/THEORY/IN/PRACTICE

编写可读代码的艺术

The Art of Readable Code

Dustin Boswell & Trevor Foucher 著

尹哲 郑秀雯 译

O'REILLY®

机械工业出版社
China Machine Press

HZ BOOKS
华章科技

编写可读代码的艺术

“软件开发的一个重要部分是要意识到你的代码以后将如何影响查看这些代码的人。两位作者高屋建瓴，带你领略这一挑战的各个方面，并且使用有指导意义的例子来解释细节。”

——Michael Hunger，软件开发人员

细节决定成败，思路清晰、言简意赅的代码让程序员一目了然；而格式凌乱、拖沓冗长的代码让程序员一头雾水。除了可以正确运行以外，优秀的代码必须具备良好的可读性，编写的代码要使其他人能在最短的时间内理解才行。本书旨在强调代码对人的友好性和可读性。

本书关注编码的细节，总结了很多提高代码可读性的小技巧，看似都微不足道，但是对于整个软件系统的开发而言，它们与宏观的架构决策、设计思想、指导原则同样重要。编码不仅仅只是一种技术，也是一门艺术，编写可读性高的代码尤其如此。如果你要成为一位优秀的程序员，要想开发出高质量的软件系统，必须从细处着手，做到内外兼修，本书将为你提供有效的指导。

主要内容：

- 简化命名、注释和格式的方法，使每行代码都言简意赅。
- 梳理程序中的循环、逻辑和变量来减小复杂度并理清思路。
- 在函数级别解决问题，例如重新组织代码块，使其一次只做一件事。
- 编写有效的测试代码，使其全面而简洁，同时可读性更高。

Dustin Boswell毕业于加州理工大学，资深软件工程师，在Google就职多年，负责Web爬虫和程序设计相关的工作。他专注于前端、后端，服务器架构、机器学习、大数据、系统和网站等技术领域的研究和实践，经验十分丰富。他现在是MyLikes的软件工程师。

Trevor Foucher资深软件工程师和技术经理，先后在Microsoft和Google工作了数十年，在Microsoft担任软件工程师、技术经理以及安全产品技术主管，在Google从事广告应用开发和搜索基础结构研发相关的工作。

O'REILLY®
oreilly.com.cn

客服热线：(010) 88378991，88361066
购书热线：(010) 68326294，88379649，68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com
华章网站：*http://www.hzbook.com*
网上购书：*www.china-pub.com*

O'Reilly Media, Inc.授权机械工业出版社出版

ISBN 978-7-111-38544-8
9 787111 385448

定价：59.00元

编写可读代码的艺术

Dustin Boswell & Trevor Foucher 著

尹哲 郑秀雯 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权机械工业出版社出版

机械工业出版社

图书在版编目（CIP）数据

编写可读代码的艺术/（美）鲍斯维尔（Boswell, D.），富歇（Foucher, T.）著；尹哲，郑秀雯译. －北京：机械工业出版社，2012.7
（O'Reilly精品图书系列）
书名原文：The Art of Readable Code
ISBN 978-7-111-38544-8

I. 编… II. ①鲍… ②富… ③尹… ④郑… III. 代码－程序设计 IV. TP311.11
中国版本图书馆CIP数据核字（2012）第109081号

北京市版权局著作权合同登记
图字：01-2012-1275号

书　　名/　编写可读代码的艺术
书　　号/　ISBN 978-7-111-38544-8
责任编辑/　谢晓芳
封面设计/　Susan Thompson，张健
出版发行/　机械工业出版社
地　　址/　北京市西城区百万庄大街22号（邮政编码100037）
印　　刷/　北京京北印刷有限公司
开　　本/　178毫米×233毫米　16开本　12.25印张
版　　次/　2012年7月第1版　2012年7月第1次印刷
定　　价/　59.00元（册）

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

O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

推荐序

这是一本关注编码细节的书。或许你会认为本书所讲皆为小道，诸如方法命名、变量定义、语句组织、任务分解等内容，俱是细枝末节，微不足道。然而，对于一个整体的软件系统而言，既需要宏观的架构决策、设计与指导原则，也必须重视微观的代码细节。正如作文，提纲主旨是文章的根与枝，但一词一句，也需精雕细作，才能立起文章的精气神。所谓“细节决定成败”，在软件历史中，有许多影响深远的重大失败，其根由往往是编码细节出现了疏漏。

我一直坚持“代码即架构”的观点，正如小说需要角色来说话一般，软件系统的质量好坏，归根结底还是需要代码来告知。代码的优劣不仅直接决定了软件的质量，还将直接影响软件成本。Yourdon和Constantine在其著作《Structured Design》中写道：软件成本由开发成本与维护成本组成，而往往维护成本要远高于开发成本。这其中耗费的主要成本就是由于理解代码和修改代码造成的。正如本书的书名所表示的含义，好的代码常常是可阅读的，要做到这一点，则近似于一种艺术之美了。

与本书相似的一本书是Robert C. Martin的《Clean Code》。该书在业界已经得到了广泛赞誉。如果你还在为写出“丑陋”的代码而烦恼，必须阅读该书。它带给你的冲击，好似《阿凡达》那无与伦比的3D电影带给你感官上的震撼。在某种程度上，本书几乎可以与《Clean Code》比肩。或许本书在深度上与《Clean Code》相比还有所不及，但在内容广度上，却远远超过了《Clean Code》。因为它关注编码本身，所以并不局限于某一种语言，而是列举了大量C++、Python、JavaScript和Java代码，涵盖了主流的静态语言和动态语言。这就使得本书的内容具有更强的普适性。

本书给出了许多改善编码质量的技巧，尤其它结合了大量真实案例，给出了具体的代码片段，并从正反两面对案例进行分析，这就使得作者的讲解不再流于空洞，既让人信服，又有助于读者理解。例如，在讲解命名如何表达意图时，作者给出了Google代码中的一个反面教材。在Google的一段代码中定义了一个宏，用于禁止“邪恶”的构造函数：

```
class ClassName {
 private:
  DISALLOW_EVIL_CONSTRUCTORS(ClassName);

 public:
  ...
};
```

宏的定义如下：

```
#define DISALLOW_EVIL_CONSTRUCTORS(ClassName) \
  ClassName(const ClassName&); \
  void operator=(const ClassName&);
```

这个宏禁止了构造函数和Copy构造函数（即“=”操作）。然而从宏的名称来看，这个含义是不明确的，会让读者认为仅仅禁用了构造函数。只需要改个名字，意图就可以变得更加清晰：

```
#define DISALLOW_COPY_AND_ASSIGN(ClassName) [...]
```

书中各章的案例非常翔实，并且总是先给出糟糕的版本，逐步分析推导，最后给出好的实现，作为直观鲜明的对照。为避免读者陷入相对独立而散乱的小案例中，作者又另辟一章内容，讲解了一个完整的案例Minute/Hour Counter。首先从问题域的提出开始，分析了接口的设计与实现，深入剖析了实现方法的命名乃至注释，抽丝剥茧，条分缕析。作者的分析显得好整以暇，有条不紊，先后给出了3个解决方案，渐进地对编码实现进行了改善，使之在性能、灵活性上都有了很好的改观，类的职责更为清晰，代码结构简洁而又易于理解。最后，作者还给出了3个解决方案的比较，从代码行数、时间复杂度、内存消耗和准确率4个因素出发，全面权衡了各个解决方案的优劣，以此来印证作者在本书中一直推崇的编码技巧。

本书作者并不满足于通过文字和代码来彰显这些技巧的力量，书中附带的漫画插图起到了很好的辅助作用。如果将本书比作一盘精美的佳肴，这些漫画就起到了调料的作用。对于技术书籍来说，这种图文并茂的方式实不多见，它为全书增添了亮色。我们赏阅漫画时的会意一笑，心底其实充满了如遇知音的喜悦。

阅读本书时，那种代码从丑陋到美丽的蜕变总是让人振奋；但是，我们不能满足于结果的获得，而应该享受这个过程。我的建议是，在阅读时，多思考作者给出的反面教材，不要急于了解结果，而应掩卷遐思，分析这段代码的问题，并结合自身经验与能力给出自己的方案。然后再比较作者的方案，两相印证，辨别两个方案各自的优劣之处。最后再仔细阅读作者的分析过程，如此才能更好地理解本书，提升自己的编码技能。

全书200页左右，与那些瀚如烟海的高文大册相比，本书显得轻而薄，但它胜在精专。作者没有囊括所有编码技巧的野心，更没有卖弄地展现自己的设计技巧和博识广学。它的专注可能会因此失去一大部分读者群，但这样的书才是我们程序员真正需要的。希望你能喜欢它！

张逸

ThoughtWorks高级咨询师

译者序

在做IT的公司里，尤其是软件开发部门，一般不会要求工程师衣着正式。在我工作过的一些环境相对宽松的公司里，很多程序员的衣着连得体都算不上（搞笑的T恤、短裤、拖鞋或者干脆不穿鞋）。我想，我本人也在这个行列里面。虽然我现在改行做软件开发方面的咨询工作，但还是改不了这副德性。衣着体面的其中一个积极方面是它体现了对周围人的尊重，以及对所从事工作的尊重。比如，那些研究市场的人要表现出对客户的尊重。而大多数程序员基本上每天主要的工作就是和其他程序员打交道。那么这说明程序员之间就不用互相尊重吗？而且也不用尊重自己的工作吗？

程序员之间的互相尊重体现在他所写的代码中。他们对工作的尊重也体现在那里。

在《Clean Code》一书中Bob大叔认为在代码阅读过程中人们说脏话的频率是衡量代码质量的唯一标准。这也是同样的道理。

这样，代码最重要的读者就不再是编译器、解释器或者电脑了，而是人。写出的代码能让人快速理解、轻松维护、容易扩展的程序员才是专业的程序员。

当然，为了达到这些目的，仅有编写程序的礼节是不够的，还需要很多相关的知识。这些知识既不属于编程技巧，也不属于算法设计，并且和单元测试或者测试驱动开发这些话题也相对独立。这些知识往往只能在公司无人问津的编程规范中才有所提及。这是我所见的仅把代码可读性作为主题的一本书，而且这本书写得很有趣！

既然是“艺术”，难免会有观点上的多样性。译者本身作为程序员观点更加“极端”一些。然而两位作者见多识广，轻易不会给出极端的建议，如“函数必须要小于10行”或者“注释不可以用于解释代码在做什么而只能解释为什么这样做”等语句很少出现在本书中。相反，作者给出目标以及判断的标准。

翻译书是件费时费力的事情，好在本书恰好涉及我感兴趣的话题。但翻译本书有一点点自相矛盾的地方，因为书中相当的篇幅是在讲如何写出易读的英语。当然这里的“英语”大多数的时候只是指“自然语言”，对于中文同样适用。但鉴于大多数编程语言都是基于英语的（至少到目前为止），而且要求很多程序员用英语来注释，在这种情况下努力学好英语也是必要的。

感谢机械工业出版社的各位编辑帮助我接触和完成这本书的翻译。这本译作基本上可以说是在高铁和飞机上完成的（我此时正在新加坡飞往香港的飞机上）。因此家庭的支持是非常重要的。尤其是我的妻子郑秀雯（是的，新加坡的海关人员也对她的名字感兴趣），她是全书的审校者。还有我“上有的老人”和“下有的小孩”，他们给予我帮助和关怀以及不断前进的动力。

尹哲

目录

前言

我们曾经在非常成功的软件公司中和出色的工程师一起工作，然而我们所遇到的代码仍有很大的改进空间。实际上，我们曾见到一些很难看的代码，你可能也见过。

但是当我们看到写得很漂亮的代码时，会很受启发。好代码会很明确告诉你它在做什么。使用它会很有趣，并且会鼓励你把自己的代码写得更好。

本书旨在帮助你把代码写得更好。当我们说“代码”时，指的就是你在编辑器里面要写的一行一行的代码。我们不会讨论项目的整体架构，或者所选择的设计模式。当然那些很重要，但我们的经验是程序员的日常工作的大部分时间都花在一些“基本”的事情上，像是给变量命名、写循环以及在函数级别解决问题。并且这其中很大的一部分是阅读和编辑已有的代码。我们希望本书对你每天的编程工作有很多帮助，并且希望你把本书推荐给你团队中的每个人。

本书内容安排

这是一本关于如何编写具有高可读性代码的书。本书的关键思想是**代码应该写得容易理解**。确切地说，使别人用最短的时间理解你的代码。

本书解释了这种思想，并且用不同语言的大量例子来讲解，包括C++、Python、JavaScript和Java。我们避免使用某种高级的语言特性，所以即使你不是对所有的语言都了解，也能很容易看懂。（以我们的经验，反正可读性的大部分概念都是和语言不相关的。）

每一章都会深入编程的某个方面来讨论如何使代码更容易理解。本书分成四部分：

表面层次上的改进

命名、注释以及审美——可以用于代码库每一行的小提示。

简化循环和逻辑

在程序中定义循环、逻辑和变量，从而使得代码更容易理解。

重新组织你的代码

在更高层次上组织大的代码块以及在功能层次上解决问题的方法。

精选话题

把"易于理解"的思想应用于测试以及大数据结构代码的例子。

如何阅读本书

我们希望本书读起来愉快而又轻松。我们希望大部分读者在一两周之内读完全书。

章节是按照"难度"来排序的：基本的话题在前面，更高级的话题在后面。然而，每章都是独立的。因此如果你想跳着读也可以。

代码示例的使用

本书旨在帮助你完成你的工作。一般来说，可以在程序和文档中使用本书的代码。如果你复制了代码的关键部分，那么你就需要联系我们获得许可。例如，利用本书的几段代码编写程序是不需要许可的。售卖或出版O'Reilly书中示例的D-ROM需要我们的许可。引用本书回答问题以及引用示例代码不需要我们的许可。将本书的大量示例代码用于你的产品文档中需要许可。

如果你在参考文献中提到我们，我们会非常感激，但并不强求。参考文献通常包括标题、作者、出版社和ISBN。例如："《The Art of Readable Code》by Dustin Boswell, and Trevor Foucher.©2012 Dustin Boswell, and Trevor Foucher，978-0-596-80229-5。"

如果你认为对代码示例的使用已经超出以上的许可范围，我们很欢迎你通过*permissions@oreilly.com*联系我们。

联系我们

有关本书的任何建议和疑问，可以通过下列方式与我们取得联系：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）
奥莱利技术咨询（北京）有限公司

我们会在本书的网页中列出勘误表、示例和其他信息。可以通过*http://oreilly.com/product/9780596802301.do*访问该页面。

要评论或询问本书的技术问题，请发送邮件到：

bookquestions@oreilly.com

有关我们的书籍、会议、资源中心以及O'Reilly网络，可以访问我们的网站：

http://www.oreilly.com

http://www.oreilly.com.cn

在Facebook上联系我们：*http://facebook.com/oreilly*

在Twitter上联系我们：*http://twitter.com/oreillymedia*

在You Tube上联系我们：*http://youtube.com/oreillymedia*

致谢

我们要感谢那些花时间审阅全书书稿的同事，包括Alan Davidson、Josh Ehrlich、Rob Konigsberg、Archie Russell、Gabe W.，以及Asaph Zemach。如果书里有任何错误都是他们的过失（开玩笑）。

我们感激那些对书中不同部分的草稿给了具体反馈的很多审阅者，包括Michael Hunger、George Heinenman以及Chuck Hudson。

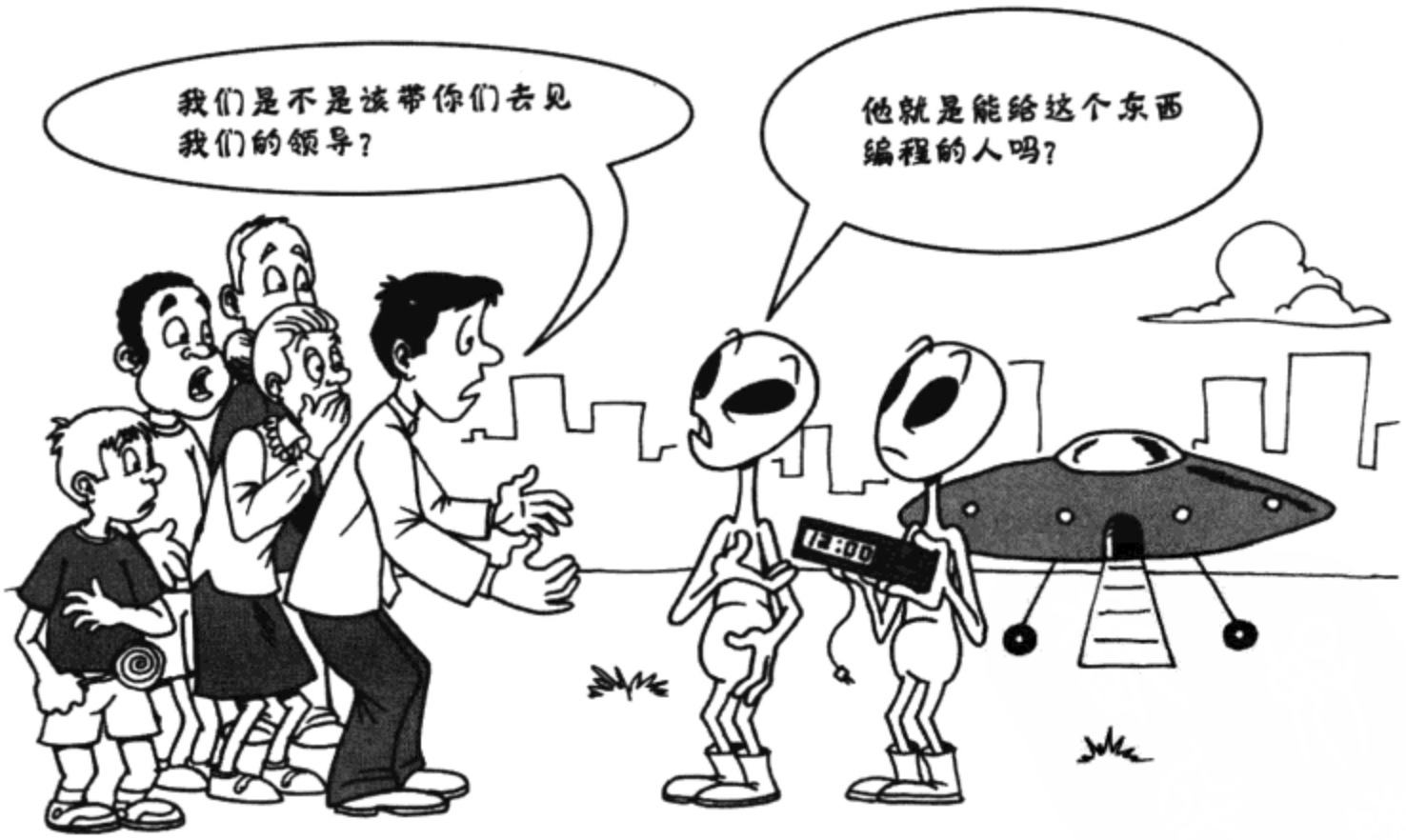
我们还从下面这些人那里得到了大量的想法和反馈：John Blackburn、Tim Dasilva、Dennis Geels、Steve Gerding、Chris Harris、Josh Hyman、Joel Ingram、Erik Mavrinac、Greg Miller、Anatole Paine和Nick White。

感谢O'Reilly团队无限的耐心和支持，他们是Mary Treseler（编辑）、Teresa Elsey（产品编辑）、Nancy Kotary（文字编辑）、Rob Romano（插图画家）、Jessica Hosman（工具）以及Abby Fox（工具）。还有我们的漫画家Dave Allred，他把我们疯狂的卡通想法展现了出来。

最后，我们想感谢Melissa和Suzanne，他们一直鼓励我们，并给我们创建条件来滔滔不绝地谈论编程话题。

第1章

代码应当易于理解



在过去的五年里，我们收集了上百个“坏代码”的例子（其中很大一部分是我们自己写的），并且分析是什么原因使它们变坏，使用什么样的原则和技术可以让它们变好。我们发现所有的原则都源自同一个主题思想。

关键思想

代码应当易于理解

我们相信这是当你考虑要如何写代码时可以使用的最重要的指导原则。贯穿本书，我们会展示如何把这条原则应用于你每天编码工作的各个不同方面。但在开始之前，我们会详细地介绍这条原则并证明它为什么这么重要。

是什么让代码变得“更好”

大多数程序员（包括两位作者）依靠直觉和灵感来决定如何编程。我们都知道这样的代码：

```
for (Node* node = list->head; node != NULL; node = node->next)
    Print(node->data);
```

比下面的代码好：

```
Node* node = list->head;
if (node == NULL) return;

while (node->next != NULL) {
    Print(node->data);
    node = node->next;
}
if (node != NULL) Print(node->data);
```

（尽管两个例子的行为完全相同。）

但很多时候这个选择会更艰难。例如，这段代码：

```
return exponent >= 0 ? mantissa * (1 << exponent) : mantissa / (1 << -exponent);
```

它比下面这段要好些还是差些？

```
if (exponent >= 0) {
    return mantissa * (1 << exponent);
} else {
    return mantissa / (1 << -exponent);
}
```

第一个版本更紧凑，但第二个版本更直白。哪个标准更重要呢？一般情况下，在写代码时你如何来选择？

可读性基本定理

在对很多这样的例子进行研究后，我们总结出，有一种对可读性的度量比其他任何的度量都要重要。因为它是如此重要，我们把它叫做“可读性基本定理”。

关键思想

代码的写法应当使别人理解它所需的时间最小化。

这是什么意思？其实很直接，如果你叫一个普通的同事过来，测算一下他通读你的代码并理解它所需的时间，这个“理解代码时间”就是你要最小化的理论度量。

并且当我们说“理解”时，我们对这个词有个很高的标准。如果有人真的完全理解了你的代码，他就应该能改动它、找出缺陷并且明白它是如何与你代码的其他部分交互的。

现在，你可能会想：“谁会关心是不是有人能理解它？我是唯一使用这段代码的人！”就算你从事只有一个人的项目，这个目标也是值得的。那个“其他人”可能就是6个月的你自己，那时你自己的代码看上去已经很陌生了。而且你永远也不会知道——说不定别人会加入你的项目，或者你“丢弃的代码”会在其他项目里重用。

总是越小越好吗

一般来讲，你解决问题所用的代码越少就越好（参见第13章）。很可能理解2000行代码写成的类所需的时间比5000行的类要短。

但少的代码并不总是更好！很多时候，像下面这样的一行表达式：

```
assert((!(bucket = FindBucket(key))) || !bucket->IsOccupied());
```

理解起来要比两行代码花更多时间：

```
bucket = FindBucket(key);
if (bucket != NULL) assert(!bucket->IsOccupied());
```

类似地，一条注释可以让你更快地理解代码，尽管它给代码增加了长度：

```
// Fast version of "hash = (65599 * hash) + c"
hash = (hash << 6) + (hash << 16) - hash + c;
```

因此尽管减少代码行数是一个好目标，但把理解代码所需的时间最小化是一个更好的目标。

理解代码所需的时间是否与其他目标有冲突

你可能在想："那么其他约束呢？像是使代码更有效率，或者有好的架构，或者容易测试等？这些不会在有些时候与使代码容易理解这个目标冲突吗？"

我们发现这些其他目标根本就不会互相影响。就算是在需要高度优化代码的领域，还是有办法能让代码同时可读性更高。并且让你的代码容易理解往往会把它引向好的架构且容易测试。

本书的余下部分将讨论如何把"易读"这条原则应用在不同的场景中。但是请记住，当你犹豫不决时，可读性基本定理总是先于本书中任何其他条例或原则。而且，有些程序员对于任何没有完美地分解的代码都不自觉地想要修正它。这时很重要的是要停下来并且想一下："这段代码容易理解吗？"如果容易，可能转而关注其他代码是没有问题的。

最难的部分

是的，要经常地想一想其他人是不是会觉得你的代码容易理解，这需要额外的时间。这样做就需要你打开大脑中从前在编码时可能没有打开的那部分功能。

但如果你接受了这个目标（像我们一样），我们可以肯定你会成为一个更好的程序员，会产生更少的缺陷，从工作中获得更多的自豪，并且编写出你周围人都爱用的代码。那么让我们开始吧！

第一部分

表面层次的改进

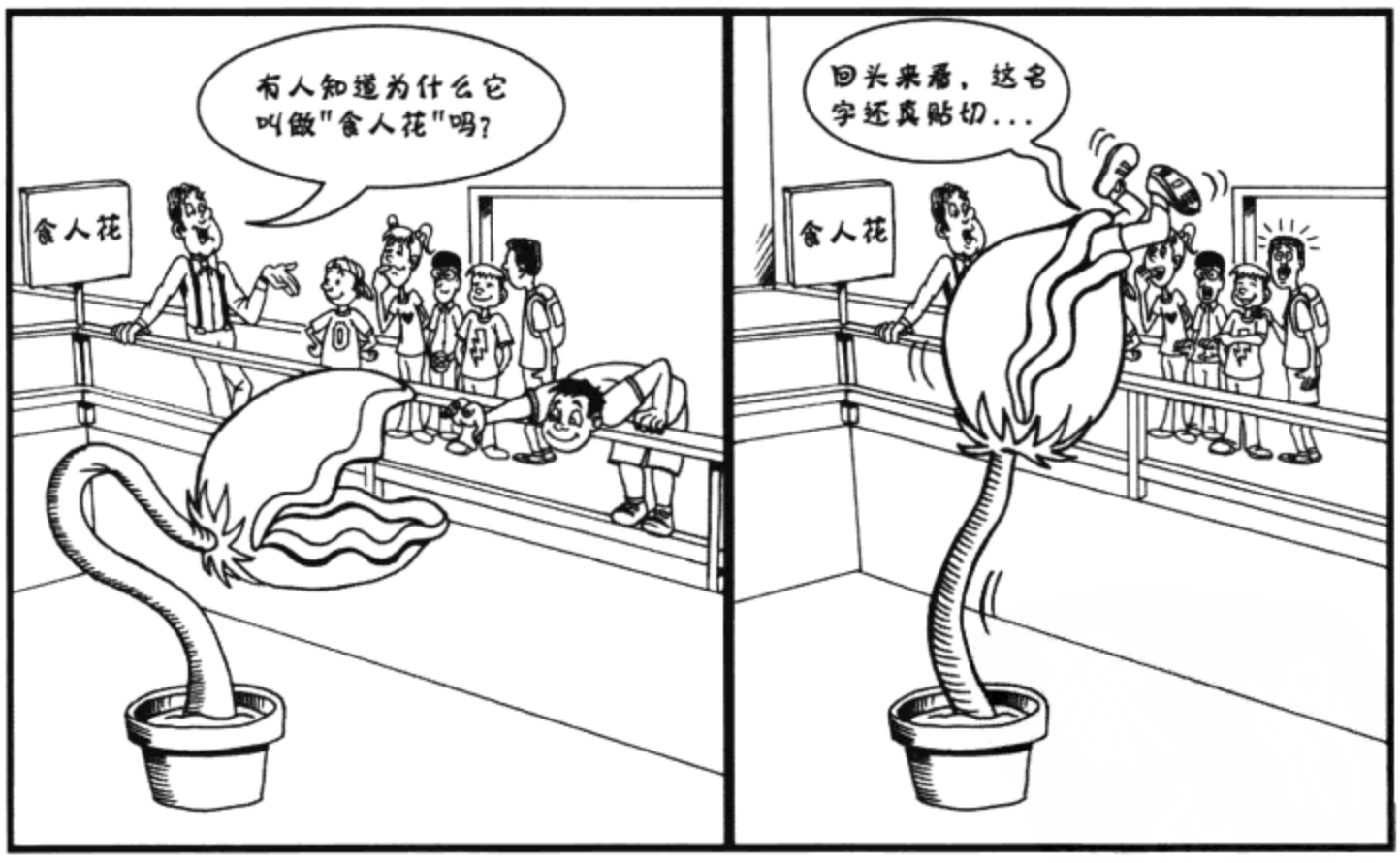
我们的可读性之旅从我们认为“表面层次”的改进开始：选择好的名字、写好的注释以及把代码整洁地写成更好的格式。这些改变很容易应用。你可以在“原位”做这些改变而不必重构代码或者改变程序的运行方式。你还可以增量地做这些修改却不需要投入大量的时间。

这些话题很重要，**因为会影响到你代码库中的每行代码**。尽管每个改变可能看上去都很小，聚集在一起造成代码库巨大的改进。如果你的代码有很棒的名字、写得很好的注释，并且整洁地使用了空白符，你的代码会变得易读得多。

当然，在表面层次之下还有很多关于可读性的东西（我们会在本书的后面涵盖这些内容）。但这一部分的材料几乎不费吹灰之力就应用得如此广泛，值得我们首先讨论。

第2章

把信息装到名字里



无论是命名变量、函数还是类，都可以使用很多相同的原则。我们喜欢把名字当做一条小小的注释。尽管空间不算很大，但选择一个好名字可以让它承载很多信息。

关键思想

把信息装入名字中。

我们在程序中见到的很多名字都很模糊，例如tmp。就算是看上去合理的词，如size或者get，也都没有装入很多信息。本章会告诉你如何把信息装入名字中。

本章分成6个专题：

- 选择专业的词。
- 避免泛泛的名字（或者说要知道什么时候使用它）。
- 用具体的名字代替抽象的名字。
- 使用前缀或后缀来给名字附带更多信息。
- 决定名字的长度。
- 利用名字的格式来表达含义。

选择专业的词

“把信息装入名字中”包括要选择非常专业的词，并且避免使用“空洞”的词。

例如，“get”这个词就非常不专业，例如在下面的例子中：

```
def GetPage(url): ...
```

“get”这个词没有表达出很多信息。这个方法是从本地的缓存中得到一个页面，还是从数据库中，或者从互联网中？如果是从互联网中，更专业的名字可以是FetchPage()或者DownloadPage()。

下面是一个BinaryTree类的例子：

```
class BinaryTree
    { int Size();
    ...
};
```

你期望Size()方法返回什么呢？树的高度，节点数，还是树在内存中所占的空间？

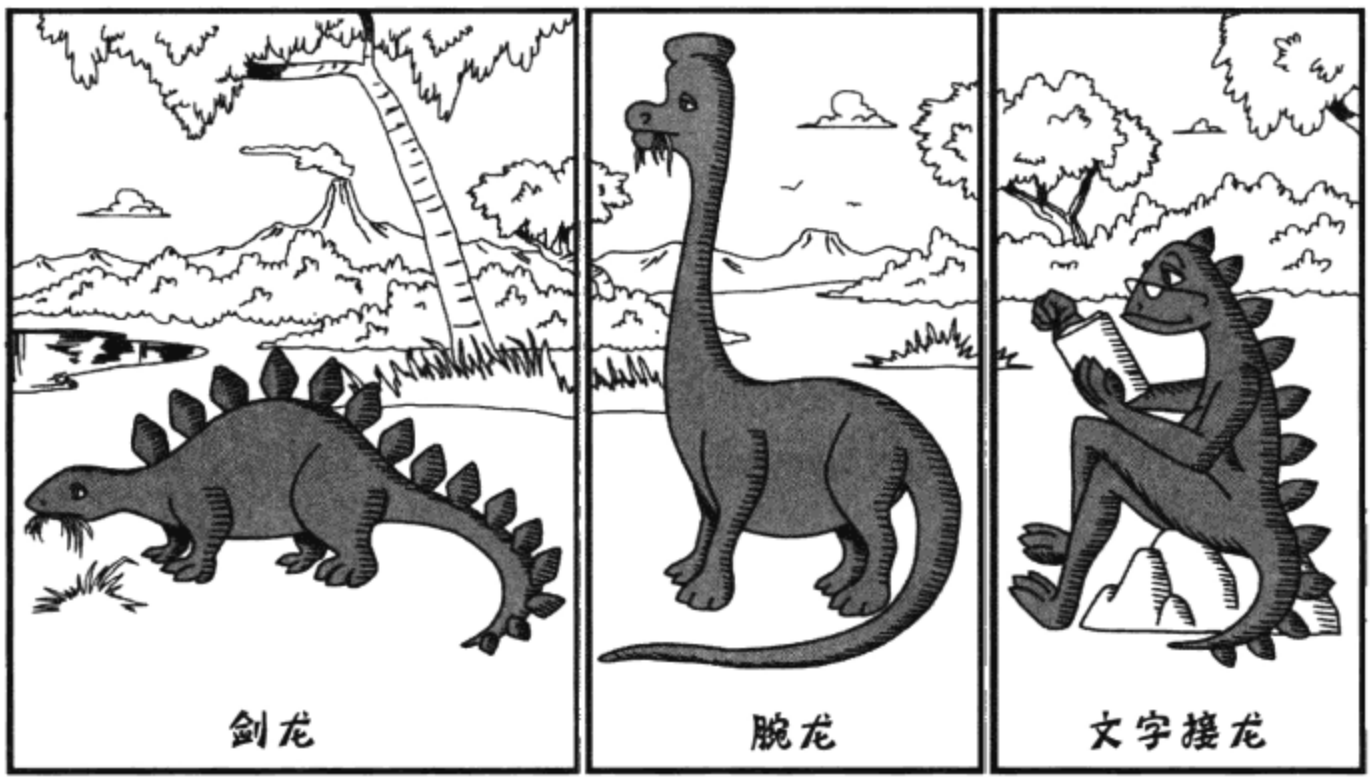
问题是Size()没有承载很多信息。更专业的词可以是Height()、NumNodes()或者MemoryBytes()。

另外一个例子，假设你有某种Thread类：

```
class Thread
    { void Stop();
    ...
};
```

Stop()这个名字还可以，但根据它到底做什么，可能会有更专业的名字。例如，你可以叫它Kill()，如果这是一个重量级操作，不能恢复。或者你可以叫它Pause()，如果有方法让它Resume()。

找到更有表现力的词



要勇于使用同义词典或者问朋友更好的名字建议。英语是一门丰富的语言，有很多词可以选择。

下面是一些例子，这些单词更有表现力，可能适合你的语境：

单词	更多选择
send	deliver、 dispatch、announce、distribute、route
find	search、extract、locate、recover
start	launch、create、begin、open
make	create、set up、build、generate、compose、add、new

但别得意忘形。在PHP中，有一个函数可以explode()一个字符串。这是个很有表现力的名字，描绘了一幅把东西拆成碎片的景象。但这与split()有什么不同？（这是两个不一样的函数，但很难通过它们的名字来猜出不同点在哪里。）

关键思想

清晰和精确比装可爱好。

避免像tmp和retval这样泛泛的名字

使用像tmp、retval和foo这样的名字往往是“我想不出名字”的托辞。与其使用这样空洞的名字，不如挑一个能描述这个实体的值或者目的的名字。

例如，下面的JavaScript函数使用了retval：

```
var euclidean_norm = function (v) {
    var retval = 0.0;
    for (var i = 0; i < v.length; i += 1)
        retval += v[i] * v[i];
    return Math.sqrt(retval);
};
```

当你想不出更好的名字来命名返回值时，很容易想到使用retval。但retval除了“我是一个返回值”外并没有包含更多信息（这里的意义往往也是很明显的）。

好的名字应当描述变量的目的或者它所承载的值。在本例中，这个变量正在累加v的平方。因此更贴切的名字可以是sum_squares。这样就提前声明了这个变量的目的，并且可能会帮忙找到缺陷。

例如，想象如果循环的内部被意外写成：

```
        retval += v[i];
```

如果名字换成sum_squares这个缺陷就会更明显：

```
sum_squares += v[i]; //我们要累加的"square"在哪里？缺陷！
```

建议

retval这个名字没有包含很多信息。用一个描述该变量的值的名字来代替它。

然而，有些情况下泛泛的名字也承载着意义。让我们来看看什么时候使用它们有意义。

tmp

请想象一下交换两个变量的经典情形：

```
if (right < left) {
    tmp = right;
    right = left;
```

```
        left = tmp;
    }
```

在这种情况下，tmp这个名字很好。这个变量唯一的目的就是临时存储，它的整个生命周期只在几行代码之间。tmp这个名字向读者传递特定信息，也就是这个变量没有其他职责，它不会被传到其他函数中或者被重置以反复使用。

但在下面的例子中对tmp的使用仅仅是因为懒惰：

```
String tmp = user.name();
tmp += " " + user.phone_number();
tmp += " " + user.email();
...
template.set("user_info", tmp);
```

尽管这里的变量只有很短的生命周期，但对它来讲最重要的并不是临时存储。用像user_info这样的名字来代替可能会更具描述性。

在下面的情况中，tmp应当出现在名字中，但只是名字的一部分：

```
tmp_file = tempfile.NamedTemporaryFile()
...
SaveData(tmp_file, ...)
```

请注意我们把变量命名为tmp_file而非只是tmp，因为这是一个文件对象。想象一下如果我们只是把它叫做tmp：

```
SaveData(tmp, ...)
```

只要看看这么一行代码，就会发现不清楚tmp到底是文件、文件名还是要写入的数据。

建议

tmp这个名字只应用于短期存在且临时性为其主要存在因素的变量。

循环迭代器

像i、j、iter和it等名字常用做索引和循环迭代器。尽管这些名字很空泛，但是大家都知道它们的意思是“我是一个迭代器”（实际上，如果你用这些名字来表示其他含义，那会很混乱。所以不要这么做！）

但有时会有比i、j、k更贴切的迭代器命名。例如，下面的循环要找到哪个user属于哪个club：

```
for (int i = 0; i < clubs.size(); i++)
    for (int j = 0; j < clubs[i].members.size(); j++)
        for (int k = 0; k < users.size(); k++)
```

```
if (clubs[i].members[k] == users[j])
    cout << "user[" << j << "] is in club[" << i << "]" << endl;
```

在if条件语句中，members[]和users[]用了错误的索引。这样的缺陷很难发现，因为这一行代码单独来看似乎没什么问题：

```
if (clubs[i].members[k] == users[j])
```

在这种情况下，使用更精确的名字可能会有帮助。如果不把循环索引命名为（i、j、k），另一个选择可以是（club_i、members_i、user_i）或者，更简化一点（ci、mi、ui）。这种方式会帮助把代码中的缺陷变得更明显：

```
if (clubs[ci].members[ui] == users[mi]) #缺陷！第一个字母不匹配。
```

如果用得正确，索引的第一个字母应该与数据的第一个字符匹配：

```
if (clubs[ci].members[mi] == users[ui])  #OK。首字母匹配。
```

对于空泛名字的裁定

如你所见，在某些情况下空泛的名字也有用处。

建议

如果你要使用像tmp、it或者retval这样空泛的名字，那么你要有个好的理由。

很多时候，仅仅因为懒惰而滥用它们。这可以理解，如果想不出更好的名字，那么用个没有意义的名字，像foo，然后继续做别的事，这很容易。但如果你养成习惯多花几秒钟想出个好名字，你会发现你的“命名能力”很快提升。

用具体的名字代替抽象的名字

在给变量、函数或者其他元素命名时，要把它描述得更具体而不是更抽象。

例如，假设你有一个内部方法叫做ServerCanStart()，它检测服务是否可以监听某个给定的TCP/IP端口。然而ServerCanStart()有点抽象。CanListenOnPort()就更具体一些。这个名字直接地描述了这个方法要做什么事情。

下面的两个例子更深入地描绘了这个概念。

例子：DISALLOW_EVIL_CONSTRUCTORS

这个例子来自Google的代码库。在C++里，如果你不为类定义拷贝构造函数或者赋值操作符，那就会有一个默认的。尽管这很方便，这些方法很容易导致内存泄漏以及其他灾难，因为它们在你可能想不到的“幕后”地方运行。

所以，Google有个便利的方法来禁止这些“邪恶”的建构函数，就是用这个宏：

```
class ClassName {
```

```
    private:
      DISALLOW_EVIL_CONSTRUCTORS(ClassName);

    public: ...
};
```

这个宏定义成：

```
#define DISALLOW_EVIL_CONSTRUCTORS(ClassName) \
    ClassName(const ClassName&); \
    void operator=(const ClassName&);
```

通过把这个宏放在类的私有部分中，这两个方法[译注1]成为私有的，所以不能用它们，即使意料之外的使用也是不可能的。

然而DISALLOW_EVIL_CONSTRUCTORS这个名字并不是很好。对于“邪恶”这个词的使用包含了对于一个有争议话题过于强烈的立场。更重要的是，这个宏到底禁止了什么这一点是不清楚的。它禁止了operator=()方法，但这个方法甚至根本就不是构造函数！

这个名字使用了几年，但最终换成了一个不那么嚣张而且更具体的名字：

```
#define DISALLOW_COPY_AND_ASSIGN(ClassName) ...
```

