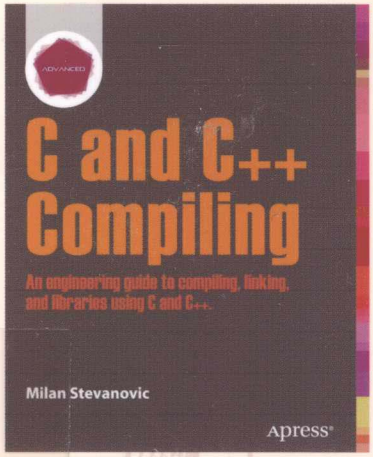
HZ BOOKS
华章IT
Apress®
C/C++技术丛书

高级C/C++编译技术

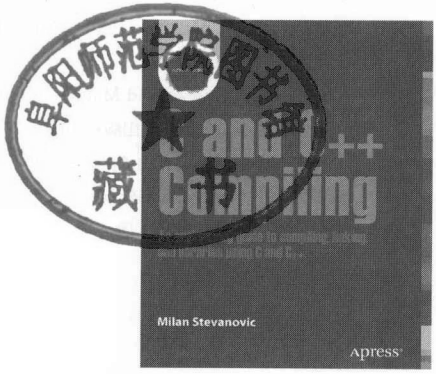


Advanced C and C++ Compiling

[美] 米兰·斯特瓦诺维奇（Milan Stevanovic） 著　卢誉声 译

机械工业出版社
China Machine Press

高级 C/C++ 编译技术



Advanced C and C++ Compiling

[美] 米兰·斯特瓦诺维奇 (Milan Stevanovic) 著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

高级 C/C++ 编译技术 /（美）斯特瓦诺维奇（Stevanovic，M.）著；卢誉声译．—北京：机械工业出版社，2015.3
（C/C++ 技术丛书）
书名原文：Advanced C and C++ Compiling

ISBN 978-7-111-49618-2

I. 高… II. ①斯… ②卢… III. C 语言－程序设计 IV. TP312

中国版本图书馆 CIP 数据核字（2015）第 048520 号

本书版权登记号：图字：01-2014-8120

高级 C/C++ 编译技术

出版发行：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：陈佳媛　　责任校对：殷　虹
印　　刷：北京市荣盛彩色印刷有限公司　　版　　次：2015 年 4 月第 1 版第 1 次印刷
开　　本：186mm×240mm　1/16　　印　　张：17.5
书　　号：ISBN 978-7-111-49618-2　　定　　价：69.00 元

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　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

The Translator's Words 译者序

虽说计算机软件编程领域的技术发展日新月异，但我始终坚信所有程序开发技术演进都万变不离其宗，其本质自始至终从未发生过改变。就编程语言来说，无论抽象到何种层次，使用何种风格，最终都会回归到本质——机器代码。而在我们常用的高级语言中，最贴近机器及操作系统底层的恐怕就是 C 语言了。

从一定程度上来讲，C 语言堪称一门革命性的语言，它完美平衡了语言中机器相关与机器无关的部分，使得我们可以用机器无关的方式来处理程序逻辑，但必要时又可以直接控制底层硬件，C 语言被广泛运用在操作系统开发中正是这一点的绝佳例证。同时，C 语言的核心是非常简单的，一切细节都暴露在程序员面前，不会因为某种语法构造而导致隐藏的性能消耗。这使得 C 语言成为程序员在追求程序效率时的一个绝佳选择。C 语言为了保证核心语言的简单（包括标准库的简单）与高效，没有将很多必要的现代语言特性融入标准中，比如 C 语言自身没有提供任何反射机制。这是一把双刃剑，也就是说，如果想在 C 语言中实现任何动态语言特性，必须使用和系统相关的特性，尤其是在动态库、共享库、插件等概念日益重要的今天，这给 C 程序员带来很多麻烦。虽说现在各种主流操作系统中都很好地支持动态库或共享库，也提供了相应接口，但是不同平台中的概念都会有细微差别，而且在具体技术细节上也会不同，这使得程序员在处理跨平台问题时会遇到很多问题。

