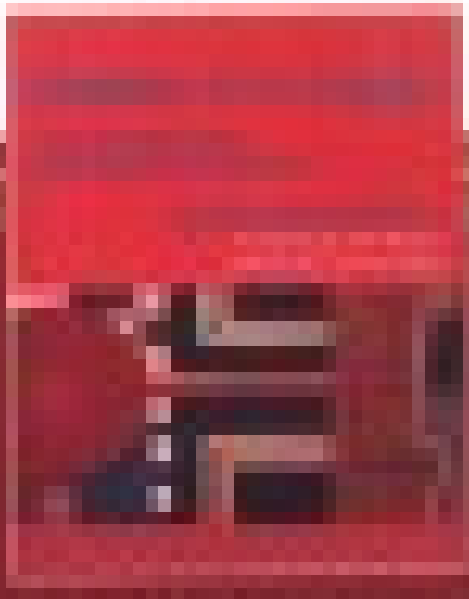
C++设计新思维

泛型编程与设计模式之应用

Modern C++ Design

— Generic Programming and Design Patterns Applied



[美] Andrei Alexandrescu 著

侯捷 於春景 译

华中科技大学出版社

C++设计新思维

泛型编程与设计模式之应用

Modern C++ Design

Generic Programming
and Design Patterns Applied

Andrei Alexandrescu 著

侯捷　於春景　译

C++设计新思维

Modern C++ Design

Andrei Alexandrescu

图书在版编目(CIP)数据

C++设计新思维/(美)Andrei Alexandrescu 著；侯 捷 於春景 译

武汉：华中科技大学出版社，2003 年 3 月

ISBN 7-5609-2906-0

Ⅰ.C…

Ⅱ.①A… ②侯… ③於…

Ⅲ.C 语言-程序设计

Ⅳ.TP312

责任编辑：周 筠(http://yeka.xilubbs.com)

出版发行：华中科技大学出版社 （武昌喻家山 邮编：430074）

录 排：华中科技大学惠友科技文印中心

印 刷：湖北新华印务有限公司

开 本：787×1092 1/16 印 张：21.75 插 页：2 字 数：400 000

版 次：2003 年 3 月第 1 版 印 次：2003 年 3 月第 1 次印刷

印 数：1—8 000 定 价：59.80 元

ISBN 7-5609-2906-0/TP・501

序言

by Scott Meyers

1991 年，我写下《*Effective C++*》第一版。那本书几乎没有讨论 template，因为它刚刚才被加入语言之中，我对它几乎一无所知。为了书中包含的一点点 template 代码，我曾通过电子邮件请别人验证，因为我手上的编译器都没有提供对 template 的支持。

1995 年，我写下《*More Effective C++*》。又一次，我几乎没有讲述 template。这一次阻止我的，既不是对 template 知识的缺乏（在那本书的初稿中，我曾打算以一整章讲述 template），也不是我的编译器在这方面有所缺陷。真正的理由是我担心，C++ 社群对 template 的理解即将经历一次巨大的变化，我对它所说的任何事情，也许很快就会被认为是陈旧的、肤浅的，甚至完全错误的。

我的担心出于两个原因。第一个原因和 John Barton 及 Lee Nackman 在 *C++ Report* 1995 年 1 月的一篇专栏文章有关。这篇文章讨论的是：如何经由 template 执行型别安全的维度分析，同时做到运行期零成本。我自己也曾在这个问题上花了不少时间，而且我知道很多人也在寻找解答，但没有人成功。Barton 和 Nackman 的创新解法让我认识到，template 在太多的地方有用，不只是用来生成“T 容器”。

以下是他们的设计示例。这段代码对两个物理量作乘法运算，而这两个物理量具有任意维数的型别：

```
template<int m1, int l1, int t1, int m2, int l2, int t2>
Physical<m1+m2, l1+l2, t1+t2> operator*(Physical<m1, l1, t1> lhs,
                                        Physical<m2, l2, t2> rhs)
{
  return Physical<m1+m2, l1+l2, t1+t2>::unit*lhs.value()*rhs.value();
}
```

即使我没有说明这段代码，但有一点很清楚：这个 function template 有 6 个参数，可没有一个是型别！template 的这种用法对我来说是头一次见到，我确实有点目眩。