例子：--run_locally （本地运行）

我们的一个程序有个可选的命令行标志叫做--run_locally。这个标志会使得这个程序输出额外的调试信息，但是会运行得更慢。这个标志一般用于在本地机器上测试，例如在笔记本电脑上。但是当这个程序运行在远程服务器上时，性能是很重要的，因此不会使用这个标志。

你能看出来为什么会有--run_locally这个名字，但是它有几个问题：

- 团队里的新成员不知道它到底是做什么的，可能在本地运行时使用它（想象一下），但不明白为什么需要它。
- 偶尔，我们在远程运行这个程序时也要输出调试信息。向一个运行在远端的程序传递--run_locally看上去很滑稽，而且很让人迷惑。
- 有时我们可能要在本地运行性能测试，这时我们不想让日志把它拖慢，所以我们不会使用--run_locally。

这里的问题是--run_locally是由它所使用的典型环境而得名。用像--extra_logging这样的名字来代换可能会更直接明了。

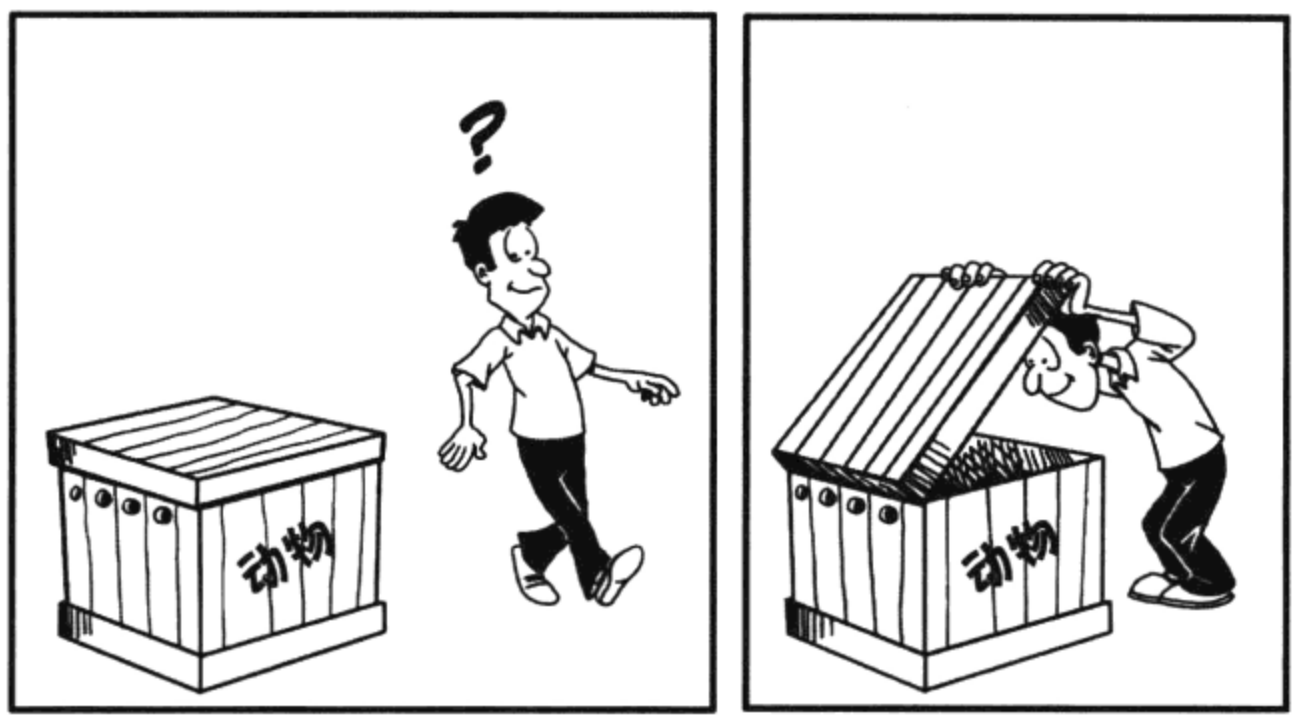
但是如果--run_locally需要做比额外日志更多的事情怎么办？例如，假设它需要建立和

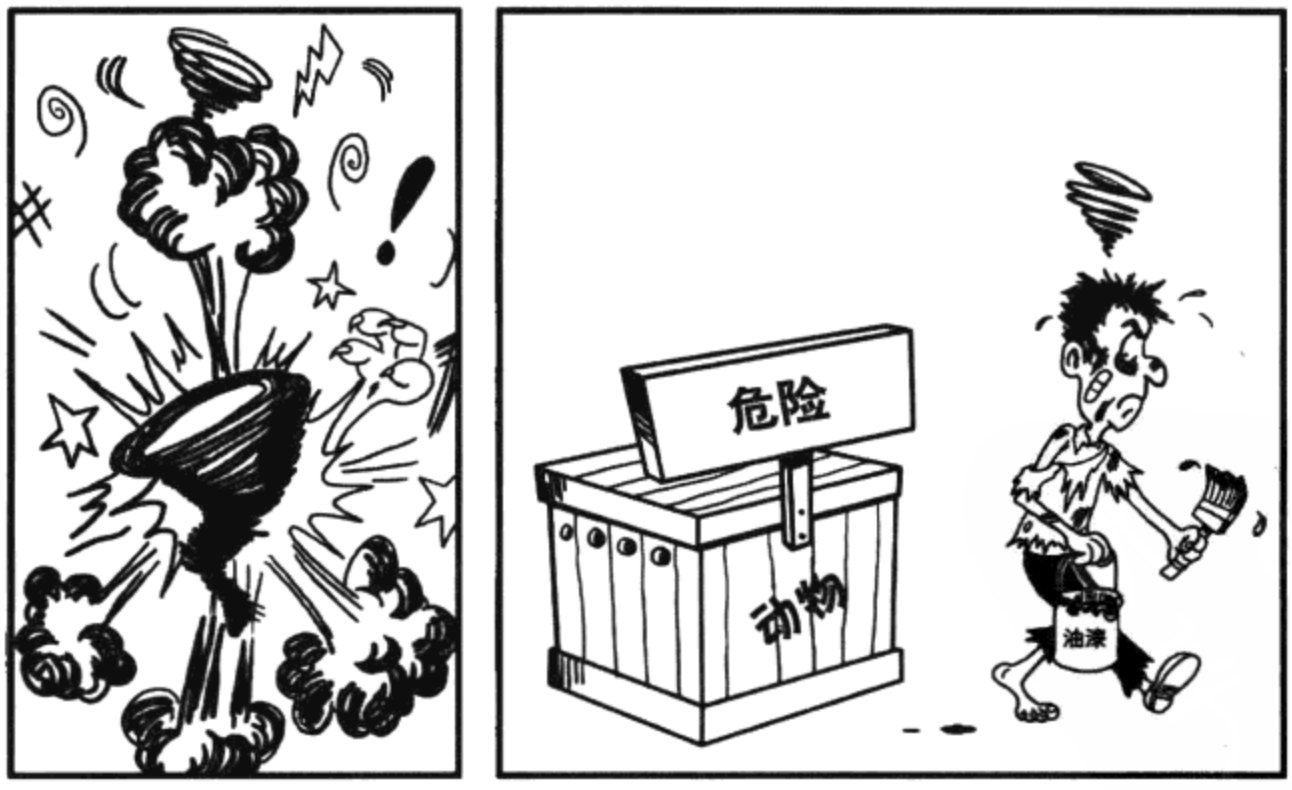
译注1： 指拷贝构造函数和赋值操作符。

使用一个特殊的本地数据库。现在--run_locally看上去更吸引人了，因为它可以同时控制这两种情况。

但这样用的话就变成了因为一个名字含糊婉转而需要选择它，这可能不是一个好主意。更好的办法是再创建一个标志叫--use_local_database。尽管你现在要用两个标志，但这两个标志非常明确，不会混淆两个正交的含义，并且你可明确地选择一个。

为名字附带更多信息





我们前面提到，一个变量名就像是一个小小的注释。尽管空间不是很大，但不管你在名中挤进任何额外的信息，每次有人看到这个变量名时都会同时看到这些信息。

因此，如果关于一个变量有什么重要事情的读者必须知道，那么是值得把额外的“词”添加到名字中的。例如，假设你有一个变量包含一个十六进制字符串：

```
string id; // Example: "af84ef845cd8"
```

如果让读者记住这个ID的格式很重要的话，你可以把它改名为hex_id。

带单位的值

如果你的变量是一个度量的话（如时间长度或者字节数），那么最好把名字带上它的单位。

例如，这里有些JavaScript代码用来度量一个网页的加载时间：

```
var start = (new Date()).getTime(); // top of the page
...
var elapsed = (new Date()).getTime() - start; // bottom of the page
document.writeln("Load time was: " + elapsed + " seconds");
```

这段代码里没有明显的错误，但它不能正常运行，因为getTime()会返回毫秒而非秒。

通过给变量结尾追加_ms，我们可以让所有的地方更明确：

```
var start_ms = (new Date()).getTime(); // top of the page
...
var elapsed_ms = (new Date()).getTime() - start_ms; // bottom of the page
document.writeln("Load time was: " + elapsed_ms / 1000 + " seconds");
```

除了时间，还有很多在编程时会遇到的单位。下表列出一些没有单位的函数参数以及带单位的版本：

函数参数	带单位的参数
Start(int delay)	delay → delay_secs
CreateCache(int size)	size → size_mb
ThrottleDownload(float limit)	limit → max_kbps
Rotate(float angle)	angle → degrees_cw

附带其他重要属性

这种给名字附带额外信息的技巧不仅限于单位。在对于这个变量存在危险或者意外的任何时候你都该采用它。

例如，很多安全漏洞来源于没有意识到你的程序接收到的某些数据还没有处于安全状态。在这种情况下，你可能想要使用像untrustedUrl或者unsafeMessageBody这样

的名字。在调用了清查不安全输入的函数后，得到的变量可以命名为trustedUrl或者safeMessageBody。

下表给出更多需要给名字附加上额外信息的例子：

情形	变量名	更好的名字
一个“纯文本”格式的密码，需要加密后才能进一步使用	password	plaintext_password
一条用户提供的注释，需要转义之后才能用于显示	comment	unescaped_comment
已转化为UTF-8格式的html字节	html	html_utf8
以“url方式编码”的输入数据	data	data_urlenc

但你不应该给程序中每个变量都加上像unescaped_或者_utf8这样的属性。如果有人误解了这个变量就很容易产生缺陷，尤其是会产生像安全缺陷这样可怕的结果，在这些地方这种技巧最有用武之地。基本上，如果这是一个需要理解的关键信息，那就把它放在名字里。

这是匈牙利表示法吗？

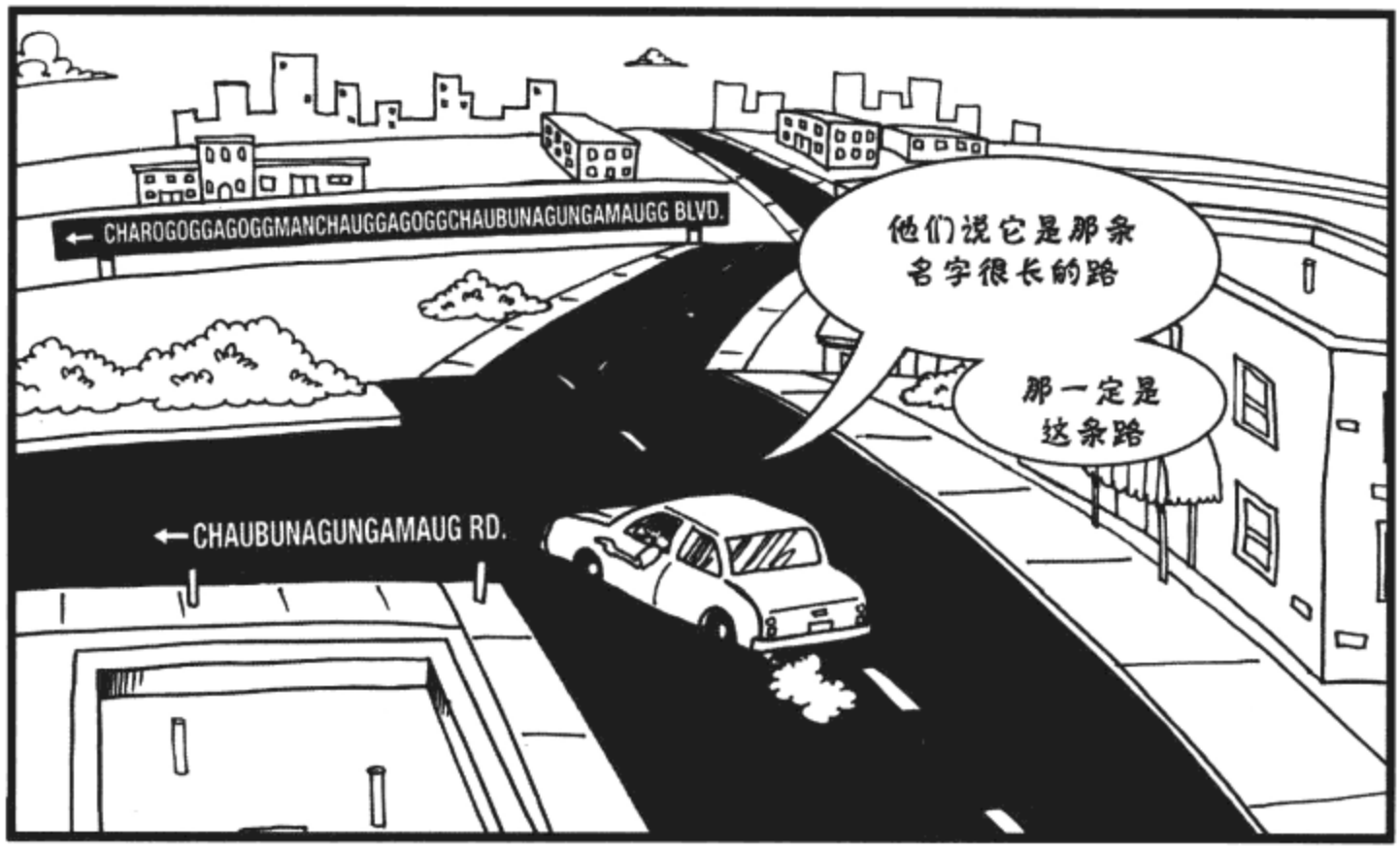
匈牙利表示法是一个在微软广泛应用的命名系统，它把每个变量的“类型”信息都编写进名字的前缀里。下面有几个例子：

名字	含义
pLast	指向某数据结构最后一个元素的指针（p）
pszBuffer	指向一个以零结尾（z）的字符串（s）的指针（p）
cch	一个字符（ch）计数（c）
mpcopx	在指向颜色的指针（pco）和指向x轴长度的指针（px）之间的一个映射（m）

这实际上就是“给名字附带上属性”的例子。但它是一种更正式和严格的系统，关注于特有的一系列属性。

我们在这一部分所提倡的是更广泛的、更加非正式的系统：标识变量的任何关键属性，如果需要的话以易读的方式把它加到名字里。你可以把这称为“英语表示法”。

名字应该有多长



当选择好名字时，有一个隐含的约束是名字不能太长。没人喜欢在工作中遇到这样的标识符：

```
newNavigationControllerWrappingViewControllerForDataSourceOfClass
```

名字越长越难记，在屏幕上占的地方也越大，可能会产生更多的换行。

另一方面，程序员也可能走另一个极端，只用单个单词（或者单一字母）的名字。那么如何来处理这种平衡呢？如何来决定是把一变量命名为d、days还是days_since_last_update呢？

这是要你自己要拿主意的，最好的答案和这个变量如何使用有关系，但下面还是提出了一些指导原则。

在小的作用域里可以使用短的名字

当你去短期度假时，你带的行李通常会比长假少。同样，“作用域”小的标识符（对于多少行其他代码可见）也不用带上太多信息。也就是说，因为所有的信息（变量的类型、它的初值、如何析构等）都很容易看到，所以可以用很短的名字。

```
if (debug) {
    map<string,int> m;
    LookUpNamesNumbers(&m);
```

```
        Print(m);
    }
```

尽管m这个名字并没有包含很多信息，但这不是个问题。因为读者已经有了需要理解这段代码的所有信息。

然而，假设m是一个全局变量中的类成员，如果你看到这个代码片段：

```
LookUpNamesNumbers(&m);
Print(m);
```

这段代码就没有那么好读了，因为m的类型和目的都不明确。

因此如果一个标识符有较大的作用域，那么它的名字就要包含足够的信息以便含义更清楚。

输入长名字——不再是个问题

有很多避免使用长名字的理由，但“不好输入”这一条已经不再有效。我们所见到的所有的编程文本编辑器都有内置的“单词补全”的功能。令人惊讶的是，大多数程序员并没有注意到这个功能。如果你还没在你的编辑器上试过这个功能，那么请现在就放下本书然后试一下下面这些功能：

1. 键入名字的前面几个字符。
2. 触发单词补全功能（见下表）。
3. 如果补全的单词不正确，一直触发这个功能直到正确的名字出现 。

它非常准确。这个功能在任何语种的任何类型的文件中都可以用。并且它对于任何单词（token）都有效，甚至在你输入注释时也行。

编辑器	命令
Vi	Ctrl+p
Emacs	Meta+/（先按ESC，然后按/）
Eclipse	Alt+/
IntelliJ IDEA	Alt+/
TextMate	ESC

首字母缩略词和缩写

程序员有时会采用首字母缩略词和缩写来命令，以便保持较短的名字，例如，把一个类命名为BEManager而不是BackEndManager。这种名字会让人费解，冒这种风险是否值得？

在我们的经验中，使用项目所特有的缩写词非常糟糕。对于项目的新成员来讲它们看上

去太令人费解和陌生，当过了相当长的时间以后，即使是对于原作者来讲，它们也会变得令人费解和陌生。

所以经验原则是：**团队的新成员是否能理解这个名字的含义**？如果能，那可能就没有问题。

例如，对程序员来讲，使用eval来代替evaluation，用doc来代替document，用str来代替string是相当普遍的。因此如果团队的新成员看到FormatStr()可能会理解它是什么意思，然而，理解BEManager可能有点困难。

丢掉没用的词

有时名字中的某些单词可以拿掉而不会损失任何信息。例如，ConvertToString()就不如ToString()这个更短的名字，而且没有丢失任何有用的信息。同样，不用DoServeLoop()，ServeLoop()也一样清楚。

利用名字的格式来传递含义

对于下划线、连字符和大小写的使用方式也可以把更多信息装到名字中。例如，下面是一些遵循Google开源项目格式规范的C++代码：

```
static const int kMaxOpenFiles = 100;

class LogReader {
  public:
    void OpenFile(string local_file);

  private:
    int offset_;
    DISALLOW_COPY_AND_ASSIGN(LogReader);
};
```

对不同的实体使用不同的格式就像语法高亮显示的形式一样，能帮你更容易地阅读代码。

该例子中的大部分格式都很常见，使用CamelCase来表示类名，使用lower_separated来表示变量名。但有些规范也可能会出乎你的意料。

例如，常量的格式是kConstantName而不是CONSTANT_NAME。这种形式的好处是容易和#define的宏区分开，宏的规范是MACRO_NAME。

类成员变量和普通变量一样，但必须以一条下划线结尾，如offset_。刚开始看，可能会觉得这个规范有点怪，但是能立刻区分出是成员变量还是其他变量，这一点还是很方便的。例如，如果你在浏览一个大的方法中的代码，看到这样一行：

```
stats.clear();
```

你本来可能要想“stats属于这个类吗？这行代码是否会改变这个类的内部状态？”如果用了member_这个规范，你就能迅速得到结论：“不，stats一定是个局部变量。否则它就会命名为stats_。”

其他格式规范

根据项目上下文或语言的不同，还可以采用其他一些格式规范使得名字包含更多信息。

例如，在《JavaScript：The Good Parts》（Douglas Crockford, O'Reilly, 2008）一书中，作者建议“构造函数”（在新建时会调用的函数）应该首字母大写而普通函数首字母小字：

```
var x = new DatePicker();    // DatePicker() is a "constructor" function
var y = pageHeight();        // pageHeight() is an ordinary function
```

下面是另一个JavaScript例子：当调用jQuery库函数时（它的名字是单个字符$），一条非常有用的规范是，给jQuery返回的结果也加上$作为前缀：

```
var $all_images = $("img");  // $all_images is a jQuery object
var height = 250;            // height is not
```

在整段代码中，都会清楚地看到$all_images是个jQuery返回对象。

下面是最后一个例子，这次是HTML/CSS：当给一个HTML标记加id或者class属性时，下划线和连字符都是合法的值。一个可能的规范是用下划线来分开ID中的单词，用连字符来分开class中的单词。

```
<div id="middle_column" class="main-content"> ...
```

是否要采用这些规范是由你和你的团队决定的。但不论你用哪个系统，在你的项目中要保持一致。

总结

本章唯一的主题是：**把信息塞入名字中**。这句话的含意是，读者仅通过读到名字就可以获得大量信息。

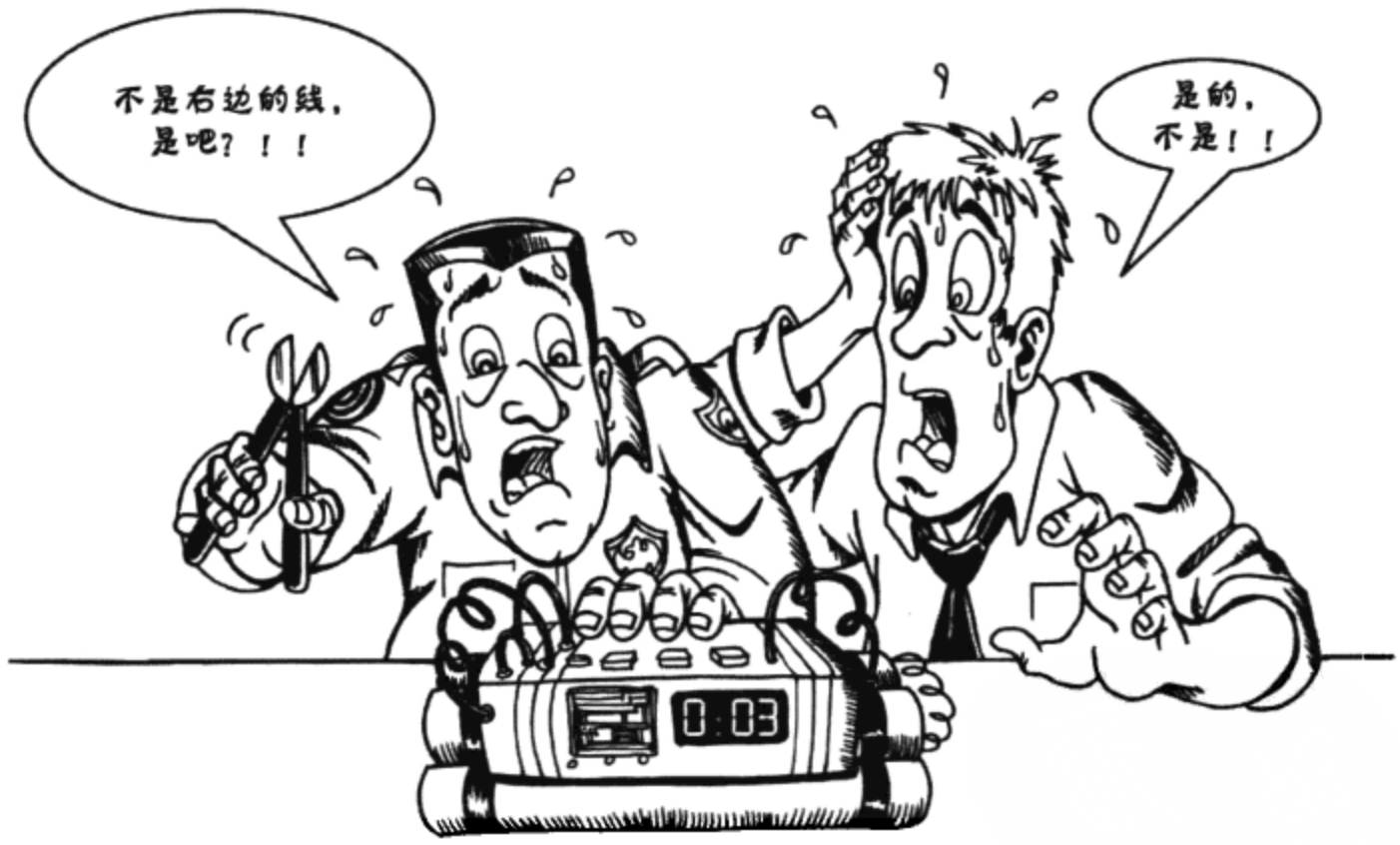
下面是讨论过的几个小提示：

- **使用专业的单词**——例如，不用Get，而用Fetch或者Download可能会更好，这由上下文决定。
- **避免空泛的名字**，像tmp和retval，除非使用它们有特殊的理由。

- **使用具体的名字来更细致地描述事物**——ServerCanStart()这个名字就比CanListenOnPort更不清楚。
- **给变量名带上重要的细节**——例如，在值为毫秒的变量后面加上_ms，或者在还需要转义的，未处理的变量前面加上raw_。
- **为作用域大的名字采用更长的名字**——不要用让人费解的一个或两个字母的名字来命名在几屏之间都可见的变量。对于只存在于几行之间的变量用短一点的名字更好。
- **有目的地使用大小写、下划线等**——例如，你可以在类成员和局部变量后面加上"_"来区分它们。

第3章

不会误解的名字



在前一章中，我们讲到了如何把信息塞入名字中。本章会关注另一个话题：小心可能会有歧义的名字。

> **关键思想**
>
> 要多问自己几遍：“这个名字会被别人解读成其他的含义吗？”要仔细审视这个名字。

如果想更有创意一点，那么可以主动地寻找“误解点”。这一步可以帮助你发现那些二义性名字并更改。

例如，在本章中，当我们讨论每一个可能会误解的名字时，我们将在心里默读，然后挑选更好的名字。

例子：Filter()

假设你在写一段操作数据库结果的代码：

```
results = Database.all_objects.filter("year <= 2011")
```

结果现在包含哪些信息？

- 年份小于或等于2011的对象？
- 年份不小于或等于2011年的对象？

这里的问题是“filter”是个二义性单词。我们不清楚它的含义到底是“挑出”还是“减掉”。最好避免使用“filter”这个名字，因为它太容易误解。

例子：Clip(text, length)

假设你有个函数用来剪切一个段落的内容：

```
# Cuts off the end of the text, and appends "..."
 def Clip(text, length):
     ...
```

你可能会想象到Clip()的两种行为方式：

- 从尾部删除length的长度
- 截掉最大长度为length的一段

第二种方式（截掉）的可能性最大，但还是不能肯定。与其让读者乱猜代码，还不如把函数的名字改成Truncate(text, length)。

然而，参数名length也不太好。如果叫max_length的话可能会更清楚。

这样也还没有完。就算是max_length这个名字也还是会有多种解读：

- 字节数
- 字符数
- 字数

如你在前一章中所见，这属于应当把单位附加在名字后面的那种情况。在本例中，我们是指“字符数”，所以不应该用max_length，而要用max_chars。

推荐用min和max来表示（包含）极限

假设你的购物车应用程序最多不能超过10件物品：

```
CART_TOO_BIG_LIMIT = 10

if shopping_cart.num_items() >= CART_TOO_BIG_LIMIT:
    Error("Too many items in cart.")
```

这段代码有个经典的“大小差一”缺陷。我们可以简单地通过把>=变成>来改正它：

```
if shopping_cart.num_items() > CART_TOO_BIG_LIMIT:
```

（或者通过把CART_TOO_BIG_LIMIT变成11）。但问题的根源在于 CART_TOO_BIG_LIMIT是个二义性名字，它的含义到底是“少于”还是“少于/且包括”。

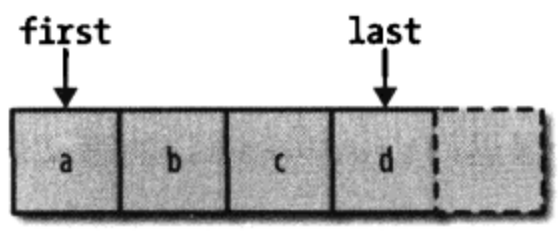
建议

命名极限最清楚的方式是在要限制的东西前加上max_或者min_。

在本例中，名字应当是MAX_ITEMS_IN_CART，新代码现在变得简单又清楚：

```
MAX_ITEMS_IN_CART = 10

if shopping_cart.num_items() > MAX_ITEMS_IN_CART:
    Error("Too many items in cart.")
```

推荐用first和last来表示包含的范围



下面是另一个例子，你没法判断它是“少于”还是“少于且包含”：

```
print integer_range(start=2, stop=4)
# Does this print [2,3] or [2,3,4] (or something else)?
```

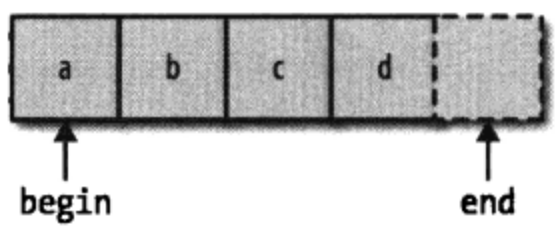
尽管start是个合理的参数名，但stop可以有多种解读。对于这样包含的范围（这种范围包含开头和结尾），一个好的选择是first/last。例如：

```
set.PrintKeys(first="Bart", last="Maggie")
```

不像stop，last这个名字明显是包含的。

除了first/last，min/max这两个名字也适用于包含的范围，如果它们在上下文中“听上去合理”的话。

推荐用begin和end来表示包含/排除范围



在实践中，很多时候用包含/排除范围更方便。例如，如果你想打印所有发生在10月16日的事件，那么写成这样很简单：

```
PrintEventsInRange("OCT 16 12:00am", "OCT 17 12:00am")
```

这样写就没那么简单了：

```
PrintEventsInRange("OCT 16 12:00am", "OCT 16 11:59:59.9999pm")
```

因此对于这些参数来讲，什么样的一对名字更好呢？对于命名包含/排除范围典型的编程规范是使用begin/end。

但是end这个词有点二义性。例如，在句子“我读到这本书的end部分了”，这里的end是包含的。遗憾的是，英语中没有一个合适的词来表示“刚好超过最后一个值”。

因为对begin/end的使用是如此常见（至少在C++标准库中是这样用的，还有大多数需要“分片”的数组也是这样用的），它已经是最好的选择了。

给布尔值命名

当为布尔变量或者返回布尔值的函数选择名字时，要确保返回true和false的意义很明确。

下面是个危险的例子：

```
bool read_password = true;
```

这会有两种截然不同的解释：

- 我们需要读取密码。
- 已经读取了密码。

在本例中，最好避免用“read”这个词，用need_password或者user_is_authenticated这样的名字来代替。

通常来讲，加上像is、has、can或should这样的词，可以把布尔值变得更明确。

例如，SpaceLeft()函数听上去像是会返回一个数字，如果它的本意是返回一个布尔值，可能HasSapceLeft()个这名字更好一些。

最后，最好避免使用反义名字。例如，不要用：

```
bool disable_ssl = false;
```

而更简单易读（而且更紧凑）的表示方式是：

```
bool use_ssl = true;
```

与使用者的期望相匹配

有些名字之所以会让人误解是因为用户对它们的含义有先入为主的印象，就算你的本意并非如此。在这种情况下，最好放弃这个名字而改用一个不会让人误解的名字。

例子：get*()

很多程序员都习惯了把以get开始的方法当做“轻量级访问器”这样的用法，它只是简单地返回一个内部成员变量。如果违背这个习惯很可能会误导用户。

以下是一个用Java写的例子，请不要这样做：

```
public class StatisticsCollector {
    public void addSample(double x) { ... }

    public double getMean() {
        // Iterate through all samples and return total / num_samples
    }
    ...
}
```

在这个例子中，getMean()的实现是要遍历所有经过的数据并同时计算中值。如果有大量

的数据的话，这样的一步可能会有很大的代价！但一个容易轻信的程序员可能会随意地调用getMean()，还以为这是个没什么代价的调用。

相反，这个方法应当重命名为像computeMean()这样的名字，后者听起来更像是有些代价的操作。（另一种做法是，用新的实现方法使它真的成为一个轻量级的操作。）

例子：list::size()

下面是一个来自C++标准库中的例子。曾经有个很难发现的缺陷，使得我们的一台服务器慢得像蜗牛在爬，就是下面的代码造成的：

```
void ShrinkList(list<Node>& list, int max_size) {
    while (list.size() > max_size) {
        FreeNode(list.back());
        list.pop_back();
    }
}
```

这里的“缺陷”是，作者不知道list.size()是一个*O(n)*操作——它要一个节点一个节点地历数列表，而不是只返回一个事先算好的个数，这就使得ShrinkList()成了一个$O(n^2)$操作。

这段代码从技术上来讲“正确”，事实上它也通过了所有的单元测试。但当把ShrinkList()应用于有100万个元素的列表上时，要花超过一个小时来完成！

可能你在想：“这是调用者的错，他应该更仔细地读文档。”有道理，但在本例中，list.size()不是一个固定时间的操作，这一点是出人意料的。所有其他的C++容器类的size()方法都是时间固定的。

假使size()的名字是countSize()或者countElements()，很可能就会避免相同的错误。C++标准库的作者可能是希望把它命名为size()以和所有其他的容器一致，就像vector和map。但是正因为他们的这个选择使得程序员很容易误把它当成一个快速的操作，就像其他的容器一样。谢天谢地，现在最新的C++标准库把size()改成了*O(1)*。

向导是谁

一段时间以前，有位作者正在安装OpenBSD操作系统。在磁盘格式化这一步时，出现了一个复杂的菜单，询问磁盘参数。其中的一个选项是进入“向导模式”（Wizard mode）。他看到这个友好的选择松了一口气，并选择了它。让他失望的是，安装程序给出了低层命名行提示符等待手动输入磁盘格式化命令，而且也没有明显的方法可以退出。很明显，这里的“向导”指的是你自己。

例子：如何权衡多个备选名字

当你要选一个好名字时，可能会同时考虑多个备选方案。通常你要在头脑中盘算一下每个名字的好处，然后才能得出最后的选择。下面的例子示范了这个评判过程。

高流量网站常常用“试验”来测试一个对网站的改变是否会对业务有帮助。下面的例子是一个配置文件，用来控制某些试验：

```
experiment_id: 100
description: "increase font size to 14pt"
traffic_fraction: 5%
...
```

每个试验由15对属性/值来定义。遗憾的是，当要定义另一个差不多的试验时，你不得不拷贝和粘贴其中的大部分。

```
experiment_id: 101
description: "increase font size to 13pt"
[other lines identical to experiment_id 100]
```

假设我们希望改善这种情况，方法是让一个试验重用另一个的属性（这就是“原型继承”模式）。其结果是你可能会写出这样的东西：

```
experiment_id: 101
the_other_experiment_id_I_want_to_reuse: 100
[change any properties as needed]
```

问题是：the_other_experiment_id_I_want_to_reuse到底应该如何命名？下面有4个名字供考虑：

1. template
2. reuse
3. copy
4. inherit

所有的这些名字对我们来讲都有意义，因为是我们把这个新功能加入配置语言中的。但我们要想象一下对于看到这段代码却又不知道这个功能的人来讲，这个名字听起来是什么意思。因此我们要分析每一个名字，考虑各种让人误解的可能性。

1. 让我们想象一下使用这个名字模板时的情形：

```
experiment_id: 101
template: 100
...
```

template有两个问题。首先，我们不是很清楚它的意思是“我是一个模板”还是“我

在用其他模板”。其次，“template”常常指代抽象事物，必须要先“填充”之后才会变“具体”。有人会以为一个模板化了的试验不再是一个“真正的”试验。总之，template对于这种情况来讲太不明确。

2. 那么reuse呢？

```
experiment_id: 101
reuse: 100
...
```

reuse这个单词还可以，但有人会以为它的意思是“这个试验最多可以重用100次”。把名字改成reuse_id会好一点。但有的读者可能会以为reuse_id的意思是“我重用的id是100”。

3. 让我们再考虑一下copy。

```
experiment_id: 101
copy: 100
...
```

copy这个词不错。但copy:100看上去像是在说“拷贝这个试验100次”或者“这是什么东西的第100个拷贝”。为了确保明确地表达这个名字是引用另一个试验，我们可以把名字改成copy_experiement。这可能是到目前为止最好的名字了。

4. 但现在我们再来考虑一下inherit：

```
experiment_id: 101
inherit: 100
...
```

大多数程序员都熟悉“inherit”（继承）这个词，并且都理解在继承之后会有进一步的修改。在类继承中，你会从另一个类中得到所有的方法和成员，然后修改它们或者添加更多内容。甚至在现实生活中，我们说从亲人那里继承财产，大家都理解你可能会卖掉它们或者再拥有更多属于你自己的东西。

但是如果要明确它是继承自另一个试验，我们可以把名字改进成inherit_from，或者甚至是inherit_from_experiement_id。

综上所述，copy_experiment和inherit_from_experiment_id是最好的名字，因为它们对所发生的事情描述最清楚，并且最不可能误解。

总结

不会误解的名字是最好的名字——阅读你代码的人应该理解你的本意，并且不会有其他的理解。遗憾的是，很多英语单词在用来编程时是多义性的，例如filter、length和limit。

在你决定使用一个名字以前，要吹毛求疵一点，来想象一下你的名字会被误解成什么。最好的名字是不会误解的。

当要定义一个值的上限或下限时，max_和min_是很好的前缀。对于包含的范围，first和last是好的选择。对于包含/排除范围，begin和end是最好的选择，因为它们最常用。

当为布尔值命名时，使用is和has这样的词来明确表示它是个布尔值，避免使用反义的词（例如disable_ssl）。

要小心用户对特定词的期望。例如，用户会期望get()或者size()是轻量的方法。

第4章

审美

很多想法来源于杂志的版面设计——段落的长度、栏的宽度、文章的顺序以及把什么东西放在封面上等。一本好的杂志既可以跳着看，也可以从头读到尾，怎么看都很容易。

好的源代码应当“看上去养眼”。本章会告诉大家如何使用好的留白、对齐及顺序来让你的代码变得更易读。

确切地说，有三条原则：

- 使用一致的布局，让读者很快就习惯这种风格。
- 让相似的代码看上去相似。
- 把相关的代码行分组，形成代码块。

审美与设计

在本章中，我们只关注可以改进代码的简单“审美”方法。这些类型的改变很简单并且常常能大幅地提高可读性。有时大规模地重构代码（例如拆分出新的函数或者类）可能会更有帮助。我们的观点是好的审美与好的设计是两种独立的思想。最好是同时在两个方向上努力做到更好。

为什么审美这么重要

假设你不得不用这个类：

```
class StatsKeeper {
public:
// A class for keeping track of a series of doubles
    void Add(double d);  // and methods for quick statistics about them
   private:   int count;         /* how many so    far
*/ public:
        double Average();
private:   double minimum;
list<double>
  past_items
      ;double maximum;
};
```

相对于下面这个更整洁的版本，你可能要花更多的时间来理解上面的代码：

```
// A class for keeping track of a series of doubles
// and methods for quick statistics about them.
class StatsKeeper {
  public:
    void Add(double d);
    double Average();

  private:
    list<double> past_items;
    int count;  // how many so far

    double minimum;
    double maximum;
};
```

很明显，**使用从审美角度讲让人愉悦的代码更容易**。试想一下，你编程的大部分时间都花在看代码上！浏览代码的速度越快，人们就越容易使用它。

重新安排换行来保持一致和紧凑

假设你在写Java代码来评估你的程序在不同的网络连接速度下的行为。你有一个TcpConnectionSimulator，它的构造函数有4个参数：

1. 网络连接的速度（Kbps）
2. 平均延时（ms）
3. 延时的“抖动”（ms）
4. 丢包率（ms）

你的代码需要3个不同的TcpConnectionSimulator实例：

```
public class PerformanceTester {
    public static final TcpConnectionSimulator wifi = new TcpConnectionSimulator(
```

```
            500, /* Kbps */
            80, /* millisecs latency */
            200, /* jitter */
            1 /* packet loss % */);

        public static final TcpConnectionSimulator t3_fiber =
            new TcpConnectionSimulator(
                45000, /* Kbps */
                10, /* millisecs latency */
                0, /* jitter */
                0 /* packet loss % */);

        public static final TcpConnectionSimulator cell = new TcpConnectionSimulator(
            100, /* Kbps */
            400, /* millisecs latency */
            250, /* jitter */
            5 /* packet loss % */);
    }
```

