HZ BOOKS

PEARSON Addison Wesley

Modern C++ Design C++ 设计新思维

STL扩展技术手册

卷Ⅰ：集合和迭代器

Extended STL, Volume 1: *Collections and Iterators*

Matthew Wilson 著

金 庆 宋晨光 郑逾洋 吴圳 译

附赠光盘

机械工业出版社
China Machine Press

C++
设计新思维

STL扩展技术手册

卷Ⅰ：集合和迭代器

Extended STL, Volume 1: *Collections and Iterators*

Matthew Wilson 著
金 庆 宋晨光 郑逾洋 吴圳 译

机械工业出版社
China Machine Press

本书以 STLSoft 为基础，广泛深入地论述了 C++标准库 STL 的相关内容。全书共三部分 43 章，包括标准模板库、扩展 STL、元素引用类别、DRY SPOT 原则、抽象泄漏法则、契约式编程、约束、垫片、不完备结构一致性的发端、资源获取、模板工具、推断式接口适配、Henney 假说、适配、遍历进程和模块、环境变量、字符串分词、聚集分散的 I/O 以及迭代器等内容。

全书通过严谨的表述与丰富的示例，将概念和理论与实际的设计和代码结合起来，从而使读者既能深刻地理解 STL 的知识，又能熟练地掌握 STL 运用方法。

本书版权登记号：图字：01-2008-1744

图书在版编目（CIP）数据

STL 扩展技术手册 卷 I：集合和迭代器/威尔森（Wilson，M.）著；金庆等译. —北京：机械工业出版社，2008.9

（C++设计新思维）

ISBN 978-7-111-24227-7

Ⅰ.S… Ⅱ.①威… ②金… Ⅲ.C 语言－程序设计－技术手册 Ⅳ.TP312-62

中国版本图书馆 CIP 数据核字（2008）第 079391 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：周茂辉

北京瑞德印刷有限公司印刷·新华书店北京发行所发行

2008 年 9 月第 1 版第 1 次印刷

186mm×240mm · 28 印张

标准书号：ISBN 978-7-111-24227-7

ISBN 978-7-89482-692-3（光盘）

定价：65.00 元（附光盘）

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

赞 誉

“Wilson 的 STL 大餐无疑是诱人的美食，无论是对泛型编程的拥护者们，还是对正在开始接纳 STL 和 C++的、殷切的 C 语言程序员们，或是正在重新看待 C++的 Java 程序员们，以及为多种平台和语言开发软件库的作者们。祝大家胃口好！”

——George Frazier，Cadence Design Systems，Inc.

“透彻论述了 STL 扩展中的细节和注意点。”

——Pablo Aguilar，C++软件工程师

“本书不仅论述了扩展 STL，而且也扩展了我的 C++编程思想。”

——Serge Krynine，C++软件工程师，RailCorp Australia

“你可能不是 100% 同意 Wilson 所说的一切，但总体上，他的书是对实用 STL 编程最有价值的、最深入的研究。”

——Thorsten Ottosen，M. C. S.，Boost 贡献者

“Wilson 是一个高明的驯兽大师，他让各式各样的第三方软件库，像驯服的野兽，跳过 STL 的火圈。他仔细地引导读者理解设计思想，指出陷阱，并确保你的脑袋没有被咬掉。”

——Adi Shavit，首席软件设计师，EyeTech Co. Ltd

“Wilson 的书提供的信息，足以改变人们对 STL 扩展的焦虑或不确定程度，让其从‘胆怯畏缩’变为‘切实可行’。”

——Garth Lancaster，EDI/自动化经理，Business Systems Group，MBF Australia

“本书将打开你的眼界，并向你揭示 STL 抽象实际上有多么的强大。”

——Nevin “:-)” Liber，19 年的 C++老手

“鲜有 C++著作讲述扩展的技艺。而 Wilson 的作品，一贯地通过展示各种可行和不可行，以及其中涉及的权衡，将扩展的技艺推向极限。”

——John O’Halloran，软件开发主管，Mediaproxy

“基本的概念和实践，带领职业程序员超越标准库。”

——Greg Peet

“这本书不只是一本关于适应 STL 并用于日常工作的书，也是一次冒险旅行。它带你经历软件设计和概念、C++的强力技术，以及真实世界软件开发中的危险。换句话说，它是一本 Matthew Wilson 风格的书。如果你对 C++的态度是严肃认真的，那我认为你应该阅读它。”

——Björn Karlsson，主设计师，ReadSoft；《Beyond the C++Standard Library：An Introduction to Boost》的作者

译 者 序

自从1998年9月C++标准定案以来，STL就作为标准库的重要组成部分，为广大程序员所熟知，也确实带来了巨大的便利。但作为一种兼顾系统编程和应用编程的语言，C++绝不可能局限于STL的象牙塔之中。只要面对的任务稍具现实性，我们就不可避免地要和所谓“遗产”API（应用程序接口）打交道。

这些编程接口在过去若干年中积累而来，固然堪称宝贵财富，但有时候也会成为C++编程的沉重负担。

- 使用方式让人隐约感到暗合STL的概念，“集合”、“迭代”、“迭代器”的影子似乎触手可及，甚至就是另一种容器实现，但由于缺少所需的编译时接口，无法直接搭配STL算法和适配器。
- 经常采取显式资源管理，其传统的“分配—使用—释放”三部曲，在C++异常大行其道的今天，显得非常脆弱，往往给程序种下慢性病的病根——资源泄露。

几乎每个C++程序员都会面对以上问题。在形形色色的解决方案中，Matthew Wilson编写的STLSoft库可算是个中翘楚，其中包含大量适配代码，让这些传统API摇身一变，不仅可以方便地搭配STL使用，同时具备了异常安全性，让客户代码简洁、清晰、透明到令人吃惊的程度。STLSoft正是Matthew Wilson写作本书的基础，他在书中包含了十余个STLSoft组件，不厌其烦地讲解实现思路和方法，对比直接使用API和使用适配组件的代码，以极其直观的方式让读者体会到二者之间的差异。

