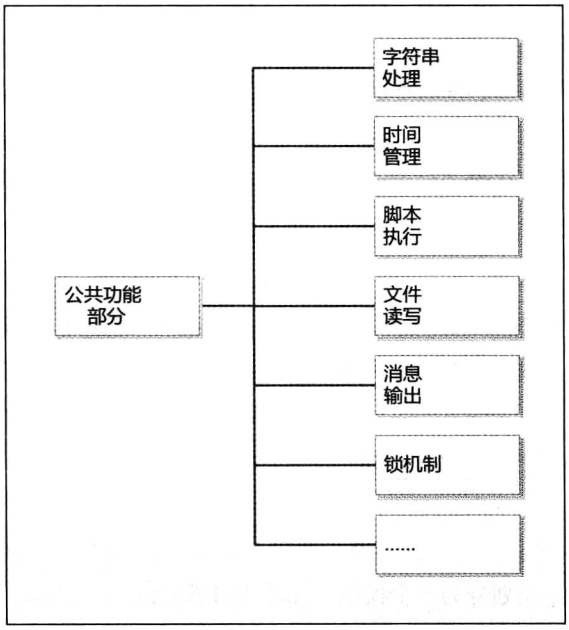

图 10.2　公共功能部分示意图

10.2.2　配置解析

该部分的源码主要实现了对配置文件的解析和处理，包括对配置文件的语法检查、正则表达式的支持、配置参数的解析和参数值的初始化等。如图 10.3 所示。

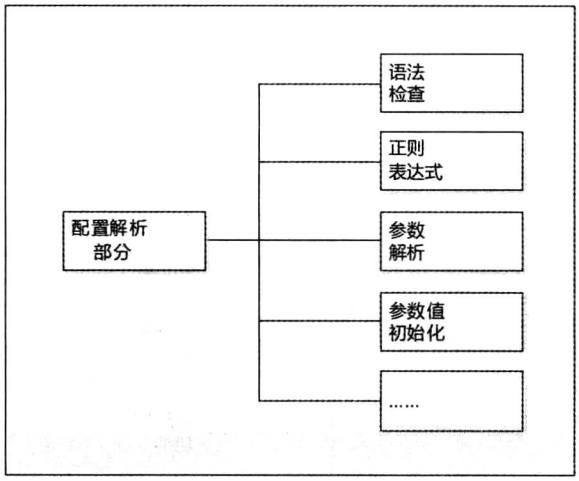

图 10.3　配置解析部分示意图

10.2.3　内存管理

该部分的源码主要实现了 Nginx 服务器对内存的管理，包括内存池的管理、共享内存的分配、缓存区的管理等。如图 10.4 所示。

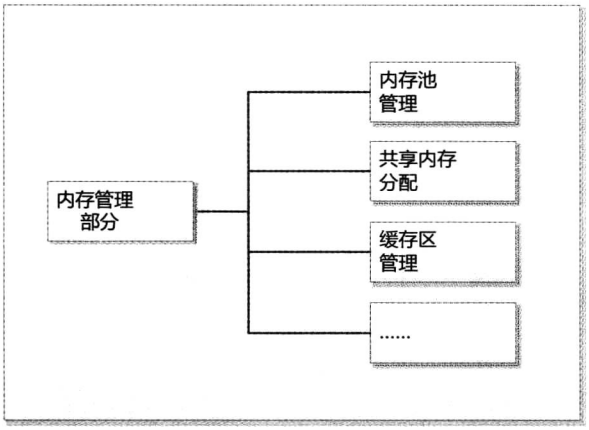

图 10.4　内存管理部分示意图

10.2.4　事件驱动

该部分的源码主要实现了 Nginx 服务器对各种事件驱动模型的支持，包括 Nginx 主进程的创建、工作进程的管理、信号的接收与处理、所有事件驱动模型的实现等。另外，一些高级输入/输出功能，如 Nginx 的异步输入/输出(AIO)、文件内存映射(Mmap)、TCP/IP 操作、散布读和聚集写(readv/writev) 等机制也都是在这部分源码中实现的。如图 10.5 所示。

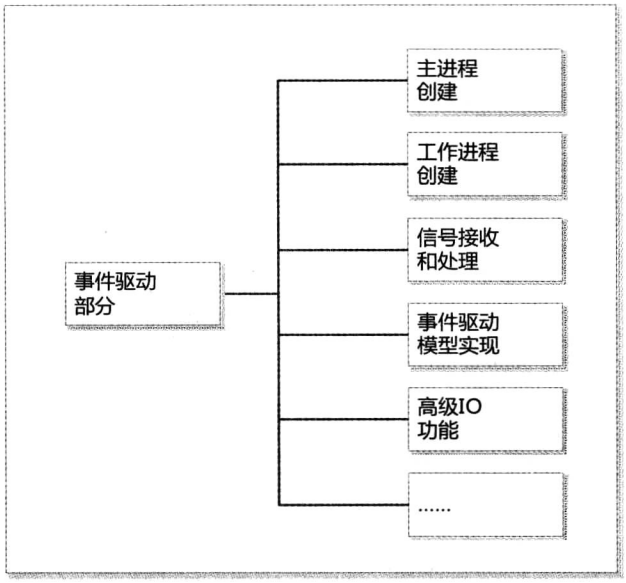

图 10.5　事件驱动部分示意图

10.2.5　日志管理

该部分的源码主要实现 Nginx 服务器的日志功能，包括错误日志（Error Log）产生和管理、任务日志（Access Log）产生和管理等。如图 10.6 所示。

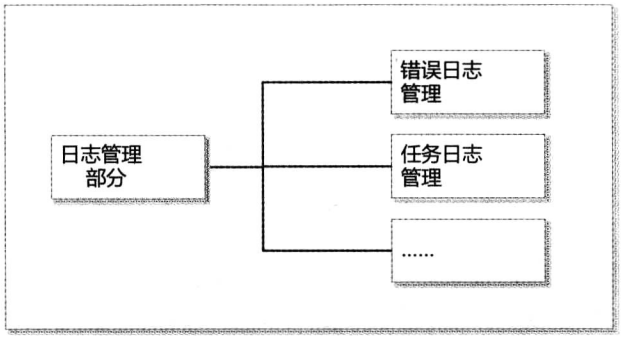

图 10.6　日志管理部分示意图

10.2.6　HTTP 服务

该部分的源码实现了 Nginx 服务器的主要功能——提供 Web 服务，包括客户端网络连接管理、客户端请求处理、虚拟主机管理、服务器组管理、服务器代理、服务器认证及访问权限管理等。如图 10.7 所示。

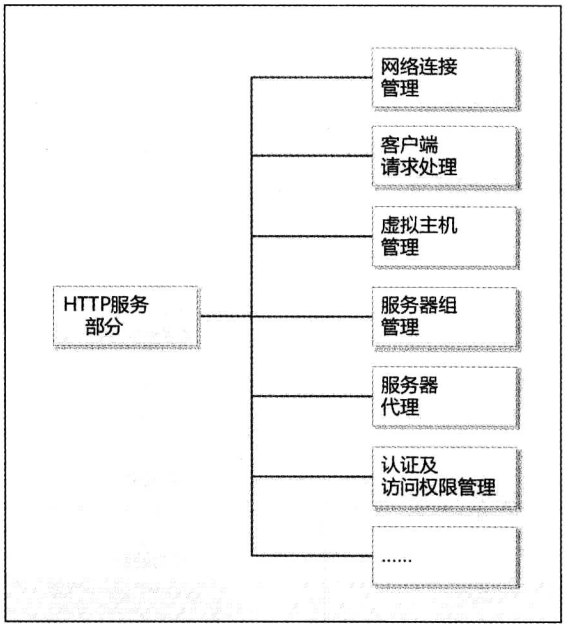

图 10.7　HTTP 服务部分示意图

10.2.7　Mail 服务

该部分的源码实现了 Nginx 服务器的邮件服务。邮件服务是 Nginx 服务器设计时要完成的主要功能，其在源码的设计结构上与 HTTP 服务部分的源码基本上相同，主要包括网络连接管理、虚拟主机管理、服务器代理等，并且增加了邮件协议的实现。如图 10.8 所示。

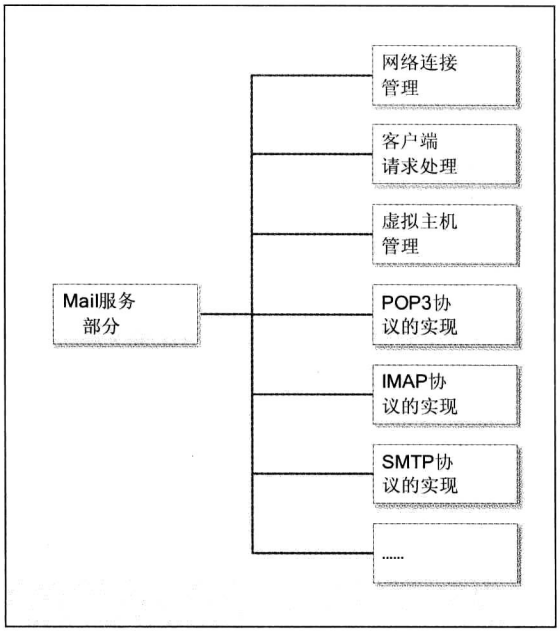

图 10.8　Mail 服务部分示意图

10.2.8　模块支持

大家都知道，Nginx 服务器将接收到的客户端请求依次经过不同的模块进行分析和处理，这些模块有些是标准的 HTTP 模块，有些是可选的 HTTP 模块，有些是 Mail 模块，还有些是第三方自定义的模块。该部分的源码主要实现对这些模块类型的定义，对模块进行初始化、管理和组织，通过回调函数调用模块，通过指令控制模块行为等。如图 10.9 所示。

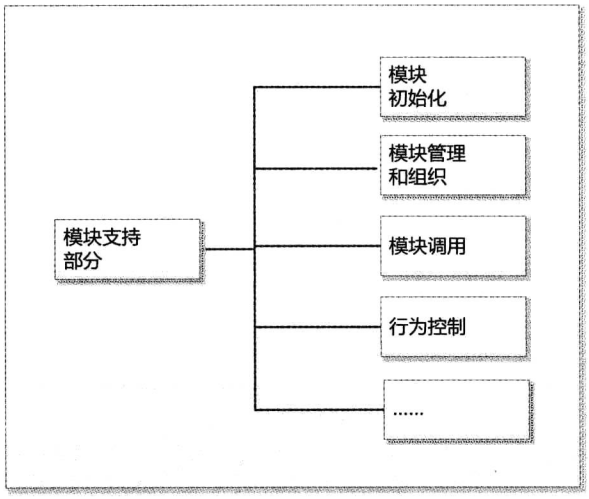

图 10.9　模块支持部分示意图

以上将 Nginx 源码按照功能实现人为地大致划分为八个部分。这八个部分的源码涵盖了 Nginx 服务器绝大多数功能和服务的实现，是理清 Nginx 源码结构的重要参考。

10.3　本章小结

在本章中，我们从源码目录结构和功能实现结构两个层次了解了 Nginx 服务器的源码组织结构。Nginx 源码的模块化设计，可谓思路清晰、构思精巧，是从事程序设计的人员参考和学习的榜样。正是由于 Nginx 源码的高质量实现，才造就了它稳定的性能和灵活的可扩展性，进而成就了其在 Web 服务器市场的异军突起之势。

从下一章开始，我们将深入到源码的具体实现，通过对重要源码片段的深入学习，在体验 Nginx 源码美妙的设计艺术的同时，对 Nginx 服务器的主要功能获得更深层的理解，同时也对前面几章中的重点内容进行回顾和复习。

第 11 章

Nginx 基本数据结构

从本章开始，我们学习 Nginx 源码的具体实现。学习体系错综复杂、功能服务实现丰富的 Nginx 服务器源码，应该从整个源码体系赖以存在的基本元素——数据结构开始学起。只有对常用的重要数据结构有所了解，才能减少在以后的学习过程中的困难。

Nginx 源码中涉及的数据结构非常丰富。几乎每一个模块的实现都会引入它特有的链表、队列或者树。为了突出重点，并且由于篇幅所限，我们不准备将所有的数据结构都罗列出来讲解，而是按照 Nginx 服务器提供功能的重要性，为大家分析一些最重要的数据结构。