这段示例代码需要有很多额外的换行来满足每行80个字符的限制（这是你们公司的编码规范）。遗憾的是，这使得t3_fiber的定义看上去和它的邻居不一样。这段代码的“剪影”看上去很怪，它毫无理由地让t3_fiber很突兀。这违反了“相似的代码应当看上去相似”这条原则。

为了让代码看上去更一致，我们可以引入更多的换行（同时还可以让注释对齐）

```
    public class PerformanceTester {
        public static final TcpConnectionSimulator wifi =
            new TcpConnectionSimulator(
                500,   /* Kbps */
                80,    /* millisecs latency */
                200,   /* jitter */
                1      /* packet loss % */);

        public static final TcpConnectionSimulator t3_fiber =
            new TcpConnectionSimulator(
                45000, /* Kbps */
                10,    /* millisecs latency */
                0,     /* jitter */
                0      /* packet loss % */);

        public static final TcpConnectionSimulator cell =
            new TcpConnectionSimulator(
                100, /* Kbps */
                400, /* millisecs latency */ 250, /* jitter */
                5 /* packet loss % */);
    }
```

这段代码有优雅一致的风格，并且很容易从头看到尾快速浏览。但遗憾的是，它占用了更多纵向的空间。并且它还把注释重复了3遍。

下面是写这个类的更紧凑方法：

```
public class PerformanceTester {
    // TcpConnectionSimulator(throughput, latency, jitter, packet_loss)
    //                              [Kbps]   [ms]    [ms]    [percent]

    public static final TcpConnectionSimulator wifi =
        new TcpConnectionSimulator(500, 80, 200, 1);

    public static final TcpConnectionSimulator t3_fiber =
        new TcpConnectionSimulator(45000, 10, 0, 0);

    public static final TcpConnectionSimulator cell =
        new TcpConnectionSimulator(100, 400, 250, 5);
}
```

我们把注释挪到了上面，然后把所有的参数都放在一行上。现在尽管注释不再紧挨相邻的每个数字，但“数据”现在排成更紧凑的一个表格。

用方法来整理不规则的东西

假设你有一个个人数据库，它提供了下面这个函数：

```
// Turn a partial_name like "Doug Adams" into "Mr. Douglas Adams".
// If not possible, 'error' is filled with an explanation.
string ExpandFullName(DatabaseConnection dc, string partial_name, string* error);
```

并且这个函数由一系列的例子来测试：

```
DatabaseConnection database_connection;
string error;
assert(ExpandFullName(database_connection, "Doug Adams", &error)
    == "Mr. Douglas Adams");
assert(error == "");
assert(ExpandFullName(database_connection, " Jake Brown ", &error)
    == "Mr. Jacob Brown III");
assert(error == "");
assert(ExpandFullName(database_connection, "No Such Guy", &error) == "");
assert(error == "no match found");
assert(ExpandFullName(database_connection, "John", &error) == "");
assert(error == "more than one result");
```

这段代码没什么美感可言。有些行长得都换行了。这段代码的剪影很难看，也没有什么一致的风格。

但对于这种情况，重新布置换行也仅能做到如此。更大的问题是这里有很多重复的串，例如"assert(ExpandFullName(database_connection...", 其中还有很多的"error"。要是真的想改进这段代码，需要一个辅助方法。就像这样：

```
CheckFullName("Doug Adams", "Mr. Douglas Adams", "");
CheckFullName(" Jake Brown ", "Mr. Jake Brown III", "");
CheckFullName("No Such Guy", "", "no match found");
CheckFullName("John", "", "more than one result");
```

现在，很明显这里有4个测试，每个使用了不同的参数。尽管所有的“脏活”都放在CheckFullName()中，但是这个函数也没那么差：

```
void CheckFullName(string partial_name,
                   string expected_full_name,
                   string expected_error) {
    // database_connection is now a class member
    string error;
    string full_name = ExpandFullName(database_connection, partial_name, &error);
    assert(error == expected_error);
    assert(full_name == expected_full_name); }
```

尽管我们的目的仅仅是让代码更有美感，但这个改动同时有几个附带的效果：

- 它消除了原来代码中大量的重复，让代码变得更紧凑。
- 每个测试用例重要的部分（名字和错误字符串）现在都变得很直白。以前，这些字符串是混杂在像database_connection和error这样的标识之间的，这使得一眼看全这段代码变得很难。
- 现在添加新测试应当更简单

这个故事想要传达的寓意是使代码“看上去漂亮”通常会带来不限于表面层次的改进，它可能会帮你把代码的结构做得更好。

在需要时使用列对齐

整齐的边和列让读者可轻松地浏览文本。

有时你可以借用“列对齐”的方法来让代码易读。例如，在前一部分中，你可以用空白把CheckFullName()的参数排成：

```
CheckFullName("Doug Adams"    , "Mr. Douglas Adams" , "");
CheckFullName(" Jake  Brown ", "Mr. Jake Brown III", "");
CheckFullName("No Such Guy"   , ""                  , "no match found");
CheckFullName("John"          , ""                  , "more than one result");
```

在这段代码中，很容易区分出CheckFullName()的第二个和第三个参数。下面是一个简单的例子，它有一大组变量定义：

```
# Extract POST parameters to local variables
details  = request.POST.get('details')
location = request.POST.get('location')
phone    = equest.POST.get('phone')
email    = request.POST.get('email')
url      = request.POST.get('url')
```

你可能注意到了，第三个定义有个拼写错误（把request写成了equest）。当所有的内容都这么整齐地排列起来时，这样的错误就很明显。

在wget数据库中，可用的命令行选项（有一百多项）这样列出：

```
commands[] = {
    ...
    { "timeout",          NULL,                cmd_spec_timeout },
    { "timestamping",     &opt.timestamping,   cmd_boolean },
    { "tries",            &opt.ntry,           cmd_number_inf },
    { "useproxy",         &opt.use_proxy,      cmd_boolean },
    { "useragent",        NULL,                cmd_spec_useragent },
    ...
};
```

这种方式使行这个列表很容易快读和从一列跳到另一列。

你应该用列对齐吗

列的边提供了“可见的栏杆”，阅读起来很方便。这是个“让相似的代码看起来相似”的好例子。

但有些程序员不喜欢它。一个原因是，建立和维护对齐的工作量很大。另一个原因是，在改动时它造成了更多的“不同”，对一行的改动可能会导致另外5行也要改动（大部分只是空白）。

我们的建议是要试试。在我们的经验中，它并不像程序员担心的那么费工夫。如果真的很费工夫，你可以不这么做。

选一个有意义的顺序，始终一致地使用它

在很多情况下，代码的顺序不会影响其正确性。例如，下面的5个变量定义可以写成任意的顺序：

```
details  = request.POST.get('details')
location = request.POST.get('location')
phone    = request.POST.get('phone')
email    = request.POST.get('email')
url      = request.POST.get('url')
```

在这种情况下，不要随机地排序，把它们按有意义的方式排列会有帮助。下面是一些想法：

- 让变量的顺序与对应的HTML表单中<input>字段的顺序相匹配。
- 从“最重要”到“最不重要”排序。
- 按字母顺序排序。

无论使用什么顺序，你在代码中应当始终使用这一顺序。如果后面改变了这个顺序，那会让人很困惑：

```
if details: rec.details  = details
if phone:   rec.phone    = phone  //Hey, where did 'location' go?
if email:   rec.mail     = email
if url:     rec.url      = url
if location: rec.location = location # Why is 'location' down here now?
```

把声明按块组织起来

我们的大脑很自然地会按照分组和层次结构来思考，因此你可以通过这样的组织方式来帮助读者快速地理解你的代码。

例如，下面是一个前端服务器的C++类，这里有它所有方法的声明：

```
class FrontendServer {
    public:
      FrontendServer();
      void ViewProfile(HttpRequest* request);
      void OpenDatabase(string location, string user);
      void SaveProfile(HttpRequest* request);
      string ExtractQueryParam(HttpRequest* request, string param);
      void ReplyOK(HttpRequest* request, string html);
      void FindFriends(HttpRequest* request);
      void ReplyNotFound(HttpRequest* request, string error);
      void CloseDatabase(string location);
      ~FrontendServer();
};
```

这不是很难看的代码，但可以肯定这样的布局不会对读者更快地理解所有的方法有什么帮助。不要把所有的方法都放到一个巨大的代码块中，应当按逻辑把它们分成组，像以下这样：

```
class FrontendServer {
    public:
      FrontendServer();
      ~FrontendServer();

      // Handlers
      void ViewProfile(HttpRequest* request);
      void SaveProfile(HttpRequest* request);
      void FindFriends(HttpRequest* request);

      // Request/Reply Utilities
      string ExtractQueryParam(HttpRequest* request, string param);
      void ReplyOK(HttpRequest* request, string html);
      void ReplyNotFound(HttpRequest* request, string error);

      // Database Helpers
      void OpenDatabase(string location, string user);
```

```
        void CloseDatabase(string location);
    };
```

这个版本容易理解多了。它还更易读，尽管代码行数更多了。原因是你可以快速地找出4个高层次段落，然后在需要时再阅读每个段落的具体内容。

把代码分成“段落”

书面文字要分成段落是由于以下几个原因：

- 它是一种把相似的想法放在一起并与其他想法分开的方法。
- 它提供了可见的“脚印”，如果没有它，会很容易找不到你读到哪里了。
- 它便于段落之间的导航。

因为同样的原因，代码也应当分成"段落"。例如，没有人会喜欢读下面这样一大块代码：

```
# Import the user's email contacts, and match them to users in our system.
# Then display a list of those users that he/she isn't already friends with.
def suggest_new_friends(user, email_password):
    friends = user.friends()
    friend_emails = set(f.email for f in friends)
    contacts = import_contacts(user.email, email_password)
    contact_emails = set(c.email for c in contacts)
    non_friend_emails = contact_emails - friend_emails
    suggested_friends = User.objects.select(email__in=non_friend_emails)
    display['user'] = user
    display['friends'] = friends
    display['suggested_friends'] = suggested_friends
    return render("suggested_friends.html", display)
```

可能看上去并不明显，但这个函数会经过数个不同的步骤。因此，把这些行代码分成段落会特别有用：

```
def suggest_new_friends(user, email_password):
    # Get the user's friends' email addresses.
    friends = user.friends()
    friend_emails = set(f.email for f in friends)

    # Import all email addresses from this user's email account.
    contacts = import_contacts(user.email, email_password)
    contact_emails = set(c.email for c in contacts)

    # Find matching users that they aren't already friends with.
    non_friend_emails = contact_emails - friend_emails
    suggested_friends = User.objects.select(email__in=non_friend_emails)

    # Display these lists on the page.
    display['user'] = user
```

```
    display['friends'] = friends
    display['suggested_friends'] = suggested_friends

    return render("suggested_friends.html", display)
```

请注意， 我们还给每个段落加了一条总结性的注释，这也会帮助读者浏览代码（参见第5章）。

正如书面文本，有很多种方法可以分开代码，程序员可能会对长一点或短一点的段落有不同的偏好。

个人风格与一致性

有相当一部分审美选择可以归结为个人风格。例如，类定义的大括号该放在哪里：

```
class Logger {
    ...
};
```

还是：

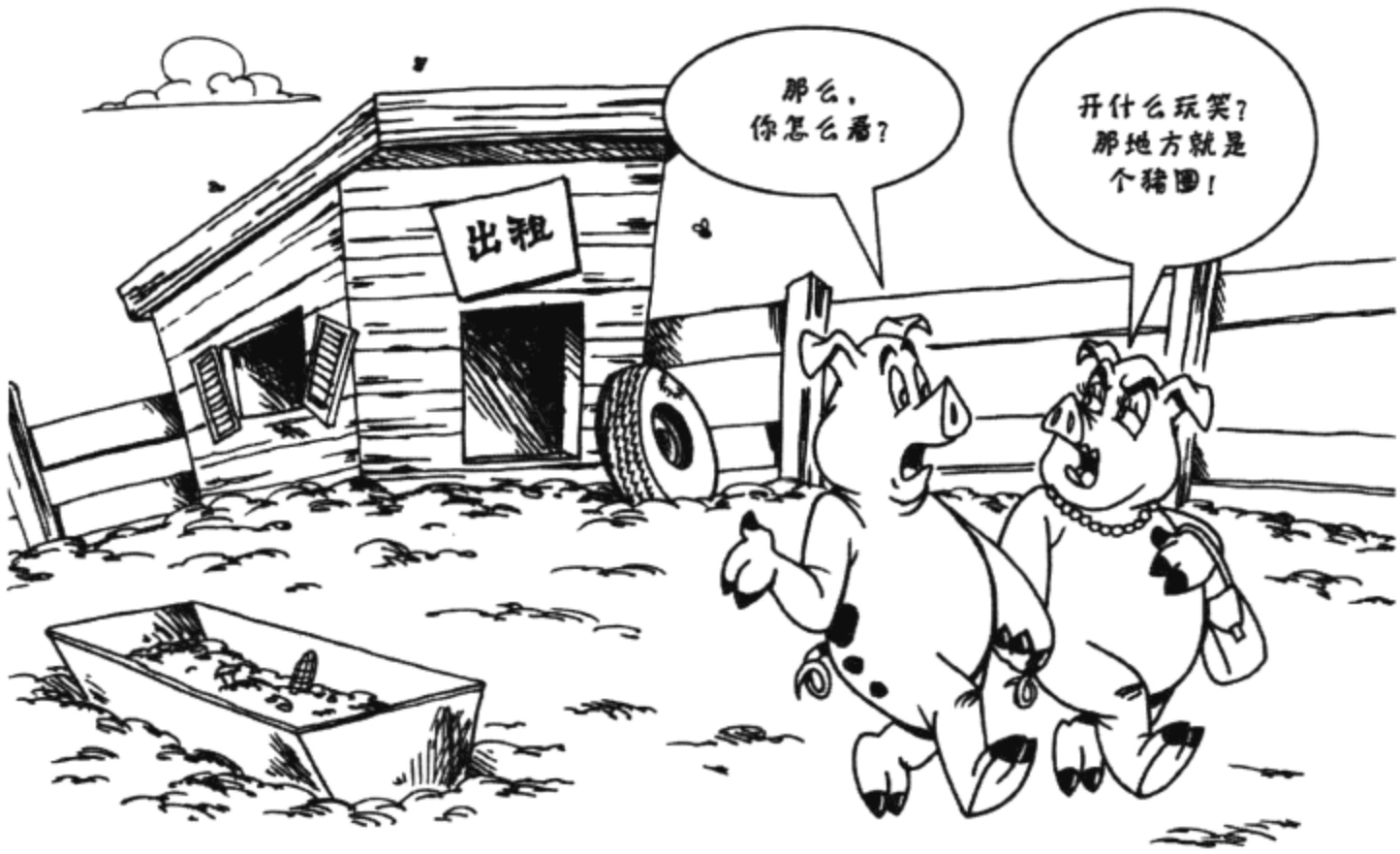
```
class Logger
{
    ...
};
```

选择一种风格而非另一种，不会真的影响到代码的可读性。但如果把两种风格混在一起，就会对可读性有影响了。

曾经在我们所从事过的很多项目中，我们感觉团队所用的风格是“错误”的，但是我们还是遵守项目的习惯，因为我们知道一致性要重要得多。

关键思想

一致的风格比“正确”的风格更重要。



总结

大家都愿意读有美感的代码。通过把代码用一致的、有意义的方式 “格式化”，可以把代码变得更容易读，并且可以读得更快。

下面是讨论过的一些具体技巧：

- 如果多个代码块做相似的事情，尝试让它们有同样的剪影。
- 把代码按“列”对齐可以让代码更容易浏览。
- 如果在一段代码中提到A、B和C，那么不要在另一段中说B、C和A。选择一个有意义的顺序，并始终用这样的顺序。
- 用空行来把大块代码分成逻辑上的“段落”。

第5章

该写什么样的注释

使用手册

需要！！

不需要

本章旨在帮助你明白应该写什么样的注释。你可能以为注释的目的是“解释代码做了什么”，但这只是其中很小的一部分。

关键思想

注释的目的是尽量帮助读者了解得和作者一样多。

当你写代码时，你的脑海里会有很多有价值的信息。当其他人读你的代码时，这些信息已经丢失了——他们所见到的只是眼前的代码。

本章会展示许多例子来说明什么时候应该把你脑海中的信息写下来。我们略去了很多对注释的世俗观点，相对地，我们更关注注释有趣的和“匮乏的”方面。

我们把本章组织成以下几个部分：

- 了解什么不需要注释。
- 用代码记录你的思想。
- 站在读者的角度，去想象他们需要知道什么。

什么不需要注释

阅读注释会占用阅读真实代码的时间，并且每条注释都会占用屏幕上的空间。那么，它最好是物有所值的。那么如何来分辨什么是好的注释，什么是没有价值的注释呢？

下面代码中所有的注释都是没有价值的：

```
// The class definition for Account
class Account {
  public:
    // Constructor
    Account();

    // Set the profit member to a new value
    void SetProfit(double profit);

    // Return the profit from this Account
```

```
    double GetProfit();
};
```

这些注释没有价值是因为它们并没有提供任何新的信息，也不能帮助读者更好地理解代码。

关键思想

不要为那些从代码本身就能快速推断的事实写注释。

这里“快速”是个重要的区别。考虑一下下面这段Python代码：

```
# remove everything after the second '*'
name = '*'.join(line.split('*')[:2])
```

从技术上来讲，这里的注释也没有表达出任何“新信息”。如果你阅读代码本身，你最终会明白它到底在做什么。但对于大多数程序员来讲，读有注释的代码比没有注释的代码理解起来要快速得多。

不要为了注释而注释

有些教授要求他们的学生在他们的代码作业中为每个函数都加上注释。结果是，有些程序员会对没有注释的函数有负罪感，以至于他们把函数的名字和参数用句子的形式重写了一遍：

```
// Find the Node in the given subtree, with the given name, using the given depth.
Node* FindNodeInSubtree(Node* subtree, string name, int depth);
```

这种情况属于“没有价值的注释”一类，函数的声明与其注释实际上是一样的。对于这条注释要么删除它，要么改进它。

如果你想要在这里写条注释，它最好也能给出更多重要的细节：

```
// Find a Node with the given 'name' or return NULL.
// If depth <= 0, only 'subtree' is inspected.
// If depth == N, only 'subtree' and N levels below are inspected.
Node* FindNodeInSubtree(Node* subtree, string name, int depth);
```

不要给不好的名字加注释——应该把名字改好

注释不应用于粉饰不好的名字。例如，有一个叫做CleanReply()的函数，加上了看上去有用的注释：

```
// Enforce limits on the Reply as stated in the Request,
// such as the number of items returned, or total byte size, etc.
void CleanReply(Request request, Reply reply);
```

这里大部分的注释只是在解释“clean”是什么意思。更好的做法是把“enforce limits”这个词组加到函数名里：

```
// Make sure 'reply' meets the count/byte/etc. limits from the 'request'
void EnforceLimitsFromRequest(Request request, Reply reply);
```

这个函数现在更加“自我说明”了。一个好的名字比一个好的注释更重要，因为在任何用到这个函数的地方都能看得到它。

下面是另一个例子，给名字不大好的函数加注释：

```
// Releases the handle for this key. This doesn't modify the actual registry.
void DeleteRegistry(RegistryKey* key);
```

DeleteRegistery()这个名字听起来像是一个很危险的函数（它会删除注册表？！）注释里的“它不会改动真正的注册表”是想澄清困惑。

我们可以用一个更加自我说明的名字，就像：

```
void ReleaseRegistryHandle(RegistryKey* key);
```

通常来讲，你不需要“拐杖式注释”——试图粉饰可读性差的代码的注释。写代码的人常常把这条规则表述成：**好代码>坏代码+好注释。**

记录你的思想

现在你知道了什么不需要注释，下面讨论什么需要注释（但往往没有注释）。

很多好的注释仅通过“记录你的想法”就能得到，也就是那些你在写代码时有过的重要想法。

加入“导演评论”

电影中常有“导演评论”部分，电影制作者在其中给出自己的见解并且通过讲故事来帮助你理解这部电影是如何制作的。同样，你应该在代码中也加入注释来记录你对代码有价值的见解。

下面是一个例子：

```
// 出乎意料的是，对于这些数据用二叉树比用哈希表快40%
// 哈希运算的代价比左/右比较大得多
```

这段注释教会读者一些事情，并且防止他们为无谓的优化而浪费时间。

下面是另一个例子：

```
// 作为整体可能会丢掉几个词。这没有问题。要100%解决太难了
```

如果没有这段注释，读者可能会以为这是个bug然后浪费时间尝试找到能让它失败的测试用例，或者尝试改正这个bug。

注释也可以用来解释为什么代码写得不那么整洁：

```
// 这个类正在变得越来越乱
// 也许我们应该建立一个'ResourceNode'子类来帮助整理
```

这段注释承认代码很乱，但同时也鼓励下一个人改正它（还给出了具体的建议）。如果没有这段注释，很多读者可能会被这段乱代码吓到而不敢碰它。

为代码中的瑕疵写注释

代码始终在演进，并且在这过程中肯定会有瑕疵。不要不好意思把这些瑕疵记录下来。例如，当代码需要改进时：

```
// TODO: 采用更快算法
```

或者当代码没有完成时：

```
// TODO(dustin)：处理除JPEG以外的图像格式
```

有几种标记在程序员中很流行：

标记	通常的意义
TODO：	我还没有处理的事情
FIXME：	已知的无法运行的代码
HACK：	对一个问题不得不采用的比较粗糙的解决方案
XXX：	危险！这里有重要的问题

你的团队可能对于是否可以使用及何时使用这些标记有具体的规范。例如，TODO: 可能只用于重要的问题。如果是这样，你可以用像todo:（小写）或者maybe-later:这样的方法表示次要的缺陷。

重要的是你应该可以随时把代码将来应该如何改动的想法用注释记录下来。这种注释给读者带来对代码质量和当前状态的宝贵见解，甚至可能会给他们指出如何改进代码的方向。

给常量加注释

当定义常量时，通常在常量背后都有一个关于它是什么或者为什么它是这个值的“故事”。例如，你可能会在代码中看到如下常量：

```
NUM_THREADS = 8
```

这一行看上去可能不需要注释，但很可能选择用这个值的程序员知道得比这个要多：

```
NUM_THREADS = 8 # as long as it's >= 2 * num_processors, that's good enough.
```

现在，读代码的人就有了调整这个值的指南了（比如，设置成1可能就太低了，设置成50又太夸张了）。

或者有时常量的值本身并不重要。达到这种效果的注释也会有用：

```
// Impose a reasonable limit - no human can read that much anyway.
const int MAX_RSS_SUBSCRIPTIONS = 1000;
```

还有这样的情况，它是一个高度精细调整过的值，可能不应该大幅改动。

```
image_quality = 0.72; // users thought 0.72 gave the best size/quality tradeoff
```

在上述所有例子中，你可能不会想到要加注释，但它们的确很有帮助。

有些常量不需要注释，因为它们的名字本身已经很清楚（例如SECONDS_PER_DAY）。但是在我们的经验中，很多常量可以通过加注释得以改进。这不过是匆匆记下你在决定这个常量值时的想法而已。

站在读者的角度

我们在本书中所用的一个通用的技术是想象你的代码对于外人来讲看起来是什么样子的，这个人并不像你那样熟悉你的项目。这个技术对于发现什么地方需要注释尤其有用。

意料之中的提问

“有什么问题吗？……除了告示牌上已经回答过的。”

当别人读你的代码时，有些部分更可能让他们有这样的想法：“什么？为什么会这样？”你的工作就是要给这些部分加上注释。

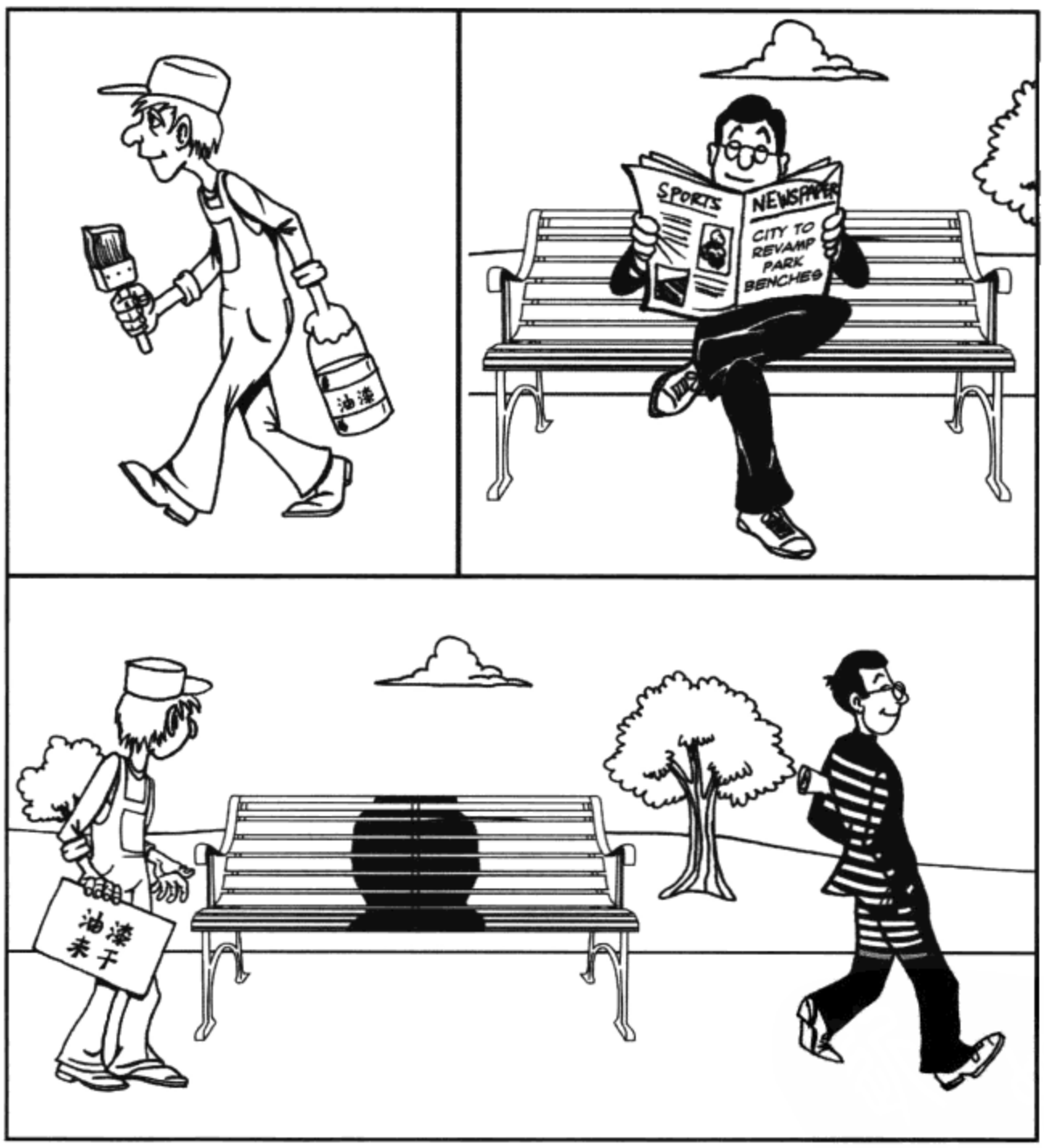
例如，看看下面Clear()的定义：

```
struct Recorder {
    vector<float> data;
    ...
    void Clear() {
        vector<float>().swap(data); // Huh? Why not just data.clear()?
    }
};
```

大多数C++程序员看到这段代码时都会想：“为什么他不直接用data.clear()而是与一个空的向量交换？”实际上只有这样才能强制使向量真正地把内存归还给内存分配器。这不是一个众所周知的C++细节。起码要加上这样的注释：

```
// Force vector to relinquish its memory (look up "STL swap trick")
vector<float>().swap(data);
```

公布可能的陷阱



当为一个函数或者类写文档时，可以问自己这样的问题：“这段代码有什么出人意料的地方？会不会被误用？”基本上就是说你需要“未雨绸缪”，预料到人们使用你的代码时可能会遇到的问题。

例如，假设你写了一个函数来向给定的用户发邮件：

```
void SendEmail(string to, string subject, string body);
```

这个函数的实现包括连接到外部邮件服务，这可能会花整整一秒，或者更久。可能有人在写Web应用时在不知情的情况下错误地在处理HTTP请求时调用这个函数。（这么做可能会导致他们的Web应用在邮件服务宕机时“挂起”。）

为了避免这种灾难，你应当为这个“实现细节”加上注释：

```
// 调用外部服务来发送邮件。（1分钟之后超时。）
void SendEmail(string to, string subject, string body);
```

下面有另一个例子：假设你有一个函数FixBrokenHtml()用来尝试重写损坏的HTML，通过插入结束标记这样的方法：

```
def FixBrokenHtml(html): ...
```

这个函数运行得很好，但要警惕当有深嵌套而且不匹配的标记时它的运行时间会暴增。对于很差的HTML输入，该函数可能要运行几分钟。

与其让用户自己慢慢发现这一点，不如提前声明：

```
// 运行时间将达到 O(number_tags * average_tag_depth)，所以小心严重嵌套的输入。
def FixBrokenHtml(html): ...
```

“全局观”注释

对于团队的新成员来讲，最难的事情之一就是理解“全局观”——类之间如何交互，数据如何在整个系统中流动，以及入口点在哪里。设计系统的人经常忘记给这些东西加注释，“只缘身在此山中”。

思考下面的场景：**有新人刚刚加入你的团队，她坐在你旁边，而你需要让她熟悉代码库。**

在你带领她浏览代码库时，你可能会指着某些文件或者类说这样的话：

- “这段代码把我们的业务逻辑与数据库粘在一起。任何应用层代码都不该直接使用它。”
- “这个类看上去很复杂，但它实际上只是个巧妙的缓存。它对系统中的其他部分一无所知。”

在一分钟的随意对话之后，你的新团队成员就知道得比她自己读源代码更多了。

这正是那种应该包含在高级别注释中的信息。

下面是一个文件级别注释的简单例子：

```
// 这个文件包含一些辅助函数，为我们的文件系统提供了更便利的接口
// 它处理了文件权限及其他基本的细节。
```

不要对于写庞大的正式文档这种想法不知所措。几句精心选择的话比什么都没有强。

总结性注释

就算在一个函数的内部，给“全局观”写注释也是个不错的主意。下面是一个例子，这段注释巧妙地总结了其后的低层代码：

```
# Find all the items that customers purchased for themselves.
for customer_id in all_customers:
    for sale in all_sales[customer_id].sales:
        if sale.recipient == customer_id:
            ...
```

没有这段注释，每行代码都有些谜团。（我知道这是在遍历all_customers……但是为什么要这么做？）

在包含几大块的长函数中这种总结性的注释尤其有用：

```
def GenerateUserReport():
    # Acquire a lock for this user
    ...
    # Read user's info from the database
    ...
```

```
# Write info to a file
...
# Release the lock for this user
```

这些注释同时也是对于函数所做事情的总结，因此读者可以在深入了解细节之前就能得到该函数的主旨。（如果这些大段很容易分开，你可以直接把它们写成函数。正如我们前面提到的，好代码比有好注释的差代码要强。）

注释应该说明“做什么”、“为什么”还是“怎么做”？

你可能听说过这样的建议：“注释应该说明‘为什么这样做’而非‘做什么’（或者‘怎么做’）”。这虽然很容易记，但我们觉得这种说法太简单化，并且对于不同的人有不同的含义。

我们的建议是你可以做任何能帮助读者更容易理解代码的事。这可能也会包含对于“做什么”、“怎么做”或者“为什么”的注释（或者同时注释这三个方面）。

最后的思考——克服“作者心理阻滞”

很多程序员不喜欢写注释，因为要写出好的注释感觉好像要花很多工夫。当作者有了这种“作者心理阻滞”，最好的办法就是现在就开始写。因此下次当你对写注释犹豫不决时，就直接把你心里想的写下来就好了，虽然这种注释可能是不成熟的。

例如，假设你正在写一个函数，然后心想：“哦，天啊，如果一旦这东西在列表中有重复的话会变得很难处理的。”那么就直接把它写下来：

```
// 哦，天啊，如果一旦这东西在列表中有重复的话会变得很难处理的。
```

看到了，这难吗？它作为注释来讲实际上没那么差——起码比没有强。可能措辞有点含糊。要改正这一点，可以把每个子句改得更专业一些：

- “哦，天啊”，实际上，你的意思是“小心：这个地方需要注意”。
- “这东西”，实际上，你的意思是“处理输入的这段代码”。
- “会变得很难处理”，实际上，你的意思是“会变得难以实现”。

新的注释可以是：

```
// 小心：这段代码不会处理列表中的重复（因为这很难做到）
```

请注意我们把写注释这件事拆成了几个简单的步骤：

1. 不管你心里想什么，先把它写下来。

2. 读一下这段注释，看看有没有什么地方可以改进。

3. 不断改进。

当你经常写注释，你就会发现步骤1所产生的注释变得越来越好，最后可能不再需要做任何修改了。并且通过早写注释和常写注释，你可以避免在最后要写一大堆注释这种令人不快的状况。

总结

注释的目的是帮助读者了解作者在写代码时已经知道的那些事情。本章介绍了如何发现所有的并不那么明显的信息块并且把它们写下来。

什么地方不需要注释：

- 能从代码本身中迅速地推断的事实。
- 用来粉饰烂代码（例如蹩脚的函数名）的“拐杖式注释”——应该把代码改好。

你应该记录下来的想法包括：

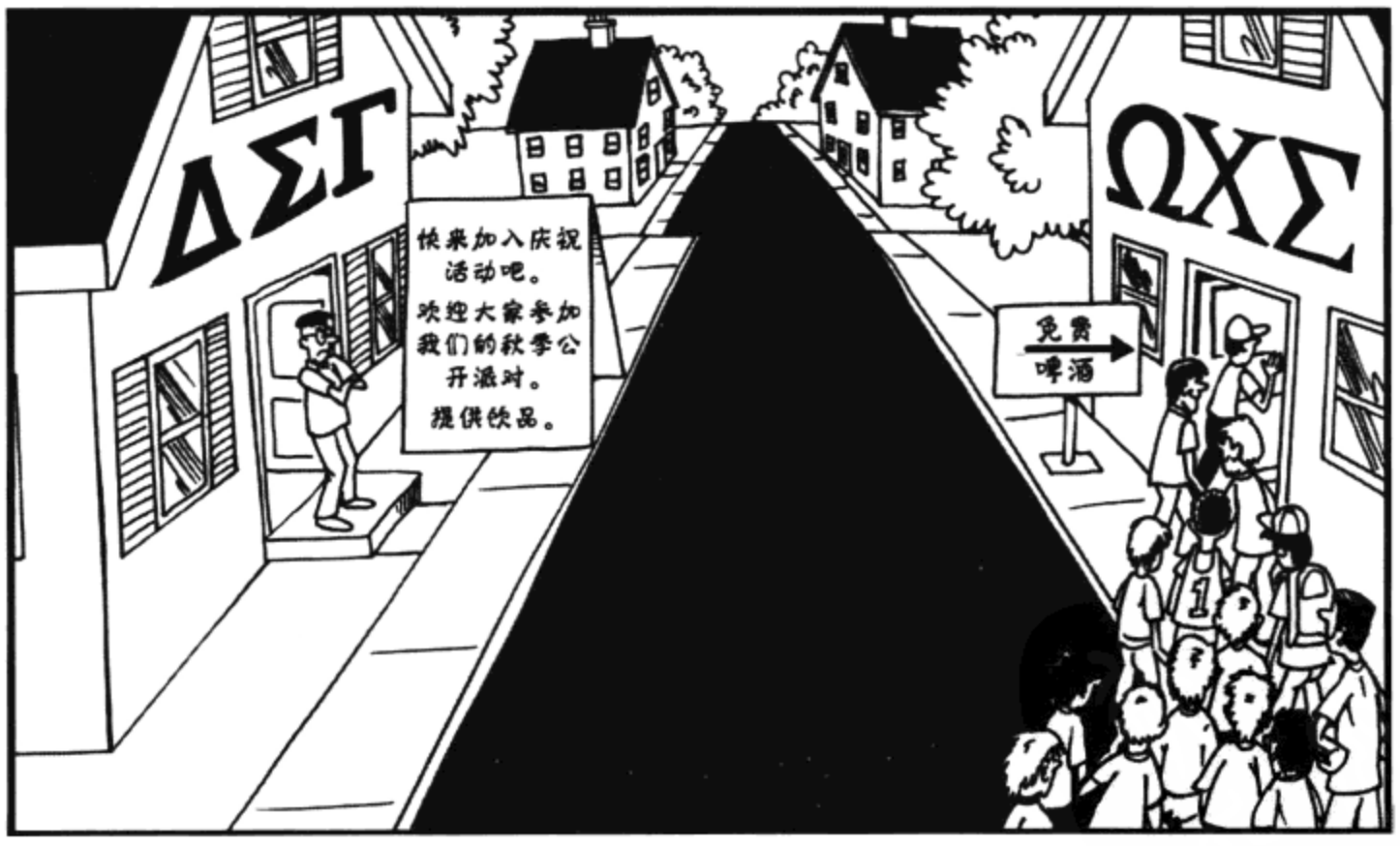
- 对于为什么代码写成这样而不是那样的内在理由（“指导性批注”）。
- 代码中的缺陷，使用像TODO:或者XXX:这样的标记。
- 常量背后的故事，为什么是这个值。

站在读者的立场上思考：

- 预料到代码中哪些部分会让读者说：“啊？”并且给它们加上注释。
- 为普通读者意料之外的行为加上注释。
- 在文件/类的级别上使用“全局观”注释来解释所有的部分是如何一起工作的。
- 用注释来总结代码块，使读者不致迷失在细节中。

第6章

写出言简意赅的注释



前一章是关于发现什么地方要写注释的。本章则是关于如何写出言简意赅的注释。

如果你要写注释，最好把它写得精确——越明确和细致越好。另外，由于注释在屏幕上也要占很多的地方，并且需要花更多的时间来读，因此，注释也需要很紧凑。

关键思想

注释应当有很高的信息/空间率。

本章其余部分将举例说明如何做到这一点。

让注释保持紧凑

下面的例子是一个C++类型定义的注释：

```
// The int is the CategoryType.
// The first float in the inner pair is the 'score',
// the second is the 'weight'.
typedef hash_map<int, pair<float, float> > ScoreMap;
```

可是为什么解释这个例子要用三行呢？用一行不就可以了吗？

```
// CategoryType -> (score, weight)
typedef hash_map<int, pair<float, float> > ScoreMap;
```

的确有些注释要占用三行那么多的空间，但这个不需要。

避免使用不明确的代词

就像经典的美国相声《谁在一垒》（Who's on First？）一样，代词可能会让事情变得令人困惑。

读者要花更多的工夫来“解读”一个代词。在有些情况下，“it”或者“this”到底指代什么是不清楚的。看下面这个例子：

```
// Insert the data into the cache, but check if it's too big first.
```

在这段注释中，“it”可能指数据也可能是指缓存。可能在读完剩下的代码后你会找到答案。但如果你必须这么做，又要注释干什么呢？

最安全的方式是，如果在有可能会造成困惑的地方把“填写”代词。在前一个例子中，假设“it”是指“data”，那么：

```
// Insert the data into the cache, but check if the data is too big first.
```

这是最简单的改进方法。你也可以重新组织这个句子来让“it”变得很明确：

```
// If the data is small enough, insert it into the cache.
```

润色粗糙的句子

在很多情况下，让注释更精确的过程总是伴随着让注释更紧凑。

下面是一个网页爬虫的例子：

```
# Depending on whether we've already crawled this URL before, give it a different
priority.
```

这个句子看上去可能没什么问题，但如果和下面这个版本相比呢？

```
# Give higher priority to URLs we've never crawled before.
```

后一个句子更简单、更小巧并且更直接。它同时还解释了未曾爬到过的URL将得到较高的优先级——前面那条注释没有包含这部分信息。

精确地描述函数的行为

假设你刚写了一个函数，它统计一个文件中的行数：

```
// Return the number of lines in this file.
int CountLines(string filename) { ... }
```

上面的注释并不是很精确，因为有很多定义“行”的方式。下面列出几个特别的情况：

- ""（空文件）——0或1行？
- "hello"——0或1行？
- "hello\n"——1或2行？
- "hello\n world"——1或2行？
- "hello\n\r world\r"——2、3或4行？

最简单的实现方法是统计换行符（\n）的个数（这就是Unix命令wc的工作原理）。下面的注释对于这种实现方法更好一些：

```
// Count how many newline bytes ('\n') are in the file.
int CountLines(string filename) { ... }
```

