Imperfect C++ 中文版

——经过实践检验的现实编程解决方案

Matthew Wilson 著

荣耀 刘未鹏 译

序言

——不完美主义实践者的哲学

在本书中，C++语言技术与良好的实践方式占有同等重要的地位。本书并非仅仅讨论在某个特定场合下什么方案才是有效的或技术上正确的，更重要的还是要看最终哪种做法才是更安全的或者更切合实际的。本书要传达的意思有四个方面：

原则 1——C++是卓越的，但并不完美。
原则 2——穿上“苦行衣”。
原则 3——让编译器成为你的仆从。
原则 4——永不言弃，总会有解决方案的。

这“四项基本原则”构成了我所谓的“不完美主义实践者”的哲学。

C++并不完美

多年以前，一位为她最小的儿子的过分骄傲的心理感到不安的母亲曾这样教导说：“如果你打算将一些好的东西告诉别人，那么你最好也准备承认其中那些糟糕的成分。”谢谢你，母亲！

C++是一门杰出的语言。它支持高阶概念，包括基于接口的设计、泛型、多态、自描述的软件组件以及元编程（meta-programming）等。此外，凭借对低阶特性的支持，它在提供对计算机的精细控制方面，包括位操作、指针以及联合（union）等，也比大多数语言更有能耐。借助于这些范围宽广的能力，外加保持的对高效性的根本性支持，C++可以说是当今最杰出的通用编程语言[注 1]。不过话说回来，C++并非完美无瑕，实际上远没到完美的程度，因而有了本书的名字——“Imperfect C++”（中文版）。

[注 1]请注意，我并不是说 C++在所有特定问题领域都是最佳语言。例如，我绝不会建议你使用 C++去编写“专家系统”，Prolog才是这方面的合适人选。而在系统脚本方面 Python和 Ruby则当仁不让。此外，我们还得承认 Java在企业级电子商务系统开发中确有过人之处。

由于一些很好的原因（有些是历史原因，也有些是当前的原因），C++不仅是一个折衷[Stro1994]的产物，而且还是一些互不相干、有时甚至是互不兼容的概念的混合体，因而其中必然存在着一些缺陷。有些缺陷只是鸡毛蒜皮，但有些就不是那么无关紧要了。许多缺陷都是从它们的“祖辈”那儿承袭而来的。其他则是由于语言将效率放在高优先级（幸好如此）才导致的。有些则可能是任何语言都无法摆脱的根本限制。正是由于如今的语言变得愈来愈复杂，也愈来愈多种多样，因而才出现了一些极有趣的问题，这些是任何人都始料未及的。

本书直面这种复杂的形势，坚信总能够克服复杂性，坚信控制权最终还是掌握在那些见多识广、经验丰富的计算机专家手中。我的目标是缓解那些使用 C++的软件开发者日常经受的不知所措以及无法作出决断的痛苦之情。

本书致力于解决的并不是软件开发者因经验不足或知识不够而遇到的问题，而是这个职业中的所有成员，从初学者甚至到最有才干和经验的那些人，所共同遭遇的问题。这些问题中部分源于语言自身固有的不完美性，部分源于人们对语言所支持的一些概念的常见误用。无论如何，它们给我们所有人带来了麻烦。

本书并不仅仅是对语言中的不完美之处进行一些简单的论述并附带一些“别这么做”列表，你可以找到很多着眼于这方面的 C++书籍。和它们不同，本书的重点在于如何为我所指出的缺陷（中的大部分）提供解决方案，并借此使得这门语言变得不再像它本来那么“不完美”。本书重在赋予开发者能量，讨论他们赖以谋生的手段——C++——中潜在的问题领域，给出了一些重要的相关信息，并为开发者提供了一些建议以及经过实践检验的技巧和技术，以帮助他们避免或应付这些问题。

