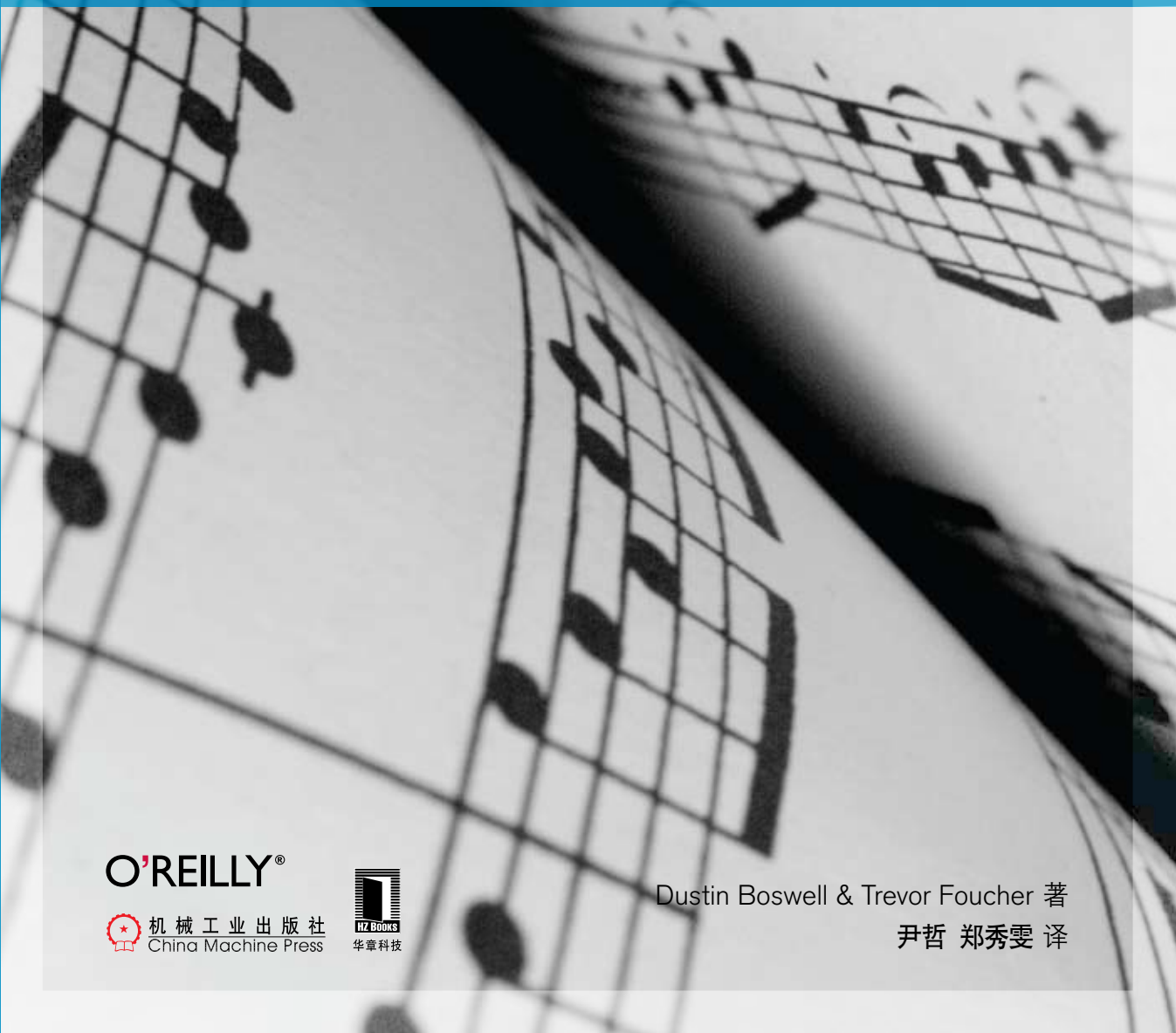
/Theory/In/Practice

编写可读代码的艺术

The Art of Readable Code



O'REILLY®

机械工业出版社
China Machine Press

HZ BOOKS
华章科技

Dustin Boswell & Trevor Foucher 著

尹哲 郑秀雯 译

编写可读代码的艺术

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：01-2011-1275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/

