计 算 机 科 学 丛 书

程序设计实践

The Practice of Programming

(美) Brian W. Kernighan
Rob Pike 著
裘宗燕 译

机械工业出版社
China Machine Press
Addison-Wesley

计算机科学丛书

程序设计实践

(美) Brian W. Kernighan
Rob Pike 著

裘宗燕 译

机械工业出版社
China Machine Press

本书是Brian W. Kernighan和Rob Pike合著的最新力作。本书从排错、测试、性能、可移植性、设计、界面、风格和记法等方面，讨论了程序设计中实际的、又是非常深刻和具有广泛意义的思想、技术和方法，它的翻译出版将填补国内目前这方面书籍的空白。本书值得每个梦想并努力使自己成为优秀程序员的人参考，值得每个计算机专业的学生和计算机工作者阅读，也可作为程序设计高级课程的教材或参考书。

本书版权登记号：图字：01-1999-3213

图书在版编目(CIP)数据

程序设计实践 / (美)柯奈汉(Kernighan, B. W), (美)派克(Pike, R.)著，裘宗燕译 -北京机械工业出版社，2000 8
(计算机科学丛书)
书名原文 The Practice of Programming
ISBN 7-111-07573-0

Ⅰ.程· Ⅱ ①柯·· ②派… ③裘… Ⅲ 程序设计 Ⅳ TP311 1

中国版本图书馆CIP数据核字 (2000) 第24741号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）
责任编辑 陈贤舜
北京牛山世兴印刷厂印刷 · 新华书店北京发行所发行
2000年8月第1版 · 2001年1月第2次印刷
787mm × 1092 mm 1/16 · 14.5印张
印数：7 001-9 000册
定价：20.00元

译 者 序

我最早看到《程序设计实践》一书的消息是在1999年3/4月IEEE的《Software》杂志上，在那里有对该书的介绍，还特别摘编了书中第7章“性能”的部分内容。B. Kernighan是全世界计算机界专业人士都非常敬重的程序专家和专业作家，他与C语言创造者D. Ritchie合著的《C程序设计语言》、与Rob Pike合著的《UNIX编程环境》(以上两本书已由机械工业出版社出版)、与P. Plauger合著的《程序设计风格的要素》和《软件工具》等都是程序和程序设计方面风靡世界、口碑极佳的经典之作。Kernighan和Pike的新书自然也非常引人注目。读了该杂志上刊出的摘编，我感到这本书里确实讨论了程序设计中许多非常重要的问题，而它们又是非常实际，有些也是自己在工作中经常遇到的。文中的分析和讨论既通俗浅显，又寓意深远，读后觉得很有收获，我也由此萌生了将这本书介绍给国内同行的念头。后经辗转，了解到机械工业出版社已经购到本书在中文版版权，便自告奋勇承担了翻译工作。经过几个月的努力，我很高兴看到这本书终于即将可以付印了。

程序设计是计算机专业领域中最核心的工作。在计算机领域中任何好的创意和设计，最终都需要通过高水平的程序设计实现，才能够真正成为有社会价值、市场价值的制成品和产品。有一件事使我非常感慨。在计算机领域走在世界最前列的美国，确实有那么一大批人，他们几十年如一日地奋斗在程序设计领域里。比如Tompson和Ritchie得了图灵奖之后仍在做程序设计工作，Kernighan和Pike也是这些人中的杰出代表。我觉得这些人是真正托起美国计算机业的脊梁。在中国何时能够有这样的一大批人，那么中国的计算机事业也就能在世界上占据一席之地了。中国的计算机事业要有真正的发展，必须造就一大批高水平的程序设计专家，这既是社会对于一批最优秀青年的召唤，也是历史对中国计算机教育界的挑战。

我认为，本书在这个方面是非常有价值的。它讨论的是程序设计中实际的、又是非常深刻、具有广泛意义的思想、技术和方法，它的翻译出版填补了国内目前这方面书籍的空白。这本书值得每个梦想并在努力使自己发展成为优秀程序员的人参考，值得每个计算机专业的学生和计算机工作者阅读，也可以作为程序设计高级课程的参考书或教材。我相信，任何在与计算机相关的领域中工作的人们，任何真正的计算机爱好者，都会在本书的阅读中有许多收获。这也正是我愿意来做这件事的原因。

