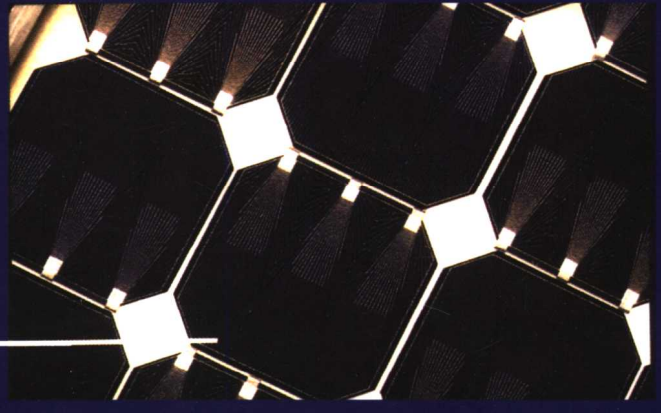
PEARSON
Addison Wesley

提高C++性能的编程技术

Efficient C++

Performance Programming Techniques

〔美〕Dov Bulka
David Mayhew 著
常晓波
朱剑平 译



Pearson Education

清华大学出版社

提高 C ++ 性能的编程技术

Efficient C ++ Performance Programming Techniques

[美] Dov Bulka David Mayhew 著

常晓波 朱剑平 译

清华大学出版社

北 京

Efficient C++ Performance Programming Techniques
Dov Bulka David Mayhew
EISBN:0-201-37950-3

图书在版编目(CIP)数据

提高 C++ 性能的编程技术/(美)布尔卡，(美)梅休著；常晓波等译. —北京：清华大学出版社，2003
书名原文：Efficient C++ Performance Programming Techniques
ISBN 7-302-06550-0

Ⅰ. 提… Ⅱ. ①布… ②梅… ③常… Ⅲ. C 语言—程序设计 Ⅳ. TP312

中国版本图书馆 CIP 数据核字(2003)第 027069 号

出 版 者：清华大学出版社(北京清华大学学研大厦，邮编 100084)
http://www.tup.com.cn
责任编辑：赵彤伟
版式设计：刘祎淼
印 刷 者：世界知识印刷厂
发 行 者：新华书店总店北京发行所
开 本：787×960 1/16 **印张**：16 **字数**：319 千字
版 次：2003 年 6 月第 1 版 2003 年 6 月第 1 次印刷
书 号：ISBN 7-302-06550-0/TP・4907
印 数：0001～4000
定 价：33.00 元

译者序

提高 C++ 性能的编程技术

性能问题始终是困扰 C++ 程序员的难题，人们大都认为 C++ 的性能不如 C，但事实并非如此。本书以详尽的实例讲解了通过临时对象、内存管理、模板、继承、虚函数、内联、引用计数、STL 等提高 C++ 性能的编程技术，使我们能够以一个全新的高度看待 C++ 这个我们自以为熟悉的编程工具。

参加本书翻译的主要人员有：常晓波和朱剑平等。感谢王欣轩程序员和刘颖先生对译稿提出的宝贵意见。在此，谨向所有为本书出版付出辛勤劳动的人员致以诚挚的感谢！

在与清华大学出版社计算机引进版图书编辑室合作的过程中，出版社的朋友们一丝不苟的工作态度使我们受益颇多。在此致以诚挚的谢意！

译　者

2001 年 12 月

序

提高 C++ 性能的编程技术

如果就 C++ 的性能问题对软件开发人员进行一次非正式调查，肯定会发现他们中的绝大多数人把性能问题视为 C++ 语言的惟一致命缺点。自从 C++ 问世以来，我们经常听到这样的话：开发性能要求苛刻的程序不应选择 C++。在开发人员的观念中，实现这一类的程序应使用标准 C，有时甚至应当使用汇编语言。

