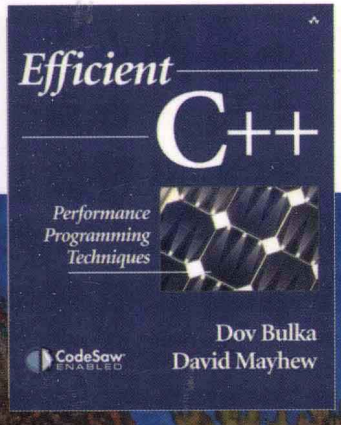
Broadview
www.broadview.com.cn
PEARSON
传世经典书丛
Eternal Classics

提高C++性能的编程技术



[美] Dov Bulka
David Mayhew 著
左 飞 薛佟佟 高 阳 译

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

提高C++性能的编程技术

Efficient C++: Performance Programming Techniques

[美] Dov Bulka David Mavbew 著

左飞 薛佟佟 高阳

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

悦读上品　得乎益友

孔子云："取乎其上，得乎其中；取乎其中，得乎其下；取乎其下，则无所得矣"。

对于读书求知而言，这句古训教我们去读好书，最好是好书中的上品——经典书。其中，科技人员要读的技术书，因为直接关乎客观是非与生产效率，阅读选材本更应慎重。然而，随着技术图书品种的日益丰富，发现经典书越来越难，尤其对于涉世尚浅的新读者，更为不易，而他们又往往是最需要阅读、提升的重要群体。

所谓经典书，或说上品，是指选材精良、内容精练、讲述生动、外延丰盈、表现手法体贴入微的读品，它们会成为读者的知识和经验库中的重要组成部分，并且拥有从不断重读中汲取养分的空间。因此，选择阅读上品的问题便成了有效阅读的首要问题。当然，这不只是效率问题，上品促成的既是对某一种技术、思想的真正理解和掌握，同时又是一种感悟或享受，是一种愉悦。

与技术本身类似，经典 IT 技术书多来自国外。深厚的积累、良好的写作氛围，使一批大师为全球技术学习者留下了璀璨的智慧瑰宝。就在那个年代即将远去之时，无须回眸，也能感受到这一部部厚重而深邃的经典著作，在造福无数读者后发出从未蒙尘的熠熠光辉。而这些凝结众多当今国内技术中坚美妙记忆与绝佳体验的技术图书，虽然尚在国外图书市场上大放异彩，却已逐渐淡出国人的视线。最为遗憾的是，迟迟未有可以填补空缺的新书问世。而无可替代，不正是经典书被奉为圭臬的原因？

为了不让国内读者，尤其是即将步入技术生涯的新一代读者，就此错失这些滋养过先行者们的好书，以出版 IT 精品图书，满足技术人群需求为己任的我们，愿意承担这一使命。本次机遇惠顾了我们，让我们有机会携手权威的 Pearson 公司，精心推出"传世经典书丛"。

在我们眼中，"传世经典"的价值首先在于——既适合喜爱科技图书的读者，也符合专家们挑剔的标准。幸运的是，我们的确找到了这些堪称上品的佳作。丛书带给我们的幸运颇多，细数一下吧。

得以引荐大师著作

有恐思虑不周，我们大量参考了国外权威机构和网站的评选结果，并得到了 Pearson 的专业支持，又进

一步对符合标准之图书的国内外口碑与销售情况进行细致分析，也听取了国内技术专家的宝贵建议，才有幸选出对国内读者最富有技术养分的大师上品。

向深邃的技术内涵致敬

中外技术环境存在差异，很多享誉国外的好书未必适用于国内读者；且技术与应用瞬息万变，很容易让人心生迷惘或疲于奔命。本丛书的图书遴选，注重打好思考方法与技术理念的根基，旨在帮助读者修炼内功，提升境界，将技术真正融入个人知识体系，从而可以一通百通，从容面对随时涌现的技术变化。

