O'REILLY®

异步图书
www.epubit.com.cn

第2版

C程序设计新思维

21st Century C

[美] 本·克莱蒙 (Ben Klemens) 著
赵岩 译

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

C程序设计新思维（第2版）

[美] 本·克莱蒙 (Ben Klemens) 著

赵岩 译

Beiji · Tokyo O'REILLY®

O'Reilly Media, Inc. 授权人民邮电出版社出版

人 民 邮 电 出 版 社

北 京

图书在版编目（CIP）数据

C程序设计新思维 : 第2版 / (美) 克莱蒙(Ben Klemens) 著 ; 赵岩译. -- 北京 : 人民邮电出版社, 2018.1
ISBN 978-7-115-46095-0

Ⅰ. ①C… Ⅱ. ①克… ②赵… Ⅲ. ①C语言－程序设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2017)第281300号

版权声明

◆ 著　[美] 本·克莱蒙（Ben Klemens）
译　赵　岩
责任编辑　胡俊英
责任印制　焦志炜

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路 11 号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷

◆ 开本：800×1000　1/16
印张：24
字数：448 千字　2018 年 1 月第 1 版
印数：1－2 400 册　2018 年 1 月北京第 1 次印刷

著作权合同登记号　图字：01-2015-7686 号

定价：79.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

内容提要

C 语言已经有几十年的历史了。经过长时间的发展和普及，C 语言的应用场景有了很大的变化，一些旧观念应该被淡化或者不再被推荐。

本书展现了传统 C 语言教科书所不具有的最新的相关技术。全书分为开发环境和语言两个部分，从编译、调试、测试、打包、版本控制等角度，以及指针、语法、文本、结构、面向对象编程、函数库等方面，对 C 程序设计的核心知识进行查缺补漏和反思。本书鼓励读者放弃那些对大型机才有意义的旧习惯，拿起新的工具来使用这门与时俱进的简洁语言。

本书适合有一定基础的 C 程序员和 C 语言学习者阅读，也适合想要深入理解 C 语言特性的读者参考。

译者序

最近非常有幸地接受了人民邮电出版社的邀请来翻译本书，这是一本非常经典的C语言著作，目前已经是第2版了。计算机图书出了很多年，大家对其自有判断，最简单的办法就是根据书名，20世纪90年代末期出版过几本比较经典的计算机图书，书名为：《**入门到精通》《21天学会**》等。不过很快大家就开始借用这种书名，最后搞得有些良莠不齐。更有甚者，最近出现了好多类似《**从入门到放弃》《**从入门到入院》的图书，彻底颠覆了以前程序员心目中这么神圣的书名。好在还有O'Reilly出版社的动物丛书，目前还都是品质和经典的象征。这套书经典到有时圈内的人们都忘了书名，只记得动物的名字，例如Perl语言的“骆驼书”以及Git的“蝙蝠书”等。

当完成最后一个字的录入，作为本书的译者，我认为应该系统地给这本书做个总结了。首先，这是一本经典的C语言图书，亚马逊上有50多条评论，评分达到4分。这个傲人的成绩主要来自于本书的两个优点：第一个优点就是系统性和大局观。C语言最开始作为开发UNIX操作系统的工具，它和UNIX操作系统有着不可分割的关系。无论UNIX派生的POSIX标准以及GNU运动，C语言都是其核心的开发语言和工具。所以如果想真正发挥C语言的威力，那必须要把这个语言放到一个更大的生态环境中去。本书通过对POSIX标准库、GNU编译器、Shell脚本、Make、Git、文档和测试，以及各种常用的函数库等一系列内容的介绍，建立了一个高效整合的C语言开发环境。C语言作为这个环境的核心开发语言，通过各种开发工具和函数库的配合，将开发环境的优点淋漓尽致地发挥出来，从而能显著地提高你的开发效率。

第二个优点就是新思维和反规则。作为物理学的爱好者，我用物理来做一个类比。牛顿创立了经典力学和万有引力。正当我们认为物理学已经完胜的时候，爱因斯坦在一边幽幽地说的那句“光会拐弯”。在爱因斯坦的结论在观测日全食方面得到验证以后，这位天才自信心爆棚并宣称“一切都是可以通过计算来确定的”。这个时候研究量子力学的波尔却传给了他一个纸条说：“上帝掷骰子!”也许，我是说也许，我们一直都相信的规则或者答案过于片面。就像有一天我6岁的女儿小米粒问我：