不久之后，我开始阅读 STL。在 Alexander Stepanov 精巧的程序库设计中，容器（containers）对算法（algorithms）一无所知，算法亦对容器一无所知；迭代器（iterator）的行为像指针（但

却有可能是对象）；容器和算法像接受函数指针一样地接受函数对象（function object）；用户可以扩充程序库，但不必继承其中任何 base class，也不必重新定义任何 virtual function。这一切都让我觉得——就像当初我看到Barton和Nackman的成果那样——我对template几乎一无所知。

所以，在《*More Effective* C++》中，我几乎没有提到 template。我还能怎样？我对 template 的认识还停留在“T 容器”阶段，而 Barton、Nackman、Stepanov，还有其他人都已证明，那种用法只不过刚刚触到 template 的皮毛而已。

1998 年，Andrei Alexandrescu 和我开始了电子邮件交流：不久之后我意识到，我得再次修正我对 template 的认识。Barton、Nackman、Stepanov 让我感到震惊的是：template 可以“做什么”；而 Andrei 的成果最初给我的印象是：template“如何”完成它所做的事情。

在 Andrei 协助推广的很多工具中，有这样一个最简单的东西；当我向人们介绍 Andrei 的工作时，我也一直将这当做一个例子。这就是 `CTAssert` template，作用和 `assert` 宏类似，但施行于“可在编译期间被核定（evaluated）”的条件句中。以下便是 `CTAssert` template：

```
template<bool> struct CTAssert;
template<> struct CTAssert<true> {};
```

仅此而已。请注意，这个 `CTAssert` 从来没被定义。请注意，它有一个针对 `true`（而非 `false`）的特化体。在这个设计中，“缺少”的东西至少和提供的东西一样重要。它让你以一种新的方式看待 template，因为大部分“源码”被刻意遗漏了。和我们大多数人以往的想法相比，这是一种极为不同的思维方式。（本书之中 Andrei 讨论了一个更为复杂的 `CompileTimeChecker` template，而不是 `CTAssert`）

后来，Andrei 将注意力转移到 idioms（惯用手法）和 design patterns（设计模式，尤其是 GoF[4] 模式）的开发上，提供了 template-based 实作品。这导致他和模式社群的一场短暂冲突，因为后者信奉一条基本原则：模式（patterns）无法以代码表述。一旦弄清 Andrei 是在致力于使模式的实现得以自动化，而非试图将模式本身以代码来表述，反对声音也就消失了。我很高兴看到，Andrei 和 GoF 之一 John Vlissides 达成了合作；在 *C++ Report* 上，他们就 Andrei 的研究成果推出了两个专栏。

在开发 templates 用以产生 idioms（惯用手法）和 design patterns（设计模式）实作品时，所有实作者需要面对的各种设计抉择，Andrei 也都必须面对。代码应该做到多线程安全吗？辅助存储器应当来自 heap 或是 stack 抑或 static pool？提领 smart pointers 之前是否应该针对 null 进行检查？程序关闭时如果 Singleton 的析构函数试图使用另一个已被摧毁的 Singleton，会发生什么事？Andrei 的目标是：为用户提供所有可能的设计选择，但不强制任何东西。

---

[4] "GoF" 意味着"Gang of Four"，指的是 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides，他们四人是模式（patterns）权威书籍《*Design Patterns: Elements of Resusable Object-Oriented Software*》（Addison-Wesley, 1995）的作者。

Andrei 的方案是：将这种选择以 policy classes 的形式封装起来，允许客户将 policy classes 当做 template 参数传递，同时为这种 classes 提供合理的缺省值，使大多数客户可以忽略这些参数。其结果令人瞠目结舌。例如本书的 SmartPointer template 只有 4 个 policy 参数，但它可以生成 300 多个不同的 smart pointer 型别，每一个都具有不同的行为特征！满足于 smart pointer 缺省行为的程序员可以忽略 policy 参数，只需指定“smart pointer 所指对象”之型别，从而获得精心制作的 smart pointer class 所带来的好处。嗯，不费吹灰之力。