C++ 的出现则让这一切变得更加错综复杂。

C++ 是一门多范式语言，在提供了面向对象和泛型编程的同时，为了保证效率，依然坚持不加入任何动态特性，或者提供某些非常有局限性的动态特性（比如 RTTI），为了实现多态则采用了虚函数表这种折中方法。除了 C++ 语法给编译器实现带来的复杂性之外，受到更大挑战的则是 C++ 的链接器和装载器实现。符号名称修饰、全局对象管理、模板支持等各种特性使得链接器和装载器为了支持 C++ 需要付出更多的努力。

同时，C++ 程序员处理动态库或共享库时也同样需要更加细心、耐心。很多时候，我们一不留意就会被各种链接器或者装载器错误提示搅得心如乱麻，而查遍代码也找不出问题所在。编译器优化的不确定因素更加剧了这种情况。另一方面，现在的计算机教育中，包括程序设计语言课程、编译原理课程、操作系统课程，但并没有一门关于链接和装载的课程，链接和装载往往只是在这些课程中一带而过。或许这是由于链接和装载是非常纯粹的技术性话题，并不像其他课程那样需要传授各种理论，但这导致很多人对链接和装载一知半解，根本没有系统认识。

这些因素叠加起来的后果是，许多初级 C/C++ 程序员遇到链接和装载问题时感到束手无策，根本不知问题出在何处。如果说 Windows 平台下的程序员得益于 Visual Studio 这种强大的 IDE，在处理动态链接库时颇为方便，那么对 Linux 程序员来说，这便是一场噩梦。他们需要手动集成各种第三方库，有的是开源的，有的是非开源的，有的只提供了源代码，有的只提供了二进制文件，有的文档详尽，有的文档匮乏……他们永远不知道会遇到什么情况。即便是使用 IDE 的初级程序员，有时也不得不手动解决各种链接问题，但往往由于他们满足于 IDE 的良好封装，而忽略其底层编译链接的细节，因而在处理这些问题时更加盲目。在这种情况下，对链接和装载（其实也包括 C/C++ 语言本身以及编译过程）认识并不深刻的那些程序员在日常工作中很容易在处理这类问题时“触礁”。由于缺乏系统认知，他们需要在一次又一次的失败实践中总结，这会走许多弯路。

由于我本人在大规模分布式实时系统研发方面具有一定的经验，因此许多人在 C/C++ 开发中遇到问题时会咨询我，其中一部分问题出在 C++ 语法上，而更多的问题则出在链接和装载上。本书在这些方面进行了翔实总结和讨论。与纯粹讲解理论与技术细节的书不同，本书一方面阐述基本的理论，另一方面则聚焦于 C/C++ 使用静态库和动态库的一些注意事项，并举例说明如何解决实际的链接与装载问题。此外，本书尽量使用通俗易懂的语言来阐述这些知识，并补充了大量示例，避免让读者纠结枯燥的理论。

相信你会和我一样，在本书中发掘出大量有价值的资料，以便在日后的工作中游刃有余，并在处理 C/C++ 相关问题的时候厘清思路，排除障碍。

在翻译本书的过程中，我不仅查阅了大量国内外的相关资料，还与英文原著作者进行了深入沟通，力求做到专业词汇准确权威，书本内容正确，意译部分既无偏差又不失原著意境。在翻译过程中得到了很多人的帮助，这里一一感谢。感谢我的家人，他们是我学习和前进的动力。感谢鲁昌华教授，他在我的成长道路上给予了很大的支持和鼓励。感谢我在思科系统（中国）研发的同事们，他们在我的学习、工作中给予了很大帮助。感谢我的好友金柳颀，感谢他在我翻译本书过程中与我的通力合作。还要感谢机械工业出版社的陈佳媛编辑

对我的信任。

现在我怀着期盼和忐忑的心情将这本译著呈献给大家，我渴望得到你的认可，更渴望和你成为朋友，如果你有任何问题或建议，请与我联系（samblg@me.com），让我们一起探讨、共同进步。

卢誉声

前　言 *Preface*

我花了相当长的时间才认识到，计算机编程艺术与烹饪艺术之间存在着惊人的相似性。

说到烹饪和编程的比较，我的脑海中首先浮现出来的是烹饪专家和程序员的工作目标非常相似：都是为了满足特定对象的需求。对一个厨师来说，他的服务对象是食客，他需要使用大量的食材，一方面满足人们的温饱与营养需求，另一方面则要让人们享受到美食带来的快乐。而对一个程序员来说，其服务对象是微处理器，程序员使用大量不同的程序来为其提供代码，不仅要让微处理器产生有意义的动作，还需要以最优的形式将代码交付给微处理器。