熊掌有了，那么鱼呢？性能可算是每个C++程序员心中挥之不去的情结。放心，Matthew Wilson对此绝无含糊其辞、蒙混过关的企图，相反，他始终将组件的运行性能列为重要的设计目标。书中大多数组件都附有实实在在的性能基准数据，清楚地表明“抽象”和“高效”并非不可兼得。

一部容量如此巨大的著作，如果全是罗列组件，讲解实现，未免显得匠气十足。而作者的过人之处在于一边埋头苦干，一边抬头看路，他将STLSoft实现代码背后的理念、观点、感悟也和盘托出。所以书中不仅随处散落着各种“提示”、“规则”、短小精悍的“插曲”，作者还以正式的篇幅讲解STL概念及扩展、元素引用类别、抽象泄露法则、鹅规则和鸭规则，讲解之后，又在书中频繁运用，让人很容易将抽象的概念、理论和实际的设计、代码相联系。十余个组件的设计固然精妙，其背后的思维方式尤其深刻，但看过这本书之后，能够以合理的方式在既有数据、算法之上抽象出STL扩展容器，或者深刻认识STL的概念、方法，为运用STL打下坚实的基础，才算是读者最重要的收获。

人生苦短。我们是否愿意在编写冗长繁复的代码之后，为更新、维护绞尽脑汁，或者在设计漏洞百出的资源管理机制后，为解决时隐时现的bug弹精竭虑；还是为一本好书全心投入，从此和简洁、高效、安全的代码结缘？答案不言而喻。

译 者

2008年3月

前　言

我的伯父 John，是那种我父母一代所谓的男子汉。他强壮、粗犷、有点吓人，很有牛仔气概，而且他有勇气承认自己害怕，不像我只愿意述说自己的小失败。所以，当他说第二次跳伞的挑战是克服已知的恐惧，我记住了。我现在已经写了两本书，可以毫无疑问地证实这种相同的恐惧。开始第二次时，你知道有多少痛苦即将来临，这不是件能轻易做到的事。那么问题是，为什么我还这样做呢？

其原因将在序言中阐述，总的来说，我试图对以下看似简单的两面性做出回应：

- C++太复杂了。
- C++是惟一足够强大，能满足我需要的语言。

这种两面性最突出的一个领域是，使用和扩展标准模板库（Standard Template Library，STL），尤其是扩展。本书（及其姊妹篇，卷 2），是我在过去 10 年左右的时间内，为解决这一挑战性课题而积累的知识和经验的精华。

目标

本书描述了一个使用和扩展 STL 的好办法。书中定义了以下内容：

- 集合的概念，以及它和容器概念的区别
- 元素引用类别的概念，包括它的重要性如何，如何定义，如何检测，及其施加在 STL 扩展集合和迭代器设计之上的妥协
- 外部迭代器失效的现象，及其令人惊讶的行为对 STL 兼容集合设计的影响
- 一个能探测任意集合特性的机制，不论该集合是否提供变动性操作

书中说明了几个问题：

- 为什么变换迭代器适配器必须按值返回元素
- 为什么过滤迭代器必须总是输入一对迭代器来进行操纵
- 如果在迭代中，底层集合有改动，该怎么办
- 为什么你应该禁止输出迭代器类中无意义的语法，以及如何利用**解引用代理**模式做到这点

书中展示了以下方法：

- 将批量返回元素的 API 适配到 STL 集合概念
- 将逐个返回元素的 API 适配到 STL 集合概念
- 共享枚举状态，以正确实现输入迭代器概念的需求
- 枚举可能是无限的集合
- 为特定迭代器类型特化标准算法，以优化性能
- 为系统环境变量定义一个安全，平台独立的 STL 扩展，通过全局变量实现
- 适配一个集合，其迭代器实例的可拷贝性要在运行时才能决定
- 提供对可逆但不可重复集合的访问
- 利用迭代器写字符缓冲区

本书会解决这些以及更多问题。它还着眼于如何构建通用的，STL 兼容的软件库，同时不牺牲健壮性，灵活性，特别是，性能。本书教你如何制作抽象蛋糕，用效率的奶油裱花，然后吃了它。

通过阅读这本书，可以有如下收获：

- 学习 STL 扩展中特殊的原则和技巧
- 查看 STL 扩展的实现，来学习更多关于 STL 的知识
- 学习实现封装的通用技术，在操作系统 API 和专用技术库之上实现封装
- 学习如何编写迭代器适配器，并理解其实现和使用上的限制，及其背后的原因

- 获得通用软件库性能优化的技术
- 在 STL 扩展中使用经过考验的软件组件

关于主题

我相信，你必须写你所知道的东西。因为本书的主要目的，是传授对 STL 扩展的过程和问题的认识，所以大部分讨论的材料来自于我的工作，即我自己的（开源的）STLSoft 库。由于我是从零开始，实现了 STLSoft 几乎所有的功能，因此，它是使我说话有权威性的最佳材料。当我们讨论设计上的错误时，这一点尤其重要，公开地详细评述他人的设计错误，不太可能取得许多积极成果。

但是，这并不意味着，阅读本书会迫使您使用 STLSoft，或者，其他软件库的追随者不能在此学到任何合乎他们惯例的东西。事实上，本书不会劝诱人们使用任何特殊的软件库，它所展现的材料是让你从里到外查看 STL 扩展，重点是 STL 的原则和扩展的实践，而不是依赖于 STLSoft 或任何其他软件库的现有知识。如果当你读完这本书，你仍然不想使用 STLSoft 库，没关系，只要你学到了有用的知识，知道如何实现和使用其他 STL 扩展即可。

在示范 STL 扩展的方法时，我不自诩我所使用的方式是最好的。C++是一种非常强大的语言，支持多种风格和技巧，以至于有时候反而会不利。举例来说，许多集合，但不是全部，最好实现为 STL 集合，而有些则是用独立的迭代器表示会更好。其中有相当多的重叠，需要忍受许多模棱两可的表达。

对于大多数讨论的 STL 扩展，我让读者（在某些情况下还有作者!）参加一个旅行，从原始的 API，先实现初步的封装，但这往往是有缺陷的，然后达成一个最优的，或者至少是最理想的版本。我不回避实现上和概念上的复杂性。事实上，为了将外部 API 塑造成 STL 的形式，某些技术必须包含一定程度的技巧。我不会为了叙述简单，假装这些东西在实现中不存在，或对它们不作任何解释。我将涵盖这些东西，并在这样做时，希望能够做到两点：（1）揭示它们的一些复杂性，（2）解释它们为什么是必要的。

