

# 代码之美

精选版



Andy Oram, Grey Wilson 著

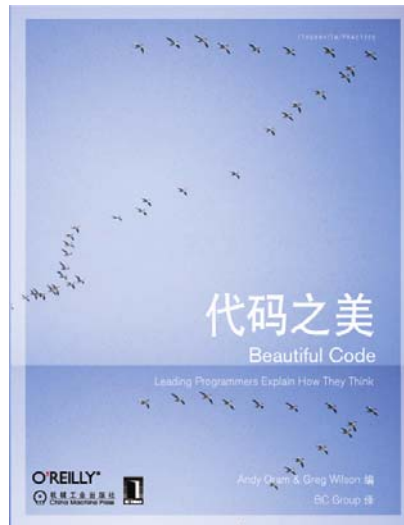
BC Group 译

**InfoQ**企业软件开发丛书

# 免费在线版本

(非印刷免费在线版)

[登录China-Pub网站购买此书完整版](#)



了解本书更多信息请登录[本书的官方网站](#)

InfoQ 中文站出品

**InfoQ中文站**  
[www.infoq.com/cn](http://www.infoq.com/cn)

本书由 InfoQ 中文站免费发放，如果您从其他渠道获取本书，请注册 InfoQ 中文站以支持作者和出版商，并免费下载更多 InfoQ 企业软件开发系列图书。

本书主页为

<http://infoq.com/cn/minibooks/beautiful-code>

© 2008 C4Media Inc.

版权所有

C4Media 是 InfoQ.com 这一企业软件开发社区的出版商

本书属于 InfoQ 企业软件开发丛书

如果您打算订购 InfoQ 的图书，请联系 [books@c4media.com](mailto:books@c4media.com)

未经出版者预先的书面许可，不得以任何方式复制或者抄袭本书的任何部分，本书任何部分不得用于再印刷，存储于可重复使用的系统，或者以任何方式进行电子、机械、复印和录制等形式传播。

本书提到的公司产品或者使用到的商标为产品公司所有。

如果读者要了解具体的商标和注册信息，应该联系相应的公司。

本书在征得华章出版公司许可下制作，以电子文档形式发布。

欢迎共同参与 InfoQ 中文站的内容建设工作，包括原创投稿和翻译等，请联系 [editors@cn.infoq.com](mailto:editors@cn.infoq.com)。

# 序

Greg Wilson

我在1982年夏天获得了第一份程序员工作。在我工作了两个星期后，一位系统管理员借给了我两本书：Kernighan和Plauger编写的《The Elements of Programming Style》(McGraw-Hill出版社)和Wirth编写的《Algorithms + Data Structures = Programs》(Prentice Hall出版社)。这两本书让我大开眼界——我第一次发现程序并不仅仅只是一组计算机执行的指令。它们可以像做工优良的橱柜一样精致，像悬索吊桥一样漂亮，或者像George Orwell的散文一样优美。

自从那个夏天以来，我经常听到人们感叹我们的教育并没有教会学生看到这一点。建筑师们需要观摩建筑物，作曲家们需要研习他人的作品，而程序员——他们只有在需要修改bug时才会去阅读其他人的代码；即使在这个时候，他们也会尽可能减少阅读量。我们曾告诉学生使用有意义的变量名，曾向他们介绍过一些基本的设计模式，但很奇怪，为什么他们编写的大多数代码都是很难看的呢！

本书将试图改变这种状况。2006年5月，我邀请了一些著名的（以及不太著名的）软件设计师来分析和讨论他们所知道的漂亮代码。正如在本书中将要介绍的，他们在许多不同的地方发现了代码的漂亮性。有些漂亮性存在于手工精心打造软件的细微之处，而有些漂亮性是蕴涵在大局之中——那些使程序能够持续发展的架构，或者用来构造程序的技术。

无论他们是在什么地方发现的这些漂亮性，我都非常感谢我们的投稿人抽出时间为我们奉献了这样的一次学习旅程。我希望你能够享受阅读此书乐趣，就像Andy和我非常享受编辑这本书的过程，此外，我还希望这本书能激发你创建出一些漂亮的作品。

# 前言

《Beautiful Code》是由Greg Wilson在 2006 年构思的，本书的初衷是希望从优秀的软件开发人员和计算机科学家中提炼出一些有价值的思想。他与助理编辑Andy Oram一起走访了世界各地不同技术背景的专家。本《代码之美》精选版是从原书中精选出其中的 6 章。

## 本书章节内容的组织

第 1 章，正则表达式匹配器，作者 Brian Kernighan，介绍了对一种语言和一个问题的深入分析以及由此产生的简洁而优雅的解决方案。

第 2 章，我编写过的最漂亮代码，作者 Jon Bentley，介绍了如何在无需执行函数的情况下测试函数的性能。

第 3 章，美丽的测试，作者 Alberto Savoia，介绍了一种全新的测试方法，不仅能够消除 bug，还可以使你成为一个更优秀的程序员。

第 4 章，NASA 火星漫步者任务中的高可靠企业系统，作者 Ronald Mak，介绍了如何使用工业标准，最佳实践和 Java 技术来满足 NASA 探险任务的高可靠性需求。

第 5 章，美丽的并发，作者 Simon Peyton Jones，通过软件事务内存（Software Transactional Memory）来消除大多数并发程序中的困难，在本章中使用 Haskell 语言来说明。

第 6 章，以 REST 方式集成业务伙伴，作者 Andrew Patzer，通过根据需求来设计一个 B2B Web Service 从而表现出设计者对程序开发人员的尊重。

# 目录

序 .....	i
前言 .....	ii
第 1 章 正则表达式匹配器 .....	1
1.1 编程实践 .....	2
1.2 实现 .....	3
1.3 讨论 .....	4
1.4 其他的方法 .....	5
1.5 构建 .....	6
1.6 小结 .....	8
第 2 章 我编写过的最漂亮代码 .....	10
2.1 我编写过的最漂亮代码 .....	10
2.2 事倍功半 .....	11
2.3 观点 .....	16
2.4 本章的中心思想是什么? .....	18
2.5 结论 .....	18
2.6 致谢 .....	20
第 3 章 美丽测试 .....	21
3.1 讨厌的二分查找 .....	22
3.2 JUnit 简介 .....	27
3.3 将二分查找进行到底 .....	29
3.4 结论 .....	47
第 4 章 NASA 火星漫步者任务中的高可靠企业系统 .....	49
4.1 任务与 CIP .....	49
4.2 任务需求 .....	50
4.3 系统架构 .....	51
4.4 案例分析：流服务 .....	54
4.5 可靠性 .....	57

---

4.6 稳定性.....	66
4.7 结束语.....	67
第 5 章 美丽的并发.....	68
5.1 一个简单的例子.....	68
5.2 软件事务内存.....	71
5.3 圣诞老人问题.....	80
5.4 对 Haskell 的一些思考 .....	90
5.5 结论.....	91
5.6 致谢.....	92
第 6 章 以 REST 方式集成业务伙伴 .....	93
6.1 项目背景.....	93
6.2 把服务开放给外部客户 .....	93
6.3 使用工厂模式转发服务.....	97
6.4 用电子商务协议来交换数据.....	98
6.5 结束语.....	104
后记 .....	106

# 第 1 章 正则表达式匹配器

Brian Kernighan

正则表达式是描述文本模式的表示法，它可以有效地构造一种用于模式匹配的专用语言。虽然正则表达式可以有多种不同的形式，但它们都有着共同的特点：模式中的大多数字符都是匹配字符串中的字符本身，但有些元字符（metacharacter）却有着特定的含义，例如\*表示某种重复，而[...]表示方括号中字符集合的任何一个字符。

实际上，在文本编辑器之类的程序中，所执行的查找操作都是查找文字，因此正则表达式通常是像“print”之类的字符串，而这类字符串将与文档中所有的“printf”或者“sprintf”或者“printer paper”相匹配。在Unix和Windows中可以使用所谓的通配符来指定文件名，其中字符\*可以用来匹配任意数量的字符，因此匹配模式\*.c就将匹配所有以.c结尾的文件。此外，还有许许多多不同形式的正则表达式，甚至在有些情况下，这些正则表达式会被认为都是相同的。Jeffrey Friedl编著的《Mastering Regular Expressions》一书对这一方面问题进行了广泛的研究。

Stephen Kleene在20世纪50年代的中期发明了正则表达式，用来作为有限自动机的表示法，事实上，正则表达式与它所表示的有限自动机是等价的。20世纪60年代中期，正则表达式最初出现在Ken Thompson版本的QED文本编辑器的程序设置中。1967年Thompson申请了一项基于正则表达式的快速文本匹配机制的专利。这项专利在1971年获得了批准，它是最早的软件专利之一[U.S. Patent 3,568,156, Text Matching Algorithm, March 2, 1971]。

后来，正则表达式技术从QED移植到了Unix的编辑器ed中，然后又被移植到经典的Unix工具grep中，而grep正是由于Thompson对ed进行了彻底地修改而形成的。这些广为应用的程序使得正则表达式为早期的Unix社群所熟知。

Thompson最初编写的匹配器是非常快的，因为它结合了两种独立的思想。一种思想是在匹配过程中动态地生成机器指令，这样就可以以机器指令执行的速度而不是解释执行的速度来运行。另一种思想是在每个阶段中都尽可能地执行匹配操作，这样无需回溯（backtrack）就可以查找可能的匹配。在Thompson后来编写的文本编辑器程序中，例如ed，匹配代码使用了一种更为简单的算法，这种算法将会在必要的时候进行回溯。从理论上来看，这种方法的运行速度要更慢，但在实际情况中，这种模式很少需要进行回溯，因此，ed和grep中的算法和代码足以应付大多数的情况。

在后来的正则表达式匹配器中，例如egrep和fgrep等，都增加了更为丰富的正则表达式类



型，并且重点是要使得匹配器无论在什么模式下都能够快速执行。功能更为强大的正则表达式正在被越来越多地使用，它们不仅被包含在用C语言开发的库中，而且还被作为脚本语言如Awk和Perl的语法的一部分。

## 1.1 编程实践

在1998年，Rob Pike和我还在编写《The Practice of Programming》（Addison-Wesley）一书。书中的最后一章是“记法”，在这一章中收录了许多示例代码，这些示例都很好地说明了良好的记法将会带来更好的程序以及更好的设计。其中包括使用简单的数据规范（例如printf）以及从表中生成代码。

由于我们有着深厚的Unix技术背景以及在使用基于正则表达式记法的工具上有着近30年的经验，我们很自然地希望在本书中包含一个对正则表达式的讨论，当然包含一个实现也是必须的。由于我们强调了工具软件的使用，因此似乎最好应该把重点放在grep中的正则表达式类型上——而不是，比方说，在shell的通配符正则表达式上——这样我们还可以在随后再讨论grep本身的设计。

然而，问题是现有的正则表达式软件包都太庞大了。grep中的代码长度超过500行（大约10页书的长度），并且在代码的周围还有复杂的上下文环境。开源的正则表达式软件包则更为庞大——代码的长度几乎布满整本书——因为这些代码需要考虑通用性，灵活性以及运行速度；因此，所有这些正则表达式都不适合用来教学。

我向Rob建议我们需要一个最小的正则表达式软件包，它可以很好地诠释正则表达式的基本思想，并且能够识别出一组有用的并且重要类的模式。理想的情况是，所需代码长度只有一页就够了。

Rob听了我的提议后就离开了他的办公室。我现在还记得，一，两个小时后他回来了，并且给了我一段大约30行的C代码，在《The Practice of Programming》一书的第9章中包含了这段代码。在这段代码实现了一个正则表达式匹配器，用来处理以下的模型。

字符	含义
c	匹配任意的字母c
.	（句点） 匹配任意的单个字符
^	匹配输入字符串的开头
\$	匹配输入字符串的结尾
*	匹配前一个字符的零个或者多个出现

这是一个非常有用的匹配器，根据我在日常工作中使用正则表达式的经验，它可以轻松解决95%的问题。在许多情况下，解决正确的问题就等于朝着创建漂亮的程序迈进了一大步。Rob

值得好好地表扬，因为他从大量可选功能集中选出了一组非常小但却重要的，并且是明确的以及可扩展的功能。

Rob的实现本身就是漂亮代码的一个极佳示例：紧凑，优雅，高效并且实用。这是我所见过的最好的递归示例之一，在这段代码中还展示了C指针的强大功能。虽然当时我们最关心的是通过使程序更易于使用（同时也更易于编写）来体现良好记法的重要性，但正则表达式代码同样也是阐述算法，数据结构，测试，性能增强以及其他重要主题的最好方式。

## 1.2 实现

在《The Practice of Programming》一书中，正则表达式匹配器是一个模拟grep程序中的一部分，但正则表达式的代码完全可以从编写环境中独立出来。这里我们并不关心主程序；像许多Unix工具一样，这个程序将读取其标准输入或者一组文件，然后输出包含与正则表达式匹配的文本行。

以下是匹配算法的代码：

```
/* match: search for regexp anywhere in text */
int match(char *regexp, char *text)
{
    if (regexp[0] == '^')
        return matchhere(regexp+1, text);
    do { /* must look even if string is empty */
        if (matchhere(regexp, text))
            return 1;
    } while (*text++ != '\0');
    return 0;
}

/* matchhere: search for regexp at beginning of text */
int matchhere(char *regexp, char *text)
{
    if (regexp[0] == '\0')
        return 1;
    if (regexp[1] == '*')
        return matchstar(regexp[0], regexp+2, text);

    if (regexp[0] == '$' && regexp[1] == '\0')
        return *text == '\0';
    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
        return matchhere(regexp+1, text+1);
    return 0;
}
```

```
    }

    /* matchstar: search for c*regexp at beginning of text */
    int matchstar(int c, char *regexp, char *text)
    {
        do { /* a * matches zero or more instances */
            if (matchhere(regexp, text))
                return 1;
        } while (*text != '\0' && (*text++ == c || c == '.'));
        return 0;
    }
```

### 1.3 讨论

函数`match(regexp, text)`用来判断文本中是否出现正则表达式；如果找到了一个匹配的正则表达式则返回1，否则返回0。如果有多个匹配的正则表达式，那么函数将找到文本中最左边的并且最短的那个。

`match`函数中的基本操作简单明了。如果正则表达式中的第一个字符是`^`（固定位置的匹配），那么匹配就一定要出现在字符串的开头。也就是说，如果正则表达式是`^xyz`，那么仅当`xyz`出现在文本的开头而不是中间的某个位置时才会匹配成功。在代码中通过把正则表达式的剩余部分与文本的起始位置而不是其他地方进行匹配来判断。如果第一个字符不是`^`，那么正则表达式就可以在字符串中的任意位置上进行匹配。在代码中通过把模式依次与文本中的每个字符位置进行匹配来判断。如果存在多个匹配，那么代码只会识别第一个（最左边的）匹配。也就是说，如果则在表达式是`xyz`，那么将会匹配第一次出现的`xyz`，而且不考虑这个匹配出现在什么位置上。

注意，对输入字符串的推进操作是在一个`do-while`循环中进行的，这种结构在C程序中使用相对较少。在代码中使用`do-while`而不是`while`通常会带来疑问：为什么不在循环的起始处判断循环条件，而是在循环末尾当执行完了某个操作之后才进行判断呢？不过，这里的判断是正确的：由于`*`运算符允许零长度的匹配，因此我们首先需要判断是否存在一个空的匹配。

大部分的匹配工作是在`matchhere(regexp, text)`函数中完成的，这个函数将判断正则表达式与文本的开头部分是否匹配。函数`matchhere`把正则表达式的第一个字符与文本的第一个字符进行匹配。如果匹配失败，那么在这个文本位置上就不存在匹配，因此`matchhere`将返回0。然而，如果匹配成功了，函数将推进到正则表达式的下一个字符和文本的下一个字符继续进行匹配。这是通过递归地调用`matchhere`函数来实现的。

由于存在着一些特殊的情况，以及需要设置终止递归的条件。因此实际的处理过程要更为

复杂些最简单的情况就是，当正则表达式推进到末尾时(`regex[0] == '\0'`)，所有前面的判断都成功了，那么这个正则表达式就与文本匹配。

如果正则表达式是一个字符后面跟着一个`*`，那么将会调用`matchstar`来判断闭包(`closure`)是否匹配。函数`matchstar(c, regex, text)`将尝试匹配重复的文本字符`c`，从零重复开始并且不断累加，直到匹配`text`的剩余字符，如果匹配失败，那么函数就认为不存在匹配。这个算法将识别出一个“最短的匹配”，这对简单的模式匹配来说是很好的，例如`grep`，这种情况下的主要问题是尽可能快地找到一个匹配。而对于文本编辑器来说，“最长的匹配”则是更为直观，且肯定是更好的，因为通常需要对匹配的文本进行替换。在目前许多的正则表达式库中同时提供了这两种方法，在《The Practice of Programming》一书中给出了基于本例中`matchstar`函数的一种简单变形，我们在后面将给出这种形式。

如果在正则表达式的末尾包含了一个`$`，那么仅当`text`此时位于末尾时才会匹配成功：

```
if (regex[0] == '$' && regex[1] == '\0')
    return *text == '\0';
```

如果没有包含`$`，并且如果当前不是处于`text`字符串的末尾（也就是说，`*text != '\0'`）并且如果`text`字符串的第一个字符匹配正则表达式的第一个字符，那么到现在为止都是没有问题的；我们将接着判断正则表达式的下一个字符是否匹配`text`的下一个字符，这是通过递归调用`matchhere`函数来实现的。这个递归调用不仅是本算法的核心，也是这段代码如此紧凑和整洁的原因。

如果所有这些匹配尝试都失败了，那么正则表达式和`text`在这个位置上就不存在匹配，因此函数`matchhere`将返回0。

在这段代码中大量地使用了C指针。在递归的每个阶段，如果存在某个字符匹配，那么在随后的递归调用中将执行指针算法（例如，`regex+1` and `text+1`），这样在随后的函数调用中，参数就是正则表达式的下一个字符和`text`的下一个字符。递归的深度不会超过匹配模式的长度，而通常情况下匹配模式的长度都是很短的，因此不会出现耗尽内存空间的危险。

## 1.4 其他的方法

这是一段非常优雅并且写得很好的代码，但并不是完美的。我们还可以做哪些其他的工作？我可能对`matchhere`中的操作进行重新安排，在处理`*`之前首先处理`$`。虽然这种安排不会对函数的执行带来影响，但却使得函数看上去要自然一些，而在编程中一个良好的规则就是：在处理复杂的情况之前首先处理容易的情况。

不过，通常这些判断的顺序是非常重要的。例如，在`matchstar`的这个判断中：

```
} while (*text != '\0' && (*text++ == c || c == '.'));
```

无论在什么情况下，我们都必须推进text字符串中的一个或多个字符，因此在text++中的递增运算一定要执行。

该代码对终止条件进行了谨慎的处理。通常，匹配过程的成功与否，是通过判断正则表达式和text中的字符是不是同时处理完来决定的。如果是同时处理完了，那么就表示匹配成功，如果其中一方在另一方之前被处理完了，那么就表示匹配失败。在下面这行代码中很明显地说明了这个判断。

```
if (regex[0] == '$' && regex[1] == '\0')
    return *text == '\0';
```

但在其他的情况下，还有一些微妙的终止条件。

如果在matchstar函数中需要识别最左边的以及最长的匹配，那么函数将首先识别输入字符c的最大重复序列。然后函数将调用matchhere来尝试把匹配延伸到正则表达式的剩余部分和text的剩余部分。每次匹配失败都会将cs的出现次数减1，然后再次开始尝试，包括处理零出现的情况：

```
/* matchstar: leftmost longest search for c*regexp */
int matchstar(int c, char *regexp, char *text)
{
    char *t;

    for (t = text; *t != '\0' && (*t == c || c == '.'); t++)
        ;
    do { /* * matches zero or more */
        if (matchhere(regexp, t))
            return 1;
    } while (t-- > text);
    return 0;
}
```

我们来看一下正则表达式(.\*)，它将匹配括号内任意长度的text。假设给定了text：

```
for (t = text; *t != '\0' && (*t == c || c == '.'); t++)
```

从开头位置起的最长匹配将会识别整个括号内的表达式，而最短的匹配将会停止在第一次出现右括号的地方。（当然，从第二个左括号开始的最长匹配将会延伸到text的末尾）

## 1.5 构建

《The Practice of Programming》一书主要讲授良好的程序设计。在编写该书时，Rob和我还在贝尔实验室工作，因此我们知道在课堂上使用这本书有什么样的效果。令人高兴的是，

我们发现这本书中的某些内容在课堂上确实有着不错的效果。从2000年教授程序设计中的重点要素时，我们就使用了这段代码。

首先，这段代码以一种全新的形式展示了递归的强大功能及其带来的整洁代码。它既不是另一种版本的快速排序（或者阶乘！）算法，也不是某种树的遍历算法。

这段代码同时还是性能试验的一个很好示例。其性能与系统中的grep并没有太大的差异，这表明递归技术的开销并不是非常大的，因此没有必要对这段代码进行调整。

此外，这段代码还充分说明了优良算法的重要性。如果在模式中包含了几个.\*序列，那么在简单的实现中将需要进行大量的回溯操作，并且在某些情况下将会运行得极慢。

在标准的Unix grep中有着同样的回溯操作。例如，下面这个命令：

```
grep 'a.*a.*a.*a.a'
```

在普通的机器上处理一个4 MB的文本文件要花费20秒的时间。

如果某个实现是基于把非确定有限自动机转换为确定有限自动机，例如egrep，那么在处理恶劣的情况时将会获得比较好的性能；它可以在不到十分之一秒的时间内处理同样的模式和同样的输入，并且运行时间通常是独立于模式的。

对于正则表达式类型的扩展将形成各种任务的基础。例如：

1. 增加其他的元字符，例如+用于表示前面字符的一个或多个出现，或者?用于表示零个或一个字符的匹配。还可以增加一些方式来引用元字符，例如\\$表示在模式中的\$字符。

2. 将正则表达式处理过程分成编译阶段和执行阶段。编译阶段把正则表达式转换为内部形式，使匹配代码更为简单或者使随后的匹配过程运行得更为迅速。对于最初设计中的简单的正则表达式来说，这种拆分并不是必须的，但在像grep这样的程序中，这种拆分是有意义的，因为这种类型的正则表达式要更为丰富，并且同样的正则表达式将会用于匹配大量的输入行。

3. 增加像[abc]和[0-9]这样的类型，这在传统的grep中分别匹配a或b或c和一个数字。可以通过几种方式来实现，最自然的方式似乎就是把最初代码中的char\*变量用一个结构来代替：

```
typedef struct RE {  
    int     type;    /* CHAR, STAR, etc. */  
    int     ch;      /* the character itself */  
    char    *ccl;    /* for [...] instead */  
    int     nccl;    /* true if class is negated [^...] */  
} RE;
```

并且修改相应的代码来处理一个结构数组而不是处理一个字符数组。在这种情况下，并不一定要把编译阶段从执行阶段中拆分出来，但这里的拆分过程是非常简单的。如果学生们把匹

配代码预编译成这个结构，那么总会比那些试图动态地解释一些复杂模式数据结构的学生要做得更好。

为字符类型编写清晰并且无歧义规范是件有难度的工作，而要用代码完美地实现处理更是难上加难，这需要大量的冗长并且晦涩的编码。随着时间的推移，我简化了这个任务，而现在大多数人会要求像Perl那样的速记，例如\d表示数字，\D表示非数字，而不再像最初那样在方括号内指定字符的范围。

4. 使用不透明的类型来隐藏RE结构以及所有的实现细节。这是在C语言中展示面向对象编程技术的好方法，不过除此之外无法支持更多的东西。在实际情况中，将会创建一个正则表达式类，其中类中成员函数的名字像RE\_new( )和RE\_match( )这样，而不是使用面向对象语言的语法。

5. 把正则表达式修改为像各种shell中的通配符那样：匹配模式的两端都被隐含地固定了，\*匹配任意数量的字符，而?则匹配任意的单个字符。你可以修改这个算法或者把输入映射到现有的算法中。

6. 将这段代码转换成Java。最初的代码很好地使用了C指针，并且把这个算法在不同的语言中实现出来是一个很好的实践过程。在Java版本的代码中将使用String.charAt( )（使用索引而不是指针）或者String.substring( )（更接近于指针）。这两种方法都没有C代码整洁，并且都不紧凑。虽然在这个练习中性能并不是主要的问题，但有趣的是可以发现了Java版本比C版本在运行速度上要慢6到7倍。

7. 编写一个包装类把这种类型的正则表达式转换成Java的Pattern类和Matcher类，这些类将以一种与众不同的方式来拆分编译阶段和匹配阶段。这是适配器（Adapter）模式或者外观（Facade）模式的很好示例，这两种模式用来在现有的类或者函数集合外部设置不同的接口。

我曾经大量地使用了这段代码来研究测试技术。正则表达式非常的丰富，而测试也不是无足轻重的，但正则表达式又是很短小的，程序员可以很快地写出一组测试代码来执行。对于前面所列出的各种扩展，我要求学生用一种紧凑的语言写出大量的测试代码（这是“记法”的另一种示例），并且在他们自己的代码中使用这些测试代码；自然我在其他学生的代码中也使用了他们的测试代码。

## 1.6 小结

当Rob Pike最初写出这段代码时，我被它的紧凑性和优雅性感到惊讶——这段代码比我以前所想像的要更为短小并且功能也更为强大。通过事后分析，我们可以看到为什么这段代码如此短小的众多原因。

首先，我们很好地选择了功能集合，这些功能是最为有用的并且最能从实现中体现出核心



思想，而没有多余的东西。例如，虽然固定模式`^`和`$`的实现只需要写3~4行代码，但在统一处理普通情况之前，它展示了如何优雅地处理特殊情况。闭包操作`*`必须出现，因为它是正则表达式中的基本记号，并且是提供处理不确定长度的惟一方式。增加`+`和`?`并不会有助于理解，因此这些符号被留作为练习。

其次，我们成功地使用了递归。递归这种基本的编程技巧通常会比明确的循环带来更短、更整洁的以及更优雅的代码，这也正是这里的示例。从正则表达式的开头和`tex`的开头剥离匹配字符，然后对剩余的字符进行递归的思想，模仿了传统的阶乘或者字符串长度示例的递归结构，但这里是用在了一种更为有趣和更有用的环境中。

第三，这段代码的确使用了基础语言来达到良好的效果。指针也可能被误用，但这里它们被用来创建紧凑的表达式，并且在这个表达式中自然地表达了提取单个字符和推进到下一个字符的过程。数组索引或者子字符串可以达到同样的效果，但在这段代码中，指针能够更好的实现所需的功能，尤其是当指针与C语言中的自动递增运算和到布尔值的隐式转换结合在一起使用时。

我不清楚是否有其他的方法能够在如此少的代码中实现如此多功能，并且同时还要提供丰富的内涵和深层次的思想。



## 第 2 章 我编写过的最漂亮的代码

Jon Bentley

我曾经听一位大师级的程序员这样称赞到，“我通过删除代码来实现功能的提升。”而法国著名作家兼飞行家 Antoine de Saint-Exupéry 的说法则更具代表性，“只有在没有任何功能可以添加，而且也没有任何功能可以删除的情况下，设计师才能够认为自己的工作已臻完美。”某些时候，在软件中根本就不存在最漂亮的代码，最漂亮的函数，或者最漂亮的程序。

当然，我们很难对不存在的事物进行讨论。本章将对经典 Quicksort（快速排序）算法的运行时间进行全面的分析，并试图通过这个分析来说明上述观点。在第一节中，我将首先根据我自己的观点来回顾一下 Quicksort，并为后面的内容打下基础。第二节的内容将是本章的重点部分。我们将首先在程序中增加一个计数器，然后通过不断地修改，从而使程序的代码变得越来越短，但程序的功能却会变得越来越强，最终的结果是只需要几行代码就可以使算法的运行时间达到平均水平。在第三节将对前面的技术进行小结，并对二分搜索树的运行开销进行简单的分析。最后的两节将给出学完本章得到的一些启示，这将有助于你在今后写出更为优雅的程序。

### 2.1 我编写过的最漂亮代码

当 Greg Wilson 最初告诉我本书的编写计划时，我曾自问编写过的最漂亮的代码是什么。这个有趣的问题在我脑海里盘旋了大半天，然后我发现答案其实很简单：Quicksort 算法。但遗憾的是，根据不同的表达方式，这个问题有着三种不同的答案。

当我撰写关于分治（divide-and-conquer）算法的论文时，我发现 C.A.R. Hoare 的 Quicksort 算法（“Quicksort”，Computer Journal 5）无疑是各种 Quicksort 算法的鼻祖。这是一种解决基本问题的漂亮算法，可以用优雅的代码实现。我很喜欢这个算法，但我总是无法弄明白算法中最内层的循环。我曾经花两天的时间来调试一个使用了这个循环的复杂程序，并且几年以来，当我需要完成类似的任务时，我会很小心地复制这段代码。虽然这段代码能够解决我所遇到的问题，但我却并没有真正地理解它。

我后来从 Nico Lomuto 那里学到了一种优雅的划分（partitioning）模式，并且最终编写出了我能够理解，甚至能够证明的 Quicksort 算法。William Strunk Jr. 针对英语所提出的“良好的写作风格即为简练”这条经验同样适用于代码的编写，因此我遵循了他的建议，“省略不必要的字词”（来自《The Elements of Style》一书）。我最终将大约 40 行左右的代码缩减为十几行的代码。因此，如果要回答“你曾编写过的最漂亮代码是什么？”这个问题，那么我的答案就是：在我编写的《Programming Pearls, Second Edition》（Addison-Wesley）一书中给出的 Quicksort 算法。在示例 2-1 中给出了用 C 语言编写的 Quicksort 函数。我们在接下来的章节中将进一步地研究和改善这个函数。

**【示例】** 2-1 Quicksort 函数

```
void quicksort(int l, int u)
{
    int i, m;
    if (l >= u) return;
```

```

swap(l, randint(l, u));
m = l;
for (i = l+1; i <= u; i++)
    if (x[i] < x[l])
        swap(++m, i);
swap(l, m);
quicksort(l, m-1);
quicksort(m+1, u);
}

```

如果函数的调用形式是 `quicksort(0, n-1)`，那么这段代码将对一个全局数组 `x[n]` 进行排序。函数的两个参数分别是将要进行排序的子数组的下标：`l` 是较低的下标，而 `u` 是较高的下标。函数调用 `swap(i, j)` 将会交换 `x[i]` 与 `x[j]` 这两个元素。第一次交换操作将会按照均匀分布的方式在 `l` 和 `u` 之间随机地选择一个划分元素。

在《Programming Pearls》一书中包含了对 Quicksort 算法的详细推导以及正确性证明。在本章的剩余内容中，我将假设读者熟悉在《Programming Pearls》中所给出的 Quicksort 算法以及在大多数初级算法教科书中所给出的 Quicksort 算法。

如果你把问题改为“在你编写那些广为应用的代码中，哪一段代码是最漂亮的？”我的答案还是 Quicksort 算法。在我和 M. D. McIlroy 一起编写的一篇文章(“Engineering a sort function,” Software-Practice and Experience, Vol. 23, No. 11) 中指出了在原来 Unix `qsort` 函数中的一个严重的性能问题。随后，我们开始用 C 语言编写一个新排序函数库，并且考虑了许多不同的算法，包括合并排序 (Merge Sort) 和堆排序 (Heap Sort) 等算法。在比较了 Quicksort 的几种实现方案后，我们着手创建自己的 Quicksort 算法。在这篇文章中描述了我们如何设计出一个比这个算法的其他实现要更为清晰，速度更快以及更为健壮的新函数——部分原因是由于这个函数的代码更为短小。Gordon Bell 的名言被证明是正确的：“**在计算机系统中，那些最廉价，速度最快以及最为可靠的组件是不存在的。**”现在，这个函数已经被使用了 10 多年的时间，并且没有出现任何故障。

考虑到通过缩减代码量所得到的好处，我最后以第三种方式来问自己在本章之初提出的问题。“你没有编写过的最漂亮代码是什么？”。我如何使用非常少的代码来实现大量的功能？答案还是和 Quicksort 有关，特别是对这个算法的性能分析。我将在下一节给出详细介绍。

## 2.2 事倍功半

Quicksort 是一种优雅的算法，这一点有助于对这个算法进行细致的分析。大约在 1980 年左右，我与 Tony Hoare 曾经讨论过 Quicksort 算法的历史。他告诉我，当他最初开发出 Quicksort 时，他认为这种算法太简单了，不值得发表，而且直到能够分析出这种算法的预期运行时间之后，他才写出了经典的“Quicksort”论文。

我们很容易看出，在最坏的情况下，Quicksort 可能需要  $n^2$  的时间来对数组元素进行排序。而在最优的情况下，它将选择中值作为划分元素，因此只需  $\lg n$  次的比较就可以完成对数组的排序。那么，对于  $n$  个不同值的随机数组来说，这个算法平均将进行多少次比较？