我们作为软件团体的一部分，目睹了这种神奇语言的发展和流行。几年前，我们满怀热情地加入了推崇 C++ 的浪潮中，我们身边的许多开发项目都投入其中。过了一段时间以后，用 C++ 实现的软件解决方案开始发生变动。这是因为它们的性能明显不理想，所以就逐渐放弃了 C++。在要求高性能的领域，对 C++ 的热情降了下来。当时我们正在提供网络软件，这些软件的运行速度是不容商量的，要优先考虑。这是因为在软件的关系链中，网络软件位于相当低的层次，所以其性能是至关重要的。大量的程序位于其上并且依赖于它。低层程序性能不佳会导致所有通往高层程序的道路变得不畅。

这样的经历并不少见。在我们周围，许多 C++ 的早期采用者都难以用 C++ 代码取得希望的性能。我们没有把困难归咎于在学习面向对象开发模式时所遇到的大幅波动，而是归咎于 C++ 这种体现该模式的主流语言。尽管 C++ 编译器还处于初期发展阶段，但 C++ 语言已经被烙上了天生就慢的烙印。这种信念迅速传播并被广泛当成事实加以接受。摒弃 C++ 的软件组织经常指出性能是他们的主要考虑。这种考虑的根源是认为 C++ 无法达到与之相对的 C 语言所能达到的性能。因此 C++ 很少在敏感性软件领域取得成功，因为这些软件领域把性能看得甚为重要，如操作系统内核、设备驱动程序、网络系统（路由器、网关、协议栈）和其他更多方面。

我们花费了多年时间解剖用 C 和 C++ 代码写成的大型系统，以便使性能达到最高。在整个过程中我们努力寻找用 C++ 产生高效程序的潜力。我们发现事

实上确实如此。在本书中，我们希望与您分享这次经历，并记录了我们在对 C++ 的效率进行研究的过程中所学到的经验教训。编写高效的 C++ 代码既不是轻而易举的，也不像火箭科学那么困难。它需要对一些性能原理的理解，以及一些有关性能缺陷的知识。

80-20 规则是软件结构领域的一个重要原则。在本书的写作中我们同样采用它：所有性能漏洞的 20%会占用 80%的时间。因此我们将选择把精力集中在最有价值的地方。我们的主要兴趣在于那些经常出现在工业化代码中并有显著影响的性能问题。本书不对性能漏洞及其解决方法进行详尽的讨论，因此，将不涉及我们认为深奥而不常见的性能缺陷。

毫无疑问，作为开发服务器端、对性能要求苛刻的通信程序的程序员，我们的实践经验是我们的观点存在的基础。由此形成的观念会在以下方面影响本书：

- 我们在实践中所遇到性能问题的情况，与在科学计算、数据库程序和其他领域中所遇到问题的本质稍有不同。这不是什么问题。通用性能原则是不分领域的，同时，在网络软件以外的领域也同样适用。
- 尽管我们竭力避免，但有时还是会虚构一些例子来说明某个观点。以前我们有过很多编码错误，因此有相当多的范例，它们来自于我们所编写的真实的产品代码。我们的经验来之不易——是从我们自己及同事所犯的错误中学来的。我们将尽可能使用实际范例来说明自己的观点。
- 对于算法渐近复杂性、数据结构以及用于访问、排序、搜索及压缩数据的最新和最优技术，我们将不进行深入研究。这些虽都是重要的主题，但是别处已给出大量介绍[Knu73、BR95、KP74]。我们将着重于那些简单、实用的日常编码和设计原则，它们会大幅提高性能。我们要指出降低性能的常见设计和编码习惯，不管是由于不知情地使用了会导致隐含高消耗的语言特性，还是由于违反了任何一些敏感(或是不太敏感)的性能原理。

如何分清理想与现实呢？C++ 的性能真的比不上 C 吗？我们认为通常所持的 C++ 性能差的观点是错误的。我们承认在一般情况下，如果把 C 和看起来与 C 相同的 C++ 版本相比，C 程序通常要快一些。但同时我们认为两种语言在外观上的相似性通常是基于它们的数据处理功能，而不是它们的正确性、健壮性和易维护性。我们认为如果让 C 程序达到 C++ 程序的水平，则速度差别就会消失，甚至可能是 C++ 版本的程序更快。