“我们人类是从哪里来的啊？”我说：“有人说是猴子变的，有人说是神创造的，你信哪种说法都可以。你告诉爸爸，你信哪种啊？”我的女儿想都没想就回答到：“是神把猴子变成人的！”

我们人类总是有一种倾向，一旦形成了自己的某些规则，那么就会自然地排斥和否定另外的反规则。而本书的难能可贵之处就在于，它不仅提出了 C 语言的一些反规则，而且通过一些例子证明这些反规则是合理的。例如，我们可以建造高效和准确的宏，可以不需要对内存的使用斤斤计较，哪怕有点内存泄漏，我们可以用 goto，但是对 switch 却完全可以放弃，等等。现代物理有一个反物质学说，当物质和反物质相遇时，二者会立即湮没，并爆发出巨大的能量。这里我借用一下：当你熟悉了规则，同时也理解了反规则，这个时候你的心中就没有了规则。剩下的就是巨大的能力。此时小李飞刀已经不带刀，此时无招已经胜有招。

俗话说：“优点不说没不了，缺点不说不得了。”下面说说本书的缺点，那就是：对每一部分的内容并没有详细地介绍！所以你不要指望着阅读完本书就能熟练地使用 shell 脚本，写出复杂的 makefile 并通过 Git 高效地与人协作。坦白地说，这也并不算是缺点。本书的目的就是告诉你，当你想做什么事情的时候，有哪些工具你可以用，而这些工具的最常见的用法又是什么。当你发现这些工具并不够用的时候，你可以去找专门的介绍相关内容的书。这个时候你会发现完整介绍某些工具的书，它的厚度足以挡住狙击步枪射出的子弹。同时，POSIX 体系内的东西，有臭名昭著的学习曲线。你要是不服就下载一个 VIM 编辑器试试看！

从安装 VIM 到现在，我一直在使用它，但我一直搞不懂如何退出 VIM。大家都说 C 语言难，客观地来讲，这个锅不能让 C 语言和本书来背。但是，你想要“会当凌绝顶，一览众山小”，那你的学习曲线必须要很陡才行！

另外，本书的第 3.3 节“用 Autotools 打包代码”，我个人认为这个技术有些过时。目前流行的 build system 是 CMake 系统，它在移植性和易用性上都要比 Autotoolson 工具要好。所以 CMake 是新开发系统的首选 build system，也是未来的潮流。

最后说说本书面向的读者对象。首先，本书并不是教材，虽然书的后面有一个简短的介绍 C 语言的附录，我没有骗你，它确实很简短。请注意我的用词，我只是说“简短”，并没有说“简单”。所以，如果你是一个 C 语言的初学者或者是零基础读者，那么本书并不适合你。

本书面向的读者对象是有一定 C 语言基础的高年级学生，或者是一些使用 C 语言作为主要开发语言的工程师。对于大学高年级的学生，它们缺乏的是一种对大的编程环境的认识。而对于使用 C 语言的从业人员，本书会让你对 C 语言有不一样的

认识。它对你多年使用 C 语言形成的习惯和风格提出了挑战，让你有一种“原来 C 语言也可以这么用”的赞叹！然后让这些反规则去湮没你心中存在多年的规则，从而爆发出巨大的能量！

最后再提醒一句，本书真的不适合初学者！否则，你看不懂会说我翻译得不好。我已经把 $E=MC^2$ 翻译成了“能量等于质量乘以光速的平方”，如果你还看不懂，就别怪译者了。

举个简单的例子，书中 8.2 节最后一段是：

这意味着如果两个文件中的两个变量有共同的名字，但是你想它们应该彼此独立的。这个时候如果你忘记 static 关键字，编译器可以把有外部链接变量的链接成一个变量。这种细微的 bug 非常容易发生，所以对于那些有内部链接的变量不要忘了使用 static。