Hoare 对于这个问题的分析非常漂亮，但不幸的是，其中所使用的数学知识超出了大多数程序员的理解范围。当我为本科生讲授 Quicksort 算法时，许多学生即使在费了很大的努力之后，还是无法理解其中的证明过程，这令我非常沮丧。下面，我们将从 Hoare 的程序开

始讨论，并且最后将给出一个与他的证明很接近的分析。

我们的任务是对示例 2-1 中的 Quicksort 代码进行修改，以分析在对元素值均不相同的数组进行排序时平均需要进行多少次比较。我们还将努力通过最短的代码、最短运行时间以及最小存储空间来得到最深的理解。

为了确定平均比较的次数，我们首先对程序进行修改以统计次数。因此，在内部循环进行比较之前，我们将增加变量 comps 的值（参见示例 2-2）。

**【示例 2-2】** 修改 Quicksort 的内部循环以统计比较次数。

```
for (i = l+1; i <= u; i++) {
    comps++;
    if (x[i] < x[l])
        swap(++m, i);
}
```

如果用一个值  $n$  来运行程序，我们将会看到在程序的运行过程中总共进行了多少次比较。如果重复用  $n$  来运行程序，并且用统计的方法来分析结果，我们将得到 Quicksort 在对  $n$  个元素进行排序时平均使用了  $1.4 n \lg n$  次的比较。

在理解程序的行为上，这是一种不错的方法。通过十三行的代码和一些实验可以反应出许多问题。这里，我们引用作家 Blaise Pascal 和 T. S. Eliot 的话，“如果我有更多的时间，那么我给你写的信就会更短。”现在，我们有充足的时间，因此就让我们来对代码进行修改，并且努力编写出更短（同时更好）的程序。

我们要做的事情就是提高这个算法的速度，并且尽量增加统计的精确度以及对程序的理解。由于内部循环总是会执行  $u-1$  次比较，因此我们可以通过在循环外部增加一个简单的操作来统计比较次数，这就可以使程序运行得更快一些。在示例 2-3 的 Quicksort 算法中给出了这个修改。

**【示例 2-3】** Quicksort 的内部循环，将递增操作移到循环的外部

```
comps += u-1;
for (i = l+1; i <= u; i++)
    if (x[i] < x[l])
        swap(++m, i);
```

这个程序会对一个数组进行排序，同时统计比较的次数。不过，如果我们的目标只是统计比较的次数，那么就不需要对数组进行实际地排序。在示例 2-4 中去掉了对元素进行排序的“实际操作”，而只是保留了程序中各种函数调用的“框架”。

**【示例 2-4】** 将 Quicksort 算法的框架缩减为只进行统计

```
void quickcount(int l, int u)
{
    int m;
    if (l >= u) return;
    m = randint(l, u);
    comps += u-1;
    quickcount(l, m-1);
    quickcount(m+1, u);
}
```

这个程序能够实现我们的需求，因为 Quicksort 在选择划分元素时采用的是“随机”方式，并且我们假设所有的元素都是不相等的。现在，这个新程序的运行时间与  $n$  成正比，并且相对于示例 2-3 需要的存储空间与  $n$  成正比来说，现在所需的存储空间缩减为递归堆栈的大小，即存储空间的大小与  $\lg n$  成正比。

虽然在实际的程序中，数组的下标 ( $l$  和  $u$ ) 是非常重要的，但在这个框架版本中并不重要。因此，我们可以用一个表示子数组大小的整数 ( $n$ ) 来替代这两个下标（参见示例 2-5）

**【示例 2-5】** 在 Quicksort 代码框架中使用一个表示子数组大小的参数

```
void qc(int n)
{
    int m;
    if (n <= 1) return;
    m = randint(1, n);
    comps += n-1;
    qc(m-1);
    qc(n-m);
}
```

现在，我们可以很自然地把这个过程整理为一个统计比较次数的函数，这个函数将返回在随机 Quicksort 算法中的比较次数。在示例 2-6 中给出了这个函数。

**【示例 2-6】** 将 Quicksort 框架实现为一个函数

```
int cc(int n)
{
    int m;
    if (n <= 1) return 0;
    m = randint(1, n);
    return n-1 + cc(m-1) + cc(n-m);
}
```

在示例 2-4、示例 2-5 和示例 2-6 中解决的都是相同的基本问题，并且所需的都是相同的运行时间和存储空间。在后面的每个示例都对这些函数的形式进行了改进，从而比这些函数更为清晰和简洁。

在定义发明家的矛盾 (inventor's paradox) (How To Solve It, Princeton University Press) 时，George Pólya 指出“计划越宏大，成功的可能性就越大。”现在，我们就来研究在分析 Quicksort 时的矛盾。到目前为止，我们遇到的问题是，“当 Quicksort 对大小为  $n$  的数组进行一次排序时，需要进行多少次比较？”我们现在将对这个问题进行扩展，“对于大小为  $n$  的随机数组来说，Quicksort 算法平均需要进行多少次的比较？”我们通过对示例 2-6 进行扩展以引出示例 2-7。

**【示例 2-7】** 伪码：Quicksort 的平均比较次数

```
float c(int n)
{
    if (n <= 1) return 0;
    sum = 0;
    for (m = 1; m <= n; m++)
        sum += n-1 + c(m-1) + c(n-m);
    return sum/n;
}
```

如果在输入的数组中最多只有一个元素，那么 Quicksort 将不会进行比较，如示例 2-6

中所示。对于更大的  $n$ ，这段代码将考虑每个划分值  $m$ （从第一个元素到最后一个，每个都是等可能的）并且确定在这个元素的位置上进行划分的运行开销。然后，这段代码将统计这些开销的总和（这样就递归地解决了一个大小为  $m-1$  的问题和一个大小为  $n-m$  的问题），然后将总和除以  $n$  得到平均值并返回这个结果。

如果我们能够计算这个数值，那么将使我们实验的功能更加强大。我们现在无需对一个  $n$  值运行多次来估计平均值，而只需一个简单的实验便可以得到真实的平均值。不幸的是，实现这个功能是要付出代价的：这个程序的运行时间正比于  $3n$ （如果是自行参考（self-referential）的，那么用本章中给出的技术来分析运行时间将是一个很有趣的练习）。

示例 2-7 中的代码需要一定的时间开销，因为它重复计算了中间结果。当在程序中出现这种情况时，我们通常会使用动态编程来存储中间结果，从而避免重复计算。因此，我们将定义一个表  $t[N+1]$ ，其中在  $t[n]$  中存储  $c[n]$ ，并且按照升序来计算它的值。我们将用  $N$  来表示  $n$  的最大值，也就是进行排序的数组的大小。在示例 2-8 中给出了修改后的代码。

**【示例 2-8】** 在 Quicksort 中使用动态编程来计算

```
t[0] = 0
for (n = 1; n <= N; n++)
    sum = 0
    for (i = 1; i <= n; i++)
        sum += n-1 + t[i-1] + t[n-i]
    t[n] = sum/n
```

这个程序只对示例 2-7 进行了细微的修改，即用  $t[n]$  来替换  $c(n)$ 。它的运行时间将正比于  $N^2$ ，并且所需的存储空间正比于  $N$ 。这个程序的优点之一就是：在程序执行结束时，数组  $t$  中将包含数组中从元素 0 到元素  $N$  的真实平均值（而不是样本均值的估计）。我们可以对这些值进行分析，从而生成在 Quicksort 算法中统计比较次数的计算公式。

我们现在来对程序做进一步的简化。第一步就是把  $n-1$  移到循环的外面，如示例 2-9 所示。

**【示例 2-9】** 在 Quicksort 中把代码移到循环外面来计算

```
t[0] = 0
for (n = 1; n <= N; n++)
    sum = 0
    for (i = 1; i <= n; i++)
        sum += t[i-1] + t[n-i]
    t[n] = n-1 + sum/n
```

现在将利用对称性来对循环做进一步的调整。例如，当  $n$  为 4 时，内部循环计算总和为：

$t[0]+t[3] + t[1]+t[2] + t[2]+t[1] + t[3]+t[0]$

在上面这些组对中，第一个元素增加而第二个元素减少。因此，我们可以把总和改写为：

$2 * (t[0] + t[1] + t[2] + t[3])$

我们可以利用这种对称性来得到示例 2-10 中的 Quicksort。

**【示例 2-10】** 在 Quicksort 中利用了对称性来计算

```
t[0] = 0
```

```

for (n = 1; n <= N; n++)
    sum = 0
    for (i = 0; i < n; i++)
        sum += 2 * t[i]
    t[n] = n-1 + sum/n

```

然而，在这段代码的运行时间中同样存在着浪费，因为它重复地计算了相同的总和。此时，我们不是把前面所有的元素加在一起，而是在循环外部初始化总和并且加上下一个元素，如示例 2-11 所示。

【示例 2-11】 在 Quicksort 中删除了内部循环来计算

```

sum = 0; t[0] = 0
for (n = 1; n <= N; n++)
    sum += 2*t[n-1]
    t[n] = n-1 + sum/n

```

这个小程序确实很有用。程序的运行时间与 N 成正比，对于每个从 1 到 N 的整数，程序将生成一张 Quicksort 的估计运行时间表。

我们可以很容易地把示例 2-11 用表格来实现，其中的值可以立即用于进一步的分析。在 2-1 给出了最初的结果行。

表 2-1 示例 2-11 中实现的表格输出

N	Sum	t[n]
0	0	0
1	0	0
2	0	1
3	2	2.667
4	7.333	4.833
5	17	7.4
6	31.8	10.3
7	52.4	13.486
8	79.371	16.921

这张表中的第一行数字是用代码中的三个常量来进行初始化的。下一行（输出的第三行）的数值是通过以下公式来计算的：

$$A3 = A2+1 \quad B3 = B2 + 2*C2 \quad C3 = A2-1 + B3/A3$$

把这些（相应的）公式记录下来就使得这张表格变得完整了。这张表格是“我曾经编写的最漂亮代码”的很好的证据，即使用少量的代码完成大量的工作。

但是，如果我们不需要所有的值，那么情况将会是什么样？如果我们更希望通过这种方式分析一部分数值（例如，在 20 到 232 之间所有 2 的指数值）呢？虽然在示例 2-11 中构建了完整的表格 t，但它只需要使用表格中的最新值。因此，我们可以用变量 t 的定长空间来替代 table t[] 的线性空间，如示例 2-12 所示。

【示例 2-12】 Quicksort 计算——最终版本

```

sum = 0; t = 0

```



```

for (n = 1; n <= N; n++)
    sum += 2*t
    t = n-1 + sum/n

```

然后，我们可以插入一行代码来测试  $n$  的适应性，并且在必要时输出这些结果。

这个程序是我们漫长学习旅途的终点。通过本章所采用的方式，我们可以证明 Alan Perlis 的经验是正确的：“简单性并不是在复杂性之前，而是在复杂性之后”（“Epigrams on Programming,” Sigplan Notices, Vol. 17, Issue 9）。

## 2.3 观点

在表 2-2 中总结了本章中对 Quicksort 进行分析的程序。

表 2-2 对 Quicksort 比较次数的统计算法的评价

示例编号	代码行数	答案类型	答案数量	运行时间	空间
2	13	Sample	1	$n \lg n$	$N$
3	13	"	"	"	"
4	8	"	"	$n$	$\lg n$
5	8	"	"	"	"
6	6	"	"	"	"
7	6	Exact	"	$3N$	$N$
8	6	"	$N$	$N^2$	$N$
9	6	"	"	"	"
10	6	"	"	"	"
11	4	"	"	$N$	"
12	4	Exact	$N$	$N$	1

在我们对代码的每次修改中，每个步骤都是很直接的；不过，从示例 2-6 中样本值到示例 2-7 中准确值的过渡过程可能是最微妙的。随着这种方式进行下去，代码变得更快和更有用，而代码量同样得到了缩减。在 19 世纪中期，Robert Browning 指出“少即是多（less is more）”，而这张表格正是一个证明这种极少主义哲学（minimalist philosophy）的实例。

我们已经看到了三种截然不同的类型的程序。示例 2-2 和示例 2-3 是能够实际使用的 Quicksort，可以用来在对真实数组进行排序时统计比较次数。示例 2-4 到示例 2-6 都实现了 Quicksort 的一种简单模型：它们模拟算法的运行，而实际上却没有做任何排序工作。从示例 2-7 到示例 2-12 则实现了一种更为复杂的模型：它们计算了比较次数的真实平均值而没有跟踪任何单次的运行。

我们在下面总结了实现每个程序所使用的技术：

- \* 示例 2-2，示例 2-4，2-7：对问题的定义进行根本的修改。
- \* 示例 2-5，示例 2-6，2-12：对函数的定义进行轻微的修改
- \* 示例 2-8：实现动态编程的新数据结构

这些技术都是非常典型的。我们在简化程序时经常要发出这样的疑问，“我们真正要解

决的问题是什么？”或者是，“有没有更好的函数来解决这个问题？”

当我把这个分析过程讲授给本科生时，这个程序最终被缩减成零行代码，化为一阵数学的轻烟消失了。我们可以把示例 2-7 重新解释为以下的循环关系：

$$C_0 = 0 \quad C_n = (n-1) + (1/n) \sum_{1 \leq i \leq n} C_{i-1} + C_{n-i}$$

这正是 Hoare 所采用的方法，并且后来由 D. E. Knuth 在他经典的《The Art of Computer Programming》(Addison-Wesley) 一书的第三卷：排序与查找中给出的方法中给出了描述。通过重新表达编程思想的技巧和在示例 2-10 中使用的对称性，使我们可以把递归部分简化为：

$$C_n = n-1 + (2/n) \sum_{0 \leq i \leq n-1} C_i$$

Knuth 删除了求和符号，从而引出了示例 2-11，这可以被重新表达为一个在两个未知量之间有着两种循环关系的系统：

$$C_0 = 0 \quad S_0 = 0 \quad S_n = S_{n-1} + 2C_{n-1} \quad C_n = n-1 + S_n / n$$

Knuth 使用了“求和因子”的数学方法来实现这种解决方案：

$$C_n = (n+1)(2H_{n+1} - 2) - 2_n \sim 1.386n \lg n$$

其中  $H_n$  表示第  $n$  个调和数 (harmonic number)，即  $1 + 1/2 + 1/3 + \dots + 1/n$ 。这样，我们就从对程序不断进行修改以得到实验数据顺利地过渡到了对程序行为进行完全的数学分析。

在得到这个公式之后，我们就可以结束我们的讨论。我们已经遵循了 Einstein 的著名建议：“尽量使每件事情变得简单，并且直到不可能再简单为止。”

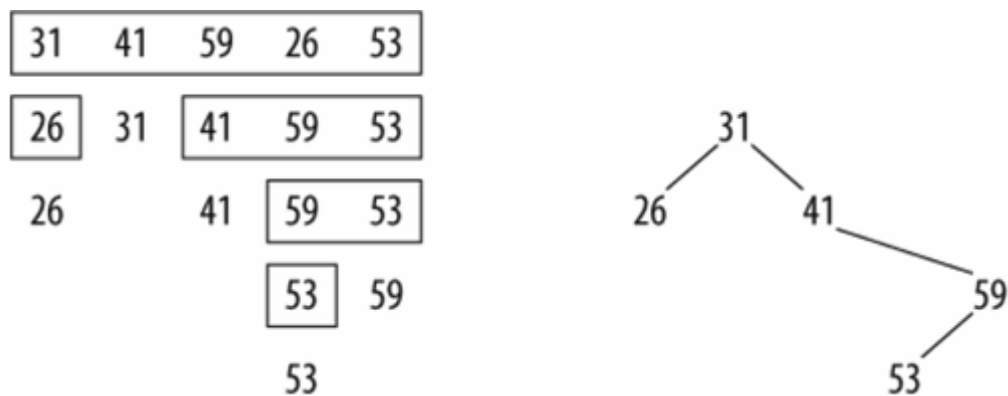
## 附加分析

Goethe 的著名格言是：“建筑是静止的音乐”。按照这种说法，我可以说“数据结构是静止的算法。”如果我们固定了 Quicksort 算法，那么就将得到了一个二分搜索树的数据结构。在 Knuth 发表的文章中给出了这个结构并且采用类似于在 Quicksort 中的循环关系来分析它的运行时间。

如果要分析把一个元素插入到二分搜索树中的平均开销，那么我们可以以这段代码作为起点，并且对这段代码进行扩展来统计比较次数，然后在我们收集的数据上进行实验。接下来，我们可以仿照前面章节中的方式来简化代码。一个更为简单的解决方案就是定义一个新的 Quicksort，在这个算法中使用理想的划分算法把有着相同关联顺序的元素划分到两边。Quicksort 和二分搜索树是同构的，如图 2-1 所示。

图 2-1 实现理想划分的 Quicksort 以及相应的二分搜索树





左边的方框给出了正在进行中的理想划分的 Quicksort，右边的图则给出了相应的从相同输入中构建起来的二分搜索树。这两个过程不仅需要进行相同次数的比较，而且还将生成相同的比较集合。通过在前面对于一组不同元素上进行 Quicksort 实验的平均性能分析，我们就可以得到将不同的元素随机插入到二分搜索树中的平均比较次数。

## 2.4 本章的中心思想是什么？

表面上看来，我“所写的”内容就是从示例 2-2 到示例 2-12 的程序。我最初是漫不经心地编写这些程序，然后将这些程序写在给本科生讲课的黑板上，并且最终写到本章中。我有条不紊地进行着这些程序的修改，并且花了大量的时间来分析这些程序，从而确信它们都是正确的。然而，除了在示例 2-11 中实现的表格外，我从来没有把任何一个示例作为计算机程序运行过。

我在贝尔实验室呆了将近二十年，我从许多教师（尤其是 Brian Kernighan，他所编写的编程内容作为本书的第 1 章）那里学到了：要“编写”一个在大众面前展示的程序，所涉及到的东西比键入这个程序要多得多。有人用代码实现了这个程序，最初运行在一些测试示例中，然后构建了完整的系统框架、驱动程序以及一个案例库来支撑这段代码。理想的情况是，人们可以手动地把编译后的代码包含到文本中，不加入任何的人为干涉。基于这种想法，我编写了示例 2-1（以及在《Programming Pearls》中的所有代码）。

为了维护面子，我希望永远都不要实现从示例 2-2 到示例 2-12 的代码，从而使我保持诚实的名声。然而，在计算机编程中的近四十年的实践使我对这个任务的困难性有着深深的敬畏（好吧，更准确地说，是对于错误的害怕）。我妥协了，把示例 2-11 用表格方式实现出来，并且无意中得到了一个完备的解答。当这两个东西完美地匹配在一起时，你可以想象一下我当时的喜悦吧！因此，我向世界提供了这些漂亮的并且未曾实现的程序，虽然在这些程序中可能会有一些还未发现的错误，但我对这些程序的正确性还是有一定信心的。我希望一些细微的错误不会掩盖我在这些程序中所展示的那些漂亮思想。

当我为给出这些没有被实现过的程序感到不安时，Alan Perlis 的话安慰了我，他说“软件是不是不像任何一个事物，它就是意味着被抛弃：软件的所有意义就是把它看作为一个肥皂泡？”

## 2.5 结论

漂亮的含义有着许多来源。本章通过简化、优雅以及精简来刻画了漂亮的含义。下面这

些名言表达的是同样的意思：

- \* 通过删除代码来实现功能的提升。
- \* 只有在不仅没有任何功能可以添加，而且也没有任何功能可以删除的情况下，设计师才能够认为自己的工作已臻完美。
- \* 有时候，在软件中根本就不存在最漂亮的代码，最漂亮的函数，或者最漂亮的程序。
- \* 良好的写作风格即为简练。省略不必要的字词。（Strunk and White）
- \* 在计算机系统中，那些最廉价、速度最快以及最为可靠的组件是不存在的（Bell）
- \* 努力做到事倍功半。
- \* 如果我有更多的时间，那么我给你写的信就会越短（Pascal）
- \* 发明家的矛盾：计划越宏大，成功的可能性就越大。（Pólya）
- \* 简单性并不是在复杂性之前，而是在复杂性之后（Perlís）
- \* 少即是多。（Browning）
- \* 尽量使每件事情变得简单，并且直到不可能再简单为止（Einstein）
- \* 软件有时候应该被视作为一个肥皂泡（Perlís）
- \* 在简单中寻找漂亮。

本章的内容到此结束。读者可以复习所学到的内容并进行模拟实验。

对于那些想要得到更具体信息的人们，我在下面给出了一些观点，这些观点分为三类

## 程序分析

深入理解程序行为的方式之一就是修改这个程序，然后在具有代表性的数据上运行这个程序，就像示例 2-2 那样。不过，我们通常会更关心程序的某个方面而不是程序的整体。例如，我们只是考虑 Quicksort 所使用的平均比较次数，而忽略了其他的方面。Sedgewick（“The analysis of Quicksort programs,” Acta Informatica, Vol. 7）研究了 Quicksort 的其他特性，例如算法所需的存储空间以及各种 Quicksort 运行时间的其他方面。我们可以关注这些关键问题，而暂时）忽略了程序其他不太重要的方面。在我的一篇文章“A Case Study in Applied Algorithm Design”（IEEE Computer, Vol. 17, No. 2）中指出了我曾经遇到过的问题：对在单元空间中找出货郎行走路线的 **strip 启发式** 算法的性能进行评价。我估计完成这个任务所要的程序大概在 100 行代码左右。在经历了一系列类似于本章前面看到的分析步骤之后，我只使用了十几行代码的模拟算法就实现了更为精确的效果（在我写完了这个模拟算法后，我发现 Beardwood 等人[“The Shortest Path Through Many Points,” Proc. Cambridge Philosophical Soc., Vol. 55]已经更完整地表述了我的模拟算法，因此已经在二十几年前就从数学上解决了这个问题）。

## 小段代码

我相信计算机编程是一项实践性的技术，并且我也同意这个观点：“任何技术都必须通过模仿和实践来掌握。”因此，想要编写漂亮代码的程序员应该阅读一些漂亮的程序以及在编写程序时模仿所学到的技术。我发现在实践时有个非常有用的东西就是小段代码，也就是一二十行的代码。编写《Programming Pearls》这本书是一件艰苦的工作，但同时也有着极大的乐趣。我实现了每一小段代码，并且亲自把每段代码都分解为基本的知识。我希望其他人在阅读这些代码时与我在编写这些代码时有着同样的享受过程。

## 软件系统

为了有针对性，我极其详尽地描述了一个小型任务。我相信其中的这些准则不仅存在于小型程序中，它们同样也适用于大型的程序以及计算机系统。Parnas (“Designing software for ease of extension and contraction,” IEEE T. Software Engineering, Vol. 5, No. 2) 给出了把一个系统拆分为基本构件的技术。为了得用快速的应用性，不要忘了 Tom Duff 的名言：“在尽可能的情况下，利用现有的代码。”

## 2.6 致谢

非常感谢 Dan Bentley, Brian Kernighan, Andy Oram 和 David Weiss 卓有见识的评语。

美丽测试 > 讨厌的二分查找

## 第 3 章 美丽测试

Alberto Savoia

许多程序员都有过这样的经历：看一段代码，觉得它不仅实现了功能，而且实现得很漂亮。通常，如果一段代码能优雅、简洁地完成了需要完成的功能，我们就认为这样的代码很漂亮。

那对于漂亮代码的测试，尤其是那种开发者在编写代码的同时编写的（或者应该编写的）测试，情况又是怎样的呢？在这一章，我将专注于讨论测试，因为测试本身也可以是漂亮的。更重要的是，它们能起到非常关键的作用，可以帮你写出更漂亮的代码。

正如我们将要看到的，有些东西，如果把它们组合起来会使测试很漂亮。跟代码不同的是，我无法让自己认为某个单一的测试很漂亮，至少跟我看待一个排序函数，并认为它漂亮的情况不一样。原因是测试天生就带有组合性和试探性。代码中的每一条 `if` 语句至少需要两个测试（一个用于条件表达式为真的情况，另一个用于为假的情况）。一条拥有多个条件的 `if` 语句，比如：

```
if ( a || b || c )
```

理论上需要 8 个测试——每一个对应 `a`、`b` 和 `c` 不同值的一个可能的组合。如果再考虑循环中的异常，多个输入参数，对外部代码的依赖，不同的软硬件平台等，所需测试的数量和类型将大大增加。

除了最简单的情况，任何代码，不管漂亮与否，都需要一组（而不是一个）测试，这些测试中的每一个都应该专注于检查代码的一个特定的方面，就像球队一样，不同的队员有不同的职责，负责球场的不同区域。

我们已经知道应该以“组”为单位来对测试进行整体评估，现在我们需要进一步了解都有哪些特性能决定一组测试是否漂亮——“漂亮”，一个很少用来修饰“测试”的形容词。

一般来讲，测试的主要目的是逐步建立，不断加强并再次确认我们对于代码的信心：即代码正确并高效地实现了功能。因此对我来讲，最漂亮的测试是那些能将我们的信心最大化

的测试，这个信心就是代码的确实实现了它被要求的功能，并将一直保持这一点。由于代码不同方面的属性需要不同类型的测试来验证，所以对于“漂亮”的评判准则也不是固定的。本章考查了能使测试漂亮的三种方法。

### 测试因简单而漂亮

简单的几行测试代码，使我能描述并验证目标代码的基本行为。通过在每次构建时自动运行那些测试，能确保代码在不断开发的过程中始终保持所要求的行为。本章将使用 **JUnit** 测试框架来给出一些比较基本的测试例子，这些只需几分钟就能编写的测试，将在项目的整个生命周期中使我们不断受益。

测试因揭示出使代码更优雅，更可维护和更易测试的方法而漂亮

换句话讲，测试能帮我们把代码变得更漂亮。编写测试的过程不仅能帮我们找出实现中的逻辑错误，还能帮我们发现结构和设计上的问题。在这一章，通过尝试编写测试，我将演示我是怎样找到了一种能使我的代码更健壮、更有可读性、结构也更好的方法的。

## 测试因其深度和广度而漂亮

深入彻底、覆盖无遗的测试会大大增强开发者的信心，这种信心就是代码不仅在一些基本的、手工挑选的情形下，而且在所有的情形下都实现了所需的功能。在这一章，我将演示怎样根据测试理论中的概念来编写和运行这类测试。

由于大多数程序开发者都已经熟悉了诸如冒烟测试（**smoke testing**）和边界测试（**boundary testing**）等基本的测试技术，我将花更多的时间来讨论更有效类型的测试和那些很少被讨论和应用的测试技术。

### 3.1 讨厌的二分查找

为了演示多种不同的测试技术，同时又保持本章的篇幅合理，需要一个简单、易描述，并能通过几行代码就能实现的例子。同时，这个例子还必须足够生动，拥有一些有趣的挑战测试的特性。最理想的情况是这个例子要有一个悠久的总是被实现出许多 bug 的历史，从而显出对彻底测试的迫切需要。最后但并非最不重要的一点：如果这个例子本身也被认为是漂亮的代码那就再好不过了。

每当讨论漂亮的代码，就很容易让人联想起 Jon Bentley 那本经典的由 Addison-Wesley 出版的《Programming Pearls》（中文名《编程珠玑》，译者注）。我就是在读那本书的时候，发现了我要找的代码例子：二分查找。

让我们快速复习一下，二分查找是一个简单而又高效的算法（但我们即将看到，要正确实现它也是有点难度的），这个算法用来确定一个预先排好顺序的数组  $x[0..n-1]$  中是否含有某个目标元素  $t$ 。如果数组包含  $t$ ，程序返回它在数组中的位置，否则返回 -1。

Jon Bentley 是这样向学生们描述该算法的：

在一个包含  $t$  的数组内，二分查找通过对范围的跟踪来解决问题。开始时，范围就是整个数组。通过将范围中间的元素与  $t$  比较并丢弃一半范围，范围就被缩小。这个过程一直持续，直到在  $t$  被发现，或者那个能够包含  $t$  的范围已成为空。

他又说到：

大多数程序员认为，有了上面的描述，写出代码是很简单的事情。他们错了。能使你相信这一点的惟一方法是现在就合上书，去亲手写写代码试试看。

我 Second Bentley 的建议。如果你从来没有写过二分查找，或者有好几年没写过了，我建议你在继续读下去之前亲手写一下；它会对你后面的内容有更深的体会。

二分查找是一个非常好的例子，因为它是如此简单，却又如此容易被写错。在《Programming Pearls》一书中，Jon Bentley 记述了他是在多年的时间里先后让上百个专业程序员实现二分查找的，而且每次都是在他给出一个算法的基本描述之后。他很慷慨，每次给他们两个小时的时间来实现它，而且允许他们使用他们自己选择的高级语言（包括伪代码）。令人惊讶的是，大约只有 10% 的专业程序员正确地实现了二分查找。

更让人惊讶的是，Donald Knuth 在他的《Sorting and Searching》[1]一书中指出，尽管第一个二分查找算法早在 1946 年就被发表，但第一个没有 bug 的二分查找算法却是在 12 年后才被发表出来。

[注]见《计算机程序设计艺术，第 3 卷：排序和查找（第二版）》，Addison-Wesley，1998。（国内由清华大学出版社出版影印版一译者注）。

然而，最让人惊讶的是，Jon Bentley 正式发表的并被证明过的算法，也就是被实现或改编过成千上万次的那个，最终还是有问题，问题发生在数组足够大，而且实现算法的语言采用固定精度算术运算的时候。

在 Java 语言中，这个 bug 导致一个 `ArrayIndexOutOfBoundsException` 异常被抛出，而在 C 语言中，你会得到一个无法预测的越界的数组下标。你可以在 Joshua Bloch 的 blog 上找到更多关于这个 bug 的信息：

<http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html>

以下就是带有这个著名的 bug 的 Java 实现：

```
public static int buggyBinarySearch(int[] a, int target) {  
    int low = 0;  
    int high = a.length - 1;  
  
    while (low <= high) {  
        int mid = (low + high) / 2;  
        int midVal = a[mid];  
  
        if (midVal < target)  
            low = mid + 1;  
        else if (midVal > target)  
            high = mid - 1;  
        else
```

```
        return mid;
    }
    return -1;
}
```

Bug 位于这一行：

```
int mid = (low + high) / 2;
```

如果low和high的和大于Integer.MAX\_VALUE（在Java中是 $2^{31}-1$ ），计算就会发生溢出，使它成为一个负数，然后被2除时结果当然仍是负数。

推荐的解决方案是修改计算中间值的方法来防止整数溢出。方法之一是用减法——而不是加法——来实现：

```
int mid = low + ((high - low) / 2);
```

或者，如果你想炫耀一下自己掌握的位移运算的知识，那个 blog（还有 Sun 微系统公司的官方 bug report[1]）建议使用无符号位移运算，这种方法或许更快，但对大多数 Java 程序员（包括我）来说，可能也比较晦涩。

[1] [http://bugs.sun.com/bugdatabase/view\\_bug.do?bug\\_id=5045582](http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=5045582)

```
int mid = (low + high) >>> 1;
```

想一下，二分查找算法的思想是多么的简单，而这么多年又有多少人的多少智力花在它上面，这就充分说明了即使是最简单的代码也需要测试，而且需要很多。Joshua Bloch 在它的 blog 中对这个 bug 作了非常漂亮的陈述：

这个 bug 给我上的最重要的一课就是要懂得谦逊：哪怕是最简单的一段代码，要写正确也并非易事，更别提我们现实世界中的系统：它们跑在大段大段的复杂代码上。



下面是我要测试的二分查找的实现。理论上讲，对于中间值的计算方法的修正，应该是解决了这段令人讨厌的代码的最后一个 **bug**，一个在好几十年的时间里，连一些最好的程序员都抓不到的 **bug**。

```
public static int binarySearch(int[] a, int target) {  
    int low = 0;  
    int high = a.length - 1;  
  
    while (low <= high) {  
        int mid = (low + high) >>> 1;  
        int midVal = a[mid];  
  
        if (midVal < target)  
            low = mid + 1;  
        else if (midVal > target)  
            high = mid - 1;  
        else  
            return mid;  
    }  
    return -1;  
}
```

这个版本的 **binarySearch** 看上去是正确的，但它仍可能有问题。或许不是 **bug**，但至少是可以而且应该被修改的地方。这些修改可以使代码不仅更加健壮，而且可读性，可维护性和可测试性都比原来更好。让我们看看是否可以通过测试来发现一些有趣的和意想不到的改善它的机会。

## 3.2 JUnit 简介

谈到漂亮测试，就很容易想到 JUnit 测试框架。因为我使用 Java，通过使用 JUnit 来构建我的漂亮的测试是一个很自然的决定。但在做之前，考虑到你对 JUnit 可能尚未熟悉，让我先对它做一个简单介绍吧。

JUnit 是 Kent Beck 和 Erich Gamma 设计的，他们创造 JUnit 来帮助 Java 开发者编写和运行自动的和自检验的测试。它有一个很简单，却又很宏伟的目标：就是使得程序开发者更容易去做他们本来就应该做的事情：测试自己的代码。

遗憾的是，我们还要走很长的路才能到达那种大多数程序员都像是被“测试病毒”所感染的阶段（在那种情况下，程序员们试着自己编写测试，并决定把它看作开发中的一个常规的重要组成部分）。然而，自从被引入开发领域，任何其他的东西都没能像 JUnit 那样使这么多的程序员开始编写测试。不过这也得感谢极限编程和其他敏捷开发方法的巨大帮助，在这些方法中，开发者参加程序测试是必须的[1]。Martin Fowler 对 JUnit 的影响作了这样的概括：“少量代码对大量的代码起了如此重要的作用，这在软件开发领域中是前所未有的事。”

[1] 能够彰显 JUnit 的成功及影响力的另一个事实是，如今针对大多数现代编程语言的测试框架都出现了，它们都是从 JUnit 那里得到的灵感，JUnit 的各类扩展也出现了。

JUnit 被特地设计得很简单，易学易用。这是 JUnit 的一个重要的设计准则。Kent Beck 和 Erich Gamma 花费了大量心思来确保 JUnit 的易学易用，于是程序员们才会真正使用它。它们自己是这样说的：

我们的第一目标就是要写出一个框架，使我们可以对程序员们真正在其中编写测试抱有希望。这个框架必须使用人们熟悉的工具，这样大家就不用学习很多新东西；必须保证编写一个新的测试所需的工作量降至最低；还必须能够消除重复劳动。[1]

[1] 《JUnit: A Cook's Tour》，Kent Beck，Erich Gamma。

<http://junit.sourceforge.net/doc/cookstour/cookstour.htm>.