11.1 ngx_module_s 结构体

该结构体是整个 Nginx 模块化架构最基本的数据结构体，它描述了 Nginx 程序中一个模块应该包含的基本属性。在 Nginx 1.2.6 的/nginx/src/core/ngx_conf_file.c 中定义了该结构体。

```
struct ngx_module_s {
ngx_uint_t          ctx_index;                          //所属分类标识
ngx_uint_t          index;                              //模块计数器
                                                        //以下预留成员暂未使用
ngx_uint_t          spare0;
ngx_uint_t          spare1;
ngx_uint_t          spare2;
ngx_uint_t          spare3;
ngx_uint_t          version;                            //模块版本
void               *ctx;                                //模块上下文
ngx_command_t      *commands;                           //模块支持的命令集
ngx_uint_t          type;                               //模块的种类
```

```
                                                                    //回调函数
ngx_int_t          (*init_master)(ngx_log_t *log);                  //主进程初始化时调用*/
ngx_int_t          (*init_module)(ngx_cycle_t *cycle);              //模块初始化时调用
ngx_int_t          (*init_process)(ngx_cycle_t *cycle);             //工作进程初始化时调用
ngx_int_t          (*init_thread)(ngx_cycle_t *cycle);              //线程初始化时调用
void               (*exit_thread)(ngx_cycle_t *cycle);              //线程退出时调用
void               (*exit_process)(ngx_cycle_t *cycle);             //退出工作进程时调用
void               (*exit_master)(ngx_cycle_t *cycle);              //退出主进程时调用
                                                                    //以下预留成员暂未使用
uintptr_t            spare_hook0;
uintptr_t            spare_hook1;
uintptr_t            spare_hook2;
uintptr_t            spare_hook3;
uintptr_t            spare_hook4;
uintptr_t            spare_hook5;
uintptr_t            spare_hook6;
uintptr_t            spare_hook7;
};
```

该结构体在初始化时，需要用到两个宏定义，其在该文件中也能找到：

```
#define NGX_MODULE_V1    0, 0, 0, 0, 0, 0, 1                        //前七个成员初始化
#define NGX_MODULE_V1_PADDING  0, 0, 0, 0, 0, 0, 0, 0               //后八个成员初始化
```

第一个用来初始化前面 7 个成员，第二个用来初始化后面的 8 个成员。

在该结构体中，笔者使用注释的形式为各个成员的用途添加了解释信息。在这个结构体中，有几个成员需要解释一下。

11.1.1　分类标识 ctx_index

Nginx 程序的模块分为 4 种：分别是 core、http、event 和 mail，每个模块在实现过程中使用的技术都不尽相同。“分类标识”就是用来表示该模块属于 4 种模块中的哪一类。

11.1.2　模块计数器 index

什么叫“模块计数器”呢？我们知道，Nginx 服务器的几乎所有功能都是以模块的形式出现的，并且这些模块可以在编译时添加或者卸载。Nginx 程序为了方便管理模块，定义了一个存放所有模块的数组，我们在/nginx/src/core/ngx_conf_file.h 文件中可以找到它：

```
extern ngx_module_t  *ngx_modules[];
```

其中，ngx_module_t 类型的定义在/nginx/src/core/ngx_core.h 文件中可以找到：

```
typedef struct ngx_module_s ngx_module_t;
```

注意

Nginx 程序习惯于使用“_s”后缀命名结构体，然后使用 typedef 定义一个同名并以“_t”作为后缀的名称作为此结构体的类型名。

编译后的 Nginx 目录中包含 objs 目录，该目录中的 ngx_modules.c 文件中包含了此版本 Nginx 快速编译后包括的所有模块的声明：

```
ngx_module_t *ngx_modules[] = {
    &ngx_core_module,                                    //Nginx 核心功能模块的声明
    &ngx_errlog_module,                                  //错误日志处理模块的声明
    &ngx_conf_module,                                    //配置解析模块的声明
    ......                                               //省略其他模块的声明
}
```

模块计数器 index 就是用于记录某一模块在声明时的序数，从 0 开始。比如，如果按照上面的源码来看，ngx_errlog_module 模块的 index 值为 1。

11.1.3　模块上下文

结构体中，指向模块上下文结构的指针类型为 void，这说明不同模块的模块上下文结构不同。实际情况确实是这样的，Nginx 服务器程序根据不同的模块类型定义了不同的模块上下文结构。

最常见到和使用的是 HTTP 类模块的模块上下文结构体 ngx_http_module_t：

```
typedef struct {
                                                         //解析 http 块时调用
    ngx_int_t   (*preconfiguration)(ngx_conf_t *cf);
    ngx_int_t   (*postconfiguration)(ngx_conf_t *cf);
                                                         //解析 main 块时调用
    void       *(*create_main_conf)(ngx_conf_t *cf);
    char       *(*init_main_conf)(ngx_conf_t *cf, void *conf);
                                                         //解析 server 块时调用
    void       *(*create_srv_conf)(ngx_conf_t *cf);
    char       *(*merge_srv_conf)(ngx_conf_t *cf, void *prev, void *conf);
                                                         //解析 location 块时调用
    void       *(*create_loc_conf)(ngx_conf_t *cf);
    char       *(*merge_loc_conf)(ngx_conf_t *cf, void *prev, void *conf);
} ngx_http_module_t;
```

结构中的所有成员都是指向回调函数的函数指针，这些函数在 HTTP 类模块初始化过程的不同阶段调用。*preconfiguration 指针指向的函数是在解析配置文件中的 http 块前调用，*postconfiguration 指针指向的函数是在完成 http 块解析后调用，*create_main_conf 指针指向的函数在初始化 http 块之前调用，*init_main_conf 指针指向的函数在初始化 http 块时调用，*create_srv_conf 指针指向的函数在初始化 server 块之前调用，*merge_srv_conf 指针指向的函数实现合并 server 块和 http 块中相同指令的配置，*create_loc_conf 指针指向的函数在初始化 location 块之前调用，*merge_loc_conf 指针指向的函数实现合并 location 块和 server 块中相同指令的配置。

11.1.4　回调函数

在 ngx_module_s 结构体中定义了 7 个函数指针，分别指向该模块自定义的回调函数。这些回调函数分别在主进程初始化、模块初始化、工作进程初始化、线程初始化、线程退出、工作进程退出和主进程退出的时候被调用。如果该模块需要在发生这些行为时执行特定的功能，就可以通过这些回调函数指针注册一个回调函数接口来实现。

11.2　ngx_command_s 结构体

该结构体描述了模块支持的指令，负责解析配置文件的指令，一个指令对应一个配置指令。我们在/nginx/src/core/ngx_conf_file.h 文件中可以找到该结构体的定义：

```
struct ngx_command_s {
    ngx_str_t        name;                                  //模块的名称
    ngx_uint_t       type;
    char *(*set)(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);//指令的执行函数
    ngx_uint_t       conf;
    ngx_uint_t       offset; //在父指令块中的偏移
    void             *post;                                 //读取配置文件时可能使用的指令
};
```

初始化一个空指令需要用到一个宏定义，在该文件中可以找到它：

```
#define ngx_null_command  { ngx_null_string, 0, NULL, 0, 0, NULL }
```

在 ngx_module_s 结构体中，定义了指向 ngx_command_t 的指针，ngx_command_t 的定义我们在/nginx/src/core/ngx_core.h 文件中可以找到：

```
typedef struct ngx_command_s  ngx_command_t;
```

11.2.1　type 成员

type 成员实际上由 32 位的无符号整型数组成，用来表示该指令在配置文件中的合法位置和可接受参数个数的标示符集合，前面 16 位表示指令的位置，后面 16 位表示参数个数。各位代表的含义如图 11.1 所示。

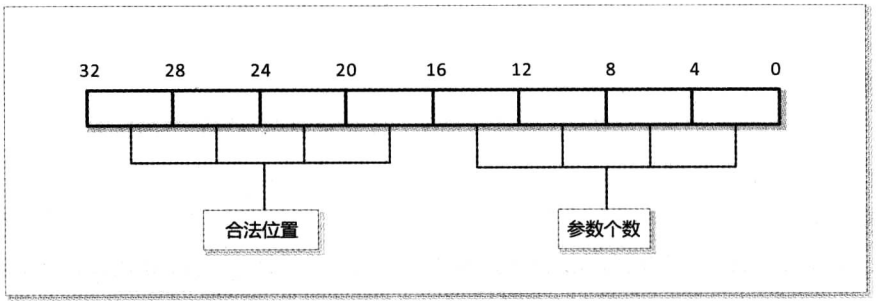

图 11.1　type 成员结构示意图

1. 表示合法位置的宏定义

Nginx 预先定义好的表示指令合法位置的宏定义根据模块所属的类型不同，分布在不同的文件中。

属于 core 类的模块，它支持的指令的合法位置以及寻址方式的宏定义可以在/nginx/src/core/ngx_conf_file.h 文件中找到：

```
#define NGX_DIRECT_CONF   0x00010000                      //直接寻址方式
#define NGX_MAIN_CONF     0x01000000                      //在全局块中（寻址）
#define NGX_ANY_CONF      0x0F000000
```

属于 http 类的模块，它支持的指令的合法位置的宏定义可以在/nginx/src/http/ngx_http_config.h 文

件中找到：

```
#define NGX_HTTP_MAIN_CONF  0x02000000                    //在 http 块中（寻址）
#define NGX_HTTP_SRV_CONF  0x04000000                     //在 server 块中（寻址）
#define NGX_HTTP_LOC_CONF  0x08000000                     //在 location 块中（寻址）
#define NGX_HTTP_UPS_CONF  0x10000000                     //在 upstream 块中（寻址）
#define NGX_HTTP_SIF_CONF   0x20000000
#define NGX_HTTP_LIF_CONF   0x40000000
#define NGX_HTTP_LMT_CONF  0x80000000
```

注意

这里的 server 块、location 块和 upstream 块的父指令块是 http 块。

属于 event 类的模块，它支持的指令的合法位置的宏定义可以在/nginx/src/event/ngx_event.h 文件中找到：

```
#define NGX_EVENT_CONF    0x02000000                      //在 event 块中（寻址）
```

属于 mail 类的模块，它支持的指令的合法位置的宏定义可以在/nginx/src/mail/ngx_mail.h 文件中找到：

```
#define NGX_MAIL_MAIN_CONF    0x02000000                  //在 mail 块中（寻址）
#define NGX_MAIL_SRV_CONF     0x04000000                  //在 server 块中（寻址）
```

注意

这里的 server 块的父指令块是 mail 块。

上面的这些宏定义，从名字上可以直接判断出其代表的意义，笔者不再赘述。这些宏可以构成指令类型 type 的前 16 位标示符。

2. 表示参数个数的宏定义

在/nginx/src/core/ngx_conf_file.h 文件中可以找到 Nginx 预先定义好的表示指令可接受参数个数的宏定义：

```
                                                          //指令接受参数的个数
#define NGX_CONF_NOARGS     0x00000001                    //不接受参数
#define NGX_CONF_TAKE1      0x00000002                    //接受 1 个参数
#define NGX_CONF_TAKE2      0x00000004                    //接受 2 参数
#define NGX_CONF_TAKE3      0x00000008                    //接受 3 个参数
#define NGX_CONF_TAKE4      0x00000010                    //接受 4 个参数
#define NGX_CONF_TAKE5      0x00000020                    //接受 5 个参数
#define NGX_CONF_TAKE6      0x00000040                    //接受 6 个参数
#define NGX_CONF_TAKE7      0x00000080                    //接受 7 个参数
#define NGX_CONF_TAKE12     (NGX_CONF_TAKE1|NGX_CONF_TAKE2)
                                                          //接受 1 个或 2 个参数
#define NGX_CONF_TAKE13     (NGX_CONF_TAKE1|NGX_CONF_TAKE3)
                                                          //接受 1 个或 3 个参数
#define NGX_CONF_TAKE23     (NGX_CONF_TAKE2|NGX_CONF_TAKE3)
                                                          //接受 2 个或 3 个参数
#define NGX_CONF_TAKE123    (NGX_CONF_TAKE1|NGX_CONF_TAKE2|NGX_CONF_TAKE3)
                                                          //接受 1 个、2 个或 3 个参数
#define NGX_CONF_TAKE1234   (NGX_CONF_TAKE1|NGX_CONF_TAKE2|NGX_CONF_TAKE3   \
```

```
                          |NGX_CONF_TAKE4)
                                                    //接受 1 个、2 个、3 个或 4 个参数
#define NGX_CONF_ARGS_NUMBER  0x000000ff            //可接受的参数个数的上限
#define NGX_CONF_1MORE        0x00000800            //接受 1 个或 1 个以上参数
#define NGX_CONF_2MORE        0x00001000            //接受 2 个或 2 个以上参数
#define NGX_CONF_MULTI         0x00000000
                                                    //指令形成的块域
#define NGX_CONF_BLOCK         0x00000100           //形成指令块域的指令
#define NGX_CONF_FLAG          0x00000200           //可接受 boolean 参数
#define NGX_CONF_ANY           0x00000400
```