最后要说的是，本书叙述了三个不同的技术故事，每一个都引人入胜。首先，它就 C++ template 的能力和灵活性提供了新的见解——如果第三章的 typelists 没有让你感到振奋，一定是因为你暮气沉沉）。第二，它标示了一个正交维度（orthogonal dimensions），告诉我们 idioms 和 patterns 的实现可以不同。对 templates 设计者和 patterns 实作者而言，这是十分重要的资讯，但是在大多数讲述 idioms 或 patterns 的文献中你都找不到这方面的研究。第三，Loki（本书介绍的 template library）源码可以免费下载，所以你可以研究 Andrei 讨论的 idioms 和 patterns 所对应的 template 实际作品。这些代码可以严格检测你的编译器对 templates 的支持程度，此外当你开始自己的 template 设计时，它还是无价的起点。当然，直接使用 Loki 也是完全可以的（而且完全合法），我知道 Andrei 也愿意你运用他的成果。

就我所知，template 的世界还在变化，速度之快就像 1995 年我回避写它的时候一样。从发展的速度来看，我可能永远不会写有关 template 的技术书籍。幸运的是一些人比我勇敢，Andrei 就是这样一位先锋。我想你会从此书得到很多收获。我自己就得到了很多。

Scott Meyers
September 2000

前卫的意义

侯捷译序

一般人对 C++ templates 的粗浅印象，大约停留在“容器（containers）”的制作上。稍有研究则会发现，templates 衍生出来的 C++ Generic Programming（泛型编程）技术，在 C++ 标准程序库中已经遍地开花结果。以 STL 为重要骨干的 C++ 标准程序库，将 templates 广泛运用于容器（containers）、算法（algorithms）、仿函数（functors）、配接器（adapters）、分配器（allocators）、迭代器（iterators）上，无处不在，无役不与，乃至于原有的 class-based iostream 都被改写为 template-based iostream。

彻底研究过 STL 源码（SGI 版本）的我，原以为从此所有 C++ templates 技法都将不出我的理解与经验。但是《*Modern C++ Design*》在在打破了我的想法与自信。这本书所谈的 template 技巧，以及据以实作出来的 Loki 程序库，让我瞠目结舌，陷入沉思…与…呃…恍惚☺。

本书分为两大部分。首先（第一篇）是基础技术的讨论，包括 template template parameters（请别怀疑，我并没有多写一个字）、policies-based design、compile-time programming、recursive templates, typelists。每一项技术都让人闻所未闻，见所未见。

第二部分（第二篇）是 Loki 程序库的产品设计与实现，包括 Small-Object Allocation[1], Generalization Functors, Singleton, Smart Pointers, Object Factories, Abstract Factory, Visitor, Multimethods。对设计模式[2]（**design patterns**）稍有涉猎的读者马上可以看出，这一部分主题都是知名的模式。换言之，作者 Andrei 尝试以 templates-based, policies-based 手法，运用第一篇完成的基础建设，将上述模式具体实现出来，使任何人能够轻松地在 Loki 程序库的基础上，享受设计模式所带来的优雅架构。

---

[1] Small-Object Allocation 属于底层服务的“无名英雄”，故而在章节组织上仍被划入第一篇。

[2] **patterns** 一词，台湾大陆两地共出现三种译法：(1) 范式，(2) 样式，(3) 模式。我个人最喜欢“范式”，足以说明 patterns 的“典范”意味。因此，繁体版以“范式”称 patterns。为尊重大陆术语习惯，简体版以“模式”称 patterns。本书所有 patterns 都保留英文名称并以特殊字型标示，例如 Object Factories, Visitors...

设计模式（Design Patterns）究竟能不能被做成“易拉罐”让人随时随地喝上一口，增强体力？显然模式社群（patterns community）中有些人不这么认为——见稍后 Scott Meyers 序文描述。我以为，论断事物不由本质，尽好口舌之辩的人，不足取也。Andrei 所拓展的天地，Loki 所达到的高度，不会因为它叫什么名字而有差异，也不会因为任何人加诸它身上的什么文字包装或批评或解释或讨好，而有不同。它，已经在那儿了。

本书涉足无人履踏之境，不但将 C++ templates 和 generics programming 技术做了史无前例的推进，又与 design patterns 达成巧妙的结合。本书所谈的技术，所完成的实际产品，究竟是狂热激进的象牙塔钻研？抑或高度实用的崭新设计思维？作为一个技术先锋，Loki 的现实价值与未来，唯赖你的判断，和时间的筛选。