如果你不了解 tentative definition(临时定义)这一 C 语言中特有的现象，就不能充分理解“这种细微的 bug”究竟是什么。所以为了让读者能更容易地阅读本书，我为本书做了一个网站: http://zhaoyan.website/xinzhi/c21/book.php，里面有我对每一章的观点，同时推荐了一些帮助读者理解的补充内容等。例如，对于上面的 tentative definition 的问题，我给出了非常好的介绍，同时还有我自己编写的一段程序对这一概念进行了说明。对于本书的每部分内容过于简短和艰深这一缺点，我也做了一些有益的补充和修正。

最后我想说的是：这本书真的很好！这本书真的很难！你有勇气挑战一下吗？

——赵岩

2017 年 5 月

前言

C 就是 Punk Rock

虽然 C 仅有为数不多的关键词，并且没有那么多细节修饰，但是它很棒[1]！你可以用 C 来做任何事情。它就像一把吉他上的 C、G 和 D 弦，你很快就可以掌握其基本原理，然后就得用你的余生来提高。不理解它的人害怕它的威力，并认为它粗糙得不够安全。虽然没有企业和组织花钱去推广它[2]，但是实际上它在所有的编程语言排名中一直被认为是最流行的语言。

这门语言已经有几十年的历史了，可以说已经进入了中年。创造它的是少数对抗管理阶层并遵从完美的 punk rock 精神的人；但那是 20 世纪 70 年代的事情了，现在这门语言已经历尽沧桑，并且成为主流的语言。

当 punk rock 变成主流的时候人们会怎样？在其从 20 世纪 70 年代出现后的几十年里，punk rock 已经从边缘走向中心：The Clash、The Offspring、Green Day 和 The Strokes 等乐队已经在全世界卖出了几百万张唱片（这还只是一小部分），我也在家附近的超市里听过被称为 grunge 的一些精简乐器版本的 punk rock 分支。Sleater-Kinney 乐队的前主唱还经常在自己那个很受欢迎的喜剧节目中讽刺 punk rocker 音乐人[3]。对这种持续的进化，一种反应是划一条界限，将原来的风格称为 punk rock，而将其余的东西称为面向大众的粗浅的 punk。传统主义者还是可以播放 20 世纪 70 年代的唱片，但如果唱片的音轨磨损了，他们可以购买数码版本，就像他们为自己的小孩购买 Ramones 牌的连帽衫一样。

外行是不明白的。有些人听到 punk 这个词时脑海里就勾画出 20 世纪 70 年代特定的景象，经常的历史错觉就是那个时代的孩子们真的在做什么不同的事情。喜欢欣

1 “it rocks”，此处原文为双关语，借用英语中 rock 的不同含义，即“摇滚乐”和“很棒”。标题中的“punk rock”为流行于 20 世纪 70 年代的一种摇滚乐风格，以狂野反叛为特色，国内也称为“朋克”。——译者注

2 这篇前言明显地，而且是必须向 *Punk Rock Language: A Polemic* 致敬，作者是 Chris Adamson。

3 像“can't get to heaven with a three-chord song”这样的歌词，可能会让 Sleater-Kinney 被归类在后 punk 时期？不幸的是，没有 ISO punk 标准为各种音乐提供精确的定义。

赏 1973 年 Iggy Pop 的黑胶唱片的传统主义者一直是那么兴趣盎然，但是他们有意无意地加强了那种 punk rock 已经停滞不前的刻板印象。

回到 C 的世界里，这里既有挥舞着 ANSI'89 标准大旗的传统主义者，也有那些拥抱变化，甚至都没有意识到如果回到 20 世纪 90 年代，他们写的代码都不可能被成功编译与运行的人。外行人是不会知道个中缘由的。他们看到从 20 世纪 80 年代起至今还在印刷的书籍和 20 世纪 90 年代起至今还存于网上的教程，他们听到的都是坚持当年的软件编写方式的、死硬的传统主义者的言论，他们甚至都不知道语言本身和别的用户都在一直进化。非常可惜，他们错过了一些好东西。