理解 STL 最好的方式之一，是学习如何实现 STL 组件，而学习如何实现 STL 组件，最好的方式是实现它们。如果你没有时间（或爱好）去实现 STL 组件，我建议你使用第二个最好的方式，就是阅读本书。

本书结构

本书分为三大部分。

第一部分：基础

这一部分是一些篇幅较小的章，为第二和第三部分讨论的材料提供基础。它首先简要回顾 STL 的主要特点，随后讨论了有关 STL 扩展的概念和原则，包括一个新概念的介绍，即*元素引用类别*。下面几章考虑基础性的概念、机制、范式和原则：一致性、约束、契约、DRY SPOT、RAII、垫片。余下几章涉及模板工具和技术，包括特征类，和*推断式接口适配*，以及在第二和第三部分讨论的实现中用到的几个关键组件。

第二部分：集合

这部分是本书的主体。每一章涉及一个或多个相关的真实世界中的集合和集合的适配，使之成为一个 STL 扩展集合组件，同时还有适当的迭代器类型。主题上，适配的对象种类繁多，如：文件系统枚举、COM 枚举器、非 STL 的容器、分散/聚集 I/O，甚至还有元素受外部更改的集合。涉及的问题包括：迭代器类别选择和元素引用类别的概念、状态分享、可变性和外部迭代器失效。

第三部分：迭代器

第二部分中的内容包含与集合关联的迭代器类型的定义，而第三部分是专门针对独立的迭代器类型的。涉及的主题范围从定制输出迭代器类型（包括对 std :: ostream_iterator 功能的简单扩展的讨论），到复杂的迭代器适配器（可以对它们所应用的底层区间，进行类型和值的过滤和转换）。

第2卷

第2卷尚未完成，其结构也尚未最后确定，但将包含如下内容：函数、算法、适配器、分配器，以及STL扩展概念：区间和视图。

补充材料

光盘

本书所附光盘包含各种免费软件库（包括所有文中涉及的库）、测试程序、工具和其他有用的软件。还包括三个完整的但未经编辑的章节（这三章没有印刷出来，是为了减少篇幅，或是为了避免太多的编译器相关性），还有许多来自其他章节的注解和小节。

在线资源

补充材料也可从网上下载：http://extendedstl.com/。

致　谢

面对我的妻子，Sarah，我一直以来都心怀歉疚。和上次一样，这本书的截稿日期也是一拖再拖，但她都无怨无悔地支持我。她希望这是我的最后一本书。婚姻不外乎妥协，我已经答应她了，这是我最后一本耗费数年（而不是月）的书。但愿我不会食言。

我还必须感谢我的母亲和 Robert，他们坚定地支持我，尤其是容忍我不分白天黑夜地提出各种语法问题（他们用格林威治时间，而我用悉尼时间）。

在本书的写作过程中，我骑行了数千千米，大多是和我（更年轻，也更健康）的朋友 Dave Treacy 一道。在那些日子里，我们一路畅行，每过几千米就停下来记录最新的灵感。和 Dave 交谈很有助于把我从 STL 中解脱出来，他的鼓励也是我最需要的。谢谢你，Dave，但是要小心哦：总有一天我们再飙车时，“那么医生”会把你远远地抛在后面。

在我们最近的交往中，Garth Lancaster 扮演了许多角色：顾问、客户、朋友、美食家，以及审稿人。Garth，感谢你频繁不断但又总是深思熟虑地提出各种建议，包括对我的程序库、书稿、职场生涯，还有我的晚餐路线。在 Nino 家的发行聚会上见！

移民澳大利亚后，Simon Smith 是我交往最久的朋友之一，这是一个天赋异秉的家伙。我这么说，部分原因是他出色的技术管理才能，让他不断在更好更大的公司得到认可以及邀请，但主要原因是他一直雇佣我分析他掌管的公司，然后判断在哪里运用他那独一无二的天赋，让它们运营更好、更快、更节省？（他的处事和为人都很友善）。

本书的许多写作是在高分贝音乐中完成的，所以我必须感谢那些杰出的芬克音乐家和乐队：808 State、Barry White、Level 42、MOS、New Order、Stevie Wonder，以及让人欲罢不能的 Fatboy Slim。来吧 Norman，再来一曲！我还想特别感谢两位艺术家，在一个过分狂热的男孩转变成一个过分狂热的叔父、丈夫和父亲的过程中，他们天籁般的音乐总是伴我左右。首先是 George Michael，为他那伤感的唱腔和芬克式的节奏，也为他终于证明慢烤的馅饼最好吃（过去 18 个月中，我一直在让我的编辑相信这句话，因为我错过了原定的截稿日期）。接下来，虽说我就是那种“四门功课得 A”的男孩（不过我从未在 Jodrell Bank 工作过），我还是感谢 Paddy MacAloon，他无疑是近 30 年来最伟大的作词家。

至于我的编辑，我必须感谢 Peter Gordon，在这又一次马拉松般的努力过程中，他总是富有技巧而又脚踏实地地指引我，还委婉地嘱咐我要“精简文字”。Peter 的助理，Kim Boedigheimer，也值得我一次又一次地感谢，她帮助安排各个相关环节，当我不断征求意见、预支稿酬，还免费拿书的时候，她总是表示尊重。也感谢 Elizabeth Ryan、John Fuller、Marie McKinley，尤其是我耐心、宽容的文稿编辑，Chrysta Meadowbrooke，她让我想起声音甜美的修女。

到这里我必须感谢我的各位审稿人，我欠他们太多了：Adi Shavit、Garth Lancaster、George Frazier、Greg Peet、Nevin ：-）Liber、Pablo Aguilar、Scott Meyers、Sean Kelly、Serge Krynine，还有 Thorsten Ottosen。千言万语也说不尽他们对我的帮助，所以按照惯例，我感谢他们正确地引导我，有时候我在可能错误的假设上固执己见，那都是我一个人的不是。