然而我一定要多说一句，算是对“唯实用论”的朋友们一些忠告。由来技术的推演，并不只是问一句“它有用吗”或“它现在有用吗”可以论断价值的。牛顿发表万有引力公式，并不知道三百年后人们用来计算轨道、登陆月球。即使在讲述“STL 运用”的课堂上，都还有人觉得太前卫，期盼却焦躁不安，遑论“STL 设计思维和内部实作”这种课，遑论 Loki 这般前卫技术。很多人的焦虑是：我这么学这么做这么写这么用，同侪大概看不懂吧，大概跟不上吧。此固值得关注，但个人的成长千万别被群体的惯性绊住脚步[3]。我们曾经鄙夷的别人的“无谓”前卫，可能只因我们故步自封，陷自己于一成不变的行为模式；或因为我们只看到自家井口的天空。当然，也可能某些前卫思想和技术，确实超越了庞大笨重迟缓的现实世界的接受度。你有选择。作为一位理性思考者，身在单纯可爱的技术圈内，请不要妄评先锋，因为他实在站在远比你（我）高得太多的山巅上。不当的言语和文字并不能为你（我）堆砌楼台使与同高。

深度 + 广度，古典 + 前卫，理论 + 应用，实验室 + 工厂，才能构筑一个不断进步的世界。

侯捷 2003/01/08 于 台湾·新竹

jjhou@ccca.nctu.edu.tw
http://www.jjhou.com（繁）
http://jjhou.csdn.net（简）

P.S. 本书译稿由我和於春景先生共同完成。春景负责初译，我负责其余一切。春景技术到位，译笔极好，初译稿便有极佳品质，减轻我的许多负担。循此以往必成为第一流 IT 技术译家。我很高兴和他共同完成这部作品。本书由我定稿，责任在我身上，勘误表由我负责。本书同步发行繁体版和简体版；基于两岸计算机术语的差异，简体版由春景负责必要转换。

P.S. 本书初译稿前三章，邱铭彰先生出力甚多，特此致谢。

P.S. STL, Boost, Loki, ACE...等程序库的发展，为 C++ 领域挹注了极大活力和竞争力，也使泛型技术在 C++ 领域有极耀眼的发展。这是 C++ 社群近年来最令人兴奋的事。如果你在 C++ 环境下工作，也许这值得你密切关注。

---

[3] 从万有引力观之，微小粒子难逃巨大质量团的吸滞（除非小粒子拥有高能量）。映照人生，这或许是一种悲哀。不过总会有那么一些高能粒子逸脱出来——值得我们转悲为喜，怀抱希望。

译序

by 於春景

三年前，当我第一次接触 template 的时候，我认为那只不过是一位“戴上了新帽子”的旧朋友：在熟悉的 class 或 function 的头顶上，你只需扣上那顶古怪的尖角帽——添上一句 `template <class T1, ...>`——然后将熟悉的数据型别替换为 `T1, T2...`，一个 template 就摇身而至！嗯，我得承认，戴上了帽子的 template 的确是个出色的代码生成器，好似具有滋生代码“魔法”的 macro，但毕竟还不能成其为“戴上了帽子的魔术师”。

后来，我开始学习 GP（generic programming）和 STL（standard template library）。我不禁哑然。在 GP 领域，template 竟扮演着如此重要的角色，以至于成为 C++ GP 的基石。在 GP 最重要的商业实作品 STL 中，template 向我们展示其无与伦比的功效。回想起自己当初对 template 的比喻，哑然失笑之余，我惊叹 template 在 GP 和 STL 中将自己的能力发挥到了“极致”。

然而，这一次，《*Modern C++ Design*》又让我默然。我不得不承认，Andrei Alexandrescu 的这部著作（及其 Loki library）带给我的，是对 template 和 GP 技术又一次震撼般的认识！

这种震撼感受，源于技术层面，触及设计范畴。template 的技术核心在于编译期动态机制。与运行期多态（runtime polymorphism）相比，这种动态机制提供的编译期多态特性，给了程序运行期无可比拟的效率优势。本书中，Andrei 对 template 编译期动态机制的运用可谓淋漓尽致。以 template 打造而成的 typelist、small-object allocator、smart pointer 不仅具有强大功能，而且体现了无限的扩充性；将 template 技术大胆地运用到 design patterns 中，更为 design patterns 的实现提供了灵活、可复用的泛型组件。

在这些令人目眩的实作技术之后，蕴涵着 Andrei 倡导并使用的 policy-based 设计技术。利用这一耳目一新的设计思想，用户代码不再仅仅是技术实作上的细节，你甚至可以让代码在编译期作出设计方案的选择！这种将“设计概念”和“template 编译期多态”结合起来的设计思维，将 C++ 程序设计技术提升到了新的高度，足以振聋发聩。