开　　本/　178毫米×233毫米　16开本　12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

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 1

第1章 代码应当易于理解 5

是什么让代码变得“更好” 6

可读性基本定理 7

总是越小越好吗 7

理解代码所需的时间是否与其他目标有冲突 8

最难的部分 8

第一部分 表面层次的改进 9

第2章 把信息装到名字里 11

选择专业的词 12

避免像tmp和retval这样泛泛的名字 14

用具体的名字代替抽象的名字 17

为名字附带更多信息 19

名字应该有多长 22

利用名字的格式来传递含义 24

总结 25

第3章 不会误解的名字 27
例子：Filter() 28
例子：Clip(text, length) 28
推荐用first和last来表示包含的范围 29
推荐用begin和end来表示包含/排除范围 30
给布尔值命名 30
与使用者的期望相匹配 31
例子：如何权衡多个备选名字 33
总结 34

第4章 审美 36
为什么审美这么重要 37
重新安排换行来保持一致和紧凑 38
用方法来整理不规则的东西 40
在需要时使用列对齐 41
选一个有意义的顺序，始终一致地使用它 42
把声明按块组织起来 43
把代码分成“段落” 44
个人风格与一致性 45
总结 46

第5章 该写什么样的注释 47
什么不需要注释 49
记录你的思想 52
站在读者的角度 54
最后的思考——克服“作者心理阻滞” 58
总结 59

第6章 写出言简意赅的注释 60
让注释保持紧凑 61
避免使用不明确的代词 61
润色粗糙的句子 62

精确地描述函数的行为 62
用输入/输出例子来说明特别的情况 63
声明代码的意图 64
“具名函数参数”的注释 64
采用信息含量高的词 65
总结 66

第二部分 简化循环和逻辑 67

第7章 把控制流变得易读 69
条件语句中参数的顺序 70
if/else语句块的顺序 71
?:条件表达式（又名“三目运算符”） 73
避免do/while循环 74
从函数中提前返回 76
臭名昭著的goto 76
最小化嵌套 77
你能理解执行的流程吗 80
总结 81

第8章 拆分超长的表达式 82
用做解释的变量 83
总结变量 83
使用德摩根定理 84
滥用短路逻辑 84
例子：与复杂的逻辑战斗 85
拆分巨大的语句 87
另一个简化表达式的创意方法 88
总结 89

第9章 变量与可读性 91
减少变量 92
缩小变量的作用域 94

只写一次的变量更好 100
最后的例子 101
总结 103

第三部分 重新组织代码 105

第10章 抽取不相关的子问题 107
介绍性的例子：findClosestLocation() 108
纯工具代码 109
其他多用途代码 110
创建大量通用代码 112
项目专有的功能 112
简化已有接口 113
按需重塑接口 114
过犹不及 115
总结 116

第11章 一次只做一件事 117
任务可以很小 119
从对象中抽取值 120
更大型的例子 124
总结 126

第12章 把想法变成代码 127
清楚地描述逻辑 128
了解函数库是有帮助的 129
把这个方法应用于更大的问题 130
总结 133

第13章 少写代码 135
别费神实现那个功能——你不会需要它 136
质疑和拆分你的需求 136
保持小代码库 138

熟悉你周边的库 139
例子：使用Unix工具而非编写代码 140
总结 141

第四部分 精选话题 143

第14章 测试与可读性 145
使测试易于阅读和维护 146
这段测试什么地方不对 146
使这个测试更可读 147
让错误消息具有可读性 150
选择好的测试输入 152
为测试函数命名 154
那个测试有什么地方不对 155
对测试较好的开发方式 156
走得太远 158
总结 158

第15章 设计并改进“分钟/小时计数器” 160
问题 161
定义类接口 161
尝试1：一个幼稚的方案 164
尝试2：传送带设计方案 166
尝试3：时间桶设计方案 169
比较三种方案 173
总结 174