- NGX_CONF_NOARGS，表示该指令不接受任何参数。
- NGX_CONF_TAKE1 ~ NGX_CONF_TAKE7，表示该指令接受 1 ~ 7 个参数。
- NGX_CONF_TAKE12，表示该指令接收 1 个或者 2 个参数，注意不是接受 12 个参数。类似的宏还有 NGX_CONF_TAKE13、NGX_CONF_TAKE23、NGX_CONF_TAKE123 和 NGX_CONF_TAKE1234 等。
- NGX_CONF_1MORE，表示该指令至少接受 1 个参数，类似的宏还有 NGX_CONF_2MORE。
- NGX_CONF_BLOCK，表示该指令形成块域，比如我们前面学习的 http 块、server 块等。
- NGX_CONF_FLAG，表示该指令可以接受布尔参数（配置中一般使用 on 和 off）。

11.2.2　函数指针 set

set 指针指向一个指令函数，该函数的主要作用是从 Nginx 配置文件中把该指令的参数转换为合适的数据结构类型，并将转换后的值保存到 Nginx 模块的配置结构体（ngx_conf_t）中。

当然，对于简单的配置指令来说，可能并不需要将参数转换为值进行保存，只是在 set 指针指向的指令函数中完成一些简单的操作。比如，在处理错误日志时，error_log 指令就是通过 set 指针调用 ngx_error_log()函数输出一条日志记录的。

11.2.3　conf 和 offset

这两个成员都是用来配置偏移量的。我们知道 set 指针指向的函数一般要转换指令的参数并将转换后的值保存到配置结构体中，offset 就是用来指定该值保存在配置结构体中的具体位置的。该值一般是 Nginx 程序通过调用 offsetof()函数计算得出的。

conf 成员用来指定配置文件结构体中各其他类型指令配置结构体在该结构体中的偏移量。Nginx 程序通过宏定义的方式定义了可以赋值给 conf 成员的偏移量。在/nginx/src/http/ngx_http_conf.h 文件中可以找到以下宏定义：

```
#define NGX_HTTP_MAIN_CONF_OFFSET  offsetof(ngx_http_conf_ctx_t, main_conf)
                                                    //指令在 http 全局块中的偏移
#define NGX_HTTP_SRV_CONF_OFFSET   offsetof(ngx_http_conf_ctx_t, srv_conf)
                                                    //指令在 http 的 server 块中的偏移
#define NGX_HTTP_LOC_CONF_OFFSET   offsetof(ngx_http_conf_ctx_t, loc_conf)
                                                    //指令在 http 的 location 块中的偏移
```

在/nginx/src/mail/ngx_mail.h 文件中可以找到以下宏定义：

```
#define NGX_MAIL_MAIN_CONF_OFFSET  offsetof(ngx_mail_conf_ctx_t, main_conf)
                                                    //指令在 mail 全局块中的偏移
#define NGX_MAIL_SRV_CONF_OFFSET   offsetof(ngx_mail_conf_ctx_t, srv_conf)
                                                    //指令在 mail 的 server 块中的偏移
```

11.3　3 个基本模块的指令集数组结构

在 Nginx 服务器程序中，不同的模块基于 ngx_command_t 结构体定义了各自的指令集数组。为了学习方便，我们将 http、event 和 mail 模块的指令集数组结构归纳一下。对于其他模块，比如前面章节中提到的 http 类 ngx_http_gzip_module 模块、ngx_http_rewrite_module 模块和 ngx_http_proxy_module 模块等，它们的指令集数组结构通常定义在各自源码文件中，结构的命名方式为 ngx_http_*module*_commands，限于篇幅，就不在这里一一列举了。

11.3.1　http 模块

http 模块定义的指令集数组在/nginx/src/http/ngx_http.c 文件中可以找到：

```
static ngx_command_t  ngx_http_commands[] = {
    { ngx_string("http"),                                   //http 指令结构
      NGX_MAIN_CONF|NGX_CONF_BLOCK|NGX_CONF_NOARGS,         //寻址位置和参数情况*/
      ngx_http_block,                                       //http 块的执行函数
      0,
      0,
      NULL },
      ngx_null_command
};
```

当 Nginx 服务器程序解析配置文件遇到 http 指令时，将会依据该指令集数组完成对这条指令的解析工作。进入 http 块后，对其他各条指令的解析，将依据相关模块的指令集数组完成解析工作。我们将在下一章学习解析配置文件的源码细节。

在这里需要提到的一点是，upsteam 块单独定义了自己的指令集数组，从源码来看它也被包含在了 http 块的实现中，我们在/nginx/src/http/ngx_http_upsteam.c 文件中可以找到它：

```
static ngx_command_t  ngx_http_upstream_commands[] = {
    { ngx_string("upstream"),                           //upstream 指令结构
      NGX_HTTP_MAIN_CONF|NGX_CONF_BLOCK|NGX_CONF_TAKE1,    //寻址位置和参数情况
      ngx_http_upstream,                                //upstream 块的执行函数
      0,
      0,
      NULL },
    { ngx_string("server"),                             //upstream 中 server 指令结构
      NGX_HTTP_UPS_CONF|NGX_CONF_1MORE,                 //寻址位置和参数情况
      ngx_http_upstream_server,                         //upstream 中 server 块的执行函数
      NGX_HTTP_SRV_CONF_OFFSET,                         //server 块在 upstream 中的偏移*/
      0,
      NULL },
```

```
    ngx_null_command
};
```

11.3.2 event 模块

这一模块的指令集与上一模块的指令集在处理方式上基本相同。由于 event 模块中不会再嵌套其他模块，因此定义的指令集包括两部分，一部分是对 events 指令的定义，另一部分是对 event 模块内指令集的定义。

在解析配置文件时，遇到“events{”指令，Nginx 服务器程序依据以下指令集数组对该条指令进行解析：

```
static ngx_command_t  ngx_events_commands[] = {
    { ngx_string("events"),                                    //events 指令结构
      NGX_MAIN_CONF|NGX_CONF_BLOCK|NGX_CONF_NOARGS,            //寻址位置和参数情况
      ngx_events_block,                                        /" events 块的执行函数 "/
      0,
      0,
      NULL },
      ngx_null_command
};
```

这一部分和 http 模块的定义方式完全相同。配置在 event 模块中的各条指令依据以下指令集数组采取相应的解析操作：

```
static ngx_command_t  ngx_event_core_commands[] = {
    { ngx_string("worker_connections"),          //event 块中 worker_connections 指令结构
      NGX_EVENT_CONF|NGX_CONF_TAKE1,             //寻址位置和参数情况
      ngx_event_connections,                     //worker_connections 指令的执行函数
      0,
      0,
      NULL },
    { ngx_string("connections"),                 //event 块中 connections 指令结构
      NGX_EVENT_CONF|NGX_CONF_TAKE1,             //寻址位置和参数情况*/
      ngx_event_connections,                     //connections 指令的执行函数
      0,
      0,
      NULL },
    { ngx_string("use"),                         //event 块中 use 指令结构
      NGX_EVENT_CONF|NGX_CONF_TAKE1,             //寻址位置和参数情况*/
      ngx_event_use,                             //use 指令的执行函数
      0,
      0,
      NULL },
    { ngx_string("multi_accept"),                //event 块中 multi_accept 指令结构
      NGX_EVENT_CONF|NGX_CONF_FLAG,              //寻址位置和参数情况*/
      ngx_conf_set_flag_slot,                    //执行 multi_accept 指令的函数
      0,
      offsetof(ngx_event_conf_t, multi_accept),
```

```
    NULL },
  { ngx_string("accept_mutex"),                    //event 块中 accept_mutex 指令结构
    NGX_EVENT_CONF|NGX_CONF_FLAG,                  //寻址位置和参数情况*/
    ngx_conf_set_flag_slot,
    0,
    offsetof(ngx_event_conf_t, accept_mutex),
    NULL },
  { ngx_string("accept_mutex_delay"),              //event 块中 accept_mutex_delay 指令结构
    NGX_EVENT_CONF|NGX_CONF_TAKE1,  //寻址位置和参数情况*/
    ngx_conf_set_msec_slot,
    0,
    offsetof(ngx_event_conf_t, accept_mutex_delay),
    NULL },
  { ngx_string("debug_connection"),                //event 块中 debug_connection 指令结构
    NGX_EVENT_CONF|NGX_CONF_TAKE1,                 //寻址位置和参数情况*/
    ngx_event_debug_connection,
    0,
    0,
    NULL },
    ngx_null_command
};
```

数组中的每个元素与我们在前面相关章节介绍的 Nginx 配置中 event 块参数配置是对应的。这一部分的定义相当于 http 类模块各自定义的指令集数组。

11.3.3　mail 模块

这一模块在指令集的定义形式上与 http 模块是完全相同的。对 mail 指令的解析处理依据/nginx/src/mail/ngx_mail.c 文件中的以下指令集数组进行：

```
static ngx_command_t  ngx_mail_commands[] = {
  { ngx_string("mail"),                                    //mail 块的指令结构
    NGX_MAIN_CONF|NGX_CONF_BLOCK|NGX_CONF_NOARGS,          //寻址位置和参数情况*/
    ngx_mail_block,                                        //执行 mail 指令的函数
    0,
    0,
    NULL },
  { ngx_string("imap"),                                    //imap 块的指令结构
    NGX_MAIN_CONF|NGX_CONF_BLOCK|NGX_CONF_NOARGS,          //寻址位置和参数情况*/
    ngx_mail_block,                                        //执行 imap 指令的函数
    0,
    0,
    NULL },
    ngx_null_command
};
```

mail 模块中的其他指令依据它们各自模块定义的指令集数组进行解析。

11.4 ngx_ pool_s 结构体

Nginx 对内存的管理是通过自身实现的内存池结构来完成的。内存池实际上是一个链表结构，由多块小的内存块通过链表的形式组成。ngx_pool_s 结构体用来描述内存池的管理分配。结构体中可以包括内存数据块、大体积内存数据块指针、用于内存回收的 cleanup 指针等重要链表结构。可以在/nginx/src/core/ngx_palloc.h 中找到 ngx_pool_s 结构体的完整定义：

```
struct ngx_pool_s {
    ngx_pool_data_t       d;
    size_t                max;
    ngx_pool_t            *current;
    ngx_chain_t           *chain;
    ngx_pool_large_t      *large;
    ngx_pool_cleanup_t    *cleanup;
    ngx_log_t             *log;
};
```

- d，表示内存池中内存数据块的结构。
- max，表示内存数据块可用于分配的最大内存。
- *current，表示当前内存数据块。
- *chain，用来指向一个 ngx_chain_t 结构。
- *large，指向大内存数据块链表结构。
- *cleanup，指向用于内存回收的链表结构。
- *log，指向日志配置结构，其中包含了 Nginx 日志存储管理的相关信息。

该结构体主要涉及内存池的分配、内存的管理和内存的回收等操作。关于该结构体，我们重点介绍一下涉及的几个重要的链表结构。

11.4.1 ngx_pool_data_t 结构体

该结构体中 d 成员的结构体类型 ngx_pool_data_t 描述了该内存块中数据的存储结构，也就是每个链表节点具体存储数据的结构，它们形成了 Nginx 内存池的主体链表结构。其定义在文件/nginx/src/core/ngx_palloc.h 中可以找到：

```
typedef struct {
    u_char               *last;
    u_char               *end;
    ngx_pool_t           *next;
    ngx_uint_t           failed;
} ngx_pool_data_t;
```

- *last，保存内存池中内存分配指针的当前位置。每次 Nginx 程序从内存池中申请内存时，从该指针保存的位置开始划分出请求的内存大小，并更新该指针到新的位置。
- *end，保存内存池的结束位置。也就是链表的末端。

- *next，内存池由多块内存区域组成，该指针用于在当前内存区域中保存下一内存区域的位置。也就是链表中当前节点用来指向下一节点的指针。
- failed，内存池分配失败的次数。

11.4.2　ngx_pool_large_s 结构体

ngx_pool_s 结构体中的*large 成员指向一个链表。ngx_pool_large_s 结构体的定义在/nginx/src/core/ngx_palloc.h 中可以找到：

```
struct ngx_pool_large_s {
    ngx_pool_large_t     *next;
    void                 *alloc;
};
```