也许只有时间才能证实，Andrei 为我们展示的，或许是 C++ 程序设计技术的一次革命；在 C++ 的历史上，《*Modern C++ Design*》将是一部重要的著作。Andrei 对 template、generic programming 技术，以及 template 在 design patterns 中的运用等课题所做的深入阐释和大胆实践，可谓前无古人。

遗憾的是，在当今主流 C++ 编译器上，Loki 很难顺利通过编译。例如面对 "template template parameter" 的“难题”，很多编译器毫无招架之力。应该说，这并不是 Andrei 和 Loki 过于超前，而是 C++ 编译器应当迅速跟进。这意味着作为 C++ 程序员的你我，也应当迅速跟进！

作为 C++ 程序员的我，已从此书获益良多。这是一部让我在翻译过程中毫不感到倦怠的巨著。它时时引发我思索，给我以启迪，并让我重拾研习 C++ 的快乐。这得感谢 Andrei。在这样一部讲述高级技术的专著中，Andrei 的讲解细致深入，条理得当，语言却又极为简明清晰。我期望中文版能保留这一特色。

除了作者之外，在翻译本书的过程中，给我更多教益的还有侯捷先生。我的初译稿便是在先生不断的鼓励和指导下完成的。先生谦和的人品和技术上的深邃见解，令我钦佩和谨记。还要感谢周筠（yeka）编辑，我的每一本译作都离不开您的参与和悉心帮助，本书也不例外。最后，感谢所有关心我的朋友，愿你们也像我一样喜爱这本书。

於春景 2002/12/15
深圳蛇口，海上世界
billyu@lostmouse.net

序言

by John Vlissides

关于 C++，还有什么没有说到的？唔，很多，本书所谈的一切几乎都是。本书提供的是编程技术——generic programming、template metaprogramming、OO programming、design patterns——的融合。这些技术分开来可以有良好的理解，但对于它们之间的协作关系，我们才刚刚开始认识。这些协同作用为 C++ 打开了全新视野，而且不仅仅在编程方面，还在于软件设计本身；对软件分析和软件体系结构来说，它也具有丰富的内涵。

Andrei 的泛型组件将抽象层次提升到了新的高度，足以使 C++ 在各方面看起来像是一种设计规格（design specification）语言。但是，不同于专用的设计语言，你还保有 C++ 全部的表达性和对它的驾轻就熟。Andrei 向你展示如何根据设计思想——singletons、visitors、proxies、abstract factories... ——来编写程序。甚至你可以经由 template 参数改变实作选择，而且几乎没有运行期开销。你不必求助于新的开发工具，也不必学习晦涩难懂的方法学（methodology）。你需要的只是一个可靠的、新型的 C++ 编译器，以及本书。

多年来，代码生成器（code generators）一直有类似承诺，但我自己的研究以及实践经验使我相信，最终，代码生成器无法匹敌。你会有“往返旅程（round-trip）”问题，“缺乏值得生成的代码”问题，“生成器不灵活”问题，“生出莫名其妙的代码”问题，当然还有“无法将自己的代码和该死的生成出来的代码整合在一起”的问题。这些问题中的任何一个都有可能成为绊脚石；而且，对大多数编程挑战而言，这些绊脚石都使得“代码自动生成”不可能成为一种解决方案。

如果能获得“代码自动生成”理论上的好处——快速、易开发、冗余降低、错误更少——而又没有它们的缺点，该有多好！这正是 Andrei 的做法所承诺的。在易于使用、可相互混合和匹配的 templates 中，泛型组件实现了出色的设计。它们完成的几乎就是代码生成器的功能：产生供编译器使用的规范代码（boilerplate code）。差别在于它们是在 C++ 之内（而非之外）完成这些功能。成果是“与应用代码的无缝整合”。同时你还是可以运用 C++ 语言的全部威力，对设计进行扩充、改写或者调整，从而符合你的需要。

无可否认，这里的一些技术很复杂，因而难以领会，特别是第 3 章的 template metaprogramming 部分。但是一旦你掌握了它，你就奠定了泛型组件架构（generic componentry）的坚实基础；后续章节中的各个泛型组件几乎就是自己构造自己。事实上我认为第 3 章关于 template metaprogramming 的内容就值回本书的价格，何况还有另外 10 个充满见地、让你获益匪浅的章节。"10" 其实是代表一个数量级。而我确信，你获得的回报会比这个数量还多得多。