虽说这种对比看起来有些牵强而且过于简单，但我们会在本书中列出一些更加合适和更具说服力的对比。

介绍烹饪方法的食谱的数量繁多。几乎所有的流行杂志都会开设专栏来介绍形形色色的美食和烹饪方法，无论是快餐式的食谱还是精致复杂的食谱、注重食材的食谱还是注重搭配稀有食材的食谱，你都能够找到。

但如果你希望自己成为烹饪大师，就会发现很难找到可供参考的资料，比如食品行业经营（批量生产、饭店或餐饮企业的经营）、食品生产的供需管理、选料和食材保鲜方面的指南或资料。很显然，这就是业余烹饪爱好者和专业食品企业之间的区别。

这种情况与程序设计其实非常相似。

我们可以轻松地从成千上万的书籍、杂志、文章、网络论坛和博客中搜集到编程语言方面的各类信息，无论是入门教程，还是“谷歌编程面试指南”这样的技巧。

但是，这类主题所涵盖的内容只能满足成为专业软件工程师一半的要求。我们不能一直沉浸在因程序实际执行（且执行正确）而带来的喜悦当中，而需要着重考虑接下来的问题：如何组织代码结构以便将来修改、如何从功能模块中提取出可重用代码，以及如何让程序能够适应不同的运行环境（无论是不同的人类语言和字符，还是在不同的操作系统环境中

运行)。

相较于其他编程主题来说，人们很少讨论这类问题。时至今日，这类问题变成了只有计算机科学专业人士（绝大多数软件架构师和构建工程师）和大学课堂上讲解编译器、链接器设计时，才会了解的“黑科技”。

由于 Linux 市场份额增加，而且越来越多的人都将 Linux 作为其编程环境，这促使开发人员开始关注 Linux 编程的相关问题。与在一些封装良好的平台（在 Windows 和 Mac 平台上利用 IDE 和 SDK 将程序员从一些特定的编程细节问题中解放出来）开发软件的开发人员不同，Linux 开发人员在日常工作中需要将来自不同项目且编码风格迥异的代码组合起来，这需要开发人员充分理解编译器、链接器的内部工作机制和程序装载机制，以及不同库的设计细节和使用方法。

本书将许多零碎的知识点进行汇总，并讨论其中那些有价值的内容，这些内容则通过一系列精心设计的简单示例进行验证。需要注意的是，本书的作者并非计算机科学科班出身。20 世纪 90 年代末至今的数字革命中，作者作为电气工程师供职于硅谷的一家多媒体行业高新技术企业，并因此掌握了相关领域的知识。希望本书的主题和内容能够让更多读者受益。

读者对象

作为一名软件设计实践顾问（虽然很忙，但我还是非常自豪的），我经常会与不同专业背景和资历的人群沟通。我在工作周中需要经常在不同的办公环境中工作，因此接触了许多开发人员（绝大多数来自硅谷），这也让我更加了解了本书的受众群体。其中包括以下几类人群。

- 第一类受众群体是来自不同工程领域的 C/C++ 开发人员（电气工程、机械、机器人技术和系统控制、航天、物理和化学等领域），这类人需要在日常工作中通过编程来解决问题。对缺乏正规计算机科学课程和理论教学的人来说，本书所提供的资料弥足珍贵。
- 第二类受众群体是具有计算机科学教育背景的初级程序员。本书能够帮助大家将主修课程中学到的知识具体化，并注重实践。将第 12 章 ~ 第 14 章的内容作为手册查阅，对资深工程师而言，也会有所受益。
- 第三类受众群体是操作系统集成和定制的爱好者。理解二进制文件及其内部工作机制将有助于在解决问题的过程中扫除障碍。