近年来，程序设计环境确实取得了长足的进步。有人可能会提出疑问，既然许多程序似乎都能在一系列指指点点之中完成，在这种情况下程序设计难道还有多少花招好玩吗？实际情况并不是那么简单。任何一个有点价值的程序都不仅是一个图形界面外壳。虽然在最终的可执行代码里，界面外壳可能占据了相当大的比例，但一个程序的特点恰恰就在那些人们自己动手写出的代码中。而良好的设计、卓越的编码实现永远都是不可替代的，是不可能自动生成的，正是它们形成了一个软件的真正价值。实际上，所有程序设计环境中有价值的东西不过是在一些局部领域中优秀代码的浓缩和沉淀。

应该看到，功能强大的程序开发环境未必能造就出更优秀的程序员。在我自己的教学实践中，遇到了越来越多这样的学生，他们是玩编程环境的高手，但距离程序设计高手却差得

太远。他们对算法和数据结构选择、程序的结构设计和良好编码的各种要素不太得要领，最熟悉的就是排错系统(debuger)和单步执行，最拿手的是在程序上打补丁。一个十行程序就能解决的问题，让他们来做，动辄就是几十上百行甚至更多，而到最后自己也不知道做得到底对不对，只知道试过一些例子，单步执行无数遍，打了数不清的补丁。可悲的是，以这类活动方式作为编程要义的不仅是这些学生，这种情况也不仅出现在校园里，不仅出现在我们周围，不仅出现在中国。这可能也是我们经常使用的系统规模庞大，而又不时崩溃的一个重要原因。无论是这些学生，或者是作为一般计算机工作者、作为教师的我们，都应该看看本书作者是如何对待编程和测试问题的，应该从中领略什么样的东西能称得上是真正的代码。

正如作者在书中反复强调的，程序设计并不就是编码。在程序设计工作中，既有工程也有科学。需要有对所面临的问题的科学分析；考虑各种设计可能性，在各种约束条件中寻找合理的平衡点；选择适当的技术、算法和数据结构，选择合适的语言和语言中合适的描述结构；编码和可能的代码调整；设法排除程序中的错误；通过系统化的测试，努力保证程序能够满足我们的需要，具有相当的可靠性和抵御外部意外事件的能力。这些都是本书中详细讨论了的题目。书中的内容还很多。在阅读本书的过程中，在每一章里，你都会发现许多新鲜生动、而又是非常深刻的东西，会发现一个又一个你确实应该知道，但至今还不知道的东西。这也是我自己在阅读和翻译这本书过程中的切身体会。

我(相信读者也会)非常感谢机械工业出版社华章公司的编辑人员，是他们的见识，使这本好书有可能较早地与中国读者见面。由于本书覆盖面广，翻译难度确实比较大，我虽然在翻译中下了不少功夫，但在译文中仍难免出现一些疏漏，这些当然都只能由我本人负责。我也真诚地希望认真的、有见识的同行和朋友们不吝赐教。纠正书中的错误也像是debug，可惜的是我们无法对书籍内容设计出一套系统的测试集，也不可能有功能强大的debuger系统。要排除这里的错误，只能靠人的眼睛、专业知识、逻辑思维和语言文字能力。而在所有这些方面，作为译者的我都还需要继续努力。

裘宗燕

2000年4月于北大

裘宗燕，北京大学数学学院信息科学系教授。关心的主要学术领域包括计算机软件理论、方法学、程序语言和符号计算。已出版多部著作及译著，主要包括：《程序设计语言基础》(译著，1990)，《Mathematica数学软件系统的应用与程序设计》(1994)，《计算概论(上)》(合作，1997)，《数据结构——C++与面向对象的途径》(合作，1998)，《从问题到程序——程序设计与C语言引论》(1999)等。

前　言

你是否曾：

浪费了许多时间去对一个错误算法做编码？

使用了一个过于复杂的数据结构？

测试一个程序而忽略了其中最简单的问题？

化了一整天功夫查找一个错误，实际上却应该在5分钟里就找到它？

需要让一个程序运行速度快3倍而且使用更少的存储？

费劲地把一个工作站上的程序移到PC上，或者相反？

试图对其他人写的某个程序做最简单的修改？

重写一个程序，原因是你根本无法理解它？

这些事都很有趣吗？

编程序的人每时每刻都会遇到这些事情。但在处理时发生的情况却很不令人满意，通常都比实际上应该的更糟。产生这种状况的原因是，常规的计算机科学和程序设计课程都不大关注那些程序设计实践方面的论题，如测试、排错、可移植性、性能、设计选择以及程序设计风格等等。许多程序员随着自己经验的增长慢慢地、毫无计划地学到了一些东西，也有些人始终都没有学好。