JUnit的官方入门文档（the JUnit Cookbook）的长度还不到两页纸：

<http://junit.sourceforge.net/doc/cookbook/cookbook.htm>.

以下是从 **cookbook**（来自 JUnit 的 4.x 版本）中抽取出来的最重要的一段

当你需要测试一样东西时，你只要做：

1. 为一个方法加上 `@org.junit.Test` 标注（annotate）；
2. 当你需要检查一个值，把 `org.junit.Assert` [1]输入进来，调用 `assertTrue()`，并传递一个 `Boolean` 对象，当测试成功时它为 `true`。

比如，为了测试同一币种的两个 **Money** 对象相加时，结果对象中的值恰好等于那两个 **Money** 对象中的值直接相加的结果，你可以这样做：

```
@Test
public void simpleAdd() {
    Money m12CHF= new Money(12, "CHF");
    Money m14CHF= new Money(14, "CHF");
    Money expected= new Money(26, "CHF");
    Money result= m12CHF.add(m14CHF);
    assertTrue(expected.equals(result));
}
```

[1] 能够彰显 JUnit 的成功及影响力的另一个事实是，如今针对大多数现代编程语言的测试框架都出现了，它们都是从 JUnit 那里得到的灵感，JUnit 的各类扩展也出现了。

（为啥上面这段这里又重复了一遍？）

只要你稍微熟悉一点 **Java** 语言，那两条操作指导和这个简单的例子就足以使你上手。它们也足以使你理解我将要写的测试。简单得让人觉得漂亮，是不是？好，我们继续。

### 3.3 将二分查找进行到底

知道了它的历史，我不想被二分查找表面的简单或看似明显的修改所欺骗，尤其是我从来没有在其他代码中用过无符号移位操作符（即>>>）。我将测试这个二分查找的修正版本，就如同我以前从来没有听说过它，也没有实现过它。我不想相信任何人的说辞，测试和证明，说它这一次确实正确。我要通过自己的测试来确信它按照它所应该的方式工作，让他成为一件确凿无疑的事情。

这是我最初的测试策略（或者说测试组）。

- 从冒烟测试开始；
- 增加边界值测试；
- 继续其他各种彻底的、全覆盖类型的测试；
- 最后，添加一些性能测试。

测试一般不会是个线程的过程。我将和你一起再次整理一遍我编写这些测试时的思路，而不是直接把最终的测试集合给你看。

#### 3.3.1 冒烟测试

让我们从冒烟测试开始。这些测试用来确保代码在最基本的使用方式下能够正常工作。它们是第一条防线，是第一组该被编写的测试，因为如果一个实现连冒烟测试都通不过，那更进一步的测试就是在浪费时间。我经常在编写代码前就编写冒烟测试，这叫做测试驱动开发（test-driven development，或 TDD）。

以下是我写的二分查找的冒烟测试：

```
import static org.junit.Assert.*;

import org.junit.Test;

public class BinarySearchSmokeTest {
```

```
@Test

public void smokeTestsForBinarySearch() {

    int[] arrayWith42 = new int[] { 1, 4, 42, 55, 67, 87, 100,
245 };

    assertEquals(2, Util.binarySearch(arrayWith42, 42));
    assertEquals(-1, Util.binarySearch(arrayWith42, 43));
}
}
```

你能看出，这个测试确实非常基本。它本身无法就代码的正确性给人带来很大的信心，但它仍然不失美丽，因为它向着更深入的测试迈出了迅速、高效的第一步。

由于这个冒烟测试执行起来极快（在我的系统中不到百分之一秒），你可能会问我为什么不多包含一些测试。问题的答案是冒烟测试之所以漂亮，部分原因就是大部分的开发结束后，他们仍然能起作用。为了重新确认我对代码的信心——就叫它“信用维护”吧——我喜欢把所有的冒烟测试组成一个测试组（**suite**），并在每一次构建时都运行它们（这种构建每天会有十来次），而且我希望这个冒烟测试组（**test suite**）能够运行得很快——最好在一两分钟内。如果你有几千个类和几千个冒烟测试，保持每一个测试尽可能小就是很重要的事了。

### 3.3.2 边界测试

边界测试，顾名思义，就是用来探测和验证代码在处理极端的或偏门的情况时会发生什么。在二分查找中，两个参数分别是数组和要查找的目标值。让我们为这两个参数分别设想一些连界情况[1]。

[1] 按照二分查找的规格说明，在调用查找函数之前，数组必须是有序的，如果它不是，结果就是未定义的。我们还假定如果数组参数为`null`，代码将抛出一个`NullPointerException`。由于大多数读者对边界测试的基本技术都比较熟悉了，我将跳过一些显而易见的测试。

第一组跑进我大脑的有趣的边界情况跟被查找的数组的长度有关。我从以下这个基本的边界测试开始：

```
int[] testArray;

@Test

public void searchEmptyArray() {

    testArray = new int[] {};

    assertEquals(-1, Util.binarySearch(testArray, 42));

}

@Test

public void searchArrayOfSizeOne() {

    testArray = new int[] { 42 };

    assertEquals(0, Util.binarySearch(testArray, 42));

    assertEquals(-1, Util.binarySearch(testArray, 43));

}
```

很显然，一个空数组就是一个很好的边界例子，长度为 1 的数组也是，因为它是最小的非空数组。这两个测试都很漂亮，它们增加了我的信心：在数组长度边界的下端，程序是正确的。

但我还想用一个非常大的数组来测试二分查找，这就是最有趣的地方了（尤其是我们已经知道了那个 **bug** 仅发生在长度超过 10 亿的数组中）

我的第一个想法是创建一个足够大的数组，大到可以确保整数溢出的 **bug** 已被修正了，但我马上意识到一个“可测试性”的问题：我的便携电脑没有足够的资源在内存中创建那么大的一个数组。但我知道，确实有的系统拥用好几个 **G** 的内存，而且在内存中放置很大的数组。不管通过哪种方法，我要确保的是，中间值在那种情形中不会溢出。

我该怎么办？

我知道,等我完成我设想的其他一些测试之后,我就拥有了足够的测试来给予自己信心:只要中间值被正确的计算而不是溢出为一个负数,基本算法和实现就是对的。以下概括了我的推理,它们得出了一种针对巨大数组的可行的测试方案。

1. 我无法使用一个足够大的数组来直接测试 `binarySearch`, 从而验证中间值计算中的溢出 `bug` 不会发生;
2. 但是,我可以编写足够的测试,让自己确信 `binarySearch` 的实现在小的数组上执行正确;
3. 当用到非常大的数值时,我也能够单独测试计算中间值的方法,这跟数组无关;
4. 于是,假如测试足以使我确信以下两件事情:
  - 只要计算中间值的过程没有问题,我的 `binarySearch` 的实现就是正确的;并且:
  - 计算中间值的过程没有问题。

那么,我可以确信当 `binarySearch` 应用于非常大的数组上时仍然是正确的。

因此,这个并不十分明显,但却很漂亮的测试策略就把讨厌的,易于溢出的计算隔离出来单独测试。

一种方法是编写一个新的函数:

```
static int calculateMidpoint(int low, int high) {  
    return (low + high) >>> 1;  
}
```

然后,把代码中的这一行:

```
int mid = (low + high) >>> 1;
```

变成:

```
int mid = calculateMidpoint(low, high);
```

再然后，就不断测试`calculateMidpoint`方法来确保他永远正确执行。

我已经听到有人在大叫：“为什么在一个为最佳性能而设计的算法中增加一个方法的开销？”别着急，我相信对代码的这一改动不仅仅是可以接受，而且恰恰是正确的，以下就是原因：

1. 如今，我可以信任编译器优化的能力，它会为我把那个方法内联(`inline`)，因此，这里并没有性能损失；
2. 改动提高了代码的可读性。我问过好几个Java程序员，他们中的大多数都不熟悉无符号位移操作，或者对它究竟如何工作没有十足的把握。对这些程序员来说，“`calculateMidpoint(low, high)`”比“`(low + high) >>> 1`”看上去更易理解。
3. 改动还提高了代码的可测试性。

这真是一个好例子，你看到了如何通过编写测试来改善代码的设计并使代码更易理解。换句话说，测试可以使你的代码更漂亮。

下面是针对新的`calculateMidpoint`方法的边界测试的例子：

```
@Test
public void calculateMidpointWithBoundaryValues() {
    assertEquals(0, calculateMidpoint (0, 1));
    assertEquals(1, calculateMidpoint (0, 2));
    assertEquals(1200000000, calculateMidpoint (1100000000,
1300000000));
    assertEquals(Integer.MAX_VALUE - 2,
        calculateMidpoint (Integer.MAX_VALUE-2,
Integer.MAX_VALUE-1));
    assertEquals(Integer.MAX_VALUE - 1,
        calculateMidpoint (Integer.MAX_VALUE-1,
Integer.MAX_VALUE));
}
```



```
}
```

我执行了这组测试，它通过了。很好。现在我可以确信，在我的二分查找所需处理的数组长度的范围内，这个用不熟悉的操作符来计算中间值的方法，正确实现了它的功能，

另一种边界情况是关于目标元素的位置的。我能想出 3 个明显的边界位置：数组中的第一项，数组中的最后一项和数组中不偏不倚正中间的那一项。为此，我编写一个简单的测试来检查这些边界的情况：

```
@Test

public void testBoundaryCasesForItemLocation() {

    testArray = new int[] { -324, -3, -1, 0, 42, 99, 101 };

    assertEquals(0, Util.binarySearch(testArray, -324)); //
first position

    assertEquals(3, Util.binarySearch(testArray, 0));      //
middle position

    assertEquals(6, Util.binarySearch(testArray, 101));    //
last position

}
```

注意在这个测试中我使用了 0 和一些负数，0 和负数不仅包含于数组中，也包含于被查找的目标元素中。在读我以前写的测试时，我发现我曾经只用正数。二分查找算法的描述中并没说所有的数都是正数，因此我应该在我的测试中引入负数和 0。于是，我得到了这么一条关于测试的至理名言：

想出更多测试用例的最好方法就是开始编写测试用例。

当我已开始考虑正数、负数和 0，我想到了另一件不错的事情：一些使用最大和最小整数值的测试。

```
public void testForMinAndMaxInteger() {

    testArray = new int[] {
```

```
Integer.MIN_VALUE, -324, -3, -1, 0, 42, 99, 101,
Integer.MAX_VALUE
    };
    assertEquals(0, Util.binarySearch(testArray,
Integer.MIN_VALUE));
    assertEquals(8, Util.binarySearch(testArray,
Integer.MAX_VALUE));
}
```

到这里,我所想到的所有的边界情况都通过测试了,于是我也开始有非常自信的感觉了。但我马上想起 **Jon Bentley** 班上那 90% 的专业程序员,他们实现了二分查找,并认为自己写对了,但事实上却并没有写对——我的信心顿时又减掉了一些。我是否对输入做了无根据的假设?我一直到写最后这个测试用例时才开始考虑负数和 0。我是否还做过其他无根据的假设?因为我是自己编写的测试,或许我无意间就选择了可以成功的情况,而遗漏了那些可能失败的。

对于程序员测试他们自己写的代码,这是一个普遍存在的问题。如果他们在实现代码时不能想像一些使用场景,那么当他们转而试着去打破代码的时,常常也仍然不能。真正漂亮的测试需要开发者付出额外的努力,跳出旧有的思维模式,探求非常规的使用场景,找寻薄弱的环节,并试着“创新”。

那么,我还有哪些没想到呢?感觉上我的冒烟测试和边界测试并不充分。我的测试集合,再加上一些归纳[1],足够使我宣称我的代码在各种情况下都能正常工作吗?我的脑海中响起 **Joshua Bloch** 的话:“……即使是最简单的代码,要写正确也不容易。”

[1] 我所说的“归纳”(induction)是指从特殊事实或事例推出普遍原理的过程。

什么样的测试能使我足够确信我的代码在面对各种不同的输入时都能正常工作,而不仅仅是针对我所写的几种情况。

### 3.3.3 随机测试

到现在为止我写的都是那种传统的、试了就正确的测试。我用了几个具体的例子来测试查找算法的代码，看它在那些情况中的行为是否跟我预期的一样正确。那些测试全部通过，因此我对我的代码也有了一定的自信。但同时我也意识到我的测试过于具体，对于所有可能的输入，只能覆盖它很小的一个子集。而我想要的、能够使我知道我的代码被全面覆盖，从而夜里可以安然入眠的，是一种对输入情况覆盖更广的测试方法。为达到这个目标，我需要两样东西：

1. 一种能产生各种不同特征的，大数据量的输入集合的方法；
2. 一组能通用于各种的输入集合的断言（**assertion**）。

让我们来解决第一个需求。

这里我所需要的是一种能产生各种不同特征，不同长度的整数数组的方法。惟一需要我做的是保证数组有序，因为这是一个前条件（**precondition**）。除了这个，其他都无所谓。以下是最初的数组产生器实现[1]。

[1] 我之所以说“最初的”，是因为我很快意识到除了正数外，我也需要在数组中包含负数，并因此修改了产生器的实现。

```
public int[] generateRandomSortedArray(int maxArraySize, int
maxValue) {
    int arraySize = 1 + rand.nextInt(maxArraySize);
    int[] randomArray = new int[arraySize];
    for (int i = 0; i < arraySize; i++) {
        randomArray[i] = rand.nextInt(maxValue);
    }
    Arrays.sort(randomArray);
    return randomArray;
}
```

为了实现我的数组产生器，我使用了 `java.util` 包中的随机数产生器和 `Arrays` 类中的一些实用方法。到现在我们一直在解决 Joshua Bloch 在他的 blog 中提到的二分查找的 bug，而这里所用到的 `Arrays.sort` 的实现中恰恰存在完全相同的 bug，但在我所用的 Java 版本中它已经被修正了。我已经在其他测试中包含了对空数组的处理，且结果也令人满意，因此这里我采用的数组的最小长度为 1。这个产生器被设计成了参数化的，因为随着测试的进行，我可能需要创建不同的测试集合：一些是包含大数字的小数组，另一些是包含小数字的大数组，等等。

现在我需要给出一些一般的陈述，来描述那些可被表达为“断言”（`assertion`）的二分查找的行为。所谓“一般”，是指对于任何输入数组和要查找的目标值，这些陈述必须永远为真。我的同事 Marat Boshernitsan 和 David Saff 把这称作推理。这里的思路是：我们有一个关于代码应如何工作的推理，而我们对这个推理的测试越充分，我们对推理的正确性的信心就越大。在下面的例子中，我将应用一个 Saff 和 Boshernissan 推理的一个简化版本。

让我们提出一些有关二分查找的推理，以下就是：

对于 `testArray` 和 `target` 的所有实例——这里 `testArray` 是一个有序的整数数组，而且不为空，`target` 是一个整数——`binarySearch` 必须保证下面的两条永远为真：

推理 1: [1] 如果 `binarySearch(testArray, target)` 返回 -1，那么 `testArray` 不包含元素 `target`；

推理 2: 如果 `binarySearch(testArray, target)` 返回 `n`，`n` 大于或等于 0，那么 `testArray` 包含元素 `target`，且在数组中的位置是 `n`。

[1] 在实际阐述推理时，我会使用，而且会提倡使用叙述性的名字，比如 `binary-SearchReturnsMinusOneImpliesArrayDoesNotContainElement`（“二分查找返回-1说明数组不包含那个元素”），但在这一章，我发现如果我使用推理 1，推理 2 等等，推理将更易理解。

下面是我测试这条推理的代码：

```
public class BinarySearchTestTheories {
```

```
Random rand;

@Before

public void initialize() {
    rand = new Random();
}

@Test

public void testTheories() {

    int maxArraySize = 1000;
    int maxValue = 1000;
    int experiments = 1000;
    int[] testArray;
    int target;
    int returnValue;

    while (experiments-- > 0) {
        testArray = generateRandomSortedArray(maxArraySize,
maxValue);

        if (rand.nextBoolean()) {
            target =
testArray[rand.nextInt(testArray.length)];
        } else {
            target = rand.nextInt();
        }
        returnValue = Util.binarySearch(testArray, target);
        assertTheory1(testArray, target, returnValue);
        assertTheory2(testArray, target, returnValue);
    }
}
```

```
    }  
}  
  
    public void assertTheory1(int[] testArray, int target, int  
returnValue) {  
        if (returnValue == -1)  
            assertFalse(arrayContainsTarget(testArray, target));  
    }  
  
    public void assertTheory2(int[] testArray, int target, int  
returnValue) {  
        if (returnValue >= 0)  
            assertEquals(target, testArray[returnValue]);  
    }  
  
    public boolean arrayContainsTarget(int[] testArray, int  
target) {  
        for (int i = 0; i < testArray.length; i++)  
            if (testArray[i] == target)  
                return true;  
        return false;  
    }
```

在主测试方法**testTherries**中，我首先决定需要运行多少次实验才能确认推理的有效性，然后把那个数字作为我的循环计数器。在循环内，我刚刚实现的随机数组产生器帮我产生出有序数组。成功的和不成功的查找我都要测试，因此我再次使用**Java**的随机数产生器来“掷硬币”（通过**rand.nextBoolean()**）。是选择一个确知存在于数组中的目标数值，还是选择一个不大可能存在于数组中的数值，我根据这种虚拟的硬币投掷做出决定。最后，我调用**binarySearch**，保存返回值，并调用针对现有的推理所设计的验证方法。

注意，为了实现推理的测试，需要写一个测试辅助函数，`arrayContainsTarget`，这个函数给了我另一个检查`testArray`是否包含目标数值的方法。对于这类测试来说，这是一个很常见的做法。尽管这个辅助方法所提供的功能与`binarySearch`相似，但它毕竟是一个简单得多（当然，也慢得多）的查找算法。我对这个辅助函数的正确性充满信心，因此我可以用它来测试一个我不那么有信心的实现。

我的测试从运行 1000 组实验开始，这些实现运行在最大长度为 1000 的数组上。测试花了不到 1 秒钟的时间，而且所有的测试都通过了。很好，现在到了多试探一些数据的时候了（记住测试就是一种充满试探性的活动）。

我修改了实现，并把 `maxArraySize` 的值设成 10 000，接着又设成 100 000。现在测试花费的时间接近一分钟，真该给我的 CPU 打个高分。我感觉我给自己的代码做了一个相当好的测验。

我的信心建立起来了，但我的信仰之一是：如果你的测试全部通过，那常常说明你的测试不够好。有这样一个测试框架，我还应该再测一下代码的哪些方面呢？

我想了一下，发现我的两个推理在形式上都是：

“如果跟 `binarySearch` 的返回值有关的某事为真，那么跟 `testArray` 和目标数值有关另一件事也必须为真。”

换言之，我的逻辑是这种形式的： $p$  隐含  $q$ （或者使用逻辑学上的符号： $p \rightarrow q$ ），这说明，对于我应该测试的东西，我只测试了一半。我还应该拥有  $q \rightarrow p$  [1] 形式的测试。

[1] 当然， $p$ ， $q$ 都可以求反，或者两者同时求反（比如， $\sim p \rightarrow \sim q$ ，或  $p \rightarrow \sim q$ ）。我随意选择了 $p$ 和 $q$ 分别代表与返回值和数组参数有关的谓词。这里关键是要认识到：当你编码的时候，你通常用 $p \rightarrow q$ 的方式思考（如果 $p$ 为真，那么 $q$ 必须发生——即所谓的“开心路线”：正常的，最一般的使用代码的方式）。然而当你测试时，你必须使自己学会逆向的思考方式（ $q \rightarrow ?$ ，或者假如 $q$ 为真，关于 $p$ ，什么必须为真？）和否定的思考方式（假如 $p$ 不为真，也就是 $\sim p$ ，那关于 $q$ ，什么必须为真？）。

如果跟 `testArray` 和目标值有关的某事为真，那么跟返回值有关另一件事必须为真。



这有点难懂，但却很重要，所以让我用一些具体的例子来详细解释一下。针对推理 1 的测试证实了当返回值是-1 时，目标元素不在数组中。但它并不能证实当目标元素不在数组中时，返回值一定是-1。换言之，如果我只测试这一个推理，当目标元素不在数组中时，某个有时返回-1，但并不总是返回-1 的实现，照样能通过所有的测试。针对推理 2，也有类似的问题存在。

我可以用变异测试（mutation test）来演示这个问题，变异测试是 Jeff Offut 发明的一种用来对测试代码进行测试的技术。基本思想就是修改被测的代码，使它带有一些已知的 bug。如果你的测试无视代码中 bug 而依然通过，那就说明这些测试不够全面，无法满足需要。

让我来给 `binarySearch` 做些大的修改，这些修改还带有任意性。我将试着这样做：如果目标值大于 424242 而且并不包含在数组中，我并不返回-1，而是返回-42。这对软件来说，是多么大的破坏啊。看下面的代码的结尾部分：

```
public static int binarySearch(int[] a, int target) {  
    int low = 0;  
    int high = a.length - 1;  
  
    while (low <= high) {  
        int mid = (low + high) / 2;  
        int midVal = a[mid];  
  
        if (midVal < target)  
            low = mid + 1;  
        else if (midVal > target)  
            high = mid - 1;  
        else  
            return mid;  
    }  
    if (target <= 424242)
```

```
        return -1;

    else

        return -42;

}
```

我想，你会同意这个比较大的修改了：如果目标值大于 **424242** 并且不包含在数组中，代码返回了一个出人意料的未指定的值。然而，我们所写的所有的测试都顺利通过了测试。

毫无疑问，我们至少需要再添加两三个推理来使测试更严格，并能捕捉到这种类型的修改。

推理 3：如果 `testArray` 不包含 `target`，它就必须返回-1。

推理 4：如果 `testArray` 在位置 `n` 上包含 `target`，那么 `binarySearch(testArray, target)` 必须返回 `n`。

对这些推理的测试如下：

```
public void assertTheory3(int[] testArray, int target, int
returnValue) {
    if (!arrayContainsTarget(testArray, target))
        assertEquals(-1, returnValue);
}
```

```
public void assertTheory4(int[] testArray, int target, int
returnValue) {
    assertEquals(getTargetPosition(testArray, target),
returnValue);
}
```

```
public int getTargetPosition(int[] testArray, int target) {
    for (int i = 0; i < testArray.length; i++)
```

```
        if (testArray[i] == target)
            return i;
    return -1;
}
```

注意我必须再编写一个辅助方法：`getTargetPosition`。这个方法拥有跟`binarySearch`完全相同的行为（但我确信它可以正确工作，缺点是它需要 $n$ 次而不是 $\log_2 n$ 次的比较）。由于`getTargetPosition`跟`arrayContainsTarget`很相似，代码重复是不好的，所以我把后者重新编写如下：

```
public boolean arrayContainsTarget(int[] testArray, int
target) {
    return getTargetPosition(testArray, target) >= 0;
}
```

我使用我的随机数组产生器再次运行了这些测试，这次返回-42的修改被马上测了出来。很好，这增加了我对代码的信心。我从代码中除去了这个故意设置的 `bug` 并再次运行测试。我期望它们会通过，但它们没有。一些针对推理 4 的测试没有通过。JUnit 失败了，并给出了如下形式的信息：

```
expected:<n> but was:<n + 1>
```

推理 4 提到：

如果`testArray`在位置 $n$ 上包含`target`，那么`binarySearch(testArray, target)`必须返回 $n$ 。

然而，在有些情况下，查找函数返回了  $n$  的下一个位置。这怎么可能？

我需要更多一些的数据。JUnit的断言（`assertion`）可以接受一个`String`类型的消息作为第一个参数，因此我修改了推理 4 的`assertEquals`，让它包含一些能在失败时给我更多信息的文本。

```
public void assertTheory4(int[] testArray, int target, int
returnValue) {
    String testDataInfo = "Theory 4 - Array=" +
        printArray(testArray)
        + " target="
        + target;
    assertEquals(testDataInfo, getTargetPosition(testArray,
target), returnValue);
}
```

现在，只要推理 4 没有通过，JUnit 就会把数组的内容连同目标值一起显示给我。我重新运行了测试（这次 `maxArraySize` 和 `maxValue` 的值都被设小，从而使输出更易读），得到的结果如下：

```
java.lang.AssertionError: Theory 4 - Array=[2, 11, 36, 66, 104, 108, 108, 108,
122,
155, 159, 161, 191] target=108 expected:<5> but was:<6>
```

我知道问题发生在哪里了。推理 4 没有将重复值纳入考量，我也没考虑它。数组中有 3 个 108。我想我需要找到关于处理重复值的说明，然后要么修改我的代码，要么修改我的推理，接着再测试。不过我将把这个作为练习留给读者（我总是想那样说！），因为我说得已经不少了，而且在我们合上这一章之前我还想再讲一点与性能测试有关的东西。

### 3.3.4 性能测试

我们已经运行过的、基于那些推理的测试给我们代码包上了一层很紧的保护网。一个有 bug 的实现还能通过所有的测试吗？我想这样的情况很难再存在了。但是，我们还忽略了一点。我们所拥有的测试对于查找来说都是很好的测试，但我们正在测的不是别的，而是二分查找。我们需要一组测试来验证“二分”特性。我们需要看到我们的实现所执行的比较次数是否达了最大  $\log_2 n$  的预期。如何才能做到这一点呢？

我首先想到的是利用系统时钟，但我很快打消了这个想法，原因是针对这个具体的问题来说，我所能用的时钟没有足够的精确度（二分查找的速度实在太快了），而我又不能真正控制运行环境。因此，我使用了另一个测试技巧：我创建了一个新的、叫做 `binarySearchComparisonsCount` 的 `binarySearch` 实现。这个版本的代码在程序逻辑上跟原来是一样的，但它维护并返回了一个比较计数，而不是返回-1 或者目标值的位置。<sup>[1]</sup> 代码如下所示：

[1] 相对于直接修改 `binarySearch` 来返回比较计数，（David Saff提出的）一个更好、更清晰、也更面向对象的设计是创建一个 `CountingComparator` 类，这个类实现Java所泛化出来的 `Comparator` 接口，然后修改 `binarySearch`，接收这个类的一个实例作为第 3 个参数。这种方法也泛化了 `binarySearch`，使它可用于整型以外的其他类型——又是一个关于测试怎样使设计更优良、代码更漂亮的好例子。

```
public static int binarySearchComparisonCount(int[] a, int
target) {
    int low = 0;
    int high = a.length - 1;

    int comparisonCount = 0;

    while (low <= high) {

        comparisonCount++;

        int mid = (low + high) >>> 1;
        int midVal = a[mid];

        if (midVal < target)
            low = mid + 1;
        else if (midVal > target)
```

```
        high = mid - 1;
    else
        return comparisonCount;
    }
    return comparisonCount;
}
```

基于这个代码，我给出了另外一个推理：

推理 5：如果testArray的长度为n，那么*binarySearchComparisonCount(testArray, target)*必须返回一个小于或等于  $1 + \log_2 n$  的数。

以下是针对这个推理的测试代码：

```
public void assertTheory5(int[] testArray, int target) {
    int numberOfComparisons =
        Util.binarySearchComparisonCount(testArray, target);
    assertTrue(numberOfComparisons <= 1 +
log2(testArray.length));
}
```

我把最后的这个推理添加到testTheories方法的推理列表中，结果就像这样：

```
while (experiments-- > 0) {
    testArray = generateRandomSortedArray( );
    if (rand.nextInt( ) % 2 == 0) {
        target =
testArray[rand.nextInt(testArray.length)];
    } else {
        target = rand.nextInt( );
    }
    returnValue = Util.binarySearch(testArray, target);
    assertTheory1(testArray, target, returnValue);
}
```

```
        assertTheory2(testArray, target, returnValue);  
        assertTheory3(testArray, target, returnValue);  
        assertTheory4(testArray, target, returnValue);  
        assertTheory5(testArray, target);  
    }  
    ...
```

我把maxArraySize分别设成不同的值来跑了几个测试，发现推理 5 好像不成立。

因为快到中午，我把实验数字设成 1 000 000，然后就去吃午饭了，我的电脑则继续运行，把每个理论测试一百万次。

当我回来的时候，我看到所有的测试都通过了。或许还有其他几件事情需要我去测试，但针对这个 `binarySearch` 的实现，我的信心已大大增加了。由于不同开发者拥有不同的背景、风格和经验水平，你也可以关注于代码的不同方面。比如，在一个已熟练掌握无符号位移操作的开发者看来，这个测试的必要性并不一定有所认为的那么大。

在这一节，我想让你们感受一下性能测试，同时告诉你们如何通过将代码和测试理论相结合，来获得对代码性能的洞察力以增强信心。我强烈建议你学习一下第 3 章，在那里 Jon Bentley 详细讨论了这个话题，并给出了漂亮的处理方法。

### 3.4 结论

在这一章，我们看到即使是最优秀的程序员和最漂亮的代码也仍然能从测试中获益。我们也看到了编写测试代码可以跟编写目标代码本身一样需要创造力，也一样具有挑战性。而且，我也希望我向你们展示了测试自身，至少在 3 个方面，也可被认为是漂亮的。

有些测试因为简单和高效而漂亮。只需几行，每次随构建自动运行的 JUnit 代码，你就能描述你想要的代码的行为和边界情况，并且能确保在代码不断开发的过程中，这些行为和边界情况一直保持。



另外一些测试，它们的漂亮是因为它们在被编写的过程中，能够通过一些细小却又很重要的方式，帮助你改善被测代码的质量。它们可能并不能发现常规的 **bug** 或缺陷，但它们能让代码中的问题浮出水面，这些问题可能是设计、可测试性和可维护性方面的问题；它们能帮助你把代码变得更漂亮。

最后，有些测试的漂亮是因为它们的覆盖面和彻底性。他们帮你获得信心：即代码在功能和性能上都达到了要求和期望，不仅仅是针对几个手工挑选的情况，而是针对大范围的输入数据和条件。

希望写出漂亮代码的开发者可以向艺术家们学习一些东西。画家常常放下手中的画笔，然后远离画布一段距离，围着它转一转，翘起脑袋，斜着看看，再从不同的角度看看，在不同的光线下看看。在寻求美的过程中，他们需要设计这样一些视角并使他们融为一体。如果你的画布是个集成开发环境（**IDE**）而你的媒介就是代码，想一想，你如何做到离开画布一段距离，用挑剔的眼光从不同的视角来审视你的作品？——这将使你成为一个更优秀的程序员，并帮你写出美丽的代码。

## 第 4 章 NASA 火星漫步者任务中的高可靠企业系统

Ronald Mak

你是否经常听到有人说“情人眼里出西施”？在这里，“情人”指的是 NASA 火星探测漫步者任务，它有着非常严格的需求，这个任务中的软件系统必须是实用的、可靠的以及稳定的。哦，这个软件还必须严格地按时交付——火星才不会接受任何延期的理由。当 NASA 在谈论满足“发射窗口（Launch Windows）”的时候，这意味要满足多个方面！

本章描述的是协作式信息管理系统（Collaborative Information Portal, CIP）的设计和开发，CIP 是一个由 NASA 开发的庞大的企业信息系统，由任务管理人员、工程师以及世界各地的科学家们使用。

火星人不会容忍任何丑陋的软件。对于 CIP 来说，漂亮的概念并不只是局限于那些优雅算法或者令人叹为观止的程序。CIP 的漂亮性体现在它是由一些熟练的编码人员构建起来的复杂软件，而这些编码人员只需知道如何把系统的各个组件组合起来。大型应用程序的漂亮性体现在多个方面，而小型程序却往往做不到这一点。这不仅是因为在大型程序中有着更多的必要性，而且还因为在大型程序中存在漂亮性的机率更大——在大型程序中经常要做一些小型程序无需去做的工作。我们将首先浏览一下 CIP 的整个基于 Java 的面向服务架构，然后通过对其中的一个服务进行案例分析，进而研究一些代码和一些使系统满足实用性、可靠性以及稳定性等需求的技术。