关于本书

我最初并没有计划去写这么一本书，甚至都没有打算写一本计算机科学领域的书。(我有

可能会去编写信号处理或程序设计艺术方面的书，但编写计算机科学方面的书？不……）

在职业生涯中，我经常处理当时我认为别人已经解决好的一些问题，而实际上这些问题并没有得到根本性解决，这是我写作此书的唯一原因。

很久以前，我决定成为一名从事高科技领域的“刺客”，将许多看似平静且体面的高科技公司从“恐怖分子”——复杂多媒体设计问题和大量严重缺陷所造成的破坏中解救出来。选择这样一个职业的结果就是，我并没有多少时间处理自己的生活，比如说孩子们想吃鸡肉而不是豌豆，我很难满足他们的这些需求。虽然我更倾向于使用傅里叶变换算法、小波、Z变换、FIR 和 IIR 滤波器、倍频程、半音程、插值和抽取算法来解决问题（与 C/C++ 编程一起使用），但我还是要解决那些我并不喜欢解决的问题。总要有人去做这些事情吧。

出乎意料的是，在搜索一些非常简单明了的问题的答案时，我只能找到一些散乱的网络文章，而且绝大多数都只是泛泛而谈。我很耐心地把这些散乱的内容组织到一起，不仅完成了我手头的设计任务，而且学习总结了很多资料。

在一个天朗气清的日子里，我开始整理设计笔记（记录我工作中经常遇到的一些问题和解决方案）。但在整理工作完成的时候，这些笔记看起来就像……嗯……就像一本书——就是这本书。

不管怎么说……

就目前就业市场的情况而言，我认为（自 2005 年左右开始）熟悉 C/C++ 语言的复杂性，甚至是算法、数据结构和设计模式，对于找到一份好工作都是远远不够的。

在开源盛行的今天，专业开发人员在日常工作中所编写的代码越来越少，取而代之的是将现有代码集成到项目中。这不仅要求开发人员能够读懂其他人编写的代码（使用不同的代码风格和实践），还需要了解如何才能以最好的方式将现有的包（绝大多数以二进制文件（库）和导出头文件的形式提供）集成到代码中。

我希望本书能够兼具教学（对亟须这些知识的读者而言）和快速查询的功能（对分析 C/C++ 二进制文件相关工作的工程师而言）。

为何采用 Linux 进行演示？

选择 Linux 并非我个人的偏好。实际上了解我的人都知道，我过去是多么喜欢使用 Windows 作为开发环境（原本这是我首选的设计平台），原因是 Windows 平台具有编写良好的文档、完善的支持和符合规范的认证组件。我设计过许多专业化软件（曾为 Palm 公司设计开发了 Windows Mobile 平台的 GraphEdit，其中包含了许多最为复杂的功能，随后又开发了多个媒体格式和 DSP 分析软件），在当时我对 Windows 的技术了如指掌，并感叹 Windows 相

关技术所带来的改变。

与此同时，Linux 的时代到来了。有关 Linux 的技术随处可见，而对开发人员来说，也必须顺应这种趋势去学习和使用它。

Linux 软件开发环境具有开放、透明和简单明了的特点。在 Linux 中，我们可以对每个程序设计阶段进行控制。同时，Linux 提供了完善的文档，再加上网络上提供的资源，就可以轻松地使用 GNU 工具链。

实际上，由于 Linux C/C++ 开发经验可以直接适用于 Mac OS 平台的底层开发，因此我最终决定选用 Linux/GNU 作为本书所涵盖的主要开发环境。

别急！Linux 与 GNU 完全是两回事！

实际上，Linux 是内核，而 GNU 中包含了 Linux 内核之上的所有软件。除了 GNU 编译器（可以在其他操作系统上使用，比如 Windows 上的 MinGW）以外，在绝大多数情况下，GNU 与 Linux 的关系其实非常紧密。为了简单起见，同时为了符合一般开发人员对开发场景的认识，特别是为了将 Linux 与 Windows 进行对比，本书将 GNU 与 Linux 作为一个整体，简称为“Linux”。

章节概览

第 2 章 ~ 第 5 章讲解的内容主要为后续内容做铺垫。拥有计算机科学背景的读者可以快速阅读这些章节（幸运的是，这些章节的内容并不长）。实际上，任何计算机科学方面的教科书都会对这些内容进行类似介绍，而且内容会更为详细。我个人推荐由 Bryant 和 O’Hallaron 编写的《深入理解计算机系统》[⊖]（《Computer Systems—A Programmer’s Perspective》）一书，原因是本书对很多主题都进行了非常有条理的梳理和总结。