C++ 不是天生就较慢或较快，两者都是有可能的，这要看怎样使用它以及需要它做什么。这与其使用方式有关系：如果运用合理，则 C++ 不仅可以让软件系统具备可接受的性能，而且可以获得出众的性能。

我们在此向许多对本书做出贡献的人表示感谢。万事开头难，正是我们的编辑 Marina Lang对本书的启动给予了帮助。Julia Sime 为早期的文稿做出了显著的贡献，

Yomtov Meged 也向我们提供了很多有价值的建议。Yomtov Meged 还指出了我们的观点与真实情况之间的细微差别。尽管有时它们会一致，但我们还是要加以区分。

十分感谢 Addison-Wesley 聘请的两位评论家，他们的反馈信息非常有价值。

同时感谢对原稿进行了检查的朋友和同事，他们是（排名不分先后）：Cyndy Ross、Art Francis、Scott Snyder、Tricia York、Michael Fraenkel、Carol Jones、Heather Kreger、Kathryn Britton、Ruth Willenborg、David Wisler、Bala Rajaraman、Don "Spike" Washburn 和 Nils Brubaker。

最后，还要感谢我们各自的妻子，她们是 Cynthia Powers Bulka 和 Ruth Washington Mayhew。

Dov Bulka
David Mayhew

引 言

提高 C++ 性能的编程技术

在使用汇编语言的年代，有经验的程序员通过计算汇编语言指令条数来估计源代码的执行速度。在诸如 RISC 之类的一些体系结构中，大多数汇编指令是每个时钟周期执行一条。其他体系结构的指令与指令之间在执行速度上表现出很大的差异，不过有经验的程序员对平均执行时间有良好的意识。只要知道代码段包含多少条指令，就可以正确估计代码段的执行要消耗多少时钟周期。源代码与汇编程序之间的对应关系通常是一对一的。汇编代码就是源代码。

在编程语言所形成的阶梯中，C 语言比汇编语言高出一层。C 源代码不能完全对应编译器所生成的汇编代码。编译器负责沟通源代码与汇编程序之间的关系。源代码与汇编代码之间的对应不再是一对一关系，不过仍然保持了一种线性关系：C 中的每一条源代码语句对应很少的几条汇编指令。如果估计每一条 C 语句要转换成 5～8 条汇编指令，那么就可以到球场去放松一下了。

C++ 打破了源代码语句数量与编译器所产生汇编语句数量之间的线性关系。与 C 语言开销相当均匀相反，C++ 语句的开销波动很大。某条 C++ 语句可能会产生 3 条汇编指令，而另一条则可能产生 300 条汇编指令。实现高性能的 C++ 代码向程序员提出了一个意料之外的要求：他们需要跨越性能的“雷区”，力争停留在每条语句 3 条指令的安全道路上，同时要避免使用那些埋有“每条语句对应 300 条指令”之“雷”的“雷道”。程序员必须能够识别可能产生巨大开销的语言构造并知道如何围绕它们进行编码和设计。这些是 C 和汇编语言程序员从来不用担心的。C 语言的惟一例外可能是宏的使用，但这远不及在 C++ 代码中对构造函数和析构函数的引用那么频繁。

C++ 编译器还可能在背后向程序的执行流程中插入代码。这对正转向使用 C++ 而没有思想准备的 C 程序员（我们中的很多人是这样过来的）来说是没有料到的。为了编写高效的 C++ 程序，C++ 开发者需要掌握有关提高 C++ 性能的技

巧，并且要超越通常的软件性能原理。在C编程中，一般不会遇到隐性的性能开销，因此意外获得良好性能是可能的。相反，这种情况在C++中很难发生。如果不了解潜在缺陷，则不可能偶然获得良好性能。

公正地说，我们看到过许多性能不佳的示例，它们源于低效的面向对象(OO)设计。自从面向对象的设计成为主流以来，软件灵活性和重用性思想被大肆提出。然而，灵活性和重用性很少会与性能和效率携手并进。在数学中，把定理还原成基本原理是令人头痛的。数学家力争使用已证明是真理的结果。然而在数学之外的领域，在特殊情况与采取捷径之间进行平衡常常是有意义的。软件设计中，在有些情况下把性能放在比重用更优先的位置来加以考虑是可以接受的。在实现设备驱动程序的read()和write()函数时，与某些地方将来可以重用相比，已知的性能要求对于软件的成功通常是更为重要的。面向对象设计中的一些性能问题是由于在错误的时间对错误的地方进行了强调而造成的。程序员应该致力于解决所面临的问题，而不是让当前解决方案服从于一些将来可能存在的未经确定的需求。