翻译与评注的双项选择

引进优秀外版著作，将其翻译为中文供国内读者阅读，较为有效与常见。但另有一些外语水平较高、喜好阅读原版的读者，苦于对技术理解不足，不能充分体会原文表述的精妙，需要有人指导与点拨。而一批本土技术精英经过长期经典熏陶及实践锤炼，已足以胜任这一工作。有鉴于此，本丛书在翻译版的同时推出融合英文原著与中文点评、注释的评注版，供不同志趣的读者自由选择。

承蒙国内一流译(注)者的扶持

优秀的英文原著最终转化为真正的上品，尚需跨越翻译鸿沟，外版图书的翻译质量一直屡遭国内读者诟病。评注版的增值与含金量，同样依赖于评注者的高卓才具。好在，本丛书得到了久经考验的权威译(注)者的认可和支持，首肯我们选用其佳作，或亲自参与评注工作。正是他们的参与保证了经典的品质，既再次为我们的选材把关，更提供了一流的中文表述。

期望带给读者良好的阅读体验

一本好书带给人的愉悦不止于知识收获，良好的阅读感受同样不可缺少，且对学业不无助益。为让读者收获与上品相称的体验，我们在图书装帧设计与选材用料上同样不敢轻率，惟愿送到读者手中的除了珠玑章句，还有舒适与熨帖的视觉感受。

所有参与丛书出版的人员，尽管能力有限，却无不心怀严谨之心与完美愿望。如果读者朋友能从潜心阅读这些上品中偶有获益，不啻为对我们工作的最佳褒奖。若有阅读感悟，敬请拨冗告知，以鼓励我们继续在这一道路上贡献绵薄之力。如有不周之处，也请不吝指教。

电子工业出版社博文视点

二〇一〇年十二月

中文版前言

在岁末年初、万象更新之际，电子工业出版社隆重推出了引进版图书大系——《传世经典书丛》，而我们有幸参与了这项浩繁而极富意义的工作。作为该系列中的一员，《提高 C++性能的编程技术》（Efficient C++: Performance Programming Techniques）是一本指导 C++程序员如何写出高性能程序的经典书籍，曾畅销欧美，可以说是一本不折不扣的经典之作。

该书的两位作者 Dov Bulka 和 David Mayhew 拥有丰富的实践经验和深厚的编程功底，他们将自己工作与学习中的宝贵经验，汇聚成本书，旨在告诉人们这样一个长期被忽略的事实——C++也能写出高效的程序！这对长久以来存在于很多程序员和软件设计师脑中的一种“偏见”构成了极大的挑战，人们总是习惯于认为 C++天生与高效就是对立的。Dov Bulka 和 David Mayhew 所写的这本书成功地否定了这一观点。

Dov Bulka 曾经在杜克大学获得计算机科学博士学位，他在软件开发以及向市场交付大型软件产品方面拥有超过 15 年的经验。他还曾经是 IBM DominoGo Web 服务器的性能设计师。David Mayhew 在弗吉尼亚理工大学获得计算机科学博士学位，并曾担任 StarBridge Technologies 的首席设计师。由于两位作者在商业应用程序开发中对于最佳性能方面的积极探索与实践，掌握了第一手资料，他们借由本书说明了 C++在开发高效程序方面的潜力，同时提出了在实际开发中获得大幅度性能提升的 C++编程技术。本书重点讨论 C++开发中程序性能与可移植能力的提升，通过各种高效技术及精确测控，本书证明 C++在这两方面都可以臻于完美。另外，本书详细讨论了临时对象、内存管理、继承、虚函数、内联、引用计数以及 STL 等一切有可能提升 C++效率的细节内容。本书还指出了在设计和编码中产生隐含操作代价的一些常见错误。

通过本书，读者可以了解 C++程序设计中关于性能提升的主要技术。因此，本书对于渴望提高 C++程序性能的读者来说将大有裨益，而且更重要的是，读者通过本书可以更深入地探讨 C++高级程序设计思路与方法。