这条注释并没有比第一个版本长很多，但包含更多信息。它告诉读者如果没有换行符，这个函数会返回0。它还告诉读者回车符（\r）会被忽略。

用输入/输出例子来说明特别的情况

对于注释来讲，一个精心挑选的输入/输出例子比千言万语还有效。

例如，下面是一个用来移除部分字符串的通用函数：

```
// Remove the suffix/prefix of 'chars' from the input 'src'.
String Strip(String src, String chars) { ... }
```

这条注释不是很精确，因为它不能回答下列问题：

- chars是整个要移除的子串，还是一组无序的字母？
- 如果在src的结尾有多个chars会怎样？

然而一个精心挑选的例子就可以回答这些问题：

```
// ...
// Example: Strip("abba/a/ba", "ab") returns "/a/"
String Strip(String src, String chars) { ... }
```

这个例子展示了Strip()的整个功能。请注意，如果一个更简单的示例不能回答这些问题的话，它就不会那么有用：

```
// Example: Strip("ab", "a") returns "b"
```

下面是另一个函数的例子，也能说明这个用法：

```
// Rearrange 'v' so that elements < pivot come before those >= pivot;
// Then return the largest 'i' for which v[i] < pivot (or -1 if none are < pivot)
int Partition(vector<int>* v, int pivot);
```

这段注释实际上很精确，但是不直观。可以用下面的例子来进一步解释：

```
// ...
// Example: Partition([8 5 9 8 2], 8) might result in [5 2 | 8 9 8] and return 1
int Partition(vector<int>* v, int pivot);
```

对于我们所选择的特别的输入/输出例子，有以下几点值得提一下：

- pivot与向量中的元素相等，用来解释边界情况。
- 我们在向量中放入重复元素（8）来说明这是一种可以接受的输入。
- 返回的向量没有排序——如果是排好序的，读者可能会误解。
- 因为返回值是1，我们要确保1不是向量中的值——否则会让人很困惑。

声明代码的意图

正如我们在前一章中提到的，很多时候注释的作用就是要告诉读者当你写代码时你是怎么想的。遗憾的是，很多注释只描述代码字面上的意思，没有包含多少新信息。

下面的例子就是一条这样的注释：

```
void DisplayProducts(list<Product> products) {
    products.sort(CompareProductByPrice);

    // Iterate through the list in reverse order
    for (list<Product>::reverse_iterator it = products.rbegin(); it != products.rend();
        ++it)
        DisplayPrice(it->price);
    ...
}
```

这里的注释只是描述了它下面的那行代码。相反，更好的注释可以是这样的：

```
// Display each price, from highest to lowest
for (list<Product>::reverse_iterator it = products.rbegin(); ... )
```

这条注释从更高的层次解释了这段程序在做什么。这更符合程序员写这段代码时的想法。

有趣的是，这段程序中有一个bug！函数CompareProductByPrice（例子中没有给出）已经把高价的项目排在了前面。这段代码所做的事情与作者的意图相反。

这是第二种注释更好的原因。除了这个bug，第一条注释从技术上讲是正确的（循环进行的确是反向遍历）。但是有了第二条注释，读者更可能会注意到作者的意图（先显示高价项目）与代码实际所做的有冲突。其效果是，这条注释扮演了冗余检查的角色。

最终来讲，最好的冗余检查是单元测试（参见第14章）。但是在你的程序中写这种解释意图的注释仍是值得的。

“具名函数参数”的注释

假设你见到下面这样的函数调用：

```
Connect(10, false);
```

因为这里传入的整数和布尔型值，使得这个函数调用有点难以理解。

在像Python这样的语言中，你可以按名字为参数赋值：

```
def Connect(timeout, use_encryption):  ...
```

```
# Call the function using named parameters
Connect(timeout = 10, use_encryption = False)
```

在像C++和Java这样的语言中，你不能这样做。然而，你可以通过嵌入的注释达到同样的效果：

```
void Connect(int timeout, bool use_encryption) { ... }

// Call the function with commented parameters
Connect(/* timeout_ms = */ 10, /* use_encryption = */ false);
```

请注意我们给第一个参数起名为timeout_ms而不是timeout。从理想角度来讲，如果函数的实际参数是timeout_ms就好了，但如果因为某些原因我们无法做到这种改变，这也是“改进”这个名字的一个便捷的方法。

对于布尔参数来讲，在值的前面加上/* name = */尤其重要。把注释写在值的后面让人困惑：

```
//不要这样做
Connect( ... , false /* use_encryption */);

// 也不要这样做
Connect( ... , false /* = use_encryption */);
```

在上面这些例子中，我们不清楚false的含义是“使用加密”还是“不使用加密”。

大多数函数不需要这样的注释，但这种方法可以方便（而且紧凑）地解释看上去难以理解的参数。

采用信息含量高的词

一旦你写了多年程序以后，你会发现有些普遍的问题和解决方案会重复出现。通常会有专门的词或短语来描述这种模式/定式。使用这些词会让你的注释更加紧凑。

例如，假设你原来的注释是这样的：

```
// This class contains a number of members that store the same information as in the
// database, but are stored here for speed. When this class is read from later, those
// members are checked first to see if they exist, and if so are returned; otherwise the
// database is read from and that data stored in these field for next time.
```

那么你可以简单地说：

```
// This class acts as a caching layer to the database.
```

另一个注释的例子：

```
// Remove excess whitespace from the street address, and do lots of other cleanup
// like turn "Avenue" into "Ave." This way, if there are two different street addresses
```

```
// that are typed in slightly differently, they will have the same cleaned-up version and
// we can detect that these are equal.
```

可以写成：

```
// Canonicalize the street address (remove extra spaces, "Avenue" -> "Ave.", etc.)
```

很多词和短语都具有多种含义，例如“heuristic”、“bruteforce”、“naive solution”等。如果你感觉到一段注释太长了，那么可以看看是不是可以用一个典型的编程场景来描述它。

总结

本章是关于如何把更多的信息装入更小的空间里。下面是一些具体的提示：

- 当像“it”和“this”这样的代词可能指代多个事物时，避免使用它们。
- 尽量精确地描述函数的行为。
- 在注释中用精心挑选的输入/输出例子进行说明。
- 声明代码的高层次意图，而非明显的细节。
- 用嵌入的注释（如Function(/*arg =*/...)）来解释难以理解的函数参数。
- 用含义丰富的词来使注释简洁。

第二部分

简化循环和逻辑

第一部分介绍了表面层次的改进，那是一些改进代码可读性的简单方法，一次一行，在没有很大的风险或者花很大代价的情况下就可以应用。

第二部分将进一步深入讨论程序的“循环和逻辑”：控制流、逻辑表达式以及让你的代码正常运行的那些变量。和第一部分的要求一样，我们的目标是让代码中的这些部分容易理解。

我们通过试着最小化代码中的“思维包袱”来达到目的。每当你看到一个复杂的逻辑、一个巨大的表达式或者一大堆变量，这些都会增加你头脑中的思维包袱。它需要让你考虑得更复杂并且记住更多事情。这恰恰与“容易理解”相反。当代码中有很多思维包袱时，很可能在不知不觉中就会产生bug，代码会变得难以改变，并且使用它也没那么有趣了。

第7章

把控制流变得易读

如果代码中没有条件判断、循环或者任何其他的控制流语句，那么它的可读性会很好。而跳转和分支等困难部分则会很快地让代码变得混乱。本章就是关于如何把代码中的控制流变得易读的。

关键思想

把条件、循环以及其他对控制流的改变做得越“自然”越好。运用一种方式使读者不用停下来重读你的代码。

条件语句中参数的顺序

下面的两段代码哪个更易读？

```
if (length >= 10)
```

还是

```
if (10 <= length)
```

对大多数程序员来讲，第一段更易读。那么，下面的两段呢？

```
while (bytes_received < bytes_expected)
```

还是

```
while (bytes_expected > bytes_received)
```

仍然是第一段更易读。可为什么会这样？通用的规则是什么？你怎么才能决定是写成a<b好一些，还是写成b>a好一些？

下面的这条指导原则很有帮助：

比较的左侧	比较的右侧
“被问询的”表达式，它的值更倾向于不断变化	用来做比较的表达式，它的值更倾向于常量

这条指导原则和英语的用法一致[译注1]。我们会很自然地说：“如果你的年收入至少是10万美元”或者“如果你不小于18岁。”而“如果18岁小于或等于你的年龄”这样的说法却很少见。

这也解释了为什么while(bytes_received < bytes_expected)有更好的可读性。bytes_received是我们在检查的值，并且在循环的执行中它在增长。当用来做比较时，byte_expected则是更“稳定”的那个值。

译注1： 当然，和中文的习惯也一致。

“尤达表示法”：还有用吗？

在有些语言中（包括C和C++，但不包括Java），可以把赋值操作放在if条件中：

```
if (obj = NULL) ...
```

这极有可能是个bug，程序员本来的意图是：

```
if (obj == NULL) ...
```

为了避免这样的bug，很多程序员**把参数的顺序调换一下**：

```
if (NULL == obj) ...
```

这样，如果把==误写为=，那么表达式if(NULL = obj)连编译也通不过。

遗憾的是，这种顺序的改变使得代码读起来很不自然（就像电影《星球大战》里的尤达大师的语气：“除非对此有话可说之于我”）。庆幸的是，现代编译器对if(obj = NULL)这样的代码会给出警告，因此“尤达表示法”是已经过时的事情了。

if/else语句块的顺序

在写if/else语句时，你通常可以自由地变换语句块的顺序。例如，你既可以写成：

```
if (a == b) {
    // Case One ...
} else {
    // Case Two ...
}
```

也可以写成：

```
if (a != b) {
    // Case Two ...
} else {
    // Case One ...
}
```

之前你可能没想过太多，但在有些情况下有理由相信其中一种顺序比另一种好：

- 首先处理正逻辑而不是负逻辑的情况。例如，用if(debug)而不是if(!debug)。
- 先处理掉简单的情况。这种方式可能还会使得if和else在屏幕之内都可见，这很好。
- 先处理有趣的或者是可疑的情况。

有时这些倾向性之间会有冲突，那么你就要自己判断了。但在很多情况下这都会有明确的选择。

例如，假设你有一个Web服务器，它会根据URL是否包含查询参数expand_all来建构一个response：

```
if (!url.HasQueryParameter("expand_all")) {
    response.Render(items);
    ...
} else {
    for (int i = 0; i < items.size(); i++) {
        items[i].Expand();
    }
    ...
}
```

当读者刚看到第一行代码时，他的脑海中马上开始思考expand_all的情况。这就像当有人说“不要去想一头粉红色的大象”时，你会不由自主地去想。“不要”这个词已经被更不寻常的“粉红色的大象”给淹没了。

这里，expand_all就是我们的“粉红色的大象”。让我们先来处理这种情况，因为它更有趣（并且也是正逻辑）：

```
if (url.HasQueryParameter("expand_all")) {
   for (int i = 0; i < items.size(); i++) {
       items[i].Expand();
```

```
    }
    ...
} else {
    response.Render(items);
    ...
}
```

另外，下面所示是负逻辑更简单并且更有趣或更危险的一种情况，那么会先处理它：

```
if not file:
    # Log the error ...
else:
    # ...
```

同样，根据具体情况的不同，这也是需要你自己来判断的。

作为小结，我们的建议很简单，就是要注意这些因素并且小心那些会使你的if/else顺序很别扭的情况。

?:条件表达式（又名“三目运算符”）

在类C的语言中，可以把一个条件表达式写成cond ? a : b这样的形式，其实就是一种对if (cond) { a } else {b }的紧凑写法。

它对于可读性的影响是富有争议的。拥护者认为这种方式可以只写一行而不用写成多行。反对者则说这可能会造成阅读的混乱而且很难用调试器来调试。

下面是一个三目运算符易读而又紧凑的应用：

```
time_str += (hour >= 12) ? "pm" : "am";
```

要避免三目运算符，你可能要这样写：

```
if (hour >= 12) {
    time_str += "pm";
} else {
    time_str += "am";
}
```

这有点冗长了。在这种情况下使用条件表达式似乎是合理的。然而，这种表达式可能很快就会变得很难读：

```
return exponent >= 0 ? mantissa * (1 << exponent) : mantissa / (1 << -exponent);
```

在这里，三目运算符已经不只是从两个简单的值中做出选择。写出这种代码的动机往往是“把所有的代码都挤进一行里”。

关键思想

相对于追求最小化代码行数，一个更好的度量方法是最小化人们理解它所需的时间。

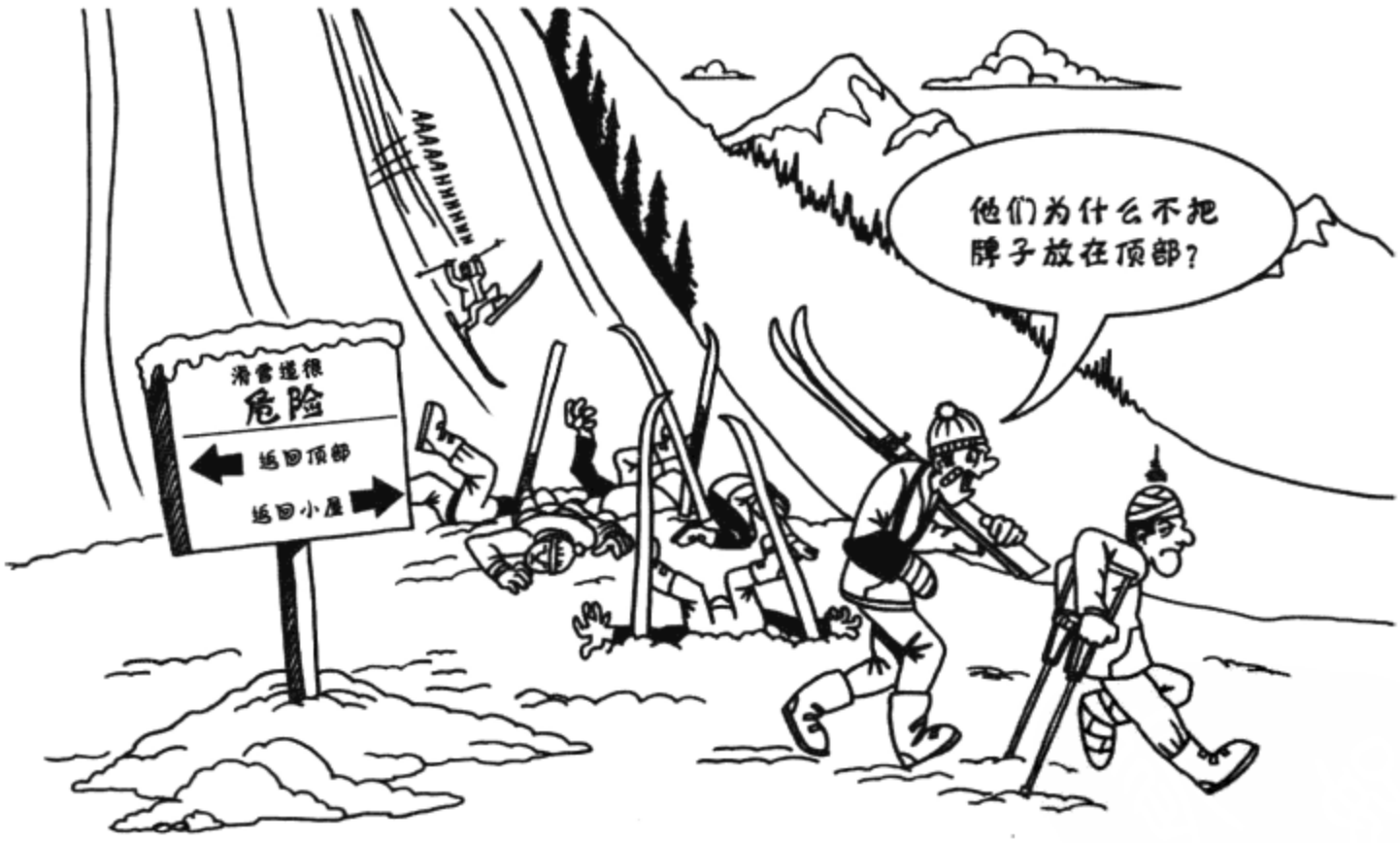
用if/else语句阐明逻辑可以使代码更自然：

```
if (exponent >= 0) {
    return mantissa * (1 << exponent);
} else {
    return mantissa / (1 << -exponent);
}
```

建议

默认情况下都用if/else。三目运算符?:只有在最简单的情况下使用。

避免do/while循环



很多推崇的编程语言，包括Perl，都有do {expression} while (condition)循环。其中的表达式至少会执行一次。下面举个例子：

```
// 在列表中从"node"开始查找给出的"name"节点。
// 不用考虑超出"max_length"的节点。
public boolean ListHasNode(Node node, String name, int max_length) {
    do {
        if (node.name().equals(name))
```

```
            return true;
        node = node.next();
    } while (node != null && --max_length > 0);

    return false;
}
```

do/while的奇怪之处是一个代码块是否会执行是由其后的一个条件决定的。通常来讲，逻辑条件应该出现在它们所“保护”的代码之前，这也是if. while和for语句的工作方式。因为你通常会从前向后来读代码，这就使得do/while循环有点不自然了。很多读者最后会读这段代码两遍。

while循环相对更易读，因为你会先读到所有迭代的条件，然后再读到其中的代码块。但仅仅是为了去掉do/while循环而重复一段代码是有点愚蠢的做法：

```
// 机械地模仿do/while循环——不要这样做！
body

while (condition) {
    body (again)
}
```

幸运的是，我们发现实践当中大多数的do/while循环都可以写成这样开头的while循环：

```
public boolean ListHasNode(Node node, String name, int max_length) {
    while (node != null && max_length-- > 0) {
        if (node.name().equals(name)) return true;
        node = node.next();
    }
    return false;
}
```

这个版本还有一个好处是对于max_length是0或者node是null的情况它仍然可以工作。

另一个要避免do/while循环的原因是其中的continue语句会很让人迷惑。例如，下面这段代码会做什么？

```
do {
    continue;
} while (false);
```

它会永远循环下去还是只执行一次？大多数程序员都不得不停下来想一想。（它只会循环一次。）

最后，C++的开创者Bjarne Stroustrup讲得好（在《C++程序设计语言》[编注1]一书中）：

> 我的经验是，do语句是错误和困惑的来源……我倾向于把条件放在“前面我能看到的地方”。其结果是，我倾向于避免使用do语句。

编注1：由机械工业出版社引进并出版，英文书名为《The C++ Programming Language》。

从函数中提前返回

有些程序员认为函数中永远不应该出现多条return语句。这是胡说八道。从函数中提前返回没有问题，而且常常很受欢迎。例如：

```
public boolean Contains(String str, String substr) {
   if (str == null || substr == null) return false;
   if (substr.equals("")) return true;
   ...
}
```

如果不用“保护语句”（guard clause）来实现这种函数将会很不自然。

想要单一出口点的一个动机是保证调用函数结尾的清理代码。但现代的编程语言为这种保证提供了更精细的方式：

语言	清理代码的结构化术语
C++	析构函数
Java、Python	try finally
Python	with
C#	using

在单纯由C语言组成的代码中，当函数退出时没有任何机制来触发特定的代码。因此，如果一个大函数有很多清理代码，提前返回可能很难做得没有问题。在这种情况下，其他的选择包括重构函数，甚至慎重地使用goto cleanup;。

臭名昭著的goto

除了C语言之外，其他语言一般不大需要goto，因为有太多更好的方式能完成同样的工作。同时goto也因为草草了事使代码难以理解而声名狼藉。

但是你还是会在各种C项目中见到对goto的使用，最值得注意的就是Linux内核。在你认定所有对goto的使用都是一种亵渎之前，仔细研究为什么某些对goto的使用比其他更好将会大有帮助。

对goto最简单、最单纯的使用就是在函数结尾有单个exit：

```
        if (p == NULL) goto exit;

        ...

    exit:
        fclose(file1);
        fclose(file2);
        ...
```

```
        return;
```

如果只允许出现这一种goto的形式，goto不会成为什么大问题。

当有多个goto的目标时可能就会有问题了，尤其当这些路径交叉时。需要特别指出的是，向前goto可能会产生真正的意大利面条式代码，并且它们肯定可以被结构化的循环替代。大多数时候都应该避免使用goto。

最小化嵌套

嵌套很深的代码很难以理解。每个嵌套层次都在读者的“思维栈”上又增加了一个条件。当读者见到一个右大括号（}）时，可能很难“出栈”来回忆起它背后的条件是什么。

下面是一个相对简单的例子——当你回头复查你在读的是哪一个条件语句块时，你是否能注意到你自己：

```
if (user_result == SUCCESS) {
   if (permission_result != SUCCESS) {
       reply.WriteErrors("error reading permissions");
       reply.Done();
       return;
   }
   reply.WriteErrors("");
} else {
   reply.WriteErrors(user_result);

}
reply.Done();
```

当你看到第一个右大括号时，你不得不去想：“哦，permission_result != SUCCESS刚刚结束，那么现在是在permission_result == SUCCESS之中了，并且还是在user_result == SUCCESS的语句块中。”

总之，你不得不始终记得user_result和permission_result的值。并且当每个if{}块结束后你都不得不切换你脑海中的值。

上例中的代码尤其不好，因为它不断地切换SUCCESS和non-SUCCESS的条件。

嵌套是如何累积而成的

在我们修正前面的示例代码之前，先来看看是什么导致它成了现在的样子。一开始，代码是很简单的：

```
if (user_result == SUCCESS) {
   reply.WriteErrors("");
```

```
} else {
   reply.WriteErrors(user_result);
}
reply.Done();
```

这段代码很容易理解——它找出该写什么错误信息，然后回复并结束。

但是后来那个程序员增加了第二个操作：

```
if (user_result == SUCCESS) {
   if (permission_result != SUCCESS) {
       reply.WriteErrors("error reading permissions");
       reply.Done();
       return;
   }
   reply.WriteErrors("");

...
```

这个改动有合理的地方——该程序员要插入一段新代码，并且她找到了最容易插入的地方。对于她来讲，新代码很整洁，而且很明确。这个改动的差异也很清晰——这看上去像是个简单的改动。

但是以后当其他人遇到这段代码时，所有的上下文早已不在了。这就是你在本节一开始读到这段代码时的情况，你不得不一下子全盘接受它。

关键思想

当你对代码做改动时，从全新的角度审视它，把它作为一个整体来看待。

通过提早返回来减少嵌套

好的，那么让我们来改进这段代码。像这种嵌套可以通过马上处理“失败情况”并从函数早返回来减少：

```
if (user_result != SUCCESS) {
   reply.WriteErrors(user_result);
   reply.Done();
   return;
}

if (permission_result != SUCCESS) {
   reply.WriteErrors(permission_result);
   reply.Done();
   return;
}

reply.WriteErrors("");
reply.Done();
```

上面这段代码只有一层嵌套，而不是两层。但更重要的是，读者不再需要从思维堆栈里“出栈”了——每个if块都以一个return结束。

减少循环内的嵌套

提早返回这个技术并不总是合适的。例如，下面代码在循环中有嵌套：

```
for (int i = 0; i < results.size(); i++) {
   if (results[i] != NULL) {
       non_null_count++;
       if (results[i]->name != "") {
           cout << "Considering candidate..." << endl;

           ...
       }
   }
}
```

在循环中，与提早返回类似的技术是continue：

```
for (int i = 0; i < results.size(); i++) {
   if (results[i] == NULL) continue;
   non_null_count++;

   if (results[i]->name == "") continue;
   cout << "Considering candidate..." << endl;

   ...
}
```

与if(...)return;在函数中所扮演的保护语句一样，这些if(...) continue;语句是循环中的保护语句。

一般来讲，continue语句让人很困惑，因为它让读者不能连续地阅读，就像循环中有goto语句一样。但是在这种情况中，循环中的每个迭代是相互独立的（这是一种“for each”循环），因此读者可以很容易地领悟到这里continue的意思就是“跳过该项”。

你能理解执行的流程吗

本章介绍低层次控制流：如何把循环、条件和其他跳转写得简单易读。但是你也应该从高层次来考虑程序的“流动”。理想的情况是，整个程序的执行路径都很容易理解——从main()开始，然后在脑海中一步步执行代码，一个函数调用另一个函数，直到程序结束。

然而在实践中，编程语言和库的结构让代码在“幕后”运行，或者让流程难以理解。下面是一些例子：

编程结构	高层次程序流程是如何变得不清晰的
线程	不清楚什么时间执行什么代码
信号量/中断处理程序	有些代码随时都有可能执行
异常	可能会从多个函数调用中向上冒泡一样地执行
函数指针和匿名函数	很难知道到底会执行什么代码，因为在编译时还没有决定
虚方法	object.virtualMethod()可能会调用一个未知子类的代码

这些结构中有些很有用，它们甚至可以让你的代码更具可读性，并且冗余更少。但是作为程序员，有时候我们得意忘形了，于是用得太多了，却没有发现以后它会多么令人难以理解。并且，这些结构使得更难以跟踪bug。

关键是不要让代码中使用这些结构的比例太高。如果你滥用这些功能，它可能会让跟踪代码像三牌赌博游戏（像卡通画中一样）。

总结

有几种方法可以让代码的控制流更易读。

在写一个比较时（`while (bytes_expected > bytes_received)`），把改变的值写在左边并且把更稳定的值写在右边更好一些（`while (bytes_received < bytes_expected)`）。

你也可以重新排列`if/else`语句中的语句块。通常来讲，先处理正确的/简单的/有趣的情况。有时这些准则会冲突，但是当不冲突时，这是要遵循的经验法则。

某些编程结构，像三目运算符（`:?`）、`do/while`循环，以及`goto`经常会导致代码的可读性变差。最好不要使用它们，因为总是有更整洁的代替方式。

嵌套的代码块需要更加集中精力去理解。每层新的嵌套都需要读者把更多的上下文“压入栈”。应该把它们改写成更加“线性”的代码来避免深嵌套。

通常来讲提早返回可以减少嵌套并让代码整洁。“保护语句”（在函数顶部处理简单的情况时）尤其有用。

第8章

拆分超长的表达式

巨型乌贼是一种神奇而又聪明的动物，但它近乎完美的身体设计有一个致命的弱点：在它的食管附近围绕着圆环形的大脑。所以如果它一次吞太多的食物，它的大脑会受到伤害。

这和代码有什么关系？嗯，大段大段的代码也可能会造成类似的效果。最近有研究表明，我们大多数人同时只能考虑3～4件“事情”[注1]。简单地说，代码中的表达式越长，它就越难以理解。

关键思想

把你的超长表达式拆分成更容易理解的小块。

在本章中，我们会看看各种可以操作和拆分代码以使它们更容易理解的方法。

用做解释的变量

拆分表达式最简单的方法就是引入一个额外的变量，让它来表示一个小一点的子表达式。这个额外的变量有时叫做“解释变量”，因为它可以帮助解释子表达式的含义。

下面是一个例子：

```
if line.split(':')[0].strip() == "root":
    ...
```

下面是和上面同样的代码，但是现在有了一个解释变量。

```
username = line.split(':')[0].strip()
if username == "root":
    ...
```

总结变量

即使一个表达式不需要解释（因为你可以看出它的含义），把它装入一个新变量中仍然有用。我们把它叫做总结变量，它的目的只是用一个短很多的名字来代替一大块代码，这个名字会更容易管理和思考。

例如，看看下面代码中的表达式。

```
if (request.user.id == document.owner_id) {
    // user can edit this document...
}
...
if (request.user.id != document.owner_id) {
```

注1：Cowan, N.(2001). The magic number 4 in short-term memory: A reconsideration of mental storage capacity. *Behavioral and Brain Sciences*, 24, 97-185.

```
    // document is read-only...
}
```

这里的表达式request.user.id == document.owner_id看上去可能并不长，但它包含5个变量，所以需要多花点时间来想一想如何处理它。

这段代码中的主要概念是："该用户拥有此文档吗？"这个概念可以通过增加一个总结变量来表达得更清楚。

```
final boolean user_owns_document = (request.user.id == document.owner_id);

if (user_owns_document) {
    // user can edit this document...
}

...

if (!user_owns_document) {
    // document is read-only...
}
```

上面的代码看上去改动并不大，但语句if (user_owns_document)更容易理解一些。并且，在一开始就定义了user_owns_document，用于提前告诉读者"这是在整个函数中都会引用的一个概念"。

使用德摩根定理

如果你学过"电路"或者"逻辑"课，你应该还记得德摩根定理。对于一个布尔表达式，有两种等价的写法：

```
1) not (a or b or c) <=> (not a) and (not b) and (not c)
2) not (a and b and c) <=> (not a) or (not b) or (not c)
```

如果你记不住这两条定理，一个简单的小结是"分别取反，转换与/或"（反向操作是"提出取反因子"）。

有时，你可以使用这些法则来让布尔表达式更具可读性。例如，如果你的代码是这样的：

```
if (!(file_exists && !is_protected)) Error("Sorry, could not read file.");
```

那么可以把它改写成：

```
if (!file_exists || is_protected) Error("Sorry, could not read file.");
```

滥用短路逻辑

在很多编程语言中，布尔操作会做短路计算。例如，语句if (a || b)在a为真时不会计算b。使用这种行为很方便，但有时可能会被滥用以实现复杂逻辑。

下面例子中的语句当初是由某一位作者写的：

```
assert((!(bucket = FindBucket(key))) || !bucket->IsOccupied());
```

用英语来讲，这段代码是在说：“得到key的bucket。如果这个bucket不是空，那么确定它是不是已经被占用。”

尽管它只有一行代码，但是它的确要让大多数程序员停下来想一想才行。现在和下面的代码比一比：

```
bucket = FindBucket(key);
if (bucket != NULL) assert(!bucket->IsOccupied());
```

它做的事情完全一样，尽管它有两行代码，但它要容易理解得多。

那么无论如何为什么要把代码写在一个巨大的表达式里呢？在当时，它看上去很智能。把逻辑解析成一小段简明的码段。这可以理解——这就像在猜一个小小的谜，我们都想让工作有乐趣。问题是这种代码对于任何读它的人来讲都是个思维上的减速带。

关键思想

要小心“智能”的小代码段——它们往往在以后会让别人读起来感到困惑。

这是否意味着你应该避免利用这种短路行为？不是的。在很多情况下可以用它达到整洁的目的，例如：

```
if (object && object->method()) ...
```

还有一个比较新的习惯用法值得一提：在像Python、JavaScript以及Ruby这样的语言中，“or”操作符会返回其中一个参数（它不会转换成布尔值），所以这样的代码：

```
x = a || b || c
```

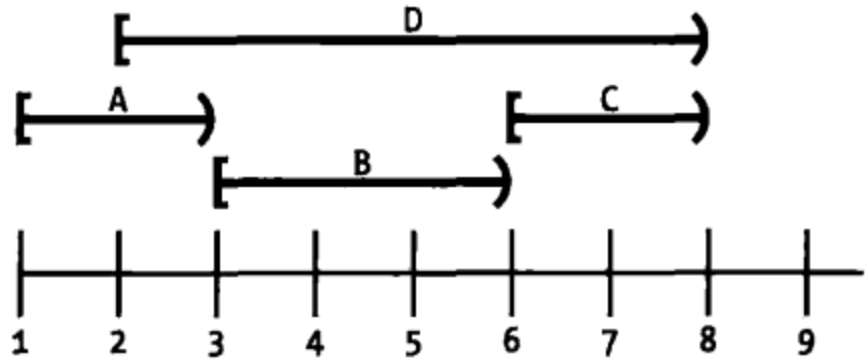
可以用来从a、b或c中找出第一个为“真”的值。

例子：与复杂的逻辑战斗

假设你在实现下面这个Range类：

```
struct Range {
   int begin;
   int end;
   // For example, [0,5) overlaps with [3,8)
   bool OverlapsWith(Range other);
};
```

下图给出一些范围的例子：

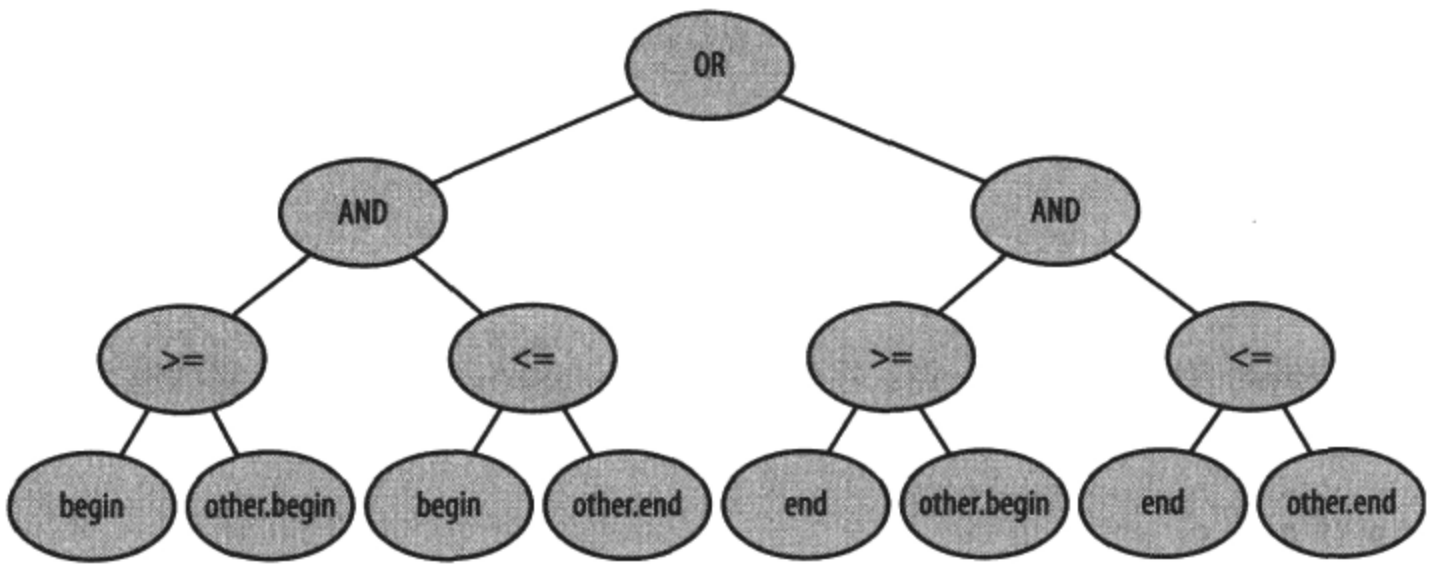


请注意终点是非包含的。因此*A*、*B*和*C*互相之间不会有重复，但是*D*与所有其他重复。

下面是对`OverlapsWith()`实现的一个尝试——它检查是否自身范围的任意一个端点在`other`的范围之内：

```
bool Range::OverlapsWith(Range other) {
   // Check if 'begin' or 'end' falls inside 'other'.
   return (begin >= other.begin && begin <= other.end) ||
          (end >= other.begin && end <= other.end);
}
```

尽管只有两行代码，但是里面包含很多东西。下图给出其中所有的逻辑。



这里面有太多的情况和条件要去考虑，这很容易滋生bug。

说到这儿，这里还真有一个bug。前面的代码会认为`Range[0,2)`与`Range[2,4)`重复，而实际上它们并不重复。

这里的问题是在比较`begin/end`值时要小心地使用`<=`或`<`。下面是对这个问题的修正：

```
return (begin >= other.begin && begin < other.end) ||
       (end > other.begin && end <= other.end);
```

现在已经改正了，是吗？实际上，还有另一个bug。这段代码忽略了`begin/end`完全包含`other`的情况。

下面是处理这种情况的修改：

```
return (begin >= other.begin && begin < other.end) ||
       (end > other.begin && end <= other.end) ||
       (begin <= other.begin && end >= other.end);
```

现在代码变得太复杂了。你不可能指望别人看了这段代码就对它的正确性有信心。那么我们该怎么办？怎么拆分这个大的表达式呢？

找到更优雅的方式

这就是那种你该停下来从整体上考虑不同方式的时机之一。开始还很简单的问题（检查两个范围是否重叠）变得非常令人费解。这通常预示着肯定有一种更简单的方法。

但是找到更优雅的方式需要创造力。那么怎么做呢？一种技术是看看能否从“反方向”解决问题。根据你所处的不同情形，这可能意味着反向遍历数组，或者往回填充数据结构而非向前。

在这里，`OverlapsWith()`的反方向是“不重叠”。判断两个范围是否不重叠原来更简单，因为只有两种可能：

1. 另一个范围在这个范围开始前结束。
2. 另一个范围在这个范围结束后开始。

我们可以很容易地把它变成代码：

```
bool Range::OverlapsWith(Range other) {
    if (other.end <= begin) return false;  // They end before we begin
    if (other.begin >= end) return false;  // They begin after we end

    return true;  // Only possibility left: they overlap
}
```

这里的每一行代码都要简单得多——每行只有一个比较。这就使得读者留有足够的心力来关注`<=`是否正确。

拆分巨大的语句

本章是关于拆分独立的表达式的，但同样的技术也可以用来拆分大的语句。例如，下面的JavaScript代码需要一次读很多东西：

```
var update_highlight = function (message_num) {
   if ($("#vote_value" + message_num).html() === "Up") {
       $("#thumbs_up" + message_num).addClass("highlighted");
       $("#thumbs_down" + message_num).removeClass("highlighted");
   } else if ($("#vote_value" + message_num).html() === "Down") {
```

```
        $("#thumbs_up" + message_num).removeClass("highlighted");
        $("#thumbs_down" + message_num).addClass("highlighted");
    } else {
        $("#thumbs_up" + message_num).removeClass("highighted");
        $("#thumbs_down" + message_num).removeClass("highlighted");
    }
};
```

代码中的每个表达式并不是很长，但当把它们放在一起时，它们就形成了一条巨大的语句，迎面扑来。

幸运的是，其中很多表达式是一样的，这意味着可以把它们提取出来作为函数开头的总结变量（这同时也是一个DRY——Don't Repeat Yourself的例子）：

```
var update_highlight = function (message_num) {
    var thumbs_up = $("#thumbs_up" + message_num);
    var thumbs_down = $("#thumbs_down" + message_num);
    var vote_value = $("#vote_value" + message_num).html();
    var hi = "highlighted";

    if (vote_value === "Up") {
        thumbs_up.addClass(hi);
        thumbs_down.removeClass(hi);
    } else if (vote_value === "Down") {
        thumbs_up.removeClass(hi);
        thumbs_down.addClass(hi);
    } else {
        thumbs_up.removeClass(hi);
        thumbs_down.removeClass(hi);
    }
};
```

创建var hi = "highlighted"严格来讲不是必需的，但鉴于这里有6次重复，有很多好处驱使我们这样做：

- 它帮助避免录入错误。（实际上，你是否注意到在第一个例子中，该字符串在第5种情况中被误写成"highhighted"？）
- 它进一步缩短了行的宽度，使代码更容易快速阅读。
- 如果类的名字需要改变，只需要改一个地方即可。

另一个简化表达式的创意方法

下面是另一个例子，同样在每个表达式中都包括了很多东西，这次是用C++写的：

```
void AddStats(const Stats& add_from, Stats* add_to) {
    add_to->set_total_memory(add_from.total_memory() + add_to->total_memory());
    add_to->set_free_memory(add_from.free_memory() + add_to->free_memory());
    add_to->set_swap_memory(add_from.swap_memory() + add_to->swap_memory());
```

```
    add_to->set_status_string(add_from.status_string() + add_to->status_string());
    add_to->set_num_processes(add_from.num_processes() + add_to->num_processes());
    ...
}
```

再一次，你的眼睛要面对又长又相似的代码，但不是完全一样。在仔细检查了10秒后，你想必会发现每一行都在做同样的事，只是每次添加的字段不同：

```
add_to->set_XXX(add_from.XXX() + add_to->XXX());
```

在C++中，可以定义一个宏来实现它：

```
void AddStats(const Stats& add_from, Stats* add_to) {
  #define ADD_FIELD(field) add_to->set_##field(add_from.field() + add_to->field())

  ADD_FIELD(total_memory);
  ADD_FIELD(free_memory);
  ADD_FIELD(swap_memory);
  ADD_FIELD(status_string);
  ADD_FIELD(num_processes);
  ...
  #undef ADD_FIELD
}
```

现在我们化繁为简，你可以看一眼代码就马上理解大致的意思。很明显，每一行都在做同样的事情。

请注意，我们不鼓吹经常使用宏——事实上，我们通常避免使用宏，因为它们会让代码变得令人困惑并且引入细微的bug。但有时，就像在本例中，它们很简单而且对可读性有明显的好处。