还有几位人士以另外的方式影响了本书的进程，我也想感谢他们：Bjarne Stroustrup、Björn Karlsson、Garth Lancaster、Greg Comeau、Kevlin Henney，以及 Scott Meyers。以上每一位都送给我建议和诤言，尽管言语细微而温和，对本书（以及卷 2）的结构却影响甚大。多谢了。

最后，我想感谢程序库 Pantheios 和 STLSoft 的用户们，他们给我鼓励，还提出正确的技术评论、请求和建议，甚至索取本书的预印本！希望最后结果不要让他们失望。

结语

顺便提一句，听 John 叔叔说第三次跳伞非常容易，所以我会振作精神准备下两部作品，这本书一交给出版社就继续工作。明年再见！

序　言

难道每门语言都难免日趋复杂，并最终绊倒在复杂性的门槛上吗？

——Adam Connor

难用的话，少用就是了。

——Melanie Krug

事物的两面性

3 年前，《Imperfect C++》快要完工时，我跟编辑说起这本《Extended STL》，当时我信心十足地声称它会是一本易读易懂且轻薄短小得可以轻松从两个抽象层之间滑过的小册子。此外我还保证会在半年之内写完。结果呢？在写这篇序言的时候，离当初约好的截稿日期已经过去了一年半有余，而且，本来计划好的一本薄薄的、约 16 至 20 章的小册子现在也膨胀成了两卷本，其中第一卷洋洋洒洒 43 章（含“插曲”章），哦，对了，CD 上还有 3 章呢。但话说回来，当初有一个保证现在仍然可以说是成立的，那就是这是一本对任何有一定 C++经验的读者来说都是轻松易懂的。

为什么本书后来的情况远远超出我当初的预计呢？并不是因为我是软件工程师——大家都知道软件工程师估计的工作量，乘以 3 之后才是实际需要的时间。而是因为（我认为）以下 4 个重要的原因：

1. STL 并不直观，花费可观的智力投资后方能熟练运用。

2. 没错，STL 在技术上已臻化境。虽然 STL 在内聚性方面超凡入圣，然而 STL 前瞻性不够，对于在它那有限的概念定义之外的抽象，它并不能妥善应付。

3. C++语言本身是不完美的。

4. C++是一门难学的语言，但你得到的回报是效率，而且同时又并没有牺牲设计能力。

最近几年 C++与时俱进，这一方面意味着 C++变得非常强大，但另一方面也暴露出了其晦涩难懂的一面。如果你试着编写一个具有一定规模并用到模板元编程的模板库，那么一种可能是你将学到许多东西，并掌握了一个强大的工具，但另一种可能是你编写出的那堆东西除了 C++狂热信徒之外谁也无法理解。

C++语言本来就是通过扩展来提升功能。除了一些很有限的应用是将 C++看作“更好的 C”来使用之外，绝大多数 C++使用都是围绕着类型定义（类、枚举、结构、联合）来进行的，而且这些自定义类型很大程度上被做成与内建类型界面一致。也正因为这个原因，C++中的许多内建操作符都是允许重载的。例如，vector 可以重载下标运算符 operator[ ]，来模拟内建数组的界面；任何可被拷贝的类型（一般）都定义了拷贝赋值运算符。如此等等，不一而足。但由于 C++的不完美、强大以及极强的可扩展性，伴随而来的便是 Joel Spolsky 所说的*抽象泄漏法则*（Law of Leaky Abstractions）：“所有非平凡的抽象在某种程度上都是有漏洞的。”简单来说，这句话就意味着，要想顺利使用非平凡的抽象，就必须对抽象下面的东西有所了解。

这也正是许多 C++开发者重新发明轮子[㊀]的原因之一。其实这里的原因并不仅仅是*非我发明症*（Not Invented Here，NIH）[㊁]，而是因为我们常常发现自己所用的第三方组件除了 80% 的功能是自己能理解并使用的之外，剩下的 20% 往往裹着一团晦涩的黑气。造成后者的原因很多：复杂性、与既有概念或惯用法的不一致、低效、效率、范围局限性、设计或实现的不优雅、糟糕的编码风格等。而且，现阶段编译技术的一些实际问题还会极大地加剧这种情况，尤其是遇到模板实例化过程中的错误消息时。

㊀ 指自己动手编写所需的库。——译者注

㊁ “非我发明症”出自《The Art of Unix Programming》，指对自我创新的能力颇为自负，并拒绝采用或购买他人或别的公司所发明的技术。——译者注

我觉得我之所以有资格写这本书，原因之一就是我花了大量的时间来研究并实现STL相关的库，而不是接受C++标准（1998）所指定的库或其他人写的库。而我决定写这本书的原因之一则是想将我在以上过程中学到的东西总结出来。如果你想要编写STL扩展，本书可以为你提供帮助。而就算你只是想使用其他人写的STL扩展，本书也同样有用，因为抽象渗漏法则决定了你很可能会不时地需要掀开抽象这块幕布往里面瞧一瞧。

UNIX编程的原则

在《The Art of UNIX Programming》（Addison-Wesley，2004）中，Eric Raymond总结了UNIX社群的最佳实践准则，这些准则来自大量不同的经验。在我们改装STL的宏大计划中，这些准则就像标灯一样为我们指明方向：

- 清晰原则：清晰胜于技巧。
- 组合原则：设计能够互相连接的组件。
- 多样性原则：质疑任何被声称为“真正惟一”的途径。
- 经济原则：和机器的时间相比，程序员的时间是更宝贵的。
- 可扩展性原则：设计着眼未来，因为未来比你想象得来得更快。
- 生成原则：避免手动编码，可以的话，编写程序来生成程序。
- 最小意外原则：在接口设计中作出的决策应该始终是那个令人最少感到意外的选择。
- 模块性原则：编写简单的模块，模块与模块间通过干净的接口连接。
- 最大意外原则：如果免不了要失败的话，要弄出最大动静，而且失败得越早越好。
- 优化原则：首先要能工作，然后才能谈得上优化。
- 吝啬原则：除非确无它法，否则不要编写大的组件。
- 健壮性原则：透明性和简单性是健壮性的父母。
- 分离原则：策略和机制分离，接口与引擎分离。
- 简单原则：设计应该是简单的，只在必须的时候才增加或暴露复杂性。
- 透明原则：设计的时候应考虑透明性，以方便检查和调试。