在这个庞大的世界里，人们面临的是错综复杂、相互联系和处于不断变化中的工具、语言和系统，以及对任何东西都提出更多要求的无情压力。在这种情况下，人们很容易忘掉那些最基本的原则——简单性、清晰性和普遍性——而正是这些东西，实际上是构成好软件的基石。人们也可能忽略工具和表示法的价值，实际上，利用它们可由计算机自动构造出某些软件，从而让计算机为其本身的程序设计做一些事情。

在这本书里，我们采用的方式是讨论一些基本的、互相关联的原则，它们可以用到计算的各个层次中。这些原则包括：简单性，它能使程序短而容易管理；清晰性，它保证程序容易理解，无论是对人还是对机器；普遍性，它意味着程序在很广泛的情形下都能工作得很好，也容易做修改以适应新出现的情况；自动化，借助计算机帮助我们做一些工作，使人能够摆脱许多平凡琐碎的事项。通过考察各种不同语言中的一些程序设计例子，从算法和数据结构，到设计、调试、排错和性能改进，我们能展现出许多具有普遍意义的工程概念，它们是独立于具体语言、操作系统和编程规范的。

本书根植于我们多年来开发和维护的大量软件、多年讲授程序设计的课程以及多年与各种各样的程序员共同工作中的经历。我们希望与人共享这些关于程序设计实践问题的经验和教训，从我们的经验中传递一些见识，提出一些方法和建议，以帮助各个层次的程序员变得更加熟练，工作更有效率。

我们的书是为许多不同类型的读者写的。如果你是学生，已经上过一两门程序设计课程，希望成为一个更好的程序员，这本书将为你展开某些问题，而这些问题在你自己的课程里没有足够的时间讨论。如果你把写程序作为自己的一部分任务，目的是为了完成其他工作而不

是做程序本身，这里的信息将帮助你更有效地编写程序。如果你是职业程序员，在学校期间没有学过这些内容，或者希望重新温习一下，或者你是个软件管理者，希望指导你的工作人员向正确方向发展，这里的材料都会是很有价值的。

我们希望这里提出的建议能帮助人们写出更好的程序。读这本书的惟一先决条件，是你已经做过一些程序设计，最好是用C、C++ 或者Java语言编写。当然，已有的经验越多，有关内容对你来说就越容易。不可能有任何东西能帮你在21天内从新手变成专家。对于这里的某些例子，Unix和Linux程序员可能比经常用Windows和Macintosh的程序员更熟悉一点。但是我们相信，无论你来自哪个环境，都一定能在这里发现一些东西，它们可能使你的生活变得更容易些。

本书全部内容分为9章，每章集中讨论程序设计实践的一个重要方面。

第1章讨论程序设计风格。好的风格对于好的程序设计是如此重要，所以我们选择在最前面讨论它。书写良好的程序比胡乱写出的程序好——它们包含的错误更少，更容易排错、容易修改。所以，最重要的是从一开始就注意风格。在这一章里还提出了好程序设计的一个重要论点：应该尽量采用对所用语言最合适的习惯用法。

算法和数据结构是第2章的论题，这是计算机科学教育计划的核心，也是程序设计课程的重要组成部分。由于许多读者已经熟悉了这方面的情况，本书中的处理方式，是对那些几乎出现在每个程序里的算法和数据结构做一个简单的总结。更复杂的算法和数据结构通常都是由这些基本东西构建起来的，因此需要掌握这些最基本的东西。

第3章讨论一个小程序的设计，以便在实际工作中展示和讨论算法和数据结构的各种有关问题。这个程序用5种不同语言实现，通过对不同实现的比较，说明同样的数据结构如何在不同语言里处理以及语言之间在表达能力和性能方面的变化情况等。