这是一本打破传统并保持 C 语言 punk 精神的书。我对将本书的代码与 1978 年 Kernighan 和 Ritchie 出版的书[1]中的 C 标准进行对比毫无兴趣。既然连我的电话都有 512MB 内存，为什么还在我的书里花费章节讲述如何为可执行文件减少几千字节呢？我正在一个廉价的红色上网本上写这本书，而它却可以每秒运行 3 200 000 000 条指令，那为什么我还要操心 8 位和 16 位所带来的操作的差异呢？我们更应该关注如何做到快速编写代码并且让我们的合作者们更容易看懂。毕竟我们是在使用 C 语言，所以我们那些易读但是并没有被完美优化的代码运行起来还是会比很多烦琐的语言明显更快。

Q&A（本书的参考引用）[2]

问题：这本书与其他书有什么不同？

答案：有些书写得好，有些书写得有趣，但是大部分 C 语言的教科书都非常相像（我曾经读过很多这样的教科书，包括[Griffiths，2012]、[Kernighan，1978]、[Kernighan，1988]、[Kochan，2004]、[Oualline，1997]、[Perry，1994]、[Prata，2004]和[Ullman，2004]。多数教材都是在 C99 标准发布并简化了很多用法之后写成的，你可以看到现在出版的这些教材的第 *N* 版仅仅在一些标注上做了一点说明，而不是认真反思了如何使用这门语言。它们都提到你可以拥有一些库来编写你自己的代码，但是书籍完成时，缺少了保障库的可靠性和可移植性的安装与开发环境。那些教科书现在仍然有效并且具有自己的价值，但是现代的 C 代码已经看起来和那些教科书里面的不太一样了。

这本书与那些教科书的不同之处，在于对这门语言及其开发环境进行了拾遗补漏。

[1] 从下文可以看到，该书的出版被认为是 C 语言诞生的标志性事件，业内常称为 *K&R*。——译者注

[2] 这里作者将问题与解答比喻为 C 语言函数入口的参考引用。——译者注

书中讲解的方式是：直接使用提供了链表结构和 XML 解析器的现成的库，而不是把这些从头再写一次。这本书也体现了如何编写易读代码和用户友好的函数接口。

问题：这本书的目标读者是谁？我需要是一个编程大师吗？

答案：你必须有某种语言的编程经验，或许是 Java，或者是类似于 Perl 的某种脚本语言。这样我就没有必要再向你讲为什么你不应该写一个很长的没有任何子函数的函数了。

本书的内容假设你已经有了通过写 C 代码而获得的 C 语言的基本知识。附录 A 提供了一个简短的有关 C 语言基础的教程，那些以前写 Python 和 Ruby 等脚本语言的读者可以阅读它。

请允许我介绍我写的另一本关于统计和科学计算的教科书 *Modeling with Data* [Klemens，2008]。那本书不仅提供了很多关于如何处理数值和统计模型的内容，它还可以用作一本独立的 C 语言的教材，并且我认为那本书还避免了很多早期教材的缺点。

问题：我是个编写应用软件的程序员，不是一个研究操作系统内核的人。为什么我应该用 C 而不是像 Python 这类可以快速编程的脚本语言呢？

答案：如果你是一个应用软件程序员的话，这本书就是为你准备的。我知道人们经常认定 C 是一种系统语言，这让我觉得真是缺少了点 punk 的反叛精神——他们是谁啊？要他们告诉我们要用什么语言？

像“我们的语言几乎和 C 一样快，但更容易编写”这样的言论很多，简直成了陈词滥调。好吧，C 显然是和 C 一样快，并且这本书的目的是告诉你 C 也像以前的教科书所暗示的那样容易使用。你没必要使用 malloc，也没必要像 20 世纪 90 年代的系统程序员那样深深卷入内存管理，我们已经有处理字符串的手段，甚至核心语法也进化到了支持更易读的代码的境界。

我当初正式学习 C 语言是为了加速一个用脚本语言 R 编写的仿真程序。和众多的脚本语言一样，R 具有 C 接口并且鼓励用户在宿主语言[1]太慢的时候使用。最终我的程序里有太多的从 R 语言到 C 语言的调用，最后我索性放弃了宿主语言。随后发生的事情你已经知道，就是我在写这本关于现代 C 语言技术的书。

问题：如果原本使用脚本语言的应用软件程序员能喜欢这本书当然好，但我是一名内核黑客。我在五年级的时候就自学了 C 语言，有时做梦都在正确编译。那这本

[1] 这里指调用 C 语言的语言。——译者注

书还有什么新鲜的吗?