你可以想像到，用于 NASA 太空任务的软件必须是高度可靠的。因为这些任务都是很昂贵的，这些耗费多年的计划和数百万美元的任务不能因为有故障的程序而受到威胁。当然，在软件工作中最困难的部分就是，对那些在离地球数百万英里之外的航空器上运行的软件进行调试和修改。此外，基地系统（ground-base system）也必须是可靠的，没有人希望由于软件 bug 而导致任务操作中断或者丢失有价值的信息。

讨论关于这种软件的漂亮性有些不可思议。在一个多层次的面向服务架构中，服务被实现为驻留在服务器上的中间件。（我们在中间件中开发了共享的可重用组件，这极大地节约了开发时间）。这些中间件把客户端程序与后端数据源解耦开来；换句话说，应用程序并不需要知道它所需的数据是如何存储的，以及存储在什么地方。当所有的中间件服务器都正常工作时，这个企业系统的终端用户甚至不会意识到他们的客户端程序正在发出远程服务请求。当中间件运行得很顺畅时，用户会认为他们正在直接访问数据源，并且所有的数据处理都是在他们的工作站或者笔记本上进行的。因此，中间件越成功，它们就会变得越透明，漂亮的中间件应该是完全不可见的！

### 4.1 任务与 CIP

火星探测漫步者任务，或者简称为 MER，它的主要目标是探索在火星表面上是否曾经存在过液态水。在 2003 年的 6 月和 7 月，NASA 向火星发射了两个相同的漫步者，即地质机器人。在 2004 年 1 月，在度过了 7 个月的旅途后，它们在火星的背面登录了。

每个漫步者都配有太阳能动力以在火星表面自行驱动。每个漫步者都配备了一些科学仪器，例如安装在机械臂终端的光谱仪。在这些机械臂上有一个钻头和一个显微影像仪，用

来分析岩石表面下的成份。每个漫步者都安装了数台相机和天线，用来把数据和影像发送回地球（参见图 4-1）

图 4-1 火星漫步者（JPL 提供的免费图像）



在无人驾驶的 NASA 任务中包含了硬件和软件。在每台火星漫步者上都有不同的软件包来独立地控制操作，并且对远在加利福尼亚州 Pasadena 附近的 NASA 喷气推进实验室 (Jet Propulsion Laboratory, JPL) 的任务控制中心发出的远程命令作出响应。在任务控制中心的基地软件包使得管理人员、工程师和科学家能够下载和分析由漫步者发回来的数据，并且设计和开发新的命令序列发送给漫步者，以及与其他人员进行协作。

在加利福尼亚州 Mountain View 附近的 NASA Ames 研究中心，我们为 MER 任务设计并开发了协作式信息管理系统 (CIP)。这个项目团队包括 10 名软件工程师和计算机科学家。另外还有 9 名包括项目经理和技术支持工程师的团队成員，主要负责 QA、软件系统构建、硬件配置和 bug 跟踪等任务。

## 4.2 任务需求

CIP 被设计用来满足 MER 任务的三个主要需求。通过满足这些需求，CIP 在任务人员之间提供了重要的“环境感知 (situational awareness)”功能：

### 时间管理

在任何大型的复杂任务中，使所有工作人员保持同步对于任务的成功来说是非常重要的，此外，在 MER 中还给出了一些特殊的时间管理需求。由于任务中的工作人员分布在世界各地，因此 CIP 需要以多个地方的时区来显示时间。而且由于漫步者是在火星的背面登录的，因此还有两个火星的时区。

最初，任务是按照火星时间来运行的，这意味着所有会议和活动（例如从火星上下载数

据)的时间都是参照某台漫步者所处的火星时区来制定的。火星上的一天比地球上的一天大概长 40 分钟,并且与地球时间非常相似,因此任务中的工作人员每天都推迟这段时间,从而与他们使用地球时间的家人和同事保持一致。这些都使得 CIP 的时间管理功能变得更为重要。

## 人员管理

随着在两个漫步者团队(每个漫步者都配有一个团队)之间人员的更迭,对每个成员变动进行跟踪变成了另一项重要的任务。CIP 管理着一个人员名册,并且记录了每个成员的工作地点,工作时间以及工作角色。

CIP 在任务人员之间还实现了一些协作。他们可以发出广播消息,共享对数据和影像的分析,上传报告以及对其他人的报告进行注释。

## 数据管理

对于 NASA 的每个行星任务和深太空任务(deep space mission)来说,从宇宙的遥远之处获得数据和影像都是非常重要的,在 MER 中也不例外。在地面上有一组天线用于接收从火星漫步者发出的数据和影像,并且当这些数据和影像到达地球后,将被传输到 JPL,并且在任务文件服务器上进行处理和存储。

当任务管理者发布已处理的数据和影像文件时,CIP 会生成元数据,以将数据按照不同的标准进行分类,这些标准包括哪一台漫步者仪器生成的数据,哪一台照相机拍摄的影像,在什么环境下,使用何种配置,以及时间和地点等。然后,CIP 用户可以根据这些标准来搜索数据和影像,并且通过互联网把这些文件从任务的文件服务器下载到个人笔记本和工作站上。

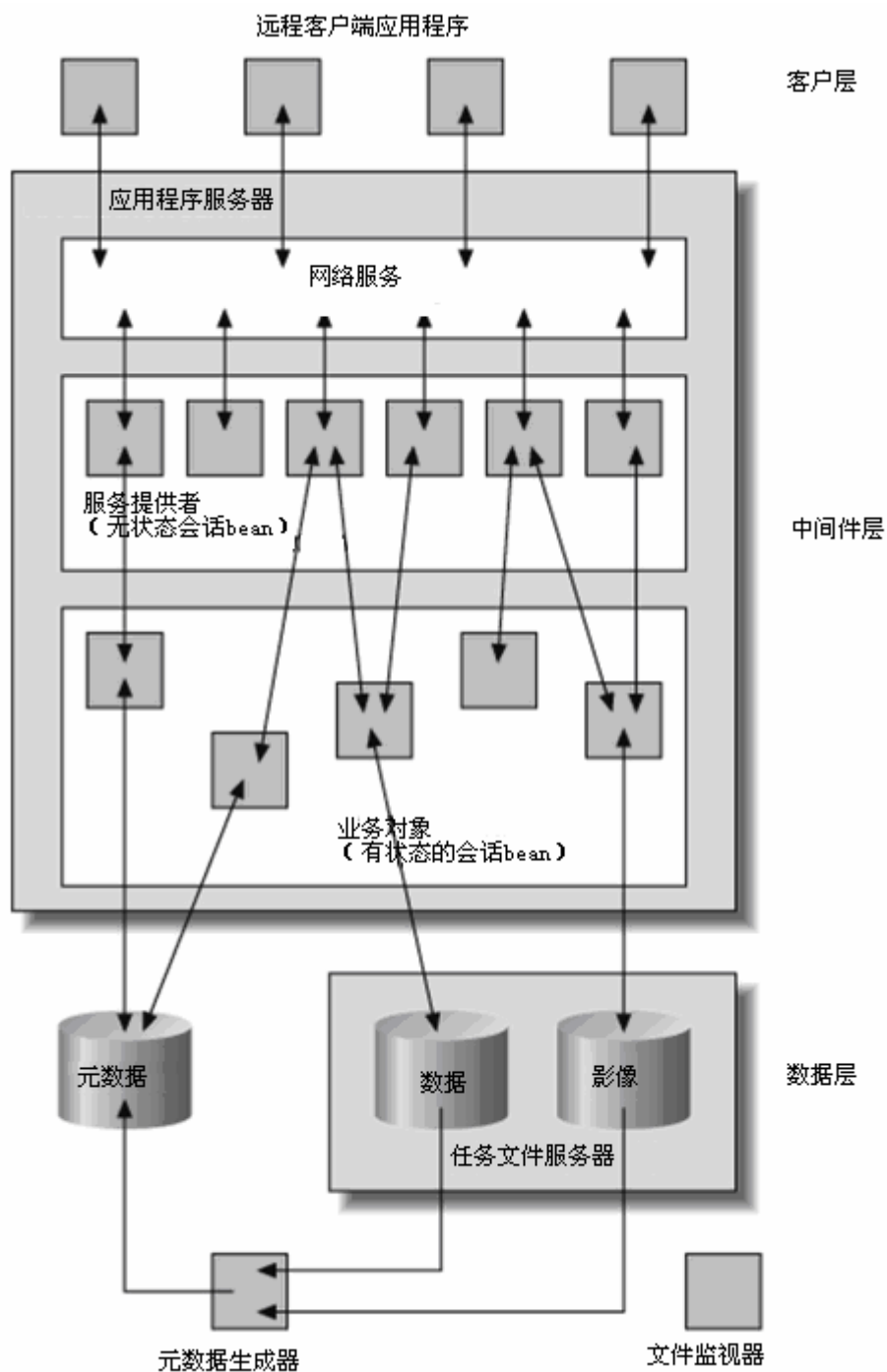
在 CIP 中还实现了数据安全。例如,根据用户角色的不同(以及她是否是美国公民),一些特定的数据和影像是禁止访问的。

## 4.3 系统架构

一个企业系统代码的漂亮性部分取决于系统的架构,即代码组织的方式。架构不仅仅是一种美学。在大型应用程序中,架构确定了软件组件之间的交互方式,并将对整个系统的可靠性产生影响。

我们使用了一个三层的面向服务架构(service-oriented architecture, SOA)来实现 CIP。我们坚持遵循工业标准和最佳实践,在所有可能的地方都使用了标准商用(commercial off-the-shelf, COST)软件。大多数程序都是用 Java 编写的,我们使用了 Java 2 企业版本(J2EE)标准(参见图 4-2)

图 4-2 CIP 的三层面向服务架构



在客户层包含了大部分基于 GUI 的独立 Java 应用程序，它们是用 Swing 组件和一些基于网页的程序开发的。符合 J2EE 标准的应用程序服务器将在中间件中运行，并且提供用于响应客户端应用程序请求的所有服务。我们使用了企业 Java Bean (EJB) 来实现这些服务。数据层包括数组源和数据工具软件。这些工具软件同样是用 Java 编写的，用来监视存放已处理的数据和影像的文件服务器。只要任务管理人员一发布新的数据和影像文件，这些工具软件将立刻生成这些文件的元数据。

通过使用基于 J2EE 的 SOA，我们可以在大型企业应用程序设计中合适的地方使用这些定义良好的 Java Bean (以及其他一些组件)。无状态的会话 bean 在处理服务请求时不会保存从一个请求到下一个请求之间的任何状态。而有状态的会话 bean 将会为客户端维护状态



信息，并且通常还会管理 bean 在某个数据存储中读取或者写入的持久信息。在开发大型的复杂应用程序时，如果在任意的设计情形中都能够有多种选择，那么将会是很有用的。

在中间件中，我们为每个服务都实现了一个无状态的会话 bean 以作为服务的提供者。这里使用的是外观（facade）设计模式，我们通过这种模式生成了一个 web service，客户端应用程序将向这个 web service 发出请求并且获得响应。每个服务还可以访问一个或者多个有状态的会话 bean，这些 bean 是一些业务对象，用来提供在多个服务请求之间维护状态所需要的逻辑，例如在响应数据请求时，下一个数据块将在数据库的什么位置上读取信息。事实上，无状态的 bean 通常是那些进行实际工作的有状态 bean 的服务分发器。

在这种类型的架构中存在着大量的漂亮性！这个架构体现了一些重要的设计原则：

#### 基于标准

一些研究机构（像 NASA Ames 研究中心，也就是我们设计和开发 CIP 的地方）非常喜欢去发明一些新的东西，即使最终的结果是重复已有的成果。而 MER 任务既没有为 CIP 开发团队提供时间，也没有提供资源，并且我们的目标是为这个任务开发一个具有产品质量的代码，而不是做研究工作。

在所有大型应用程序中，成功的关键是集成，而不是编码。在遵循工业标准和最佳实践背后所带来的漂亮性，就是我们通过 COTS 组件减少了编码的工作，并且这些接口有很强的通用性，因此各个组件能够很好地在一起工作。这也确保了我们能够向任务管理人员按时交付实用的和可靠的代码。

#### 松散的耦合

我们在客户应用程序和中间件服务之间实现的是松散的耦合。这意味着只要开发应用程序的程序员和开发服务程序的程序员在某个接口上达成了一致，他们就可以并行开发各自的代码。只要接口保持稳定，那么其中任何一方的任何修改都不会影响到另一方。松散的耦合是使我们能够按时交付大型的多层 SOA 应用程序另一个主要的因素。

#### 语言的独立性

客户端应用程序和中间件服务使用 web service 进行彼此之间的通信。web service 协议是一个独立于语言的工业标准。虽然大多数的 CIP 客户应用程序都是用 Java 语言编写的，但这个中间件同样可以为一些用 C++ 或者 C# 语言编写的应用程序提供服务。如果我们能够使服务和 Java 客户端在一起工作，那么就可以很容易地将这个服务与另一个用其他语言编写的并且支持 web service 的客户端放在一起工作。这极大地扩展了 CIP 的可用性和有效性，而消耗的时间和资源是非常少的。

#### 模块化

随着应用程序代码量的增长，模块化的重要性将呈指数增长。在 CIP 中，每个服务都是完备的中间件组件，独立于其他的服务。如果某个服务需要与其他服务联合工作，它可以向其他服务发出服务请求，就好像它是一个客户程序。这使得我们能够独立地创建服务，并且在我们的开发工作中增加另一层并行。模块化的服务是漂亮的，在成功开发的大型 SOA 应用程序中经常可以看到。

在客户层，应用程序通常混合使用多个服务，或者把多个服务的返回结果组合在一起，或者使用某个服务的结果作为请求传递给另一个服务。

#### 可伸缩性

在任务控制中心发布已处理的数据和影像文件时，尤其是当一台漫步者获得了有趣的发现之后，对 CIP 的使用将出现峰值。我们必须确保 CIP 的中间件能够处理这样的峰值，尤其是当用户正在很焦急地等待下载和查看最新文件的时候。如果在这些时候系统的速度变慢，或者出现更糟糕的情况，系统崩溃，都会是令人无法忍受和难以面对的。

J2EE 架构的漂亮性之一在于它处理可伸缩性的方式。在应用服务器中将维护一个 bean

池，并且根据需求能够自动创建多个实例的无状态会话 **bean** 服务提供者。这是一个很重要的“免费”J2EE 功能，中间件服务的开发人员非常乐于接受这个功能。

可靠性

作为一个经过工业界严格检验的标准，J2EE 的架构被证明是非常可靠的。我们避免了超越设计范围的情况。因此，在经过两年的运行之后，CIP 创造了正常运行时间超过 99.9% 的记录。

我们超越了 J2EE 本身所提供的可靠性。正如你将在随后的案例分析中看到的，我们在服务中进行了一些额外的改进以进一步提升可靠性。

## 4.4 案例分析：流服务

你已经从架构中看到了 CIP 的一些漂亮性。现在我们来关注其中的一个中间件服务——一流服务——并将其作为案例进行分析，我们将研究一些能够使任务满足严格的实用性、可靠性以及稳定性等需求的技术。你将看到这些技术并不是特别奇特的；系统的漂亮性在于在什么地方使用这样的技术。

### 4.4.1 实用性

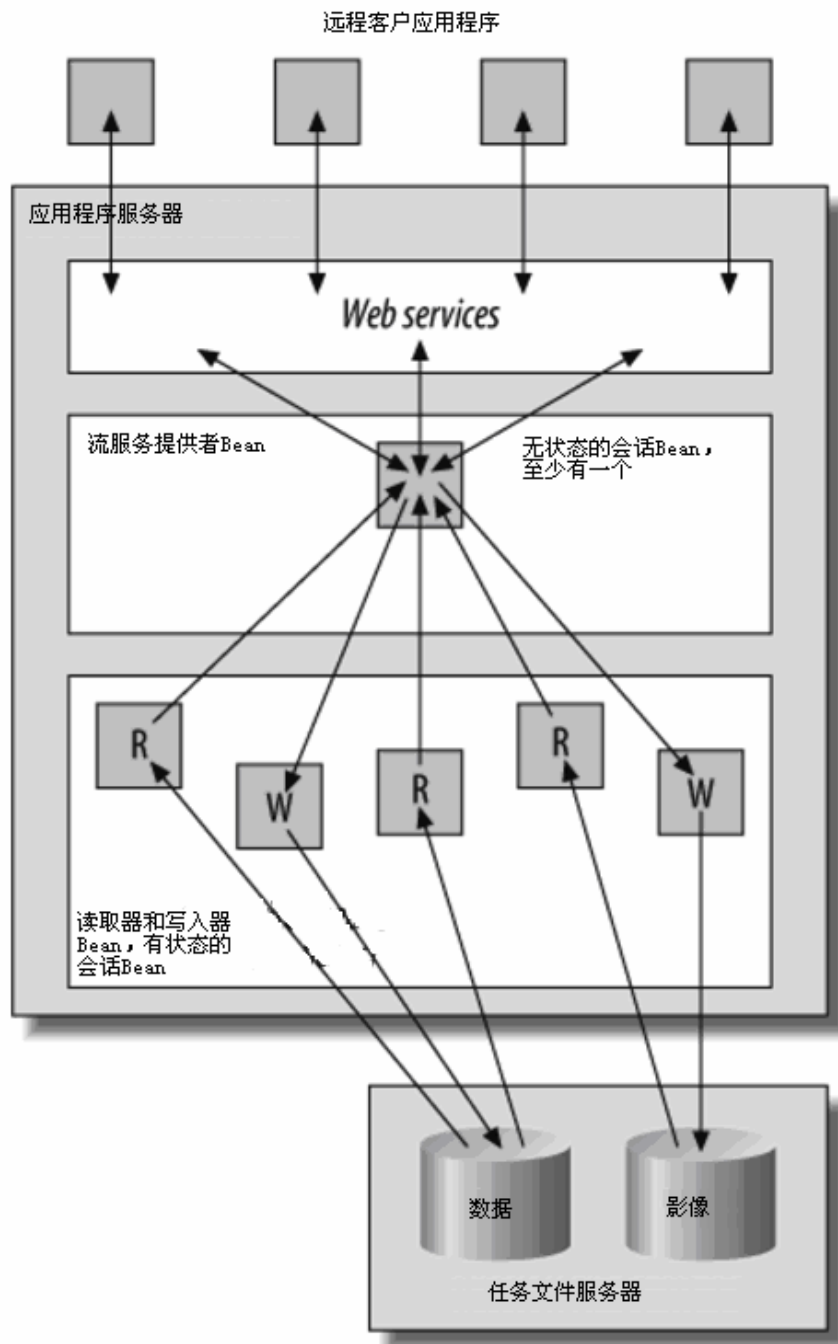
MER 任务的数据管理需求之一就是允许用户把位于 JPL 任务文件服务器上的数据和影像文件下载到他们的个人工作站或者笔记本上。正如前面所谈及的，CIP 的数据层工具生成元数据，使用户可以基于不同的标准来检索他们想要的文件。同样，用户也可以把包含他们分析的报告上传到服务器。

CIP 的流服务用来执行文件的下载和上传工作。我们之所以把这个服务称之为流服务，是因为这个服务的功能是使数据能够通过互联网在 JPL 的任务文件服务器和用户本地电脑之间安全地流动。这个服务使用了 **web service** 协议，因此客户端应用程序可以用任何一种支持这种协议的语言来编写，并且这些应用程序可以自由地设计适合它们自己的 GUI。

### 4.4.2 服务架构

和所有其他的中间服务一样，在流服务中使用了 **web service** 来接收客户端的请求并返回响应。每个请求最初是由流服务提供者来填充内容，这个提供者是用一个无状态的会话 **bean** 来实现的。服务提供者创建了一个利用有状态的会话 **bean** 实现的文件读取器来进行实际的工作，从而把被请求的文件内容下载到客户端。相应地，服务提供者还创建了一个文件写入器，这同样是用有状态的会话 **bean** 来实现的，用于上传文件内容（参见图 4-3）。

图 4-3 流服务的架构



在任意时刻，多个用户可以同时下载或者上传文件，并且任何单个用户都可以同时进行多个下载或者多个上传操作。因此，在中间件中可能同时存在着多个文件读取器和文件写入器。单个无状态的流服务提供者 bean 能够处理所有这些请求，除非负载变得很沉重，此时应用程序服务器可以创建更多的提供者 bean。

为什么每个文件读取器和文件写入器都必须是一个有状态的会话 bean？这是因为在响应来自客户端的“读取数据块”或者“写入数据块”的请求时，除非这个文件很小，否则流服务将对这个文件的内容进行分块并且每次传输一块内容。（下载块的大小是在中间件服务器上配置的，而上传块的大小则是由客户端应用程序来指定的）。在一个请求到下一个请求之间，有状态的 bean 将记录在任务文件服务器上打开的源文件或者目标文件，以及在文件中下一个将要进行读取块或者写入块的位置。

虽然这是一个很简单的架构，但它能够非常有效地处理来自多个用户的同时下载请求。



图 4-4 给出了从任务文件服务器上下载一个文件到用户本地机器上的事件序列。

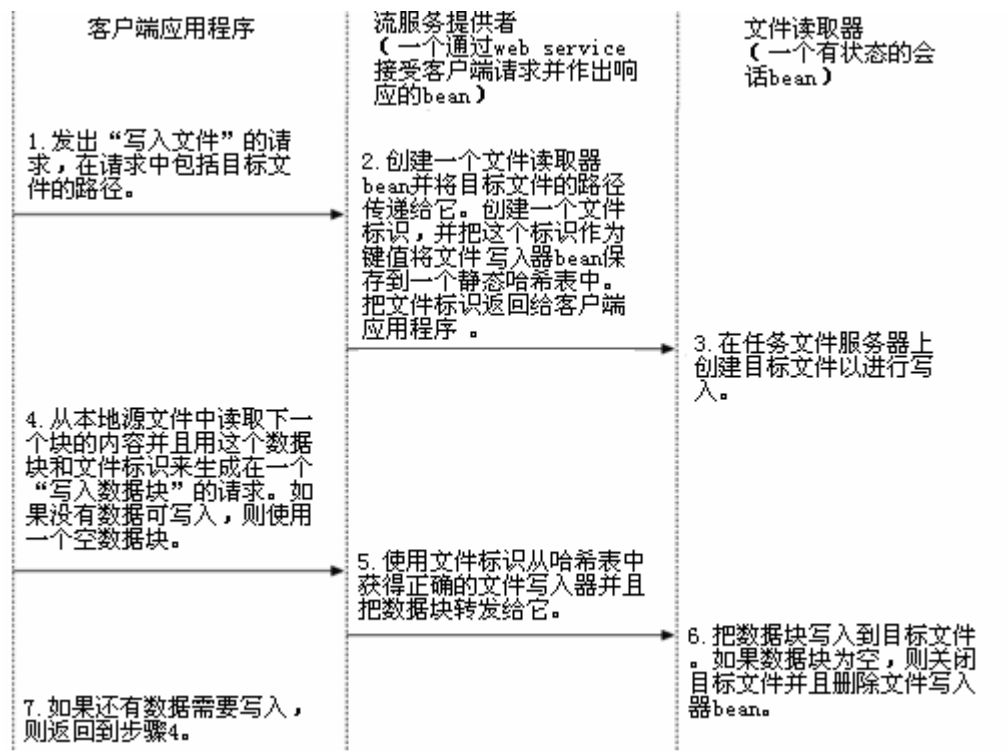
图 4-4 两层服务处理文件读取的过程



注意，流服务提供者在两个服务请求之间并不会维护任何状态。它的作用是作为一个快速的服务分发器，把工作包装起来并且发送给有状态的文件读取器 bean。由于它并不需要记录请求或者维护状态，因此它可以处理来自多个客户应用程序的混合请求。每个文件读取器 bean 都为单个客户应用程序维护了状态信息（到什么位置去读取下一块数据），应用程序发出多个“读取数据块”的请求来下载一个完整的文件。这个架构使得流服务能够同时把多个文件下载到多个客户端，而同时还能能为所有这些请求提供满足需要的吞吐量。

将文件从用户的本地机器上传到任务文件服务器的事件序列也是很清楚的。如图 4-5 所示。

图 4-5 两层服务处理文件写入的过程



除了文件标识之外，在每个客户请求中还包含了一个用户标识，但在上面的表格中并没有给出。当客户端应用程序向中间件的用户管理服务发送了一个成功登录（包括用户名和密码）请求时，它将首先获得一个用户标识，这样就认证了用户。在用户标识中包含了识别特定用户会话的信息，包括用户的角色。它使得流服务能够验证某个请求是否来自于一个合法的用户。此外还将通过检测用户的角色来判断她是否有下载某个文件的权限。例如，MER 任务不允许国外（非美国）用户下载某些特定的文件，在 CIP 中将遵守所有这些安全限制。

## 4.5 可靠性

可靠的代码能够持续良好地执行而不会遇到问题。程序很少崩溃。你可以想像到，在火星漫步者上的代码必须是极其可靠的，因为要进行现场的维护将是非常困难的事情。不过，MER 任务也希望任务控制中心所使用的基地软件同样是可靠的。一旦任务开始进行，没有人希望因软件问题而中断整个操作。

正如前面所提到的，CIP 项目采用了几种方法来确保系统内在的可靠性：

- 遵循工业标准和最佳实践，包括 J2EE。
- 在任何可能的地方都使用那些已被验证过的 COTS 软件，包括来自于权威中间件制造商的商业应用程序服务器。
- 使用带有模块化服务的面向服务架构
- 实现简单、直接的中间件服务

我们通过额外的技术来进一步提高可靠性：服务日志和监测。这些功能不但在调试小型程序时是很有用的，而且在跟踪大型应用程序的运行行为上也能够起到重要的作用。

## 4.5.1 日志

在开发过程中，我们使用了开源的 Apache Log4J Java 包来记录在中间件服务中发生的每个事件。这对于开发中的调试工作来说是非常有用的。通过日志，我们可以编写出更为可靠的代码。当出现 bug 时，日志可以告诉我们在出现 bug 之前进行的是什么操作，这对于我们修正 bug 非常有帮助。

我们最初试图减少日志的数量，而只是记录在 CIP 开始运行之前发生的严重错误消息。但我们最终保留下了大部分的日志信息，由于它们对整体性能的影响几乎可以忽略不计。随后，我们发现这些日志能够为我们提供大量有用的信息，不仅包括在每个服务上所运行的操作的信息，而且还包括客户程序如何使用服务的信息。通过分析这些日志（我们称之为“日志发掘（log mining）”），我们能够根据经验数据来调节服务以获得更好的性能（请参见本章后面的“动态重配置”一节）。

下面是一段来自于流服务提供者 bean 的代码，这段代码给出了我们如何记录文件下载的过程。方法 `getDataFile()` 用来处理每个来自于客户应用程序的“获取数据文件”的请求（通过 web service）。这个方法将及时记录下这个请求（第 15-17 行），包括请求者的用户 ID 和所请求的源文件的路径。

```

1  public class StreamerServiceBean implements SessionBean
2  {
3      static {
4          Globals.loadResources("Streamer ");
5      };
6
7      private static Hashtable readerTable = new Hashtable( );
8      private static Hashtable writerTable = new Hashtable( );
9
10     private static BeanCacheStats cacheStats = Globals.queryStats;
11
12     public FileToken getDataFile(AccessToken accessToken, String filePath)
13         throws MiddlewareException
14     {
15         Globals.StreamerLogger.info(accessToken.userId( ) +
16                                     ": Streamer.getDataFile("
17                                     + filePath + ")");
18         long startTime = System.currentTimeMillis( );
19
20         UserSessionObject.validateToken(accessToken);
21         FileToken fileToken = doFileDownload(accessToken, filePath);
22         cacheStats.incrementTotalServerResponseTime(startTime);
23         return fileToken;
24     }
25

```

在方法 `doFileDownload()` 中将创建一个新的文件标识（第 30 行）和一个文件读取器（第 41 行），然后调用读取器 bean 的 `getDataFile()` 方法（第 42 行）。`cacheStats` 这个域用于运行

时的监测，将在随后进行介绍。

```
26     private static FileToken doFileDownload(AccessToken accessToken,
27                                             String filePath)
28         throws MiddlewareException
29     {
30         FileToken fileToken = new FileToken(accessToken, filePath);
31         String key = fileToken.getKey( );
32
33         FileReaderLocal reader = null;
34         synchronized (readerTable) {
35             reader = (FileReaderLocal) readerTable.get(key);
36         }
37
38         /创建一个文件读取器，开始下载。
39         if (reader == null) {
40             try {
41                 reader = registerNewReader(key);
42                 reader.getDataFile(filePath);
43
44                 return fileToken;
45             }
46             catch(Exception ex) {
47                 Globals.StreamerLogger.warn("Streamer.doFileDownload("
48                                           + filePath + "): " +
49                                           ex.getMessage( ));
50                 cacheStats.incrementFileErrorCount( );
51                 removeReader(key, reader);
52                 throw new MiddlewareException(ex);
53             }
54         }
55         else {
56             throw new MiddlewareException("File already being downloaded: " +
57                                           filePath);
58         }
59     }
60
```

readDataBlock()方法处理每个来自于客户程序的“读取数据块”请求。它将查找正确的文件读取器 bean（第 71 行）并且调用读取器 bean 的 readDataBlock()方法（第 79 行）。在源文件的末尾，将删除文件读取器 bean（第 91 行）。

```
61     public DataBlock readDataBlock(AccessToken accessToken, FileToken fileToken)
62         throws MiddlewareException
63     {
64         long startTime = System.currentTimeMillis( );
```

```
65      UserSessionObject.validateToken(accessToken);
66
67      String key = fileToken.getKey( );
68
69      FileReaderLocal reader = null;
70      synchronized (readerTable) {
71          reader = (FileReaderLocal) readerTable.get(key);
72      }
73
74      //使用读取器 bean 来下载下一个数据块
75      if (reader != null) {
76          DataBlock block = null;
77
78          try {
79              block = reader.readDataBlock( );
80          }
81          catch(MiddlewareException ex) {
82              Globals. StreamerLogger.error("Streamer.readDataBlock("
83                                          + key + ")", ex);
84              cacheStats.incrementFileErrorCount( );
85              removeReader(key, reader);
86              throw ex;
87          }
88
89          //是否到了文件末尾?
90          if (block == null) {
91              removeReader(key, reader);
92          }
93
94          cacheStats.incrementTotalServerResponseTime(startTime);
95          return block;
96      }
97      else {
98          throw new MiddlewareException(
99              "Download source file not opened: " +
100              fileToken.getFilePath( ));
101      }
102  }
103
```

registerNewReader()和 removeReader()这两个方法分别用来创建和销毁有状态的文件读取器 bean，如下所示：

```
104     private static FileReaderLocal registerNewReader(String key)
105         throws Exception
106     {
```

```
107         Context context = MiddlewareUtility.getInitialContext( );
108         Object queryRef = context.lookup("FileReaderLocal");
109
110         //创建读取器服务 bean 并且注册。
111         FileReaderLocalHome home = (FileReaderLocalHome)
112             PortableRemoteObject.narrow(queryRef, FileReaderLocalHome.class);
113         FileReaderLocal reader = home.create( );
114
115         synchronized (readerTable) {
116             readerTable.put(key, reader);
117         }
118
119         return reader;
120     }
121
122     private static void removeReader(String key, FileReaderLocal reader)
123     {
124         synchronized (readerTable) {
125             readerTable.remove(key);
126         }
127
128         if (reader != null) {
129             try {
130                 reader.remove( );
131             }
132             catch(javax.ejb.NoSuchObjectLocalException ex) {
133                 //省略以下代码
134             }
135             catch(Exception ex) {
136                 Globals.StreamerLogger.error("Streamer.removeReader("
137                     + key + ")", ex);
138                 cacheStats.incrementFileErrorCount( );
139             }
140         }
141     }
142 }
```

以下是来自于文件读取器 bean 的代码。cacheStats 和 fileStats 这两个域用于运行时的监测，将在随后进行介绍。getDataFile()方法记录文件下载的开始事件（第 160-161 行）

```
143 public class FileReaderBean implements SessionBean
144 {
145     private static final String FILE = "file";
146
147     private transient static BeanCacheStats cacheStats = Globals.queryStats;
148     private transient static FileStats      fileStats  = Globals.fileStats;
```

```
149
150     private transient int          totalSize;
151     private transient String        type;
152     private transient String        name;
153     private transient FileInputStream fileInputStream;
154     private transient BufferedInputStream inputStream;
155     private transient boolean        sawEnd;
156
157     public void getDataFile(String filePath)
158         throws MiddlewareException
159     {
160         Globals.StreamerLogger.debug("Begin download of file '"
161                                     + filePath + "'");
162         this.type    = FILE;
163         this.name    = filePath;
164         this.sawEnd = false;
165
166         try {
167
168             //从数据文件创建输入流
169             fileInputStream = new FileInputStream(new File(filePath));
170             inputStream      = new BufferedInputStream(fileInputStream);
171
172             fileStats.startDownload(this, FILE, name);
173         }
174         catch(Exception ex) {
175             close( );
176             throw new MiddlewareException(ex);
177         }
178     }
179
```

readDataBlock()方法将从源文件中读取每个数据块。当它读完了整个源文件时，它将记下任务的完成事件（第 191-193 行）：

```
180     public DataBlock readDataBlock( )
181         throws MiddlewareException
182     {
183         byte buffer[] = new byte[Globals.流 BlockSize];
184
185         try {
186             int size = inputStream.read(buffer);
187
188             if (size == -1) {
189                 close( );
190
```

```
191                Globals.StreamerLogger.debug("Completed download of " +
192                                                type + " '" + name + "': " +
193                                                totalSize + " bytes");
194
195                cacheStats.incrementFileDownloadedCount( );
196                cacheStats.incrementFileByteCount(totalSize);
197                fileStats.endDownload(this, totalSize);
198
199                sawEnd = true;
200                return null;
201            }
202            else {
203                DataBlock block = new DataBlock(size, buffer);
204                totalSize += size;
205                return block;
206            }
207        }
208        catch(Exception ex) {
209            close( );
210            throw new MiddlewareException(ex);
211        }
212    }
213 }
```