该链表的实现非常简单，我们主要说明一下 Nginx 程序为什么要使用这样一个链表。

ngx_pool_s 结构体中的 max 成员规定了一个内存数据块的大小。当 Nginx 程序向内存池申请内存时，如果申请的空间小于 max 大小，就申请在 ngx_pool_s 结构体中 d 的内存数据块；如果申请的空间大于 max 大小，就需要申请 ngx_pool_s 结构体中*large 所指的大体积内存区。注意，ngx_pool_large_s 结构体中的*alloc 成员的类型为 void，因此，位于大体积内存区上的数据可以是任意自定义类型。

11.4.3　ngx_pool_cleanup_s 结构体

ngx_pool_s 结构体中的* cleanup 成员指向用于内存回收的链表。ngx_pool_cleanup_s 结构体的定义在文件/nginx/src/core/ngx_palloc.h 中也可以找到：

```
struct ngx_pool_cleanup_s {
    ngx_pool_cleanup_pt  handler;
    void                 *data;
    ngx_pool_cleanup_t   *next;
};
```

该链表节点的结构也不复杂，*data 指针存储了任意类型的数据，这些数据是将要被清除的；*next 指针形成了链表结构，指向下一个将要被清除的内存数据；handler 是对*data 进行清除操作的数据处理函数，类型 ngx_pool_cleanup_pt 的定义也在同一文件中：

```
typedef void (*ngx_pool_cleanup_pt)(void *data);
```

11.5　Nginx socket 相关的数据结构

与 Nginx 服务器网络套接字 socket 操作有关的基本数据结构有三个结构体：分别是 ngx_listening_s 结构体、ngx_http_conf_addr_t 结构体和 ngx_http_conf_port_t 结构体。我们在本章首先学习这三个数据结构的重要成员组成，在本书后面的相关章节，会详细学习 Nginx 服务器网络套接字 socket 的操作。

11.5.1　ngx_listening_s 结构体

该结构体用于描述 Nginx 服务器在运行过程中使用的网络套接字 socket 的详细属性信息。每个这样的结构体会对应 Nginx 配置中配置的多个虚拟主机，以及对应一个 ngx_connection_s 结构体。该结

构体的完整定义在/nginx/src/core/ngx_connection.h 文件中可以找到，成员比较复杂，我们只介绍与后面学习相关的重要成员。

```
struct ngx_listening_s {
    ngx_socket_t        fd;
    struct sockaddr  *sockaddr;
    socklen_t           socklen;                          //sockaddr 长度
    ......
    int                 rcvbuf;
    int                 sndbuf;
    ......
    ngx_connection_handler_pt  handler;
    void                *servers;
    ......
    ngx_connection_t *connection;
    unsigned            open:1;
    unsigned            remain:1;
    unsigned            ignore:1;
    unsigned            bound:1;                          //绑定标志位
    unsigned            inherited:1;                      //继承标志位
    unsigned            nonblocking_accept:1;
    unsigned            listen:1;
    unsigned            nonblocking:1;
    unsigned            shared:1;                         //进程或线程共享标志位
    unsigned            addr_ntop:1;
};
```

- fd，存放打开的 socket 描述符。
- *sockaddr，存放 socket 路径的相关信息。
- socklen_t，保存当前 socket 信息的长度。
- rcvbuf 和 sndbuf，接收缓存区和发送缓存区的长度。
- handler，指向处理函数的函数指针，ngx_connection_handler_pt 定义在/nginx/src/core/ngx_core.h 文件中：

```
typedef void (*ngx_connection_handler_pt)(ngx_connection_t *c);
```

- *servers，保存所有虚拟主机的相关信息。
- *connection，指向该 socket 对应的 ngx_connection_s 结构。在 ngx_connection_s 结构体中也有一个成员指向该 socket 结构。
- 后面类型为 unsigned 的变量是一些标志位，用来标识当前 socket 的状态。

11.5.2　ngx_http_conf_port_t 结构体

该结构体用于描述监听端口的配置信息，定义在文件/nginx/src/http/ngx_http_core_module.h 中：

```
typedef struct {
    ngx_int_t                family;
```

```
    in_port_t                   port;
    ngx_array_t                 addrs;
} ngx_http_conf_port_t;
```

- family 指定协议簇，Nginx 常用的有 AF_INET、AF_INET64 和 AF_UNIX。
- port 存放监听端口。
- addrs 是存放在该端口上所有监听地址的数组。每个数组元素都是一个 ngx_http_conf_addr_t 结构体。

11.5.3　ngx_http_conf_addr_t 结构体

在学习 ngx_http_conf_port_t 结构体时，我们将监听端口相同的地址存放在一个数组中，数组的每个监听地址配置信息都存放在一个 ngx_http_conf_addr_t 结构体中，包括监听的所有 server 块的 ngx_http_core_srv_conf_t 结构，以及 hash、wc_head 和 wc_tail 这些 hash 结构。该结构体还保存了以 server name 为 key，ngx_http_core_srv_conf_t 为 value 的哈希表，用于快速查找对应虚拟主机的配置信息。该结构体仍然定义在文件/nginx/src/http/ngx_http_core_module.h 中：

```
typedef struct {
    ngx_http_listen_opt_t      opt;
    ngx_hash_t                 hash;
    ngx_hash_wildcard_t        *wc_head;
    ngx_hash_wildcard_t        *wc_tail;
    //the default server configuration for this address:port
    ngx_http_core_srv_conf_t  *default_server;
    ngx_array_t                servers;         //ngx_http_core_srv_conf_t 结构数组
} ngx_http_conf_addr_t;
```

- ngx_http_listen_opt_t 重放监听 socket 的配置信息。
- hash、*wc_head、*wc_tail 完成了对哈希表的定义。该哈希表以 server_name 作为键（key），以 ngx_http_core_srv_conf_t 结构为值（value）。其中，hash 哈希表中的 server_name 不包括通配符；*wc_head 指向的哈希表中，server_name 包含前缀通配符；*wc_ tail 指向的哈希表中，server_name 包含后缀通配符。

如果我们的 Nginx 服务器支持正则表达式，则该结构体中还包括另外两个成员：

```
#if (NGX_PCRE)
    ngx_uint_t                 nregex;
    ngx_http_server_name_t     *regex;
#endif
```

11.6　ngx_event_s 结构体

该结构体存储了网络连接 IO（事件）状态的详细信息。在 Nginx 中，所有的 ngx_event_s 结构体组成了两个全局链表：一个完成连接读操作的描述，另一个完成连接写操作的描述。该结构体定义在文件/nginx/src/event/ngx_event.h 中：

```
struct ngx_event_s {
    void            *data;                          //事件上下文数据
    unsigned         write:1;
    unsigned         accept:1;
    //used to detect the stale events in kqueue, rtsig, and epoll
    unsigned         instance:1;
    /*
     * the event was passed or would be passed to a kernel;
     * in aio mode - operation was posted.
    unsigned         active:1;
    unsigned         disabled:1;
    unsigned         ready:1;                        //在AIO模式下标示是否有请求事件处理
    unsigned         oneshot:1;
    unsigned         complete:1;                     //在AIO模式下标示请求事件是否处理完成
    unsigned         eof:1;
    unsigned         error:1;
    unsigned         timedout:1;
    unsigned         timer_set:1;
    unsigned         delayed:1;
    unsigned         read_discarded:1;
    unsigned         unexpected_eof:1;
    unsigned         deferred_accept:1;
    unsigned         pending_eof:1;
    ngx_uint_t       index;
    ngx_log_t        *log;
    ngx_rbtree_node_t   timer;
    unsigned         closed:1;
                                                     //以下标志用于判别工作进程的工作状态
    unsigned         channel:1;
    unsigned         resolver:1;
    unsigned         locked:1;
    unsigned         posted_ready:1;
    unsigned         posted_timedout:1;
    unsigned         posted_eof:1;
                                                     //形成双向链表结构
    ngx_event_t      *next;
    ngx_event_t      **prev;
};
```

结构体中定义了大量标志位用来标示网络连接 IO 的属性，通过变量的命名可以了解大多数标示的作用，比较难理解的笔者为其添加了注释。data 变量定义为 void *类型，它可以指向任意类型的数据结构，next 和 prev 变量分别指向链表的后一个节点和前一个节点。

11.7 ngx_connection_s 结构体

该结构体用于描述一个网络连接。我们已经知道，Nginx 服务器使用事件驱动模型来处理网络连接请求。每当 Nginx 服务器产生一个新的 socket 时，就会构造一个 ngx_connection_s 结构体，将该 socket

的属性和数据信息保存下来。该结构体的完整定义在/nginx/src/core/ngx_connection.h 文件中可以找到，其中包含的成员较多，我们仅对其中重要的成员进行介绍。

```
struct ngx_connection_s {
    void                *data;
    ngx_event_t         *read;
    ngx_event_t         *write;
```

- *data，用来关联其他的 ngx_connection_s 结构体。
- *read，设置该连接的读事件。
- *write，设置该连接的写事件。

```
    ngx_socket_t        fd;
    ngx_recv_pt         recv;
    ngx_send_pt         send;
    ngx_recv_chain_pt   recv_chain;
    ngx_send_chain_pt   send_chain;
    ngx_listening_t     *listening;
    off_t               sent;
    ngx_log_t           *log;
    ngx_pool_t          *pool;
    struct sockaddr     *sockaddr;
    socklen_t           socklen;
    ngx_str_t           addr_text;
    struct sockaddr     *local_sockaddr;
...
}
```

前面这几个参数与事件驱动模型有关。这里的参数用于设置网络连接相关的属性，重点需要理解下面几个参数：

- fd，用于设置连接 socket 的 socket 描述字。
- recv 和 recv_chain，指向完成（批量）数据接收函数的函数指针。
- send 和 recv_chain，指向完成（批量）数据发送函数的函数指针。

这四个函数指针的定义在不同的系统平台中是有差异的。我们来讨论一下 Nginx 程序是如何处理这种差异的。

这四个函数指针在 Unix/Linux 平台上的定义我们可以在/nginx/os/unix/ngx_os.h 文件中找到：

```
typedef ssize_t (*ngx_recv_pt)(ngx_connection_t *c, u_char *buf, size_t size);
typedef ssize_t (*ngx_recv_chain_pt)(ngx_connection_t *c, ngx_chain_t *in);
typedef ssize_t (*ngx_send_pt)(ngx_connection_t *c, u_char *buf, size_t size);
typedef ngx_chain_t *(*ngx_send_chain_pt)(ngx_connection_t *c, ngx_chain_t *in,
off_t limit);
```

可以看到，成员 recv 和 send 与成员 recv_chain 和 recv_chain 指向的函数主要区别在于它们处理的数据类型有所不同。在程序中这四个成员的赋值是在/nginx/event/ngx_event.h 文件中的宏定义：

```
#define ngx_recv            ngx_io.recv
#define ngx_recv_chain      ngx_io.recv_chain
```

```
#define ngx_udp_recv          ngx_io.udp_recv
#define ngx_send              ngx_io.send
#define ngx_send_chain        ngx_io.send_chain
```

而其中 ngx_io 的类型为 ngx_os_io_t 结构体，其定义也在/nginx/os/unix/ngx_os 文件中：

```
typedef struct {
    ngx_recv_pt          recv;
    ngx_recv_chain_pt    recv_chain;
    ngx_recv_pt          udp_recv;
    ngx_send_pt          send;
    ngx_send_chain_pt    send_chain;
    ngx_uint_t           flags;
} ngx_os_io_t;
```

从上面宏定义和 ngx_os_io_t 结构体定义的文件以及定义内容来看，大家应该可以猜测到，ngx_io 实际上是 Nginx 程序中统一的 IO 结构，而 ngx_os_io_t 则是针对不同系统平台分别定制的特定的 IO 结构。分析到这里，大家是不是觉得很有一点面向对象的感觉呢？Nginx 程序确实是通过这样的方式来处理因平台系统不同造成的网络 IO 差异的。

我们回到 ngx_connection_s 结构体，继续学习其他成员。

- *listening，用于设置该网路连接的 socket 监听，它对应一个 ngx_listening_s 结构体，而该 ngx_listening_s 结构体中的*connection 成员又将指向该 ngx_connection_s 结构体。
- *sent 用于设置该连接已发送或者已接收数据的数量。
- *log 用于存放 Nginx 服务器的日志配置。
- *pool 指向 Nginx 程序建立的内存池的首地址。