答案：C 语言在过去的几年里真的进化了很多。就像我下面要讨论的那样，各个编译器对新功能的支持的时间也不一样，感谢自从 ANSI 标准发布后，又发布了两个新的 C 语言标准。也许你应该读一下第 10 章，找找有什么能叫你感到惊讶的。本书的一部分，如讲解指针的经常被人错误理解的一些概念（第 6 章），也覆盖了自从 1980 年以后变化的内容。

并且，开发环境也升级了。很多我提到的工具，如 make 和 debugger，你已经很熟悉了，但是我发现别人可能还不知道。Autotools 已经改变了代码发布的方式，Git 也改变了我们合作编程的方式 。

问题：我实在忍不住要问，为什么这本书中有差不多三分之一的篇幅都没有 C 代码?

答案：这本书本意就是讲述一些其他 C 语言教科书没有讲到的内容，排在首位的就是工具和环境。如果你没有使用调试器（独立的或者集成在你的 IDE 中），你就是在自讨苦吃。教科书经常把 debugger 放到最后面，有的根本就不提。与他人共享代码也需要另外的工具集，如 Autotools 和 Git。代码并不存在于真空里，其他的教科书都在假设读者只需要了解 C 语言语法就会有生产力了，那就让我写一点不同于这些教科书的内容吧。

问题：有太多的用于 C 开发的工具，你在本书中如何取舍呢?

相比大部分语言，C 语言社区有更高的内部互通性。GNU 提供了太多的 C 语言扩展，还有那些只工作在 Windows 平台的 IDE，只存在于 LLVM 中的编译器扩展等。这就是为什么过去的教科书不去讲解工具的原因。但是现在，有些系统应用得非常普遍。很多工具来自于 GNU；LLVM 和相关工具虽然不是主流，但是也打下了坚实的基础。不管你用什么，Windows、Linux 或者你从你的云计算提供商那里取得的任何东西，这里我介绍的工具全部是容易并且可以快速地安装的。我提到了一些平台相关的工具，但是仅限那么几例。

我并没有介绍集成开发环境（IDEs），因为很少的集成开发环境能跨平台工作（尝试建立一个 Amazon Elastic Computer Cloud 实例，然后在上面安装 Eclipse 和它的 C 插件），而且 IDEs 的选择大部分被个人的喜好所左右。IDE 有一个项目建造系统，它通常与别的 IDE 的项目建造系统不兼容。IDE 的项目文件在你分发到外面的时候就不能用了。除非你硬性规定所有的人（在教室、特定办公室或者某些计算平台上）都必须使用相同的 IDE。

问题：我能上网，一两秒的功夫就能找到命令和语法的细节。那么说真的，为什么我还要读这本书？

答案：的确。在 Linux 或 Mac 机器上你只要用一个带有 man operator 的命令行就能查到运算符优先级表，那么我为什么还要把它放在这本书里？

我可以和你上同样的 Internet，我甚至花了很多的时间阅读网上的内容。所以我有了一个之前没谈到的、准备现在讲的好主意：当介绍一个新工具的时候，如 gprof 或者 GDB，我给你那些你必须知道的方向，然后你可以去自己习惯的搜索引擎中查找相关问题。这也是其他教科书所没有的（这样的内容还不少呢）。

标准：难以抉择

除非特别地说明，本书的内容遵从 ISO C99 和 C11 标准。为了使你明白这意味着什么，下面给你介绍一点 C 语言的历史背景，让我们回顾一下主要的 C 标准（而忽略一些小的改版和订正）。

K&R（1978 前后）

> Dennis Ritchie、Ken Thompson 以及一些其他的贡献者发明了 C 语言，并编写了 UNIX 操作系统。Brian Kernighan 和 Dennis Ritchie 最终在他们的书中写下了第一版关于这个语言的描述，同时这也是 C 语言的第一个事实上的标准[Kernighan，1978]。