用户和程序之间、程序不同部分之间的界面是程序设计的基本问题。软件的成功很大程度上取决于这些界面设计和实现。第4章用一个小函数库作为例子，展示了界面的演化过程，这个库用于分析处理一种广泛使用的数据格式。虽然这个例子很小，但却显示出界面设计的许多重要问题：抽象、信息隐蔽、资源管理和错误处理等等。

虽然我们总是希望第一次就能把程序写正确，但是，程序错误以及相应的排错过程实际上是无法避免的。第5章讨论了系统化地、有效地排除程序错误的各种策略和技术。有关论题中包括常见错误的标志、“数值推断”的重要性等。程序输出中模式性的东西常常能告诉我们问题出在哪里。

测试的目的就是企图给出一种合理的保证，说明一个程序能正常工作，或者说明它经过改造之后仍然是正确的。第6章强调的是如何通过手工和机械的方式进行系统化的程序测试。边界条件测试针对潜在的薄弱点进行仔细检查。机械化和测试台测试使人可以比较容易地用合理代价完成昂贵的测试。应力测试提供与典型用户所用方法不同的另一类测试，它可能搜寻出另外一些种类的错误。

计算机的速度已经是风驰电掣，编译器已经做得如此之好，因此，许多程序在写出之时就已经足够快了。但是，另一些程序则实在是太慢，或者是使用了太多的存储，甚至同时存在这两方面的问题。第7章给出了一种有序方法处理这些问题，设法改造程序，使之能更有效地使用资源，并使它在提高效率的同时仍然是正确和有效的。

第8章讨论可移植性问题。一个成功的程序可能在环境不断变化的情况下使用相当长时间，

或者需要适用于新的系统、新的硬件、甚至新的国家。可移植性的目标就是设法减少程序的维护，使程序在移植到新环境时要做的改动达到最少。

计算是一个语言丰富的领域，在这里，不止有我们常用来做程序设计的通用语言，还有许多集中应用在较窄领域中的专门语言。第9章给出了一些例子，说明在计算领域里记法的重要性。还想说明怎样利用这方面的东西来简化程序，指导程序设计，或者帮助我们编写那种能够写程序的程序。

由于要讨论的是程序设计，我们不得不给出许多代码。大部分例子是为本书特别写的，也有些小例子取自其他来源。为把这些代码写好，我们已经做了很大努力，而且在6个系统上测试过这些代码，直接采用机器可读的形式。关于《程序设计实践》这本书的更多信息可以在下面的网址中找到：

```
http://tpop.awl.com
```

本书里的大部分程序是用C语言写的，也有一些用C++ 或Java写的程序例子，还有关于脚本语言的简短讨论。在最低的层次上，C和C++ 几乎完全一样，因此我们的C程序也是C++ 程序。C++ 和Java是C语言的直系后代，它们从C继承的远不止是一点语法，更多的是其效率和表达能力，还有丰富的类型系统和函数库。我们在自己的工作中经常使用这三种语言，此外还使用一些其他语言。语言的选择依赖于问题本身：操作系统最好用某种高效的、限制较少的语言来写，比如用C或者C++；快速原型用某个命令解释器或者脚本语言写常常是最容易的，比如用Awk或Perl；对于用户界面，Visual Basic或Tcl/Tk，还有Java，都是强有力的候选者。

对于我们的例子，如何选择语言还有一个重要的教学因素。正如没有一种语言能够很好地解决所有问题一样，也没有一种语言对说明这里的所有问题都是最好的。更高级的语言实际上是预先确定了某些设计决策。如果用低级一些的语言，我们将能讨论对问题的各种解答，通过揭示其中的更多细节，有可能更好地处理它们。经验证明，即使是在使用高级的语言机制时，了解它们与低层次问题的关系也是非常有价值的。如果没有这种洞察力，我们就很容易陷入性能或者其他奇怪的问题之中。所以，我们在写例子时经常使用C语言，即使是在那些实际可能会采用其他语言的地方。

实际上，本书的大部分内容是独立于任何特定程序设计语言的。数据结构的实现受到所使用语言的影响，在有些语言里选择的可能性少些，有些语言可能支持更多的选择。但是话说回来，在这方面做选择的方式都是一样的。在不同语言里，如何做测试、如何排除程序错误等也都会有许多不同，但其中的策略和技巧又是类似的。改进程序效率的大部分技术都可以应用于任何语言。