软件低效的根源

静态C++的开销不是所有性能问题的根源。即使是除去编译器产生的开销也不足以说明问题。如果真是这样，那么C程序将因为没有静态开销而自动获得令人吃惊的性能。但事实上通常是一些其他因素影响着软件的性能，特别是C++的性能。这些因素是什么呢？图0.1给出了软件分类中的每一个层次。

在最高层次中，软件的效率取决于两个主要因素的效率：

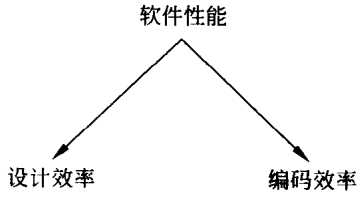


图0.1 软件性能的高层分类

- **设计效率** 这与程序的高层设计有关。解决这一层次的性能问题需要理解程序的大体构造。在相当大的范围内，这是与语言无关的。任何编码效率都无法掩盖低劣的设计。
- **编码效率** 中、小型的实现问题属于这种情况。解决这一类性能问题通常需做局部修改。例如，要把常量表达式放在循环的外面以防止多余计算，这并不需要查看太多的代码段。需要理解的代码段仅限于循环体范围之内。

还可以把这个高层次划分进一步分为更细的子题，如图0.2所示。

上述设计效率可进一步分为两项：

- **算法和数据结构** 从技术上来讲，每一个程序本身就是一种算法。所谓“算法和数据结构”，实际上是指存取、搜索、排序、压缩和对大型数据集合进行其他方式管理

所用算法的众所周知的子集。性能通常自动地与程序中所使用的算法和数据结构相关联，似乎没有其他什么影响因素。但应指出，认为软件性能只会从这一方面降低的观念是错误的。算法和数据结构的高效是必要的，但不是充分的，其本身还不足以保证整个程序拥有出色的性能。

图 0.2　设计性能概念的进一步划分

- **程序分解**　这包括把全部任务分解成相关的子任务、对象层次、函数、数据和函数流程。这是程序的高层设计，包括组件设计和组件与组件间的通信。仅包含一个组件的程序是很少见的。典型的 Web 应用程序至少要和 Web 服务器（通过 API）、TCP 套接字及数据库交互。在涉及跨越这些组件间的每一 API 层时，就会存在效率陷阱。

可以对编码效率进一步划分，如图 0.3 所示。

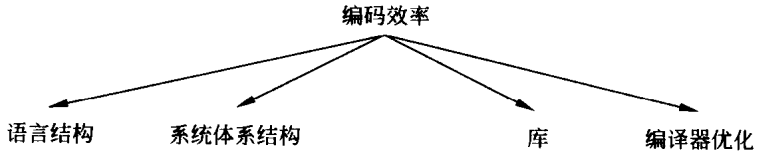


图 0.3　编码性能概念的进一步划分

我们把编码效率划分为 4 项：

- **语言结构**　C ++ 在其前辈 C 中增加了功能和灵活性。这些新增优点不是无偿的——某些C ++ 语言构造可能会产生交换开销。我们将在全书中讨论这个问题。自然，这个主题是 C ++ 所特有的。
- **系统体系结构**　系统设计者付出大量努力为程序员提供理想的系统状况：无限的内存、专用 CPU、并行线程执行和开销均衡的内存访问。当然，所有这些都不是真实的，只是感觉起来像是如此。开发软件时不考虑系统体系结构是方便的。然而，要获得出色的性能，就不能忽视这些体系结构项，因为它们对性能有极大的影响。在提到性能时，我们必须清楚以下几点：