优秀C++库的七个标志

除了以上原则之外，本书（及第2卷）中的内容也遵循优秀C++库的七个标志：效率、可发现性、透明性、表达力、健壮性、灵活性、模块性以及可移植性。

效率

当年，我刚走出大学校门的时候，身负四年C、Modula-2、Prolog以及SQL编程经验（本科阶段）外加三年用C++写光网络模拟器的经验（研究生阶段），一时间以为自己牛得不行。而且更糟的是我觉得任何使用C/C++之外的语言的程序员都是无知、没脑子、不成熟的。实际上，我至少又花了12年时间才对C++初窥门径。我的最大错误是，没有认识到其他语言的优点。

如今我理智了许多，我意识到软件工程是一块广大的领域，其中有着许多不同的需求。执行时间并非惟一重要的因素，更不用说还有凌驾于它之上的“经济原则”了。比如说编写系统脚本时，用C++要花三天才能写完（只为了获得10%的性能提升）的程序使用Python或（复杂一点的）Perl或（最好是）Ruby可能只要三十分钟就搞定了。许多时候，当同一数量级上的性能差异无伤大雅时，选择Python/Perl/Ruby要明智得多。以前我觉得C++才是王道，现在呢，我在许多任务中都选用Ruby。后者不会让你抓狂到掉头发。不过，话说回来，当需要编写的是健壮的高性能软件时，C++仍然是我的首选。

如果你不需要也不想编写高效代码的话，那就别用C++，也别读这本书。

（不过你还是应该买下它！）

许多人会认为选择C++还有其他原因，尤其是const正确性、强编译期类型安全、语言设施对实现

用户自定义类型的上乘支持、泛型编程等等。我承认，这些的确很重要，而且许多其他语言也的确缺乏这些特性，但是当权衡了一般软件开发中的所有考虑（尤其是组合原则、经济原则、最小意外原则、透明原则和优化原则）之后，（至少对我来说）很明显效率才是那个不可或缺的考虑。C++的许多顽固的反对者仍然声称C++的效率并不高。对于这些迷失的灵魂，我只能说，如果你的C++程序运行缓慢，那只能是因为你没有正确使用它。更多的人会争辩说他们也可以用其他语言写出高效的程序。我承认在少数特定的应用领域内的确如此，但如果有人认为有任何其他语言可以在效率和应用范畴方面和C++抗衡的话，那他就大错特错了。

你可能会问，那我们为什么非得要效率不可呢？对此坊间流传的说法是，除非材料科学取得突破性进展，否则电子基板上的性能提升的数量级终将到顶。我没上过什么“谦卑预言学校”，因此在明确的证据出现之前我会对这个说法保留一点怀疑。不过就算我们在非量子的基板上进一步榨取出“剩余价值”来，有一点还是确凿无疑的，那就是（由于非技术的原因）操作系统还会变得越来越庞大和缓慢，而软件的复杂性也会继续水涨船高，同时软件也会运行在越来越多样化的硬件设施上。效率很重要，并且效率可能一直都会重要下去。

那么这种对效率的强调与优化原则是否相悖呢？乍一看可能的确如此。然而你得注意，库的用户群通常是很广泛的，而且库的生命期通常也很长，这就意味着总会有一部分用户对性能是有需求的。因此，一个成功的库除了要保证正确性之外，通常还要考虑效率。

STL的设计理念之一就是要非常高效，因而如果正确使用它的话效率的确会很高，本书后面就会有很多这样的例子。STL之所以高效，部分原因是因为它对扩展是开放的。但另一方面，STL也非常容易被不当使用（扩展），这时便会导致低效的代码。因此，本书的重点之一便是要倡导C++库开发中的效率友好的实践方式，我觉得这一点无可非议。

可发现性与透明性

虽说效率是使用C++的一个有动机，但在另外两个因素，透明性和可发现性面前它往往会略失风采，几乎任何你考虑使用的库，透明性和可发现性都是必要的。Raymond的《The Art of UNIX Programming》中对这两个软件属性作了一般性的详细定义。不过本书中我就用自己的有关定义了，因为它们更贴近C++（和C）库：

定义 可发现性是指要想使用一个组件需要先花上多大功夫来理解它。

定义 透明性（transparency）是指要想修改一个软件需要先花上多大功夫来理解它。

可发现性主要是指一个组件的接口有多明了（即“一望便知”），这里所说的接口包括形式、一致性、正交性、命名惯例、参数相关性、方法命名、惯用法的使用等等。除此之外可发现性还包括了文档、指南、范例等——只要是能帮助读者领悟到组件怎么用的资料或信息都算。所谓可发现（discoverable）接口是指容易被正确地使用，且不容易被错误地使用的接口。而透明性则更多关注于代码——文件布局、括号的使用、局部变量名字、（有用的）注释等等。不过除此之外它也包括实现文档（如果有的话）。这两个属性通过多个抽象层次相关联：如果一个组件的可发现性不佳，那么使用该组件的代码便会在透明性上打折扣。

另外我还想告诉你一个个人经验：在我的专业生涯中，真正被我在商业代码中大量使用的非专有C++库可谓少之又少（而且就连这寥寥可数的几个库在灵活性方面也仍然大有欠缺）。那要用C++来完成任务怎么办呢？自己写库。这话听上去很容易让人觉得我是个典型的“非我发明症”患者。然而事实远非如此，在其他语言中我使用了大量的第三方库：我用过几十个C库（或者具有C API的库），而且感觉很不错。而在其他语言如D、.NET、Java、Perl、Python、Ruby中，我更是会毫不犹豫地使用第三方库。那你要问了，为什么就C++中不行呢？

答案跟效率有点关系，但更主要的还是因为可发现性和透明性（与基于运行时多态的面向对象毫无关系。不过关于这一点的讨论不在本书的讨论范畴之内，而且也有点超出了我的专长领域和兴趣）。