无论用什么语言写程序，作为程序员的工作都是尽可能地利用手头的工具，尽可能地把事情做好。优秀的程序员可以克服一种不良语言或者一个伪劣的操作系统带来的坏影响，但是，即使最优异的程序设计环境也帮不了一个糟糕的程序员的忙。我们的希望是，无论你当前的经验和技术如何，这本书都能帮助你更好地编程并从中取得更多的乐趣。

我们衷心地感谢所有那些朋友和同事，他们读过本书的草稿，给了我们许多很有帮助的建议。Jon Bentley，Russ Cox，John Lakos，John Linderman，Peter Memishian，Ian Lance Taylor，Howard Trickey和Chris Van Wyk极其认真地通读过本书的手稿，有人还读过不止一遍。我们感谢Tom Cargill，Chris Cleeland，Steve Dewhurst，Eric Grosse，Andrew Herron，Gerard Holzmann，Doug McIlroy，Paul McNamee，Peter Nelson，Dennis Ritchie，Rich

Stevens, Tom Szymanski, Kentaro Toyama, John Wait, Daniel C. Wang, Peter Weinberger, Margaret Wright和Cliff Young对手稿各个方面提出的极有价值的意见。我们也感谢Al Aho, Ken Arnold, Chuck Bigelow, Joshua Bloch, Bill Coughran, Bob Flandrena, Renée French, Mark Kernighan, Andy Koenig, Sape Mullender, Evi Nemeth, Marty Rabinowitz, Mark V. Shaney, Bjarne Stroustrup, Ken Thompson和Phil Wadler等的好建议和经过深思熟虑的提议。谢谢你们大家。

Brian W. Kernighan
Rob Pike

目　录

第1章 风 格

人们看到最好的作家有时并不理会修辞学的规则。还好，当他们这样做虽然付出了违反常规的代价，读者还经常能从句子中发现某些具有补偿性的价值。除非作者自己也明确其做法的意思，否则最好还是按规矩做。

William Strunk和E. B. White，《风格的要素》

下面这段代码取自一个许多年前写的大程序：

```
if ( (country == SING) || (country == BRNI) ||
     (country == POL) || (country == ITALY) )
{
        /*
         * If the country is Singapore, Brunei or Poland
         * then the current time is the answer time
         * rather than the off hook time.
         * Reset answer time and set day of week.
         */
        ...
```

这段代码写得很仔细，具有很好的格式。它所在的程序也工作得很好。写这个系统的程序员会对他们的工作感到骄傲。但是这段摘录却会把细心的读者搞糊涂：新加坡、文莱、波兰和意大利之间有什么关系？为什么在注释里没有提到意大利？由于注释与代码不同，其中必然有一个有错，也可能两个都不对。这段代码经过了执行和测试，所以它可能没有问题。注释中对提到的三个国家间的关系没有讲清楚，如果你要维护这些代码，就必须知道更多的东西。

上面这几行实际代码是非常典型的：大致上写得不错，但也还存在许多应该改进的地方。

本书关心的是程序设计实践，关心怎样写出实际的程序。我们的目的是帮助读者写出这样的软件，它至少像上面的代码所在的程序那样工作得非常好，而同时又能避免那些污点和弱点。我们将讨论如何从一开始就写出更好的代码，以及如何在代码的发展过程中进一步改进它。

我们将从一个很平凡的地方入手，首先讨论程序设计的风格问题。风格的作用主要就是使代码容易读，无论是对程序员本人，还是对其他人。好的风格对于好的程序设计具有关键性作用。我们希望最先谈论风格，也是为了使读者在阅读本书其余部分时能特别注意这个问题。

写好一个程序，当然需要使它符合语法规则、修正其中的错误和使它运行得足够快，但是实际应该做的远比这多得多。程序不仅需要给计算机读，也要给程序员读。一个写得好的程序比那些写得差的程序更容易读、更容易修改。经过了如何写好程序的训练，生产的代码更可能是正确的。幸运的是，这种训练并不太困难。