  ◇ 内存不是无限的。内存之所以看起来是无限的，是因为使用了虚拟内存

系统。

◇ 内存访问开销是不均衡的。在缓存、主存和磁盘访问之间存在着数量级上的差别。

◇ 我们的程序没有专用 CPU，在一段时间内只能得到一个时间片。

◇ 在一台单处理器的计算机上，并行线程并不是真正地并行执行，它们是轮流执行的。

- **库** 选择实现所用的库也会影响性能。对于初学者来说，相对其他库，某些库可能会更快地完成某些任务。由于通常情况下不会访问库的源代码，所以很难说出库调用是如何实现其服务的。例如，为了把一个整型值转换成字符串，可以选择：

```
sprintf (string, ;"%d", i) ;
```

或者选择整型转换到 ASCII 的函数调用[KR88]：

```
itoa ( i, string ) ;
```

哪个更有效？差别明显吗？

即使库中已存在某种特定的服务，仍然可以选择编写自己的版本。设计库时经常运用灵活性和可重用性思想。通常，灵活性和可重用性与性能之间存在一种平衡。对于某些要求严格的代码段，如果把性能考虑放在其他两者之上，那么使用自己的实现来覆盖库提供的服务可能是合乎情理的。因为应用程序在各自特定的需求方面差别甚大，所以设计出对任何人、在任何地点和时间都是完美实现的库是很难的。

- **编译器优化** 这不过是一个比"杂项"更具说明性的名字，它包括所有那些不适合分在其他编码分类中的小型编码陷阱，如循环展开、把常量表达式放到循环之外，以及其他类似的消除多余计算的技术。大多数编译器会完成许多这样的优化。但不要指望任何特定的编译器完成特定优化。有的编译器会把循环展开两次，而有的编译器则会把循环展开 4 次，还有一些编译器根本就不展开循环。为了最终控制权，必须把编码问题掌握在自己手中。

我们的目标

许多书籍和文章将 C++ 作为支持面向对象设计模式的语言而对其优点加以赞美。人们把 C++ 看成是解决软件危机的最新方案。对于那些对此不熟悉的人来说，软件危机就是指在开发代码方面的无能为力，这些代码本应足够简单，以便于理解和维护，以便于最关键的扩展；同时还应足够强大，能够对复杂问题提供解决方案[CE95]。从其他结构化语言转移到 C++ 的开发者常常会被这样的思想冲昏了头脑：使用 C++ 创建的高度灵活的和可重用的代码是十分易于维护和扩展的。然而，却忽视了一个重要的问题，即运行

效率。我们将通过 C ++ 编程的前景分析相关的性能主题。读完本书之后，您会对常见的 C ++ 性能缺陷形成一种清楚的认识，同时也将明确地知道在不影响设计的清晰性和简单性的基础上避免这些缺陷的方法。实际上，高性能的解决方案常常也是最简单的解决方案。本书将帮助开发者在受益于 C ++ 扩展功能及面向对象设计固有优越性的同时，开发出像其 C 副本一样高效的 C ++ 代码。一位著名的物理学家曾经说过，所谓专家是指那些把所有可能发生的错误限制在很小范围内的人。尽管犯错误是一种不错的学习途径，但是从其他人（在此就是作者）所犯的错误中进行学习则更好。

本书的第二个目标是汇总有关 C ++ 的性能问题。作为 C ++ 开发者，有关性能问题的答案是不易得到的。它们分散于长篇的书籍和杂志文章清单中，这些杂志和文章引出了该问题的不同片段。您必须自己搜索这些主题并把它们汇总起来。没有多少开发者愿意做这件事，他们太忙了。如果能够对完全着眼于有关 C ++ 性能的重要主题进行一次汇总，那将是有用的。

软件效率：是否很重要

在一个处理器速度每 18 个月就提高一倍（摩尔定律）的时代，我们真的有必要担心软件的效率吗？事实是，不管芯片技术如何迅猛发展，软件效率依然是要突出考虑的问题。1971 年，Intel 4004 是首先集成在单块芯片上的商用处理器。人们称它为微处理器。从此，微处理器便开始了每 18 个月速度翻一番的飞速发展。今天的微处理器比 Intel 4004 快万倍。如果处理器速度是软件低效的原因所在，那么这个问题应该已得到解决并早被忘掉了。然而，软件效率仍然是多数开发组织所考虑的问题。这是为什么呢？