这四个参数用于设置接收到的对端 socket 的地址属性和本地监听 socket 的地址属性。

- *sockaddr 指向保存对端监听 socket 的地址结构。
- socklen 记录对端 socket 的信息长度。
- addr_text 存放 socket 地址。
- *local_sockaddr 指向保存本地监听 socket 的地址结构。该信息来自 ngx_listening_t 结构体，最初来源于配置文件。

以上介绍的成员是该结构体中的主要成员，在本书后面的学习中会多次遇到。该结构体与 ngx_event_s 结构体是紧密相关的，许多成员的赋值都来自于 ngx_event_s 结构体中对应的成员。究其原因，就是 Nginx 服务器使用了事件驱动机制来处理网络请求。

11.8　ngx_cycle_s 结构体

该结构体是 Nginx 程序启动过程中使用的主要结构体。Nginx 程序启动的核心过程基本上是围绕该结构体开始的。我们在/nginx/src/core/ngx_cycle.h 文件中可以找到该结构体的完整定义：

```
struct ngx_cycle_s {
    void                  ****conf_ctx;
```

```
    ngx_pool_t              *pool;
    ngx_log_t               *log;
    ngx_log_t               new_log;
```

- ****conf_ctx，以数组的形式存放所有模块的上下文，每个模块对应于数组中的一个元素。
- *pool，指向 Nginx 程序使用的内存池首地址。
- *log 和 new_log，保存 Nginx 日志属性和内容。

```
    ngx_connection_t        **files;
    ngx_connection_t        *free_connections;
    ngx_uint_t              free_connection_n;
    ngx_queue_t             reusable_connections_queue;
```

- **files，最终指向建立的网络连接列表。
- *free_connections 和 free_connection_n，指向空闲的网络链接和存放空闲链接计数。
- reusable_connections_queue，可重用网络连接队列。

```
    ngx_array_t             listening;
    ngx_array_t             paths;
    ngx_list_t              open_files;
    ngx_list_t              shared_memory;
```

- listening，存放监听 socket 的数组。元素的类型为 ngx_listening_t 结构体。
- paths，存放缓存在磁盘上的路径的数组。元素的类型为 ngx_path_t 结构体。
- open_files 和 files_n，存放所有打开的文件描述符的列表，保存打开文件的个数。
- shared_memory，存放 Nginx 程序在运行过程中使用的所有共享内存区域的列表。元素的类型为 ngx_shm_zone_t 结构体。

```
    ngx_uint_t              connection_n;
    ngx_uint_t              files_n;
    ngx_connection_t        *connections;
    ngx_event_t             *read_events;
    ngx_event_t             *write_events;
```

- connection_n，Nginx 服务器当前建立的网络连接计数。
- *connections，存放 Nginx 服务器网络连接的链表。ngx_connection_t 结构体的定义细节参见本章上一节的相关内容。
- *read_events，存放事件驱动模型涉及的读事件。每个网络连接关联一个读事件。ngx_event_t 结构体的定义细节参见 11.6 节的相关内容。
- *write_events，存放事件驱动模型涉及的写事件。每个网络连接关联一个写事件。

```
    ngx_cycle_t             *old_cycle;
    ngx_str_t               conf_file;
    ngx_str_t               conf_param;
    ngx_str_t               conf_prefix;
    ngx_str_t               prefix;
    ngx_str_t               lock_file;
    ngx_str_t               hostname;
};
```

- *old_cycle，缓存过时的全局信息。
- conf_file、conf_param 和 conf_prefix，存放配置文件的内容、参数及前缀等信息。如果配置是默认的，则直接从 Nginx 编译后目录中的 objs/ngx_auto_conf.h 文件加载。
- prefix，存放 Nginx 系统安装的路径。
- lock_file，存放 Nginx 程序的锁文件。
- hostname，存放 Nginx 程序运行所在环境的主机名。

该结构体中存储了 Nginx 服务器在运行过程中所需的全局变量。我们在后面章节学习 Nginx 程序实现源码时，还会对该结构体中的重要成员的使用进行详细说明。

11.9　ngx_conf_s 结构体

该结构体用于 Nginx 在解析配置文件时描述每个指令的属性，也是 Nginx 程序中非常重要的一个数据结构，我们在/nginx/src/core/ngx_conf_file.h 文件中可以找到它的定义：

```
struct ngx_conf_s  {
    char                 *name;
    ngx_array_t          *args;
    ngx_cycle_t          *cycle;
    ngx_pool_t           *pool;
    ngx_pool_t           *temp_pool;
    ngx_conf_file_t      *conf_file;
    ngx_log_t            *log;
    void                 *ctx;
    ngx_uint_t           module_type;
    ngx_uint_t           cmd_type;
    ngx_conf_handler_pt  handler;
    char                 *handler_conf;
};
```

- *name，存放当前解析到的指令。
- *args，存放该指令包含的所有参数。
- *cycle，参见 11.8 节“ngx_cycle_s 结构体”。
- *pool，参见 11.4 节“ngx_pool_s 结构体”。
- *temp_pool，用于解析配置文件的临时内存池，解析完成后释放。其结构体类型的细节参见 11.4 节“ngx_pool_s 结构体”。
- *conf_file，存放 Nginx 配置文件的相关信息。ngx_conf_file_t 结构体的定义我们在该文件中也能找到：

```
typedef struct {
    ngx_file_t          file;                    //文件的属性
    ngx_buf_t           *buffer;                 //文件的内容
    ngx_uint_t          line;                    //文件的行数
```

```
} ngx_conf_file_t;
```

- *log，描述日志文件的相关属性。
- *ctx，描述指令的上下文。
- module_type，支持该指令的模块的类型，core、http、event 和 mail 中的一种。
- cmd_type，指令的类型。
- handler，指令自定义的处理函数。
- *handler_conf，自定义处理函数需要的相关配置。

在该结构体中，有两点需要详细介绍一下。

11.9.1　配置上下文*ctx

我们在第3章中就已经学习过，Nginx 的配置文件是分块配置的，常见的有 http 块、server 块、location 块以及 upsteam 块和 mail 块等。每一个这样的配置块代表一个作用域。高一级配置块的作用域包含了多个低一级配置块的作用域，也就是有作用域嵌套的现象。这样，配置文件中的许多指令都会同时包含在多个作用域内。比如，http 块中的指令都可能同时处于 http 块、server 块和 location 块等三层作用域内。

在 Nginx 程序解析配置文件时，每一条指令都应该记录自己所属的作用域范围，而配置文件上下文 ctx 变量就是用来存放当前指令所属的作用域的。在 Nginx 配置文件的各种配置块中，http 块可以包含子配置块，这在存储结构上比较复杂。我们以 http 指令为例，来说明指令上下文*ctx 的使用。

在/nginx/src/http/ngx_http_conf.h 文件中可以找到 ngx_http_conf_ctx_t 结构体的定义：

```
typedef struct {
    void        ** main_conf;
    void        **srv_conf;
    void        **loc_conf;
} ngx_http_conf_ctx_t;
```

该结构体描述了 http 块的配置上下文。http 块中的 server 块和 location 块中的配置结构会通过指针以数组的形式保存在**main_conf 和**serv_conf 中，http 块本身的配置参数会保存在**main_conf 中，这样 http 块通过该结构体就存储了自己的配置上下文，然后将其以指针的形式保存在 ngx_conf_s 结构体的*ctx 成员中。

与 http 指令类似的还有 mail 指令等。其他不可以包含子配置块的指令在指令上下文的使用上原理一样，但更简单一些。

11.9.2　指令类型 type

这里提到的指令类型和 ngx_command_s 结构体中的 type 含义不同。Nginx 程序中的指令有哪些类型呢？它们以宏的形式定义在不同的源码头文件中。我们在/nginx/src/core/ngx_conf_file.h 文件中可以找到：

```
#define NGX_DIRECT_CONF          0x00010000
#define NGX_MAIN_CONF            0x01000000
#define NGX_ANY_CONF             0x0F000000
```

这些是 core 类型模块支持的指令类型。其中的 NGX_DIRECT_CONF 类指令在 Nginx 程序进入配置解析函数之前已经初始化完成，所以在进入配置解析函数之后可以将它们直接解析并存储到实际的数据结构中，从配置文件的结构上来看，它们一般指的就是那些游离于配置块之外、处于配置文件全局块部分的指令。NGX_MAIN_CONF 类指令包括 event、http、mail、upstream 等可以形成配置块的指令，它们没有自己的初始化函数。Nginx 程序在解析配置文件时如果遇到 NGX_MAIN_CONF 类指令，将转入对下一级指令的解析。

在/nginx/src/event/ngx_conf_file.h 文件中可以找到：

```
#define NGX_EVENT_CONF          0x02000000
```

这是 event 类型模块支持的指令类型。

在/nginx/src/http/ngx_http_conf.h 文件中可以找到：

```
#define NGX_HTTP_MAIN_CONF      0x02000000
#define NGX_HTTP_SRV_CONF       0x04000000
#define NGX_HTTP_LOC_CONF       0x08000000
#define NGX_HTTP_UPS_CONF       0x10000000
#define NGX_HTTP_SIF_CONF       0x20000000
#define NGX_HTTP_LIF_CONF       0x40000000
#define NGX_HTTP_LMT_CONF       0x80000000
```

这些是 http 类型模块支持的指令类型，

在/nginx/src/mail/ngx_mail.h 文件中可以找到：

```
#define NGX_MAIL_MAIN_CONF      0x02000000
#define NGX_MAIL_SRV_CONF       0x04000000
```

这些是 mail 类型模块支持的指令类型。

11.10　ngx_signal_t 结构体

Nginx 服务器的启动、停止和升级都是通过信号控制的。在下一章中，我们会学习到信号控制源码的相关知识，因此在这里需要先介绍一下 Nginx 服务器程序存放信号信息的 ngx_signal_t 结构体。

该结构体的定义我们在/nginx/src/os/unix/ngx_process.c 文件中可以找到：

```
typedef struct {
    int     signo;
    char   *signame;
    char   *name;
    void  (*handler)(int signo);
} ngx_signal_t;
```

- signo，信号的编号。
- *signame，信号的字符串表现形式，如“SIGIO”。
- *name，信号的名称，如“stop”。
- *handler，信号处理函数。

Nginx 服务器程序将所有可能遇到的信号保存在一个结构数组中，该数组我们仍然可以在

/nginx/src/os/unix/ngx_process.c 文件中找到：

```
ngx_signal_t  signals[] = {
    { ngx_signal_value(NGX_RECONFIGURE_SIGNAL),   //RECONFIGURE 信号，重新读取配置信息
      "SIG" ngx_value(NGX_RECONFIGURE_SIGNAL),
      "reload",
      ngx_signal_handler },
    { ngx_signal_value(NGX_REOPEN_SIGNAL),    //RECONFIGURE 信号，重新运行工作进程
      "SIG" ngx_value(NGX_REOPEN_SIGNAL),
      "reopen",
      ngx_signal_handler },
    { ngx_signal_value(NGX_NOACCEPT_SIGNAL),  //NOACCEPT 信号，工作进程不接受事件
      "SIG" ngx_value(NGX_NOACCEPT_SIGNAL),
      "",
      ngx_signal_handler },
    { ngx_signal_value(NGX_TERMINATE_SIGNAL), //TERMINATE 信号，工作进程终止
      "SIG" ngx_value(NGX_TERMINATE_SIGNAL),
      "stop",
      ngx_signal_handler },
    { ngx_signal_value(NGX_SHUTDOWN_SIGNAL),  //SHUTDOWN 信号，结束网络通信，进程退出
      "SIG" ngx_value(NGX_SHUTDOWN_SIGNAL),
      "quit",
      ngx_signal_handler },
    { ngx_signal_value(NGX_CHANGEBIN_SIGNAL), //CHANGEBIN 信号，热升级 Nginx 运行程序
      "SIG" ngx_value(NGX_CHANGEBIN_SIGNAL),
      "",
      ngx_signal_handler },
                                              //以下是常见的系统信号
    { SIGALRM, "SIGALRM", "", ngx_signal_handler },
    { SIGINT, "SIGINT", "", ngx_signal_handler },
    { SIGIO, "SIGIO", "", ngx_signal_handler },
    { SIGCHLD, "SIGCHLD", "", ngx_signal_handler },
    { SIGSYS, "SIGSYS, SIG_IGN", "", SIG_IGN },
    { SIGPIPE, "SIGPIPE, SIG_IGN", "", SIG_IGN },
    { 0, NULL, "", NULL }
};
```