以下是流服务日志中的一些内容：

2004-12-21 19:17:43,320 INFO : jqpublic:

Streamer.getDataFile(/surface/tactical/sol/120/jpeg/1P138831013ETH2809P2845L2M1.JPG)

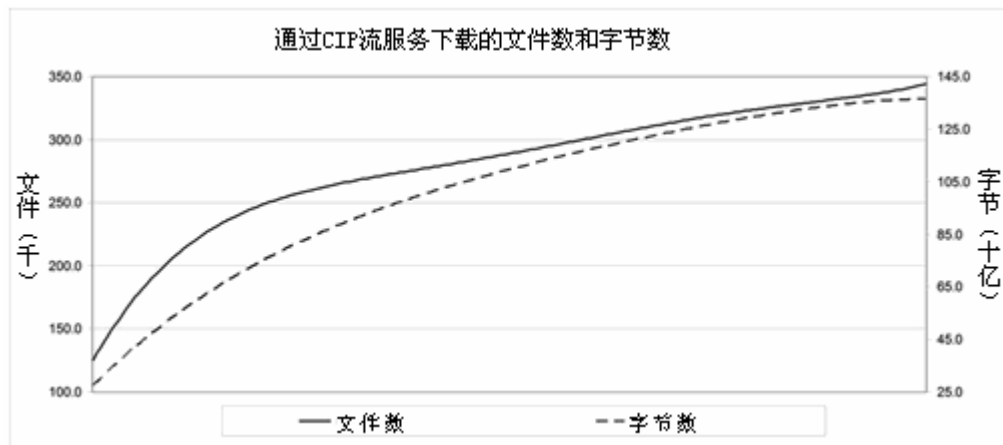
2004-12-21 19:17:43,324 DEBUG: Begin download of file '/surface/tactical/sol/120/  
jpeg/1P138831013ETH2809P2845L2M1.JPG'

2004-12-21 19:17:44,584 DEBUG: Completed download of file '/surface/tactical/sol/120/  
jpeg/1P138831013ETH2809P2845L2M1.JPG': 1876 bytes

在图 4-6 中给出了一张有用的信息图表，其中包括我们从日志发掘中得到的信息。这张图给出了在任务期间几个月的时期内下载数量的趋势（文件的数量以及下载的字节数），从这张图中可以看出当某台漫步者获得有趣的发现时出现的下载峰值。

图 4-6 通过“发掘” CIP 流服务日志而得到的图





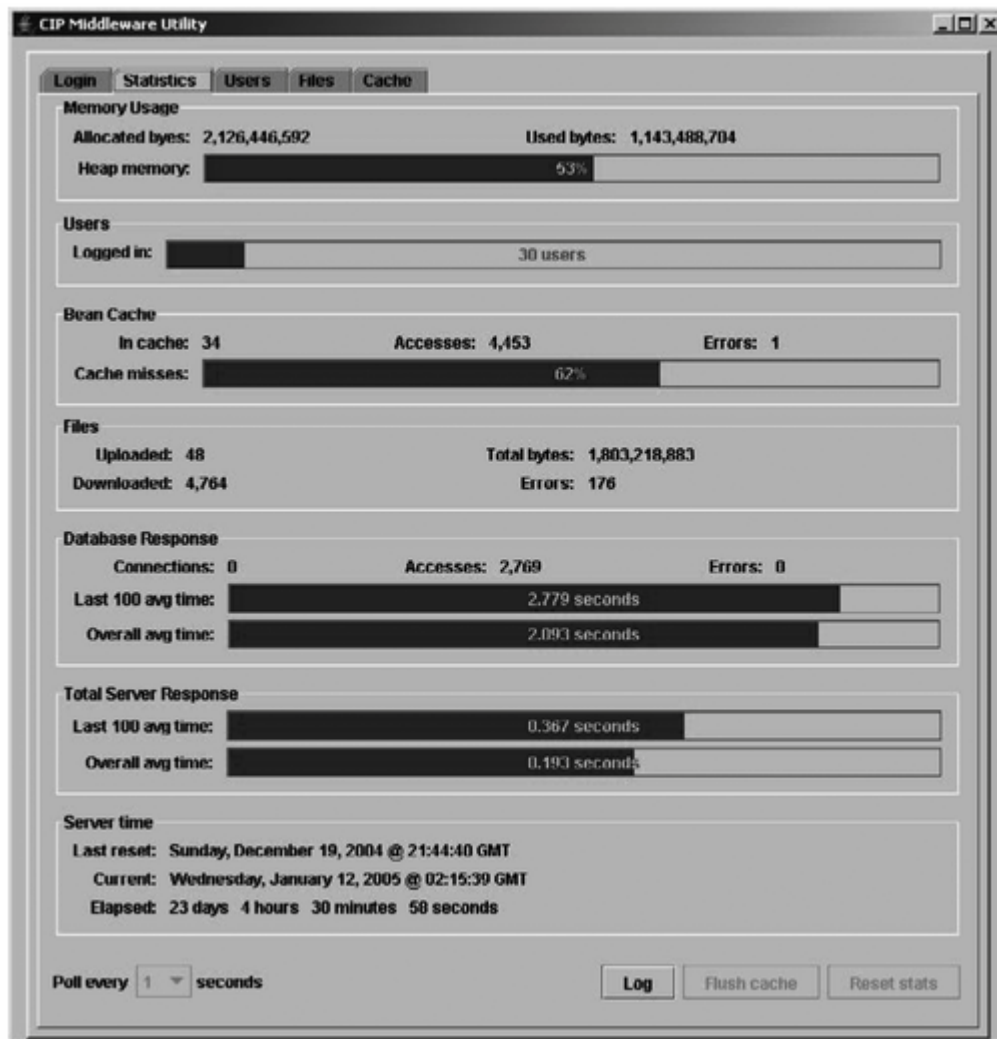
## 监测

日志使我们能够通过研究服务在一段时期内的操作来分析服务的性能。与日志方法不同的是，运行时监测在定位问题及其成因中是最有帮助的，它能有助于我们看到服务当前的运行情况。它使我们能够进行动态调节以改善性能或者防止任何潜在的问题。正如在前面提到的，对于任何大型应用程序来说，监测运行时的行为对于应用程序的成功来说是非常重要的。

在前面给出的代码中包含了对在全局静态对象中保存的性能数据进行更新的语句，这些全局对象是通过 `cacheStats` 和 `fileStats` 这两个域来引用的。中间件监测服务将根据请求来查询这个性能数据。虽然并没有给出这些域所引用的全局对象，但是你应该能够想像出它们所包含的内容。关键是收集有用的运行时性能数据并不是件复杂的任务。

我们把 CIP 中间件监测工具编写为一个客户端应用程序，这个程序将定期地把请求发送给中间件监测服务来获得当前的性能数据。图 4-7 给出了这个工具中 **Statistics** 标签页的截图，其中显示了通过流服务已经被下载的和上传的文件数量以及字节数量、已经发生的文件错误的数量（例如客户端应用程序指定了无效文件名的错误）以及其他的运行时统计数据。

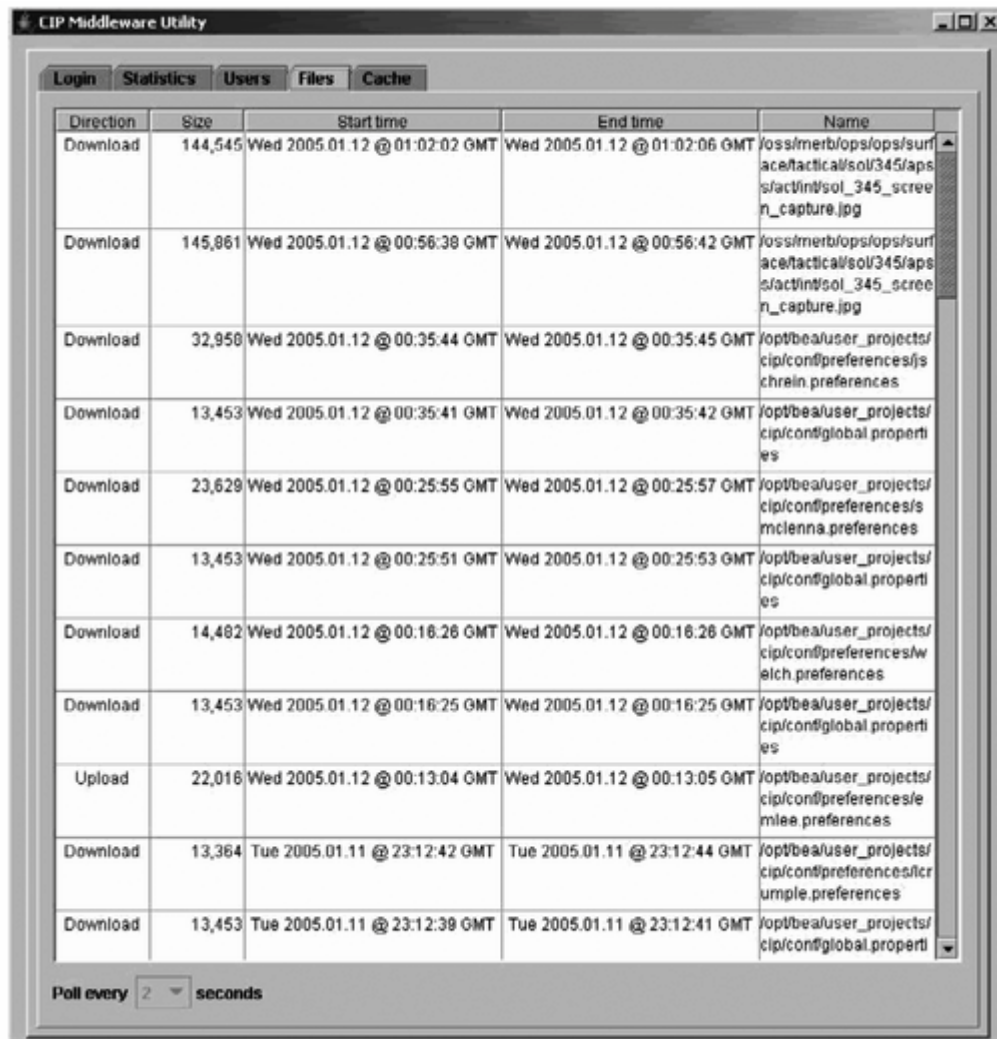
图 4-7 中间件监测工具中 **Statistics** 标签页的截图



在流服务提供者 bean 中的 `doFileDownload()` 和 `readDataBlock()` 这两个方法都会更新全局文件错误的计数(在“日志”一节代码中的第 50 行和第 84 行)。`getDataFile()` 和 `readDataBlock()` 这两个方法将增加全局的总服务响应时间(第 22 行和第 94 行)。正如在图 4-7 中所看到的,中间件监测工具在“Total Server Response”项中显示平均响应时间。

文件读取器 bean 的 `getDataFile()` 方法将记录每个文件下载的起始时刻(第 172 行)。`readDataBlock()` 方法将增加全局的总文件数和字节数量(第 195 行和第 196 行)并且记录下载完成的时刻(第 197 行)。在图 4-8 中给出监测工具的“File”标签页的截图,显示的是当前和最近的文件下载和上传等行为。

图 4-8 中间件监测工具中 File 标签页的截图



The screenshot shows the 'CIP Middleware Utility' window with the 'Files' tab selected. It displays a table of file transfer operations. The table has columns for Direction, Size, Start time, End time, and Name. The data shows various download and upload operations between different paths, including screenshots and preference files. At the bottom, there is a 'Poll every' dropdown set to '2' seconds.

Direction	Size	Start time	End time	Name
Download	144,545	Wed 2005.01.12 @ 01:02:02 GMT	Wed 2005.01.12 @ 01:02:06 GMT	/oss/merb/ops/ops/surface/tactical/sol/345/aps/sfactint/sol_345_screenshot_capture.jpg
Download	145,861	Wed 2005.01.12 @ 00:56:38 GMT	Wed 2005.01.12 @ 00:56:42 GMT	/oss/merb/ops/ops/surface/tactical/sol/345/aps/sfactint/sol_345_screenshot_capture.jpg
Download	32,950	Wed 2005.01.12 @ 00:35:44 GMT	Wed 2005.01.12 @ 00:35:45 GMT	/opt/bea/user_projects/cip/conf/preferences/jschrain.preferences
Download	13,453	Wed 2005.01.12 @ 00:35:41 GMT	Wed 2005.01.12 @ 00:35:42 GMT	/opt/bea/user_projects/cip/conf/global.properties
Download	23,629	Wed 2005.01.12 @ 00:25:55 GMT	Wed 2005.01.12 @ 00:25:57 GMT	/opt/bea/user_projects/cip/conf/preferences/smcenna.preferences
Download	13,453	Wed 2005.01.12 @ 00:25:51 GMT	Wed 2005.01.12 @ 00:25:53 GMT	/opt/bea/user_projects/cip/conf/global.properties
Download	14,482	Wed 2005.01.12 @ 00:16:28 GMT	Wed 2005.01.12 @ 00:16:28 GMT	/opt/bea/user_projects/cip/conf/preferences/welch.preferences
Download	13,453	Wed 2005.01.12 @ 00:16:25 GMT	Wed 2005.01.12 @ 00:16:25 GMT	/opt/bea/user_projects/cip/conf/global.properties
Upload	22,016	Wed 2005.01.12 @ 00:13:04 GMT	Wed 2005.01.12 @ 00:13:05 GMT	/opt/bea/user_projects/cip/conf/preferences/emilee.preferences
Download	13,364	Tue 2005.01.11 @ 23:12:42 GMT	Tue 2005.01.11 @ 23:12:44 GMT	/opt/bea/user_projects/cip/conf/preferences/sicrumple.preferences
Download	13,453	Tue 2005.01.11 @ 23:12:39 GMT	Tue 2005.01.11 @ 23:12:41 GMT	/opt/bea/user_projects/cip/conf/global.properties

Poll every 2 seconds

## 4.6 稳定性

变动是不可避免的，而漂亮的代码即使在投入运行之后仍然能够优雅地处理变动。我们采取了一组方法来保证 CIP 是稳定的并且能够处理运行参数中的变动。

- 我们避免在中间件服务中使用硬编码的参数。
- 我们尽可能地使得由于对已投入运行的中间件服务进行修改而造成的客户端应用程序中断降至最低。

### 4.6.1 动态重配置

大多数中间件服务都有关键的运行参数。例如，正如我们在前面所看到的，流服务按照块的方式下载文件，因此它就有有一个有关块大小的参数。我们不是把块大小硬编码到程序中，而是把这个值放在一个参数文件中，服务在每次启动的时候都会去读取这个参数文件。这个操作发生在流服务提供者 bean 被加载的时候（“日志”一节代码的第 3-5 行）。

在所有中间件服务共享和加载的 `middleware.properties` 文件中，包含了下面这行：

```
middleware.Streamer.blocksize = 65536
```

然后文件读取器 `bean` 的 `readDataBlock()` 方法可以使用这个值(“日志”一书代码的第 183 行)。

每个中间件服务都可以在启动的时候加载数个参数值。熟练的编码人员需要掌握的技巧之一就是要知道一个服务的哪些关键值被开放为可装载参数。他们在开发期间肯定是非常有帮助的；例如，我们能够在开发过程中尝试不同的块大小而无需每次都重新编译流服务。

可装载的参数对于把代码投入运行或许还要更加重要。在大多数的产品环境中，对已经投入运行的软件进行修改是非常困难和昂贵的。**MER** 任务也存在这样的问题，因此成立了一个正式的变更控制部门 (**change control board**) 来审查那些需要在任务运行时对代码进行的修改。

当然，避免硬编码的参数值是基本的编程 101 格言之一，它既适用于小型程序，也适用于大型程序。但这对于大型程序来说尤其重要，在这些程序中可能有着非常多的参数值，分散在代码的每个角落。

## 4.6.2 热交换

热交换 (**Hot Swapping**) 是我们在 **CIP** 中间件中采用的商业应用服务器的重要功能之一。这使得我们可以用一个中间件服务来替换另一个正在运行的中间件而无需先停止它 (以及 **CIP**)。

当我们需要在修改参数值之后强制服务重新进行加载时，可以使用热交换，我们可以通过重新加载一次服务来实现。当然，像流服务这样使用有状态会话 `bean` (文件读取器和文件写入器 `bean`) 的服务将会丢失所有的状态信息。因此，只有当服务处于“安静”时期，我们才可以对服务进行热交换，因为此时我们知道服务目前并没有被使用。对于流服务来说，我们可以通过中间件监测工具的 **Files** 标签页 (参见图 4-8) 来知道何时处于这种情况。

热交换在大型企业应用程序的运行环境中是最有意义的，在这些环境中，在替换部分程序的同时保持其他部分仍然运行是一项很重要的需求。而对于小型程序来说，最好的方式就是把程序送回去修改。

## 4.7 结束语

协作式信息管理系统证明了——是的，即使在像 **NASA** 这样庞大的政府部门中——成功地按时交付一个严格满足实用性、可靠性和稳定性等需求的大型复杂企业软件系统是可能的。火星漫步者已经远远超越了预期要求，它很好地说明了如何成功地设计和构建同时位于火星上和地球上的硬件和软件，以及编码人员使用的技巧。

与小型程序不同，大型应用程序的漂亮性并不一定只存在于优美的算法中。对于 **CIP** 来说，漂亮性是在于它的面向服务架构实现以及大量虽然简单却经过仔细挑选的组件——编码人员只需知道将这些组件组合在一起即可。

## 第 5 章 美丽的并发

Simon Peyton Jones

免费午餐已经结束<sup>[1]</sup>。以往我们习惯于通过购买新一代CPU来加快程序的运行速度，但现在，这样的好时光已经一去不复返了。虽说下一代芯片会带有多个CPU，但每个单独的CPU的速度却不会再变快了。所以，如果想让程序跑得更快的话，你就必须得学会编写并行程序<sup>[2]</sup>。

<sup>[1]</sup>Herb Sutter, "The free lunch is over: a fundamental turn toward concurrency in software," Dr. Dobbs' Journal, March 2005.

<sup>[2]</sup>Herb Sutter and James Larus, "Software and the concurrency revolution," ACM Queue, Vol. 3, No. 7, September 2005.

并行程序的执行是非确定性的，因而测试起来自然也就不容易；并发程序中有的bug甚至可能会无法重现。我对漂亮程序的定义是“简单而优雅，乃至显然没有任何错误”，而不仅仅是“几乎没有任何明显的错误”<sup>[3]</sup>。要想编写出能够可靠运行的并行程序，程序的美感是尤其要注意的方面。可惜一般来说并行程序总归没有它们相应的非并行（顺序式的）版本漂亮；尤其是在模块性方面：并行程序的模块性相对较弱。这一点我们后面会看到。

<sup>[3]</sup>This turn of phrase is due to Tony Hoare.

本章将介绍软件事务内存（STM）。软件事务内存是一项很有前景的技术，它提出了一种针对共享内存并行处理器编程的新手段，正如刚才所言，传统并行程序的模块性较弱，而这正是软件事务内存的强项。我希望你读完本章后能和我一样对这项新技术感到振奋。当然，软件事务内存也并非万灵药，但它的确对并发领域中令人望而却步的问题发起了一次漂亮且令人鼓舞的冲击。

### 5.1 一个简单的例子：银行账户

假设有这样一个简单的编程任务：

编写一段程序，将钱从一个银行账户转移到另一个账户。为了简化起见，两个账户都是存放在内存里面的——也就是说你不用考虑跟数据库的交互。要求是你的代码在并发环境中也必须能够正确执行，这里所谓的并发环境也就是指可能存在多个线程同时调用你的转账函数，而所谓能够正确执行则是指任何线程都不能“看到”系统处于不一致的状态（比如看到其中一个账户显示已被取出了一笔钱然而同时另一个账户却显示还没有收到这笔钱）。

这个例子虽说有点人为捏造的味道，但它的优点是简单，因而我们便能将注意力集中在它的解决方案上，后面你会看到，Haskell 结合事务内存将会给这个问题带来新的解决方案。不过此前还是让我们先来简单回顾一下传统的方案。

### 5.1.1 加锁的银行账户

目前，用于协调并发程序的主导技术仍是锁和条件变量。在一门面向对象的语言中，每个对象上都带有一个隐式的锁，而加锁则由 `synchronized` 方法来完成，但原理与经典的加锁解锁是一样的。在这样一门语言中，我们可以将银行账户类定义成下面这样：

```
class Account {
    Int balance;
    synchronized void withdraw( Int n ) {
        balance = balance - n; }
    void deposit( Int n ) {
        withdraw( -n ); }
}
```

`withdraw` 方法必须是 `synchronized` 的，这样两个线程同时调用它的时候才不会把 `balance` 减错了。`synchronized` 关键字的效果就等同于先对当前账户对象加锁，然后运行 `withdraw` 方法，最后再对当前账户对象解锁。

有了这个类之后，我们再来编写 `transfer` 转账函数：

```
void transfer( Account from, Account to, Int amount ) {
    from.withdraw( amount );
    to.deposit( amount ); }
```

对于非并发程序来说以上代码完全没问题，然而在并发环境中就不一样了，另一个线程可能会看到转账操作的“中间状态”：即钱从一个账户内被取走了，而同时另一个账户却还没收到这笔钱。值得注意的是，虽然 `withdraw` 和 `deposit` 这两个函数都是 `synchronized`，但这种情况还是会出现。在 `from` 上调用 `withdraw` 会将 `from` 加锁，执行提款操作，然后对其解锁。类似的，在 `to` 上调用 `deposit` 会将 `to` 加锁，执行存款操作，然后对其解锁。然而，关键是在这两个调用之间有一个间隙，在这个状态下待转账的钱既不在 `from` 账户中也不在 `to` 账户中。

在一个金融程序中，这种情况可能是无法容忍的。那我们该如何解决这个问题呢？通常的方案可能是在外面再包一层显式的加锁解锁操作，如下：

```
void transfer( Account from, Account to, Int amount ) {
    from.lock(); to.lock();
    from.withdraw( amount );
    to.deposit( amount );
    from.unlock(); to.unlock() }
```



但这种做法有一个致命的缺陷：它可能会导致死锁。我们考虑这样一种情况：在一个线程将一笔钱从 A 账户转到 B 账户的同时，另一个线程也正在将一笔钱从 B 账户转到 A 账户（当然，发生这种事情的几率很小）。这时便可能会出现两个线程各锁一个账户并都在等着对方释放另一账户的情况。

问题找出来了（不过，并发环境下的问题可不总是像这么容易就能找出来的），标准的解决方案是规定一个全局统一的锁顺序，并按照递增顺序来进行加锁。采用这种做法的代码如下：

```
if from < to
  then { from.lock(); to.lock(); }
  else { to.lock(); from.lock(); }
```

这个方法是可行的，但前提是必须事先知道要对哪些锁进行加锁，而后面这个条件并不是总能满足的。例如，假设 `from.withdraw` 的实现当账户余额不足时就会从 `from2` 上提款。遇到这种情况除非等到我们从 `from` 中提了钱否则是无法知道是否该对 `from2` 加锁的，而另一方面，一旦已经从 `from` 中提了钱，再想按“正确”顺序加锁便不可能了。更何况 `from2` 这个账户可能根本就只应对 `from` 可见，而不应被 `transfer` 知道。而且退一步说，就算 `transfer` 知道 `from2` 的存在，现在需要加的锁也已经由两个变成了三个（事先还要记得将这些锁正确排序）。

还有更糟的，如果我们需要在某些情况下阻塞（block），情况就会更加复杂。例如，要求 `transfer` 在 `from` 账户内余额不足的时候阻塞。这类问题通常的解决办法是在一个条件变量上等待，并同时释放 `from` 的锁。但更进一步，如果要求当 `from` 和 `from2` 中的总余额不够的时候阻塞呢？

### 5.1.2 “生锈”的锁

简而言之，在如今的并发编程领域占主导地位的技术，锁和条件变量，从根本上是有缺陷的。以下便是基于锁的并发编程中的一些公认的困难（其中有些我们在前文的例子中已经看到过了）。

锁加少了

容易忘记加锁，从而导致两个线程同时修改同一个变量。

锁加多了

容易加锁加得太多了，结果（轻则）妨碍并发，（重则）导致死锁。

锁加错了

在基于锁的并发编程中，锁和锁所保护的数据之间的联系往往只存在于程序员的大脑里，而不是显式地表达在程序代码中。结果就是一不小心便会加错了锁。

## 加锁的顺序错了

在基于锁的并发编程中，我们必须小心翼翼地按照正确的顺序来加锁（解锁），以避免随时可能会出现死锁；这种做法既累人又容易出错，而且，有时候极其困难。

## 错误恢复

错误恢复也是个很麻烦的问题，因为程序员必须确保任何错误都不能将系统置于一个不一致的、或锁的持有情况不确定的状态下。

## 忘记唤醒和错误的重试

容易忘记叫醒在条件变量上等待的线程；叫醒之后又容易忘记重设条件变量。

然而，基于锁的编程，其最根本的缺陷，还是在于锁和条件变量不支持模块化编程。这里“模块化编程”是指通过粘合多个小程序来构造大程序的过程。而基于锁的并发程序是做不到这一点的。还是拿前面的例子来说吧：虽然withdraw和deposit这两个方法都是并发正确的，但如果原封不动的话，你能直接用它们实现出一个transfer来吗？不能，除非让锁协议暴露出来。而且遇到选择和阻塞的话还会更头疼。例如，假设withdraw在账户余额不足的情况下会阻塞。你就会发现，除非暴露锁条件，否则你根本无法直接利用withdraw函数从“A账户或B账户（取决于哪个账户有足够余额）”进行提款；而且就算知道了锁条件，事情仍还是麻烦。另一些文献中也对锁并发的这种困难作了论述。<sup>[4]</sup>

<sup>[4]</sup> Edward A. Lee, “The problem with threads,” IEEE Computer, Vol. 39, No. 5, pp. 33–42, May 2006; J. K. Ousterhout, “Why threads are a bad idea (for most purposes),” Invited Talk, USENIX Technical Conference, January 1996; Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy, “Composable memory transactions,” ACM Symposium on Principles and Practice of Parallel Programming (PPoPP ’05), June 2005.

## 5.2 软件事务内存

软件事务内存（STM）是迎接并发挑战的一种很有前途的新技术，本节将会详细说明这一点。我选用Haskell来介绍STM，Haskell是我见过的最美丽的编程语言，而且STM能够特别优雅地融入到Haskell中。如果你还不了解Haskell，别担心，边看边学。

### 5.2.1 Haskell中的副作用（Side Effects）和输入/输出（I/O）

Haskell中的transfer函数写出来就像这样：

```
transfer :: Account -> Account -> Int -> IO ( )
```



```
-- Transfer 'amount' from account 'from' to account 'to'
transfer from to amount = ...
```

以上代码的第二行，即以“--”开头的那行，是一个注释。代码的第一行是对transfer的函数类型的声明（类型声明以“::”前导）<sup>[5]</sup>，“Account -> Account -> Int -> IO ()”读作“接受一个Account，加上另一个Account（两个Account，代表转账的源账户和目标账户），以及一个Int（转账的数额），返回一个IO()类型的值”。最后一个类型（即返回类型）“IO ()”说的是“transfer函数返回的是一个动作（action），这个动作被执行的时候可能会产生副作用（side effects），并返回一个‘()’类型的值”。“()”类型读作“单元（unit）”，该类型只有一个可能的值，也写作“()”，有点类似于C里面的void。transfer将IO()作为返回类型说明了执行过程中的副作用是我们调用transfer的惟一原因。那么，在介绍下面的内容之前，我们就必须首先知道Haskell是怎么对待副作用的。

<sup>[5]</sup>你可能会觉得奇怪，为什么在这个类型签名里面有三个“->”（难道不应该是一个吗——位于参数类型与返回类型之间？）其实这是因为Haskell支持所谓的currying，后者在任何介绍Haskell的书（比如Haskell: The Craft of Functional Programming, by S. J. Thompson [Addison-Wesley]）中或wikipedia上都能见到它的踪影。但currying不是本章要讲的重点，你大可以忽略除了最后一个“->”之外的所有“->”，即除了最后一个类型之外，其他都是函数的参数类型。

那么副作用是什么呢？副作用就是我们读写可变（mutable）状态所造成的影响（effect）。输入/输出是副作用的绝佳范例。例如，下面是两个Haskell函数，它们都具有输入/输出副作用：

```
hPutStr  :: Handle -> String -> IO ()
hGetLine :: Handle -> IO String
```

任何类型形如“IO t”（其中t可以为()，也可以为其他类型，如String）的值都是一个动作（action）。也就是说，在上面的例子中，(hPutStr h “hello”)是一个动作<sup>[1]</sup>，执行这个动作的效果便是在句柄h上输出“hello”<sup>[2]</sup>。类似的，(hGetLine h)也是一个动作，当它被执行的时候就会从句柄h代表的输入设备中读入一行输入并将其作为一个String返回。此外，利用Haskell的do关键字，我们可以将几个小的带副作用的程序“粘合”成一个大的。例如，下面的hEchoLine函数读入一个串并将它打印出来：

<sup>[1]</sup>在Haskell里面调用一个函数很简单，只须将函数名和它的各个参数并排写在一块儿就行了。在大多数语言中你都需要写成hPutStr(h, “hello”), 但Haskell里面只要写成(hPutStr h “hello”)就行了。

<sup>[2]</sup>Haskell中的句柄相当于C里面的文件描述（file descriptor）：指明对哪个文件或管道进行读写。跟Unix里面一样，Haskell里面也预定义了三个句柄：stdin、stdout和stderr。

```
hEchoLine :: Handle -> IO String
hEchoLine h = do { s <- hGetLine h
```

```
    ; hPutStr h ("I just read: " ++ s ++ "\n")
    ; return s }
```

`do { $a_1$ ; ...;  $a_n$ }` 结构将数个较小的动作 ( $a_1 \cdots a_n$ ) 粘合成一个较大的动作。因此上面的代码中, `hEchoLine h` 这个动作被执行的时候便会首先调用 `hGetLine h` 来读取一行输入, 并将这行输入命名为 `s`。接着调用 `hPutStr`, 将 `s` 加上前导的 “I just read: ”<sup>[3]</sup> 一并打印出来。最后串 `s` 被返回。最后一行 `return s` 比较有趣, 因为 `return` 并非像在其他命令式语言中那样是语言内建的操作, 而只是一个普普通通的函数, 其函数类型如下:

<sup>[3]</sup> ++ 操作符的作用是将两个串拼接起来。

```
return :: a -> IO a
```

也就是说, 像 `return v` 这样一个操作被执行的时候, 将会返回 `v`, 同时并不会导致任何副作用<sup>[9]</sup>。 `return` 函数可以作用于任何类型的值, 这一点体现在其函数类型中: `a` 是一个类型变量, 代表任何类型。

<sup>[9]</sup> “IO” 的意思是一个函数可能有副作用, 但并不代表它就一定会带来副作用。

输入/输出是一类重要的副作用。还有一类重要的副作用便是对可变 (mutable) 变量的读写。例如, 下面这个函数将一个可变变量的值增 1:

```
incRef :: IORef Int -> IO ()
incRef var = do { val <- readIORef var
                 ; writeIORef var (val+1) }
```

`incRef var` 是一个动作。它首先执行 `readIORef var` 来获得变量 `var` 的值, 并将该值绑定到 `val`; 接着它调用 `writeIORef` 将 `val+1` 写回到 `var` 里面。 `readIORef` 和 `writeIORef` 的类型如下:

```
readIORef  :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

类型形如 `IORef t` 的值相当于一个指针或引用, 指向或引用一个 `t` 类型的可变值 (类似于 C 里面的 `t*`)。具体到 `incRef`, 其参数类型为 `IORef Int`, 因为 `incRef` 只操作 `Int` 型变量。

现在, 我们已经知道了如何将数个较小的动作组合成一个较大的——但一个动作到底如何才算被真正调用呢? 在 Haskell 里, 整个程序其实就是一个 IO 动作, 名叫 `main`。要运行这个程序, 我们只需执行 `main`。例如下面就是一个完整的程序:

```
main :: IO ()
```

```
main = do { hPutStr stdout "Hello"
           ; hPutStr stdout " world\n" }
```

该程序是一个顺序式 (sequential) 程序，因为 do 块将两个 IO 动作按顺序连接了起来。另一方面，要构造并发程序的话，我们便需要另一个原语 (primitive)：forkIO：

```
forkIO :: IO a -> IO ThreadId
```

forkIO是Haskell内建的函数，它的参数是一个IO动作，forkIO所做的事情就是创建一个并发的Haskell线程来执行这个IO动作。一旦这个新线程建立，Haskell的运行系统便会将它与其他Haskell线程并行执行。例如假设我们将前面的main函数修改为<sup>[1]</sup>：

<sup>[10]</sup>其实，main的第一行我们本可以写成“tid <- forkIO(hPutStr …)”的，这行语句会把forkIO的返回值（一个ThreadId）绑定到tid。然而由于本例中我们并不使用返回的ThreadId，所以就把“tid <-”省略了。

```
main :: IO ( )
main = do { forkIO (hPutStr stdout "Hello")
           ; hPutStr stdout " world\n" }
```

现在，这两个 hPutStr 操作便能够并发执行了。至于哪个先执行（从而先打印出它的字符串）则是不一定的。Haskell 里面由 forkIO 产生出来的线程是非常轻量级的：只占用几百个字节的内存，所以一个程序里面就算产生上千个线程也是完全正常的。

读到这里，你可能会觉得 Haskell 实在是门又笨拙又麻烦的语言，incRef 的三行代码说穿了就做了 C 里面的一个 x++而已！没错，在 Haskell 里面，实施副作用的方式是非常显式且冗长的。然而别忘了，首先 Haskell 主要是一门函数式编程语言。大多数代码都是在 Haskell 的函数式内核里写的，后者的特点是丰富、高表达力、简洁。因而 Haskell 编程的精神就是“有节制地使用副作用”。

其次我们注意到，在代码中显示声明副作用的好处是代码能够携带许多有用的信息。考虑下面两个函数：

```
f :: Int -> Int
g :: Int -> IO Int
```

通过它们的类型我们便可以一眼看出，f 是一个纯函数，无副作用。给它一个特定的数（比如 42），那么每次对它的调用（f 42）都会返回同样的结果。相比之下，g 就具有副作用了——这一点明明白白地显示在它的类型中。对 g 的不同的调用，就算参数相同，也可能会得到不同的值，因为，比如说，它可以通过 stdin 读取数据，或对一个可变变量进行修改。后面你会发现，这种显式声明副作用的做法其实非常有用。

最后，前面提到的所谓动作（action，比如 I/O 动作）其实本身也是值（Haskell 中的动作也是一等公民）：它们也可以被作为参数传递或作为返回值返回。比如下面就是一个纯粹用 Haskell 写的（而非内建的）模拟（简化的）for 循环的函数：

```
nTimes :: Int -> IO () -> IO ()
nTimes 0 do_this = return ()
nTimes n do_this = do { do_this; nTimes (n-1) do_this }
```

这是一个递归函数，其第一个参数是一个 Int，表示要循环多少次，第二个参数则是一个动作（action）：do\_this；该函数返回一个动作，后者被执行的时候会把 do\_this 重复做 n 遍。比如下面这段代码利用 nTimes 来重复输出 10 个“Hello”：

```
main = nTimes 10 (hPutStr stdout "Hello\n")
```

这其实从效果上就等同于允许用户自定义程序的控制结构了。

话说回来，本章的目的并不是要对 Haskell 作一个全面的介绍，而且即便是对于 Haskell 里面的副作用我们也只是稍加阐述。如果你想进一步了解的话，可以参考我写的一篇指南“Tackling the awkward squad”<sup>[11]</sup>。

<sup>[11]</sup> Simon Peyton Jones, “Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell,” C. A. R. Hoare, M. Broy, and R. Steinbrueggen, editors, Engineering theories of software construction, Marktoberdorf Summer School 2000, NATO ASI Series, pp. 47 - 96, IOS Press, 2001.

## 5.2.2 Haskell 中的事务

OK，终于可以回到我们的 transfer 函数了。其代码如下：

```
transfer :: Account -> Account -> Int -> IO ()
-- Transfer 'amount' from account 'from' to account 'to'
transfer from to amount
  = atomically (do { deposit to amount
                    ; withdraw from amount })
```

里面的那个 do 块你应该不觉得陌生了吧：它先是调用 deposit 将 amount 数目的钱存入 to 账户，然后再从 from 账户中提取 amount 数目的钱。至于 deposit 和 withdraw 这两个辅助函数我们待会再来写，现在我们先来看看对 atomically 的调用。atomically 的参数是一个动作，它会将该动作作为一个原子来执行。更精确地说，atomically 有如下两个特性作保证：

原子性

`atomically act` 调用所产生的副作用对于其他线程来说是“原子的”。这就保证了另一个线程不可能观察到钱被从一个账户中取出而同时又来不及存入另一个账户中去的中间状态。

## 隔离性

在`atomically act`执行的过程中，`act`这个动作与其他线程完全隔绝，不受影响。这就好像在`act`开始执行的时候世界停顿了，直到`act`执行完毕之后世界才又开始恢复运行。

至于`atomically`函数的执行模型，简单的做法是：存在一个惟一的全局锁；`atomically act`首先获取该锁，然后执行动作`act`，最后释放该锁。这个实现虽然保证了原子性，但粗暴地禁止了任意两个原子块在同一时间执行。

上面说的这个模型有两个问题。第一，它并没有保证隔离性：比如一个线程在执行一个原子块的过程中访问了一个 `IRef`（此时该线程持有全局锁），另一个线程此时照样可以直接对同一个 `IRef` 进行写操作（只要这个写操作不在原子块内）。这就破坏了隔离性保证。第二，它极大的损害了执行性能，因为即便各个原子块之间互不相干，也必须被串行化执行。

第二个问题待会在“事务内存实现”一节会详细讨论。目前先来看第一个问题。第一个问题可以通过类型系统轻易解决。我们将 `atomically` 函数赋予如下类型：

```
atomically :: STM a -> IO a
```

`atomically`的参数是一个类型为`STM a`的动作。`STM`动作类似于`IO`动作，它们都可能具有副作用，但`STM`动作的副作用的容许范围要小得多。`STM`中你可以做的事情主要就是对事务变量（类型为`TVar a`）进行读写，就像我们在`IO`动作里面主要对`IRef`进行读写一样<sup>[12]</sup>。

<sup>[12]</sup>这儿其实有一个命名上的不一致：`STM`变量被命名为`TVar`，然而普通变量却被命名为`IRef`——其实要么是`TVar/IOVar`，要么是`TRef/IRef`才对。但事到如今已经没法再改了。

```
readTVar  :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ( )
```

跟`IO`动作一样，`STM`动作也可以由`do`块组合起来，实际上，`do`块针对`STM`动作进行了重载，`return`也是；这样它们便可以运用于`STM`和`IO`两种动作了<sup>[13]</sup>。例如，下面是`withdraw`的代码：

<sup>[13]</sup>其实Haskell并没有特别针对`IO`和`STM`动作来重载`do`和`return`，`IO`和`STM`其实只是一个更一般的模式的特例，这个更一般的模式便是所谓的`monad`（P. L. Wadler在“The essence of functional programming” 20th ACM Symposium on Principles of Programming Languages [POPL '92], Albuquerque, pp. 1 - 14, ACM, January 1992 中有描述），`do`和`return`的重载便是通过用Haskell的非常泛化的“类型的类型”（`type-class`）系统来表达`monad`而得以实现的（described in P. L. Wadler and S. Blott, “How to make ad-hoc

polymorphism less ad hoc,” Proc 16th ACM Symposium on Principles of Programming Languages, Austin, Texas, ACM, January 1989; and Simon Peyton Jones, Mark Jones, and Erik Meijer, “Type classes: an exploration of the design space,” J. Launch-bury, editor, Haskell workshop, Amsterdam, 1997)。

```
type Account = TVar Int

withdraw :: Account -> Int -> STM ( )
withdraw acc amount
  = do { bal <- readTVar acc
        ; writeTVar acc (bal - amount) }
```

我们用一个包含一个 Int（账户余额）的事务变量来表示一个账户。withdraw 是一个 STM 动作，将账户中的余额提走 amount。

为了完成 transfer 的定义，我们可以通过 withdraw 来定义 deposit：

```
deposit :: Account -> Int -> STM ( )
deposit acc amount = withdraw acc (- amount)
```

注意，transfer 从根本上执行了四个基本的读写操作：对 to 账户的一次读和一次写；以及对 from 账户的一次读和一次写。这四个操作是被当成一个原子来执行的，其执行满足本节（“一个简单的例子：银行账户”）开头的要求。

Haskell 的类型系统优雅地阻止了我们在事务之外读写 TVar。例如假设我们这样写：

```
bad :: Account -> IO ( )
bad acc = do { hPutStr stdout "Withdrawing..."
               ; withdraw acc 10 }
```

以上代码不能通过编译，因为 hPutStr 是一个 IO 动作，而 withdraw 则是一个 STM 动作，这两者不能放在同一个 do 块中。但如果我们把 withdraw 再放在一个 atomically 调用当中就可以了，如下：

```
good :: Account -> IO ( )
good acc = do { hPutStr stdout "Withdrawing..."
               ; atomically (withdraw acc 10) }
```

### 5.2.3 事务内存实现

有了前面提到过的原子性和隔离性保证，我们其实便可以放心使用 STM 了。不过我常常还是觉得一个合理的实现模型会给直觉理解带来很大的帮助，本节就来介绍这么一个实现模型。但要注意的是，这只是所有可能实现中的一种。STM 抽象的一个漂亮之处就在于它提供了一个小巧干净的接口，而实现这个接口可以有多种方式，可简单可复杂。



在可行的实现方案中，有一个方案特别吸引人，那就是在数据库实现里被采用的所谓的“乐观执行（optimistic execution）”。当`atomically act`被执行的时候，Haskell运行时系统会为它分配一个线程本地的事务日志，该日志最初的时候是空的，随着`act`动作被一步步执行（其间并不加任何形式的锁），每次对`writeTVar`的调用都会将目标`TVar`变量的地址和新值写入日志；而并不是直接写入到那个`TVar`变量本身。每次对`readTVar`的调用都会首先寻找日志里面有没有早先被写入的新值，没有的话才会从目标`TVar`本身读取，并且，在读取的时候，一份拷贝会被顺便存入到日志中。同一时间，另一个线程可能也在运行着它自己的原子块，疯狂地读写同样一组`TVar`变量。

在`act`这个动作执行完毕之后，运行时系统首先会对日志进行验证，如果验证成功，就会提交（commit）日志。那么验证是怎么进行的呢？运行时系统会检查日志中缓存的每个`readTVar`的值是否与它们对应的真正的`TVar`相匹配。是的话便验证成功，并将日志中缓存的写操作结果全都提交到相应的`TVar`变量上。

必须注意的是，以上验证-提交的整个过程是完全不可分割的：底层实现会将中断禁止掉，或使用锁或CAS（compare-and-swap）指令等任何可行的方法来确保这个过程对于其他线程来说就像“一瞬间”的事情一样。但由于所有这些底层工作都由实现来完成，所以程序员不用担心也不用考虑它是怎么完成的。

一个自然而然的问题是：如果验证失败呢？如果验证失败，就代表该事务看到的是不一致的内存视图。于是事务被中止（abort），日志被重新初始化，然后整个事务从头再来过。这个过程就叫做重新执行（re-execution）。由于此时`act`动作的所有副作用都还没有真正提交到内存中，因此重新执行它是完全没问题的。然而有一点必须注意：`act`不能包含任何除了对`TVar`变量读写之外的副作用，比如下面这种情况：

```
atomically (do { x <- readTVar xv
                ; y <- readTVar yv
                ; if x>y then launchMissiles
                  else return () })
```

`launchMissiles::IO ()`这个函数的副作用是“头晕、恶心、呕吐”。由于这个原子块执行的时候并没有加锁，所以如果同时有其他线程也在修改变量`xv`和`yv`的话，该线程就可能观察到不一致的内存视图。而一旦这种情况发生，发射导弹（`launchMissiles`）可就闯了大祸了，因为等到导弹发射完了才发现验证失败就已经来不及了。不过幸运的是，Haskell的类型系统会阻止冒失的程序员把`IO`动作（比如这个`launchMissiles`）放在STM动作中执行，所以，以上代码会被类型系统阻止。这从另一个方面显示了将`IO`动作跟STM动作区分开来的好处。

#### 5.2.4 阻塞和选择

到目前为止我们介绍的原子块从根本上还缺乏一种能力：无法用来协调多个并发线程。这是因为还有两个关键的特性不具备：阻塞和选择。本节就来介绍如何扩

充基本的 STM 接口从而使之包含以上两个特性（当然，在完全不破坏模块性的前提下）。

假设当一个线程试图从一个账户中提取超过账户余额的钱时这个线程便会阻塞。并发编程中这类情况很常见：例如一个线程在读取到一个空的缓冲区时阻塞；或在等待一个事件的时候阻塞，等等。为了支持这种场景，我们往 STM 中加入 retry 功能，retry 的类型为：

```
retry :: STM a
```

以下是 withdraw 的一个修改过的版本，该版本当余额不足的时候便会转入到阻塞状态：

```
limitedWithdraw :: Account -> Int -> STM ()
limitedWithdraw acc amount
  = do { bal <- readTVar acc
        ; if amount > 0 && amount > bal
        then retry
        else writeTVar acc (bal - amount) }
```

retry 的语意很简单：当一个 retry 语句被执行的时候，当前事务便被“丢弃”并等待某个时候再重新执行。当然，这里的“某个时候”可以是立即，但这样做不够高效：如果重新执行的时候账户余额根本没有改变的话还是白搭，结果又是 retry。一个高效的实现不会这么做，而是会一直阻塞，直到有其他线程对 acc 进行了写操作（译注：即改变了账户余额）。那么，问题是实现怎么知道要在 acc 变量上等待呢？很简单，因为该事务在到达 retry 这一点之前的执行路径上读取的就是 acc，而这个读取动作会被事务日志完完整整地记录下来（译注：所以只要往日志里“瞄一眼”就知道了）。

limitedWithdraw 中的条件有一个非常普遍的模式：检查一个布尔条件是否满足，如果不满足则 retry。这个模式很容易抽象出来做成一个函数：

```
check :: Bool -> STM ()
check True = return ()
check False = retry
```

利用这个 check 函数来重新表达 limitedWithdraw 如下（是不是简洁了一些？）：

```
limitedWithdraw :: Account -> Int -> STM ()
limitedWithdraw acc amount
  = do { bal <- readTVar acc
        ; check (amount <= 0 || amount <= bal)
        ; writeTVar acc (bal - amount) }
```



接下来我们考虑选择（choice）。假设你想要从 A 账户上取钱，但前提是 A 上必须要有足够的钱，如果没有的话，你便改从 B 上取。为此我们必须实现以下能力：如果前一条路径需要 retry，则选择另一条路径。于是 STM Haskell 加入了另一个原语：orElse，来支持选择。orElse 的类型为：

```
orElse :: STM a -> STM a -> STM a
```

跟atomically函数一样，orElse的参数也是动作，orElse将小的动作粘合成较大的动作。其语意如下：(orElse a1 a2) 首先会执行动作a1，如果a1 发生了retry，那么它便会转而执行a2。如果a2 也retry了，那么整个动作(a1 和a2)便被retry。orElse的用法很简单：

```
limitedWithdraw2 :: Account -> Account -> Int -> STM ( )
-- (limitedWithdraw2 acc1 acc2 amt) withdraws amt from acc1,
-- if acc1 has enough money, otherwise from acc2.
-- If neither has enough, it retries.
limitedWithdraw2 acc1 acc2 amt
    = orElse (limitedWithdraw acc1 amt) (limitedWithdraw acc2 amt)
```

由于 orElse 返回的也是一个 STM 动作，因此我们便可以将 orElse 调用的结果给另一个 orElse，如此嵌套，便可以实现任意数目的备选路径。

### 5.2.5 基本 STM 操作小结

本节我们介绍了 STM Haskell 支持的所有关键的事务内存操作。表 5-1 是一个小结。注意，里面有一个操作：newTVar 到目前为止我们还没有提到。newTVar 是用来创建新的 TVar 变量的，下一节我们会用到它。

**表 5-1 STM Haskell 支持的关键操作**

操作	类型签名
atomically	STM a -> IO a
retry	STM a
orElse	STM a -> STM a -> STM a
newTVar	a -> STM (TVar a)
readTVar	TVar a -> STM a
writeTVar	TVar a -> a -> STM ( )

## 5.3 圣诞老人问题

本节将为你展示一个完整的、可运行的STM程序。一个众所周知的例子便是所谓的“圣诞老人问题”<sup>[1]</sup>，这个问题最初由Trono提出<sup>[2]</sup>：

<sup>[1]</sup> My choice was influenced by the fact that I am writing these words on December 22.

<sup>[2]</sup> J. A. Trono, “A new exercise in concurrency,” SIGCSE Bulletin, Vol. 26, pp. 8 - 10, 1994.

问题描述是这样的：圣诞老人总是在睡觉，直到被他的（放假归来的）九头驯鹿中的任意一头叫醒，或被他的十个矮人中的任意三个（一组）叫醒。如果是被驯鹿叫醒的，他就把驯鹿们套上雪橇出门去给小朋友们送礼物，回来之后再解开驯鹿（放假）。如果是被一组（三个）矮人叫醒的，他就将这三个矮人一个个带进书房，跟他们交流玩具的制造，然后再将他们一个个领出去（好让他们回去继续工作）。如果发现既有一组矮人又有一群驯鹿在等他的话，圣诞老人便优先选择驯鹿。

使用一个众所周知的例子的好处便是在一些其他语言中已经有了描述得很好的解决方案了，这样你就能够将我们马上要介绍的方案跟其他语言中既有的方案进行一目了然的比较。值得注意的是，Trono的论文中给出了一个基于信号量的方案，那个方案只是部分正确的；Ben-Ari用Ada95 和Ada给出了解决方案<sup>[1]</sup>；Benton也用Polyphonic C#（译注：C#的一个扩展，主要加入基于Join Calculus的并发编程模型）写了一个解决方案<sup>[2]</sup>。

<sup>[15]</sup> Nick Benton, “Jingle bells: Solving the Santa Claus problem in Polyphonic C#,” Technical report, Microsoft Research, 2003.

<sup>[16]</sup> Mordechai Ben-Ari, “How to solve the Santa Claus problem,” Concurrency: Practice and Experience, Vol. 10, No. 6, pp. 485 - 496, 1998.

### 5.3.1 驯鹿和矮人

用STM Haskell 来解决这个问题，基本理念是这样的：圣诞老人分别给矮人和驯鹿各创建一个“群”。每个矮人（或驯鹿）都试图去加入相应的群。如果成功的话，便得到两扇“门”。第一扇门允许圣诞老人控制什么时候让小矮人们进入书房，并得知什么时候他们全都进去了。类似的，第二扇门则控制小矮人们离开书房。圣诞老人等待他创建的任何一个群准备好，并用那个准备好的群的两扇门来控制他们的小帮手们（矮人或驯鹿）完成工作。也就是说，不管是驯鹿还是矮人，他们一直都在做一个无限循环：试图加入群——在圣诞老人的看管下穿过“门”——等待一段时间（比如驯鹿便会去休假）——再次试图加入群。

用Haskell来描述上述逻辑，我们便得到如下的代码（这是针对矮人的）<sup>[1]</sup>：

<sup>[17]</sup> 为什么是elf1 而不是elf，是因为这个函数只做一重循环，而实际上矮人会不断重复加入群。后面我们会基于elf1 来定义elf。

```
elf1 :: Group -> Int -> IO ( )
```

```
elf1 group elf_id = do { (in_gate, out_gate) <- joinGroup group
                        ; passGate in_gate
                        ; meetInStudy elf_id
                        ; passGate out_gate }
```

elf1 的参数是一个“群” (Group) 以及一个Int，后者惟一指代该小矮人的身份。这个Int只在调用meetInStudy的时候用到，meetInStudy简单地打印出一行消息表明正在发生的事情<sup>[18]</sup>：

<sup>[18]</sup>putStr是一个库函数，它会调用hPutStr stdout。

```
meetInStudy :: Int -> IO ( )
meetInStudy id = putStr ("Elf " ++ show id ++ " meeting in the
study\n")
```

小矮人调用 joinGroup 加入群，然后调用 passGate 来穿过门，这两个函数如下：

```
joinGroup :: Group -> IO (Gate, Gate)
passGate   :: Gate -> IO ( )
```

驯鹿们的代码几乎完全一样，惟一的区别就是驯鹿的任务是送礼物而不是进书房讨论：

```
deliverToys :: Int -> IO ( )
deliverToys id = putStr ("Reindeer " ++ show id ++ " delivering
toys\n")
```

由于 IO 动作也是值，所以我们便可以将驯鹿和小矮人们的逻辑抽象出一个公共模式来，如下：

```
helper1 :: Group -> IO ( ) -> IO ( )
helper1 group do_task = do { (in_gate, out_gate) <- joinGroup
group
                            ; passGate in_gate
                            ; do_task
                            ; passGate out_gate }
```

helper1 的第二个参数是一个 IO 动作，代表待执行的任务。helper1 负责将这个任务放在两个 passGate 调用之间执行。有了这个辅助函数，我们便可以基于它来定义小矮人和驯鹿函数了：

```
elf1, reindeer1 :: Group -> Int -> IO ( )
elf1      gp id = helper1 gp (meetInStudy id)
reindeer1 gp id = helper1 gp (deliverToys id)
```

### 5.3.2 门和群

先来看“门”这个抽象概念。“门”支持如下接口：

```
newGate      :: Int -> STM Gate
passGate     :: Gate -> IO ( )
operateGate  :: Gate -> IO ( )
```

每个门都有一个固定的容限  $n$ 。这个  $n$  是在我们新建门的时候指定的。此外门还有一个剩余容许量  $m$ ，这是一个可变的 (mutable) 变量。passGate 被调用的时候， $m$  就会减一。如果剩余容许量减至零，那么对 passGate 的调用就会阻塞。一个门最初被创建起来的时候剩余容许量为 0，因此任何人都不能进入。而圣诞老人则通过调用 operateGate 来打开这扇门，operateGate 会将门的剩余容许量置为  $n$ 。

下面便是门 (Gate) 的一个可能实现：

```
data Gate = MkGate Int (TVar Int)

newGate :: Int -> STM Gate
newGate n = do { tv <- newTVar 0; return (MkGate n tv) }

passGate :: Gate -> IO ( )
passGate (MkGate n tv)
    = atomically (do { n_left <- readTVar tv
                      ; check (n_left > 0)
                      ; writeTVar tv (n_left-1) })

operateGate :: Gate -> IO ( )
operateGate (MkGate n tv)
    = do { atomically (writeTVar tv n)
          ; atomically (do { n_left <- readTVar tv
                            ; check (n_left == 0) }) }
```

第一行是类型声明，声明 Gate 为一个新的数据类型，并具有一个数据构造子叫 MkGate<sup>[19]</sup>，该构造子有两个成员：一个 Int 型数据，表示该门的容限。另一个则是一个 TVar，表示在门关闭之前还有多少人可以穿过它。如果该 TVar 为 0，就表明门是关闭的。

<sup>[19]</sup>Haskell 中的类型声明不像 C 里面的结构声明，MkGate 只是一个结构 tag。

函数 newGate 负责创建一个新的门，它先是分配一个 TVar，然后通过调用 MkGate 这个构造子来创建一个 Gate 对象。无独有偶，passGate 利用模式匹配来将 MkGate 构造子拆分开来（译注：即 (MkGate n tv)），然后 passGate 将 Tvar 的值减一，

并利用 `check` 来确保 `tv > 0`。前面“阻塞和选择”一节我们实现 `withdraw` 的时候也用到过这个 `check`。最后是 `operateGate` 函数，`operateGate` 将门的最大容限值写入 `Tvar`，并等待 `Tvar` 被减至 0。

群（Group）的接口如下：

```
newGroup    :: Int -> IO Group
joinGroup   :: Group -> IO (Gate, Gate)
awaitGroup  :: Group -> STM (Gate, Gate)
```

跟门类似，群刚创建起来的时候也是空的，并带有一个指定的容限。小矮人们可以通过调用 `joinGroup` 来加入群，如果群已满那么 `joinGroup` 就会阻塞。而圣诞老人则会调用 `awaitGroup` 来等待群被加满；群满了之后，圣诞老人就会通过 `awaitGroup` 的返回值得到该群对应的两扇门，然后群立即被用两扇新门重新初始化，好让另一组焦急的小矮人们开始集合。

下面是 `newGroup` 的一个可能的实现：

```
data Group = MkGroup Int (TVar (Int, Gate, Gate))

newGroup n = atomically (do { g1 <- newGate n; g2 <- newGate n
                             ; tv <- newTVar (n, g1, g2)
                             ; return (MkGroup n tv) })
```

同样，`Group` 是一个新声明的数据类型，其构造子 `MkGroup` 具有两个成员：该 `Group` 的容限以及一个 `TVar`，后者包含该 `Group` 剩余的名额以及两个 `Gate` 对象。要创建一个新的 `Group` 对象（`newGroup`），先要创建两个 `Gate` 对象（`g1`，`g2`），并初始化一个新的 `TVar`（`tv`），然后调用 `Group` 的构造子 `MkGroup`。

而 `joinGroup` 和 `awaitGroup` 基本就是基于上面的这些数据结构来实现的：

```
joinGroup (MkGroup n tv)
  = atomically (do { (n_left, g1, g2) <- readTVar tv
                    ; check (n_left > 0)
                    ; writeTVar tv (n_left-1, g1, g2)
                    ; return (g1, g2) })

awaitGroup (MkGroup n tv)
  = do { (n_left, g1, g2) <- readTVar tv
        ; check (n_left == 0)
        ; new_g1 <- newGate n; new_g2 <- newGate n
        ; writeTVar tv (n, new_g1, new_g2)
        ; return (g1, g2) }
```

注意，`awaitGroup` 在重新初始化 `Group` 对象的时候会新建两个 `Gate` 对象。这确保了当圣诞老人在书房里讨论时，一个新群可以同时处于集结之中；如果不新建 `Gate` 对象的话，新群中的小矮人便可能会将旧群中打瞌睡的那些家伙给挤出去 [1]。

[1] 假设旧群满了，此时 `joinGroup` 已返回给该群中的每个小矮人两扇门，而圣诞老人调用的 `awaitGroup` 随后也观察到群已满，于是打开门，同时群被重新开放，但注意，群上附着的两扇门还是原来的两扇，这就表示，当另一组小矮人调用 `joinGroup` 的时候得到的返回值仍然还是那两扇门，于是新群中的小矮人便和旧群中的小矮人们同挤一扇门，如果这时（在进门时）旧群中的某个小矮人不幸睡着了（比如线程时间片用完了），便会被新群中的小矮人捷足先登了——译者注。

回顾一下这节，你可能会注意到，有些对群和门的操作是 `I0` 类型的（比如 `newGroup` 和 `joinGroup`），而有些则是 `STM` 类型的（比如 `newGate` 和 `awaitGroup`）。那么为什么这么设置呢？就拿 `newGroup` 来说吧，`newGroup` 有一个 `I0` 型的操作，也就是说我们无法在 `STM` 动作中调用它。但实际上我将 `newGroup` 做成 `I0` 型只是为了方便起见：我本可以把 `newGroup` 定义中的 `atomically` 调用去掉的，这样 `newGroup` 便成了 `STM` 类型的了，但这么一来每次调用 `newGroup` 的时候我们便都需要手动将其包在 `atomically` 之中了（`atomically(newGroup n)`）。而另一方面，将 `newGate` 做成 `STM` 操作的好处是能让它的可组合性（`composability`）更好，只不过就本应用程序来说 `newGroup` 并不需要这个可组合性，所以我才将它做成 `I0` 的，然而。由于我想在 `newGroup` 中调用 `newGate`，因此 `newGate` 的可组合性便有意义了，这便是我将 `newGate` 设为 `STM` 操作的原因。

一般来说，在设计一个库的时候，你应当尽可能地把函数的类型设为 `STM`。你可以把 `STM` 动作看作组合积木，小的 `STM` 动作可以（通过 `do {...}`, `retry`, `orElse`）组合起来成为更大的 `STM` 动作。然而，一旦你将一个块用 `atomically` 包裹起来之后，它就变成了一个 `I0` 动作，就再也不能利用 `atomically` 跟其他动作组合起来了。但 `I0` 动作也有它自己的优点：一个 `I0` 动作可以执行任意的、不可撤销的输入/输出（比如 `launchMissiles`）。

[1]（参见 `newGroup` 的实现——译者注）

因此，好的库设计应当尽可能的暴露 `STM` 动作（而不是 `I0` 动作），因为 `STM` 动作是可组合的；它们的类型表明了它们不会执行不可撤销的操作。而另一方面，库的用户总是可以很容易地将 `STM` 动作包装成 `I0` 动作（外面加一层 `atomically` 调用即可），但反过来就不行了。

然而，有时候还是必须使用 `I0` 动作的。比如 `operateGate`。`operateGate` 中的两个对 `atomically` 的调用无法并成一个，因为第一个 `atomically` 调用具有一个外部可见的副作用（开门），而第二个 `atomically` 调用则需要等到所有的小矮人们都醒过来并穿过了这扇门之后才能执行结束，否则便会一直阻塞。[1] 因此 `operateGate` 必须是 `I0` 类型的。

[1]所以，前一个 `atomically` 操作不生效，后一个 `atomically` 操作是不可能完成的——译者注。

### 5.3.3 主程序

我们首先把程序的骨架实现出来，注意，代表圣诞老人的函数（`santa`）还没有实现，但先不管它：

```
main = do { elf_group <- newGroup 3
           ; sequence_ [ elf elf_group n | n <- [1..10] ]

           ; rein_group <- newGroup 9
           ; sequence_ [ reindeer rein_group n | n <- [1..9] ]

           ; forever (santa elf_group rein_group) }
```

第一行创建了一个大小为 3 的群。第二行则稍微需要解释一下：它利用了所谓的列表内涵式（`list comprehension`）来创建一组 IO 动作，然后调用 `sequence_` 来顺序执行它们。列表内涵式 “[`e | x <- xs`]” 读作 “由一切 `e` 所构成的列表，其中 `x` 来自列表 `xs`”。因此本例中 `sequence_` 的参数为：

```
[elf elf_group 1, elf elf_group 2, ..., elf elf_group 10]
```

这些调用每个都会返回一个 IO 动作，后者在被执行的时候会新建一个小矮人线程。而 `sequence_` 函数则接受一组 IO 动作作为参数，返回的也是一个 IO 动作，后者被执行的时候会按列表中的顺序执行那组作为参数的 IO 动作<sup>[20]</sup>。

<sup>[20]</sup>类型 “[IO a]” 读作 “一个由类型为 IO a 的值构成的列表”。另外你可能会奇怪 `sequence_` 后面为什么要加上一个下划线，其实这是因为另外还有一个与它相关的函数也叫 `sequence`，这个 `sequence` 的类型则是 `[IO a] -> IO [a]`，其功能是将一组动作执行后的结果收集到一个列表中。`sequence` 和 `sequence_` 都定义在 `Prelude` 库中（`Prelude` 库是缺省导入的）。

```
sequence_ :: [IO a] -> IO ()
```

`elf` 函数是基于 `elf1` 写的，但有两点区别。首先是我们想要 `elf` 重复执行，每重循环延迟一个不确定的时间间隔；其次，我们想要它在单独的线程中运行：

```
elf :: Group -> Int -> IO ThreadId
elf gp id = forkIO (forever (do { elf1 gp id; randomDelay })))
```

`forkIO` 将它的参数（一个动作）放在一个单独的 Haskell 线程中执行（见前面的小节 “Haskell 中的副作用和输入/输出”）。`forkIO` 的实参是一个对 `forever` 的调用；`forever`，顾名思义，会将一个动作重复执行（一个与它类似但有微妙差别的函数是 `nTimes`，见 “Haskell 中的副作用和输入/输出”）：



```

forever :: IO () -> IO ()
-- Repeatedly perform the action
forever act = do { act; forever act }

```

最后，表达式(`elf1 gp id`)是一个 IO 动作，我们想要将它不确定地重复执行，每次执行随机延迟一段时间：

```

randomDelay :: IO ()
-- Delay for a random time between 1 and 1,000,000 microseconds
randomDelay = do { waitTime <- getStdRandom (randomR (1, 1000000))
                  ; threadDelay waitTime }

```

主程序中剩下的部分含义就很明显了。创建 9 头驯鹿和创建 10 个小矮人的方式是一样的：

```

reindeer :: Group -> Int -> IO ThreadId
reindeer gp id = forkIO (forever (do { reindeer1 gp id;
randomDelay })))

```

主程序的最后一行代码利用 `forever` 来运行 `santa`。下面我们就来说说最后一个问题——圣诞老人 (Santa) 的实现。

### 5.3.4 圣诞老人的实现

圣诞老人是这个问题里面最有趣的，因为他会进行选择。他必须等到一组驯鹿或一组小矮人在那儿等他的时候才会继续行动。一旦他选择了是带领驯鹿还是小矮人之后，他便将他们带去做该做的事。圣诞老人的代码如下：