苦行僧式编程

在我们阅读过的很多教科书中，即便是非常好的，也只是告诉我们 C++能够提供的解决问题的手段，前提还得是你必须将 C++中的有关特性有效地利用起来。然而，这些书常常又会在后面接着说道：“这样做其实没有实质性的意义”或者“严格来说那样做就稍微有点过头了”。不止一个以前的同事曾把我拖进类似的激烈争论之中。通常人们的理由可以归结为：“我是一个有经验的程序员，我并不会犯 XYZ 阻止我犯的那些错误，干嘛要为之烦神呢？”。

唉！

这个论点简直不堪一驳。我也是个有经验的程序员，但我每天至少会犯一个低级错误，假如不是养成了严格的习惯的话，则会是一天十个！他们的这种态度其实就是在假定他们的代码永远都不会被没有经验的程序员看到。此外，他们的说法要得以成立，等于是在说代码作者永远也不会学习或者改变观念、习惯以及方法论。最后一点，到底怎样才算是“有经验的程序员”呢？[注 2]

[注 2]如今这个问题似乎变得没有定论，因为你看到每个人的履历表上的自我评价全都是 10 分。

上面提到的这类人不喜欢引用、常量成员、访问控制、explicit、具体类（concrete classes）、封装、不变式，他们甚至在编码时根本就无视可移植性或可维护性。但他们就是喜欢重载、重写（override）、隐式转换、C 风格强制（C-style casts），并到处使用 int。他们还喜欢全局变量、混合式 typedef、

dynamic_cast、RTTI、专有的编译器扩展以及友元。他们在编码风格上总是不一致，让事情看起来似乎比原本还要复杂。

请容许我暂时扯开话题，讲述一个历史典故。自从1162年被亨利二世封为坎特伯雷大主教之后，Thomas A Becket就经历了一场人格上的转变，从原先物质主义的生活中改过自新，真正开始关心起贫穷的人们，并为他以前的无度行为深刻地进行了忏悔。后来人们在准备埋葬他的躯体的时候，发现他穿着一件粗糙的、爬满跳蚤的苦行衣。事后人们才知道原来他生前天天被修道士们鞭打。天哪！

我个人觉得为了忏悔和净化自己的灵魂，这种做法未免有点过了。不过，回想起当初自己滥用C++的强大能力干了许多糟糕的事情（见附录B），如今我尝试采用一种更有节制的做法，因此就有了“在编程时穿上苦行衣”这一说法[注3]。

[注3]如果对你们来说苦行衣的比喻有点极端，不合你的口味，你也可以和瑜珈类比，即做起来辛苦，但获得的回报相比付出而言还是值得的。

当然，我并不是说得像头悬梁、锥刺股那般，也不是说我在编码时不再听开得震天响的舞曲，都不是，我只是说，我会尽我所能让我的软件来严厉地对待我，以便在我企图去误用它时将我打住。我喜欢const，许多const，我会在任何可能的地方使用它。我尽可能使用private，我优先使用引用（references）。我尽可能实施不变式。我将资源返还给它的出处，即便我知道有其他安全的“捷径”：“呃，你知道的，它在该操作系统以前的版本上可是一点问题也没有，你更新系统那是你的问题，跟我可没关系！”。我使用所谓的“概念性typedef”来增强C++类型检查。我使用九种编译器，并借助于一个工具（见附录C），让它们的使用变得更加简便直观。我还使用了一个更为有效的NULL。

我并不是为了获得“年度程序员”奖项提名才这么做的。这些只不过是我一直以来懒惰所导致的结果。所有优秀的工程师都有懒惰的习惯。懒惰意味着你不想在运行期查错；懒惰意味着你不想第二次犯同样的错误；懒惰意味着尽可能地榨取你的编译器的能力，这样你才得以清闲。

让编译器成为你的仆从

“batman”一词起源于大英帝国时代，意指勤务兵或个人仆从。如果你能够正确地对待编译器的话，你就可以让编译器成为你的左右手，或者说仆从（或者你的“超级英雄”，随你喜欢）。