附录 深入阅读 175

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。

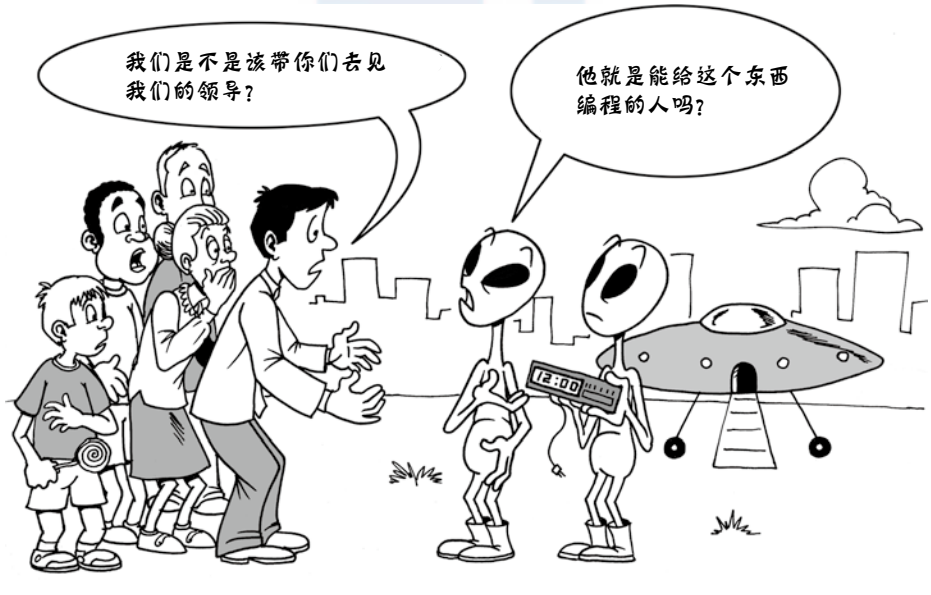
我们还从下面这些人那里得到了大量的想法和反馈：John Blackburn、Tim Dasilva、Dennis Geels、Steve Gerding、Chris Harris、Josh Hyman、Joel Ingram、Erik Mavrinac、Greg Miller、Anatole Paine和Nick White。

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。

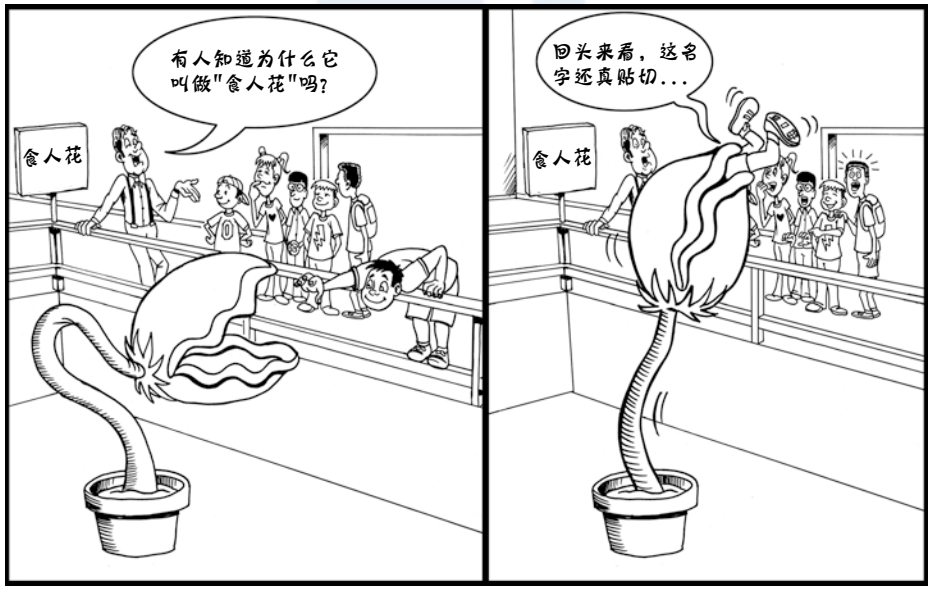
这些话题很重要，**因为会影响到你代码库中的每行代码**。尽管每个改变可能看上去都很小，聚集在一起造成代码库巨大的改进。如果你的代码有很棒的名字、写得很好的注释，并且整洁地使用了空白符，你的代码会变得易读得多。