```

santa :: Group -> Group -> IO ()
santa elf_gp rein_gp
  = do { putStr "-----\n"
        ; (task, (in_gate, out_gate))
          <- atomically (orElse
                        (chooseGroup rein_gp "deliver toys")
                        (chooseGroup elf_gp "meet in my study"))

        ; putStr ("Ho! Ho! Ho! let's " ++ task ++ "\n")
        ; operateGate in_gate
        -- Now the helpers do their task
        ; operateGate out_gate }

where
  chooseGroup :: Group -> String -> STM (String, (Gate, Gate))
  chooseGroup gp task = do { gates <- awaitGroup gp
                            ; return (task, gates) }

```



圣诞老人进行选择的关键就在那个 `orElse` 上。`orElse` 首先会试图选择驯鹿（驯鹿优先），如果驯鹿没有准备好就选择小矮人。`chooseGroup` 会对相应的群调用 `awaitGroup`，并返回一个对偶（pair）“`(task, gates)`”，其中 `task` 是一个字符串，代表待执行的任务（“`deliver toys`”或“`meet in my study`”），`gates` 则本身又是一个对偶，它包含的是两扇门，圣诞老人通过操作这两扇门来带领一群小矮人或驯鹿完成任务。一旦在驯鹿和小矮人之间的选择完成了，圣诞老人便打印出一则消息表示待执行的任务，并依次操纵（`operateGate`）两扇门。

该实现工作起来自然是没问题的，但不妨让我们来看看另一个实现，这个实现更具一般性，因为圣诞老人的程序显示出了一个非常普遍的模式：一个线程（本例中是圣诞老人）在一个原子事务中作了一次选择，并根据选择的结果接着执行一个或多个事务。另一个典型的场景是：从多个消息队列中获取一则消息，并针对该消息做一些事情，然后重复这个过程。在圣诞老人问题里面，这里的后续操作对小矮人和对驯鹿基本是一样的——两种情况下圣诞老人都得打印一则消息并操纵两扇门。如果对小矮人的逻辑和对驯鹿的逻辑差别很大的话刚才上面那种做法就行不通了，一个补救的办法是使用一个布尔变量来表示到底选择了小矮人还是驯鹿，并根据具体选择了哪一方来决定做什么事情；但一旦选择的可能性多了，这种做法同样还是不够方便。下面是一个更好的解决方案：

```
santa :: Group -> Group -> IO ()
santa elf_gp rein_gp
  = do { putStr "-----\n"
        ; choose [(awaitGroup rein_gp, run "deliver toys"),
                  (awaitGroup elf_gp, run "meet in my study")] }
where
  run :: String -> (Gate, Gate) -> IO ()
  run task (in_gate, out_gate)
    = do { putStr ("Ho! Ho! Ho! let's " ++ task ++ "\n")
          ; operateGate in_gate
          ; operateGate out_gate }
```

`choose`函数就像一个“保险命令”一样：它接受一组对偶（pairs），等到某个对偶的第一个元素可以“开火”了，便执行其第二个元素。因此`choose`的类型如下<sup>[21]</sup>：

<sup>[21]</sup>在Haskell中，类型`[ty]`的意思是一个元素类型为`ty`的列表。本例中`choose`的参数为一个由对偶（`(ty1, ty2)`）构成的列表；其中对偶的第一个元素的类型为`STM a`，第二个元素则是一个函数，类型为`a -> IO ()`。

```
choose :: [(STM a, a -> IO ())] -> IO ()
```

刚才提到“保险命令”的比喻，这里的“保险”就是对偶的第一个元素，即一个STM动作，返回类型为`a`；当这个STM动作“准备好了”（不发生`retry`）之后，`choose`便可以将其返回值传给对偶的第二个元素，当然，后者必须是一个函数，且接受一个`a`类型的参数。了解了以上这些之后再来阅读 `santa` 的代码就应该毫无困难了。`santa` 利用 `awaitGroup` 来等待一个群准备好；`choose` 拿到 `awaitGroup`

返回的两扇门之后便将它们传给 `run` 函数，后者依次操纵这两扇门——`operatorGate` 会一直阻塞，直到所有小矮人（或驯鹿）都穿过门之后才会返回。

`choose` 的代码虽然只有寥寥数行，但要真正弄明白它还是得费点脑筋的：

```
choose :: [(STM a, a -> IO ( ))] -> IO ( )
choose choices = do { act <- atomically (foldr1 orElse actions)
                    ; act }

where
  actions :: [STM (IO ( ))]
  actions = [ do { val <- guard; return (rhs val) }
            | (guard, rhs) <- choices ]
```

首先来看 `actions`，`actions` 是一个列表，它的每个元素都是一个 STM 动作。`choose` 将 `actions` 和 `orElse` 用 `foldr1` 结合起来（`foldr1 orElse [x1, ..., xn]` 的结果是 `x1 orElse x2 orElse x3 ... orElse xn`）。这些 STM 动作（即 `actions` 列表里面的元素）每个又都返回一个 IO 动作（所以才有 `STM(IO( ))` 这个类型），后者也就是它一旦被选中之后要做的事情。`choose` 首先在各个动作之间作一次原子选择，取得返回出来的动作（`act`，类型为 `IO( )`），然后执行该动作。而列表 `actions` 又是如何构造出来的呢？答案是只需对 `choices` 列表里面的每个对偶 `(guard, rhs)`，运行 `guard`（一个 STM 动作），再将 `guard` 的返回值作为参数交给 `rhs`，并返回后者执行的结果即可。

### 5.3.4 编译并运行程序

以上便是这个例子的全部代码。要运行它，只需在程序开头再添上几个 `import` 语句即可<sup>[1]</sup>：

<sup>[1]</sup>代码也可以在这里下载到：<http://research.microsoft.com/~simonpj/papers/stm/Santa.hs.gz>

```
module Main where
import Control.Concurrent.STM
import Control.Concurrent
import System.Random
```

使用 GHC (Glasgow Haskell Compiler) 编译代码<sup>[2]</sup>：

<sup>[2]</sup>GHC 是免费的，在这里下载：<http://haskell.org/ghc>。

```
$ ghc Santa.hs -package stm -o santa
```

最后运行：

```
$ ./santa
```

```
Ho! Ho! Ho! let's deliver toys
Reindeer 8 delivering toys
Reindeer 7 delivering toys
Reindeer 6 delivering toys
Reindeer 5 delivering toys
Reindeer 4 delivering toys
Reindeer 3 delivering toys
Reindeer 2 delivering toys
Reindeer 1 delivering toys
Reindeer 9 delivering toys
-----
Ho! Ho! Ho! let's meet in my study
Elf 3 meeting in the study
Elf 2 meeting in the study
Elf 1 meeting in the study
...and so on...
```

## 5.4 对 Haskell 的一些思考

Haskell 首先是一门函数式编程语言，但我觉得它同样也是世界上最漂亮的命令式语言。如果把 Haskell 当成一门命令式语言来看待的话，我们会发现，它具有一些不寻常的特性：

- 动作（有副作用）和纯值（无副作用）被严格区分开来。
- 动作也是真正的值。它们可以被作为参数、作为返回值、放在列表内等等，所有这些操作都不会带来副作用[1]。

[1]（[因为并不会导致动作被执行](#)——译者注

利用动作作为第一类值，我们便可以借助于它来定义应用相关的控制结构，而不是受制于语言设计者定义的那一套。例如，`nTimes` 就是一个简单的 `for` 循环结构。而刚才提到的 `choose` 则实现了一种可以称之为“保险命令”的功能。动作也是值的这一性质还有许多其他应用。在 `main` 函数中，我们利用 Haskell 丰富的表达力（列表内涵式）生成了一系列动作，然后再利用 `sequence_` 来依次执行这些动作。同样，前面在定义 `helper1` 的时候，我们也是利用的这一性质：我们从一块代码中抽象出了一个动作，从而提升了代码的模块性。当然，圣诞老人的实现代码量本就不多，动用 Haskell 的这些强大的抽象能力仿佛有点杀鸡使用牛刀的感觉，不过一来这里只是为了展现 Haskell 的优点，二来对于大型程序来说，动作也是值的，这一性质无论说多重要都不为过。

此外，Haskell 还有许多强大之处文中并没有提到，如高阶函数，惰性求值，数据类型，多态，类型类（`type class`）等等，因为本文关注的是并发。Haskell 程序很少有像本文中的例子这样“命令式”的！如果想了解更多 Haskell 的知识，

可以访问<http://haskell.org>, 上面有书、指南、编译器、解释器、库、邮件列表和许多其他资源。

## 5.5 结论

本文的主要目的是要让你相信, 用 STM Haskell 写出的并发程序从根本上比利用传统的锁和条件变量写出的并发程序的模块性更好。不过首先值得注意的一点是事务内存帮我们完全避免了基于锁的并发编程中种种令人头疼的经典问题(见“生锈的锁”一节)。所有这些问题在 STM Haskell 中完全消失不见了。Haskell 的类型系统会防止你在原子块之外读写一个 TVar, 而且, 由于不再存在对程序员可见的锁, 因此加哪把锁、按什么顺序加锁的问题也就不复存在了。此外 STM 还有其他好处, 但这里限于篇幅我写不下了(包括不再有忘记唤醒以及异常和错误恢复这些令人头疼的问题)。

不过, 我最想说的一个问题还是可组合性(composability)问题, 正如“生锈的锁”一节提到的, 这是基于锁的编程中一个最严重的问题。但在 STM Haskell 中, 任何 STM 类型的函数都可以顺序地或通过选择来与其他任何 STM 类型的函数组合起来形成一个新的 STM 型的函数, 新的函数能确保具有组成它的各个函数的原子属性。特别是阻塞(retry)和选择(orElse)这两个功能, 如果用锁来实现的话从根本上就不具备模块性, 然而在 STM Haskell 中则是完全模块化的。例如, 考虑下面这个事务, 它用到的 limitedWithdraw 函数曾在“阻塞和选择”一节定义:

```
atomically (do { limitedWithdraw a1 10
                ; limitedWithdraw2 a2 a3 20 })
```

这个事务从阻塞中恢复的前提是账户 a1 上至少要有 10 块钱, 而 a2 和 a3 之中则至少要有有一个账户上多于 20 块钱。关键是, 这么复杂的阻塞条件并不需要程序员显式写出, 而且, 如果 limitedWithdraw 系列函数是位于一个成熟的库中的话, 程序员甚至根本不知道它们的阻塞条件是什么。总之一句话: STM 是模块性的, 小的程序可以粘合成大的程序, 无需暴露其内部实现。

本文只能说是对事务内存作了一个简单的概览, 实际上事务内存还有许多其他有意思的主题文中没有提到, 比如重要的有嵌套事务、异常、线程进展、饿死、不变式等。其中许多在关于 STM Haskell 的一些论文中都讨论过<sup>[23]</sup>。

<sup>[23]</sup> Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy, “Composable memory transactions,” ACM Symposium on Principles and Practice of Parallel Programming (PPoPP ’05), June 2005; Tim Harris and Simon Peyton Jones, “Transactional memory with data invariants,” First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT ’06), Ottawa, June 2006, ACM; Anthony Discolo, Tim Harris, Simon Marlow, Simon Peyton Jones, and Satnam Singh, “Lock-free data structures using STMs in Haskell,” Eighth International Symposium on Functional and Logic Programming (FLOPS ’06), April 2006.

说实话，事务内存和Haskell可谓天造地设的一对。STM的实现理论上可能要跟踪每一次内存读写，然而Haskell中的STM实现却只需跟踪TVar操作就行了，而TVar操作只占有内存操作的极小一部分。此外，由于Haskell中的动作也是值，再加上Haskell丰富的类型系统，就使得我们无需对语言作任何扩展便能够提供强大的静态保证。不过话说回来，事务内存并非不适用于主流的命令式语言，虽然实现起来可能没这么优雅，并且可能需要更多的语言支持。目前STM是一个热门的研究课题；Larus和Rajwar对这个领域的研究作了一个全面的概述<sup>[1]</sup>。

<sup>[1]</sup> James Larus and Ravi Rajwar, Transactional Memory, Morgan & Claypool, 2006.

STM之于传统的并发编程技术就好比高阶语言之于汇编语言——你仍然还有可能写出有问题的程序，但许多棘手的bug却不可能再出现了，而且关注程序的高阶性质也使得编程容易得多。虽然并发编程并无银弹，但STM看起来是一个很有前途的进展，它能帮你编写出更美的代码。

## 5.6 致谢

很多人对本文的改进提了很好的建议和意见：Bo Adler, Justin Bailey, Matthew Brecknell, Paul Brown, Conal Elliot, Tony Finch, Kathleen Fisher, Greg Fitzgerald, Benjamin Franksen, Jeremy Gibbons, Tim Harris, Robert Helgesson, Dean Herington, David House, Brian Hulley, Dale Jordan, Marnix Klooster, Chris Kuklewicz, Evan Martin, Greg Meredith, Neil Mitchell, Jun Mukai, Michal Palka, Sebastian Sylvan, Johan Tibell, Aruthur van Leeuwen, Wim Vanderbauwhede, David Wakeling, Dan Wang, Peter Wasilko, Eric Willigers, Gaal Yahas, and Brian Zimmer。特别要感谢Kirsten Chevalier, Andy Oram, 和 Greg Wilson他们对文章作了非常详细的审阅。

## 第 6 章 以 REST 方式集成业务伙伴

Andrew Patzer

在几年以前，当时我还是一个顾问，曾经在 1 到 2 年的时间里，似乎每个与我交谈过的客户都非常肯定地认为在他的业务中需要一个 Web 服务解决方案。当然，虽然在我的客户中很少有人能真正地理解这意味着什么或者为什么他需要这种架构，但由于他们总是不断地在互联网上杂志上以及博览会上听说到 Web 服务，因此他们认为最好能搭上 Web 服务这趟车，以免错过机会。

请不要误会我的意思。我并不是要反对 Web 服务。我只是不热衷于仅仅根据一些时髦的东西来做出技术决策。本章将给出一些使用 Web 服务架构的理由，并将研究在与外部系统进行集成时所要考虑的一些选择。

在本章中，我将分析一个真实的项目，在这个项目中包括把一组服务开放给某个业务伙伴，此外我们还将讨论一些相关的设计决策。在项目中使用到的技术包括 Java (J2EE)、XML、Rosettanet 电子商务协议和一个用来与运行在 AS/400 系统上的程序进行通信的函数库。我还将讲述接口的使用和工厂 (Factory) 设计模式，我正是通过这种模式使得系统对于将来的销售商来说是可扩展的，而这些销售商可能使用不同的协议或者可能需要访问不同的服务。

### 6.1 项目背景

在本章中讨论的项目开始于一位客户的电话：“我们需要一组 Web 服务把我们的系统和一位销售商的系统集成在一起。”这位客户是一个大型的电子元件制造商。他们使用的系统是 MAPICS，这是一个用 RPG 编写的制造系统，运行在 AS/400 机器上。他们的主要销售商正在升级自己的业务系统并且需要修改与订单管理系统的连接方式以检查产品的现货供应能力和订单状态。

之前，销售商处的操作人员只是远程连接到制造商的 AS/400 系统，并且通过按下“热键 (hotkey)”（我记得是 F13 或者 F14）来访问所需的界面。在随后的代码中你将看到，我为他们开发的新系统叫做 hotkey，这是因为 hotkey 这个单词已经变成了他们常用语言的一部分，就好像 google 在今天已经演变成了一个动词一样。

既然销售商正在实现一个新的电子商务系统，那么它就需要一种自动的方式来把制造商的数据集成到自己的系统。由于这只是客户的销售商之一，尽管是最大的销售商，但客户系统还是需要考虑支持在将来加入其他的销售商以及他们可能使用的任何协议和需求。还有一点就是维护和扩展这个系统的软件人员的技术水平相对较低。虽然他们在其他的领域是非常棒的，但 Java 开发（以及所有类型的 Web 开发）对于他们来说仍然是很新的东西。因此，我知道我所构建的系统必须是简单的并且易于扩展的。

### 6.2 把服务开放给外部客户

在进行这个项目之前，我向我们的用户组和关于 SOAP（简单对象访问协议，Simple Object Access Protocol）和 Web 服务架构的会议做了一些技术报告。因此，当客户电话打



来的时候，似乎我所讲述的东西正是这个客户正在寻找的解决方案。然而，在理解了他们的真正需求之后，我认为更好的方式是通过 HTTP 上简单的 GET 和 POST 请求把一组服务开放出去，并且在服务中交换描述这些请求和响应的 XML 数据。不过我在当时并不知道，这种架构形式现在通常叫做 REST，或者叫做具象状态传输（Representational State Transfer）。

我如何决定在 SOAP 上使用 REST？下面是一些在选择 Web 服务架构时需要考虑的决策因素：

有多少不同的系统需要访问这些服务，并且在当时有多少系统是已知的？

虽然这个制造商知道目前只有一个销售商需要访问这个系统，但它也承认其他的销售商在将来也可能需要访问。

是否有一些终端用户需要预先了解这些服务，或者这些服务是否需要是自描述的，以便于让匿名用户自动进行连接？

因为在制造商及其所有的销售商之间存在一个确定的关系，所以需要保证每个潜在的终端用户都要预先知道如何访问制造商的系统。

在单个事务中需要维护什么样的状态？一个请求是否依赖于前一个请求的结果？

在我们的情况中，每个事务都包含一个请求和一个不依赖于其他任何信息的结果。

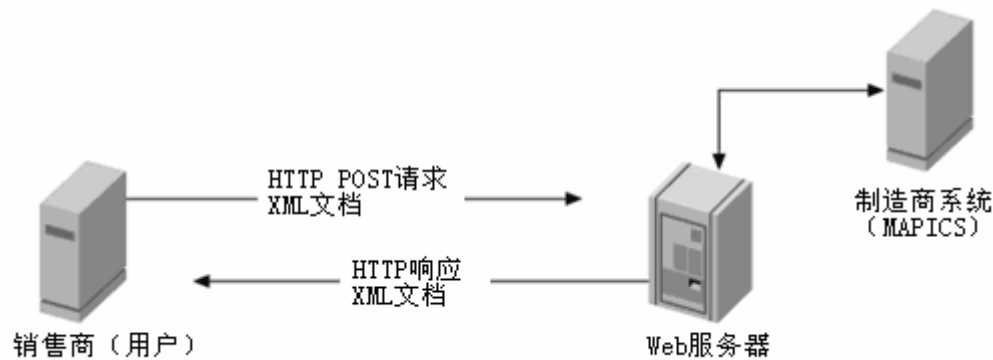
在这个项目中对上述问题的回答使我得出了显而易见的决策：在 HTTP 协议上开放一组已知的服务，并且使用在双方系统中都能够理解的标准电子商务协议来交换数据。如果制造商希望匿名用户也能够查询产品的现货供应能力，那么我可以选择完整的 SOAP 解决方案，因为这将使系统能够发现这些服务以及可编程的接口而无需预先了解系统。

我目前从事生物信息领域，在这个领域中明确需要 SOAP 形式的 Web 服务架构。我们利用了一个叫作 BioMoby (<http://www.biomoby.org>) 的项目来定义 Web 服务并且把它们发布到一个中央存储仓库，以使得其他小组能够正确地把我们的服务应用在构建数据管道的工作流中，从而帮助生物学家们集成不同集合的数据并且对结果进行不同的分析。这是一个完美的示例，它很好地说明了为什么有人选择在 REST 上的 SOAP。匿名用户可以访问我们的数据和工具而甚至无需预先知道它们的存在。

## 6.2.1 定义服务接口

在确定如何实现这个软件时，首先要确定的就是用户如何发出请求和接受响应。在和这个销售商（主要用户）的一位技术代表讨论之后，我了解到他们的新系统可以通过 HTTP POST 请求发送一个 XML 文档，并且把结果作为一个 XML 文档来分析。XML 必须遵循 Rosettanet 电子商务协议（后面还将做更详细的讨论）的格式，但就目前来说，只要知道这个系统能够通过发送 XML 格式的请求和响应在 HTTP 上进行通信就足够了。在图 6-1 中说明了各个系统之间的常见交互。

图 6-1 后端系统的服务接口



这个制造商最近被一个更大的公司收购了，这个公司在整个组织内都使用 IBM 的产品。因此，我已经知道了所使用的是什么样的应用服务器及相关技术。我把这个服务实现为一个运行在 IBM WebSphere 上的 Java Servlet。由于我知道这个软件将需要使用基于 Java API 来访问运行在 AS/400 服务器上的函数，因此很容易做出这个决策。

下面是在 web.xml 文件中的代码，这个文件描述是将为用户提供必要接口的 servlet：

```
<servlet>
    <servlet-name>HotKeyService</servlet-name>
    <display-name>HotKeyService</display-name>
    <servlet-class>com. xxxxxxxxxxxx. hotkey. Service</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>HotKeyService</servlet-name>
    <url-pattern>/HotKeyService</url-pattern>
</servlet-mapping>
```

servlet 本身仅处理 POST 请求，这是通过重载 Servlet 接口的 doPost 方法并且提供了标准生命周期方法的默认实现来完成的。在下面的代码中给出了这个服务的完整实现，但当我最初对问题进行分解并设计一个解决方案时，我首先在代码中写下了一系列的注释作为占位符用来表示将要插入代码的位置。然后我将逐步用代码来代替每个伪码注释，直至最终得出可以运行的实现为止。这将有助于我把注意力放在每段代码如何关联到整个解决方案上：

```
public class Service extends HttpServlet implements Servlet {

    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        //读取请求数据并且保存在 StringBuffer 中
        BufferedReader in = req.getReader( );
        StringBuffer sb = new StringBuffer( );
        String line;
        while ((line = in.readLine( ))!= null) {
            sb.append(line);
        }

        HotkeyAdaptor hotkey = null;
```



```
if (sb.toString( ).indexOf("Pip3A2PriceAndAvailabilityRequest") > 0) {
    //价格和现货供应信息请求
    hotkey = HotkeyAdaptorFactory.getAdaptor(
        HotkeyAdaptorFactory.ROSETTANET,
        HotkeyAdaptorFactory.PRODUCTAVAILABILITY);
}
else if (sb.toString( ).indexOf("Pip3A5PurchaseOrderStatusQuery ") > 0)
{
    //订单状态
    hotkey = HotkeyAdaptorFactory.getAdaptor(
        HotkeyAdaptorFactory.ROSETTANET,
        HotkeyAdaptorFactory.ORDERSTATUS);
}

boolean success = false;

if (hotkey != null) {
    /*传入 XML 请求数据*/
    hotkey.setXML(sb.toString( ));
    /*解析请求数据*/
    if (hotkey.parseXML( )) {
        /*执行 AS/400 程序*/
        if (hotkey.executeQuery( )) {
            /*返回响应的 XML */
            resp.setContentType("text/xml");
            PrintWriter out = resp.getWriter( );
            out.println(hotkey.getResponseXML( ));
            out.close( );
            success = true;
        }
    }
}

if (!success) {
    resp.setContentType("text/xml");
    PrintWriter out = resp.getWriter( );
    out.println("Error retrieving product availability.");
    out.close( );
}

}
```

仔细阅读这段代码，你可以看到它首先读取请求数据并且将其保存起来以用于后面的操作。然后它将在这个数据中进行查找以确定其是哪种类型的请求：是价格和现货供应信息请求，还是订单状态查询请求。在确定了请求的类型后，将会创建相应的辅助对象。注意，我使用接口 `HotkeyAdaptor` 来获得多个实现而无需为每种类型的请求编写大段重复的代码。

这个方法的其他功能包括解析 XML 请求数据，在 AS/400 系统上执行合适的查询，创建 XML 响应并把它通过 HTTP 写回到用户。在下一节中，你将看到我如何通过接口和非常流行的工厂设计模式把实现细节隐藏起来。

## 6.3 使用工厂模式转发服务

这个系统的需求之一就是通过最少的编程工作来满足将来不同类型系统的多种请求。我相信我能够通过把实现简化为单个命令接口来满足这个需求，在这个接口中将开放一些基本的方法来响应各种请求：

```
public interface HotkeyAdaptor {  
  
    public void setXML(String _xml);  
    public boolean parseXML( );  
    public boolean executeQuery( );  
    public String getResponseXML( );  
  
}
```

那么，这个 servlet 如何确定对接口的哪种实现进行实例化？它将首先在请求数据中查找特定的字符串以判断请求的类型是什么。然后，它使将用工厂对象的静态方法来选择合适的实现。

这个 servlet 知道，我们所使用的实现将会为每个方法提供合适的响应。通过使用主 servlet 中的接口，我们只需把执行代码编写一次，而无需考虑它所处理的请求是何种类型或者是谁发出的请求。所有的细节都被封装在接口的每个独立实现中。以下是这个 servlet 中的一些代码：

```
HotkeyAdaptor hotkey = null;  
  
if (sb.toString( ).indexOf("Pip3A2PriceAndAvailabilityRequest") > 0) {  
    //价格和现货供应信息请求  
    hotkey = HotkeyAdaptorFactory.getAdaptor(  
        HotkeyAdaptorFactory.ROSETTANET,  
        HotkeyAdaptorFactory.PRODUCTAVAILABILITY);  
}  
else if (sb.toString( ).indexOf("Pip3A5PurchaseOrderStatusQuery ") > 0)  
{  
    //订单状态  
    hotkey = HotkeyAdaptorFactory.getAdaptor(  
        HotkeyAdaptorFactory.ROSETTANET,  
        HotkeyAdaptorFactory.ORDERSTATUS);  
}
```

在 `HotkeyAdaptorFactory` 这个工厂对象中定义了一个静态方法，这个方法包含两个参

数，分别表示它所使用的是哪种协议以及是哪种类型的请求。这些参数的值都被定义为工厂对象本身的静态常量。在下面的代码中可以看到，工厂对象只是简单地使用一个 switch 语句来选择合适的实现：

```
public class HotkeyAdaptorFactory {

    public static final int ROSETTANET = 0;
    public static final int BIZTALK = 1;
    public static final int EBXML = 2;

    public static final int PRODUCTAVAILABILITY = 0;
    public static final int ORDERSTATUS = 1;

    public static HotkeyAdaptor getAdaptor(int _vocab, int _target) {

        switch (_vocab) {
            case (ROSETTANET) :
                switch (_target) {
                    case (PRODUCTAVAILABILITY) :
                        return new HotkeyAdaptorRosProdAvailImpl( );
                    case (ORDERSTATUS) :
                        return new HotkeyAdaptorRosOrdStatImpl( );
                    default :
                        return null;
                }
            case (BIZTALK) :
            case (EBXML) :
            default :
                return null;
        }
    }
}
```

虽然这看上去是一个非常简单的抽象，但在使得经验欠缺的程序员能够阅读并且理解这段代码之前，我们付出了很多的努力。当需要增加一个新的销售商，并且这个销售商正在使用 Microsoft 的 BizTalk 服务，同时他还希望通过电子的方式下订单时，程序员就可以用一个简单的模板来增加这个新的需求。

## 6.4 用电子商务协议来交换数据

在这个项目中，有些新的东西是关于标准电子商务协议使用的。当销售商向我提出使用 Rosettanet 标准来交换请求和响应的需求时，我不得不首先做了一些研究。我首先从 Rosettanet 网址 (<http://www.rosettanet.org>) 上下载了我所感兴趣的具体标准，从中找到了一张详细讲解业务伙伴之间典型交易的示意图，以及关于 XML 请求和响应的规范。

由于我需要做许多反复试验的工作，因此首要的事情就是建立一个测试环境，这样我可以模拟与销售商的交互过程而无需每次再与他们的工作人员联合测试。我使用了 Apache Commons HttpClient 来管理 HTTP 交换：

```
public class TestHotKeyService {

    public static void main (String[] args) throws Exception {

        String strURL = "http://xxxxxxxxxx/HotKey/HotKeyService";
        String strXMLFilename = "SampleXMLRequest.xml";
        File input = new File(strXMLFilename);

        PostMethod post = new PostMethod(strURL);
        post.setRequestBody(new FileInputStream(input));
        if (input.length( ) < Integer.MAX_VALUE) {
            post.setRequestContentLength((int)input.length( ));
        } else {
            post.setRequestContentLength(
                EntityEnclosingMethod.CONTENT_LENGTH_CHUNKED);
        }

        post.setRequestHeader("Content-type", "text/xml; charset=ISO-8859-1");

        HttpClient httpclient = new HttpClient( );
        System.out.println("[Response status code]: " +
            httpclient.executeMethod(post));
        System.out.println("\n[Response body]: ");
        System.out.println("\n" + post.getResponseBodyAsString( ));

        post.releaseConnection( );

    }

}
```

在试验了几种不同类型的请求并分析了结果之后，我加快了自己的学习曲线。我坚定不移地认为，越快开始编写代码，就越能提高学习效果。你从书中、某个网站上的一篇文章或者一组 API 文档中只能学习部分知识。只有尽早地着手把学到的东西应用起来，才能发现许多通过简单研究这个问题无法发现的事情。

Rosettanet 标准与其他标准一样，是非常详尽的和完整的。在完成任何任务的时候，你可能最终只会用到其中的一小部分内容。对于这个项目来说，我只需要设置一些标准的标识域，以及在价格查询时设置一个产品编号和有效期，或者在查询订单状态时设置一个订单号。

## 6.4.1 用 XPath 解析 XML

XML 请求数据不仅仅只是简单的 XML。正如在前面所提到的，Rosettanet 标准是非常详尽和完整的。如果没有 XPath，那么解析这样的一个文档将是一场可怕的恶梦。通过使用 XPath 映射，我可以定义到我所感兴趣的每个节点的精确路径，并且能够很容易地提取出必要的信息。我把这些映射实现为一个 HashMap，然后在其中进行迭代，并且提取出特定的节点再用这些值来创建一个新的 HashMap。这些值将被同时用在 executeQuery 和 getResponseXML 这两个方法中，我在后面将会介绍这些方法：

```
public class HotkeyAdaptorRosProdAvailImpl implements HotkeyAdaptor {

    String inputFile;           //请求 XML
    HashMap requestValues;      //保存请求中的 XML 值
    HashMap as400response;      //保存从 RPG 调用中返回的参数

    /*声明 XPath 映射并且用一个静态的初始化块来赋值*/
    public static HashMap xpathmappings = new HashMap( );
    static {
        xpathmappings.put("from_ContactName",
            "//Pip3A2PriceAndAvailabilityRequest/fromRole/PartnerRoleDescription/ContactInf
            ormation/contactName/FreeFormText");
        xpathmappings.put("from_EmailAddress",
            "//Pip3A2PriceAndAvailabilityRequest/fromRole/PartnerRoleDescription/ContactInf
            ormation/EmailAddress");
    }

    //为了保持简洁性而省略 xpath 映射...

    public HotkeyAdaptorRosProdAvailImpl( ) {
        this.requestValues = new HashMap( );
        this.as400response = new HashMap( );
    }

    public void setXML(String _xml) {
        this.inputFile = _xml;
    }

    public boolean parseXML( ) {

        try {
            Document doc = null;
            DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance( );
            DocumentBuilder db = dbf.newDocumentBuilder( );
            StringReader r = new StringReader(this.inputFile);
            org.xml.sax.InputSource is = new org.xml.sax.InputSource(r);
```

```
doc = db.parse(is);

Element root    = doc.getDocumentElement( );

Node node = null;

Iterator xpathvals = xpathmappings.values().iterator( );
Iterator xpathvars = xpathmappings.keySet().iterator( );
while (xpathvals.hasNext() && xpathvars.hasNext( )) {
    node                =                XPathAPI.selectSingleNode(root,
String)xpathvals.next( ));
    requestValues.put((String)xpathvars.next( ),
                      node.getChildNodes().item(0).getNodeValue( ));
}

}

catch (Exception e) {
    System.out.println(e.toString( ));
}

return true;
}

public boolean executeQuery( ) {
    //以下代码省略...
}

public String getResponseXML( ) {
    //以下代码省略...
}

}
```

在 `executeQuery` 方法中包含了在访问运行于 AS/400 系统之上的 RPG 代码所需的代码，这是为了获得在随后构造响应 XML 文档时必要的响应数据。在许多年以前，我曾工作过的一个项目是把一个 MAPICS 系统（运行在 AS/400 上的 RPG）和一个我用 Visual Basic 编写的新系统集成在一起。我在数据交换的两端分别用 AS/400 上的 RPG、CL 以及 PC 上面的 Visual Basic 来编写代码。这使我不得不做了好几个演讲报告，在报告中我努力向许多 RPG 程序员讲述如何把他们的遗留系统和现代的客户/服务器软件集成起来。在当时，这确实是一件复杂而又神秘的事情。

从那以后，IBM 使这项任务变得非常容易并且为我们提供了一个 Java 函数库来做所有的工作（**这就是我在这个项目中得到的所有东西**）。在以下的代码中使用了来自 IBM 的 Java 库：