总结

很难思考巨大的表达式。本章给出了几种拆分表达式的方法，以便读者可以一段一段地消化。

一个简单的技术是引入“解释变量”来代表较长的子表达式。这种方式有三个好处：

- 它把巨大的表达式拆成小段。
- 它通过用简单的名字描述子表达式来让代码文档化。
- 它帮助读者识别代码中的主要概念。

另一个技术是用德摩根定理来操作逻辑表达式——这个技术有时可以把布尔表达式用更整洁的方式重写（例如`if (!(a && !b))`变成`if (!a || b)`）。

本章给出了一个，把一个复杂的逻辑条件拆分成小的语句的例子，就像"if (a < b) ..."。实际上，在本章所有改进过的示例代码中，所有的if语句内都没有超过两个值。这是理想情况。可能不是总能做到这样——有时需要把问题“反向”或者考虑目标的对立面。

最后，尽管本章是关于拆分独立的表达式的，同样，这些技术也常应用于大的代码块。所以，你可以在任何见到复杂逻辑的地方大胆地去拆分它们。

第9章

变量与可读性

马戏导演的家

在本章里，你会看到对于变量的草率运用如何让程序更难理解。确切地说，我们会讨论三个问题：

1. 变量越多，就越难全部跟踪它们的动向。
2. 变量的作用域越大，就需要跟踪它的动向越久。
3. 变量改变得越频繁，就越难以跟踪它的当前值。

下面三节讨论如何处理这些问题。

减少变量

在第8章中，我们讲了如何引入“解释”或者“总结”变量来使代码更可读。这些变量很有帮助是因为它们把巨大的表达式拆分开，并且可以作为某种形式的文档。

在本节中，我们感兴趣的是减少不能改进可读性的变量。当移除这种变量后，新代码会更精练而且同样容易理解。

在下一节中的几个例子讲述这些不必要的变量是如何出现的。

没有价值的临时变量

在下面的一小段Python代码中，考虑now这个变量：

```
now = datetime.datetime.now()
root_message.last_view_time = now
```

now是一个值得保留的变量吗？不是，下面是原因：

- 它没有拆分任何复杂的表达式。
- 它没有做更多的澄清——表达式datetime.datetime.now()已经很清楚了。
- 它只用过一次，因此它并没有压缩任何冗余代码。

没有了now，代码一样容易理解。

```
root_message.last_view_time = datetime.datetime.now()
```

像now这样的变量通常是在代码编辑过后的“剩余物”。now这个变量可能从前在多个地方用到。或者可能那个程序员料想now会多次用到，但实际上再没用到过它。

减少中间结果

下面的例子是一个JavaScript函数，用来从数组中删除一个值：

```
var remove_one = function (array, value_to_remove) {
    var index_to_remove = null;
    for (var i = 0; i < array.length; i += 1) {
        if (array[i] === value_to_remove) {
            index_to_remove = i;
            break;
        }
    }
    if (index_to_remove !== null) {
        array.splice(index_to_remove, 1);
    }
};
```

变量index_to_remove只是用来保存临时结果。有时这种变量可以通过得到后立即处理它而消除。

```
var remove_one = function (array, value_to_remove) {
    for (var i = 0; i < array.length; i += 1) {
        if (array[i] === value_to_remove) {
            array.splice(i, 1);
            return;
        }
    }
};
```

通过让代码提前返回，我们不再需要index_to_remove，并且大幅简化了代码。

通常来讲，“速战速决”是一个好的策略。

减少控制流变量

有些时候你会在代码的循环中见到如下模式：

```
boolean done = false;
```

```
while (/* condition */ && !done) {
    ...

    if (...) {
        done = true;
        continue;
    }
}
```

甚至可以在循环里多处把变量done设置为true。

这样的代码通常是为了满足某些心照不宣的规则，即你不该从循环中间跳出去。根本就没有这样的规则！

像done这样的变量，称为“控制流变量”。它们唯一的目的就是控制程序的执行——它们没有包含任何程序的数据。在我们的经验中，控制流变量通常可以通过更好地运用结构化编程而消除。

```
while (/* condition */) {
    ...
    if (...) {
        break;
    }
}
```

这个例子改起来很简单，但是如果有多个嵌套循环，一个简单的break根本不够怎办呢？在这种更复杂的情况下，解决方案通常包括把代码挪到一个新函数中（要么是循环中的代码，要么是整个循环）。

你希望你的同事随时都觉得是在面试吗？

来自微软的Eric Brechner曾说过一个好的面试问题起码要涉及三个变量。[注1] 可能是因为同时处理三个变量会强迫你努力思考！这对于面试来讲还说得过去，因为你要尝试找到候选人的极限。但是你希望你的同事在读你的代码时感觉就像你在面试他们吗？

缩小变量的作用域

我们都听过“避免全局变量”这条建议。这是一条好的建议，因为很难跟踪这些全局变量在哪里以及如何使用它们。并且通过“命名空间污染”（名字太多容易与局部变量冲突），代码可能会意外地改变全局变量的值，虽然本来的目的是使用局部变量，或者反过来也有同样的效果。

注1：《代码之道》英文书名《Hard Code》，由机械工业出版社引进并出版，作者 Eric Brechner。

实际上，让所有的变量都“缩小作用域”是一个好主意，并非只是针对全局变量。

关键思想

让你的变量对尽量少的代码行可见。

很多编程语言提供了多重作用域/访问级别，包括模块、类、函数以及语句块作用域。通常越严格的访问控制越好，因为这意味着该变量对更少的代码行“可见”。

为什么要这么做？因为这样有效地减少了读者同时需要考虑的变量个数。如果你能把所有的变量作用域都减半，那么这就意味着同时需要思考的变量个数平均来讲是原来的一半。

例如，假设你有一个很大的类，其中有一个成员变量只由两个方法用到，使用方式如下：

```
class LargeClass {
    string str_;

    void Method1() {
        str_ = ...;
        Method2();
    }

    void Method2() {
        // Uses str_
    }

    // Lots of other methods that don't use str_ ...
};
```

从某种意义上来讲，类的成员变量就像是在该类的内部世界中的“小型全局变量”。尤其对大的类来讲，很难跟踪所有的成员变量以及哪个方法修改了哪个变量。这样的小型全局变量越少越好。

在本例中，最好把str_“降格”为局部变量：

```
class LargeClass {
    void Method1() {
        string str = ...;
        Method2(str);
    }

    void Method2(string str) {
        // Uses str
    }

    // Now other methods can't see str.
};
```

另一个对类成员访问进行约束的方法是“尽量使方法变成静态的”。静态方法是让读者知道“这几行代码与那些变量无关”的好办法。

或者还有一种方式是“把大的类拆分成小一些的类”。这种方法只有在这些小一些的类事实上相互独立时才能发挥作用。如果你只是创建两个类来互相访问对方的成员，那你什么目的也没达到。

把大文件拆分成小文件，或者把大函数拆分成小函数也是同样的道理。这么做的一个重要的动机就是数据（即变量）分离。

但是不同的语言有不同的管理作用域的规则。我们接下来给出一些与变量作用域相关的更有趣规则。

C++中if语句的作用域

假设你有以下C++代码：

```
PaymentInfo* info = database.ReadPaymentInfo();
if (info) {
    cout << "User paid: " << info->amount() << endl;
}

// Many more lines of code below ...
```

变量info在此函数的余下部分仍在作用域内，因此，读这段代码的人要始终记得它，猜测它是否或者怎样再次用到。

但是在本例中，info只有在if语句中才用到。在C++语言中，我们实际上可以把info定义在条件表达式中：

```
if (PaymentInfo* info = database.ReadPaymentInfo()) {
    cout << "User paid: " << info->amount() << endl;
    }
```

现在读者可以在info超出作用域后放心地忘掉它了。

在JavaScript中创建“私有”变量

假设你有一个长期存在的变量，只有一个函数会用到它：

```
submitted = false; // Note: global variable

var submit_form = function (form_name) {
    if (submitted) {
        return;  // don't double-submit the form
    }
    ...
```

```
        submitted = true;
    };
```

像submitted这种全局变量会让读代码的人非常不安。看上去好像只有submit_form()使用submitted，但你就是没办法确定。实际上，另一个JavaScript文件可能也在用一个叫submitted的全局变量，却不是为了同一个目的！

你可以把submitted放在一个“闭包”中来避免这个问题：

```
var submit_form = (function () {
    var submitted = false; // Note: can only be accessed by the function below

    return function (form_name) {
        if (submitted) {
            return;  // don't double-submit the form
        }
        ...
        submitted = true;
    };
}());
```

请注意在最后一行上的圆括号——它会使外层的这个匿名函数立即执行，返回内层的函数。

如果你以前没见过这种技巧，可能一开始它看上去有些怪。它的效果是营造一个“私有”作用域，只有内层函数才能访问。现在读者不必再去猜“submitted还在什么地方用到了？”或者担心与其他同名的全局变量冲突。（这方面的更多技巧，参见《JavaScript：The Good Parts》，原作者Douglas Crockford［O'Reilly，2008］）

JavaScript全局作用域

在JavaScript中，如果你在变量定义中省略var关键字（例如，写成x = 1而非var x = 1），这个变量会放在全局作用域中，所有的JavaScript文件和<script>块都可以访问它。下面是一个例子：

```
<script>
    var f = function () {
        // DANGER: 'i' is not declared with 'var'!
        for (i = 0; i < 10; i += 1) ...
    };

    f();
</script>
```

这段代码不慎把i放在了全局作用域中，那么以后的代码块也能看到它：

```
<script>
    alert(i); // Alerts '10'. 'i' is a global variable!
</script>
```

很多程序员没有注意到这个作用域规则，这个令人吃惊的行为可以产生奇怪的bug。这种bug的一个共同形式是，当两个函数都创建了有相同名字的局布变量时，忘记了使用var。这些函数会在背地里“交谈”，然后可怜的程序员可能会认为他的计算机疯了或者RAM坏了。

对于JavaScript通用的“最佳实践”是“**总是用var关键字来定义变量**”（例如：var x = 1）。这个方法把变量的作用域约束在定义它的（最内层）函数之中。

在Python和JavaScript中没有嵌套的作用域

像C++和Java这样的语言有“语句块作用域”，定义在if、for、try或者类似结构中的变量被限制在这个语句块的嵌套作用域里。

```
if (...) {
    int x = 1;
}
x++; // Compile-error! 'x' is undefined.
```

但是在Python和JavaScript中，在语句块中定义的变量会“溢出”到整个函数。例如，请注意在下面这段完全正确的Python代码中对example_value的使用：

```
# No use of example_value up to this point.
if request:
    for value in request.values:
        if value > 0:
            example_value = value
            break

for logger in debug.loggers:
    logger.log("Example:", example_value)
```

这条作用域规则让很多程序员感到意外，并且写成这样的代码也很难读。在其他语言中，可能更容易找到example_value最初是在哪里定义的——你只要沿着你所在的函数“左手边”一路找下去就可以了。

前面的例子同时也有错误：如果在代码的第一部分中没有设置example_value，那么第二部分会产生异常："NameError: 'example_value' is not defined"。我们可以改正它并让代码更可读，把example_value的定义移到它与使用点的“最近共同前辈”（就嵌套而言）处就可以了：

```
example_value = None

if request:
    for value in request.values:
        if value > 0:
            example_value = value
            break
```

```
if example_value:
    for logger in debug.loggers:
        logger.log("Example:", example_value)
```

然而，在这个例子中其实example_value完全可以不要。example_value只保存一个中间结果，如第9章所述，这种变量可以通过“尽早完成任务”来消除。在这个例子中，这意味着在找到example_value时马上给它写日志。

下面是修改过的新代码：

```
def LogExample(value):
    for logger in debug.loggers:
        logger.log("Example:", value)

if request:
    for value in request.values:
        if value > 0:
            LogExample(value)  # deal with 'value' immediately
            break
```

把定义向下移

原来的C语言要求把所有的变量定义放在函数或语句块的顶端。这个要求很令人遗憾，因为对于有很多变量的函数，它强迫读者马上思考所有这些变量，即使是要到很久之后才会用到它们。（C99和C++去掉了这个要求。）

在下面的例子中，所有的变量都无辜地定义在函数的顶部：

```
def ViewFilteredReplies(original_id):
    filtered_replies = []
    root_message = Messages.objects.get(original_id)
    all_replies = Messages.objects.select(root_id=original_id)
        root_message.view_count += 1
        root_message.last_view_time = datetime.datetime.now()
        root_message.save()

        for reply in all_replies:
            if reply.spam_votes <= MAX_SPAM_VOTES:
                filtered_replies.append(reply)

        return filtered_replies
```

这段示例代码的问题是它强迫读者同时考虑3个变量，并且在它们间不断切换。

因为读者在读到后面之前不需要知道所有变量，所以可以简单地把每个定义移到对它的使用之前：

```
def ViewFilteredReplies(original_id):
    root_message = Messages.objects.get(original_id)
    root_message.view_count += 1
    root_message.last_view_time = datetime.datetime.now()
```

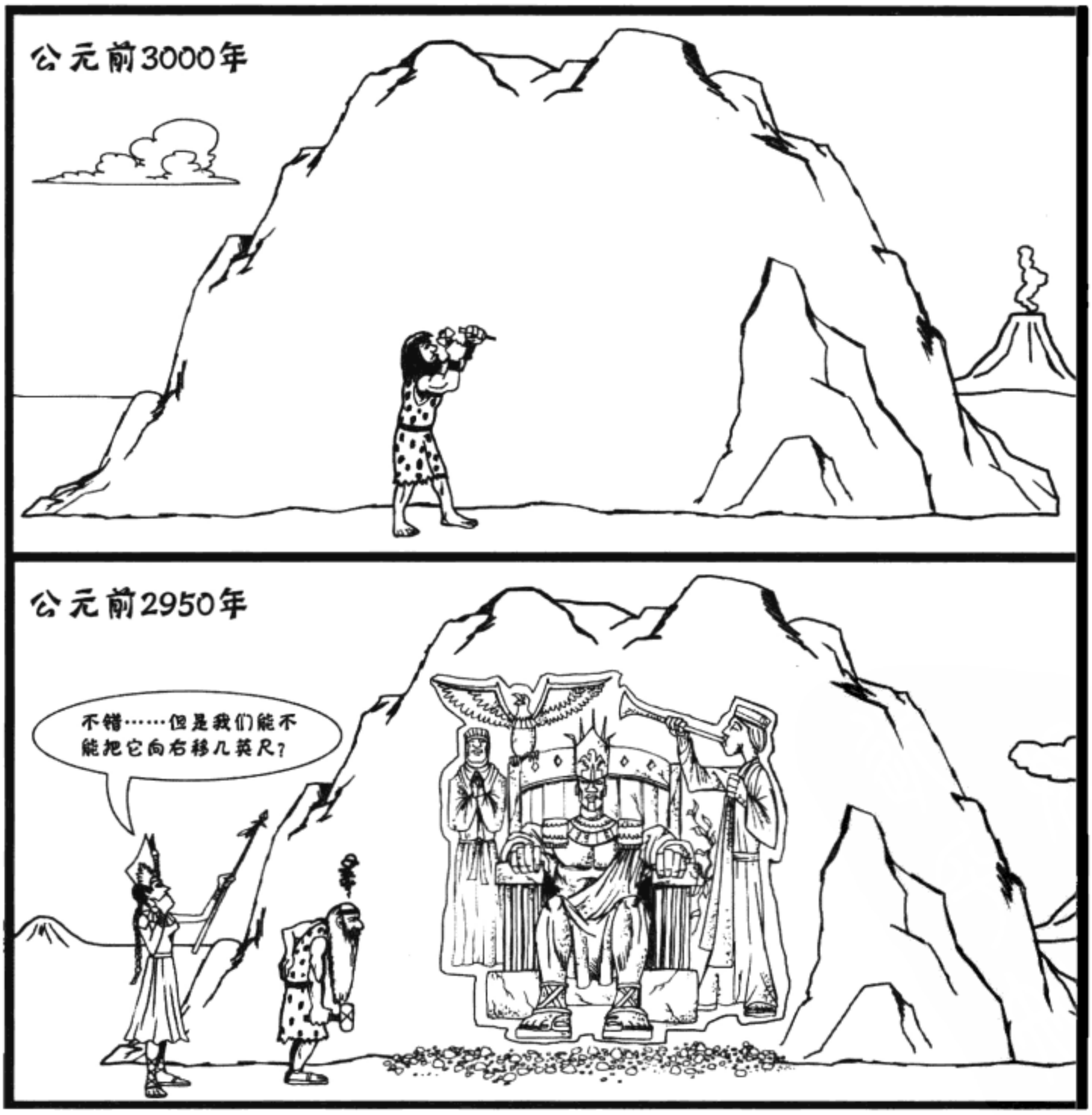
```
    root_message.save()

    all_replies = Messages.objects.select(root_id=original_id)
    filtered_replies = []
    for reply in all_replies:
        if reply.spam_votes <= MAX_SPAM_VOTES:
        filtered_replies.append(reply)

    return filtered_replies
```

你可能会想到底all_replies是不是个必要的变量，或者这么做是不是可以消除它：

```
for reply in Messages.objects.select(root_id=original_id):
    ...
```

在本例中，all_replies是一个相当好的解释，所以我们决定留下它。

只写一次的变量更好



到目前为止，本章讨论了很多变量参与“整个游戏”是怎样导致难以理解的程序的。不断变化的变量更难让人理解。跟踪这种变量的值更有难度。

要解决这种问题，我们有一个听起来怪怪的建议：**只写一次的变量更好。**

“永久固定”的变量更容易思考。当前，像这种常量：

```
static const int NUM_THREADS = 10;
```

不需要读者思考很多。基于同样的原因，鼓励在C++中使用`const`（在Java中使用`final`）。

实际上，在很多语言中（包括Python和Java），一些内置类型（如string）是不可变的。如James Gosling（Java的创造者）所说：“（常量）往往不会引来麻烦。”

但是就算你不能让变量只写一次，让变量在较少的地方改动仍有帮助。

关键思想

操作一个变量的地方越多，越难确定它的当前值。

那么怎么做呢？能把一个变量改成只写一次吗？很多时间这需要修改代码的结构，就像你将在下面的例子中所见到的那样。

最后的例子

作为本章最后一个例子，我们要给出一个能演示目前为止我们所讨论过的多条原则的例子。

假设你有一个网页，上面有几个文本输入字段，布置如下：

```
<input type="text" id="input1" value="Dustin">
<input type="text" id="input2" value="Trevor">
<input type="text" id="input3" value="">
<input type="text" id="input4" value="Melissa">
...
```

如你所见，`id`从`input1`开始增加。

你的工作是写一个叫`setFirstEmptyInput()`的函数，它接受一个字符串并把它放在页面上第一个空的`<input>`字段中（在给出的示例中是`"input3"`）。这个函数应当返回已更新的那个DOM元素（如果没有剩下任何空字段则返回`null`）。下面是完成这项工作的代码，它没有遵守本章中的原则：

```
var setFirstEmptyInput = function (new_value) {
```

```
    var found = false;
    var i = 1;
    var elem = document.getElementById('input' + i);
    while (elem !== null) {
        if (elem.value === '') {
            found = true;
            break;
        }
        i++;
        elem = document.getElementById('input' + i);
    }
    if (found) elem.value = new_value;
    return elem;
};
```

这段代码可以完成工作，但看上去并不完美。什么地方不对？如何改进？

有很多途径可以用来思考对这段代码的改进，但我们会从它所使用的变量这个角度开始考虑：

- var found
- var i
- var elem

所有三个变量都存在于整个函数中，并且写入了多次。让我们来尝试改进它们中的每一个。

如我们在本章前面讨论过的，中间变量（如found）通常可以通过提前返回来消除。下面是这一点改进：

```
var setFirstEmptyInput = function (new_value) {
    var i = 1;
    var elem = document.getElementById('input' + i);
    while (elem !== null) {
        if (elem.value === '') {
            elem.value = new_value;
            return elem;
        }
        i++;
        elem = document.getElementById('input' + i);
    }
    return null;
};
```

接下来，看一下elem。它在整个代码中以循环的方式多次用到，这让我们很难跟踪它的值。这段代码让我们觉得elem就是在迭代的值，实际上只是在累加i。所以把while循环重写成对i的for循环。

```
var setFirstEmptyInput = function (new_value) {
```

```
        for (var i = 1; true; i++) {
            var elem = document.getElementById('input' + i);
            if (elem === null)
                return null;  // Search Failed. No empty input found.

            if (elem.value === '') {
                elem.value = new_value;
                return elem;
            }
        }
    };
```

特别地，请注意elem是如何成为一个只写一次的变量的，它的生命周期只在循环内。用true来作为for循环的条件并不多见，但作为交换，我们可以在同一行里看到i的定义与修改。（传统的while(true)也是个合理的选择。）

总结

本章是关于程序中的变量是如何快速累积而变得难以跟踪的。你可以通过减少变量的数量和让它们尽量“轻量级”来让代码更有可读性。具体有：

- **减少变量**，即那些妨碍的变量。我们给出了几个例子来演示如何通过立刻处理结果来消除“中间结果”变量。
- **减小每个变量的作用域**，越小越好。把变量移到一个有最少代码可以看到它的地方。眼不见，心不烦。
- **只写一次的变量更好**。那些只设置一次值的变量（或者const、final、常量）使得代码更容易理解。

第三部分

重新组织代码

第二部分讨论了如何改变程序的“循环与逻辑”来让代码更有可读性。我们描述了几种技巧，这些技巧都需要对代码结构做出微小的改动。

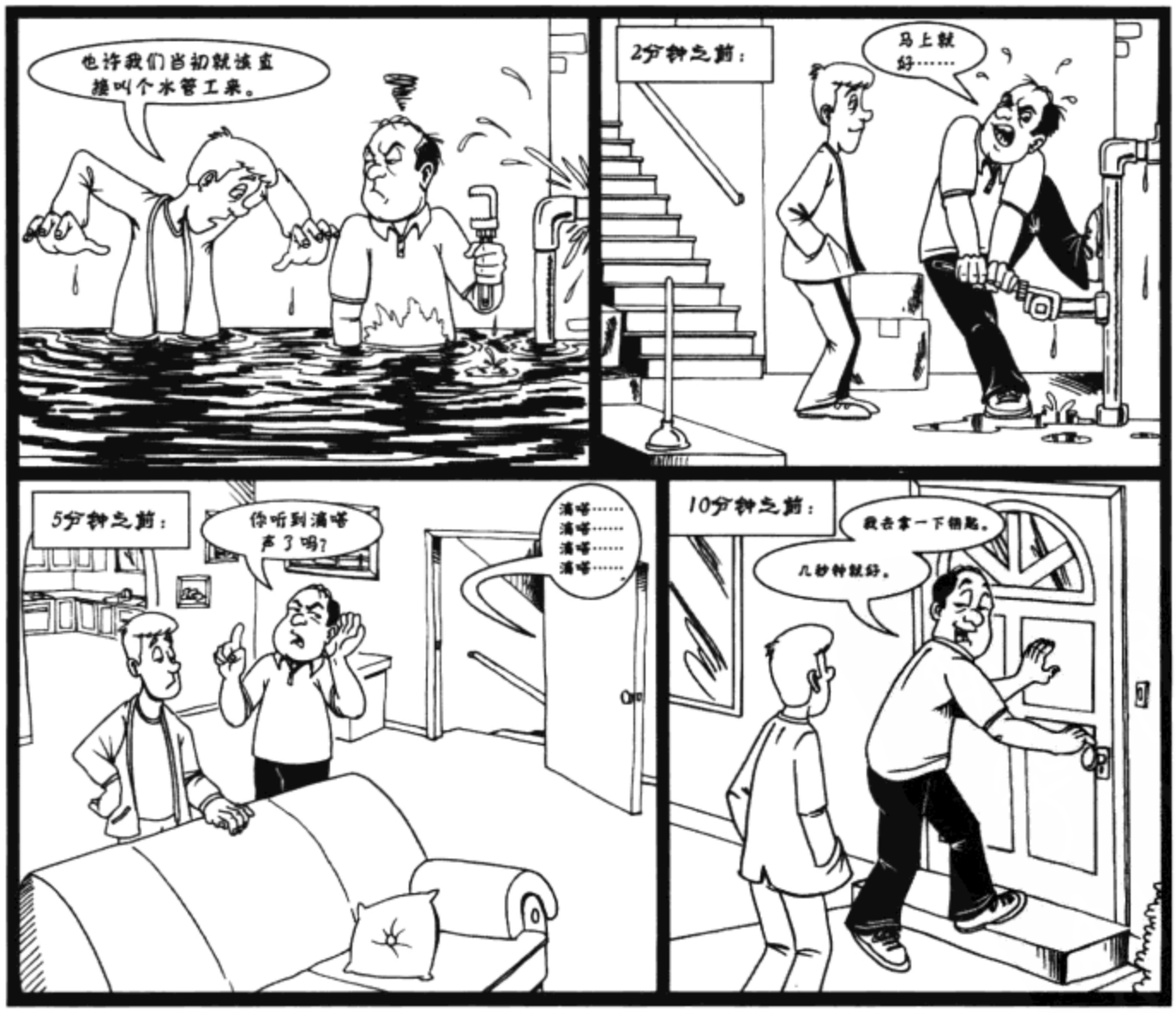
该部分会讨论可以在函数级别对代码做的更大的改动。具体来讲，我们会讲到三种组织代码的方法：

- 抽取出那些与程序主要目的“不相关的子问题”。
- 重新组织代码使它一次只做一件事情。
- 先用自然语言描述代码，然后用这个描述来帮助你找到更整洁的解决方案。

最后，我们会讨论你可以把代码完全移除或者一开始就避免写它的那些情况——唯一可称为改进代码可读性的最佳方法。

第10章

抽取不相关的子问题



所谓工程学就是关于把大问题拆分成小问题再把这些问题的解决方案放回一起。把这条原则应用于代码会使代码更健壮并且更容易读。

本章的建议是“**积极地发现并抽取出不相关的子逻辑**”。我们是指：

1. 看看某个函数或代码块，问问你自己：这段代码高层次的目标是什么？
2. 对于每一行代码，问一下：它是直接为了目标而工作吗？这段代码高层次的目标是什么呢？
3. 如果足够的行数在解决不相关的子问题，抽取代码到独立的函数中。

你每天可能都会把代码抽取到单独的函数中。但在本章中，我们决定关注抽取的一个特别情形：不相关的子问题，在这种情形下抽取出的函数无忧无虑，并不关心为什么会调用它。

你将会看到，这是个简单的技巧却可以从根本上改进你的代码。然而由于某些原因，很多程序员没有充分使用这一技巧。这里的诀窍就是主动地寻找那些不相关的子问题。

在本章中，我们会看到几个不同的例子，它们针对你将遇到的不同情形来讲明这些技巧。

介绍性的例子：findClosestLocation()

下面JavaScript代码的高层次目标是“找到距离给定点最近的位置”（请勿纠结于斜体部分所用到的高级几何知识）：

```
// Return which element of 'array' is closest to the given latitude/longitude.
// Models the Earth as a perfect sphere.
var findClosestLocation = function (lat, lng, array) {
    var closest;
    var closest_dist = Number.MAX_VALUE;
    for (var i = 0; i < array.length; i += 1) {
        // Convert both points to radians.
        var lat_rad = radians(lat);
        var lng_rad = radians(lng);
        var lat2_rad = radians(array[i].latitude);
        var lng2_rad = radians(array[i].longitude);

        // Use the "Spherical Law of Cosines" formula.
        var dist = Math.acos(Math.sin(lat_rad) * Math.sin(lat2_rad) +
                             Math.cos(lat_rad) * Math.cos(lat2_rad) *
                             Math.cos(lng2_rad - lng_rad));

        if (dist < closest_dist) {
            closest = array[i];
            closest_dist = dist;
        }
    }
```

```
        return closest;
    };
```

循环中的大部分代码都旨在解决一个不相关的子问题："计算两个经纬坐标点之间的球面距离"。因为这些代码太多了，把它们抽取到一个独立的spherical_distance()函数是合理的：

```
    var spherical_distance = function (lat1, lng1, lat2, lng2) {
        var lat1_rad = radians(lat1);
        var lng1_rad = radians(lng1);
        var lat2_rad = radians(lat2);
        var lng2_rad = radians(lng2);

        // Use the "Spherical Law of Cosines" formula.
        return Math.acos(Math.sin(lat1_rad) * Math.sin(lat2_rad) +
                         Math.cos(lat1_rad) * Math.cos(lat2_rad) *
                         Math.cos(lng2_rad - lng1_rad));
    };
```

现在，剩下的代码变成了：

```
    var findClosestLocation = function (lat, lng, array) {
        var closest;
        var closest_dist = Number.MAX_VALUE;
        for (var i = 0; i < array.length; i += 1) {
            var dist = spherical_distance(lat, lng, array[i].latitude, array[i].longitude);
            if (dist < closest_dist) {
                closest = array[i];
                closest_dist = dist;
            }
        }
        return closest;
    };
```

这段代码的可读性好得多，因为读者可以关注于高层次目标，而不必因为复杂的几何公式分心。

作为额外的奖励，spherical_distance()很容易单独做测试。并且spherical_distance()是那种在以后可以重用的函数。这就是为什么它是一个"不相关"的子问题——它完全是自包含的，并不知道其他程序是如何使用它的。

纯工具代码

有一组核心任务大多数程序都会做，例如操作字符串、使用哈希表以及读/写文件。

通常，这些"基本工具"是由编程语言中内置的库来实现的。例如，如果你想读取文件的整个内容，在PHP中你可以调用file_get_contents("filename")，或者在Python中你可以用open("filename").read()。

但有时你要自己来填充这中间的空白。例如，在C++中，没有简单的方法来读取整个文件。取而代之的是你不可避免地要写这样的代码：

```
ifstream file(file_name);

// Calculate the file's size, and allocate a buffer of that size.
file.seekg(0, ios::end);
const int file_size = file.tellg();
char* file_buf = new char [file_size];

// Read the entire file into the buffer.
file.seekg(0, ios::beg);
file.read(file_buf, file_size);
file.close();

...
```

这是一个不相关子问题的经典例子，应该把它抽取到一个新的函数中，比如ReadFileToString()。现在，你代码库的其他部分可以当做C++语言中确实有ReadFileToString()这个函数。

通常来讲，如果你在想：“我希望我们的库里有XYZ()函数”，那么就写一个！（如果它还不存在的话）经过一段时间，你会建立起一组不错的工具代码，后者可以应用于多个项目。

其他多用途代码

当调试JavaScript代码时，程序员经常使用alert()来弹出消息框，把一些信息显示给他们看，这是Web版本的“printf()调试”。例如，下面函数调用会用Ajax把数据提交给服务器，然后显示从服务器返回的字典。

```
ajax_post({
    url: 'http://example.com/submit',
    data: data,
    on_success: function (response_data) {
        var str = "{\n";
        for (var key in response_data) {
            str += "  " + key + " = " + response_data[key] + "\n";
        }
        alert(str + "}");

        // Continue handling 'response_data' ...
    }
});
```

这段代码的高层次目标是“对服务器做Ajax调用，然后处理响应结果”。但是有很多代码都在处理不相关的子问题，也就是美化字典的输出。把这段代码抽取到一个像format_pretty(obj)这样的函数中很简单：

```
var format_pretty = function (obj) {
    var str = "{\n";
    for (var key in obj) {
        str += "  " + key + " = " + obj[key] + "\n";
    }
    return str + "}";
};
```

意料之外的好处

出于很多理由，抽取出format_pretty()是个好主意。它使得用代码更简单，并且format_petty()是一个很方便的函数。

但是还有一个不那么明显的重要理由：**当format_pretty()中的代码自成一体后改进它变得更容易。**当你在使用一个独立的小函数时，感觉添加功能、改进可读性、处理边界情况等都更容易。

下面是format_pretty(obj)无法处理的一些情况。

- 它期望obj是一个对象。如果它是个普通字符串（或者undefined），那么当前的代码会抛出异常。
- 它期望obj的每个值都是简单类型。否则如果它包含嵌套对象的话，当前代码会把它们显示成[object Object]，这并不是很漂亮。

在我们把format_pretty()拆分成自己的函数之前，感觉要做这些改进可能会需要大量的工作。（实际上，迭代地输出嵌套对象在没有独立的函数时是很难的。）

但是现在增加这些功能就很简单了。改进后的代码如下：

```
var format_pretty = function (obj, indent) {
    // Handle null, undefined, strings, and non-objects.
    if (obj === null) return "null";
    if (obj === undefined) return "undefined";
    if (typeof obj === "string") return '"' + obj + '"';
    if (typeof obj !== "object") return String(obj);

    if (indent === undefined) indent = "";

    // Handle (non-null) objects.
    var str = "{\n";
    for (var key in obj) {
        str += indent + "  " + key + " = ";
        str += format_pretty(obj[key], indent + "  ") + "\n";
    }
    return str + indent + "}";
};
```

上面的代码把前面提到的不足都改正了，产生的输出如下：

```
{
    key1 = 1
    key2 = true
    key3 = undefined
    key4 = null
    key5 = {
        key5a = {
            key5a1 = "hello world"
        }
    }
}
```

创建大量通用代码

ReadFileToString()和format_pretty()这两个函数是不相关子问题的好例子。它们是如此基本而广泛适用，所以很可能会在多个项目中重用。代码库常常有个专门的目录来存放这种代码（例如util），这样它们就很方便重用。

通用代码很好，因为**“它完全地从项目的其他部分中解耦出来”**。像这样的代码容易开发，容易测试，并且容易理解。想象一下如果你所有的代码都如此会怎样！

想一想你使用的众多强大的库和系统，如SQL数据库、JavaScript库和HTML模板系统。你不用操心它们的内部——那些代码与你的项目完全分离。其结果是，你项目的代码库仍然较小。

从你的项目中拆分出越多的独立库越多越好，因为你代码的其他部分会更小而且更容易思考。

这是自顶向下或者自底向上式编程吗？

自顶向下编程是一种风格，先设计高层次模块和函数，然后根据支持它们的需要来实现低层次函数。

自底向上编程尝试首先预料和解决所有的子问题，然后用这些代码段来建立更高层次的组件。

本章并不鼓吹一种方法比另一种好。大多数编程都包括了两者的组合。重要的是最终的结果：移除并单独解决子问题。

项目专有的功能

在理想情况下，你所抽取出的子问题对项目一无所知。但是就算它们不是这样，也没有问题。分离子问题仍然可能创造奇迹。

下面是一个商业评论网站的例子。这段Python代码创建一个新的Business对象并设置它的name、url和date_created：

```
business = Business()
business.name = request.POST["name"]

url_path_name = business.name.lower()
url_path_name = re.sub(r"['\.]", "", url_path_name)
url_path_name = re.sub(r"[^a-z0-9]+", "-", url_path_name)
url_path_name = url_path_name.strip("-")
business.url = "/biz/" + url_path_name

business.date_created = datetime.datetime.utcnow()
business.save_to_database()
```

url应该是一个“干净”版本的name。例如，如果name是“A.C. Joe's Tire & Smog, Inc.”，url就是 "/biz/ac-joes-tire-smog-inc"。

这段代码中的不相关子问题是“把名字转换成一个有效的URL”。这段代码很容易抽取。同时，我们还可以预先编译正则表达式（并给它们以可读的名字）：

```
CHARS_TO_REMOVE = re.compile(r"['\.]+")
CHARS_TO_DASH = re.compile(r"[^a-z0-9]+")

def make_url_friendly(text):
    text = text.lower()
    text = CHARS_TO_REMOVE.sub('', text)
    text = CHARS_TO_DASH.sub('-', text)
    return text.strip("-")
```

现在，原来的代码可以有更“常规”的外观了：

```
business = Business()
business.name = request.POST["name"]
business.url = "/biz/" + make_url_friendly(business.name)
business.date_created = datetime.datetime.utcnow()
business.save_to_database()
```

读这段代码所花的工夫要小得多，因为你不会被正则表达式和深层的字符串操作分散精力。

你应该把make_rul_friendly()的代码放在哪里呢？那看上去是相当通用的一个函数，因此把它放到一个单独的util/目录中是合理的。另一方面，这些正则表达式是按照美国商业名称的思路设计的，所以可能这段代码应该留在原来文件中。这实际上并不重要，以后你可以很容易地把这个定义移到另一个地方。更重要的是make_url_friendly()完全被抽取出来。

简化已有接口

人人都爱提供整洁接口的库——那种参数少，不需要很多设置并且通常只需要花一点工夫就可以使用的库。它让你的代码看起来优雅：简单而又强大。

但如果你所用的接口并不整洁，你还是可以创建自己整洁的“包装”函数。

例如，处理JavaScript浏览器中的cookie比理想情况糟糕很多。从概念上讲，cookie是一组名/值对。但是浏览器提供的接口只提供了一个document.cookie字符串，语法如下：

```
name1=value1; name2=value2; ...
```

要找到你想要的cookie，你不得不自己解析这个巨大的字符串。下面的例子代码用来读取名为"max_results"的cookie的值。

```
var max_results;
var cookies = document.cookie.split(';');
for (var i = 0; i < cookies.length; i++) {
    var c = cookies[i];
    c = c.replace(/^[ ]+/, '');  // remove leading spaces
    if (c.indexOf("max_results=") === 0)
        max_results = Number(c.substring(12, c.length));
}
```

这段代码可真难看。很明显，它等着我们创建一个get_cookie()函数，这样我们就只需要写：

```
var max_results = Number(get_cookie("max_results"));
```

创建或者改变一个cookie的值更奇怪。你得把document.cookie设置为一个必需严格满足下面语法的值：

```
document.cookie = "max_results=50; expires=Wed, 1 Jan 2020 20:53:47 UTC; path=/";
```

这条语句看上去像是它会重写所有其他的已有cookie，但是（魔术般地）它没有！

设置cookie更理想的接口应该像这样：

```
set_cookie(name, value, days_to_expire);
```

擦除cookie也不符合直觉：你得把cookie设置成在过去的时间过期才行。更理想的接口应该是很简单的：

```
delete_cookie(name);
```

这里我们学到的是“**你永远都不要安于使用不理想的接口**”。你总是可以创建你自己的包装函数来隐藏接口的粗陋细节，让它不再成为你的阻碍。

按需重塑接口

程序中很多代码在那里只是为了支持其他代码——例如，为函数设置输入或者对输出做后期处理。这些“粘附”代码常常和程序的实际逻辑没有任何关系。这种传统的代码是抽取到独立函数的最好机会。

例如，假设你有一个Python字典，包含敏感的用户信息，如{"username": "...", "password": "..."}，你需要把这些信息放入一个URL中。因为它很敏感，所以你决定要对字典先加密，这会用到一个Cipher类。

但是Cipher期望的输入是字节串，不是字典。而且Cipher返回一个字节串，但我们需要的是对于URL安全的东西。Cipher还需要几个额外的参数，使用起来还有些麻烦。

开始以为很简单的任务变成了一大堆粘附代码：

```
user_info = { "username": "...", "password": "..." }
user_str = json.dumps(user_info)
cipher = Cipher("aes_128_cbc", key=PRIVATE_KEY, init_vector=INIT_VECTOR, op=ENCODE)
encrypted_bytes = cipher.update(user_str)
encrypted_bytes += cipher.final() # flush out the current 128 bit block
url = "http://example.com/?user_info=" + base64.urlsafe_b64encode(encrypted_bytes)
...
```

尽管我们要解决的问题是把用户的信息编码成URL，这段代码的主体只是在“把Python对象编码成对URL友好的字符串”。把子问题抽取出来并不难：

```
def url_safe_encrypt(obj):
    obj_str = json.dumps(obj)
    cipher = Cipher("aes_128_cbc", key=PRIVATE_KEY, init_vector=INIT_VECTOR, op=ENCODE)
    encrypted_bytes = cipher.update(obj_str)
    encrypted_bytes += cipher.final()  # flush out the current 128 bit block
    return base64.urlsafe_b64encode(encrypted_bytes)
```

然后得到的程序中执行“真正”逻辑的代码很简单：

```
user_info = { "username": "...", "password": "..." }
url = "http://example.com/?user_info=" + url_safe_encrypt(user_info)
```

过犹不及

像我们在本章的开头所说的那样，我们的目标是“积极地发现和抽取不相关的子问题”。我们说“积极地”是因为大多数程序员不够积极。但也可能会过于积极，导致过犹不及。

例如，前一节中的代码可能会进一步拆分，如下：

```
user_info = { "username": "...", "password": "..." }
url = "http://example.com/?user_info=" + url_safe_encrypt_obj(user_info)

def url_safe_encrypt_obj(obj):
    obj_str = json.dumps(obj)
    return url_safe_encrypt_str(obj_str)

def url_safe_encrypt_str(data):
    encrypted_bytes = encrypt(data)
```

```
    return base64.urlsafe_b64encode(encrypted_bytes)

def encrypt(data):
    cipher = make_cipher()
    encrypted_bytes = cipher.update(data)
    encrypted_bytes += cipher.final() # flush out any remaining bytes
    return encrypted_bytes

def make_cipher():
    return Cipher("aes_128_cbc", key=PRIVATE_KEY, init_vector=INIT_VECTOR, op=ENCODE)
```