参与本书翻译工作的还有南京航空航天大学计算机科学与技术学院研究生宋

通、刘彧、张锐恒和殷科科。他们的谦逊与协作精神，以及对于学术问题的孜孜以求和扎实的技术功底都给我们留下了深刻的印象。在此对他们的辛勤付出表示最诚挚的谢意！

计算机程序设计其实是一门妙不可言的艺术。但是真正能将自己从埋头苦干的工匠变成收放自如的设计师，领会程序设计之美，却非易事。无论何时，在使自己变得更加优秀的过程中，一本好书的作用永远不能被忽视！笔者真诚地希望本书能够在这个过程中帮到各位读者。为此，我们始终以审慎、严谨的态度对待此书的翻译工作，力求最大程度地贴合中国读者的阅读习惯，并且不丧失原作的风采。然而，翻译和出版终究是留有缺憾的艺术，纰漏和欠缺往往在所难免，我们真诚地希望广大读者朋友不吝赐教与指正，这将成为我们将此书不断完善的最强大动力，联系信箱：beckham@vip.163.com，或访问笔者的个人博客 http://baimafujinji.blog.51cto.com/。

译 者

2010 年冬

序　言

如果就C++的性能问题对软件开发者进行一次非正式调查，您会毫无疑问地发现，他们中绝大多数人把性能问题视为C++语言相较于其他优秀语言的阿特琉斯之踵。自从C++诞生以来，我们多次听到这样的话：开发性能要求严格的程序尽量不选择C++。在开发人员看来，编写这一类的程序首选的语言通常是标准C，有时甚至是汇编语言。

作为软件社区的一分子，我们有机会目睹了这种神话般语言的发展和壮大。几年前，我们充满热情地加入了投入C++怀抱的浪潮中，我们身边的许多开发项目也都很积极地投身于其中。一段时间过后，以C++实现的软件解决方案开始发生变动。人们发现C++的性能不太理想，所以就逐渐放弃了它。在对性能要求严格的领域，人们对于C++的热情冷却了下来。当时我们正在为别人提供网络软件，这些软件的运行速度是不容协商的，也就是说速度必须放在第一位。由于网络软件位于软件层级链中相对低的层次，而且大量的应用程序位于网络软件之上并且依赖于它，所以网络软件的性能要求比较严格。较低层级链上的程序性能不佳会对较高层级链上的程序产生影响。

并不是我们才有这样的经验。在我们周围，不少骨灰级的C++使用者也很难用C++取得他们所期盼的性能。没有人认为问题在于面向对象开发模式有多难学，反而怪罪C++语言本身。尽管C++编译器本质上还处于“婴儿期”，但C++语言已经被打上了“天生就慢”的标签。这种看法迅速传播并被广泛地当做事实接受。不使用C++的软件企业经常说他们这么做是因为性能是他们考虑的主要因素。这种考虑源于人们认为C++语言无法达到与之相对的C语言所达到的性能。因此，C++很少在一些对性能要求很严格的软件领域取得成功，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）等。

我们花费了很多年时间用来剖析用C和C++代码写成的大型系统，以充分发挥

这些代码的性能。在整个过程中我们竭尽全力寻找用 C++产生高效程序的潜力。我们发现事实上 C++确实是有这个潜力的。在本书中，我们试着与您分享这些经历，以及在剖析 C++效率的过程中所得到的一些经验教训。编写高效的 C++代码，并非轻而易举，也不是比登天还难。要做到这一点，需要理解性能规律，以及掌握一些有关 C++性能陷阱和缺陷的知识。

80-20 法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认同它的存在：20%的性能缺陷将会占用我们 80%的时间。因此我们把精力集中在最有价值的地方。我们主要对那些经常在工业化代码中出现，并有显著影响的性能问题进行剖析。本书不对各种可能出现的性能缺陷及其解决方法进行详尽的讨论，因此，将不讨论我们认为深奥而不常见的性能缺陷。