你的编程苦行衣越粗糙，你的编译器就能够为你服务得越周到。然而，有时你的编译器对语言尽心尽责的行为也会反过来妨碍你的意图，顽固地拒绝履行你知道是合理的（或者至少是你想要完成的）事情。

这本书将会为你提供一些技巧和技术，通过它们你得以从编译器那儿夺回控制权，并作出最终决策：获取你想要的而不是你被给予的。这件工作并不轻松，但我们得承认，软件开发者才是开发过程中的主宰，而语言、编译器和库只不过是被我们所驱使的工具而已。

永不言败

尽管我接受的大部分教育都属于理科方面的，但实际上我更是一个工程师。我喜欢那些早期的科幻小说，其中的那些英勇的工程师总能在面临棘手的情况时找到出路。这也正是本书所采用的方式。理论是有的，而且我们一开始也是跟着理论走。但每当我们在这门语言的边缘“行走”时，我们总会发现当前大多数编译器都不是很遵从理论，因此我们必须在这一现实的约束下编程。正如 Yogi Berra 所说的，“从理论上说，理论与实践是没有任何区别的，但从实践的角度来说，它们之间区别就大了。”

这种看待问题的方式能够为我们带来强大的效果。工程师的努力（而不是理论上的归纳），再加上顽强地和 C++中的不完美之处抗争，使我最终获得以下一系列的发现：

- 借助于垫片（Shims）（第 29 章）以及垫片所带来的类型隧道（Type Tunneling）机制（第 34 章）进行显式泛化（explicit generalization）。
- 对 C++类型系统进行了一些扩展（第 18 章），使我们能够将概念上互不相干、但基于同一基类型（base type）实现的类型彼此区分开，并能够基于它们进行函数重载。
- 一个编译器无关的机制，提供了动态加载的 C++对象之间的二进制兼容性（第 8 章）。
- 在 C++规则下重现 C 中强大的 NULL，同时不违背后者的精神（第 15 章）。
- 一个具有最大限度的“安全性”且可移植的 operator bool()（第 24 章）。
- 对“封装”概念的精细分类，借此我们得以对基本的数据结构进行高效地表示和操纵，并扩充我们的“不完美工具箱”（第 2、3 两章）。
- 一个用于高效地分配动态大小的内存块的灵活工具（第 32 章）。
- 一个能够进行快速、非侵入式的串拼接操作的机制（第 25 章）。
- 对“如何编写能够在不同的错误处理模型下工作的代码”这一问题的中肯评价（第 19 章）。
- 一个用于控制单件对象（singleton objects）的顺序性的直观机制（第 11 章）。
- 一个在时间和空间上均高效的 C++属性（property）的实现（第 35 章）。

本书并不打算成为使用 C++语言的完备指南，它只是想帮助开发者挣脱现实中的约束，以便找到应对那些不完美之处的解决方案，并鼓励以一种超越常规的思考方式来考虑问题。

当然，我本人也绝非完美，正所谓“金无足赤，人无完人”。我也干过蠢事，而且我还有点离经叛道。我有个糟糕的习惯，在应该写 private 的时候我会写 protected。在或许应该使用 IOStream（C++输入输出流）

的时候我会偏好于使用 printf()。我喜欢数组和指针，而且我还是 C 兼容 API 的忠实拥护者。我甚至并不是完全忠实地遵循苦行僧式编程哲学的，但我坚信这一哲学信条是实现你的目标的最可靠、最快捷的途径，当然，前提是你必须随时随地尽可能地坚持它。

“Imperfect C++”的精神

除了不完美主义实践者的哲学信条之外，本书大体上还反映了我在编写 C++代码时遵循的指导原则。它们总的来说（尽管并非完全）跟“C 精神”[Como-SOC]和“C++精神”[Como-SOP]这一对孪生法则是相吻合的：