引入这么多小函数实际上对可读性是不利的，因为读者要关注更多东西，并且按照执行的路径需要跳来跳去。

为代码增加一个函数存在一个小的（却有形的）可读性代价。在前面的情况里，付出这种代价却什么也没有得到。如果你项目的其他部分也需要这些小函数，那么增加它们是有道理的。但是目前为止，还没有这个需要。

总结

对本章一个简单的总结就是“**把一般代码和项目专有的代码分开**”。其结果是，大部分代码都是一般代码。通过建立一大组库和辅助函数来解决一般问题，剩下的只是让你的程序与众不同的核心部分。

这个技巧有帮助的原因是它使程序员关注小而定义良好的问题，这些问题已经同项目的其他部分脱离。其结果是，对于这些子问题的解决方案倾向于更加完整和正确。你也可以在以后重用它们。

阅读参考

Martin Fowler，《Refactoring: Improving the Design of Existing code》（Fowler etal., Addison-Wesley Professional,1999）描述了重构的“抽取方法”，而且列举了很多其他重构代码的方法。

Kent Beck，《 Smalltalk Best Practice Patterns》（Prentice Hall,1996）描述了“组合方法模式”，其中列出了几条把代码拆分成得多小函数的原则。尤其是其中的一条原则“把一个方法中的所有操作保持在一个抽象层次上”。

这些思想和我们的建议“抽取不相关的子问题”相似。本章所讨论的是抽取方法中的一个简单而又特定的情况。

第11章

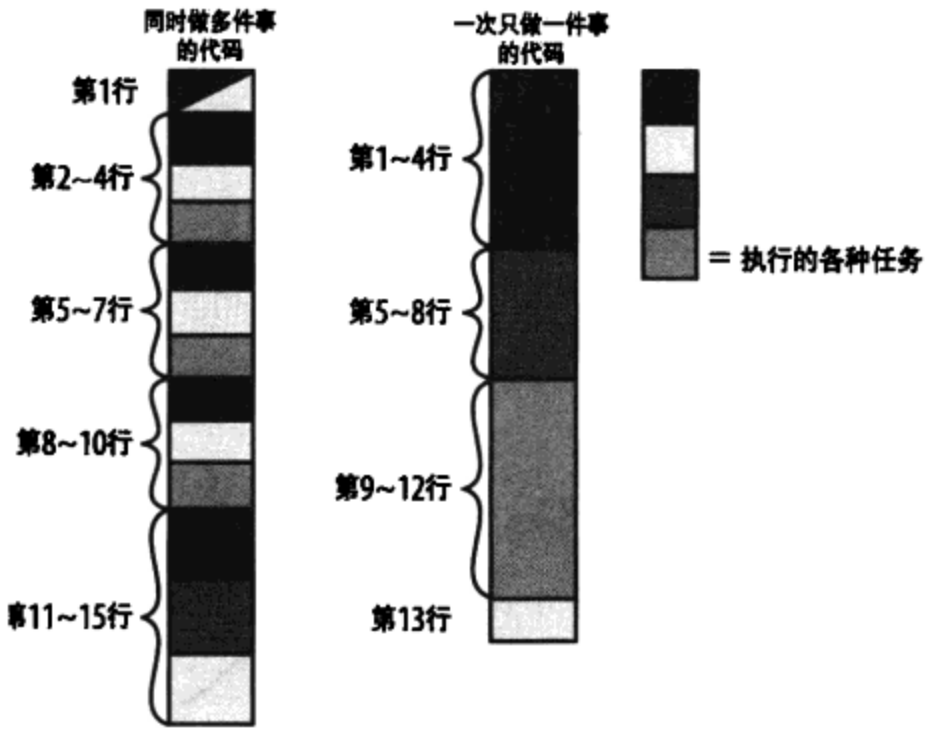
一次只做一件事

同时在做几件事的代码很难理解。一个代码块可能初始化对象，清除数据，解析输入，然后应用业务逻辑，所有这些都同时进行。如果所有这些代码都纠缠在一起，对于每个“任务”都很难靠其自身来帮你理解它从哪里开始，到哪里结束。

关键思想

应该把代码组织得一次只做一件事情。

换个说法，本章是关于如果给代码“整理碎片”的。下图演示了这个过程：左边所示为一段代码所做的各种任务，右边所示是同一段代码在组织成一次只做一件事情后的样子。



你也许听说过这个建议：“一个函数只应当做一件事”。我们的建议和这差不多，但不是关于函数边界的。当然，把一个大函数拆分成多个小一些的函数是好的。但是就算你不这样做，你仍然可以在函数内部组织代码，使得它感觉像是有分开的逻辑段。

下面是用于使代码“一次只做一件事”所用到的流程：

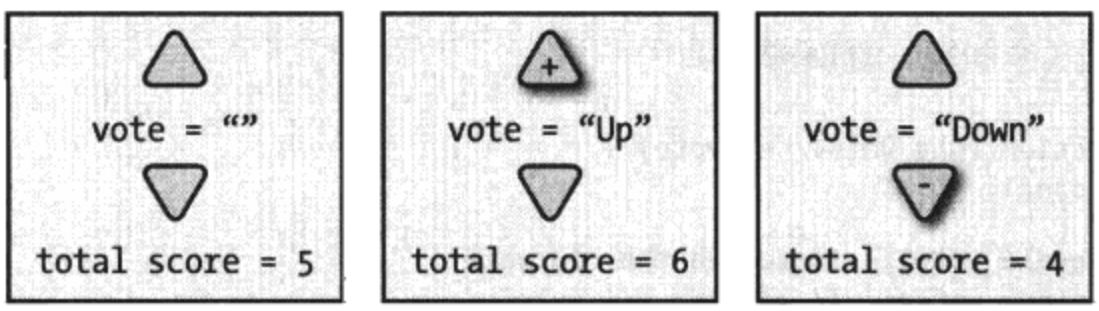
1. 列出代码所做的所有“任务”。这里的“任务”没有很严格的定义——它可以小得如“确保这个对象有效”，或者含糊得如“遍历树中所有结点”。
2. 尽量把这件任务拆分到不同的函数中，或者至少是代码中不同的段落中。

在本章中，我们会给出几个例子说明如何来做。

任务可以很小

假设有一个博客上的投票插件，用户可以给一条评论投“上”或“下”票。每条评论的总分为所有投票的和：“上”票对应分数为+1，“下”票–1。

下面是用户投票可能的三种状态，以及它如何影响总分：



当用户按了一个按钮（创建或改变她的投票），会调用以下JavaScript代码：

```
vote_changed(old_vote, new_vote); // each vote is "Up", "Down", or ""
```

下面这个函数计算总分，并且对old_vote和new_vote的各种组合都有效：

```
var vote_changed = function (old_vote, new_vote) {
    var score = get_score();

    if (new_vote !== old_vote) {
        if (new_vote === 'Up') {
            score += (old_vote === 'Down' ? 2 : 1);
        } else if (new_vote === 'Down') {
            score -= (old_vote === 'Up' ? 2 : 1);
        } else if (new_vote === '') {
            score += (old_vote === 'Up' ? -1 : 1);
        }
    }

    set_score(score);
};
```

尽管这段代码很短，但它做了很多事情。其中有很多错综复杂的细节，很难看一眼就知道是否里面有“偏一位”错误、录入错误或者其他bug。

这段代码好像是只做了一件事情（更新分数），但实际上是同时做了两件事：

1. 把old_vote和new_vote解析成数字值。
2. 更新分数。

我们可以分开解决每个任务来使代码变简单。下面的代码解决第一个任务，把投票解析成数字值：

```
var vote_value = function (vote) {
```

```
        if (vote === 'Up') {

            return +1;
        }
        if (vote === 'Down') {
            return -1;
        }
        return 0;
    };
```

现在其余的代码可以解决第二个问题，更新分数：

```
    var vote_changed = function (old_vote, new_vote) {
        var score = get_score();

        score -= vote_value(old_vote);  // remove the old vote
        score += vote_value(new_vote);  // add the new vote

        set_score(score);
    };
```

如你所见，要让自己确信代码可以工作，这个版本需要花费的心力小得多。“容易理解”在很大程度上就是这个意思。

从对象中抽取值

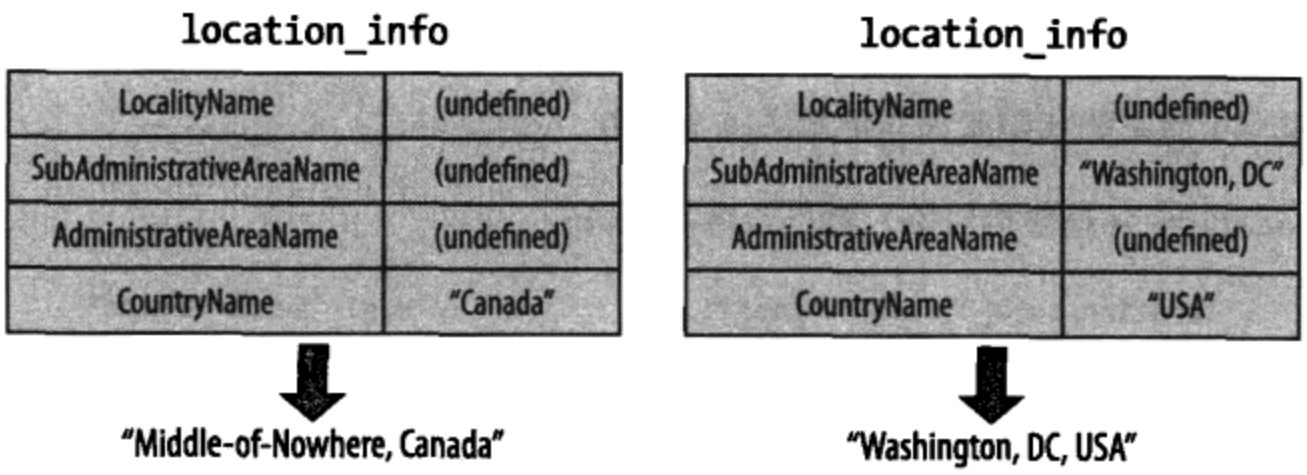
我们曾有过一段JavaScript代码，用来把用户的位置格式化成“城市，国家”这样友好的字符串，比如Santa Monica，USA（圣摩尼卡，美国）或者Pairs，France（巴黎，法国）。我们收到的是一个location_info字典，其中有很多结构化的信息。我们所要做的就是从所有的字段中找到“City”和“Country”然后把它们接在一起。

下图给出了输入/输入的示例：

到目前为止这看上去很简单，但是微妙之处在于“4个值中的每个或所有都可能缺失”。下面是解决方案：

- 当选择“City”时，“LocalityName”（城市/乡镇），如果有的话。然后是“SubAdministrativeAreaName”（大城市/国家），然后是“AdministrativeAreaName”（州/地区）。
- 如果三个都没有的话，那么赋予“City”一个默认值“Middle-of-Nowhere”。
- 如果“CountryName”不存在，就会用“Planet Earth”这个默认值。

下图给出两个处理缺失值的例子。



我们写了下面的代码来实现这个任务：

```
var place = location_info["LocalityName"];  // e.g. "Santa Monica"
if (!place) {
    place = location_info["SubAdministrativeAreaName"];  // e.g. "Los Angeles"
}
if (!place) {
    place = location_info["AdministrativeAreaName"];  // e.g. "California"
}
if (!place) {
    place = "Middle-of-Nowhere";
}
if (location_info["CountryName"]) {
    place += ", " + location_info["CountryName"];  // e.g. "USA"
} else {
    place += ", Planet Earth";
}

return place;
```

当然，这个有点乱，但是它能完成工作。

但是几天之后，我们需要改进功能：对于美国之内的位置，我们想要显示州名而不是国家名（如果可能的话）。所以不再是“Santa Monica, USA”，而是变成了“Santa Monica, California”。

把这个功能添加进前面的代码中会让它变得更难看。

应用“一次只做一件事情”原则

与其强行让这段代码满足我们的需要，不如我们停了下来并且意识到它现在已经同时在完成多个任务了：

1. 从字典location_info中提取值。
2. 按喜好顺序找到“City”，如果找不到就给默认值“Middle-of-Nowhere”。
3. 找到“Country”，如果找不到的话就用“Planet Earth”。
4. 更新place。

所以我们反而重写了原来的代码来独立地解决每个任务。

一个任务（从location_info中提取值）自己很容易解决：

```
var town    = location_info["LocalityName"];                // e.g. "Santa Monica"
var city    = location_info["SubAdministrativeAreaName"]; // e.g. "Los Angeles"
var state   = location_info["AdministrativeAreaName"];    // e.g. "CA"
var country = location_info["CountryName"];                 // e.g. "USA"
```

做到这儿，我们已经用完了location_info，不用再记得那些又长又违反直觉的键值了。反而我们得到了4个简单的变量。

下一步，我们要找出返回值中的“第二部分”是什么：

```
// Start with the default, and keep overwriting with the most specific value.
var second_half = "Planet Earth";
if (country) {
    second_half = country;
}
if (state && country === "USA") {
    second_half = state;
}
```

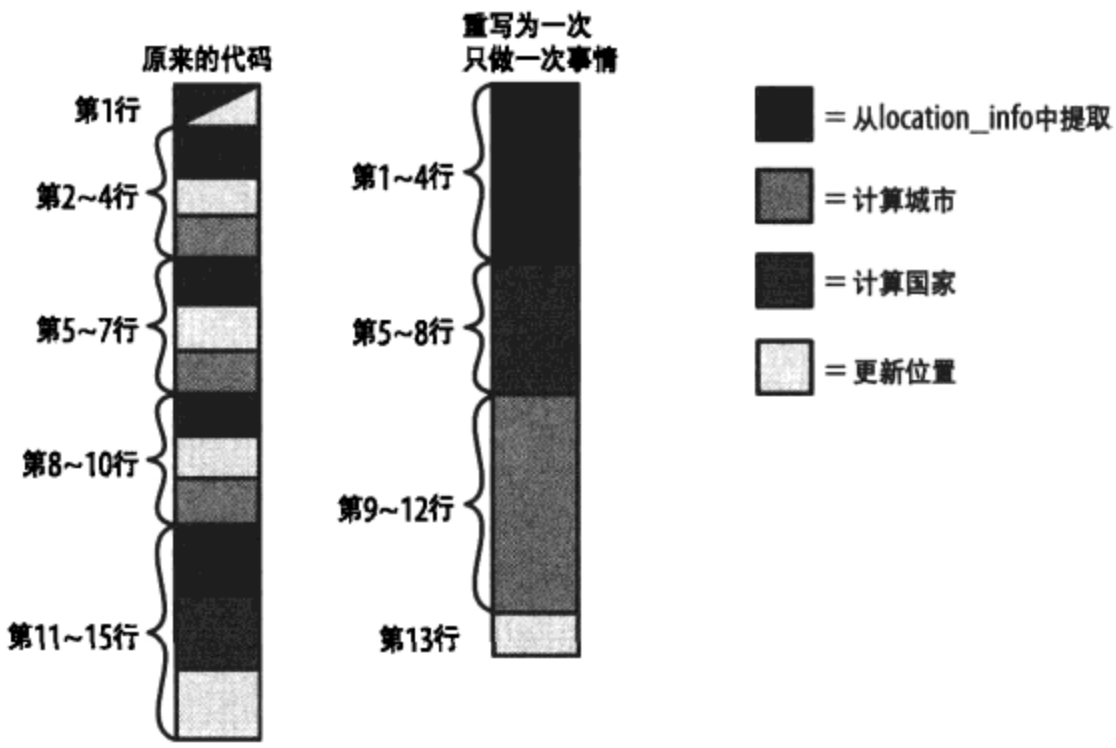
类似地，我们可以找出“第一部分”：

```
var first_half = "Middle-of-Nowhere";
if (state && country !== "USA") {
    first_half = state;
}
if (city) {
    first_half = city;
}
if (town) {
    first_half = town;
}
```

最后，我们把信息结合在一起：

```
return first_half + ", " + second_half;
```

本章开头展示的"碎片整理"实际上体现了原来的方案和这个新版本。下面是同一幅图，添加了更多细节：



如你所见，在第二个方案中把4个任务整理到独立的区域中了。

另一种做法

在重构代码时，经常有很多种做法，这个例子也不例外。一旦你把一些任务分离开，代码变得更容易让人思考，你可能会想到重构代码的更好方法。

例如，早先的一连串if语句需要小心地去读才能知道每种情况是否都对。在那段代码中其实有两个子任务同时在进行：

1. 遍历一系列变量，找出可用变量中最满意的那一个。
2. 依据国家是否为“USA”而采用不同的列表。

回顾从前的代码，你可以看到“if USA”的逻辑交织在其他的逻辑中。我们可以分别处理USA和非USA的情况：

```
var first_half, second_half;

if (country === "USA") {
    first_half = town || city || "Middle-of-Nowhere";
    second_half = state || "USA";
} else {
    first_half = town || city || state || "Middle-of-Nowhere";
    second_half = country || "Planet Earth";
}

return first_half + ", " + second_half;
```

如果你不了解JavaScript的话，a || b || c的写法会逐个计算直到找到第一个“真”值（在本例中，是指一个定义的非空字符串）。这段代码的好处是观察喜好列表很容易，也容易更新。大多数if语句都被扫地出门，业务逻辑所占的代码更少了。

更大型的例子

我们做过一个网页爬虫系统，会在下载每个网页后调用一个叫UpdateCounts()的函数来增加不同的统计数据：

```
void UpdateCounts(HttpDownload hd) {
    counts["Exit State" ][hd.exit_state()]++;          // e.g. "SUCCESS" or "FAILURE"
    counts["Http Response"][hd.http_response()]++;     // e.g. "404 NOT FOUND"
    counts["Content-Type" ][hd.content_type()]++;      // e.g. "text/html"
```

哦，那是我们希望代码成为的样子！

实际上，HttpDownload对象没有上面所示的任何方法。相反，HttpDownload是一个非常大并且非常复杂的类，有很多嵌套类，并且我们得自己把它们挖出来。更糟糕的是，有时有些值谁不知道是什么，这种情况下我们只能用“unknown”作为默认值。

由于这些原因，实际的代码非常乱：

```
// WARNING: DO NOT STARE DIRECTLY AT THIS CODE FOR EXTENDED PERIODS OF TIME.
void UpdateCounts(HttpDownload hd) {
// Figure out the Exit State, if available.
if (!hd.has_event_log() || !hd.event_log().has_exit_state()) {
    counts["Exit State"]["unknown"]++;
} else {
    string state_str = ExitStateTypeName(hd.event_log().exit_state());
    counts["Exit State"][state_str]++;
}
// If there are no HTTP headers at all, use "unknown" for the remaining elements.
if (!hd.has_http_headers()) {
    counts["Http Response"]["unknown"]++;
    counts["Content-Type"]["unknown"]++;
    return;
}

HttpHeaders headers = hd.http_headers();

// Log the HTTP response, if known, otherwise log "unknown"
if (!headers.has_response_code()) {
    counts["Http Response"]["unknown"]++;
} else {
    string code = StringPrintf("%d", headers.response_code());
    counts["Http Response"][code]++;
}

// Log the Content-Type if known, otherwise log "unknown"
if (!headers.has_content_type()) {
```

```
            counts["Content-Type"]["unknown"]++;
        } else {
            string content_type = ContentTypeMime(headers.content_type());
            counts["Content-Type"][content_type]++;
        }
    }
```

如你所见，代码很多，逻辑也很多，甚至还有几行重复的代码。读这种代码一点也不有趣。

特别是，这段代码在不同的任务间来回切换。下面是代码里通篇交织着的几个任务：

1. 使用"unknown"作为每个键的默认值。
2. 检测HttpDownload的成员是否缺失。
3. 抽取出值并将其转换成字符串。
4. 更新counts[]。

我们可以通过把其中一些任务分割到代码中单独的区域来改进这段代码：

```
    void UpdateCounts(HttpDownload hd) {
        // Task: define default values for each of the values we want to extract
        string exit_state = "unknown";
        string http_response = "unknown";
        string content_type = "unknown";

        // Task: try to extract each value from HttpDownload, one by one
        if (hd.has_event_log() && hd.event_log().has_exit_state()) {
            exit_state = ExitStateTypeName(hd.event_log().exit_state());
        }
        if (hd.has_http_headers() && hd.http_headers().has_response_code()) {
            http_response = StringPrintf("%d", hd.http_headers().response_code());
        }
        if (hd.has_http_headers() && hd.http_headers().has_content_type()) {
            content_type = ContentTypeMime(hd.http_headers().content_type());
        }

        // Task: update counts[]
        counts["Exit State"][exit_state]++;
        counts["Http Response"][http_response]++;
        counts["Content-Type"][content_type]++;
    }
```

如你所见，这段代码有三个分开的区域，各自目标如下：

1. 为我们感兴趣的三个键定义默认值。
2. 对于每个键，如果有的话就抽取出值，然后把它们转换成字符串。
3. 对于每个键/值更新counts[]。

这些区域好的地方是它们互相之前是独立的——当你在读一个区域时，你不必去想其他的区域。

请注意尽管我们列出4个任务，但我们只能拆分出3个。这完全没问题：你一开始列出的任务只是个开端。即使只拆分出它们中的一些就能对可读性有很大帮助，就像这个例子中一样。

进一步的改进

对于当初的大段代码来讲这个新版本算是有了改进。请注意我们甚至不用创建新函数来完成这个清理工作。像前面提到的，“一次只做一件事情”这个想法有助于不考虑函数的边界。

然而，也可以用另一种方法改进这段代码，通过引入3个辅助函数：

```
void UpdateCounts(HttpDownload hd) {
    counts["Exit State"][ExitState(hd)]++;
    counts["Http Response"][HttpResponse(hd)]++;
    counts["Content-Type"][ContentType(hd)]++;
}
```

这些函数会抽取出对应的值，或者返回"unknown"。例如：

```
string ExitState(HttpDownload hd) {
    if (hd.has_event_log() && hd.event_log().has_exit_state()) {
        return ExitStateTypeName(hd.event_log().exit_state());
    } else {
        return "unknown";
    }
}
```

请注意在这个做法中甚至没有定义任何变量！像第9章所提到的那样，保存中间结果的变量往往可以完全移除。

在这种方法里，我们简单地把问题从不同的角度“切开”。两种方法都很有可读性，因为它们让读者一次只需要思考一件事情。

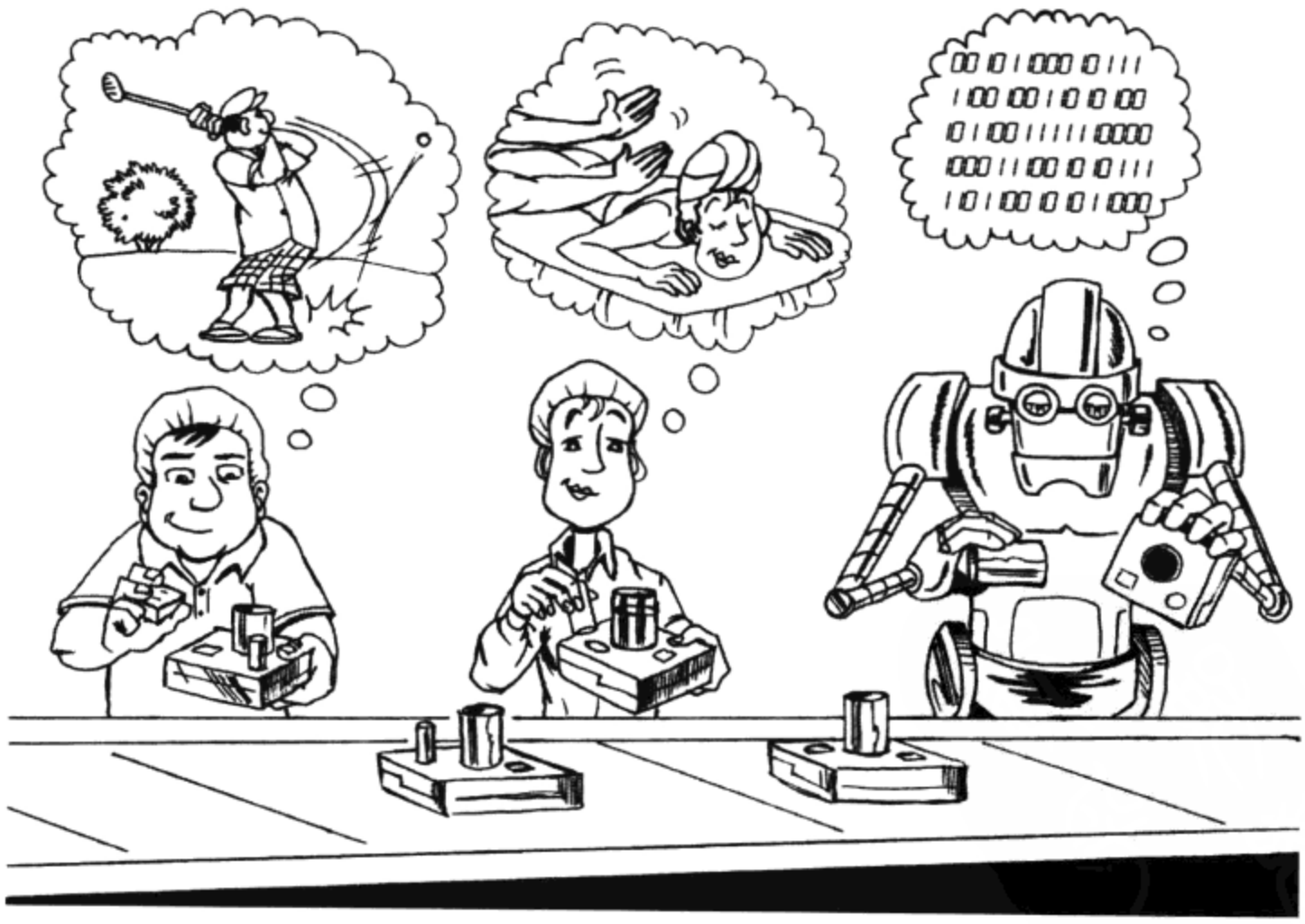
总结

本章给出了一个组织代码的简单技巧：**一次只做一件事情**。

如果你有很难读的代码，尝试把它所做的所有任务列出来。其中一些任务可以很容易地变成单独的函数（或类）。其他的可以简单地成为一个函数中的逻辑“段落”。具体如何拆分这些任务没有它们已经分开这个事实那样重要。难的是要准确地描述你的程序所做的所有这些小事情。

第12章

把想法变成代码



如果你不能把一件事解释给你祖母听的话说明你还没有真正理解它。

阿尔伯特·爱因斯坦

当把一件复杂的事向别人解释时，那些小细节很容易就会让他们迷惑。把一个想法用“自然语言”解释是个很有价值的能力，因为这样其他知识没有你这么渊博的人才可以理解它。这需要把一个想法精炼成最重要的概念。这样做不仅帮助他人理解，而且也帮助你自己把这个想法想得更清晰。

在你把代码“展示”给读者时也应使用同样的技巧。我们接受代码是你解释程序所做事情的主要手段这一关点。所以代码应当用“自然语言”编写。

在本章中，我们会用一个简单的过程来使你编写更清晰的代码：

1. 像对着一个同事一样用自然语言描述代码要做什么。
2. 注意描述中所用的关键词和短语。
3. 写出与描述所匹配的代码。

清楚地描述逻辑

下面是来自一个网页的一段PHP代码。这段代码在一段安全代码的顶部。它检查是否授权用户看到这个页面，如果没有，马上返回一个页面来告诉用户他没有授权：

```
$is_admin = is_admin_request();
if ($document) {
    if (!$is_admin && ($document['username'] != $_SESSION['username'])) {
        return not_authorized();
    }
} else {
    if (!$is_admin) {
        return not_authorized();
    }
}

// continue rendering the page ...
```

这段代码中有相当多的逻辑。像你在本书第二部分所读到的，这种大的逻辑树不容易理解。这些代码中的逻辑可以简化，但是怎么做呢？让我们从用自然语言描述这个逻辑开始：

```
授权你有两种方式：
1. 你是管理员
2. 你拥有当前文档（如果有当前文档的话）
否则，无法授权你。
```

下面是受这段描述启发写出的不同方案：

```
if (is_admin_request()) {
    // authorized
} elseif ($document && ($document['username'] == $_SESSION['username'])) {
    // authorized
} else {
    return not_authorized();
}

// continue rendering the page ...
```

这个版本有点不寻常，因为它有两个空语句体。但是代码要少一些，并且逻辑也简单，因为没有反义（前一个方案中有三个“not”）。起码它更容易理解。

了解函数库是有帮助的

我们曾有一个网站，其中有一个“提示框”，用来显示对用户有帮助的建议，比如：

> 提示：登录后可以看到过去做过的查找。[显示另一条提示！]

这种提示有几十条，全都藏在HTML中：

```
<div id="tip-1" class="tip">Tip: Log in to see your past queries.</div>
<div id="tip-2" class="tip">Tip: Click on a picture to see it close up.</div>
...
```

当读者访问这个页面时，会随机地让其中的一个div块变得可见，其他的还保持隐藏状态。

如果单击“Show me another tip！”链接，它会循环到下一个提示。下面是实现这一功能的一些代码，使用了JavaScript库jQuery：

```
var show_next_tip = function () {
    var num_tips = $('.tip').size();
    var shown_tip = $('.tip:visible');

    var shown_tip_num = Number(shown_tip.attr('id').slice(4));
    if (shown_tip_num === num_tips) {
        $('#tip-1').show();
    } else {
        $('#tip-' + (shown_tip_num + 1)).show();
    }
    shown_tip.hide();
};
```

这段代码还可以。但可以把它做得更好。让我们从描述开始，用自然语言来说这段代码要做的事情是：

> 找到当前可见的提示并隐藏它。
> 然后找到它的下一个提示并显示。
> 如果没有更多提示，循环回第一个提示。

根据这个描述，下面是另一个方案：

```
var show_next_tip = function () {
    var cur_tip = $('.tip:visible').hide(); // find the currently visible tip and hide it
    var next_tip = cur_tip.next('.tip');    // find the next tip after it
    if (next_tip.size() === 0) {            // if we're run out of tips,
        next_tip = $('.tip:first');         // cycle back to the first tip
    }
    next_tip.show();                        // show the new tip
}
```

这个方案的代码行数更少，并且不用直接操作整型。它与人们对此代码的理解一致。

在这个例子中，jQuery有一个.next()给我们用，这一点很有帮助。编写精练代码的一部分工作是了解你的库提供了什么。

把这个方法应用于更大的问题

前一个例子把过程应用于小块代码。在下一个例子中，我们会把它应用于更大的函数。你会看到，这个方法可以帮助你识别哪个片段可以分离，从而让你可以拆分代码。

假设我们有一个记录股票采购的系统。每笔交易都有4块数据：

- time（一个精确的购买日期和时间）
- ticker_symbol（公司简称，如：GOOG）
- price（价格，如：$600）
- number_of_shares（股票数量，如：100）

由于一些奇怪的原因，这些数据分布在三个分开的数据库表中，如下图所示。在每个数据库中，time是唯一的主键。

time	ticker_symbol
3:45	IBM
3:59	IBM
4:30	GOOG
5:20	AAPL
6:00	MSFT

time	price
3:45	$120
4:30	$600
5:00	$25
5:20	$200
6:00	$25

time	number_of_shares
3:45	50
3:59	200
4:10	75
4:30	100
5:20	80

现在，我们要写一个程序来把三个表联合在一起（就像在SQL中的JOIN操作所做的那样）。这个步骤应该是简单的，因为这些行都是按time来排序的，但是有些行缺失了。你希望找到这3个time匹配的所有行，忽略任何不匹配的行，就像前面图中所示的那样。

下面是一段Python代码，用来找到所有的匹配行：

```
def PrintStockTransactions():
    stock_iter = db_read("SELECT time, ticker_symbol FROM ...")
    price_iter = ...
    num_shares_iter = ...

    # Iterate through all the rows of the 3 tables in parallel.
    while stock_iter and price_iter and num_shares_iter:
        stock_time = stock_iter.time
        price_time = price_iter.time
        num_shares_time = num_shares_iter.time

        # If all 3 rows don't have the same time, skip over the oldest row
        # Note: the "<=" below can't just be "<" in case there are 2 tied-oldest.
        if stock_time != price_time or stock_time != num_shares_time:
            if stock_time <= price_time and stock_time <= num_shares_time:
                stock_iter.NextRow()
            elif price_time <= stock_time and price_time <= num_shares_time:
                price_iter.NextRow()
            elif num_shares_time <= stock_time and num_shares_time <= price_time:
                num_shares_iter.NextRow()
            else:
                assert False  # impossible
            continue

        assert stock_time == price_time == num_shares_time

        # Print the aligned rows.
        print "@", stock_time,
        print stock_iter.ticker_symbol,
        print price_iter.price,
        print num_shares_iter.number_of_shares

        stock_iter.NextRow()
        price_iter.NextRow()
        num_shares_iter.NextRow()
```

这个例子中的代码能运行，但是在循环中为了跳过不匹配的行做了很多事情。你的脑海中也许闪过了一些警告：“这么做不会丢失一些行吗？它的迭代器会不会越过数据流的结尾”？

那么如何来把它变得更可读呢？

用自然语言描述解决方案

再一次，让我们退一步来用自然语言描述我们要做的事情：

> 我们并行地读取三个行迭代器。
> 只要这些行不匹配，向前找直到它们匹配。
> 然后输出匹配的行，再继续向前。
> 一直做直到没有匹配的行。

回头看看原来的代码，最乱的部分就是处理“向前找直到它们匹配”的语句块。为了让代码表现得更清楚，我们可以把所有这些乱糟糟的逻辑抽取到，名叫AdvanceToMatchingTime()的新函数中。

下面是代码的新版本，它使用了新的函数：

```
def PrintStockTransactions():
    stock_iter = ...
    price_iter = ...
    num_shares_iter = ...

    while True:
        time = AdvanceToMatchingTime(stock_iter, price_iter, num_shares_iter)
        if time is None:
            return

        # Print the aligned rows.
        print "@", time,
        print stock_iter.ticker_symbol,
        print price_iter.price,
        print num_shares_iter.number_of_shares
        stock_iter.NextRow()
        price_iter.NextRow()
        num_shares_iter.NextRow()
```

如你所见，这段代码容易理解得多，因为我们隐藏了所有行对齐的混乱细节。

递归地使用这种方法

很容易想象你将如何编写AdvancToMatchingTime()——最坏的情况就是它看上去和第一个版本中难看的代码块很像：

```
def AdvanceToMatchingTime(stock_iter, price_iter, num_shares_iter):
    # Iterate through all the rows of the 3 tables in parallel.
    while stock_iter and price_iter and num_shares_iter:
        stock_time = stock_iter.time
        price_time = price_iter.time
        num_shares_time = num_shares_iter.time

        # If all 3 rows don't have the same time, skip over the oldest row
        if stock_time != price_time or stock_time != num_shares_time:
            if stock_time <= price_time and stock_time <= num_shares_time:
                stock_iter.NextRow()
            elif price_time <= stock_time and price_time <= num_shares_time:
                price_iter.NextRow()
            elif num_shares_time <= stock_time and num_shares_time <= price_time:
                num_shares_iter.NextRow()
            else:
                assert False  # impossible
            continue

        assert stock_time == price_time == num_shares_time
```

```
        return stock_time
```

但是让我们把我们的方法同样应用于AdvanceToMatchingTime()来进改这段代码。下面是对于这个函数所要做的事情的描述：

```
看一下每个当前行：如果它们匹配，那么就完成了。
否则，向前移动任何"落后"的行。
一直这样做直到所有行匹配（或者其中一个迭代器结束）
```

这个描述清晰得多，并且比以前的代码更优雅。一件值得注意的事情是描述从未提及stock_iter或者其他解决问题的细节。这意味着我们可以同时把变量重命名得更简单，更通用。下面是这样做后得到的代码：

```
def AdvanceToMatchingTime(row_iter1, row_iter2, row_iter3):
    while row_iter1 and row_iter2 and row_iter3:
        t1 = row_iter1.time
        t2 = row_iter2.time
        t3 = row_iter3.time

        if t1 == t2 == t3:
            return t1

        tmax = max(t1, t2, t3)

        # If any row is "behind," advance it.
        # Eventually, this while loop will align them all.
        if t1 < tmax: row_iter1.NextRow()
        if t2 < tmax: row_iter2.NextRow()
        if t3 < tmax: row_iter3.NextRow()

    return None  # no alignment could be found
```

如你所见，这段代码比以前清楚得多。该算法变得更简单，并现在那种微妙的比较更少了。我们用了像t1这样的短名字，却不用再考虑那些具体涉及的数据库字段。

总结

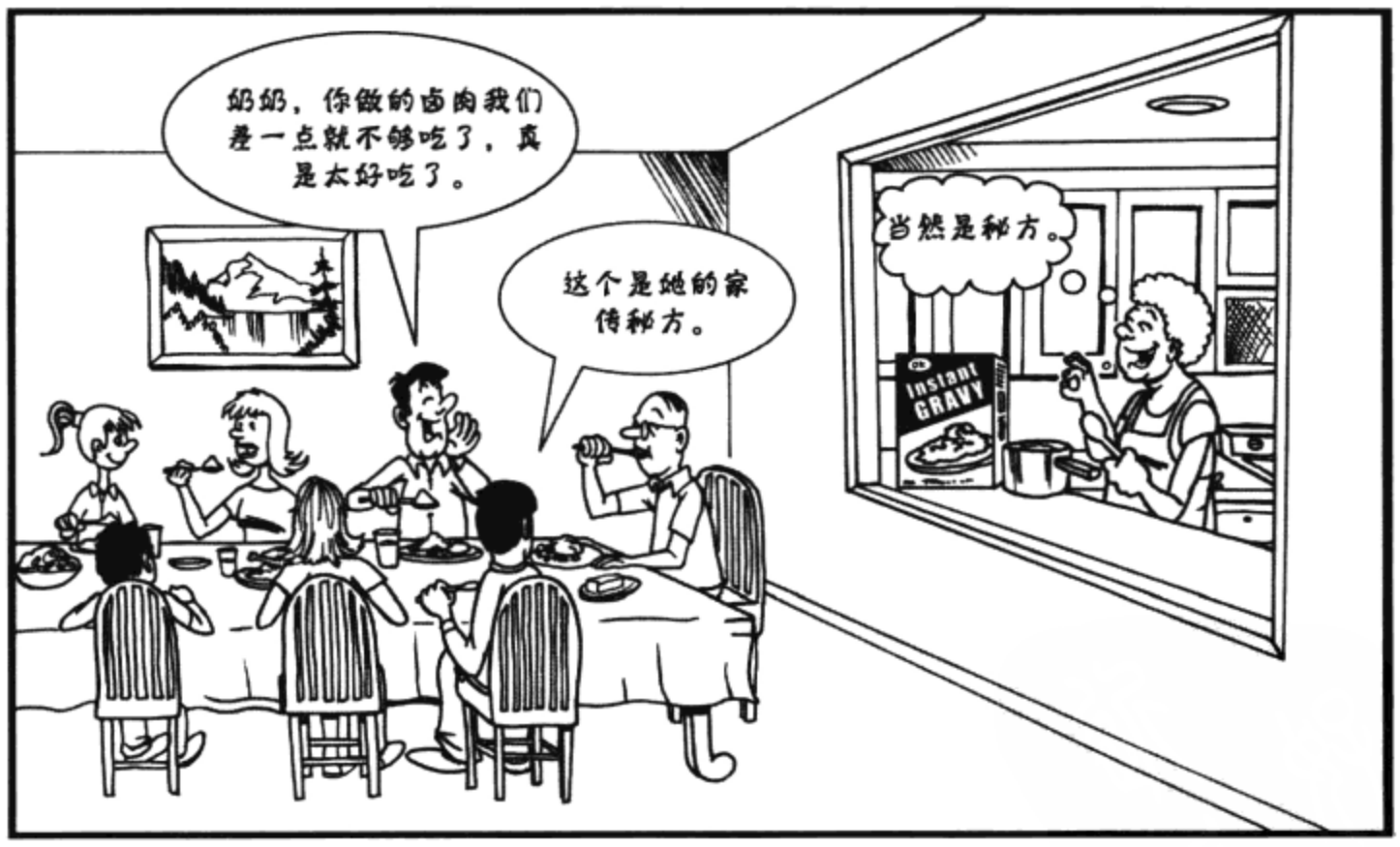
本章讨论了一个简单的技巧，用自然语言描述程序然后用这个描述来帮助你写出更自然的代码。这个技巧出人意料地简单，但很强大。看到你在描述中所用的词和短语还可以帮助你发现哪些子问题可以拆分出来。