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。

HZ BOOKS
华章图书

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？

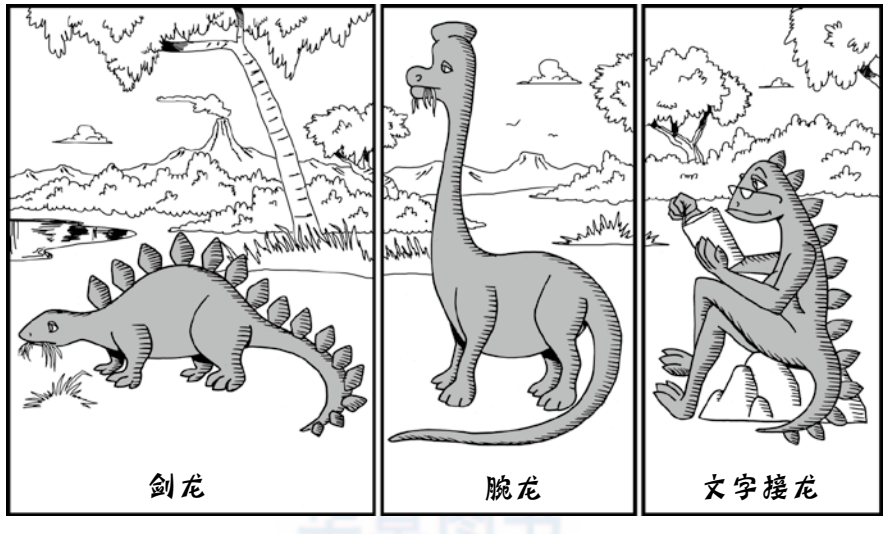
问题是Size()没有承载很多信息。更专业的词可以是Height()、NumNodes()或者MemoryBytes()。

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。

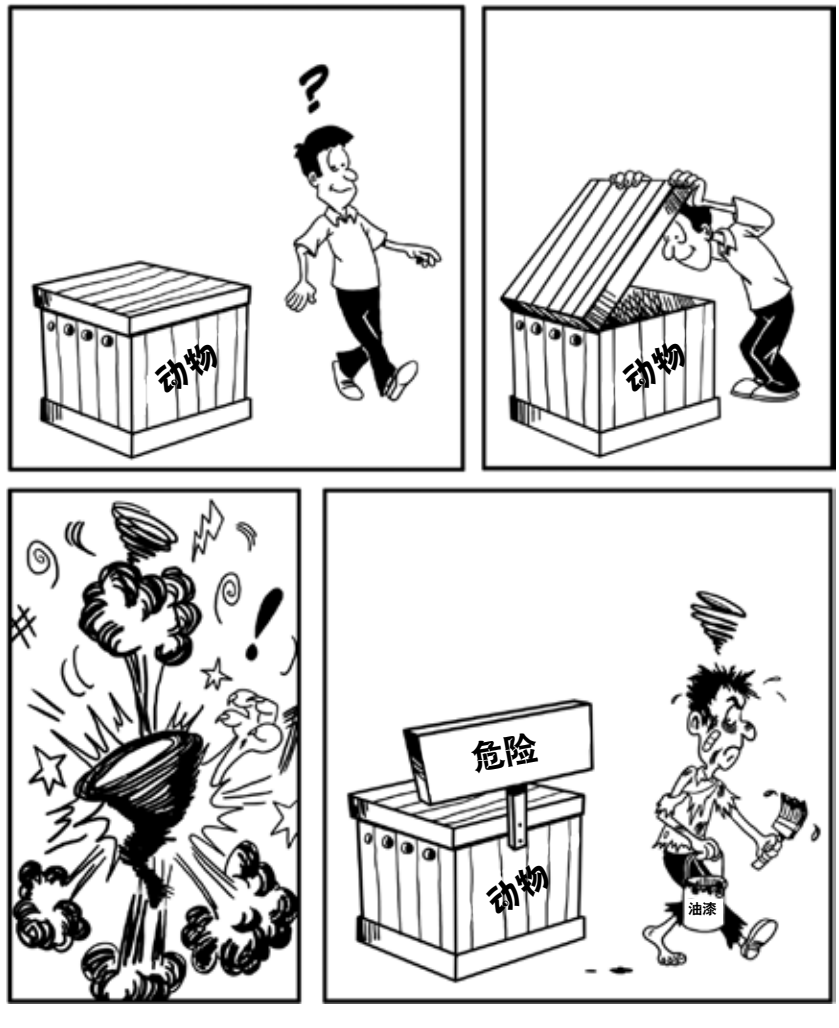
但是如果--run_locally需要做比额外日志更多的事情怎么办？例如，假设它需要建立和