好的软件工程师对最佳方案有一种直觉，其中之一便是对使用一个组件的代价和好处具有长期磨练出来的感觉。如果一个组件在性能和/或特性方面有不少保证，但在可发现性方面却不容乐观的话，这种感觉便出现了，你内心的声音告诉你别使用它。而这也正是“自己动手，丰衣足食”的时候了。

进一步说，就算可发现性还不错（甚至非常好），那也并不意味着该组件就是好选择，因为它在透明性方面仍可能不行。透明性也很重要。简单明了的接口固然很好，但如果实现一团乱麻的话，想要修改或改进也是无从下手。其次，如果权衡到最后你还是倾向于使用一个组件而不管它的低可发现性的话，那么你可能会需要“偷窥”它内部的实现来弄明白究竟如何用它。第三，你可能想要知道如何实现一个类似的库，这也正是开源时代的库的重要角色之一。最后一点，有一个常识是，如果实现糟糕，那么其他方面也未必好到哪儿去。

对我来说可发现性和透明性是C++库的两个极重要的性质，这一点不仅在本书中被重点强调，在我的代码中也如此：在我写的库当中你会看到清楚明确的（有人可能会觉得是“学究式”的）代码结构和文件布局。当然，我并不是说所有代码都是透明的，“但是我在努力，Ringo，我真的很努力”[⊖]。

表达力

人们使用C++的另一个原因便是C++具有极强的表达力（强大）。

定义 *表达力（Expressiveness）是指，用尽可能少的语句清晰地完成任务。*

表达力强的代码有三大好处。一，提供更高的生产率，这是因为不仅需要编写的代码变少了，而且代码的抽象层次也更高。二，促进了代码复用，进而带来了更好的健壮性，因为被复用的组件的实现在它们被使用的上下文中得到了更多的覆盖。三，使代码更少bug。bug的数量与代码行数在很大程度上是有一定关系的，此外，当代码工作在高抽象层面上时，控制流问题和显式内存管理的减少也会在一定程度上减小bug发生的机会。

举个具体的例子，以下C代码片段的功能是删除当前目录下的所有文件：

```
DIR* dir = opendir(".");
if(NULL ! = dir)
{
  struct dirent* de;
  for(; NULL ! = (de = readdir(dir)); )
  {
    struct stat st;
    if( 0 = = stat(de - >d_name, &st) &&
        S_IFREG = = (st.st_mode & S_IFMT))
    {
      remove(de - >d_name);
    }
  }
  closedir(dir);
}
```

我猜大多数合格的C程序员只要看一看就能知道以上代码是干什么的。比如说吧，假设你是一个有经验的VMS/Windows程序员，而你最先接触到的UNIX代码就是以上这段代码。我敢打赌你立即就能领会代码的意思，惟一可能不理解的地方只有S_IFREG和S_IFMT这两个文件属性常量。这段代码透明性很高，这意味着opendir/readdir API的可发现性很高，后者的确是事实。然而另一方面这段代码的表达力却不算很强，它很冗长，而且里面包含了许多显式的控制流，每次你想要完成类似的任务时都得重

⊖ 《低俗小说》（Pulp Fiction）最后Jules的一段话：“或许，你是那个罪人，而我是那个引路人，自私和邪恶的是这整个世界。我很喜欢那样，但是那不是真相。真相是，你是弱者，而我是恶人。但是我在努力，Ringo，我真的很努力的想去当那个引路人。”——译者注

复这份逻辑并加上一点自己的小改动，经典的复制粘贴，真是活受罪。

现在让我们来看看 C++/STL 式的做法，在我的 STLSoft 的 UNIXSoft 子项目中有这样一个类：unixstl::readdir_sequence（第 19 章），其用法如下：

```
readdir_sequence entries(".", readdir_sequence::files);
std::for_each(entries.begin(), entries.end(), ::remove);
```

与前面的 C 范例不同的是，以上这段代码不多不少，不肥不瘦，每一个单词都适得其所。对任何熟悉 STL 中的迭代器对（区间）和算法惯用法的人来说，这段代码的可发现性是非常高的。其第二行相当于说："对区间 entries. begin( )，entries. end( )）内的每个元素执行 remove( )。"这里用的是半开半闭区间，以方括号开始，圆括号结束，表示从 entries. begin( ) 到 entries. end( )（不含 entries. end( )）的所有元素。就算原来不知道 readdir_sequence 类，也没有任何文档，只要你知道或能猜到 readdir 可能意味着什么的话，这段代码的含义都是极其显然的，这便意味着透明性，而透明性则进一步意味着 readdir_sequence 的可发现性。

不过，高阶表达力也有它的缺点。首先，过多的抽象是透明性的敌人（从而有可能进而影响可发现性）。这就是为什么我们应该把抽象层次相对放低的原因：抽象太多的话，系统整个的透明性便会打折扣（即便在某个给定层次上的抽象是好的）。其次，由于一小段代码背后往往做了大量的工作，性能方面可能便会有所受损：一般来说，抽象下层的代码的效率也很重要。最后，组件的用户被限制在抽象层所暴露出来的特性上。在我们的例子中，readdir_sequence 提供了 flags 来让你过滤文件还是目录，但并没有让你基于文件属性或文件大小来进行过滤。高层抽象可能会提供一些看似随便选择的功能，这可能会带来问题（毕竟，每个人的需求都不尽相同！）。

此外对于 STL 组件来说还有另外两个问题。一是许多 STL 库，包括好几个标准库实现，在可读性方面都很差，这便对透明性产生了负面的影响，使得运行期 bug 很难调试。而编译期 bug 的情况也好不到哪儿去，甚至更糟。即便是在最新的 C++编译器上，模板实例化过程中的错误也难以阅读，这就意味着一旦代码有问题，透明性和可发现性就会极度下降[⊖]。甚至就算你的代码很简单，编译器给出的错误消息也仍可能令人丈二和尚摸不着头脑[⊖]。因而，编写 STL 扩展库的一个重要方面便是要预料到这些晦涩难懂的编译错误，并相应地在代码里面加入一些非功能性的代码来缓解它们，以免用户在遇到这些情况的时候被吓着。后面讲到组件的实现的时候你会看到一些例子。