设想您正努力把产品（比如 Web 应用程序服务器）卖给一家排名位于世界财富 500 强的企业。他们需要每秒 600 笔交易的处理能力来运行其在线业务。在一台最新的、功能最强大的服务器硬件上，在快要用尽资源的情况下，您的应用程序只能支持每秒 60 笔业务。如果客户打算使用您的软件，那么为了达到每秒 600 笔业务的目标，他们至少需要用 10 台服务器组成服务器群集。于是您所提供的解决方案就从硬件、软件许可证、网络管理和维护等方面增加了成本。更糟糕的是，客户邀请了两家您的竞争者用他们的解决方案来投标。如果竞争者有更为高效的实现方法，那么他们将需要较少的硬件来满足所要求的性能，这样他们会提供更为便宜的解决方案。在这种情况下，处理器的速度是不变的，本例中的软件供应商是在相同的硬件条件下进行竞争的。通常的结果是最高效的解决方案将会竞标成功。

同时必须调查处理速度与通信速度之间的对比。如果传输数据比计算机可生成的数据要多，那么计算机（处理器和软件）会成为新的瓶颈。物理上的限制很快会制止处理器梦幻般的发展[Lew 1]，而通信速度却不然。与处理速度一样，通信速度也在飞速发展。

在1970年时，每秒4 800比特的速度已经被认为是高速通信了。今天，每秒几百兆比特已成为常事。通信速度的发展之路是无法看到尽头的[Lew2]。

光纤通信技术似乎不会在近期成为对发展构成威胁的关键性技术障碍。一些实验室正在进行每秒100GB的全光纤网络方面的实验。当前最大的障碍不是技术，而是基础设施。高速网络使得信息世界重新布线成为必要，需要从铜缆转变成光纤。这场战役已经打响。通信适配器的速度已经超过了连接它们的计算机设备。正在出现的诸如100 Mbps LAN适配器和高速ATM路由器使计算机速度面临严重危机。过去，低效的软件被链路的缓慢所掩盖。诸如SNA和TCP/IP之类的流行通信协议非常容易拥塞一块令牌环适配器，从而无法检查出软件的性能瓶颈。而100 Mbps FDDI和Fast Ethernet却不会被拥塞。如果向协议的发送/接收路径中添加1 000条指令，它们可能不会降低令牌环连接的吞吐量，这是因为协议实现输送数据的速度仍然要快于令牌环连接的数据消化的速度。但是附加的1 000条指令会立即在Fast Ethernet适配器上显示出吞吐量的降低。现在，很少有计算机能够使高速链路饱和，而且要做到这一点变得越来越困难。光纤通信技术超过了微处理器速度的增长率。计算机(处理器和软件)很快成为新的瓶颈，而且这种状态会保持下去。

长话短说，软件性能是重要的，而且一直如此，这一点不会改变。随着处理器和通信技术的发展，它们重新定义了“快”的含义。它们引发了一种新型的带宽饥饿型和周期饥饿型应用程序，它们推动着技术前沿的发展。软件效率现在变得比以前更为重要。无论处理器速度的增长是否会走到尽头，很显然它都会跟在通信速度的后面。这就使效率的担子落到了软件身上。执行速度的进一步提高很大程度上依赖于软件效率，而不仅仅是处理器。

术语

在继续深入探讨之前，先要为术语的澄清说上几句。“性能”可以代表几种度量，最常用的两种是空间效率和时间效率。空间效率寻求软件解决方案的最小内存占用，而时间效率寻求最小的处理器周期占用。时间效率通常由响应时间和吞吐量来表示。其他度量包括编译时间和可执行文件的大小。

内存的快速降价已经使空间效率的议题成为过去，拥有大量RAM(随机访问存储器)的桌面PC已很常见。现在的公司客户不再考虑空间问题。在我们为客户工作的过程中，所遇到的大部分情况是对运行时效率的考虑。由于客户决定了需求，所以我们将接受他们的需求，着眼于时间效率。以后，我们将把性能限定为时间效率这种概念。通常只有在空间影响到运行时性能的情况下，我们才会考虑空间问题，如提供缓存和分页等。