John Vlissides
IBM T.J. Watson Research
September 2000

前言
Preface

也许你正在书店里捧着这本书，问自己该不该买下它。或者，你正在公司的图书室里，犹豫该不该花时间阅读它。我知道你时间宝贵，所以我开门见山。如果你曾经问过自己：如何撰写更高级的 C++ 程序？如何应付即使在很干净的设计中仍然像雪崩一样发生的不相干细节？如何构建可复用组件，使得每次将这些组件应用到下一个程序时都无需对它们大动干戈？如果你曾这样问过自己，那么，本书正是为你所写。

想象这样的情景。你刚从一次设计会议回来，带着一些打印图表，上面有你潦草写下的注解。哦，对象之间传递的事件型别（event type）不再是 char 而是 int 了，于是你修改一行代码。指向 Widget 的 smart pointers 太慢了，得取消一些检查措施，让它们快一点，于是你又修改一行代码。另一个部门刚才添加 Gadget class，你的 object factory 必须支持它，于是你再次改动一行代码。

你修改了这个设计。编译，链接，搞定。

且慢，场景有点问题，不是吗？现实情形更可能是：你匆匆从会议中赶回来，因为有一大堆工作要做。于是你开始地毯式搜索，并在代码上大动干戈：添加新的代码、引入臭虫、消除臭虫…这就是程序员的生活，不是吗？本书也许不能保证你实现第一场景，但它朝着那个方向迈出坚实的一步。它对软件设计师展示的 C++，宛如一种新语言。

传统上，代码是软件系统中最琐碎、最复杂的环节。尽管历史上出现了各种层次的编程语言，支持各种设计方法（譬如面向对象方法），但在蓝图和代码之间，总是横亘着一条鸿沟。这是因为，代码必须仔细关照具体实现和某些辅助性任务中极其细节的问题。因此，设计意图往往被无尽的细节吞噬。

本书提供了一组可复用的设计产品——所谓“泛型组件”，以及设计这些组件所需要的技术。这些泛型组件为用户带来的明显好处，集中于程序库方面，而处于更广泛的系统体系结构空间中。本书提供的编程技术和实作品（implementation）所反映的任务和议题，传统上属于设计范

畴之中，是编写代码之前必须完成的东西。由于身处较高层次，泛型组件就有可能以一种不同寻常但简洁、易于表达、易于维护的方式，将复杂的体系结构反映到代码中。

这里结合了三个要素：设计模式（design patterns）、泛型编程（generic programming）、C++。结合这些要素后，我们获得极高层次的可复用性，无论是横向或纵向。从横向空间来看，少量 library code 就可以实现组合性的、实质上具有无穷数量的结构和行为。从横向空间来看，由于这些组件的通用性，它们可广泛应用于各种程序中。

本书极大地归功于设计模式（design patterns）——面临面向对象程序开发中的常见问题时，它是强有力的解决方案。设计模式是经过提炼的出色设计方法，对于很多情况下碰到的问题，它都是合理而可复用的解决方案。设计模式致力于提供深具启发、易于表达和传递的设计词汇。它们所描述的，除了问题（problem）之外，还有久经考验的解法及其变化形式，以及选择每一种方案所带来的后果。设计模式超越了任何一种设计语言所能表达的东西——无论那种语言多么高级。本书遵循并结合某些设计模式，提供的组件可以解决广泛的具体问题。

泛型编程是一种典范（paradigm），专注于将型别（type）抽象化，形成功能需求方面的一个精细集合，并利用这些需求来实现算法。由于算法为其所操作的型别定义了严格、精细的接口，因此，相同的算法可以运用于广泛的型别集（a wide collection of types）。本书提供的实作品采取泛型编程技术，以最小代价获得足以和手工精心编写的代码相匹敌的专用性、高度简洁和效率。