```
public boolean executeQuery( ) {
```

```
StringBuffer sb = new StringBuffer( );

sb.append(requestValues.get("from_ContactName")).append("|");
sb.append(requestValues.get("from_EmailAddress")).append("|");
sb.append(requestValues.get("from_TelephoneNumber")).append("|");
sb.append(requestValues.get("from_BusinessIdentifier")).append("|");
sb.append(requestValues.get("prod_BeginAvailDate")).append("|");
sb.append(requestValues.get("prod_EndAvailDate")).append("|");
sb.append(requestValues.get("prod_Quantity")).append("|");
sb.append(requestValues.get("prod_ProductIdentifier")).append("|");

try {
    AS400 sys = new AS400("SS100044", "ACME", "HOUSE123");

    CharConverter ch = new CharConverter( );
    byte[] as = ch.stringToByteArray(sb.toString( ));

    ProgramParameter[] parmList = new ProgramParameter[2];
    parmList[0] = new ProgramParameter(as);
    parmList[1] = new ProgramParameter(255);

    ProgramCall pgm = new ProgramCall(sys,
        "/QSYS.LIB/DEVOBJ.LIB/J551231.PGM", parmList);
    if (pgm.run( ) != true) {
        AS400Message[] msgList = pgm.getMessageList( );
        for (int i=0; i < msgList.length; i++) {
            System.out.println(msgList[i].getID( ) + " : " +
                msgList[i].getText( ));
        }
    }
    else {
        CharConverter chconv = new CharConverter( );
        String response =

chconv.byteArrayToString(parmList[1].getOutputData( ));

        StringTokenizer st = new StringTokenizer(response, "|");

        String status = (String) st.nextToken().trim( );
        as400response.put("Status", status);
        String error = (String) st.nextToken().trim( );
        as400response.put("ErrorCode", error);
        String quantity = (String) st.nextToken().trim( );
        as400response.put("Quantity",
```



```
String.valueOf(Integer.parseInt(quantity)));

if (status.toUpperCase( ).equals("ER")) {
    if (error.equals("1")) {
        as400response.put("ErrorMsg",
            "Account not authorized for item
availability.");
    }
    if (error.equals("2")) {
        as400response.put("ErrorMsg", "Item not found.");
    }
    if (error.equals("3")) {
        as400response.put("ErrorMsg", "Item is obsolete.");
        as400response.put("Replacement",
            (String) st.nextToken().trim( ));
    }
    if (error.equals("4")) {
        as400response.put("ErrorMsg",
            "Invalid quantity amount.");
    }
    if (error.equals("5")) {
        as400response.put("ErrorMsg",
            "Preference profile processing error.");
    }
    if (error.equals("6")) {
        as400response.put("ErrorMsg",
            "ATP processing error.");
    }
}

}

}

catch (Exception e) {
    System.out.println(e.toString( ));
}

return true;
}
```

这种方法首先构造一个传递给 AS/400 程序的参数字符串（通过管道符号来分隔），在这个程序中将解析字符串，提取请求数据，并且返回一个带有状态和错误码的字符串（通过管道符号来分隔）以及操作结果。如果没有错误，那么这个与 AS/400 交互的结果将被存储在另一个 HashMap 中，并用来构造 XML 响应文档。如果有错误，那么将写入到响应中。

## 6.4.2 构造 XML 响应

我总是乐于看到人们创建 XML 文档的许多方式。我通常告诉人们 XML 文档只是一个庞大的文本字符串。因此，使用 `StringBuffer` 来写出一个 XML 文档比构建一个文档对象模型 (Document Object Model, DOM) 或者使用专门的 XML 生成库要更加容易。

在这个项目中，我只是创建了一个 `StringBuffer` 对象，并且把遵循 Rosettanet 标准的 XML 文档中的每行文本都添加在这个对象中。虽然我省略了几行代码，但以下的代码还是可以告诉你如何来构造响应：

```
public String getResponseXML( ) {

    StringBuffer response = new StringBuffer( );
    response.append("<Pip3A2PriceAndAvailabilityResponse>").append("\n");
    response.append("    <ProductAvailability>").append("\n");
    response.append("
<ProductQuantity>").append(as400response.get("Quantity")).append("</ProductQuan
tity>").append("\n");
    response.append("    </ProductAvailability>").append("\n");
    response.append("    <ProductIdentification>").append("\n");
    response.append("        <PartnerProductIdentification>").append("\n");
    response.append("
<GlobalPartnerClassificationCode>Manufacturer</GlobalPartnerClassificationCode>
").append("\n");
    response.append("
<ProprietaryProductIdentifier>").append(requestValues.get("prod_ProductIdentifi
er")).append("</ProprietaryProductIdentifier>").append("\n");
    response.append("        </PartnerProductIdentification>").append("\n");
    response.append("    </ProductIdentification>").append("\n");
    response.append("
</ProductPriceAndAvailabilityLineItem>").append("\n");
    response.append("</Pip3A2PriceAndAvailabilityResponse>").append("\n");

    return response.toString( );
}
```

## 6.5 结束语

当我回首这段两年前编写的代码时，很自然地进行了自我反省并且考虑我是否可以用更好的方式来编写这段代码。虽然我可以编写一些不同的实现代码，但我认为自己仍然会按照相同的方式来设计它。这段代码经受了时间的考验，因为客户自己在添加新的销售商和新的请求类型时，很少需要借助像我这样的外部技术人员。

目前，作为生物信息部门的主管，当我向手下的人员讲授面向对象设计原则和 XML 解析技巧时，我通常会用这段代码来说明一些问题。我本应该在本章中写一些最近开发的代码，

但我认为在这段代码中说明了几个基本的原则，并且这些原则对于任何年轻的软件开发人员来说都是应重点理解的。

# 后记

Andy Oram

《Beautiful Code》介绍了人类在一个奋斗领域：计算机系统的开发领域中的创造性和灵活性。在每章中的漂亮代码都来自独特解决方案的发现，而这种发现来源于作者超越既定边界的远见卓识，并且识别出被多数人忽视的需求以及找出令人叹为观止的问题的解决方案。

大多数作者都面临着种种限制——包括物理环境，可用资源，或者特殊的需求定义——这些限制通常会使我们很难想象出解决方案。而其他一些作者则是在已经存在解决方案的领域中重新研究，并且提出新的观点以及更好地实现某个功能。

本书的所有作者都从他们的项目中获得了一些经验。不过在阅读完本书后，我们同样可以总结出一些更广泛的经验。

首先，在可靠和真实的规则能够真正应用之前，需要进行多次尝试。因为，人们在维护稳定性、可靠性以及其他软件工程要求的标准时经常会遇到重重困难。在这种情况下，我们通常没有必要抛弃支持这种承诺的原则。有时候，从另一个角度来思考问题或许能够揭示一种新的方向，从而使我们在满足需求的同时无需牺牲那些好的技术。

另一方面，在有些章节中强调了这条古老的原则：在打破原则之前，人们必须首先了解这个规则。有些作者在获得一种不同的解决方案之前积累了数十年的经验——而正是这些经验给了他们自信，从而以创造性的方式打破规则。

此外，书中的一些经验还提倡跨学科研究。许多作者都是新的领域中进行研究并在黑暗中不断探索。在这种情况下，全新的创造力和个人智慧将起到重要的作用。

最后，我们从书中学到的漂亮的解决方案并不会持续很长时间。在新的环境中总会要求新的解决方式。因此，如果阅读了本书并且认为，“无法在自己的任何一个项目上使用这些作者的解决方案”，那么也不要担心——这些作者在做下一个项目的时候，也会使用不同的解决方案。

我在这本书上全身心地工作了两个月，以帮助作者完善他们的主题和更好地表达他们的观点。阅读这些天才发明家的文章的确令人鼓舞甚至是令人情绪高涨的。它给了我尝试新鲜事物的冲动，我希望读者在阅读本书时也能有同样的感受。

## 作者简介

John Bentley 是美国 Avaya 实验室的一位计算机科学家。他的研究领域包括编程技术、算法设计以及软件工具与界面设计。他已编写了数本关于编程的书籍，还撰写了大量的文章，主题涉及从算法理论到软件工程的各个方向。他于 1974 年在斯坦福大学获得学士学位，并于 1974 年获得硕士学位以及于 1976 年在北卡罗来纳大学获得博士学位，随后在卡耐基-梅

隆大学任教 6 年，教授计算机科学。1982 年他加入贝尔实验室，并于 2001 年离开贝尔实验室并加入 Avaya 实验室。他曾是西点军校和普林斯顿大学的访问教授、曾经参与开发过软件工具、电话交换机、电话以及网络服务。

Tim Bray 于 1987-1989 年间在加拿大的安大略省滑铁卢大学负责牛津英语词典项目，1989 年与他人联合创建了 Open Text 公司，在 1995 年启动了最早的公共网页搜索引擎之一，在 1996 至 1999 年间与他人共同发明了 XML 1.0 并合作编写了《Namespaces in XML》规范，在 1999 年他创建了 Antarctica Systems 公司，并于 2002-2004 年被 Tim Berners-Lee 任命在 W3C 技术架构组中工作。目前，他在 Sun Microsystems 公司 Web Technologies 部门任主管，他有一个很受欢迎的博客，并且参与主持 IETF AtomPub 工作组。

Bryan Cantrill 是 Sun Microsystems 公司的一位杰出的工程师，在他的职业生涯中主要从事 Solaris 内核的开发。最近他与同事 Mike Shapiro 和 Adam Leventhal 一起设计并实现了 DTrace，这是一个用于产品系统动态控制的工具，获得了《华尔街日报》2006 年度的最高创新奖。

Douglas Crockford 毕业于公立学校。他是一位登记选民，拥有自己的汽车。他曾开发过办公自动化系统。他曾在 Atari 公司从事过游戏和音乐研究。他曾是 Lucasfilm 有限公司技术部门的主管，以及 Paramount 公司 New Media 部门的主管。他创建了 Electric Communities 公司并且担任 CEO。他还是 State 软件公司的创建者和 CTO，正是在这个公司中他发明了 JSON 数据格式。他现在是 Yahoo! 公司的一位架构师。

Rogério Atem de Carvalho 是巴西校园技术教育联合中心（Federal Center for Technological Education of Campos, CEFET Campos）的一位教师兼研究人员。他在奥地利的维也纳获得了 2006 年度 IFIP 杰出学术领导奖（Distinguished Academic Leadership Award），以表彰他在免费/开源企业资源计划（ERP）上所做的研究工作。他的研究领域还包括决策支持系统和软件工程。

Jeff Dean 于 1999 年加入 Google，目前是 Google 系统架构小组的成员。他在 Google 主要负责开发 Google 的网页抓取、索引、查询服务以及广告系统等，他对搜索质量实现了多次改进，并实现了 Google 分布式计算架构的多个部分。在加入 Google 之前，他工作于 DEC/Compaq 的 Western 实验室，主要从事软件分析工具、微处理器架构以及信息检索等方面的研究。他于 1996 年在华盛顿大学获得了博士学位，与 Craig Chambers 一起从事面向对象语言的编译器优化技术方面的研究。在毕业之前，他还在世界卫生组织的艾滋病全球规划署工作过。

Jack Dongarra 于 1972 年在芝加哥大学获得数学学士学位，并于 1973 年在伊利诺理工大学获得计算机科学硕士学位，又于 1980 年在新墨西哥大学获得应用数学博士学位。他在美国阿贡国家实验室（Argonne National Laboratory）一直工作到 1989 年，并成为了一名著名科学家。他现在被任命为田纳西大学计算机科学系的计算机科学杰出教授。他是美国橡树岭国家实验室（Oak Ridge National Laboratory，ORNL）计算机科学与数学部的杰出的研究人员，曼彻斯特大学计算机科学与数学学院的 Turing Fellow，美国莱斯大学计算机科学系的副教授。他的研究领域包括线性代数中的数值算法，并行计算，高级计算机架构的应用，程序设计方法学以及用于并行计算机的工具。他的研究工作包括开发、测试高质量的数学软件以及整理相关文档。

他在以下开源软件包和系统的设计及实现上做出了贡献: ISPACK, LINPACK, the BLAS, LAPACK, ScaLAPACK, Netlib, PVM, MPI, NetSolve, Top500, ATLAS, 和 PAPI。他公开发表了大约 200 篇文章、论文、报告以及技术备忘录,还参与编写了数本著作。他于 2004 年获得了 IEEE Sid Fernbach 奖,以表彰他在高性能计算机的应用中使用了创新的方法。他不仅是 AAAS, ACM 和 IEEE 的成员,还是美国工程院的院士。

R. Kent Dybvig 是印第安纳大学计算机科学系的一位教授。在印第安纳大学任教两年之后,他于 1987 年在北卡罗来纳大学获得了博士学位。他在设计和实现编程语言的研究上做出了重要的贡献,包括控制运算符、句法抽象、程序分析、编译器优化、寄存器分配、多线程以及自动存储管理等。在 1984 年,他创建了 Chez Scheme 软件并一直是主要的开发人员。Chez Scheme 的特点在于快速的编译时间、可靠性以及能够高效地运行内存需求巨大的复杂程序,它已经被用于构建企业集成、网页服务、虚拟现实、机器人药品抽检、电路设计以及其他的商业系统。它还可以用于各种层次的计算机教育以及许多其他领域中的研究。Dybvig 是《The Scheme Programming Language, Third Edition》(MIT Press 出版社)一书的作者,以及即将发布的“Revised<sup>6</sup> Report on Scheme”文档的编辑。

Michael Feathers 是 Object Mentor 公司的顾问。在过去七年间,他一直活跃于 Agile 社群,他的工作主要是与世界各地不同的团队合作,培训以及指导。在加入 Object Mentor 公司之前,Michael 设计过一种编程语言,并为这种语言写了一个编译器。他还设计了一个庞大的多平台类库以及用于控制的框架。Michael 开发了 CppUnit,也就是最初把 JUnit 移植到 C++;以及 FitCpp,也就是把 FIT 移植到 C++。在 2005 年,Michael 编写了《Working Effectively with Legacy Code》(Prentice Hall 出版社)一书。在与各个团队合作的间隙,他的大多数时间都花在研究大型代码库中的设计修改方式方面。

1995 年,Karl Fogel 和 Jim Blandy 一起创建了 Cyclic 软件公司,这是第一个提供商业 CVS 支持的公司。1997 年,Karl 增加了对 CVS 匿名只读存储仓库访问的支持,这样就可以更方便地访问开源项目中的开发代码。1999 年,他工作于 CollabNet 公司,主要从事管理 Subversion 的创建和开发工作,这是 CollabNet 公司和一群开源志愿者们从头开始编写的开源版本控制系统。2005 年,他编写了《Producing Open Source Software: How to Run a Successful Free Software Project》(O'Reilly 出版社;在<http://producingoss.com>上有联机版本)一书。2006 年,他在 Google 担任了短期的开源技术专家之后离开 Google 并成为了 Question-Copyright.org 网站的全职编辑。他目前仍然参与了多个开源项目,包括 Subversion 和 GNU Emacs。

Sanjay Ghemawat 是一位 Google Fellow,工作于 Google 的系统架构小组。他设计并实现了分布式的存储系统,文本索引系统,性能分析工具,一种数据表示语言,一个 RPC 系统,一个 malloc 函数实现以及许多其他的库。在加入 Google 之前,他是 DEC 系统研究中心的一位研究人员,主要从事系统性能分析和优化 Java 编译器的工作,他还实现了一个 Java 虚拟机。他于 1995 年在麻省理工大学获得博士学位,研究领域为面向对象数据库的实现。

Ashish Gulhati 是互联网隐私服务 Neomailbox 的首席开发员,以及 Cryptonite 的开发员,这是一个支持 OpenPGP 协议的安全网页邮件系统。他有着 15 年的商业软件开发经验,是印度最早的数字版权活动家之一和 F/OSS 程序员,他编写了大量的开源 Perl 模块,这些模块可以从 CPAN 上下载。在 1993~1994 年间,他在《PC Quest》和《DataQuest》等杂志上发表了大量文章,这是在印度主流计算机刊物中最早向读者介绍自由软件,GNU/ Linux,



Web 和 Internet 的文章，在这些文章发表多年以后，印度才拥有了商业的互联网访问，这些文章还构成了 PC Quest Linux Initiative 活动的重要组成部分，这个活动促使自 1995 年以来，在印度分发了一百万份 Linux 光盘。在获得了一组可穿戴的计算机后，他很快地成为了一个电子人。

Elliotte Rusty Harold 是新奥尔良人，他会定期返回新奥尔良去吃一大碗海鲜干波汤 (Gumbo)。不过，他目前住在布鲁克林附近的 Prospect Heights，和他生活在一起还有他的妻子 Beth，狗 Shayna，和两只猫 Charm（以夸克命名）和 Marjorie（以他的岳母命名）。他是纽约科技大学的一位副教授，主要讲授 Java、XML 以及面向对象编程。他的 Cafe au Lait 网站 (<http://www.cafeaulait.org>) 是互联网上最流行的独立 Java 网站之一；他的另一个网站 Cafe con Leche (<http://www.cafeconleche.org>) 则成为了最流行 XML 站点之一。他编写的书籍包括《Java I/O》，《Java Network Programming》和《XML in a Nutshell》（这三本书都由 O'Reilly 出版社出版），以及 XML Bible (Wiley 出版社)。他目前的研究领域包括用 Java 来处理 XML 的 XOM 库、Jaxen XPath 引擎以及 Amateur 媒体播放器。

Brian Hayes 为《American Scientist》杂志编写计算机专栏，他还拥有一个博客 <http://bit-player.org>。过去，他还为《Scientific American》、《Computer Language》、以及《The Sciences》等杂志编写过类似的专栏。他编写的《Infrastructure: A Field Guide to the Industrial Landscape》(Norton 出版社) 一书于 2005 年发行。

Simon Peyton Jones，硕士，于 1980 年毕业于剑桥大学三一学院。在工作两年后，他在伦敦大学学院担任了 7 年的讲师，然后在格拉斯哥大学担任了 9 年的教授，后来于 1998 年加入微软研究中心。他的研究领域包括函数式编程语言及其实现和应用。他领导了一系列的研究项目，主要研究用于单处理器机器和并行机的高质量函数式语言系统的设计和实现。他是函数式语言 Haskell 的主要设计者，此外他还是被广泛应用的 Glasgow Haskell 编译器 (GHC) 首席设计师。他还编写了两本关于函数式语言实现的教科书。

Jim Kent 是加利福尼亚大学圣克鲁兹分校基因信息小组 (Genome Bioinformatics Group) 的一位研究学家。Jim 从 1983 年起就开始编程。在职业生涯的前半段，他主要从事绘画和动画软件的开发，他开发了 Aegis Animator、Cyber Paint 以及 Autodesk Animator 等获奖软件。1996 年，由于厌倦了基于 Windows API 的开发工作，他决定在生物学上追求他的兴趣，并于 2002 年获得了博士学位。在研究生期间，他编写 GigAssembler——这个程序计算出了第一批人类基因组——比 Celera 公司发布的第一批基因组提前了一天，从而使得这批基因组成为免费的专利并且避免了其他的法律问题。Jim 发表了 40 余篇科学论文。他目前的研究工作主要是编写程序，数据库和网站以帮助科学家分析和了解基因组。

Brian Kernighan 于 1964 年在多伦多大学获得学士学位，并于 1969 年在普林斯顿大学获得电子工程博士学位。他在贝尔实验室的计算科学研究中心一直工作到 2000 年，目前就职于普林斯顿大学的计算机科学系。他编写了 8 本著作以及大量的技术论文，并拥有 4 项专利。他的研究领域包括编程语言、工具、为非专业用户设计易用的计算机操作界面等。他还致力于非技术读者的技术教育工作。

Adam Kolawa 是 Parasoft 公司的创建者之一和 CEO，这家公司是自动错误预防 (Automated Error Prevention, AEP) 解决方案的领先提供商。Kolawa 有着多年在各种软件开发流程中的经验，这使得他对高科技企业有着独特的视野，以及成功辨识技术潮流的



非凡能力。因此，他策划了几个成功商业软件产品的开发过程来满足在提高软件质量中不断增长的工业需求——经常在这种潮流被广泛接受之前。Kolawa 参与编写了《Bulletproofing Web Applications》(Hungry Minds 出版社)一书，他还撰写了 100 余篇评论和技术文章，发表在《The Wall Street Journal》、《CIO》、《Computerworld》、《Dr. Dobbs' Journal》以及《IEEE Computer》等期刊上。此外，他还撰写了大量关于物理学和并行处理方面的科学论文。他现在的签约媒体包括 CNN、CNBC、BBC 和 NPR。Kolawa 拥有加利福尼亚理工大学理论物理博士学位，并拥有 10 项专利发明。2001 年，Kolawa 获得了软件类别的 Los Angeles Ernst & Young's Entrepreneur of the Year 奖项。

Greg Kroah-Hartman 是目前 Linux 内核的维护人员，负责多个驱动程序子系统以及驱动程序内核、sysfs、kobject、kref 和 debugfs 等代码。他还为启动 linux-hotplug 和 udev 等项目提供了帮助，是内核稳定维护团队中的重要人员。他编写了《Linux Kernel in a Nutshell》(O'Reilly 出版社)，并参与编写了《Linux Device Drivers, Third Edition》(O'Reilly 出版社)。

Andrew Kuchling 有着 11 年的软件工程师经验，他是 Python 开发群体中的长期成员。他的一些与 Python 相关的工作包括编写和维护数个标准的库模块，编写一系列的“What's new in Python 2.x”文章以及其他一些文档，策划了 2006 年和 2007 年的 PyCon 会议，并是 Python 软件基金会的主管。Andrew 于 1995 年毕业于麦吉尔大学并获得计算机科学学士学位。他的个人网页是<http://www.amk.ca>。

Piotr Luszczek 毕业于波兰克拉科夫矿业与冶金大学，并获得硕士学位，他的研究领域是并行的核外 (out-of-core) 库。他将稠密矩阵计算核应用于稀疏矩阵直接求解算法和迭代数值线性几何算法中的创新研究使他获得了博士学位。他把这种思想用来开发使用核外技术容错库。目前，他是田纳西大学诺克斯维尔分校的一位研究教授。他的研究工作包括大型超级计算机安装的标准化评价。他开发了一个自适应的软件库，能够自动选择最优的算法来有效地利用现有硬件以及有选择地处理输入数据。他还感兴趣于高性能编程语言的设计和实现。

Ronald Mak 是高级计算机科学研究所 (Research Institute for Advanced Computer Science) 的一位资深科学家，在 NASA Ames 研究中心工作时，他是协同信息系统 (Collaborative Information Portal, CIP) 的架构师和首席开发人员。在漫步者登录火星之后，他分别在 JPL 和 Ames 对探测任务提供支持。然后，他获得了加利福尼亚大学圣克鲁兹分校的学术任命，并且他再次与 NASA 签约，这次的工作是设计帮助宇航员返回月球的企业软件。Ron 是 Willard & Lowe Systems (<http://www.willardlowe.com>) 公司的创建人之一和 CTO，这是一个针对企业信息管理系统的咨询公司。他编写了数本关于计算机软件的书籍，他在斯坦福大学分别获得了数学科学学位和计算机科学学位。

Yukihiro "Matz" Matsumoto 是一位程序员，他是一位日本籍的开源倡导者，他发明了最近非常流行的 Ruby 语言。他从 1993 年开始研发 Ruby，这和 Java 语言一样久远。现在他工作于日本 Network Applied Communication Laboratory (NaCl，网址为 [netlab.jp](http://netlab.jp)) 公司，该公司从 1997 年起开始赞助 Ruby 的开发。因为他的真实姓名太长而难以记住，并且对于非日本的演讲者来说难以发音，因此在网上他使用了昵称 Matz。

Arun Mehta是一位电子工程师和计算机科学家，他曾在印度、美国和德国进行过研究和教学工作。他是印度早期计算机活动家，他努力实现了一些方便消费者(consumer-friendly)的政策，以帮助把现代通信延伸到偏远地区和贫困地区。他目前的研究领域包括农村无限通信以及帮助残疾用户的技术。他是印度哈里亚纳邦Radaur地区JMIT大学计算机工程系的教授和主任。他的网址包括<http://india-gii.org>, <http://radiophony.com>和<http://holisticit.com>。

Rafael Manhaes Monnerat 是 CEFET CAMPOS 的一位 IT 分析家，以及 Nexedi SARL 的海外顾问。他的研究领域包括免费/开源系统、ERP 以及最新的编程语言。

Travis E. Oliphant 于 1995 年在美国杨百翰大学获得电子与计算机工程学士学位和数学学士学位，并于 1996 年在本校获得电子与计算机工程硕士学位。他于 2001 年在明尼苏达罗切斯特的梅奥研究生院获得了生物医学工程博士学位。他是 Python 语言中科学计算库 SciPy 和 NumPy 的主要编写者。他的研究领域包括显微阻抗成像，异构领域中的 MRI 重构以及生物医学逆问题。他目前是杨百翰大学电子与计算机工程的副教授。

Andy Oram是O'Reilly Media的编辑。他从 1992 年开始就在这家公司工作，Andy目前主要关注自由软件和开源技术。他在O'Reilly的工作成果包括第一批Linux系列丛书以及 2001 年的P2P系列丛书。他的编程技术和系统管理技术大多都是自学的。Andy还是Computer Professionals for Social Responsibility协会的成员并且经常在O'Reilly Network(<http://oreillynnet.com>)和其他一些刊物上撰写文章，这些文章的主题包括互联网上的政策问题，以及影响技术创新的潮流及其对社会的影响。他的网址为<http://www.praxagora.com/andyo>。

William R. Otte 是田纳西范德堡大学电子工程与计算机系 (EECS) 的一位博士研究生。他的研究领域是分布式实时嵌入 (DRE) 系统的中间件，目前从事 CORBA 组件的部署和配置引擎 (DAnCE) 开发工作。这个工作主要研究运行时规划技术，基于组件的应用程序的适应性，以及对应用程序服务质量和容错需求的规范与实施。在攻读研究生之前，William 于 2005 年在范德堡大学计算机系毕业并获得学士学位，之后在软件集成系统学院 (ISIS) 工作了一年。

Andrew Patzer 是威斯康星大学医学院生物信息系的主管。过去 15 年 Andrew 是一位软件开发人员并且编写了许多文章和书籍，包括《Professional Java Server Programming》(Peer Information 公司) 和《JSP Examples and Best Practices》(Apress 出版社)。Andrew 目前的研究领域为生物信息领域，利用像 Groovy 这样的动态语言来发掘大量有效的生物数据并帮助科学研究人员进行分析。

Charles Petzold是一位自由作家，主要研究领域为Windows应用程序编程。他是《Programming Windows》(Microsoft Press出版社)的作者，1988 年至 1999 年之间共出版了五版，教育了整整一代程序员的Windows API编程技术。他最新的书籍包括《Applications = Code + Markup: A Guide to the Microsoft Windows Presentation Foundation》(Microsoft Press出版社)，以及《Code: The Hidden Language of Computer Hardware and Software》(Microsoft Press出版社)，在这本书中他对数字技术进行了独特的研究。他的网址是<http://www.charlespetzold.com>。

T. V. Raman 的研究领域包括网页技术和听觉用户界面。在 20 世纪 90 年代初，在他的博士论文中介绍了音频格式的概念，叫作 AsTeR: **A**udio **S**ystem **F**or **T**echnical **R**eadings（技术读物语音系统），这是一个为技术文档生成高质量听觉表示的系统。Emacspeak 则将这些思想应用到更广泛的计算机用户界面领域。Raman 现在是 Google 的一位研究人员，主要研究 Web 应用程序。

Alberto Savoia 是 Agitar 软件公司的创建人之一和 CTO。在创建 Agitar 之前，他是 Google 的高级工程主管；在这之前，他还是 Sun Microsystems 实验室软件研究中心的主管。Alberto 的主要研究领域是软件开发技术——尤其是那些帮助程序员在设计和开发阶段进行测试和代码验证的工具和技术。

Douglas C. Schmidt 是田纳西范德堡大学电子工程与计算机（EECS）系的一位教授，计算机科学与工程系的副主任，以及软件集成系统学院（ISIS）的高级研究人员。他是分布式计算模式和中间件框架方面的专家，并且已经发表了超过 350 篇的技术论文和 9 本书籍，内容涉及的主题很广，包括高性能通信软件系统，高速网络协议并行处理，实时分布式对象计算，并发与分布式系统的面向对象模式，以及模型驱动的开发工具。在他的学术研究之外，Dr. Schmidt 还是 PrismTechnologies 公司的 CTO，并且在领导开发应用广泛开源的中间件平台上有着 15 年的经验，在这些平台上包含了丰富的组件以及实现高性能分布式系统中核心模式的领域特定语言。Dr. Schmidt 于 1994 年于加利福尼亚大学欧文分校获得计算机科学博士学位。

Christopher Seiwald 编写了 Perforce（一种软件配置管理系统）、Jam（一种构建工具）和“漂亮代码的七个要素”（本书的第 32 章，变动的代码，正是从这篇文章中提取出了有价值的思想）。在创建 Perforce 之前，他在 Ingres 公司管理网络开发小组，他花了数年时间来使得异步网络代码看上去很漂亮。现在他是 Perforce 软件公司的 CEO，并且仍然从事编码工作。

Diomidis Spinellis 是希腊雅典经济与商业大学管理科学与技术系的副教授。他的研究领域包括软件工程工具，编程语言和计算机安全。他在伦敦帝国理工大学获得了软件工程硕士学位和计算机科学博士学位。他发表了超过 100 篇的技术论文，所涉及的领域包括软件工程，信息安全以及普适计算。他还编写了两本开源方面的书籍：《Code Reading》（获得 2004 年度 Software Development Productivity 奖）和《Code Quality》（这两本书都由 Addison-Wesley 出版社出版）。他是 IEEE Software 编辑委员会的成员，主编“Tools of the Trade”专栏。Diomidis 是一位 FreeBSD 提交者（**Committer**），并且编写了许多开源软件包、软件库以及工具。

Lincoln Stein 是一位硕士/博士，他的研究领域为生物信息数据的集成与虚拟化。在从哈佛大学医学院毕业后，他在麻省理工大学 Whitehead 基因研究所工作，开发用于老鼠和人类的基因图谱数据库。他在冷泉港实验室开发了各种基因数据库，包括 WormBase，线虫基因数据库；Gramene，用于水稻和其他单子叶植物的比较基因映射数据库；国际 Hap-Map 项目数据库；以及人类基因基础数据库 Reactome。Lincoln 还编写了《books How to Set Up and Maintain a Web Site》（Addison-Wesley 出版社）、《Network Programming in Perl》（Addison-Wesley 出版社）、《Official Guide to Programming with CGI.pm》（Wiley 出版社）以及《Writing Apache Modules with Perl and C》（O'Reilly 出版社）等书籍。

Nevin Thompson 把 Yukihiro Matsumoto 编写的第 29 章内容, 把代码当作文章, 从日文翻译到英文。他的客户包括日本最大的电视网络, 以及 Technorati Japan 公司和 Creative Commons 组织。

Henry S. Warren, Jr. 在 IBM 工作了 45 年, 他历经了从 IBM 704 到 PowerPC 的发展过程。他参与过多个军方指挥与控制系统的开发工作, 在纽约大学 Jack Schwartz 教授指导下从事 SETL 项目。从 1973 年起, 他在 IBM 研究部门工作, 主要方向为编译器和计算机架构。Hank 目前正在参与 Blue Gene Petaflop 超级计算机项目。他在纽约大学克朗数学研究所获得了计算机博士学位。他是《Hacker's Delight》(Addison-Wesley 出版社)一书的作者。

Laura Wingerd 多年 Sybase 和 Ingres 的数据库产品开发工作形成了她早期对软件配置管理的观点。她在 Perforce 软件公司创建之初就加盟了这家公司, 并且从她给 Perforce 客户的建议中获得了大量的 SCM 经验。她编写了《Practical Perforce》(O'Reilly 出版社)一书以及许多与 SCM 相关的白皮书。她在 Google 的技术演讲 The Flow of Change 中首次露面。Laura 现在是 Perforce 软件公司产品技术部的副主管, 主要负责推动合理的 SCM 流程以及研究新的并且更好的 Perforce 使用方式。

Greg Wilson 在爱丁堡大学获得了计算机科学博士学位, 他的研究领域包括高性能科学计算, 数据虚拟化以及计算机安全。他现在是多伦多大学计算机科学系的一位副教授, 并且是《Dr. Dobbs' Journal》杂志的特约编辑。

Andreas Zeller 于 1991 年毕业于德国达姆斯塔特理工大学, 并于 1997 年在不伦瑞克理工大学获得计算机科学博士学位。2001 年以来, 他一直在德国萨尔兰登大学的计算机科学系担任教授。Zeller 主要研究大型程序以及它们的发展历史, 他开发了大量的方法来分析在开源软件以及 IBM、Microsoft、SAP 以及其他公司的商业软件中失败的原因。他编写的《Why Programs Fail: A Guide to Systematic Debugging》(Morgan Kaufmann 出版社)获得了《Software Development Magazine》杂志 2006 年度的 Productivity 大奖。