ANSI C89

> 后来 Bell 实验室向美国国家标准协会（ANSI）交出了这个语言的管理权。1989 年，ANSI 出版了他们的标准，并在 *K&R* 的基础上做出了一定的提高。*K&R* 的书籍的第 2 版包含了这个语言的完整规格，也就是说在几万名程序员的桌子上都有这个标准的印刷版[Kernighan，1988]。1990 年，ANSI 标准被 ISO 基本接受，没有做重大的改变，但是人们似乎更喜欢用 ANSI，89 这个词来称呼这个标准（或者用来做很棒的 T 恤衫标语）。

10 年过去了。C 成为了主流，考虑到几乎所有的 PC、每台 Internet 服务器的基础代码或多或少都是用 C 编写的，C 语言已经成为了主流，这已经是人类的努力可以达到的最大的极限了。

在此期间，C++分离出来并大获成功（虽然也不是那么大）。C++是 C 身上发生的最好的事情了。当所有其他的语言都在试图添加一些额外的语法以跟随面向对象的

潮流，或者跟随其作者脑袋里的什么新花招的时候，C 就是恪守标准。需要稳定和可移植性的人使用 C，需要越来越多的人把大量的金钱投入到了 C++语言上，这样的结果就是：你好我好，大家过年，每个人都高兴。

ISO C99

10 年之后，C 标准经历了一次主要的改版。为数值和科学计算增添了一些附加功能，如复数的标准数据类型以及泛型(type-generic)函数。一些从 C++中产生的便利措施被采纳，包括单行注释（实际上起源于 C 的前期语言，BCPL），以及可以在 for 循环的开头声明变量。因为一些新添加的关于如何声明和初始化的语法，以及一些表示法上的便利，使用泛型函数变得更加容易。出于安全考量以及并不是所有人都说英语原因，一些特性也被做了调整。

当你想着单单 C89 的影响其实并不大，以及全球大范围地运行着 C 代码时，你就理解了 ISO 做出的任何改变都是会被广泛批评的——甚至你不做任何改变，别人还想找茬骂你呢[1]。的确，这个标准是有争论的。有两种常用的方式来表达一个复数（直角坐标和极坐标）——那么 ISO 会采用哪一个？既然所有的好代码都没采用变长的宏输入机制来编写，为什么我们还需要这个机制？换句话说，纯洁主义者批评 ISO 是屈服于外界的压力才给 C 语言增加了更多的特性。

当我写这本书的时候，多数的编译器在支持 C99 的同时都增加或减少了一些特性；如 long double 类型看起来就引发了很多问题。然而，这里还是有一个明显的特例：Microsoft 至今拒绝在其 Visual Studio C++编译器中添加 C99 支持。在本书第 6 页“1.2 在 Windows 下编译 C”一节中讲述了几种在 Windows 环境中编译 C 的方法，所以不能使用 Visual Studio 最多也就是有点不方便，这好比一个行业奠基人告诉我们不能使用 ISO 标准的 C，这样标准就更有 punk rock 风格了。

C11

觉察到了对所谓背叛行业趋势的批评后，ISO 组织在第三版的标准中做出了为数不多的几个重大改变。我们有了可以编写泛型函数的方法，并且对安全性和非英语支持做出了进一步的改进。

C11 标准在 2011 年 12 月发布后不久，编译器的开发者以惊人的速度完成了对新标准的支持。目前一些主流编译器已经声称做到了几乎全部的标准兼容。但是标准定义了编译器的行为，也定义了标准库和库支持。如线程和原子性等，

[1] 译者注：世界上就两种语言，没人用的和大家骂的。

有些系统上实现了，有些系统上还在开发。

POSIX 标准

事物的规律就是这样，伴随着 C 语言的进化，这门语言同时也和 UNIX 操作系统协同发展，并且你将会从本书中看到，这种相互协同的发展对日常工作是有意义的。如果某件事情在 UNIX 命令行中很容易利用，那么很有可能是因为这件事情在 C 中也很容易实现；某些 UNIX 工具之所以存在，也是为了帮助 C 代码的编写。

UNIX

> C 和 UNIX 都是在 20 世纪 70 年代由 Bell 实验室设计的。在 20 世纪的多数时间里，Bell 一直面临垄断调查，并且 Bell 有一项与美国联邦政府达成的协议，就是 Bell 将不会把自身的研究扩张到软件领域。所以 UNIX 被免费发放给学者们去研究和重建。UNIX 这个名字是一个商标，原本由 Bell 实验室持有，但随后就像一张棒球卡一样在数家公司之间转卖。