第二个问题是，像上面那样因表达力而明显获益的例子在 STL 中并非总是成立。你往往发现想要的函数在库里面并没有提供，于是要么你自己写一个自定义的函数类（仿函数），结果降低表达力（因为你必须跳出当前作用域去编写单独的函数类）。要么呢，你就得求助于函数适配器，后者则会降低代码的可发现性和透明性。本书的第二卷会介绍一些对付这类情况的技术，但没有哪个在效率、可发现性、透明性、灵活性和可移植性方面全都是完美无缺的。

健壮性

代码再漂亮，不能工作那也没用。C++有时被遣责为纵容恶习的语言。当然，我是不赞同这种观点

⊖ 简单统计一下你平常上 C++新闻组或论坛上的帖子有多少是关于编译错误的，其中又有多少是涉及模板的就知道了。——译者注

⊖ 试着把以下这段代码输入编译器，然后欣赏壮观的编译错误吧。——译者注

```
#include <algorithm>
#include <list>

int main()
  {
  std::list<int> li;
  std::sort(li.begin(), li.end());
}
```

的，而且我还要说，只要适度的注意，C++程序可以是极其健壮的（我在现实中最爱的付薪工作便是编写运行若干年不出错的网络服务器。当然，不出错的缺点便是维护工作的钱就赚不到了，因为根本没机会去维护。在澳洲，我写的软件携带数十亿的事务穿梭于大陆之间，稳健无比。而我从未有机会通过修正软件的 bug 来赚点小钱，唉，没办法，难道本来不就该如此吗）。

而模板的使用则导致了需要更多的手段才能确保软件的健壮性，因为编译器仅当实例化一个模板的时候才真正完成类型检查。因而模板库中的 bug 便可能在很长一段时间内都不被编译器检查出来，直到被某个特定的实例化触发。要想缓解这种情况，C++库，尤其是模板库，便需要大量运用契约式编程中的 enforcements（第 7 章）来侦测无效状态，以及运用约束（第 8 章）来防止编译器进行不该进行的实例化。

清晰原则、组合原则、模块性原则、分离原则都与健壮性密不可分。健壮性是本书重点讨论的内容之一。

灵活性

最小意外原则和组合原则是说组件的工作方式应该符合用户的预期。而模板技术刚好对此有不俗的贡献，运用模板技术，我们可以定义适用于任何类型的功能。一个经典的例子就是 std :: max( ) 函数模板：

```
template <typename T>
T max(T const& t1, T const& t2);
```

该函数提供的这种通用性既容易实现又容易理解。不过要想故意刁难它也不难：

```
int  i1 = 1;
long  l1 = 11;
max(i1, l1);     // Compile  error!
```

这是小问题。有的问题则更严重一点。比如，你想用某个类来加载一个动态库（譬如这个类叫 dynamic_library），动态库的路径被你放在一个 C 风格的字符串内，如下：

```
char const*      pathName = . . .
dynamic_library  dl(pathName);
```

现在，假设你要换用 std :: string 来存放路径，那你就要修改两行代码，尽管从逻辑上你做的还是原来的操作：

```
std::string const&   pathName = . . .
dynamic_library      dl(pathName.c_str());
```

这就违反了组合原则。dynamic_library 类应该能够同时应付 C 风格串和 std :: string 对象。不满足这一点便会带来不必要的麻烦，而且也会导致混乱的、不利于修改的代码。

模块性

模块性不好往往导致膨胀且脆弱的单片式框架，结果就是令用户感到不愉快、性能和健壮性糟糕、表达力和灵活性受损（还没提到编译时间问题呢！）。由于 C++使用了静态类型检查并沿袭了 C 的声明/包含模型，因此我们经常会发现代码中存在不恰当的耦合。尽管C/C++支持类型前导声明，但也只有当你使用的是类型的指针或引用时才能这么干。况且后者跟C++所倡导的值语义还有所不同。

模板库，如果运用得当的话，在模块性方面绝对是一柄利刃。由于模板实例化时，编译器只判断结构一致性（Structural Conformance，10. 1 节），因此，编写模板库成为可能，模板库可以与多种类型一起工作，只需这些类型符合特定的概念（concept），并且模板库不需要预先包含这些类型。STL 便是这方面的绝佳范例，其他成功的 C++库莫不如此。

可移植性

除非你深信目前你的库被使用的上下文（架构、操作系统、编译器、编译器设定、标准/第三方库

等等）将长期有效，否则作为一个成功的库的作者，就必须考虑可移植性。这个逻辑几乎是没有例外的，这便意味着在实践当中几乎所有的库作者都必须花精力来避免他们的库被遗弃。

你只需扫一眼你机器上的系统头文件就会发现可移植性问题是怎么回事了。但可移植性并不像你想象得那么简单，否则的话一大堆聪明的程序员也就不会老在上面栽跟头了。编写可移植的代码要求对你所依赖的假设始终有一个清楚的了解。这些假设的范围很广，比较明显的像硬件架构和操作系统，微妙一些的如库的版本，甚至库中的错误以及绕过它们的方法。

可移植性问题的另一个来源就是C++方言。大多数编译器都提供选项来关掉一些C++语言特性，其实这就相当于提供了一门C++方言（或语言子集）。例如一些很小的组件经常是在关掉了异常支持的情况下构建的。只要小心行事（并花些功夫），这类情况下的可移植性是可以解决的，后面讲到的一些组件就是例子。

STL扩展从本质上需要高度的可移植性，从应付不同的操作系统差异，到应付编译器bug和语言方言，这也是可移植性在本书中受到重点对待的原因之一。

权衡：适应性满足、方言化和新的和老的惯用法

极少有库能够在前面提到的七个方面都能得高分，这也在情理之中。只有很小规模且功能很紧凑的库才可能做到这一点。而对于其他库而言要做到这一点则要求在七个因素之间进行恰当的权衡。而本节的内容就是我在作出成功的权衡中用到的策略。

对于重实践轻教条的人来说，大多数问题都没有一个单一而清晰的答案。我们现在面临的这个问题同样如此。STL的强大毋庸置疑，但它同样也是晦涩的，并且使用STL的人一不小心便会写出无法维护、低效、不可移植或干脆让人无法理解的代码。在你编写自己的C++模板库时，可用的技术有多种，于是你一不小心便会掉进自己的风格和技术中，建立只属于自己的私有惯用法。这便是所谓的**方言化**了，方言化会妨碍软件工程师们之间的交流。