第 6 章 ~ 第 11 章是本书的核心章节。为求整体内容简洁明了，我花费了相当大的精力，并尝试使用一些日常生活中常见事物的文字和图片，来阐述那些最为重要的核心概念。如果你不是计算机科学科班出身，那么有必要先理解这些内容。其实这些章节是本书主题的要点。

第 12 章 ~ 第 14 章主要概括了一些实践方面的内容，便于读者快速查找相关的概念。这些章节针对一些特定平台的二进制文件分析工具进行了总结，然后在实践部分涵盖了完成独立任务的方法。

⊖ 本书中文版由机械工业出版社引进并出版，ISBN：978-7-111-32133-0。

目　录 Contents

第 1 章 Chapter 1

多任务操作系统基础

构建可执行文件的精髓在于实现对程序执行过程的最大化控制。为了能够真正理解可执行文件结构中各部分的功能和含义，我们有必要对程序执行过程中的操作进行详细了解，操作系统内核和可执行文件中信息的相互作用在这里扮演了重要的角色。特别是在程序启动的最初阶段，在运行时环境数据（如用户配置和各种运行时事件等）对程序产生影响以前，这种相互作用通常会非常频繁。

为了能够真正理解可执行文件结构中各部分的功能和含义，首先要理解与程序运行相关的一些技术细节。本章将为读者着重阐述现代多任务操作系统的功能。

现代多任务操作系统在重要功能性方面的实现方法非常相似。因此，本章会把主要精力放在阐述与平台无关的一些概念上。除此之外，我们还将着重研究和分析一些特定平台的复杂性（将无所不在的 Linux 和 ELF 格式与 Windows 对比）。

1.1 一些有用的抽象概念

计算机技术领域的变化日新月异。集成电路技术带来的元件不仅种类繁多（光学元件、电磁元件和半导体元件）而且在功能性方面不断改进。按照摩尔定律，集成电路上可容纳的晶体管数目大约每隔两年便会增加一倍。而与晶体管数量密切相关的处理能力也将提升一倍。

经验告诉我们，若想要完全应对这种快速变化，唯一的方法就是在经常变动的实现层次之上，利用抽象和泛化的方法为计算机系统定义全局目标与体系结构。这种方法的核心在于描述抽象的方式，该方式要确保在去除相对无关的实现细节后，任何新的实现与核心定义都能够保持一致。整个计算机体系结构可以用图 1-1 展示的一组结构化抽象表示。

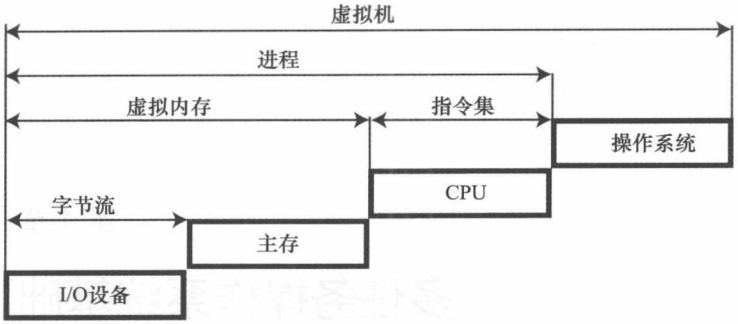


图 1-1 计算机体系结构抽象

在抽象的最底层，我们利用典型的字节流方式来处理各种类型的 I/O 设备（鼠标、键盘、控制杆、轨迹球、光笔、扫描仪、条形码扫描器、打印机、绘图仪、数码相机和网络摄像头）的输入和输出。事实上，在忽略不同设备用途、实现方法和功能性的情况下，对计算机系统设计而言，我们只需关注这些设备产生或接收（或两者皆有）的字节流。

下一个抽象层次是虚拟内存，它用来表示系统中多种不同的存储资源。虚拟内存是本书将要讨论的重要主题之一。实际上这种特定抽象的方式代表了多种不同的物理存储设备，这不仅影响了实际的软硬件的设计，而且是设计编译器、链接器和装载器的基础。