随着其代码被不断研究、重新实现，并被黑客们以不同的方式改进，UNIX 的变体迅速增加。因此带来了一点不兼容的问题，即程序或脚本变得不可移植，于是标准化工作的迫切性很快就变得显而易见。

POSIX

> 这个标准最早由电气和电子工程师协会（IEEE）在 1988 年建立，提供了一个类 UNIX 操作系统的公共基础。它定义的规格中包括 shell 脚本如何工作，像 ls、grep 之类的命令行应该如何工作，以及 C 程序员希望能用到的一些 C 库等。举个例子，命令行用户用来串行运行命令的管道机制被详细地定义了，这意味着 C 语言的 popen（打开管道）函数是 POSIX 标准，而不是 ISO C 标准。POSIX 标准已经被改版很多次了；本书编写的时候是 POSIX:2008 标准，这也是当我谈到 POSIX 标准的时候所指代的。POSIX 标准的操作系统必须通过提供 C99 命令来提供 C 编译器。
>
> 这本书用到 POSIX 标准的时候，我会告诉大家。
>
> 除了来自 Microsoft 的一系列操作系统产品，当前几乎所有你可以列举出的操作系统都是建立在 POSIX 兼容的基础上：Linux、Mac OS X、iOS、WebOS、Solaris、BSD——甚至 Windows Servers 也提供 POSIX 子系统。对于那些例外的操作系统，1.2“在 Windows 下编译 C 程序”一节将告诉你如何安装 POSIX 子系统。

最后，有两个 POSIX 的实现版本因为有较高的流行度和影响力，值得我们注意。

BSD

在 Bell 实验室发布 UNIX 给学者们剖析之后，加州大学伯克利分校的一群好人做了很多明显的改进，最终重写了整个 UNIX 基础代码，产生了伯克利软件发行版（Berkeley Software Distribution，BSD）。如果你正在使用一台 Apple 公司生产的电脑，你实际上在使用一个带有迷人图形界面的 BSD。BSD 在几个方面超越了 POSIX，因此我们还会看到，有一两个函数虽然不属于 POSIX，但是如此有用而不容忽略（其中最重要的救命级函数是 asprintf）。

GNU

GNU 这个缩写代表 GNU's Not UNIX，代表了另一个独立实现和改进 UNIX 环境的成功故事。大多数的 Linux 发行版使用 GNU 工具。有趣的是，你可以在你的 POSIX 机器上使用 GNU 编译器组合（GNU Compiler Collection，gcc）——甚至 BSD 也用它。并且，gcc 对 C 和 POSIX 的几个方面做了一点扩充并成为事实上的标准，当本书中需要使用这些扩充的时候我会加以说明。

从法律意义上说，BSD 授权比 GNU 授权稍微宽容。由于很多群体对这些授权的政治和商业意义深感担心，实际上你会经常发现多数工具同时提供 GNU 和 BSD 版本。例如，GNU 的编译器组合（gcc）和 BSD 的 clang 都可以说是顶级的 C 编译器。来自两个阵营的贡献者紧密跟随对方的工作，所以我们可以认为目前存在的差异将会随着时间逐渐消失。

法律解读

美国法律不再提供版权注册系统：除了很少的特例，只要某人写下什么就自然获得了该内容的版权。

发行某个库必然要通过将其从一个硬盘复制到另一个硬盘这样的操作，而且即便带有一点争议，现实中还是存在几种常用机制允许你有权利复制一个有版权的内容。

- GNU 公共许可证：其允许无限制地复制和使用源代码和可执行文件。不过有一个前提：如果你发行一个基于 GPL 许可证的源代码程序或库，你也必须将你的程序的源代码伴随程序发行。注意，如果你是在非商业环境下使用这样的程序，你不需要发行源代码。像用 gcc 编译你的源代码之类的运行 GPL 许可证的程序本身并不会使你具有发行源代码的义务，因为这个程序的数据（比如你编译出的可执行文件）并不认为是基于或派生于 gcc 的。例如：GNU 科学计算库。