C++ 是本书使用的唯一工具。在本书中，你不会看到漂亮的窗口系统、复杂的网络程序库或灵巧的日志记录（logging）机制。相反的，你会发现很多基础组件；这些组件易于实现以上所有系统（甚至更多）。C++ 具有实现这一切所需要的广度，其底层的 C 内存模型保证了最原始效率（raw performance），对多态（polymorphism）的支持成就了面向对象技术，templates 则展现为一种令人难以置信的代码生成器。Templates 遍及本书所有代码，因为它们可以令用户和程序库之间保持最密切的协作。在遵循程序库约束的基础上，程序库的用户可以完全控制代码的生成方式。泛型组件库的角色在于，它可以让用户指定的型别和行为，与泛型组件结合起来，形成合理的设计。由于所采技术之静态特性，在结合和匹配相应组件时，产生的错误通常在编译期便得以发现。

本书最明显的意图在于创建泛型组件，这些组件预先实现了设计模块，主要特点是灵活、通用、易用。泛型组件并不构成 framework。实际上它们采用的做法是互补性的；虽然 framework 定义了独立的 classes，用来支持特定的对象模型，但泛型组件(s) 是轻量级设计工具，互相独立，可自由组合和匹配。实现 frameworks 时泛型组件可带来很大帮助。

本书读者

本书预定的读者主要分为两类。第一类是富有经验的 C++ 程序员，他们希望经由本书掌握最先进的程序库编写技术。本书提供了新而强大的 C++ 技术，这些技术具有惊人能力，有一些甚至令人匪夷所思。这些技术对于撰写高级程序库极有帮助。当然，希望更上层楼的中阶 C++程序员也会发现本书十分有益，特别是如果他们愿意付出一些毅力。本书虽然有时给出一些高难度的 C++ 代码，但都有详尽说明。

本书预定的第二类读者是繁忙的程序员，他们需要完成工作，但无法投入时间进行深入学习。他们可以快速略过最复杂的细节，把注意力放在如何使用本书提供的程序库上。每一章都有一个导入说明，并以概览（Quick Facts）结束。程序员们会发现这种安排方式为理解和使用本书组件提供了有益的参考。这些组件可以分开理解，它们功能强大但很安全，而且让人乐于使用。

你需要扎实的 C++ 经验，以及强烈的求知欲。你也需要对 templates 和 STL（Standard Template Library）有一定的掌握。

如果你已经了解 design patterns（Gamma 等著, 1995），那当然好，但并非必要。书中对于用到的 patterns 和 idioms（惯用手法）都有详细介绍。本书并不是 patterns 方面的专著，并不试图完整阐述 patterns。由于 patterns 是程序库设计者从实践的角度提出的，所以即使那些曾经关注 patterns 的读者也会发现，他们的视野如今有了更新——如果他们曾经受到束缚的话。

Loki

本书讲述一个实际的 C++ 程序库，称为 Loki。Loki 是挪威神话中的智慧之神，同时也是一个淘气鬼；作者希望，这个程序库的创意和灵活会让你想起那个有趣的挪威神话人物。程序库中的所有元素都位于名字空间（namespace）`Loki` 之内；这一名称并未出现于书中范例程序上，因为那会为代码带来非必要的缩排格式，并增加代码的数量。Loki 是免费的，你可以从 http://www.awl.com/cseng/titles/0-201-70431-5 下载它。

除了线程（threading）部分，Loki 完全以标准 C++ 写成。唉，这也意味着目前很多编译器无法处理其中的某些部分。我在 Metrowerks CodeWarrior Pro 6.0 和 Comeau C++ 4.2.38 上实作并测试过 Loki，并且都在 Windows 平台上。KAI C++ 处理 Loki 代码好象也没有问题。随着供应商逐渐发行更新更好的编译器，你将能够运用 Loki 提供的所有功能。

本书提供的 Loki 代码和范例采用了一种很普及的写码标准，这一标准最早由 Herb Sutter 倡导。我相信你很快便能适应它。简单地说：

- classes、functions、枚举型别（enumerated type）看起来像 `LikeThis`。
  （译注：由于版面上的需要，中译本的 functions 看起来像 `LikeThis`）
- 变量和枚举值看起来像 `likeThis`。
- 成员变量看起来如 `likeThis_`。
- template 参数如果只可能是用户自定义型别，那么它会被声明为 class；如果还可能是基本型

别，那么它会被声明为 typename。

内容组织

本书由两大篇组成：技术篇和组件篇。第一篇（1~4 章）讲述的是，在泛型编程中——特别是在泛型组件的构造中——所运用的 C++ 技术。它展示了与 C++ 相关的大量功能和技术：policy-based 设计、partial template specialization、typelists、local classes 等等。你可以按部就班地阅读本篇，然后回过头来参考特定章节。