在对时间效率的讨论中，我们将经常会交替性地提到术语“步长”和“指令计数”。这

两个术语都表示代码段所产生的汇编语言指令。在 RISC 体系结构中，如果代码段展示出合理的“引用位置”（即缓存命中），那么指令数和时钟周期之间的比率大概为 1。在 CISC 体系结构中，该比率的值为 2 或者更大。但是在任何情况下，不管处理器的体系结构如何，较少的指令数量总是意味着较少的执行时间。合适的指令数量对高性能来说是必要但不充分的。因此，指令数量是一种粗略的性能估计，然而它确实是有用的。它将和时间度量一起用于效率的评估。

本书结构

我们使用现实中的例子开始性能之旅。第 1 章是一个关于 C++ 代码的战争故事。这些 C++ 代码给出了糟糕的性能，我们同时说明了解决这个问题的做法。该范例将让您了解一些性能方面的经验教训，它们可以很好地应用于不同的情况。

C++ 中的面向对象设计可能隐藏着性能开销。这是我们为支持面向对象设计功能所付出的代价。在第 2、3 和 4 章中，我们将讨论这种代价的重要性、影响它的因素、如何以及何时避免它。

第 5 章专门讲述临时对象。创建临时对象是 C++ 的特性，它会使新的 C++ 程序员放松警惕。C 程序员不习惯 C 编译器“背后”产生的明显开销。如果打算编写高效的 C++ 程序，那么知道 C++ 编译器何时会产生临时对象以及避免产生临时对象的方法将是基本的要求。

第 6 章和第 7 章的主题是内存管理。随意地分配和收回内存需要付出的代价会很高。诸如 new()和 delete()之类的函数是为灵活性和通用性而设计的。它们在多线程环境中处理长度可变的内存块。同样，它们的速度也是折中的。您时常会对自己的代码做出简化假设，这会显著地提高内存分配与收回的速度。这两章将讨论几种可行的假设以及用来平衡它们的高效内存管理器。

内联可能是仅次于按引用传递对象的第二个最为常用的性能提高技巧。它不像听起来那么简单。就像 register 一样，inline 关键字只不过是一个编译器经常忽略的提示而已。在第 8、9 和 10 章里我们将讨论 inline 被忽略的情况以及预料之外的结果。

性能、灵活性和重用性很少能全面兼顾。标准模板库（Standard Template Library）是扭转这种倾向并把三者合并成强大组件的一种努力。我们将在第 11 章中分析 STL 的性能。

引用计数是有经验的 C++ 程序员经常使用的一种技术。一本致力于 C++ 性能的书籍是离不开这项技术的，我们将在第 12 章中对此进行讨论。

不能总是用少数的几个亮点来补救软件的性能。性能降低通常是许多细小的局部低效所造成的。每个这样的局部低效本身并不明显，但它们综合在一起就导致了性能的明

显下降。几年来，在为不同的C++产品解决许多性能漏洞时，我们发现某些性能漏洞经常出现。我们把这些漏洞分成两组：编码低效和设计低效。编码这一组是触手可及的，可以在不理解全部设计的情况下实现小范围的局部优化。在第13章中，我们对此类的各种情况进行讨论。另一组包括设计优化，实际上是全局性的优化。这些优化要作用于源代码中的所有代码，这是第14章讨论的主题。

第15章讨论可伸缩性问题，这是多处理器环境中出现的独特的性能项，我们不会在单处理器环境中遇到。本章我们要讨论使用并行性的设计与编码问题。同时本章还会对多线程编程和同步所使用的一些术语和概念进行讲述。我们在本书的几个其他地方引用了线程同步的概念。如果您对这些概念的了解有限，那么第15章将有助于深入理解这些概念。

第16章将关注底层系统。出色的性能同样需要对底层操作系统和处理器体系结构有初步的认识。该章我们将讨论诸如缓存、分页和线程等问题。

目　录

提高 C ++ 性能的编程技术