数组中一共包含了 12 个信号，前面 6 个是 Nginx 服务器特别支持的信号。涉及的宏定义有：

```
#define NGX_SHUTDOWN_SIGNAL      QUIT
#define NGX_TERMINATE_SIGNAL     TERM
#define NGX_NOACCEPT_SIGNAL      WINCH
#define NGX_RECONFIGURE_SIGNAL   HUP
#define NGX_REOPEN_SIGNAL        INFO
#define NGX_CHANGEBIN_SIGNAL     XCPU
```

11.11　ngx_process_t 结构体

该结构体是 Nginx 服务器程序用于存放工作进程信息的数据结构，每一个工作进程对应一个这样的结构体，所有的结构体构成一个 ngx_process_t 结构数组，也就是 Nginx 服务器的进程表。

我们在/nginx/src/os/unix/ngx_process.c 文件中可以找到该结构体的定义：

```
typedef struct {
    ngx_pid_t           pid;
    int                 status;
    ngx_socket_t        channel[2];
    ngx_spawn_proc_pt   proc;
    void                *data;
    char                *name;
    unsigned            respawn:1;
    unsigned            just_spawn:1;
    unsigned            detached:1;
    unsigned            exiting:1;
    unsigned            exited:1;
} ngx_process_t;
```

- pid，当前工作进程的 ID 号。
- status，当前进程的退出状态。
- channel[2]，保存由 socketpair 创建的一对 socket 句柄。这对句柄用于进程间交互。
- proc，指向工作进程执行的函数。*data 通常用来指向进程的上下文结构，*name 为新建进程的名称，默认为“new binary process”。
- 后边定义为 unsigned 类型的几个变量标识进程的状态，分别为是否是重新创建的、是否是首次创建的、是否已分离、是否正在退出、是否已经退出。

11.12　本章小结

本章梳理了 Nginx 服务器程序中涉及的重要基础数据结构体。ngx_module_s 结构体涉及模块组织，ngx_command_s 和 ngx_conf_s 结构体涉及指令解析，ngx_pool_s 结构体是内存管理的主要结构，nginx_connection_s 和 ngx_signal_t 结构体分别涉及网络管理和信号管理，它们与进程间通信也有一定的关系，ngx_event_s 结构体涉及事件驱动模型的实现，ngx_process_t 和 ngx_cycle_s 结构体在进程管理方面发挥主要作用，我们将在后面几章介绍各结构体在 Nginx 服务器中的使用。

第 12 章

Nginx 的启动初始化

在上一章中，我们学习了 Nginx 程序涉及的主要数据结构和它们之间的相互依赖关系。这些数据结构在整个 Nginx 源码中占有重要地位，是 Nginx 功能实现的基础。

通过前面相关章节的学习，相信大家对 Nginx 服务器的启动过程有了一定的了解，但期间到底发生了哪些具体行为呢？这是本章学习的重点。本章将从 Nginx 服务器的入口函数 main()开始，对 Nginx 服务器的启动初始化过程进行分解解析。

在本章中，我们学习的主要内容有：

- 源码分析 Nginx 服务器初始化的详细过程
- 源码分析 Nginx 服务器启动的详细过程

12.1 Nginx 启动过程概览

Nginx 服务器程序的 main()函数在/nginx/src/core/nginx.c 文件中可以找到，其包含近 200 行语句，但是从结构上看并不复杂，主要完成了相关变量及模块的初始化、进程启动等工作。

12.1.1 程序初始化

程序初始化过程包括了对 Nginx 程序全局变量、主要数据结构、基础功能模块的初始化工作。按照执行顺序，主要包含以下过程：

- 解析输入参数，通过输入参数确定 Nginx 服务器的具体行为。
- 初始化时间和日志，备份输入参数，并初始化相关的全局变量。一些变量的值依赖于 Nginx 服务器所在操作系统的相关信息，比如内存页面大小、系统支持的最大文件打开数等。

- 保存输入参数。
- 初始化描述网络套接字的相关结构。
- 初始化 ngx_module_t 数组。
- 读取并保存 Nginx 配置参数。
- 初始化 ngx_cycle_s 结构体。
- 保存工作进程 ID 到 PID 文件。

在 Nginx 服务器程序的整个初始化过程中，ngx_cycle_s 结构体是最重要的数据结构，因此对该结构体的初始化是我们学习的重点，我们将在后文中进行详细介绍。

我们通过图 12.1 将 Nginx 服务器程序的初始化过程直观地展示出来。

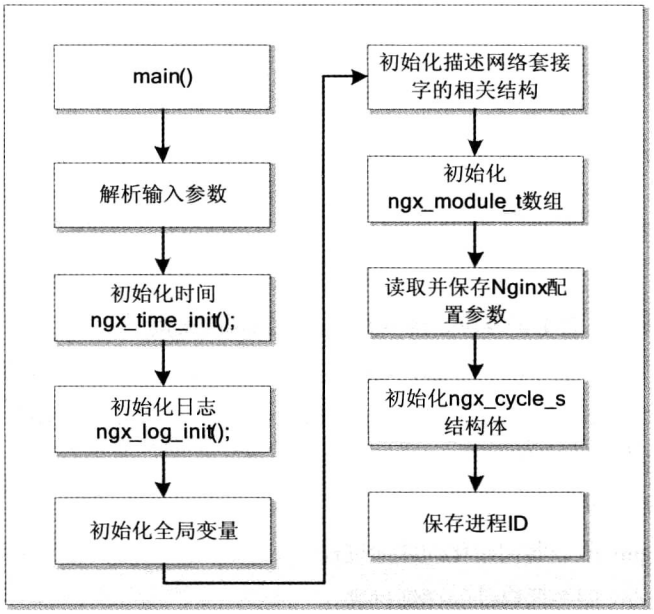

图 12.1　Nginx 初始化过程的主要工作

Nginx 服务器程序完成初始化工作之后，就开始启动进程的工作了。Nginx 服务器程序的进程模型分为 Single 和 Master 两种，其中 Single 模型是以单进程方式进行工作的，一般不会在实际应用中使用；Master 模型是以 Master-Worker 多进程方式进行工作的，它是实际应用环境中使用的主要模式。在程序中，通过全局变量 ngx_process 的值来判断 Nginx 服务器当前的工作模式，进而启动相应模式的进程。我们重点学习 Master 模型下程序的执行。

12.1.2　启动多进程

启动多进程的过程和执行一般的多进程程序是一样的，主要使用 fork()函数产生子进程。主进程就是我们前文提到的 master process（主程序），通过一个 for 循环来接收和处理外部信号，对 Nginx 服务器的启停进行控制；产生出来的子进程就是我们前文提到的 worker process（工作进程），每个工作进程执行一个 for 循环来实现 Nginx 服务器对事件的接收和处理，以提供 Nginx 服务器的各项功能。

图 12.2 展示了 Nginx 服务器程序在 Master 模型下的多进程启动过程。

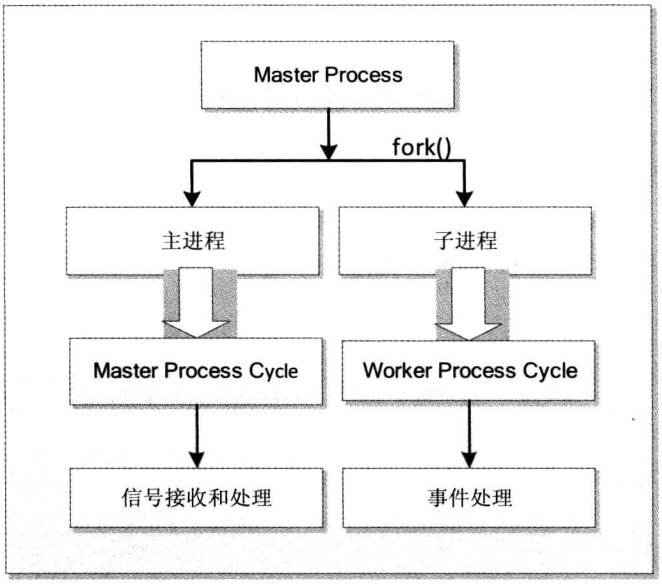

图 12.2　Master 模型下多进程启动过程

在该过程中，两个主要的 for 循环是我们学习的重点。

12.2　Nginx 的初始化

main()函数是整个 Nginx 服务器程序执行的起点，我们在本节中将以 main()源码作为主线，以上一节的内容为纲，深入探索 Nginx 服务器的初始化过程。

注意

笔者在介绍 Nginx 服务器程序源码时将对整块源码进行分段解析，在每一段的开始和结尾使用“// main”表示该段程序的所属函数，使用“->”表示函数的调用关系。对源码的解析不再以注释的形式添加。

```
// main
int ngx_cdecl
main(int argc, char *const *argv)
{
    ngx_int_t        i;
    ngx_log_t       *log;                                    // 保存日志结构
    ngx_cycle_t     *cycle, init_cycle;                      // 初始化时的主体结构体
    ngx_core_conf_t  *ccf;                                   // 保存配置上下文
// main
```

这一段是 main()的开始部分。ngx_cdecl 宏用于显式声明应使用的调用约定，它在跨平台移植时有用。在 Linux 版本的 Nginx 程序中，该宏被定义为空，即：

```
#define  ngx_cdecl
```

有关函数调用约定声明超出了本书的讨论范围，感兴趣的朋友可以参阅维基百科中的相关解释，网址为 http://en.wikipedia.org/wiki/X86_calling_conventions。

ngx_cycle_t 结构体是 Nginx 服务器程序初始化过程中最重要的结构体。我们在后边的源码片段中将会看到该结构体的初始化细节。由于 Nginx 服务器支持平滑升级，因此在整个 main()函数的实现过程中会涉及两个 ngx_cycle_t 结构体，按照程序中的声明，我们分别称它们为 init_cycle 结构和 cycle 结构。在平滑升级服务器的情况中，init_cycle 结构负责保存升级前的信息，cycle 结构负责保存升级后的信息。在其他情况下，init_cycle 结构中将只保存必要的程序信息，最后会完全转移到 cycle 结构中。cycle 结构中会备份旧的 init_cycle 结构的所有信息。

```
// main
    ngx_debug_init();
    if (ngx_strerror_init() != NGX_OK) {            // 初始化 Nginx 自定义的错误输出列表
        return 1;
    }
    if (ngx_get_options(argc, argv) != NGX_OK) {  // 获取运行 Nginx 命令时的参数
        return 1;
    }
// main
```

该段源码中涉及三个主要操作。ngx_debug_init()在 FreeBSD 和 MacOSX 平台下有确切的执行过程，在其他情况下定义为空。ngx_strerror_init()用于初始化 Nginx 服务器自定义的标准错误输出列表。这在初始化 Nginx 日志输出之前是有用的。

12.2.1　读取并处理启动参数

ngx_get_options()函数负责解析 Nginx 程序的输入参数。我们来看一下该函数的实现过程：

```
// main->ngx_get_options
static ngx_int_t
ngx_get_options(int argc, char *const *argv)
{
    u_char    *p;
    ngx_int_t  i;
    for (i = 1; i < argc; i++) {                        //遍历传入的参数
        p = (u_char *) argv[i];
        if (*p++ != '-') {
            ngx_log_stderr(0, "invalid option: \"%s\"", argv[i]);
            return NGX_ERROR;
        }
        while (*p) {
            switch (*p++) {                             // 开始解析命令的参数
            case '?':
            case 'h':
                ngx_show_version = 1;
                ngx_show_help = 1;
                break;
            case 'v':
```

```
            ngx_show_version = 1;
            break;
        case 'V':
            ngx_show_version = 1;
            ngx_show_configure = 1;
            break;
        case 't':
            ngx_test_config = 1;
            break;
        ...                                          // switch 的其他 case 语句，在后面介绍
                                                     // init_cycle 结构初始化时会提到相关参数
        default:
            ngx_log_stderr(0, "invalid option: \"%c\"", *(p - 1));
            return NGX_ERROR;
        }
    }
next:
    continue;
}
return NGX_OK;
}
// main->ngx_get_options
```