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？

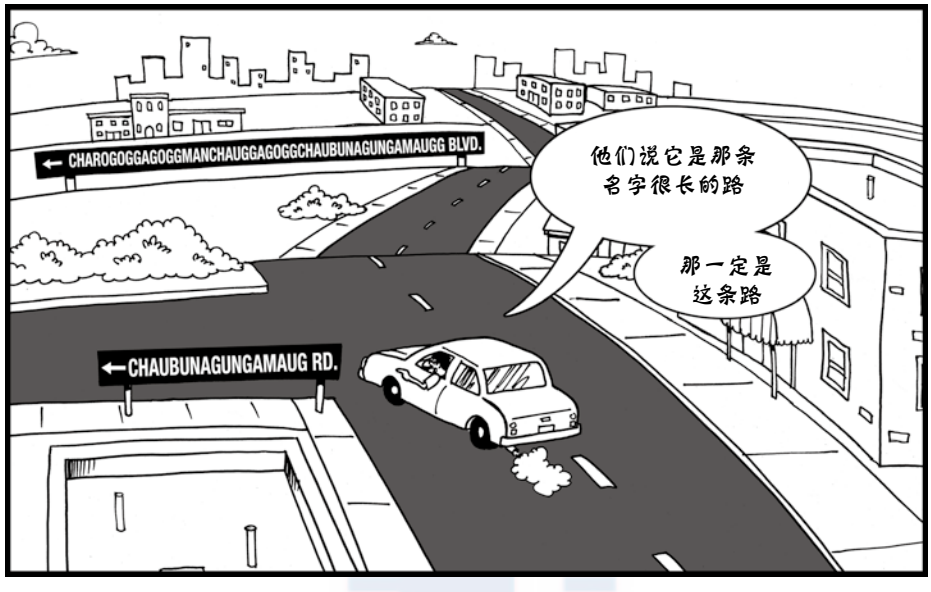
匈牙利表示法是一个在微软广泛应用的命名系统，它把每个变量的“类型”信息都编写进名字的前缀里。下面有几个例子：

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。

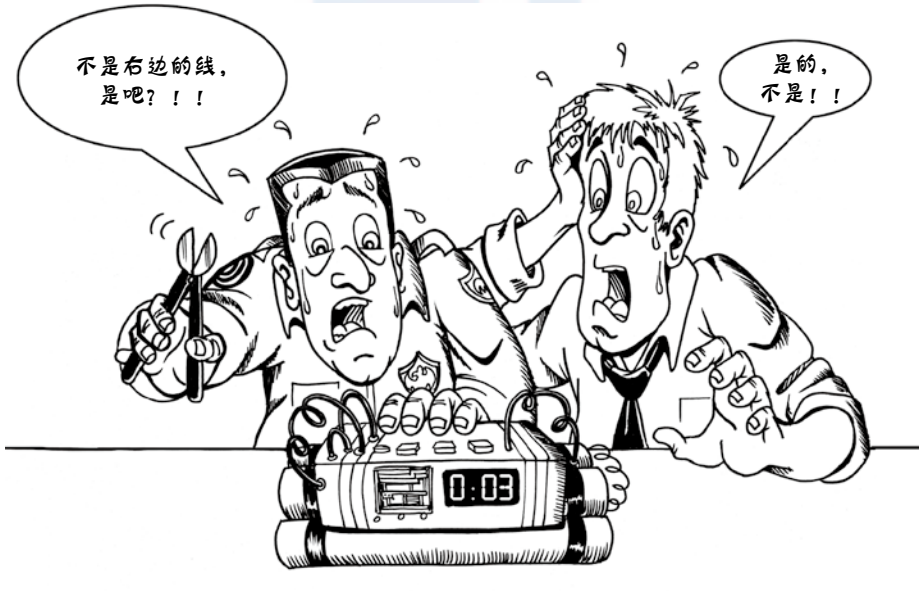
下面是讨论过的几个小提示：

- **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

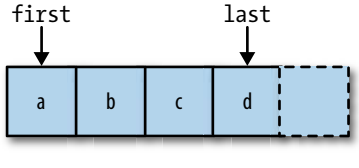
命名极限最清楚的方式是在要限制的东西前加上max_或者min_。

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)?
```

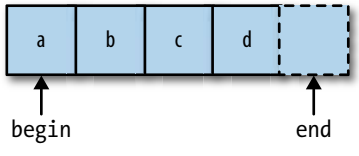
尽管start是个合理的参数名，但stop可以有多种解读。对于这样包含的范围（这种范围包含开头和结尾），一个好的选择是first/last。例如：

```
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。

软件开发/编程

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

此简体中文版仅限于在中华人民共和国境内（但不允许在中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

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

定价：59.00元