但是这个“用自然语言说事情”的过程不仅可以用于写代码。例如，某个大学计算机实验室的规定声称当有学生需要别人帮它调试程序时，他首先要对房间角落的一只专用的泰迪熊解释他遇到的问题。令人惊讶的是，仅仅通过大声把问题描述出来，往往就能帮这个学生找到解决的办法。这个技巧叫做“橡皮鸭技术”。

另一个看待这个问题的角度是：如果你不能把问题说明白或者用词语来做设计，估计是缺少了什么东西或者什么东西缺少定义。把一个问题（或想法）变成语言真的可以让它更具体。

第13章

少写代码



知道什么时候不写代码可能对于一个程序员来讲是他所要学习的最重要的技巧。你所写的每一行代码都是要测试和维护的。通过重用库或者减少功能，你可以节省时间并且让你的代码库保持精简节约。

关键思想

最好读的代码就是没有代码。

别费神实现那个功能——你不会需要它

当你开始一个项目，自然会很兴奋并且想着你希望实现的所有很酷的功能。但是程序员倾向于高估有多少功能真的对于他们的项目来讲是必不可少的。很多功能结果没有完成，或者没有用到，也可能只是让程序更复杂。

程序员还倾向于低估实现一个功能所要花的工夫。我们乐观地估计了实现一个粗糙原型所要花的时间，但是忘记了在将来代码库的维护、文件以及后增的“重量”所带来的额外时间。

质疑和拆分你的需求

不是所有的程序都需要运行得快，100%准确，并且能处理所有的输入。如果你真的仔细检查你的需求，有时你可以把它削减成一个简单的问题，只需要较少的代码。让我们来看一些例子。

例子：商店定位器

假设你要给某个生意写个“商店定位器”。你以为你的需求是：

> 对于任何给定用户的经度/纬度，找到距离该经度/纬度最近的商店。

为了100%正确地实现，你要处理：

- 当位置处于国际日期分界线两侧的情况。
- 接近北极或南极的位置。
- 按“每英里所跨经度”不同，处理地球表面的曲度。

处理所有这些情况需要相当多的代码。

然而，对于你的应用程序来讲，只有在德州的30家店。在这么小的范围里，上面列出的三个问题并不重要。结果是，你可以把需求缩减为：

对于德州附近的用户，在德州找到（近似）最近的商店。

解决这个问题很简单，因为你只要遍历每个商店并计算它们与这个经纬度之间的欧几里得距离就可以了。

例子：增加缓存

我们曾有一个Java程序，它经常要从磁盘读取对象。这个程序的速度受到了这些读取操作的限制，因此我们希望能实现缓存之类的功能。一个典型的读取序列像是这样：

```
读对像A
读对像A
读对像A
读对像B
读对像B
读对像C
读对像D
读对像D
```

如你所见，有很多对同一对象的重复访问，因此缓存绝对会有帮助。

当面对这样的问题时，我们首先的直觉是使用那种丢掉最近没有使用的条目的缓存。在我们的库中没有这样的缓存，因此我们必须实现一个自己的。这不是问题，因为我们以前实现过这种数据结构（它包含一个哈希表和一个单向链表——一共有大约100行代码）。

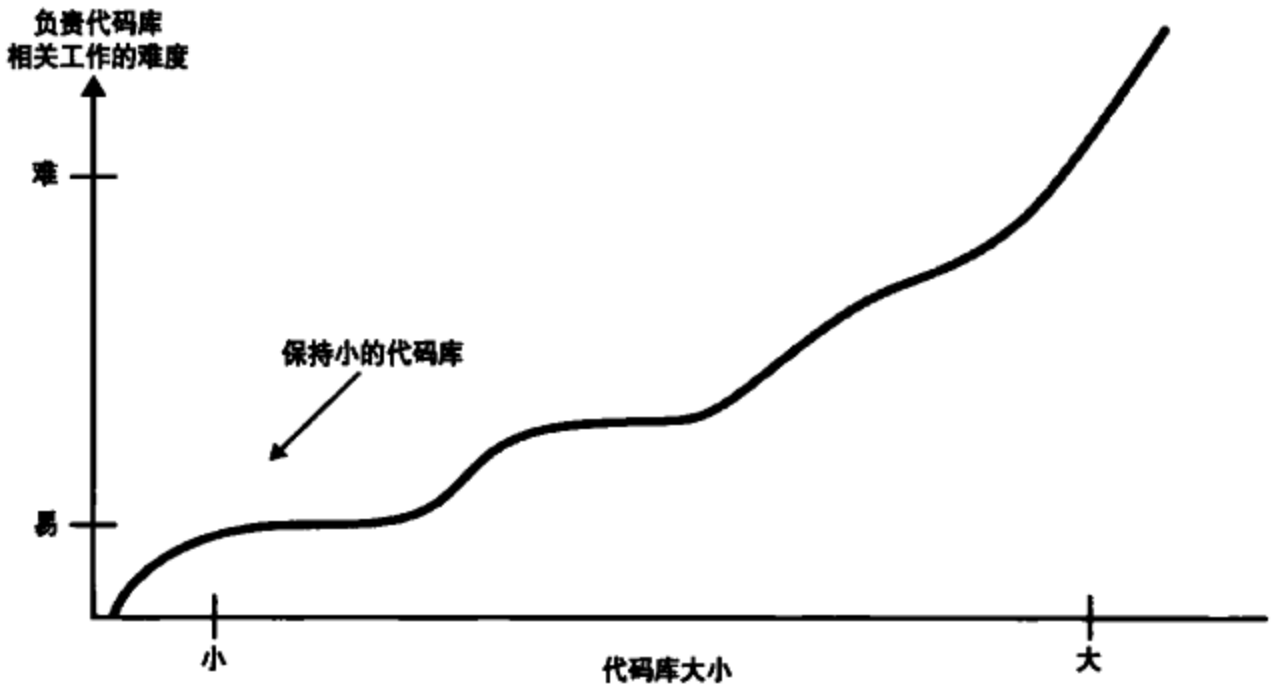
然而，我们注意到这种重复访问总是处于一行的。因此不要实现LRU（最近最少使用）缓存，我们只要实现有一个条目的缓存：

```
DiskObject lastUsed;  // class member

DiskObject lookUp(String key) {
    if (lastUsed == null || !lastUsed.key().equals(key)) {
        lastUsed = loadDiskObject(key);
    }

    return lastUsed;
}
```

这样我们就用很少的代码得到了90%的好处，这段程序所占的内存也很小。

怎么说“减少需求”和“解决更简单的问题”的好处都不为过。需求常常以微妙的方式互相影响。这意味着解决一半的问题可能只需要花四分之一的工夫。

保持小代码库



在你第一次开始一个软件项目，并且只有一两个源文件时，一切都很顺利。编译和运行代码转眼就完成，很容易做改动，并且很容易记住每个函数或类定义在哪里。

然后，随着项目的增长，你的目录中加进了越来越多的源文件。很快你就需要多个目录来组织它们了。很难再记得哪个函数调用了哪个函数，而且跟踪bug也要做多一点的工作。

最后，你就有了很多源代码分布在很多不同的目录中。项目很大，没有一个人自己全部理解它。增加新功能变得很痛苦，而且使用这些代码很费力还令人不快。

我们所描述的是宇宙的自然法则——随着任何坐标系统的增长，把它粘合在一起所需的复杂度增长得更快。

最好的解决办法就是“**让你的代码库越小，越轻量级越好**”，就算你的项目在增长。那么你就要：

- 创建越多越好的“工具”代码来减少重复代码（见第10章）。
- 减少无用代码或没有用的功能（见下图）。
- 让你的项目保持分开的子项目状态。
- 总的来说，要小心代码的“重量”。让它保持又轻又灵。

删除没用的代码

园丁经常修剪植物以让它们活着并且生长。同样地，修剪掉碍事和没用的代码也是个好主意。

一旦代码写好后，程序员往往不情愿删除它，因为它代表很多实际的工作量。删掉它可能意味着承认在上面所花的时间就是浪费。不要这么想！这是一个有创造性的领域——摄影家、作者和电影制版人也不会保留他们所有的工作。

删除独立的函数很简单，但有时“无用代码”实际上交织在你的项目中，你并不知情。下面是一些例子：

- 你一开始把系统设计成能处理多语言文件名，现在代码中到处都充满了转换代码。然而，那段代码不能很好地工作，实现上你的程序也从来没有用到过任何多语言文件名。
- 为什么不删除这个功能呢？
- 你希望你的程序在内存耗尽的情况下仍能工作，因此你有很多要小聪明的逻辑来试着从内存耗尽的情况下恢复。这是个好主意，但在实践中，当系统内存耗尽时，你的程序将变成不稳定的僵尸——所有的核心功能都不可用，再点一下鼠标它就死了。

  为什么不通过一句简单的提示“系统内存不足，抱歉”并删除所有内存不足的代码，终止程序呢？

熟悉你周边的库

很多时候，程序员就是不知道现有的库可以解决他们的问题。或者有时，它们忘了库可以做什么。知道你的库能做什么以便你可以使用它，这一点很重要。

这里有一条比较中肯的建议：**每隔一段时间，花15分钟来阅读标准库中的所有函数/模块/类型的名字。**这包括C++标准模板库（STL）、Java API、Python内置的模块以及其他内容。

这样做的目的不是记住整个库。这只是为了了解有什么可以用的，以便下次你写新代码时会想：“等一下，这个听起来和我在API中见到的东西有点像……”我们相信提前做这种准备很快就会得到回报，起码因为你会更倾向于使用库了。

例子：Python中的列表和集合

假设你有一个使用Python写的列表（如[2,1,2]），你想要一个拥有不重复元素的列表（在上例中，就是[2,1]）。你可以用字典来完成这个任务，它有一个键列表保证元素是唯一的：

```
def unique(elements):
    temp = {}
    for element in elements:
        temp[element] = None  # The value doesn't matter.
    return temp.keys()

unique_elements = unique([2,1,2])
```

但是你可以用较少人知道的集合类型：

```
unique_elements = set([2,1,2]) # Remove duplicates
```

这个对象是可以枚举的，就像一个普通的list一样。如果你很想要一个list对象，你可以用：

```
unique_elements = list(set([2,1,2])) # Remove duplicates
```

很明显，这里集合才是正确的工具。但如果你不知道set类型，你可能会写出像前面unique()一样的代码。

为什么重用库有这么大的好处

一个常被引用的统计结果是一个平均水平的软件工程师每天写出10行可以放到最终产品中的代码。当程序员们刚一听到这个，他们根本不相信——“10行代码？我一分钟就写出来了！”

这里的关键词是“最终产品中的”。在一个成熟的库中，每一行代码都代表相当大量的设计、调试、重写、文档、优化和测试。任何经受了这样达尔文进化过程一样的代码行就是很有价值的。这就是为什么重用库有这么大的好处，不仅节省时间，还少写了代码。

例子：使用Unix工具而非编写代码

当一个Web服务器经常性地返回HTTP响应代码4xx或者5xx，这是个有潜在问题的信号（4xx是客户端错误；5xx是服务器端错误）。所以我们想要编写个程序来解析一个Web服务器的访问日志并找出哪些URL导致了大部分的错误。

访问日志一般看起来像是这个样子：

```
1.2.3.4 example.com [24/Aug/2010:01:08:34] "GET /index.html HTTP/1.1" 200 ...
2.3.4.5 example.com [24/Aug/2010:01:14:27] "GET /help?topic=8 HTTP/1.1" 500 ...
3.4.5.6 example.com [24/Aug/2010:01:15:54] "GET /favicon.ico HTTP/1.1" 404 ...
...
```

基本上，它们都包含有以下格式的行：

```
browser-IP host [date] "GET /url-path HTTP/1.1" HTTP-response-code ...
```

写个程序来找出带有4xx或5xx响应代码的最常见网址可能只要20行用C++或者Java写的代码。然而，在Unix中，可以输入下面的命令：

```
cat access.log | awk '{ print $5 " " $7 }' | egrep "[45]..$" \
| sort | uniq -c | sort -nr
```

就会产生这样的输出：

```
95 /favicon.ico 404
13 /help?topic=8 500
11 /login 403
...
<count> <path> <http response code>
```

这条命令的好处在于我们不必写任何“真正”的代码，或者向源代码管理库中签入任何东西。

总结

冒险、兴奋——绝地武士追求的并不是这些。

——尤达大师

本章是关于写越少代码越好的。每行新的代码都需要测试、写文档和维护。另外，代码库中的代码越多，它就越“重”，而且在其上开发就越难。

你可以通过以下方法避免编写新代码：

- 从项目中消除不必要的功能，不要过度设计。
- 重新考虑需求，解决版本最简单的问题，只要能完成工作就行。
- 经常性地通读标准库的整个API，保持对它们的熟悉程度。

第四部分

精选话题

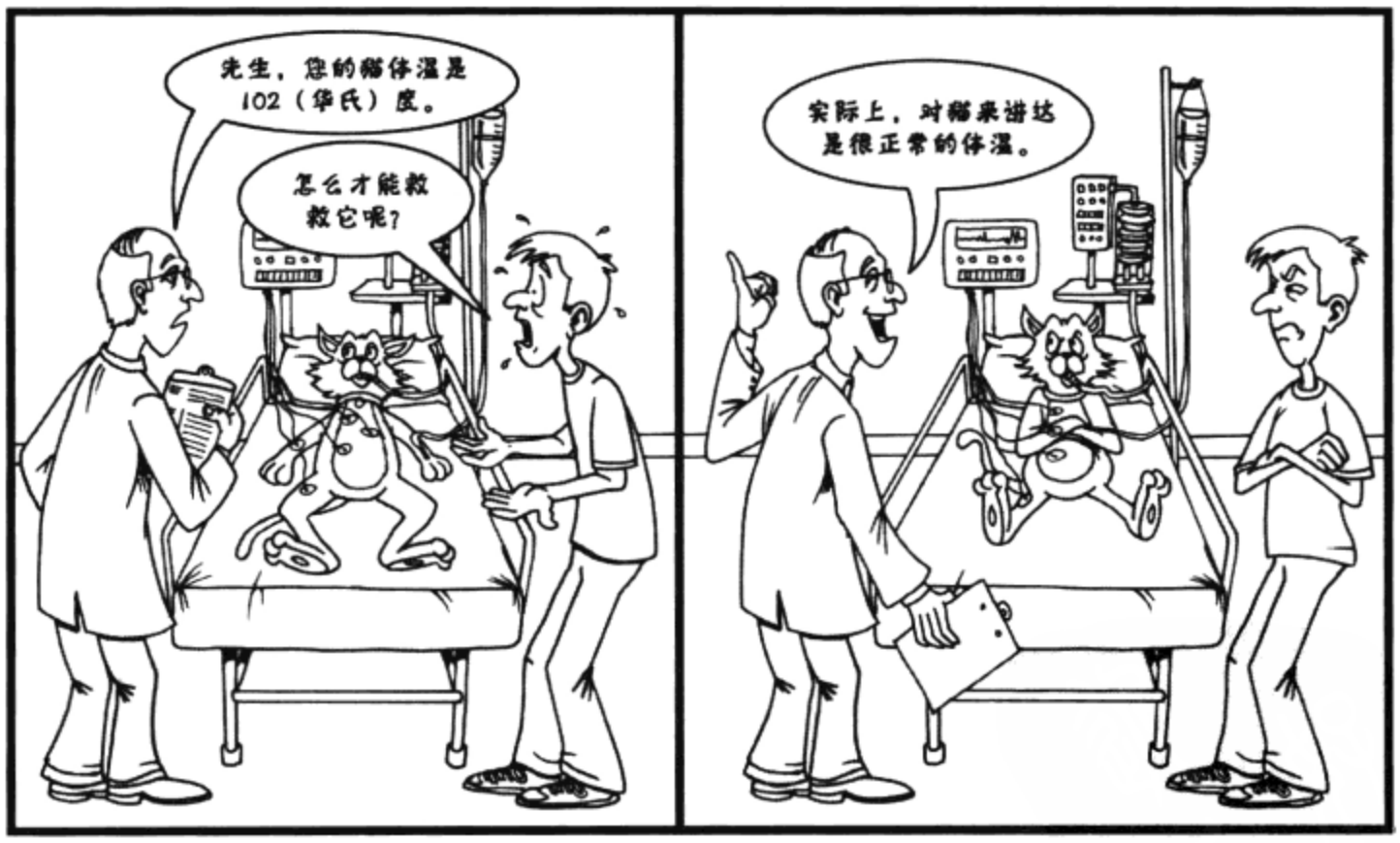
本书前三部分覆盖了使代码简单易读的各种技巧。在该部分中，我们会把这些技术应用在两个精选出的话题中。

首先，我们会讨论测试——如何同时写出有效而又可读的测试。

然后，我们会历经一个设计和实现专门设计的数据结构的过程（一个“分钟/小时计数器”），这将是一个性能与好的设计以及可读性互动的例子。

第14章

测试与可读性



在本章中，我们会揭示一些写出整洁并且有效测试的简单技巧。

测试对不同的人意味着不同的事。在本章中，“测试”是指任何仅以检查另一段（“真实”）代码的行为为目的的代码。我们会关注测试的可读性方面，不会讨论你是否应该在写真实代码之前写测试代码（“测试驱动的开发”）或者测试开发的其他哲学方面。

使测试易于阅读和维护

测试代码的可读性和非测试代码是同样重要的。其他程序员会经常来把测试代码看做非正式的文档，它记录了真实代码如何工作和应该如何使用。因此如果测试很容易阅读，使用者对于真实代码的行为会有更好的理解。

> **关键思想**
>
> **测试应当具有可读性，以便其他程序员可以舒服地改变或者增加测试。**

当测试代码多得让人望而止步，会发生下面的事情：

- **程序员会不敢修改真实代码。**“啊，我们不想纠结于那段代码，更新它的那些测试将会是个噩梦！”
- 当增加新代码时，**程序员不会再增加新的测试。**一段时间后，测试的模块越来越少，你不再对它有信心。

相反，你希望鼓励你代码的使用者（尤其是你自己！）习惯于测试代码。他们应该能在新改动破坏已有测试时做出分析，并且应该感觉增加新测试很容易。

这段测试什么地方不对

在代码库中，有一个函数，它对于一个打过分的搜索结果列表进行排序和过滤。下面是函数的声明：

```
// Sort 'docs' by score (highest first) and remove negative-scored documents.
void SortAndFilterDocs(vector<ScoredDocument>* docs);
```

该函数的测试最初如下所示：

```
void Test1() {
    vector<ScoredDocument> docs;
    docs.resize(5);
    docs[0].url = "http://example.com";
    docs[0].score = -5.0;
    docs[1].url = "http://example.com";
    docs[1].score = 1;
```

```
    docs[2].url = "http://example.com";
    docs[2].score = 4;
    docs[3].url = "http://example.com";
    docs[3].score = -99998.7;
    docs[4].url = "http://example.com";
    docs[4].score = 3.0;

    SortAndFilterDocs(&docs);

    assert(docs.size() == 3);
    assert(docs[0].score == 4);
    assert(docs[1].score == 3.0);
    assert(docs[2].score == 1);
}
```

这段测试代码起码有8个不同的问题。在本章结束前，你将能够找出并改正它们。

使这个测试更可读

做为一条普遍的测试原则，你应当**“对使用者隐去不重要的细节，以便更重要的细节会更突出”**。

前一节中的测试代码明显违反了这条原则。该测试的所有细节都摆在那里，比如像建立一个vector<ScoredDocument>这样不重要的细枝末节。例子中大部分代码都包含url、score和docs[]，这些只是背后的C++对象如何创建的细节，不是关于所测试内容的高层次描述。

作为清理这些代码的第一步，可以创建一个这样的辅助函数：

```
void MakeScoredDoc(ScoredDocument* sd, double score, string url) {
    sd->score = score;
    sd->url = url;
}
```

使用这个函数，测试代码变得紧凑一点了：

```
void Test1() {
    vector<ScoredDocument> docs;
    docs.resize(5);
    MakeScoredDoc(&docs[0], -5.0, "http://example.com");
    MakeScoredDoc(&docs[1], 1, "http://example.com");
    MakeScoredDoc(&docs[2], 4, "http://example.com");
    MakeScoredDoc(&docs[3], -99998.7, "http://example.com");
    ...
}
```

但是它还是不够好——在我们面前还是有不重要的细节。例如，参数“http://example.com”看着很不自然。它每次都是一样的，而且具体URL是什么根本没关系——只要填进一个有效的ScoredDocument就可以了。

我们被迫要看的另一个不重要的细节是docs.resize(5)和&docs[0]、&docs[1]等。让我们修改辅助函数来做更多事情，并给它命名为AddScoredDoc()：

```
void AddScoredDoc(vector<ScoredDocument>& docs, double score) {
    ScoredDocument sd;
    sd.score = score;
    sd.url = "http://example.com";
    docs.push_back(sd);
}
```

使用这个函数，测试代码更紧凑了：

```
void Test1() {
    vector<ScoredDocument> docs;
    AddScoredDoc(docs, -5.0);
    AddScoredDoc(docs, 1);
    AddScoredDoc(docs, 4);
    AddScoredDoc(docs, -99998.7);
    ...
}
```

这段代码已经好多了，但仍然不满足“高度易读和易写”测试的要求。如果你希望增加一个测试，其中用到一组新的scored docs，这会需要大量的拷贝和粘贴。那么，我们怎么样来进一步改进它呢？

创建最小的测试声明

要改进这段测试代码，让我们使用从第12章学来的技巧。让我们用自然语言来描述我们的测试要做什么：

```
我们有一个文档列表，它们的分数为[-5，1，4，-99998.7，3]。
在SortAndFilterDocs()之后，剩下的文档应当有的分数是[4，3，1]，而且顺序也是这样。
```

如你所见，在描述中没有在任何地方提及vector<ScoredDocument>。这里最重要的是分数数组。理想的情况下，测试代码应该看起来这样：

```
CheckScoresBeforeAfter("-5，1，4，-99998.7，3"，  "4，3，1");
```

我们可以把这个测试的基本内容精练成一行代码！

然而这没什么大惊小怪的。大多数测试的基本内容都能精练成“**对于这样的输入/情形，期望有这样的行为/输出**”。并且很多时候这个目的可以用一行代码来表达。这除了让代码紧凑而又易读，让测试的表述保持很短还会让增加测试变得很简单。

实现定制的“微语言”

注意到CheckScoresBeforeAfter()需要两个字符串参数来描述分数数组。在较新版本的C++中，可以这样传入数组常量：

```
CheckScoresBeforeAfter({-5, 1, 4, -99998.7, 3}, {4, 3, 1});
```

因为当时我们还不能这么做，所以我们就把分数都放在字符串中，用逗号分开。为了让这个方法可行，CheckScoresBeforeAfter()就不得不解析这些字符串参数。

一般来讲，定义一种定制的微语言可能是一种占用很少的空间来表达大量信息的强大方法。其他例子包含printf()和正则表达式库。

在本例中，编写一些辅助函数来解析用逗号分隔的一系列数字应该不会非常难。下面是CheckSocresBeforeAfter()的实现方式：

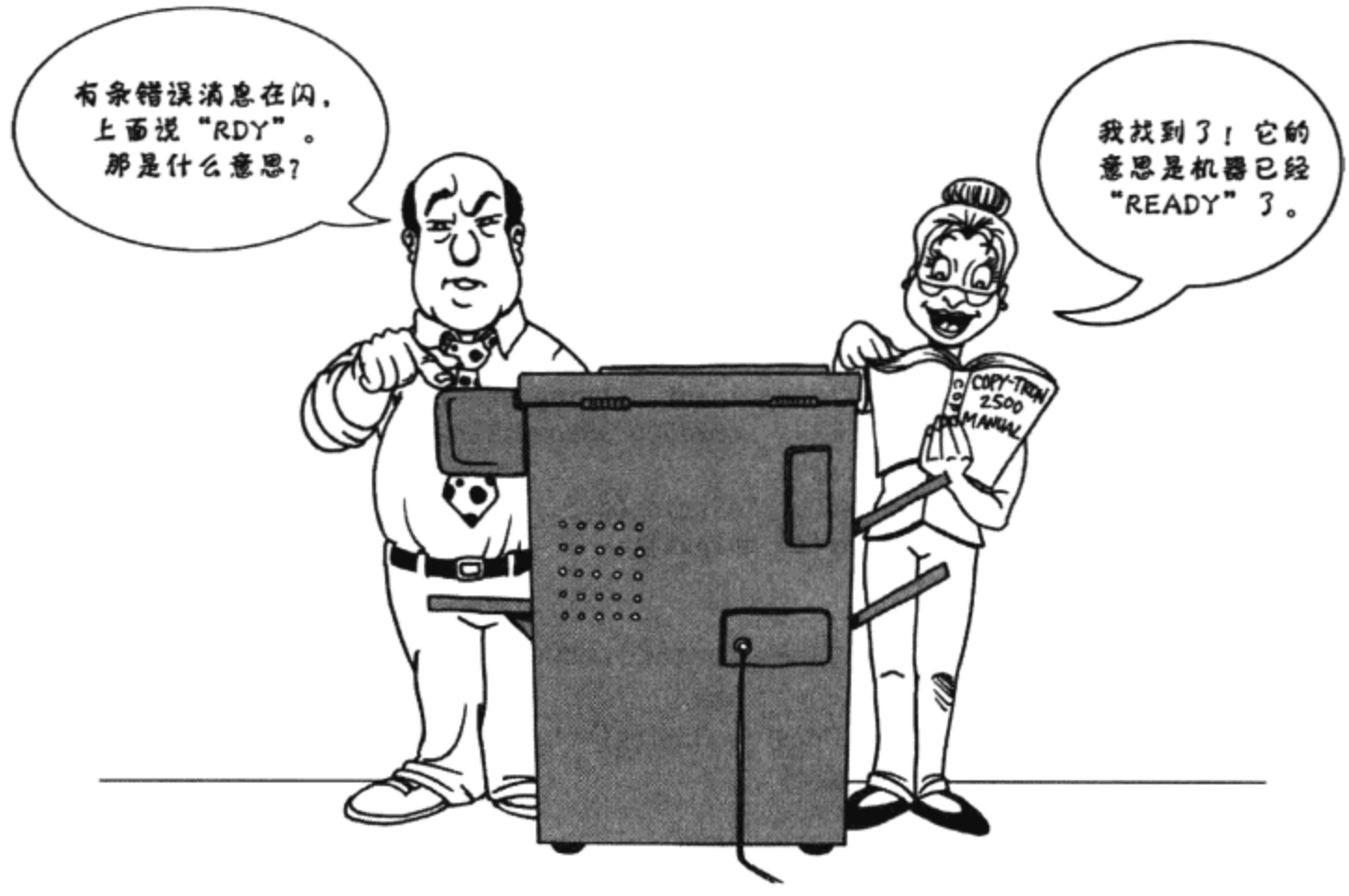
```
void CheckScoresBeforeAfter(string input, string expected_output) {
    vector<ScoredDocument> docs = ScoredDocsFromString(input);
        SortAndFilterDocs(&docs);
        string output = ScoredDocsToString(docs);
        assert(output == expected_output);
}
```

为了更完善，下面是用来在string和vector<ScoredDocument>之间转换的辅助函数：

```
vector<ScoredDocument> ScoredDocsFromString(string scores) {
    vector<ScoredDocument> docs;

    replace(scores.begin(), scores.end(), ',', ' ');

    // Populate 'docs' from a string of space-separated scores.
    istringstream stream(scores);
    double score;
    while (stream >> score) {
        AddScoredDoc(docs, score);
    }

    return docs;

}

string ScoredDocsToString(vector<ScoredDocument> docs) {
    ostringstream stream;
    for (int i = 0; i < docs.size(); i++) {
        if (i > 0) stream << ", ";
        stream << docs[i].score;
    }

    return stream.str();
}
```

乍一看这里有很多代码，但是它能带给你难以想象的能力。因为你可以只调用CheckScoresBeforeAfter()一次就写出整个测试，你会更倾向于增加更多的测试（就像我们在本章后面要做的那样）。

让错误消息具有可读性



现在的代码已经很不错了，但是当assert(output ++ expected_output)这一行失败时会发生什么呢？它会产生一行这样的错误消息：

```
Assertion failed: (output == expected_output),
    function CheckScoresBeforeAfter, file test.cc, line 37.
```

显然，如果你看到了这个错误，你会想：“output和expected_output出错时的值是什么呢？”

更好版本的assert()

幸运的是，大部分语言和库都有更高级版本的assert()给你用。所以不用这样写：

```
assert(output == expected_output);
```

你可以使用C++的Boost库：

```
BOOST_REQUIRE_EQUAL(output, expected_output)
```

现在，如果测试失败，你会得到更具体的消息：

```
test.cc(37): fatal error in "CheckScoresBeforeAfter": critical check
    output == expected_output failed ["1, 3, 4" != "4, 3, 1"]
```

这更有帮助。

如果有的话，你应该使用这些更有帮助的断言方法。每当你的测试失败时，你就会受益。

其他语言中更好的ASSERT()选择

在Python中，内置语句assert a == b会产生一条简单的错误消息：

```
File "file.py", line X, in <module>
    assert a == b
AssertionError
```

不如用unittest模块中的assertEqual()方法：

```
import unittest

class MyTestCase(unittest.TestCase):
    def testFunction(self):
        a= 1
        b= 2
        self.assertEqual(a, b)

if __name__ == '__main__':
     unittest.main()
```

它会给出这样的错误消息：

```
File "MyTestCase.py", line 7, in testFunction
    self.assertEqual(a, b)
AssertionError: 1 != 2
```

无论你用什么语言，都可能会有一个库/框架（例如XUnit）来帮助你。了解那些库对你有好处！

手工打造错误消息

使用BOOST_REQUIRE_EQUAL()，可以得到更好的错误消息：

```
output == expected_output failed ["1, 3, 4" != "4, 3, 1"]
```

然而，这条消息还能进一步改进。例如，如果能看到原本触发这个错误的输入一定会有帮助。理想的错误消息可以像是这样：

```
CheckScoresBeforeAfter() failed,
   Input:           "-5, 1, 4, -99998.7, 3"
   Expected Output: "4, 3, 1"
   Actual Output:   "1, 3, 4"
```

如果这就是你想要的，那么就写出来吧！

```
void CheckScoresBeforeAfter(...) {
    ...
    if (output != expected_output) {
        cerr << "CheckScoresBeforeAfter() failed," << endl;
        cerr << "Input:           \"" << input << "\"" << endl;
        cerr << "Expected Output: \"" << expected_output << "\"" << endl;
        cerr << "Actual Output:   \"" << output << "\"" << endl;
        abort();
}
```

这个故事的寓意就是错误消息应当越有帮助越好。有时，通过建立“定制的断言”来输出你自己的消息是最好的方式。

选择好的测试输入

有一门为测试选择好的输入的艺术。我们现在看到的这些是很随机的：

```
CheckScoresBeforeAfter("-5, 1, 4, -99998.7, 3",  "4, 3, 1");
```

如何选择好的输入值呢？好的输入应该能彻底地测试代码。但是它们也应该很简单易读。

关键思想

基本原则是，你应当选择一组最简单的输入，它能完整地使用被测代码。

例如，假设我们刚刚写了：

```
CheckScoresBeforeAfter("1, 2, 3", "3, 2, 1");
```

尽管这个测试很简单，它没有测试SortAndFilterDocs()中“过滤掉负的分数”这一行为。如果在代码的这部分中有bug，这个输入不会触发它。

另一个极端是，假设我们这样写测试：

```
CheckScoresBeforeAfter("123014, -1082342, 823423, 234205, -235235",
                       "823423, 234205, 123014");
```

这些值复杂得没有必要。（并且甚至也没能完整地测试代码。）

简化输入值

那么我们能做些什么来改进这些输入值呢？

```
CheckScoresBeforeAfter("-5, 1, 4, -99998.7, 3",  "4, 3, 1");
```

嗯，首先能可能会注意到的是非常“嚣张”的值–99 998.7。这个值的含义只是“任何负

数”，所以简单的值就是–1。（如果–99998.7是想说明它是个“非常负的负数”，明确地用像–1e100这样的值会更好。）

关键思想

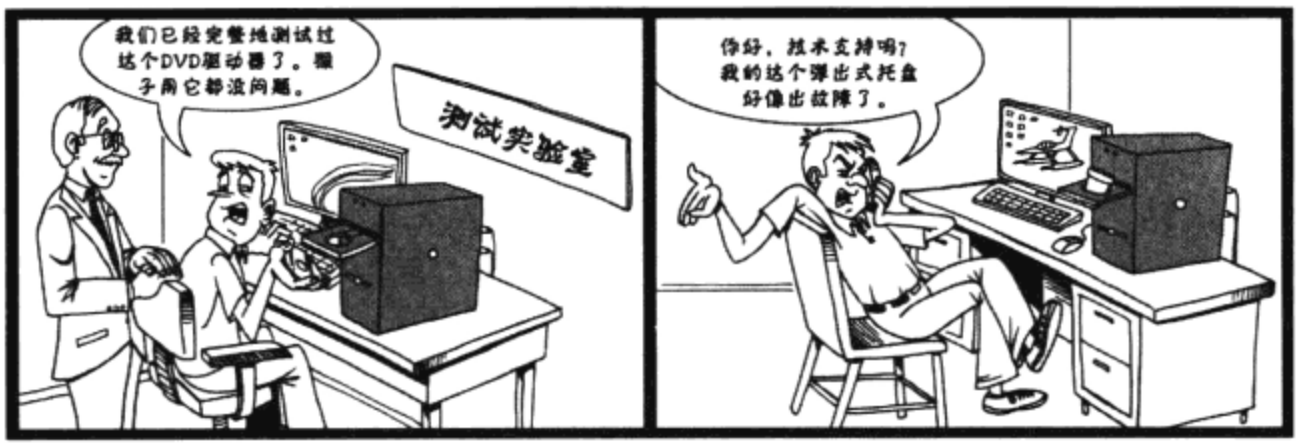
又简单又能完成工作的测试值更好。

测试中其他的值还不算太差，但既然我们都已经改了，那么把它们也简化成尽量简单的整数。并且，只要有一个负数来测试负数会被移除就可以了。下面是测试的新版本：

```
CheckScoresBeforeAfter("1, 2, -1, 3", "3, 2, 1");
```

我们简化了测试的值，却并没有降低它的效果。

大型“破坏性”测试



对于大的、不切实际的输入进行测试当然是有价值的。例如，你可能会想包含一个这样的测试：

```
CheckScoresBeforeAfter("100, 38, 19, -25, 4, 84, [lots of values] ...",
                       "100, 99, 98, 97, 96, 95, 94, 93, ...");
```

这样的大型输入在发现bug方面很有作用，比如缓冲区溢出或者其他出乎意料的情况。

但是这样的代码又大多看上去又吓人，对代码的压力测试来讲并无很好的效果。相反，用编程的方法来生成大型输入会更有效果，例如，生产100 000个值。

一个功能的多个测试

与其建立单个“完美”输入来完整地执行你的代码，不如写多个小测试，后者往往会更容易、更有效并且更有可读性。

每个测试都应把代码推往某一个方向，尝试找到某种bug。例如，下面有SortAndFilterDocs()的4个测试：

```
CheckScoresBeforeAfter("2, 1, 3", "3, 2, 1");          // Basic sorting
CheckScoresBeforeAfter("0, -0.1, -10", "0");           // All values < 0 removed
CheckScoresBeforeAfter("1, -2, 1, -2", "1, 1");        // Duplicates not a problem
CheckScoresBeforeAfter("", "");                        // Empty input OK
```

如果要非常地彻底，还可以写更多的测试。有分开的测试用例还可以使下一个负责代码相关工作的人更轻松。如果有人不小心引入了一个bug，测试的失败会指向那个具体的失败测试用例。

为测试函数命名

测试代码一般以函数的形式组织起来——你所测试的每个方法和/或情形对应一个测试函数。例如，测试SortAndFilterDocs()的测试代码是在函数Test1()中：

```
void Test1() {
   ...
}
```

为测试函数选择一个好名字可能看上去很无聊而且也无关紧要，但是不要因此而诉诸没有意义的名字，像是Test1()、Test2()这样。

反而，你应当用这个名字来描述这个测试的细节。如果读测试代码的人可以很快搞明白这些的话，这一点尤其便利：

- 被测试的类（如果有的话）
- 被测试的函数
- 被测试的情形或bug

一种构造好的测试函数名的简单方式是把这些信息拼接在一起，可能再加上一个“Test_”前缀。

例如，不要用Test1()这个名字，可以用Test_<FuncitonName>()这样的格式：

```
void Test_SortAndFilterDocs() {
   ...
}
```

依照测试的精细程度不同，你可能会考虑为测试的每种情形写一个单独的测试函数。可以使用Test_<FunctionName>_<Situation>()这样的格式：

```
void Test_SortAndFilterDocs_BasicSorting() {
```

```
    ...
}

void Test_SortAndFilterDocs_NegativeValues() {
    ...
}

...
```

这里不要怕名字太长或者太繁琐。在你的整个代码库中不会调用这个函数，因此那此要避免使用长函数名的理由在这里并不适用。测试函数的名字的作用就像是注释。并且，如果测试失败了，大部分测试框架会输出其中断言失败的那个函数的名字，因此一个具有描述性的名字尤其有帮助。

请注意如果你在使用一个测试框架，可能它已经有方法命名的规则和规范了。例如，在Python的unittest模块中它需要测试方法的名字以"test"开头。

当为测试代码的辅助函数命名时，标明这个函数是否自身有任何断言或者只是一个普通的“对测试一无所知”的辅助函数。例如，在本章中，所有调用了assert()的辅助数都命名成Check...()。但是函数AddScoredDoc()就只是像普通辅助函数一样命名。

那个测试有什么地方不对

在本章的开头，我们声称在这个测试中至少有8个地方不对：

```
void Test1() {
    vector<ScoredDocument> docs;
    docs.resize(5);
    docs[0].url = "http://example.com";
    docs[0].score = -5.0;
    docs[1].url = "http://example.com";
    docs[1].score = 1;
    docs[2].url = "http://example.com";
    docs[2].score = 4;
    docs[3].url = "http://example.com";
    docs[3].score = -99998.7;
    docs[4].url = "http://example.com";
    docs[4].score = 3.0;

    SortAndFilterDocs(&docs);

    assert(docs.size() == 3);
    assert(docs[0].score == 4);
    assert(docs[1].score == 3.0);
    assert(docs[2].score == 1);
}
```

现在我们已经学到了一些编写更好测试的技巧，让我们来找出它们：

1. 这个测试很长，并且充满了不重要的细节，你可以用一句话来描述这个测试所做的事情，因此这条测试的语句不应该太长。
2. 增加新测试不会很容易。你会倾向于拷贝/粘贴/修改，这样做会让代码更长而且充满重复。
3. 测试失败的消息不是很有帮助。如果测试失败的话，它只是说`Assertion failed: docs.size() == 3`，这并没有为进一步调试提供足够的信息。
4. 这个测试想要同时测试完所有东西。它想要既测试对负数的过滤又测试排序的功能。把它们拆分成多个测试会更可读。
5. 这个测试的输入不是很简单。尤其是，样本分数-99998.7很“嚣张”，尽管它是什么值并不重要但是它会引起你的注意。一个简单的负数值就足够了。
6. 测试的输入没有彻底地执行代码。例如，它没有测试到当分数为0时的情况。（这种文档会过滤掉吗？）
7. 它没有测试其他极端的输入，例如空的输入向量、很长的向量，或者有重复分数的情况。
8. 测试的名字`Test1()`没有意义——名字应当能描述被测试的函数或情形。

对测试较好的开发方式

有些代码比其他代码更容易测试。对于测试来讲理想的代码要有明确定义的接口，没有过多的状态或者其他的“设置”，并且没有很多需要审查的隐藏数据。

如果你写代码的时候就知道以后你要为它为写测试的话，会发生有趣的事情：**你开始把代码设计得容易测试**！幸运的是，这样的编程方式一般来讲也意味着会产生更好的代码。对测试友好的设计往往很自然地会产生有良好组织的代码，其中不同的部分做不同的事情。

测试驱动开发

测试驱动开发（TDD）是一种编程风格，你在写真实代码之前就写出测试。TDD的支持者相信这种流程对没有测试的代码来讲会做出极大的质量改进，比写出代码之后再写测试要大得多。