毫无疑问，我们的观点来自于我们作为程序员在开发服务器端、对性能要求严格的通信程序中获取的实际经验。由此形成的观点会在以下方面影响本书：

- 在实践中所遇到性能问题与在科学计算、数据库程序和其他领域中所遇问题在本质上稍有不同。这没什么大不了的。普遍的性能原则适用于各个领域（包括在网络软件以外的领域）。
- 尽管我们尽量避免，但有时候还是会人为地虚构一些例子来佐证一些观点。以前我们犯过足够多的编码错误，所以能提供足够多的来自于我们所编写的真实的产品代码中的例子。我们的经验得来不易——是从自己和同事所犯的错误中学习而来的。我们将尽可能使用实际范例来证明我们的观点。
- 我们将不对算法渐近复杂性、数据结构，以及数据的访问、排序、搜索及压缩的最新和最优技术进行深入的研究。这些确实是重要的论题，但是其他地方已给出大量的论述［Knu73、BR95、KP74］。我们将着重论述那些简单、实用、常见的、会大幅提高性能的编码和设计原则。也许您在不知情的状态下使用了会导致隐含性高消耗的语言特性，也许您违反了敏感（或是不太敏感）的性能原则，这些会导致性能降低的常见的设计和编码方法，我们都将一一指出。

我们如何区分传说与现实呢？C++的性能真的比 C 语言的要差么？笔者认为人们通常所持的 C++性能差的观点是不正确的。确实，在一般情况下，如果把 C 语言和看起来与 C 语言相同的 C++版本相比，前者通常要快一些。但同时笔者也认为两

种语言在表面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。我们的观点是如果让 C 语言程序在上述方面达到 C++程序的级别，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，这两者都是有可能的，关键要看怎样使用它以及想从它那里得到什么。这与如何使用 C++有关系：运用得当的话，C++不仅可以让软件系统具备可接受的性能，甚至还可以获得出众的性能。

在此我们向那些对本书做出过贡献的人表示感谢。万事开头难，我们的编辑 Marina Lang 对本书写作项目的启动给予了帮助。Julia Sime 为早期的书稿做出了不小的贡献，Yomtov Meged 也给我们提供了很多有价值的建议。他还指出了我们的想法与客观情况之间的细微差别。尽管有时它们碰巧会一致，但对我们而言还是有必要加以区分的。

十分感谢 Addison-Wesley 聘请的两位评审，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。

最后但并非最不重要，还要感谢我们各自的妻子：Cynthia Powers Bulka 和 Ruth Washington Mayhew。

Dov Bulka

David Mayhew

目　录

导　读

在使用汇编语言编程的时代，有经验的程序员通过计算汇编指令的数量来估计程序的执行速度。在某些体系结构中，例如 RISC，绝大多数汇编指令可以在单个时钟周期内完成。而其他体系结构中的指令执行速度存在很大差异，但是有经验的程序员们能够对指令延迟的平均水平形成一种很好的直觉。如果知道在您的程序片段中存在多少指令，就能准确地估计执行该段程序所消耗的时钟周期数。从源代码到汇编代码的映射是简单的一对一关系，汇编代码就是源代码的另一种形式。

在编程语言的层次中，C 语言比汇编语言高一层。C 语言源代码与相应的编译器编译出来的汇编源代码不是完全同一的。编译器的任务就是缩小源代码与汇编代码之间的差别。从源代码到汇编代码的映射不再是一对一的恒等映射。然而，这里仍然存在一个线性的关系：每一条源代码级的 C 语言语句对应于少量的汇编指令。如果你认为每条 C 语言代码被翻译为 5 到 8 条汇编指令，这就与实际情况八九不离十了。

C++打破了这种源代码级语句与编译器生成的汇编语句在数量上清晰的线性关系。C 语句的开销大体上是统一的，而 C++语句的开销却变化很大。一条 C++语句可以生成 3 条汇编指令，而另一条就可能生成 300 条。编写高效的 C++代码，对程序员提出了一个新的意想不到的要求：穿越一个性能雷区，在一条生成 3 条指令的小路上前进，同时避免进入生成 300 条指令的雷区。程序员必须辨识那些会产生大开销的语句结构，并且知道如何利用它们进行设计或者编程。这些都是 C 和汇编语言程序员从来不需要考虑的。对于 C 程序员而言，唯一需要他们在语句开销上多费点心思的可能是 C 语言中的宏的调用，但它很难像 C++中的构造函数和析构函数一般被频繁地调用。