程序设计风格的原则根源于由实际经验中得到的常识，它不是随意的规则或者处方。代码应该是清楚的和简单的——具有直截了当的逻辑、自然的表达式、通行的语言使用方式、

有意义的名字和有帮助作用的注释等，应该避免耍小聪明的花招，不使用非正规的结构。一致性是非常重要的东西，如果大家都坚持同样的风格，其他人就会发现你的代码很容易读，你也容易读懂其他人的。风格的细节可以通过一些局部规定，或管理性的公告，或者通过程序来处理。如果没有这类东西，那么最好就是遵循大众广泛采纳的规矩。我们在这里将遵循《C程序设计语言》(The C Programming Language)一书中所使用的风格，在处理Java和C++ 程序时做一些小的调整。

我们一般将用一些好的和不好的小程序设计例子来说明与风格有关的规则，因为对处理同样事物的两种方式做比较常常很有启发性。这些例子不是人为臆造的，不好的一个都来自实际代码，由那些在太多工作负担和太少时间的压力下工作的普通程序员(偶然就是我们自己)写出来。为了简单，这里对有些代码做了些精练，但并没有对它们做任何错误的解释。在看到这些代码之后，我们将重写它们，说明如何对它们做些改进。由于这里使用的都是真实代码，所以代码中可能存在多方面问题。要指出代码里的所有缺点，有时可能会使我们远离讨论的主题。因此，在有的好代码例子里也会遗留下一些未加指明的缺陷。

为了指明一段代码是不好的，在本书中，我们将在有问题的代码段的前面标出一些问号，就像下面这段：

```
? #define ONE 1
? #define TEN 10
? #define TWENTY 20
```

为什么这些#define有问题？请想一想，如果某个具有TWENTY个元素的数组需要修改得更大一点，情况将会怎么样。至少这里的每个名字都应该换一下，改成能说明这些特殊值在程序中所起作用的东西。

```
#define INPUT_MODE 1
#define INPUT_BUFSIZE 10
#define OUTPUT_BUFSIZE 20
```

1.1 名字

什么是名字？一个变量或函数的名字标识这个对象，带着说明其用途的一些信息。一个名字应该是非形式的、简练的、容易记忆的，如果可能的话，最好是能够拼读的。许多信息来自上下文和作用范围(作用域)。一个变量的作用域越大，它的名字所携带的信息就应该越多。

全局变量使用具有说明性的名字，局部变量用短名字。根据定义，全局变量可以出现在整个程序中的任何地方，因此它们的名字应该足够长，具有足够的说明性，以便使读者能够记得它们是干什么用的。给每个全局变量声明附一个简短注释也非常有帮助：

```
int npending = 0;  // current length of input queue
```

全局函数、类和结构也都应该有说明性的名字，以表明它们在程序里扮演的角色。

相反，对局部变量使用短名字就够了。在函数里，n可能就足够了，npoints也还可以，用numberOfPoints就太过分了。

按常规方式使用的局部变量可以采用极短的名字。例如用i、j作为循环变量，p、q作为指针，s、t表示字符串等。这些东西使用得如此普遍，采用更长的名字不会有什么益处或收获，可能反而有害。比较：

```
for (theElementIndex = 0; theElementIndex < numberOfElements;
        theElementIndex++)
```

```
?       elementArray[theElementIndex] = theElementIndex;
```

和

```
        for (i = 0; i < nelems; i++)
            elem[i] = i;
```

人们常常鼓励程序员使用长的变量名，而不管用在什么地方。这种认识完全是错误的，清晰性经常是随着简洁而来的。

现实中存在许多命名约定或者本地习惯。常见的比如：指针采用以p结尾的变量名，例如nodep；全局变量用大写开头的变量名，例如Global；常量用完全由大写字母拼写的变量名，如CONSTANTS等。有些程序设计工场采用的规则更加彻底，他们要求把变量的类型和用途等都编排进变量名字中。例如用pch说明这是一个字符指针，用strTo和strFrom表示它们分别是将要被读或者被写的字符串等。至于名字本身的拼写形式，是使用npending或numPending还是num_pending，这些不过是个人的喜好问题，与始终如一地坚持一种切合实际的约定相比，这些特殊规矩并不那么重要。

命名约定能使自己的代码更容易理解，对别人写的代码也是一样。这些约定也使人在写代码时更容易决定事物的命名。对于长的程序，选择那些好的、具有说明性的、系统化的名字就更加重要。