Nginx 服务器程序对 Nginx 输入参数的处理与一般 Linux 程序差别不大，其通过循环结构对保存在 argv 变量中的参数进行解析，针对不同的参数对相应的全局变量赋值。在 main()的下文中，程序根据这些全局变量确定 Nginx 服务器要实现的具体功能。

```
// main
    if (ngx_show_version) {
        ngx_write_stderr("nginx version: " NGINX_VER NGX_LINEFEED);
        if (ngx_show_help) {                                  // 输出 nginx 命令的帮助信息
            ngx_write_stderr(
                "Usage: nginx [-?hvVtq] [-s signal] [-c filename] "
                           "[-p prefix] [-g directives]" NGX_LINEFEED
                           NGX_LINEFEED                       // 输出 Nginx 命令的语法结构
                "Options:" NGX_LINEFEED
                "  -?,-h         : this help" NGX_LINEFEED    // 显示该帮助信息
                "  -v            : show version and exit" NGX_LINEFEED // 打印版本号并退出
                "  -V            : show version and configure options then exit"
                                                              // 打印版本号和配置并退出
                                 NGX_LINEFEED
                "  -t            : test configuration and exit" NGX_LINEFEED
                                                              // 测试配置正确性并退出
                "  -q            : suppress non-error messages "   // 测试配置时只显示错误
                                  "during configuration testing" N: send signal to a master
process: " // 向主进程发送信号 GX_LINEFEED
                "  -s signal
                                 "stop, quit, reopen, reload" NGX_LINEFEED
```

```
#ifdef NGX_PREFIX
            "  -p prefix   : set prefix path (default: " // 指定Nginx服务器路径前缀

                             NGX_PREFIX ")" NGX_LINEFEED
#else
            "  -p prefix    : set prefix path (default: NONE)" NGX_LINEFEED

#endif
            "  -c filename  : set configuration file (default: "
                                                    // 指定Nginx配置文件路径
                             NGX_CONF_PATH ")" NGX_LINEFEED
            "  -g directives : set global directives out of configuration "
                                                    // 指定Nginx附加配置文件路径
                             "file" NGX_LINEFEED NGX_LINEFEED
            );
        }
// main
```

该段源码根据 ngx_show_version 和 ngx_show_help 两个全局变量的赋值在标准输出端输出 Nginx 服务器的版本信息和帮助信息。ngx_write_stderr()负责向标准输出端输出文本信息。Nginx 服务器的标准输出端定义为：

```
#define ngx_stderr  STDERR_FILENO
```

即 Linux 平台的标准错误输出端。

```
// main
       if (ngx_show_configure) {
          ngx_write_stderr(
#ifdef NGX_COMPILER
             "built by " NGX_COMPILER NGX_LINEFEED
#endif
#if (NGX_SSL)                                    //支持SSL
#ifdef SSL_CTRL_SET_TLSEXT_HOSTNAME
             "TLS SNI support enabled" NGX_LINEFEED
#else                                            //不支持SSL
             "TLS SNI support disabled" NGX_LINEFEED
#endif
#endif
             "configure arguments:" NGX_CONFIGURE NGX_LINEFEED);
       }
       if (!ngx_test_config) {                   //判断是否对Nginx配置文件进行语法检查
          return 0;
       }
    }
// main
```

该段源码根据全局变量 ngx_show_configure 的赋值在标准输出端输出 Nginx 在配置时的相关配置信息。全局变量 ngx_test_config 的赋值决定是否对 Nginx 配置文件进行语法检查。

```
// main
    // TODO ngx_max_sockets = -1;
    ngx_time_init();                                          //初始化 Nginx 时间
#if (NGX_PCRE)                                                //支持正则表达式
    ngx_regex_init();
#endif
    ngx_pid = ngx_getpid();                                   //获取 PID
// main
```

该段源码包含两个初始化工作。ngx_regex_init()负责完成支持正则表达式的准备工作。另外一个函数 ngx_getpid()负责获取当前 Nginx 进程的进程号。

ngx_time_init()负责初始化自身的时钟管理机制，其实现源码非常简单：

```
// main->ngx_time_init
void
ngx_time_init(void)
{
    ngx_cached_err_log_time.len = sizeof("1970/09/28 12:00:00") - 1;
    ngx_cached_http_time.len = sizeof("Mon, 28 Sep 1970 06:00:00 GMT") - 1;
    ngx_cached_http_log_time.len = sizeof("28/Sep/1970:12:00:00 +0600") - 1;
    ngx_cached_http_log_iso8601.len = sizeof("1970-09-28T12:00:00+06:00") - 1;
    ngx_cached_time = &cached_time[0];
    ngx_time_update();                                        //更新 Nginx 时间
}
// main->ngx_time_init
```

可以看到，主要工作是在另一个函数 ngx_time_update()中实现的。我们将在第 13 章学习 Nginx 的时间管理机制。

```
// main
    log = ngx_log_init(ngx_prefix);                           //初始化日志系统
    if (log == NULL) {
        return 1;
    }
    // STUB
#if (NGX_OPENSSL)                                             //支持 OPENSSL 安全协议
    ngx_ssl_init(log);
#endif
// main
```

在该段源码中，首先由 ngx_log_init()函数负责初始化 Nginx 的日志功能。如果配置使用 Nginx 服务器的 SSL 功能，还会使用 ngx_ssl_init()函数负责完成支持 SSL 功能的准备工作。

我们分析一下 ngx_log_init()函数。该函数实现的功能比较简单，主要的代码片段为：

```
// main->ngx_log_init
    ......
    ngx_log.log_level = NGX_LOG_NOTICE;                       //设置默认日志级别
    ......
    ngx_log_file.fd = ngx_open_file(name, NGX_FILE_APPEND,
                                    NGX_FILE_CREATE_OR_OPEN,
```

```
                                NGX_FILE_DEFAULT_ACCESS);
                                                        //打开（创建）日志文件
    if (ngx_log_file.fd == NGX_INVALID_FILE) {
        ngx_log_stderr(ngx_errno,
                       "[alert] could not open error log file: "
                       ngx_open_file_n " \"%s\" failed", name);
        ngx_log_file.fd = ngx_stderr;
    }
    ......
// main->ngx_log_init
```

函数中调用了 ngx_open_file()函数用于打开 Nginx 服务器的日志文件，并将文件描述符保存起来。日志文件的完整路径存放在变量 name 中。当该日志文件不存在时，首先创建该日志文件。

在 ngx_log_init()函数的源码中，我们还可以找到设置日志输出级别的语句，也就是上面源码中的第一句。我们在第 2 章介绍 error_log 配置参数时提到，Nginx 日志的级别是可选项，由低到高分为 debug、info、notice、warn、error、crit、alert、emerg 等。Nginx 源码将这些日志级别以宏的形式定义在文件/nginx/src/core/ngx_log.h 中：

```
#define NGX_LOG_STDERR            0
#define NGX_LOG_EMERG             1
#define NGX_LOG_ALERT             2
#define NGX_LOG_CRIT              3
#define NGX_LOG_ERR               4
#define NGX_LOG_WARN              5
#define NGX_LOG_NOTICE            6
#define NGX_LOG_INFO              7
#define NGX_LOG_DEBUG             8
```

在文件/nginx/src/core/ngx_log.c 中定义字符串数组 err_levels 与此对应：

```
static ngx_str_t err_levels[] = {                                    //与上面一一
对应
    ngx_null_string,
    ngx_string("emerg"),
    ngx_string("alert"),
    ngx_string("crit"),
    ngx_string("error"),
    ngx_string("warn"),
    ngx_string("notice"),
    ngx_string("info"),
    ngx_string("debug")
};
```

我们再回到 Nginx 服务器程序的 main()函数：

```
// main
    ngx_memzero(&init_cycle, sizeof(ngx_cycle_t));                   //内存管理初始化
    init_cycle.log = log;
    ngx_cycle = &init_cycle;
    init_cycle.pool = ngx_create_pool(1024, log);                    //创建内存池
```

```
    if (init_cycle.pool == NULL) {
        return 1;
    }
// main
```

在该段代码中，程序初始化了 init_cycle 结构中的两个成员 log 和 pool，其中 log 中保存的就是上面初始化日志的结果。ngx_create_pool()函数用于创建内存池，我们将在第 14 章学习 Nginx 内存管理的实现细节。这里，该函数创建了大小为 1024 字节的内存池供 init_cycle 结构使用。

```
// main
    if (ngx_save_argv(&init_cycle, argc, argv) != NGX_OK) { //保存传入的参数到全局变量
        return 1;
    }
    if (ngx_process_options(&init_cycle) != NGX_OK) {        //保存传入的参数到
init_cycle 结构
        return 1;
    }
// main
```

在该段代码中，ngx_save_argv()函数用于将此次启动 Nginx 程序指定的参数备份到全局变量中。ngx_process_options()函数用于将 Nginx 服务器在启动时指定的参数保存到 init_cycle 结构的相应成员中。

```
// main->ngx_process_options
static ngx_int_t
ngx_process_options(ngx_cycle_t *cycle)
{
    u_char  *p;
    size_t   len;
    if (ngx_prefix) {                                    // Nginx 服务器程序的安装路径
        len = ngx_strlen(ngx_prefix);
        p = ngx_prefix;
// main->ngx_process_options
```

其中，ngx_prefix 全局变量中保存了 Nginx 服务器程序的安装路径。该全局变量在解析“-p”参数时被赋值：

```
// mian->ngx_get_options
case 'p':
    if (*p) {
        ngx_prefix = p;
        goto next;
    }
    if (argv[++i]) {                                     // 解析指定的路径字符串
        ngx_prefix = (u_char *) argv[i];
        goto next;
    }
    ngx_log_stderr(0, "option \"-p\" requires directory name");
    return NGX_ERROR;
// mian->ngx_get_options
```

举例来说，我们平滑升级 Nginx 服务器时，使用以下命令将 Nginx 服务器的安装路径改变为新版本的路径：

```
#nginx  -p  /new_nginx/
```

ngx_prefix 变量中保存的值为/new_nginx/。

```
// main->ngx_process_options
        ......
        cycle->conf_prefix.len = len;
        cycle->conf_prefix.data = p;
        cycle->prefix.len = len;
        cycle->prefix.data = p;
......
#ifndef NGX_PREFIX
        p = ngx_pnalloc(cycle->pool, NGX_MAX_PATH);
        ......
        if (ngx_getcwd(p, NGX_MAX_PATH) == 0) {              //获取 Nginx 当前的工作路径
            ngx_log_stderr(ngx_errno, "[emerg]: " ngx_getcwd_n " failed");
            return NGX_ERROR;
        }
......
        cycle->conf_prefix.len = len;
        cycle->conf_prefix.data = p;
        cycle->prefix.len = len;
        cycle->prefix.data = p;
#else
#ifdef NGX_CONF_PREFIX                          //如果定义了 CONF_PREFIX，则保存在
&cycle->conf_prefix,
        ngx_str_set(&cycle->conf_prefix, NGX_CONF_PREFIX);
#else                                           //否则使用 PREFIX 代替 CONF_PREFIX
        ngx_str_set(&cycle->conf_prefix, NGX_PREFIX);
#endif
        ngx_str_set(&cycle->prefix, NGX_PREFIX);
#endif
    }
// main->ngx_process_options
```

该段代码完成在不同情况下对 init_cycle 结构中的 config_prefix 成员和 prefix 成员的初始化工作。这两个成员的不同之处我们通过一个例子来说明。

在编译 Nginx 源码前，假设我们使用了如下的配置：

```
#./configure --prefix=/usr/local/nginx
```

那么，init_cycle 结构的 config_prefix 成员中应该存放的是“/usr/local/nginx/conf”，也就是 Nginx 服务器配置文件存放的路径。prefix 成员中应该存放的是“/usr/local/nginx”，也就是 Nginx 服务器的安装目录。

```
// main->ngx_process_options
    if (ngx_conf_file) {
```

```
        cycle->conf_file.len = ngx_strlen(ngx_conf_file);
        cycle->conf_file.data = ngx_conf_file;
    } else {
        ngx_str_set(&cycle->conf_file, NGX_CONF_PATH);
    }
    if (ngx_conf_full_name(cycle, &cycle->conf_file, 0) != NGX_OK) { //判断是否是完整路径
        return NGX_ERROR;
    }
    for (p = cycle->conf_file.data + cycle->conf_file.len - 1;
         p > cycle->conf_file.data;
         p--)
    {
        if (ngx_path_separator(*p)) {
            cycle->conf_prefix.len = p - ngx_cycle->conf_file.data + 1;
            cycle->conf_prefix.data = ngx_cycle->conf_file.data;
            break;
        }
    }
// main->ngx_process_options
```