C++编译器还可能在你不知道的情况下将代码插入程序的执行流中。这对刚使用 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）来说是全新的体验。编写高效的 C++程序要求开发者掌握有关提高 C++性能的技巧，这些技巧是

无法根据通常的软件性能原理而获知的。在C编程中，您一般不会碰到隐性的性能开销，因此有可能获得超出预期的好性能。但是在C++编程中，就很难碰到这样的情况：如果不了解潜在的错误，就无法指望通过偶然性获得好性能。

平心而论，我们见到过许多性能低下的例子，它们来源于低效率的面向对象（OO）设计，当面向对象设计成为主流之后，软件灵活性和可重用性思想被过分宣扬。然而，对于程序而言，灵活性和可重用性之于性能和效率，好比鱼与熊掌难以兼得。在数学中，把每一条定理概括成基本原理是很麻烦的。数学家尽量使用那些已经被证明过的结果去推理。然而在数学之外的领域，利用特殊情况，以及采取捷径，在很多时候是有意义的，在软件开发时，有时候优先照顾性能而不是可重用性是可以接受的。当编写设备驱动程序中的`read()`和`write()`函数时，相比于未来代码的重用性而言，性能方面的表现对于程序的成功则更为重要，这一点是被广泛认可的。某些面向对象设计中的性能问题就是由于在错误的时间、错误的地方强调了性能而造成的。程序员应当集中精力解决当前已有的问题，而不是使当前的解决方案服从于某些未知的将来可能出现的需求。

软件低效的根源

静态的C++开销并不是所有性能问题的根源。即使排除编译器产生的开销也不
3 总足以解释性能上的问题。倘若真是那样，则C程序将由于没有静态开销而自动获得惊人的性能。事实上一些其他因素影响着软件的一般性能，尤其是C++的性能。这些因素是什么呢？图0.1中显示了软件性能归类的第一层。

图0.1 软件性能的高层分类

在最高层中，软件的效率由两个主要因素决定：

- **设计效率** 它牵扯到程序的高层设计，为了解决这一层次的性能问题，您必须熟悉程序的全局构造。在很大程度上，这是与语言无关的。无论何等编码效率都无法弥补糟糕的设计。

- **编码效率** 中、小型的实现问题归于这一类。解决这一类的性能问题常常需进行局部修改。举例来说，在代码段中你无须看很多行，就能够通过把常量表达式放在循环的外面来避免多余的计算。需要理解的代码段就在循环体之内。

这个高层次还可以被进一步分为更小的子主题，如图 0.2 所示。

图 0.2 对设计效率在概念上进一步划分

设计效率可进一步分为两项：

- **算法与数据结构** 从严格意义上说，每一个程序本身就是一种算法。我们所说的“算法与数据结构”，事实上就是访问、搜索、排序、压缩和其他一些管理大型数据集合所用算法的常用子集。

  人们通常自动地把性能与程序使用的算法及数据结构联系在一起，似乎其他因素都无关紧要。需要说明，软件性能仅取决于这一方面是一种错误的观念。算法与数据结构的高效是必要条件，但不是充分条件：它本身不足以保证程序整体的高性能。

- **程序分解** 这牵扯到把整个任务分解成相关的子任务、对象层次、函数、数据和函数流。这是程序的高层次设计，包括组件设计以及组件间通信。几乎没有程序仅由单个组件构成。一个典型的 Web 应用程序至少要由一个 Web 服务器、若干 TCP 套接字及数据库（通过 API）相互配合而成。当涉及组件跨越 API 层的情况时，就会有导致性能低效的陷阱出现。

还可以对编码效率进行进一步分解，如图 0.3 所示。

我们把编码效率分解为 4 个小项：

- **语言结构** C++在其原型 C 中增加了新能力和灵活性。这些新增的益处并不