这是一个争论很激烈的话题，我们不想搅进来。至少，我们发现仅通过在写代码时想着测试这件事就能帮助把代码写得更好。

但不论你是否使用TDD，其结果总是你用代码来测试另一些代码。本章旨在帮助你把测试做得既易读又易写。

在所有的把一个程序拆分成类和方法的途径中，解耦合最好的那一个往往就是最容易测试的那个。另一方面，假设你的程序内部联系很强，在类与类之间有很多方法的调用，并且所有的方法都有很多参数。不仅这个程序会有难以理解的代码，而且测试代码也会很难看，并且既难读又难写。

有很多“外部”组件（需要初始化的全局变量、需要加载的库或者配置文件等）对写测试来讲也是很讨厌的。

一般来讲，如果你在设计代码时发现：“嗯，这对测试来讲会是个噩梦”，这是个好理由让你停下来重新考虑这个设计。表14-1列出一些典型的测试和设计问题：

表14-1：可测试性差的代码的特征，以及它所带来的设计问题

特征	可测试性的问题	设计问题
使用全局变量	对于每个测试都要重置所有的全局状态（否则，不同的测试之间会互相影响）	很难理解哪些函数有什么副作用。没办法独立考虑每个函数，要考虑整个程序才能理解是不是所有的代码都能工作
对外部组件有大量依赖的代码	很难给它写出任何测试，因为要先搭起太多的脚手架。写测试会比效无趣，因此人们会避免写测试	系统会更可能因某一依赖失败而失败。对于改动来讲很难知道会产生什么样的影响。很难重构类。系统会有更多的失败模式，并且要考虑更多恢复路径
代码有不确定的行为	测试会很古怪，而且不可靠。经常失败的测试最终会被忽略	这种程序更可能会有条件竞争或者其他难以重现的bug。这种程序很难推理。产品中的bug很难跟踪和改正

另一方面，如果对于你的设计容易写出测试，那是个好现象。表14-2列出一些有益的测试和设计的特征。

表14-2：可测试性较好的代码的特征，以及它所产生的优秀设计

特征	对可测试性的好处	对设计的好处
类中只有很少或者没有内部状态	很容易写出测试，因为要测试一个方法只要较少的设置，并且有较少的隐藏状态需要检查	有较少状态的类更简单，更容易理解
类/函数只做一件事	要测试它只需要较少的测试用例	较小/较简单的组件更加模块化，并且一般来讲系统有更少的耦合

表14-2：可测试性较好的代码的特征，以及它所产生的优秀设计（续）

特征	对可测试性的好处	对设计的好处
每个类对别的类的依赖很少；低耦合	每个类可以独立地测试（比多个类一起测试容易得多）	系统可以并行开发。可以很容易修改或者删除类，而不会影响系统的其他部分
函数的接口简单，定义明确	有明确的行为可以测试。测试简单接口所需的工作量较少	接口更容易让程序员学习，并且重用的可能性更大

走得太远

对于测试的关注也会过多。下面是一些例子：

- **牺牲真实代码的可读性，只是为了使能测试。**把真实代码设计得具有可测试性，这应该是个双赢的局面：真实的代码变得简单而且低耦合，并且也更容易为它写测试。但是如果你仅仅是为了测试它而不得不在真实代码中插入很多难看的塞子，那肯定有什么地方不对了。
- **着迷于100%的测试覆盖率。**测试你代码的前面90%通常要比那后面的10%所花的工夫少。后面那10%包括用户接口或者很难出现的错误情况，其中bug的代价并不高，花工夫来测试它们并不值得。

  事实上你永远也不会达到100%的测试覆盖率。如果不是因为漏掉的bug，也可能是因为漏掉的功能或者你没想到说明书应该改一改。

  根据你的bug的成本不同，对于你花在测试代码上的开发时间有一个合理的范围。如果你在建一个网站原型，可能写任何测试都是不值得的。另一方面，如果你在为一架飞船或者一台医用设备编写控制器，测试可能是你的重点。
- **让测试成为产品开发的阻碍。**我们曾见过这样的情形，测试，本应只是项目的一个方面，却主导了整个项目。测试成了要敬畏的上帝，程序员只是走走这些仪式和过场，没有意识到他们在工程上宝贵的时间花在别的地方可能会更好。

总结

在测试代码中，可读性仍然很重要。如果测试的可读性很好，其结果是它们也会变得很容易写，因此大家会写更多的测试。并且，如果你把事实代码设计得容易测试，代码的整个设计会变得更好。

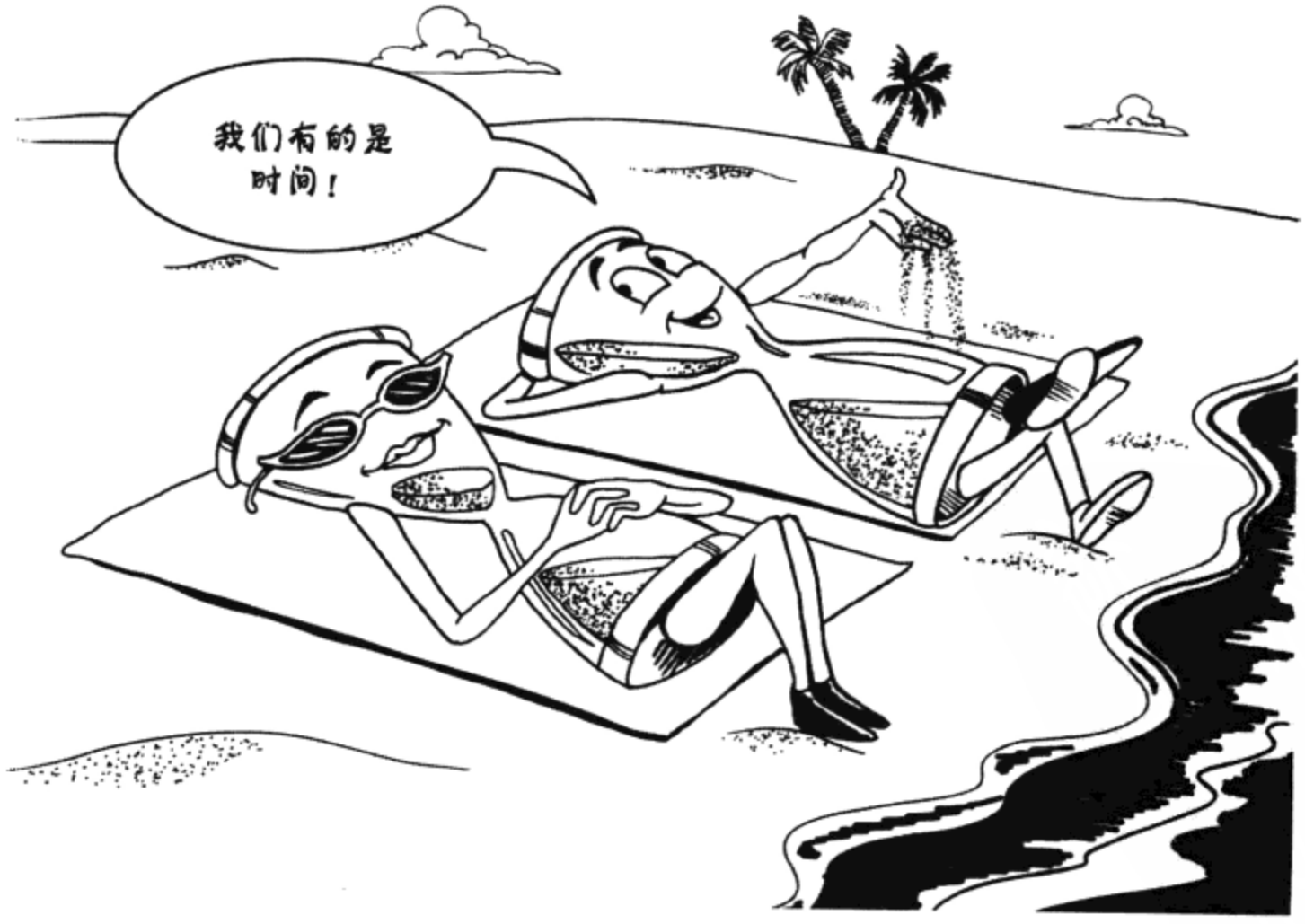
以下是如何改进测试的几个具体要点：

- 每个测试的最高一层应该越简明越好。最好每个测试的输入/输出可以用一行代码来描述。
- 如果测试失败了，它所发出的错误消息应该能让你容易跟踪并修正这个bug。
- 使用最简单的并且能够完整运用代码的测试输入。
- 给测试函数取一个有完整描述性的名字，以使每个测试所测到的东西很明确。不要用Test1()，而用像Test_<FunctionName>_<Situation>这样的名字。

最重要的是，要使它易于改动和增加新的测试。

第15章

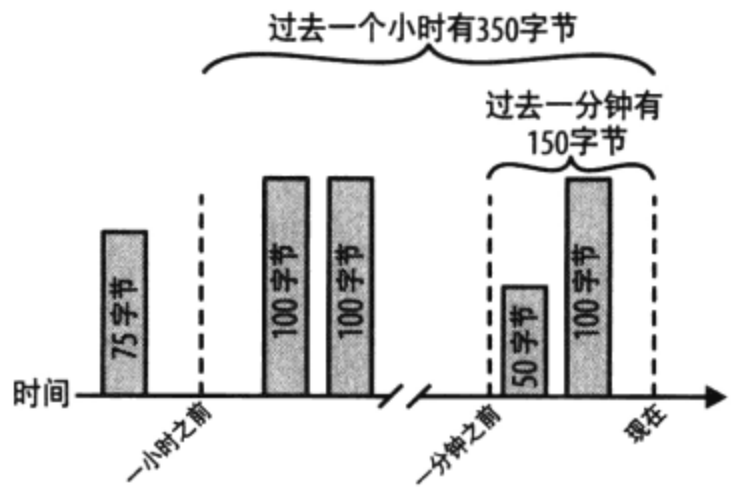
设计并改进“分钟/小时计数器”



让我们来看一件真实产品所用代码中的数据结构：一个“分钟/小时计数器”。我们会带你走过一个工程师可能会经历的自然的思考过程，首先试着解决问题，然后改进它的性能和增加功能。最重要的是，我们也会试着让代码保持易读，就用本书中所有的原则。在这个过程中我们也会转错几个地方，或者产生些其他的错误。看看你能不能理解并找出这些地方。

问题

我们需要跟踪在过去的一分钟和一个小时里Web服务器传输了多少字节。下面的图示说明了如何维护这些总和：



这个问题相当直接明了，但你将会看到，要有效地解决这个问题是个有趣的挑战。让我们从定义类接口开始。

定义类接口

下面是用C++写的第一个类接口版本：

```
class MinuteHourCounter {
  public:
    // Add a count
    void Count(int num_bytes);

    // Return the count over this minute
    int MinuteCount();

    // Return the count over this hour
    int HourCount();
};
```

在实现这个类之前，让我们看一遍这些名字和注释，看看是否有什么地方我们想改一改。

改进命名

MinuteHourCounter这个类名是很好的。它很专门、具体，并且容易读出来。

有了类名，方法名MinuteCount()和HourCount()也是合理的。你可能会给它们起GetMinuteCount()和GetHourCount()这样的名字，但这并没什么帮助。如第3章所述，对很多人来讲“get”暗示着“轻量级的访问器”。你将会看到，其他的实现并不会是轻量级的，所以最好不要“get”这个词。

然而方法名Count()是有问题的。我们问同事他们认为Count()会做什么，其中一些人认为它的意思是“返回所有时间里的总的计数”。这个名字有点违反直觉。问题是Count既是个名词又是个动词，既可以是“我想要得到你所见过的所有样本的计数”的意思也可以是“我想要你对样本进行计数”的意思。

下面几个名字可供代替Count()：

- Increment()
- Observe()
- Record()
- Add()

Increment()是会误导人的，因为它意味着一个只会增加的值。（在该情况中，小时计数会随时间波动。）

Observe()还可以，但是有点模糊。

Record()也有名词/动词的问题，所以不好。

Add()很有趣，因为它既可以是“以算术方法增加”的意思，也可以是“添加到一个数据列表”——在该情况中，两种情况兼而有之，所以Add()正合适。那么我们就要把这个方法重命名为void Add(int num_bytes)。

但是参数名num_bytes太有针对性了。是的，我们主要的用例的确是对字节计数，但是MinuteHourCounter没必要知道这一点。其他人可能用这个类来统计查询或者数据库事务的次数。我们可以用更通用的名字，如delta，但是delta这个词常常用在值有可能为负的场合，这可不是我们希望的。count这个名字应该可以——它简单、通用并且暗示“非负数”。同时，它使我们可以在更明确的背景下加入"count"这个词。

改进注释

下面是目前为止的类接口：

```
class MinuteHourCounter {
  public:

  // Add a count
  void Add(int count);

  // Return the count over this minute
  int MinuteCount();

  // Return the count over this hour
  int HourCount();
};
```

让我们看一遍每个方法的注释并且改进它们。看看第一个：

```
// Add a count
void Add(int count);
```

这条注释现在完全是多余的了——要么删除它，要么改进它。下面是一个改进的版本：

```
// Add a new data point (count >= 0).
// For the next minute, MinuteCount() will be larger by +count.
// For the next hour, HourCount() will be larger by +count.
void Add(int count);
```

现在让我们来看看MinuteCount()的注释：

```
// Return the count over this minute
int MinuteCount();
```

当我们问同事这段注释是什么意思时，得到了两种互相矛盾的解读：

1. 返回现在所在的时间（如12:12 p.m.）所在的分钟中的计数。
2. 返回过去60秒内的计数，和时钟边界无关。

第二种解释才是它实际的工作方式。所以让我们把这个混淆用更明确和具体的语言解释清楚。

```
// Return the accumulated count over the past 60 seconds.
int MinuteCount();
```

（同样地，我们也可以改进HourCount()的注释。）

下面是目前为止包含所有改动的类定义，还有一条类级别的注释：

```
// Track the cumulative counts over the past minute and over the past hour.
// Useful, for example, to track recent bandwidth usage.
class MinuteHourCounter {
```

```
    // Add a new data point (count >= 0).
    // For the next minute, MinuteCount() will be larger by +count.
    // For the next hour, HourCount() will be larger by +count.
    void Add(int count);

    // Return the accumulated count over the past 60 seconds.
    int MinuteCount();

    // Return the accumulated count over the past 3600 seconds.
    int HourCount();
};
```

（出于简洁的考虑，我们在后面会省略掉代码中的这些注释。）

得到外部视角的观点

你可能已经注意到，我们已经有两次通过同事来帮助我们解决问题了。询问外部视角的观点是测试你的代码是否“对用户友好”的好办法。要试着对他们的第一印象持开放的态度，因为其他人可能会有同样的结论。并且这些“其他人”里可能就包含6个月之后的你自己。

尝试1：一个幼稚的方案

让我们来进入下一步，解决这个问题。我们会从一个很直接的方案开始：就是保持一个有时间戳的“事件”列表：

```
class MinuteHourCounter {
    struct Event {
        Event(int count, time_t time) : count(count), time(time) {}
        int count;
        time_t time;
    };

    list<Event> events;

public:
    void Add(int count) {
        events.push_back(Event(count, time()));
    }

    ...
};
```

然后我们就可以根据需要计算最近事件的个数。

```
class MinuteHourCounter {
    ...
    int MinuteCount() {
        int count = 0;
        const time_t now_secs = time();
```

```
        for (list<Event>::reverse_iterator i = events.rbegin();
             i != events.rend() && i->time > now_secs - 60; ++i) {
            count += i->count;
        }
        return count;
    }

    int HourCount() {
        int count = 0;
        const time_t now_secs = time();
        for (list<Event>::reverse_iterator i = events.rbegin();
             i != events.rend() && i->time > now_secs - 3600; ++i) {
            count += i->count;
        }
        return count;
    }
};
```

这段代码易于理解吗？

尽管这个方案是“正确”的，可是其中有很多可读性的问题：

- **for循环太大，一口吃不下。**大多数读者在读这部分代码时会显著地慢下来（至少他们应该慢下来，如果他们要确定这里的确没有bug的话）。
- **MinuteCount()和HourCount()几乎一模一样。**如果他们可以共享重复代码就可能让这段代码少一些。这个细节非常重要，因为这些重复的代码相对更复杂。（让有难度的代码约束在一起更好些。）

一个更易读的版本

MinuteCount()和HourCount()中的代码只有一个常量不一样（60和3600）。明显的重构方法是引入一个辅助方法来处理这两种情况：

```
class MinuteHourCounter {
    list<Event> events;

    int CountSince(time_t cutoff) {
        int count = 0;
        for (list<Event>::reverse_iterator rit = events.rbegin();
             rit != events.rend(); ++rit) {
            if (rit->time <= cutoff) {
                break;
            }
            count += rit->count;
        }
        return count;
    }

public:
    void Add(int count) {
```

```
            events.push_back(Event(count, time()));
        }

        int MinuteCount() {
            return CountSince(time() - 60);
        }

        int HourCount() {
            return CountSince(time() - 3600);
        }
    };
```

在这段新代码中有几件事情值得一提。

首先，请注意CountSince()的参数是一个绝对的cutoff，而不是一个相对的secs_ago（60或3600）。两种方式都可行，但是这样做对CountSince()来讲更容易些。

其次，我们把迭代器从i改名为rit。i这个名字更常用在整型索引上。我们考虑过用it这个名字，这是迭代器的一个典型名字。但这一次我们用的是一个反向迭代器，并且这一点对于代码的正确性至关重要。通过在名字前面加一个前缀r，使它在如rit != events.rend()这样的语句中看上去对称。

最后，把条件rit->time <= cutoff从for循环中抽取出来，并把它作为一条单独的if语句。为什么这么做？因为保持循环的“传统”格式for（begin；end；advance）最容易读。读者会马上明白它是“遍历所有的元素”，并且不需要再做更多的思考。

性能问题

尽管我们改进了代码的外观，但这个设计还有两个严重的性能问题：

1. **它一直不停地在变大。**

   这个类保存着所有它见过的事件——它对内存的使用是没有限制的！最好MinuteHourCounter能自动删除超过一个小时以前的事件，因为不再需要它们了。

2. **MinuteCount()和HourCount()太慢了。**

   CountSince()这个方法的时间为$O(n)$，其中n是在其相关的时间窗口内数据点的个数。想象一下一个高性能服务器每秒调用Add()几百次。每次对HourCount()的调用都可能要对上百万个数据点计数！最好MinuteHourCounter能记住minute_count和hour_count变量，并随每次对Add()的调用而更新。

尝试2：传送带设计方案

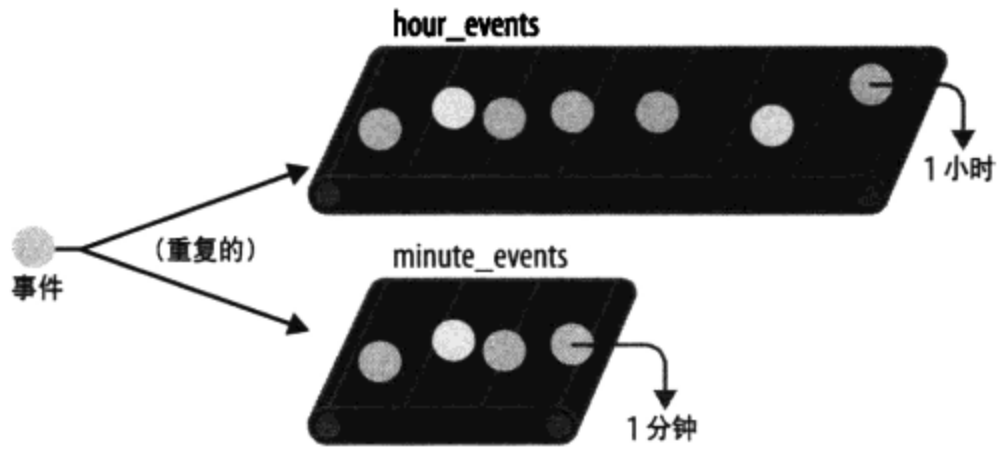
我们需要一个设计来解决前面提到的两个问题：

1. 删除不再需要的数据。

2. 更新事先算好的minute_count总和和hour_count变量总和。

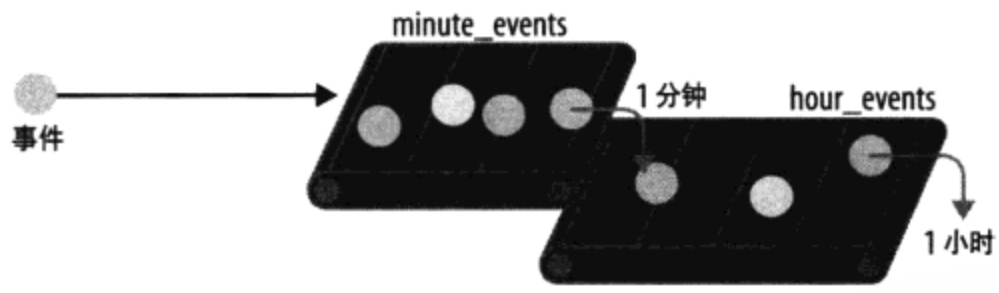
我们打算这样做：我们会像传送带一样地使用list。当新数据在一端到达，我们会在总数上增加。当数据太陈旧，它会从另一端“掉落”，并且我们会从总数中减去它。

有几种方法可以实现这个传送带设计。一种方法是维护两个独立的list，一个用于过去一分钟的事件，一个用于过去一小时。当有新事件到达时，在两个列表中都增加一个拷贝。



这种方法很简单，但它效率并不高，因为它为每个事件创建了两个拷贝。

另一种方法是维护两个list，事件先会进入第一个列表（“最后一分钟里的事件”），然后这个列表会把数据传送给第二个列表（“最后一小时［但不含最后一分钟］里的事件”）。



这种“两阶段”传送带设计看上去更有效，所以让我们按这个方法实现。

实现两阶段传送带设计

让我们从列出类中的成员开始：

```
class MinuteHourCounter {
    list<Event> minute_events;
    list<Event> hour_events;  // only contains elements NOT in minute_events

    int minute_count;
```

```
    int hour_count;  // counts ALL events over past hour, including past minute
};
```

这个传送带设计的要点在于要能随时间的推移“切换”事件，使事件从minute_events移到hour_events，并且minute_count和hour_count相应地更新。要做到这一点，我们会创建一个叫做ShiftOldEvents()的辅助方法。当我们有了这个方法以后，这个类的剩余部分很容易实现：

```
void Add(int count) {
    const time_t now_secs = time();
    ShiftOldEvents(now_secs);

    // Feed into the minute list (not into the hour list--that will happen later)
    minute_events.push_back(Event(count, now_secs));

    minute_count += count;
    hour_count += count;
}

int MinuteCount() {
    ShiftOldEvents(time());
    return minute_count;
}

int HourCount() {
    ShiftOldEvents(time());
    return hour_count;
}
```

明显，我们把所有的脏活儿都放到了ShiftOldEvents()里：

```
// Find and delete old events, and decrease hour_count and minute_count accordingly.
void ShiftOldEvents(time_t now_secs) {
    const int minute_ago = now_secs - 60;
    const int hour_ago = now_secs - 3600;

    // Move events more than one minute old from 'minute_events' into 'hour_events'
    // (Events older than one hour will be removed in the second loop.)
    while (!minute_events.empty() && minute_events.front().time <= minute_ago) {
        hour_events.push_back(minute_events.front());

        minute_count -= minute_events.front().count;
        minute_events.pop_front();
    }

    // Remove events more than one hour old from 'hour_events'
    while (!hour_events.empty() && hour_events.front().time <= hour_ago) {
        hour_count -= hour_events.front().count;
        hour_events.pop_front();
    }
}
```

这样就完成了吗？

我们已经解决了前面提到了对性能的两点担心，并且我们的方案是可行的。对很多应用来讲，这个解决方案就足够好了。但它还是有些缺点的。

首先，这个设计很不灵活。假设我们希望保留过去24小时的计数。这可能需要改动大量的代码。你可能已经注意到了，ShiftOldEvents()是一个很集中的函数，在分钟与小时数据间做了微妙的互动。

其次，这个类占用的内存很大。假设你有一个高流量的服务，每分钟调用Add()函数100次。因为我们保留了过去一小时内所有的数据，所以这段代码可能会需要用到大约5MB的内存。

一般来讲，Add()被调用得越多，使用的内存就越多。在一个产品开发环境中，库使用大量不可预测的内存不是一件好事。最好不论Add()被调用得多频繁，MinuteHourCounter能用固定数量的内存。

尝试3：时间桶设计方案

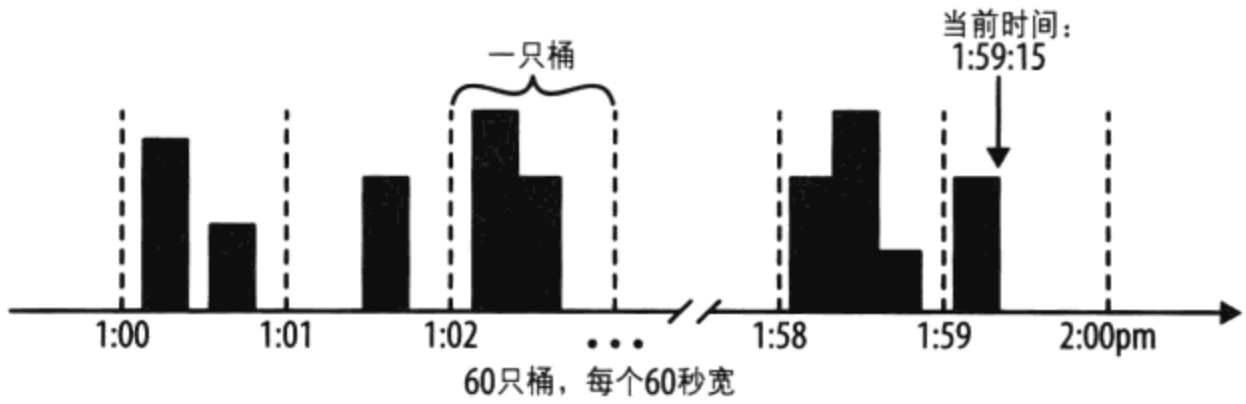
你应该已经注意到，前面的两个实现都有一个小bug。我们用time_t来保存时间戳，它保存的是一个以秒为单位的整数。因为这个近似，所以MinuteCount()实际上返回的是介于59～60秒钟的结果，根据调用它的时间而不同。

例如，如果一个事件发生在time = 0.99秒，这个time会近似成t=0秒。如果你在time = 60.1秒调用MinuteCount()，它会返回t=1,2,3...60的事件的总和。因此会遗漏第一个事件，尽管它从技术上来讲发生在不到一分钟以前。

平均来讲，MinuteCount()会返回相当于59.5秒的数据。并且HourCount()会返回相当于3 599.5秒的数据（一个微不足道的误差）。

可以通过使用亚秒粒度来修正这个误差。但是有趣的是，大多数使用MinuteHourCounter的应用程序不需要这种程度的精度。我们会利用这一点来设计一个新的MinuteHourCounter，它要快得多并且占用的空间更少。它是在精度与物有所值的性能之间的一个平衡。

这里的关键思想是把一个小时间窗之内的事件装到桶里，然后用一个总和累加这些事件。例如，过去1分种里的事件可以插入60个离散的桶里，每个有1秒钟宽。过去1小时里的事件也可以插入60个离散的桶里，每个1分钟宽。



如图一样使用这些桶，方法MinuteCount()和HourCount()的精度会是1/60，这是合理的。[注1]

如果要更精确，可以使用更多的桶，以使用更多内存为交换。但重要的是这种设计使用固定的、可预知的内存。

实现时间桶设

如果只用一个类来实现这个设计会产生很多错综复杂的代码，很难理解。相反，我们会按照第11章中的建议，创建一些不同的类来处理问题的不同部分。

一开始，首先创建一个不同的类来保存一个时间段里的计数（如最后一小时）。把它命名为TrailingBucketCounter。它基本上是MinuteHourCounter的泛化版本，用来处理一个时间段。以下是接口：

```
// A class that keeps counts for the past N buckets of time.
class TrailingBucketCounter {
    public:
    // Example: TrailingBucketCounter(30, 60) tracks the last 30 minute-buckets of time.
    TrailingBucketCounter(int num_buckets, int secs_per_bucket);

    void Add(int count, time_t now);

    // Return the total count over the last num_buckets worth of time
    int TrailingCount(time_t now);
};
```

你可能会想为什么Add()和TrailingCount()需要当前时间（time_t now）来做参数——如果用这些方法自己来计算当前的time()不是更方便吗？

注1：与前面的方案相似，最后一只桶平均只有实际的一半。用这种设计，我们可以用保持61只桶而不是60只桶并且忽略当前“正在进行中”的桶来进行补救。但是会让数据有点部分“堆积”。一个更好的修正是把当前正在进行的桶与最老的桶里的一个互补部分结合，得到一个既无偏差又最新的计数。这种实现留给读者作为练习。

尽管这看上去有点怪，但传入当前时间有两个好处。首先，它让TrailingBucketCounter成为一个“时钟无关”的类，一般来讲这更容易测试而且会避免bug。其次，它把所有对time()的调用保持在MinuteHourCounter中。对于时间敏感的系统，如果能把所有获得时间的调用放在一个地方会有帮助。

假设TrailingBucketCounter已经实现了，那么MinuteHourCounter就很容易实现了：

```
class MinuteHourCounter {
    TrailingBucketCounter minute_counts;
    TrailingBucketCounter hour_counts;

public:
    MinuteHourCounter() :
        minute_counts(/* num_buckets = */ 60, /* secs_per_bucket = */ 1),
        hour_counts(  /* num_buckets = */ 60, /* secs_per_bucket = */ 60) {
    }

    void Add(int count) {
        time_t now = time();
        minute_counts.Add(count, now);
        hour_counts.Add(count, now);
    }

    int MinuteCount() {
        time_t now = time();
        return minute_counts.TrailingCount(now);
    }

    int HourCount() {
        time_t now = time();
        return hour_counts.TrailingCount(now);
    }
};
```

这段代码更容易读，也更灵活——如果我们想增加桶的数量（通过增加内存使用来改善精度），那将会很容易。

实现TrailingBucketCounter

现在所有剩下的工作就是实现TrailingBucketCounter类了。再一次，我们会创建一个辅助类来进一步拆分这个问题。

我们会创建一个叫做ConveyorQueue的数据结构，它的工作是处理其下的计数与总和。TrailingBucketCounter类可以关注根据过去了多少时间来移动ConveyorQueue。

下面是ConveyorQueue接口：

```
// A queue with a maximum number of slots, where old data "falls off" the end.
class ConveyorQueue {
    ConveyorQueue(int max_items);
```

```
    // Increment the value at the back of the queue.
    void AddToBack(int count);

    // Each value in the queue is shifted forward by 'num_shifted'.
    // New items are initialized to 0.
    // Oldest items will be removed so there are <= max_items.
    void Shift(int num_shifted);

    // Return the total value of all items currently in the queue.
    int TotalSum();
};
```

假设这个类已经实现了，请看TrailingBucketCounter多么容易实现：

```
class TrailingBucketCounter {
    ConveyorQueue buckets;
    const int secs_per_bucket;
    time_t last_update_time;  // the last time Update() was called

    // Calculate how many buckets of time have passed and Shift() accordingly.
    void Update(time_t now) {
        int current_bucket = now / secs_per_bucket;
        int last_update_bucket = last_update_time / secs_per_bucket;

        buckets.Shift(current_bucket - last_update_bucket);
        last_update_time = now;
    }

public:
    TrailingBucketCounter(int num_buckets, int secs_per_bucket) :
        buckets(num_buckets),
        secs_per_bucket(secs_per_bucket) {
    }

    void Add(int count, time_t now) {
        Update(now);
        buckets.AddToBack(count);
    }

    int TrailingCount(time_t now) {
        Update(now);
        return buckets.TotalSum();
    }
};
```

现在它拆成两个类（TrailingBucketCounter和ConveyorQueue），这是第11章所讨论的又一个例子。我们也可以不用ConveyorQueue，直接把所有的东西存放在TrailingBucketCounter里。但是这样代码更容易理解。

实现ConveyorQueue

现在剩下的只是实现ConveyorQueue类：

```
// A queue with a maximum number of slots, where old data gets shifted off the end.
```

```
class ConveyorQueue {
    queue<int> q;
    int max_items;
    int total_sum;  // sum of all items in q

public:
    ConveyorQueue(int max_items) : max_items(max_items), total_sum(0) {
    }

    int TotalSum() {
        return total_sum;
    }

    void Shift(int num_shifted) {
        // In case too many items shifted, just clear the queue.
        if (num_shifted >= max_items) {
            q = queue<int>();  // clear the queue
            total_sum = 0;
            return;
        }

        // Push all the needed zeros.
        while (num_shifted > 0) {
            q.push(0);
            num_shifted--;
        }

        // Let all the excess items fall off.
        while (q.size() > max_items) {
            total_sum -= q.front();
            q.pop();
        }
    }

    void AddToBack(int count) {
        if (q.empty()) Shift(1);  // Make sure q has at least 1 item.
        q.back() += count;
        total_sum += count;
    }
};
```

现在我们完成了！我们有一个又快又能有效地使用内存的MinuteHourCounter，外加一个更灵活的TrailingBucketCounter，它很容易重用。例如，很容易就能创建一个功能更齐全的RecentCounter来计算更多时间间隔范围，比如过去一天或者过去十分钟。

比较三种方案

让我们来比较一下本章中见到的这些方案。下表给出代码的大小和性能状况（假设在一个每秒100次Add()调用的高流量用例中）：

方案	代码行数	每次HourCount()调用的代价	内存使用情况	HourCount()的误差
幼稚方案	33	O(每小时事件数)（约360万）	无约束	1/3600
传送代设计	55	O(1)	O(每小时事件数)（约5MB）	1/3600
时间桶设计（60只桶）	98	O(1)	O(桶的个数)（约500字节）	1/60

请注意最后那个有三个类的方案的代码数量比任何其他的尝试都多。然而，性能好得多，并且设计更灵活。而且，每个类自己都更容易读。这是一个正面的改进：有100行易读的代码比有50行不易读的要好。

有时，把一个问题拆成多个类可能引入类之间的复杂度（在有单个类的方案中是不会有的）。然而，在本例中，有一个简单的“线性”链条连接着每个类，并且只有一个类暴露给了最终用户。总体来讲，拆分这个问题所得到的好处更大。

总结

让我们回顾得到最后的MinuteHourCounter设计所走过的路。这是个典型的代码片段演进过程。

首先，我们从编写一个幼稚的方案开始。这帮助我们意识到两个设计上的挑战：速度和内存使用情况。

接下来，我们尝试用“传送带”设计方案。这种设计改进了速度和内存使用情况，但对于高性能应用来讲还是不够好。并且，这个设计不是很灵活：让这段代码能处理其他的时间间隔需要很多工作。

我们最终的设计解决了前面的问题，通过把问题拆分成子问题。下面是创建的三个类，自下向上，以及每个类所解决的子问题：

ConveyorQueue
: 一个最大长度的队列，可以“移位”并且维护其总和。

TrailingBucketCounter
: 根据过去了多少时间来移动ConveyorQueue，并且按所给的精度来维护单一（最后）的时间间隔中的计数。

MinuteHourCounter
: 简单地包含两个TrailingBucketCounters，一个用来统计分钟，一个用来统计小时。

附录

深入阅读

我们通过分析来自产品代码中的上百个代码例子来找出在实践中什么是有用的，从而写出这本书。但是我们也读了很多书和文章，这对于我们的写作也很有帮助。

如果你想学到更多，你可能会喜欢下面这些资源。下面的列表怎么说都不算完整，但是它们是个好的开端。

关于写高质量代码的书

《Code Complete: A Practical Handbook of Software Construction, 2nd edition》, by Steve McConnell (Microsoft Press, 2004)

一本严谨的大部头，是关于软件建构的所有方面的，包括代码质量以及其他。

《Refactoring: Improving the Design of Existing Code》, by Martin Fowler et al. (Addison-Wesley Professional, 1999)

一本关于增量代码改进哲学的好书，包含很多不同重构方法的具体分类，以及要在尽管不破坏东西的情况下做出这些改动所需的步骤。

《The Practice of Programming》, by Brian Kernighan and Rob Pike (Addison-Wesley Professional, 1999)

讨论了编程的多个方面，包含调试、测试、可移植性和性能，有很多代码示例。

《The Pragmatic Programmer: From Journeyman to Master》, by Andrew Hunt and David Thomas (Addison-Wesley Professional, 1999)

一系列好的编程和工程原则，按短小的讨论来组织。

《Clean Code: A Handbook of Agile Software Craftsmanship》, by Robert C. Martin (Prentice Hall, 2008)

和本书类似（但是专门为Java），还拓展了其他如错误处理和并发等话题。

关于各种编程话题的书

《JavaScript: The Good Parts》, by Douglas Crockford (O'Reilly, 2008)

我们认为这本书的精神与我们的书相似，尽管该书不是直接关于可读性的。它是关于如何使用JavaScript语言中不容易出错而且更容易理解的一个清晰子集的。

《Effective Java, 2nd edition》, by Joshua Bloch (Prentice Hall, 2008)

一本杰出的书，是关于让你的Java程序更易读和更少bug的。尽管它是关于Java的，但其中很多原则对所有的语言都适用。强烈推荐。

《Design Patterns: Elements of Reusable Object-Oriented Software》, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional, 1994)

这本书是软件工程师用来讨论面向对向编程所用的“模式”这种通用语言的原始出处。作为通用的、有用的模式的一览，它帮助程序员在第一次自己解决棘手问题时避免经常出现的陷阱。

《Programming Pearls, 2nd edition》, by Jon Bentley (Addison-Wesley Professional, 1999)

关于真实软件问题的一系列文章。每一章都有解决真实世界中问题的真知灼见。

《High Performance Web Sites》, by Steve Souders (O'Reilly, 2007)

尽管这不是一本关于编程的书，但这本书也值得注意，因为它描述了几种不需要写很多代码就可优化网站的方法（与本书第13章的目的一致）。

《Joel on Software: And on Diverse and ...》, by Joel Spolsky

来自于 http://www.joelonsoftware.com/的一些优秀文章。Spolsky的作品涉及软件工程的很多方面，并且对很多相关话题都深有见解。一定要读一读“Things You Should Never Do, Part I”和“The Joel Test: 12 Steps to Better Code”。

历史上重要的书目

《Writing Solid Code》, by Steve Maguire (Microsoft Press, 1993)

很遗憾这本书有点过时了，但它绝对在如何让代码中的bug更少方面给出了出色的建议，从而影响了我们。如果你读这本书，你会注意到很多和我们的建议重复的地方。

《Smalltalk Best Practice Patterns》, by Kent Beck (Prentice Hall, 1996)

尽管例子是用Smalltalk写的，但这本书有很多好的编程原则。

《The Elements of Programming Style》, by Brian Kernighan and P.J. Plauger (Computing McGraw- Hill, 1978)

最早的关于“写代码的最清晰方法”的书之一。大多数例子是用Fortran和PL1写的。

《Literate Programming》, by Donald E. Knuth (Center for the Study of Language and Information, 1992)

我们发自肺腑地赞同Knuth的说法：“与其把我们主要的任务想象成指示计算机做什么，不如让我们关注解释给人类我们希望让计算机做什么”（p.99）。但要小心：这本书中的大部分内容是关于Knuth的WEB文档编程环境的。WEB实际上是一种语言，使用可以像写文学作品一样来写程序，以代码为辅助内容。

我们自己用过衍生自WEB的系统，我们认为当代码变化频繁时（这很常见），相对于用我们所建议的实践方法，保持这种所谓的“文学编程”来更新代码更难。

作者简介

尽管在马戏团长大，**Dustin Boswell**很早就发现他更擅长计算机而不是杂技。Dustin在加州理工学院得到了他的本科学位，在那里他爱上了计算机科学，于是后来去圣地亚哥加利福尼亚大学读研究生。他在Google工作了5年，从事过不同的项目，包括网页爬虫的基础结构。他建立了数个网站并且喜欢从事“大数据”和机器学习方面的工作。Dustin目前在一家Internet创业公司工作，他空闲时间会去圣摩尼卡山中徒步旅行，并且他刚刚做了父亲。

Trevor Foucher在微软和Google从事了超过10年的大型软件开发。他现在是Google的一名搜索基础结构工程师。在他的业余时间里，他参与游戏聚会，阅读科幻小说，并且是他妻子的时装创业公司的COO。Trevor毕业于加州大学伯克利分校，获得电气工程和计算机科学的本科学位。