该段代码用于初始化 init_cycle 结构中的 conf_file 成员。该成员的值来源于全局变量 ngx_conf_file，而该全局变量在解析“-c”参数时被赋值：

```
// main->ngx_get_options
case 'c':
    if (*p) {
        ngx_conf_file = p;
        goto next;
    }
    if (argv[++i]) {                                          // 解析指定的路径字符串
        ngx_conf_file = (u_char *) argv[i];
        goto next;
    }
    ngx_log_stderr(0, "option \"-c\" requires file name");
    return NGX_ERROR;
// main->ngx_get_options
```

比如，使用下面的命令设置 Nginx 服务器的配置文件：

```
#nginx  -c  conf/new_nginx.conf
```

conf_file 成员首先被赋值为“conf/new_nginx.conf”，然后通过 ngx_conf_full_name()函数的处理，将该值补全为 Nginx 服务器配置文件的绝对路径。最后，需要根据更新后的配置文件路径更新 init_cycle 结构中的 conf_prefix 成员的值。

```
// main->ngx_process_options
    if (ngx_conf_params) {
        cycle->conf_param.len = ngx_strlen(ngx_conf_params);
        cycle->conf_param.data = ngx_conf_params;
    }
```

```
// main->ngx_process_options
```

该段代码初始化了 init_cycle 结构中的 conf_param 成员。该成员的值来源于全局变量 ngx_conf_params，该变量在解析“-g”参数时被赋值：

```
// main->ngx_get_options
case 'g':
    if (*p) {
        ngx_conf_params = p;
        goto next;
    }
    if (argv[++i]) {
        ngx_conf_params = (u_char *) argv[i];
        goto next;
    }
    ngx_log_stderr(0, "option \"-g\" requires parameter");
    return NGX_ERROR;
// main->ngx_get_options
```

比如，使用下面的命令设置 Nginx 服务器的补充配置文件：

```
#nginx  -g  /nginx/conf/param.conf
```

成员 conf_param 存放的就是“/nginx/conf/param.conf”，该文件中存放了 Nginx 服务启动时应用于全局的额外配置。

```
// main->ngx_process_options
    if (ngx_test_config) {
        cycle->log->log_level = NGX_LOG_INFO;
    }
    return NGX_OK;
}
// main->ngx_process_options
```

当 Nginx 服务器在启动时指定了“-t”参数，那么要对 Nginx 配置文件进行语法检查。执行语法检查是由另外的功能模块实现的，该程序设置了这种情况下日志的输出级别。

到这里，处理和保存启动参数的工作已经做完了。我们回到 main()函数，继续分析。

```
// main
    if (ngx_os_init(log) != NGX_OK) {                    //获取运行环境中的一些相关参数
        return 1;
    }
// main
```

Nginx 服务器的高效运行与运行环境具有密切的联系。我们在前面相关章节学习 Nginx 配置的过程中，经常会遇到根据系统环境变量以及内核配置参数来优化配置的情况。ngx_os_init()函数用来获取运行环境中的一些相关参数，并将其保存到全局变量中，涉及的有存放系统内存页大小的 ngx_pagesize 变量，存放 CPU 个数的 ngx_ncpu 变量，存放支持打开的最大网络套接字数量的 ngx_max_sockets 变量等信息。

```
// main
    if (ngx_crc32_table_init() != NGX_OK) {          //建立循环冗余校验表
```

```
        return 1;
    }
// main
```

在该段代码中，初始化用于循环冗余校验的表，方便以后在用到循环冗余校验时可以采用高效的查表法。相关的算法超出了本书的讨论范围，此处就不做具体分析了。

```
// main
    if (ngx_add_inherited_sockets(&init_cycle) != NGX_OK) {
        return 1;
    }
// main
```

该段代码完成对已有 socket 的继承工作，我们通过一节的内容来分析一下该函数的源码和实现机制。

12.2.2　继承 socket

在 Nginx 服务器升级等情况中，为了保证 Web 服务的平滑过渡，新的 Nginx 进程需要能够继承旧的 Nginx 进程打开的 socket 描述符，继续保证与客户端的网络连接。这些已有的文件描述符存放在环境变量 NGINX 中，用冒号或者分号隔开，每个文件描述符是一个整数值。类似的结构如：

```
10000:10002:10003;10004:
```

我们来分析一下相关的 ngx_add_inherited_sockets()函数源码。

```
// main->ngx_add_inherited_sockets
static ngx_int_t
ngx_add_inherited_sockets(ngx_cycle_t *cycle)
{
    u_char         *p, *v, *inherited;
    ngx_int_t       s;
    ngx_listening_t   *ls;
    inherited = (u_char *) getenv(NGINX_VAR);              //设置 Nginx 的环境变量
    if (inherited == NULL) {
        return NGX_OK;
    }
// main->ngx_add_inherited_sockets
```

这里 NGINX_VAR 宏的定义为：

```
#define NGINX_VAR   "NGINX"
```

使用 getenv()函数获取环境变量 NGINX 的值。当然，在第一次启动 Nginx 服务器时，环境变量 NGINX 的值为空，程序运行到此处就退出该函数，并返回 NGX_OK。

```
// main->ngx_add_inherited_sockets
    ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0,
                  "using inherited sockets from \"%s\"", inherited);
    if (ngx_array_init(&cycle->listening, cycle->pool, 10, sizeof(ngx_listening_t))
        != NGX_OK)
    {
        return NGX_ERROR;
    }
```

```
    for (p = inherited, v = p; *p; p++) {
        if (*p == ':' || *p == ';') {                    //通过冒号或者分号取出列表中的 socket
            s = ngx_atoi(v, p - v);
            if (s == NGX_ERROR) {
                ngx_log_error(NGX_LOG_EMERG, cycle->log, 0,
                              "invalid socket number \"%s\" in " NGINX_VAR
                              " environment variable, ignoring the rest"
                              " of the variable", v);
                break;
            }
            v = p + 1;
            ls = ngx_array_push(&cycle->listening);          //保存解析出来的 socket
            if (ls == NULL) {
                return NGX_ERROR;
            }
            ngx_memzero(ls, sizeof(ngx_listening_t));
            ls->fd = (ngx_socket_t) s;
        }
    }
    ngx_inherited = 1;                                       //标记继承工作的结束
// main->ngx_add_inherited_sockets
```

该段代码将环境变量 NGINX 中的 socket 描述符依次解析出来并保存在 init_cycle 结构的成员 listening 数组中，数组中的每个元素是一个 ngx_listening_t 结构，该结构中的 fd 成员中存放了解析出来的 socket 文件描述符号。完成对 socket 描述符的继承后，设置全部变量 ngx_inherited 为 1。

```
// main->ngx_add_inherited_sockets
    return ngx_set_inherited_sockets(cycle);                 //完成继承 socket
}
// main->ngx_add_inherited_sockets
```

最后使用 ngx_set_inherited_sockets()函数，从 init_cycle 结构的成员 listening 数组中逐一取出每个元素（ngx_listening_t 结构），初始化每个元素除 fd 成员外的其他成员。

到这里，我们分析完了 Nginx 服务器继承 socket 的机制。我们回到 main()函数继续分析。

```
// main
    ngx_max_module = 0;
    for (i = 0; ngx_modules[i]; i++) {                       //遍历所有的模块，建立模块索引
        ngx_modules[i]->index = ngx_max_module++;
    }
// main
```

我们在前面章节中已经知道，Nginx 服务器的每个功能模块使用 ngx_module_t 结构描述，编译好的 Nginx 服务器中包含的所有模块定义在程序安装目录下 obj/ngx_modules.c 文件中。该代码中的 for 循环描述了每个 ngx_module_t 结构中的 index 成员。

```
// main
    cycle = ngx_init_cycle(&init_cycle);                     //建立新的 cycle 结构
    if (cycle == NULL) {
        if (ngx_test_config) {
```

```
        ngx_log_stderr(0, "configuration file %s test failed",
init_cycle.conf_file.data);
      }
      return 1;
   }
// main
```

该代码中的 ngx_init_cycle()函数是 Nginx 服务器初始化过程中的核心部分，我们详细分析一下。

12.2.3　初始化时间及建立新的 cycle 结构

ngx_init_cycle()函数包含 800 多行源码，完成了对 init_cycle 结构中大多数重要成员的一系列初始化工作。传入该函数的 init_cycle 结构是前面刚刚初始化过部分成员的 ngx_cycle_t 结构，在本函数中我们将在新建立的内存池上重新给 ngx_cycle_t 结构分配内存，用 cycle 变量指向该内存区域，并把刚才已经初始化了的成员值转移到新的结构中。我们针对重要的源码片段进行分析和学习。

```
// main->ngx_init_cycle
   ngx_timezone_update();                                    //初始化时区
   // force localtime update with a new timezone
   tp = ngx_timeofday();                                     //获取时间缓存中的时间
   tp->sec = 0;
   ngx_time_update();                                        //更新缓存时间
// main->ngx_init_cycle
```

在该段代码中，ngx_timezone_update()函数用于根据 Nginx 服务器运行系统平台的不同初始化时区。ngx_timeofday()函数实际上是宏定义，用于获取时间缓存中的时间，定义如下：

```
extern volatile ngx_time_t  *ngx_cached_time;
#define  ngx_timeofday()  (ngx_time_t *) ngx_cached_time
```

变量 ngx_cached_time 指向时间缓存中最新缓存的时间，实际上就是在前面介绍的 ngx_time_update()函数中进行初始化的时间：

```
// main->ngx_time_init ->ngx_time_update
   tp = &cached_time[slot];
   tp->sec = sec;
   tp->msec = msec;
   ......
   ngx_cached_time = tp;
// main->ngx_time_init ->ngx_time_update
```

接着来看 init_cycle 结构的初始化工作：

```
// main->ngx_init_cycle
   pool = ngx_create_pool(NGX_CYCLE_POOL_SIZE, log);      //建立新的内存池
   if (pool == NULL) {
      return NULL;
   }
   pool->log = log;
   cycle = ngx_pcalloc(pool, sizeof(ngx_cycle_t));        //为内存池分配空间
   if (cycle == NULL) {
      ngx_destroy_pool(pool);
```

```
        return NULL;
    }
// main->ngx_init_cycle
```

该段代码使用 ngx_create_pool()函数建立了新的内存池，内存池的大小可以通过配置文件指定，如果未指定则使用程序默认的 16384 字节：

```
#ifndef NGX_CYCLE_POOL_SIZE
#define NGX_CYCLE_POOL_SIZE     16384
#endif
```

该内存池将为Nginx服务器程序运行的整个生命周期提供内存分配和管理。紧接着的ngx_pcalloc()函数从内存池中申请了 ngx_cycle_t 结构大小的内存块供新的 cycle 结构使用。

接下来的代码主要是从传入的 init_cycle 结构中将已经初始化了的成员值重新赋值给 cycle 结构中的相应成员，比如 cycle->log、cycle->conf_prefix、cycle->prefix、cycle->conf_file、cycle->conf_param 等，过程比较简单，各个成员的含义在第 11 章中相关的数据结构中学习过，我们在此不做更多详细介绍了。

接着程序给 cycle->pathes 数组分配空间，该数组用于管理 Nginx 服务器程序运行过程中涉及的所有路径字符串；给 cycle->open_files 链表分配空间，其用于管理程序运行过程中打开的文件；给 shared_memory 链表分配空间，其用于管理程序运行中各进程间使用的共享内存；给 cycle->listening 数组和 cycle->resuable_connections_queue 队列分配空间，cycle->resuable_connections_queue 队列中将会存放可重用网络连接，供 Nginx 服务器回收使用已经打开却长时间未被使用的网络连接；获取主机名称和初始化 cycle->hostname 成员等。需要注意的是，上面提到的这些数据结构只对其进行空间分配，并不从 init_cycle 结构中复制具体的内容。

12.2.4　建立 core 模块上下文结构

core 模块是 Nginx 服务器运行的核心。接下来，程序的主要工作就是建立 core 模块上下文结构，结构类型为 ngx_core_conf_t 结构体。程序首先获取 core 模块指向模块上下文结构的指针 module，然后调用 module->create_conf(cycle)完成对 core 模块上下文结构的建立工作。我们看一下相关的代码：

```
// main->ngx_init_cycle
    for (i = 0; ngx_modules[i]; i++) {
        if (ngx_modules[i]->type != NGX_CORE_MODULE) {      //筛选出 core 模块
            continue;
        }
        module = ngx_modules[i]->ctx;
        if (module->create_conf) {
            rv = module->create_conf(cycle);                //建立 core 模块上下文结构
            if (rv == NULL) {
                ngx_destroy_pool(pool);
                return NULL;
            }
            cycle->conf_ctx[ngx_modules[i]->index] = rv;
        }
    }
```