下一个抽象层次是指令集，它用来对物理 CPU 进行抽象。虽然高级程序员一定对指令集所具备的功能及其带来的处理性能十分感兴趣，但就本书所希望讨论的主题来看，这部分抽象的细节不是本书详细讨论的重点。

最后一个抽象层次是操作系统的复杂性。通常来说，操作系统在某些方面的设计（主要是多任务机制）对软件的体系结构有着决定性的影响。对多方都需要访问的共享资源来说，需要有完善的实现方法才可以避免重复的无用代码问题，而这个问题直接为我们引出了共享库设计的概念。

在接着分析整个计算机系统的复杂性之前，我们先了解一下与存储器使用相关的一些关键问题。

1.2 存储器层次结构与缓存策略

在计算机系统中，有一些和存储器相关的趣事：

- 人们对存储器容量的需求总是无法满足，而且存储器容量总是供不应求。每当存储器技术在容量（更快的存储器的容量）方面取得重大飞跃的时候，就会发现其实有一些技术已经为此等候多时。这些技术在理论上完全可行，只是需要等到拥有足够数量的物理存储器供其使用时才能够被真正实现出来。
- 存储器技术似乎是导致处理器性能障碍的主要原因。这被称为“处理器与存储器之间

的速度鸿沟”(the processor-memory gap)。

- 存储器的访问速度与其存储容量成反比。通常来说，容量最大的存储设备的访问时间要比容量最小的存储设备慢上好几个数量级。

现在，我们从程序员、设计师和工程师的视角来简单地看一下整个系统。在理想情况下，我们希望系统能够以最快的速度访问所有可用的存储器，但其实我们都知道，这是不可能实现的。紧接着的一个问题就是：我们能否做些什么来改善这种情况呢?

一个细节让我们大可放心，那就是在实际情况下，系统并不总是使用所有的存储器，而仅仅是在某些时段内使用某一部分存储器。在这种情况下，我们只需为立即需要执行的程序预留最快的存储器，而让那些并非立即执行的代码或数据使用较慢的存储设备。当 CPU 从最快的存储器中获取计划立即需要执行的指令时，硬件设备会预测接下来会执行程序的哪一部分，并将这部分代码交给较慢的存储器，然后等待执行。在执行到存储在较慢存储器上的代码之前，这些代码会转存到较快的存储器中。这种策略称为“缓存”。

我们可以把缓存比作现实生活中一般家庭的食物供给。除非是生活在荒无人烟的地方，否则我们一般不会采购一整年所需的食物带回家。相反，我们通常会在家中安置一些比较大的存储设备（比如冰箱、食品储藏室和食品架），用来存放未来一到两周的食物。当意识到食物快吃完的时候，我们就会去食品杂货店采购恰好能够填满家中存储设备的食物。

程序的执行通常会受到一些外部因素的影响（用户设置只是其中之一），这让缓存机制的实现变得十分困难。程序执行流程（通过跳转和中断等指令的数量来衡量）的可确定性越高，缓存机制工作得越好。相反，若程序的执行流程发生变化，那么之前缓存的指令也就不再有效，因此这部分缓存的指令会被丢弃，而程序所需的那部分指令则需要重新从较慢的存储器中获取。

缓存策略的实现无处不在，横跨多个级别的存储器，如图 1-2 所示。

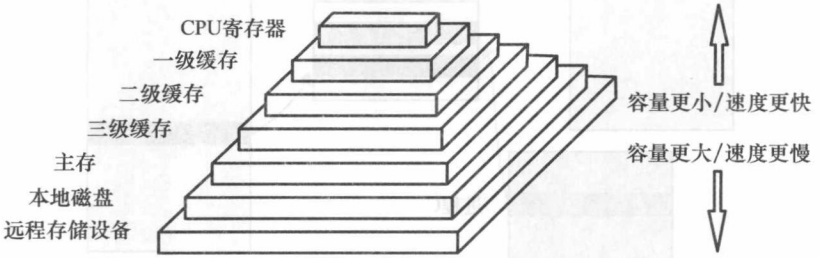


图 1-2　存储器的缓存层次结构原理图

1.3　虚拟内存

存储器缓存的通用方法在下一个体系结构层次中得到了具体实现，我们在这一架构层中用名为“进程”的抽象概念来表示正在运行的程序。