另一方面，从库使用者的角度来说，我们也经常会发现需要克服这类方言化行为甚至干脆是糟糕的设计所带来的（可发现性上的）理解困难，并逐渐对使用的库建立起一个让人感到比较舒服的认识。这时你便落进了一个“效用局部最大化”陷阱当中。一方面，有可能你正使用的库的确是最适合手头任务的，并且你也最有效地运用了你的库，但同样可能的是存在另一个库比现在这个有效得多，或者说，现在这个库换种用法会有效得多。你的这种状况便被称为**适应性满足**，即对目前能工作的方案感到满意。适应性满足会导致方言化或/和对好的惯用法的忽视。但由于我们软件工程师并没有无限的时间去寻求给定问题的最佳方案，再加上我们使用的软件工具也在日益庞大和复杂，因此适应性满足几乎是无可避免的。

其实我们每个人都只在复杂性的海洋中舀取了一瓢水，在这个小水洼中我们作为聪明的人类迅速适应并感到舒适。而一旦感到舒适了，我们的工作便不再那么复杂和令人不快，于是便可能逐渐被那些喜欢表达力、效率和灵活性，排斥耦合、低可移植性、低可发现性的人们接纳。最终我们的工作到底传播得多广泛则取决于市场和技术价值。然而无论如何，一旦传播得够广，我们的代码便会被完全接受，人们便开始觉得它正常而简单了，虽然事实上可能未必如此。对此最好的例子也许就是STL本身了。

不管是作为库的作者、用户，还是作者兼用户，我们都需要一些手段来帮助我们避免方言化和适应性满足。这些手段包括惯用法、反惯用法、以及窥视黑盒内部。有经验的程序员往往会忘记，惯用法其实并不是天生就符合直觉的。比如英语中的许多单词的拼写就远远不合直觉，只是我们适应了它们而已。使用鼠标来点击按钮、菜单项和滚动栏也并不直观。同样，整个STL以及C++的大部分内容也都并不是天生就符合直觉的。

例如，C++中的类缺省情况下都是可复制的（指可复制构造加上可赋值）。其实我（以及其他一些人）觉得这一语义是有问题的。而且，防止一个类被复制的手法从任何意义上来讲都并不直观：

```
class NonCopyable
{
```

```
public:// Member Types
 typedef NonCopyable    class_type;
 ...
private:// Not to be implemented
 NonCopyable(class_type const&);
 class_type& operator =(class_type const&);
};
```

这是个广泛认可的做法，广泛到几乎可以说是C++常识了。于是它就成了惯用法。类似的，STL也是一个巨大的惯用法。一旦你对STL有了一定的经验，使用标准库提供的基本STL便习惯成自然了。当STL扩展引入新的概念和实践方式时，便需要新的惯用法来“驯服”它们。

既然有了有效的惯用法，不管是新的还是旧的，便会有“反惯用法”。STL实践者必须对此警惕。例如，将一个迭代器看成指针便是一个反惯用法，你很可能会被它反咬一口。

指出并加强既有的惯用法、描述有用的新惯用法、警惕假冒的/反惯用法、窥视黑盒内部，这些便是本书（及其第二卷）所要采用的战术手法，利用这些手法，我们在成功C++库的七个特征之间保持平衡。

例库

我是个比较务实的人，所以我喜欢的一般都是些通过实际经验传授知识的书。（一上来就直奔抽象，我肯定头大。）因此，书中大部分例子都来源于我自己的工作，特别是其中的几个开源项目。有人可能会觉得我这么说是在骗人，其实真实目的是想给自己的库做广告。我得承认，我的确有这个动机，但这远非我的主要目的。真正的原因是，使用自己的工作，我便心中有数，确切地知道自己在说什么，而后者正是在写书时极端重要的一条。此外，讨论自己工作中的错误和问题不会冒犯任何人或招来官司。而且，如若不信我上面所说的话，你完全可以下载我的库自己看看。

STLSoft

STLSoft是我的得意之作，在过去的大约五年里逐渐进入C++社群。STLSoft不仅完全免费，而且还是可移植的（主要是编译器间可移植，但有些时候在操作系统间也是可移植的），易用的。最后，也是最重要的是，它的效率很高。此外，像我所有的开源库一样，它使用的也是修改版的BSD许可证。

STLSoft的易用性和易扩展性来源于它的两个特点。一是它是由100%的头文件构成的。你只要包含正确的头文件就行了（别忘了将STLSoft添加到你的开发环境的包含路径中）。二是STLSoft中的组件的抽象层次被有意地放低（*吝啬原则*），并尽量避免混用技术和操作系统特性（*简单原则*）。事实上，STLSoft分为多个子项目，各自针对特定的技术领域。

STLSoft提供的许多特性都支持类似STL和非STL的编程风格，但其实其主要目的还是在一个相对较低的抽象层上提供通用的组件和设施，以支持商业项目和其他开源库。总的来说，STLSoft是一个高效的，高灵活性的，高可移植性的，以及低耦合性的库。当必须进行折中时，我选择的是牺牲一些表达力和抽象性来成全以上这些性质。

STLSoft子项目

有意思的是，STLSoft中最主要的子项目也叫STLSoft，其中包含了大部分平台和技术无关的代码；例如分配器和分配器适配器（见卷2），算法（见卷2），迭代器和迭代器适配器（第三部分），内存工具类（utilities），字符串操作函数和类（第27章），用于定义（时间和空间上都很高效的）C++类属性（properties）的类（见《Imperfect C++》第35章），元编程组件（第12，13和41章），垫片（第9章），编译器和标准库特性甄别（和替换），等等。STLSoft子项目中的所有组件都位于sltsoft名字空间内。

另外三个最大的子项目是COMSTL、UNIXSTL和WinSTL，其组件分别位于comstl、unixstl和winstl名字空间内。COMSTL提供了一大堆实用组件，用于COM（Component Object Model，组件对象模型）编程，此外还在COM枚举器和COM集合的概念之上，提供了STL兼容的序列适配器。这些分别