第二篇建立在第一篇的基础上，实作出多个泛型组件。这些并非纸上谈兵；他们是具有工业强度的组件，可应用于现实世界的应用程序中。C++ 开发者在日常工作中经常遇到的议题，例如 smart pointers、object factories、functor objects，在此都有深入的探讨，并提供泛型实作。文中提供的实作品满足了基本需要，解决了基本问题。本书并不讲述这一块那一块代码做些什么，它采行的方法是：讨论问题，选择设计决策，然后逐步实现这些设计决策。

第 1 章提供的是 policies，一种有助于产生灵活设计的 C++ 技巧。

第 2 章讨论和泛型编程有关的通用 C++ 技巧。

第 3 章实作 typelists，一种功能强大、用于操纵型别的数据结构。

第 4 章介绍一个重要的辅助工具：小型对象分配器。

第 5 章介绍泛化仿函数的概念；在运用 Command 模式的设计中，它很有用处。

第 6 章讲述 Singleton 对象。

第 7 章讨论和实现了 Smart Pointers。

第 8 章讲述 generic Object Factories.

第 9 章探讨 Abstract Factory 设计模式，并提供一份实作品。

第 10 章以泛型方式实现了 Visitor 设计模式的几个变型。

第 11 章实现了数个 MutiMethod 引擎；这些方案体现设计上的各种选择。

“设计”涵盖许多重要工作，C++ 程序员必须以规律的、标准的、合格的基础和准则来对付。我个人认为 Object Factories（第 8 章）是所有高品质多态设计（polymorphic design）的基石。Smart Pointers（第 7 章）是大大小小许多 C++ 应用程序的重要组件。Generalized Functors（第 5 章）有极为宽广的应用，一旦你拥有它，许多复杂的设计问题都能够迎刃而解。其他更特殊的泛型组件，例如 Visitor（第 10 章）或 MultiMethod（第 11 章），也都有重要而合适的应用，并将语言的支持推向极致。

致谢

Acknowledgements

我要感谢我的父母，他们勤劳地度过了那段最为漫长艰辛的岁月。

我要特别强调的是，这本书，连同我的大部分职业生涯，如果没有 Scott Meyers，就都不会存在。自 1998 年于 C++ World Conference 结识 Scott 以来，他就一直帮助我，使我做得更多更好。Scott 第一个热情地鼓励我，让我将我的早期想法付诸实践。他将我引荐给 John Vlissides，促成了另一个具有丰硕成果的合作；他说动 Herb Sutter，让我成为 *C++ Report* 的专栏作家；他将我介绍给 Addison-Wesley 出版公司，实质上强迫着我开始这本书的写作，而那时我对纽约的销售人员一点都不了解。整本书的创作过程中，Scott 一直帮助我，给我审阅和建议，和我分享写作的痛苦，但没有得到任何好处。

多谢 John Vlissides，他不但提出深邃的见解，让我相信我的方案中存在问题，还为我提出了更好的方案。第 9 章之所以存在，正是因为 John 坚持“事情可以做得更好”。

感谢 P.J. Plauger 和 Marc Briand，他们鼓励我为 *C/C++ User Journal* 撰写文章，那时我以为专栏作家是外星人。

感谢我的编辑 Debbie Lafferty，她给了我不断的支持，并提出敏锐的建议。

我在 RealNetworks 的同事，特别是 Boris Jerkunica 和 Jim Knaack，给我很大的帮助；他们为我营造了自由、竞争、向上的气氛。我为此感谢他们。

我也十分感激 comp.lang.c++.moderated 和 comp.std.c++ Usenet 新闻群组的所有参与者。这些朋友极大地促进了我对 C++ 的认识。

我还要把我的感谢献给本书初稿审阅者：Mihail Antonescu, Bob Archer（书稿的最完整审阅者），Allen Broadman, Ionut Burete, Mirel Chirita, Steve Clamage, James O. Coplien, Doug Hazen, Kevlin Henney, John Hickin, Howard Hinnant, Sorin Jianu, Zoltan Kormos, James Kuyper, Lisa Lippincott, Jonathan H. Lundquist, Petru Marginenean, Patrick McKillen, Florin Mihaila, Sorin Oprea, John potter,