C++ 的名字空间和Java的包为管理各种名字的作用域提供了方法，能帮助我们保持名字的意义清晰，又能避免过长的名字。

保持一致性。相关的东西应给以相关的名字，以说明它们的关系和差异。

除了太长之外，下面这个Java类中各成员的名字一致性也很差：

```
?   class UserQueue {
?       int noOfItemsInQ, frontOfTheQueue, queueCapacity;
?       public int noOfUsersInQueue() {...}
?   }
```

这里同一个词“队列(queue)”在名字里被分别写为Q、Queue或queue。由于只能在类型UserQueue里访问，类成员的名字中完全不必提到队列，因为存在上下文。所以：

```
?   queue.queueCapacity
```

完全是多余的。下面的写法更好：

```
class UserQueue {
    int nitems, front, capacity;
    public int nusers() {...}
}
```

因为这时可以如此写：

```
queue.capacity++;
n = queue.nusers();
```

这样做在清晰性方面没有任何损失。在这里还有可做的事情。例如items和users实际是同一种东西，同样东西应该使用一个概念。

函数采用动作性的名字。函数名应当用动作性的动词，后面可以跟着名词：

```
now = date.getTime();
putchar('\n');
```

对返回布尔类型值(真或者假)的函数命名，应该清楚地反映其返回值情况。下面这样的语句

```
?   if (checkoctal(c)) ...
```

是不好的，因为它没有指明什么时候返回真，什么时候返回假。而：

```
if (isoctal(c)) ...
```

就把事情说清楚了：如果参数是八进制数字则返回真，否则为假。

要准确。名字不仅是个标记，它还携带着给读程序人的信息。误用的名字可能引起奇怪的程序错误。

本书作者之一写过一个名为isoctal的宏，并且发布使用多年，而实际上它的实现是错误的：

```
? #define isoctal(c) ((c) >= '0' && (c) <= '8')
```

正确的应该是：

```
#define isoctal(c) ((c) >= '0' && (c) <= '7')
```

这是另外一种情况：名字具有正确的含义，而对应的实现却是错的，一个合情合理的名字掩盖了一个害人的实现。

下面是另一个例子，其中的名字和实现完全是矛盾的：

```
? public boolean inTable(Object obj) {
?     int j = this.getIndex(obj);
?     return (j == nTable);
? }
```

函数getIndex如果找到了有关对象，就返回0到nTable-1之间的一个值；否则返回nTable值。而这里inTable返回的布尔值却正好与它名字所说的相反。在写这段代码时，这种写法未必会引起什么问题。但如果后来修改这个程序，很可能是由别的程序员来做，这个名字肯定会把人弄糊涂。

练习1-1 评论下面代码中名字和值的选择：

```
? #define TRUE 0
? #define FALSE 1
?
? if ((ch = getchar()) == EOF)
?     not_eof = FALSE;
```

练习1-2 改进下面的函数：

```
? int smaller(char *s, char *t) {
?     if (strcmp(s, t) < 1)
?         return 1;
?     else
?         return 0;
? }
```

练习1-3 大声读出下面的代码：

```
? if ((falloc(SMRHSHSCRTCH, S_IFEXT|0644, MAXRODDHSH)) < 0)
?     ...
```

1.2 表达式和语句

名字的合理选择可以帮助读者理解程序，同样，我们也应该以尽可能一目了然的形式写好表达式和语句。应该写最清晰的代码，通过给运算符两边加空格的方式说明分组情况，更一般的是通过格式化的方式来帮助阅读。这些都是很琐碎的事情，但却又是非常有价值的，就像保持书桌整洁能使你容易找到东西一样。与你的书桌不同的是，你的程序代码很可能还会被别人使用。

用缩行显示程序的结构。采用一种一致的缩行风格，是使程序呈现出结构清晰的最省力的方法。下面这个例子的格式太糟糕了：

```
    for(n++;n<100;field[n++]='\0');
    *i = '\0'; return('\n');
```

重新调整格式，可以改得好一点：

```
    for (n++; n < 100; field[n++] = '\0')
        ;
    *i = '\0';
    return('\n');
```

更好的是把赋值作为循环体，增量运算单独写。这样循环的格式更普通也更容易理解：

```
for (n++; n < 100; n++)
    field[n] = '\0';
*i = '\0';
return '\n';
```