- 次级 GPL 许可证：与 GPL 有很多相似，但是具有一个明显的区别：如果你以共享库的方式连接一个 LGPL 库，你的代码不算作派生的代码，也没有发行源代码的义务。也就是说，你可以采用不暴露源代码的方式发行一个与 LGPL 库连接的程序。例如：Glib。
- BSD 许可证：要求使用者维持 BSD 授权原始码原有的版权声明和免责声明，但不要求同时提供你的原始码。

请注意以下的免责声明：笔者不是律师，这段小常识只是完整法律文件的简单声明，读者如果无法判断自身所处的状况或者相关细节，请阅读原始文件或者请教律师。

附加内容

第 2 版

我以前是一个愤世嫉俗主义者，认为如果你写了第 2 版，那么你的主要目的就是让那些卖你第 1 版二手书的人不开心。本书的第 2 版如果没有第 1 版被发表的话，是不会，也不可能这么快的就出版的。（反正现在很多读者都在阅读电子版本了。）

与第 1 版相比，最大的增加就是并发线程，也就是并行计算部分了。它集中描述了 OpenMP 和原子变量和结构。OpenMP 并不是 C 语言标准，但它是 C 生态系统中非常可靠的一部分，所以它应该在本书的范围内。原子变量是在 2011 年 12 月发布的 C 标准修订版中加入的，一年以后本书第 1 版出版的时候，还没有编译器去支持它。现在好了，我们不仅可以在理论上进行讲解，同时还可以有现实的实现和测试代码了。参考第 12 章。

第 1 版本得到了很多细心读者的反馈。他们发现了很多可能导致 bug 的内容，从一些我在命令行上使用的斜杠，到句子中的一些可能会引起误会的词。这个世界上没有什么东西是没有错的，但是有了读者的反馈，这本书现在更加的正确和有用了。

本版中添加的其他内容：

- 附录 A 对于从其他语言转过来的读者，提供了一个 C 语言的简单的教程。我本来不想在第 1 版中包含这一部分内容，因为有太多的 C 语言教程了，但是包括了这部分内容让本书更加有用了。
- 应大家的要求，我扩展了关于使用调试器的内容。见本书第 32 页“2.1 使用

调试器”。

- 第 1 版有一部分讲述了如何写一个接受变长参数的函数，如 sum(1,2.2)和 sum(1,2.2,3,8,16)都是合法的。但是如果你想传送多个变长列表的时候该怎么办呢？例如点积函数把两个变长向量相乘，dot((2,4),(−1,1)和 dot((2,4,8,16),(−1,1,−1,1)) 10.4“多列表”介绍这一部分内容。
- 我重写了第 11 章，利用新函数来扩展对象。主要添加的部分是虚函数表的实现。
- 关于预处理器，我也多写了一些，主要是在 8.1.2“测试宏”中介绍了一些测试宏的概念和用法。也包括了_Static_assert 关键字。
- 我一直坚守诺言，本书中不包括关于正则表达式解析的内容（因为网上和其他书籍有太多这一部分内容了）。但是我在 13.2.1“解析正则表达式”部分加上了一个演示，展示了如何使用 POSIX 的正则表达式解析函数。比起其他的语言，这些函数还处在比较原始的形态上。
- 第 1 版中关于字符串的讨论主要依赖于 asprintf 函数，它是一个 sprintf 类似的函数，当写一个字符串的时候，会自动分配需要的内存。这是一个被 GNU 广泛分发的版本，但是很多读者却被限制使用，所以在本版本中，我加入了例子 9-3，演示了如何利用 C 语言标准的部分去实现这样一个函数。
- 第 7 章最大的主题是精细地去管理那些会造成麻烦的数值类型，所以第 1 版并没有提到很多在 C99 标准中新加入的数值类型。如 int_least32_t，uint_fast64_t 等[C99，§7.18；C11，§7.20]。很多读者鼓励我至少提一些有用的类型，如 intptr_t 和 intmax_t，好吧，我从善如流。

本书使用的排版约定

本书使用如下排版约定：

斜体(italic)

用来表示新术语、URL、E-mail 地址、文件名、文件扩展名等。

等宽字体（`Constant width`）

用来表示程序列表，同时在段落中引用的程序元素（例如变量、函数名、数据库、数据类型、环境变量、声明和关键字等）也用该格式表示。