使用表达式的自然形式。表达式应该写得你能大声念出来。含有否定运算的条件表达式比较难理解：

```
    if (!(block_id < actblks) || !(block_id >= unblocks))
        ...
```

在两个测试中都用到否定，而它们都不是必要的。应该改变关系运算符的方向，使测试变成肯定的：

```
if ((block_id >= actblks) || (block_id < unblocks))
    ...
```

现在代码读起来就自然多了。

用加括号的方式排除二义性。括号表示分组，即使有时并不必要，加了括号也可能把意图表示得更清楚。在上面的例子里，内层括号就不是必需的，但加上它们没有坏处。熟练的程序员会忽略它们，因为关系运算符(< <= == != >= >)比逻辑运算符(&&和||)的优先级更高。

在混合使用互相无关的运算符时，多写几个括号是个好主意。C语言以及与之相关的语言存在很险恶的优先级问题，在这里很容易犯错误。例如，由于逻辑运算符的约束力比赋值运算符强，在大部分混合使用它们的表达式中，括号都是必需的。

```
while ((c = getchar()) != EOF)
    ...
```

字位运算符(&和|)的优先级低于关系运算符(比如==)，不管出现在哪里：

```
    if (x&MASK == BITS)
        ...
```

实际上都意味着

```
    if (x & (MASK==BITS))
        ...
```

这个表达式所表达的肯定不会是程序员的本意。在这里混合使用了字位运算和关系运算符号，表达式里必须加上括号：

```
if ((x&MASK) == BITS)
    ...
```

如果一个表达式的分组情况不是一目了然的话，加上括号也可能有些帮助，虽然这种括

号可能不是必需的。下面的代码本来不必加括号:

```
?    leap_year = y % 4 == 0 && y % 100 != 0 || y % 400 == 0;
```

但加上括号，代码将变得更容易理解了:

```
leap_year = ((y%4 == 0) && (y%100 != 0)) || (y%400 == 0);
```

这里还去掉了几个空格：使优先级高的运算符与运算对象连在一起，帮助读者更快地看清表达式的结构。

分解复杂的表达式。C、C++和Java语言都有很丰富的表达式语法结构和很丰富的运算符。因此，在这些语言里就很容易把一大堆东西塞进一个结构中。下面这样的表达式虽然很紧凑，但是它塞进一个语句里的东西确实太多了:

```
?    *x += (*xp=(2*k < (n-m) ? c[k+1] : d[k--]));
```

把它分解成几个部分，意思更容易把握:

```
if (2*k < n-m)
    *xp = c[k+1];
else
    *xp = d[k--];
*x += *xp;
```

要清晰。程序员有时把自己无穷尽的创造力用到了写最简短的代码上，或者用在寻求得到结果的最巧妙方法上。有时这种技能是用错了地方，因为我们的目标应该是写出最清晰的代码，而不是最巧妙的代码。

下面这个难懂的计算到底想做什么?

```
?    subkey = subkey >> (bitoff - ((bitoff >> 3) << 3));
```

最内层表达式把bitoff右移3位，结果又被重新移回来。这样做实际上是把变量的最低3位设置为0。从bitoff的原值里面减掉这个结果，得到的将是bitoff的最低3位。最后用这3位的值确定subkey的右移位数。

上面的表达式与下面这个等价:

```
subkey = subkey >> (bitoff & 0x7);
```

要弄清前一个版本的意思简直像猜谜语，而后面这个则又短又清楚。经验丰富的程序员会把它写得更短，换一个赋值运算符:

```
subkey >>= bitoff & 0x7;
```

有些结构似乎总是要引诱人们去滥用它们。运算符?:大概属于这一类:

```
?    child=(!LC&&!RC)?0:(!LC?RC:LC);
```

如果不仔细地追踪这个表达式的每条路径，就几乎没办法弄清它到底是在做什么。下面的形式虽然长了一点，但却更容易理解，其中的每条路径都非常明显:

```
if (LC == 0 && RC == 0)
    child = 0;
else if (LC == 0)
    child = RC;
else
    child = LC;
```

运算符?:适用于短的表达式，这时它可以把4行的if-else程序变成1行。例如这样:

```
max = (a > b) ? a : b;
```

或者下面这样:

```
printf("The list has %d item%s\n", n, n==1 ? "" : "s");
```