C 精神：

- 相信程序员：“Imperfect C++”不羞于作出丑陋的决策。
- 不要阻止程序员去做必须完成的事情：“Imperfect C++”实际上会帮助你完成你想做的事情。
- 保持解决方案的简洁：书中的大部分代码正是如此，而且是高度平台无关和可移植的。
- 使它快！即使影响到可移植性的保证也在所不惜：效率被给予高度的重视，尽管我们偶尔会为此牺牲可移植性。

C++精神：

- C++是 C 的一种方言，不过 C++针对现代软件开发作了一些增强：我们在不少重要的场合下仍然依赖于和 C 的互操作性。
- 尽管 C++是一门比 C 庞大得多的语言，但你并不需要为你不使用的东西付出代价（这样时间和空间的额外开销就会被保持为最小。而那些确实存在开销的地方也得被整体观察才能作出定论，因为你要比较的是等价的程序，而不是在特性 X 和特性 Y 之间进行比较）。
- 尽可能地在编译期捕获错误：“Imperfect C++”在任何适当的地方使用静态断言（static assertions）和约束（constraints）。
- 尽量避开预处理（大多数情况下 inline、const、template 等才是正道）：我们会看到各种各样的技术，它们使用 C++语言而不是预处理器来实现我们的目标。

除了这些原则外，本书还会以身作则地示范以下的做法，即在任何可能的情况下，

- 编写不依赖于特定的编译器（扩展和特性）、操作系统、错误处理模型、线程模型以及字符编码策略的代码；
- 在不可能使用编译期错误侦测的情况下采用契约式设计（Design-by-Contract）（见 1.3 节）。

编码风格

为了使得本书的篇幅不至于过长，我不得不在示例代码中大幅略去我惯常采用的严格编码风格（当然，有

些人可能认为那是“学究气”的编码风格）。第 17 章描述了我在布局类定义代码时通常遵循的原则。其他编码风格，如大括号和间隔缩进风格等，就无伤大雅了。如果你有兴趣，你可以很容易地从配书光盘中找到这方面的大量材料。

术语

计算机语言就是二进制语言，而人类的语言则是各种模糊的、不精确的语言。这里我给出一些术语的定义，本书其余部分将沿用这些术语：

客户代码：表示使用其他代码的代码，通常（但不局限于）指的是客户应用程序代码使用库代码的情形。

编译单元：来自一个源文件以及该源文件所包含（依赖）的全部头文件的全部源代码所构成的整体。

编译环境：编译环境是由编译器、库、操作系统构成的一个整体环境，我们编写的代码就是在这种环境中编译的。感谢 Kernighan 和 Pike [Kern1999] 对此的定义。

仿函数（Functor）：这是一个被广泛使用的术语，其含义代表的是函数对象（Function Object）或 Functional，但这个称呼并不是标准的称呼。实际上我更倾向于使用函数对象（FunctionObject）这一术语。但有人说服了我 [注 4]，说仿函数（Functor）更好一些，因为它只有一个短短的单词，更有特色，更醒目，也更容易查找，特别是在网上查找的时候。

[注 4] 要责怪就责怪本书的几个审阅者好了。

泛用性（Generality）：我或许从来都没有弄懂“泛型（genericity）”这个词，或者说至少没有弄懂它在编程上下文中的含义，尽管我有时候也会把这个词拿出来用一下。我猜它的意思是“编写对一系列类型都能够起作用的模板代码，这些类型在它们被使用的方式（而不是定义）上有联系”，只有在这种情况下我才会使用“泛型（genericity）”这个词。而 Generality（泛用性）[Kern1999] 则似乎表意更恰当一些，感觉上更适用于表达这一概念，因为我不仅关心我的代码是否能够在其他（模板参数）类型上工作，同样也关心它们能否跟其他头文件和库一起工作。

除了这些概念上的术语之外，我还使用了一些特殊的语言相关的术语。我不知道你怎么想，但我确实发现 C++ 中的术语挺混乱的，因此我打算花一点时间来回顾一些定义。下面的定义来自 C++ 标准，但我将它们以一种更简单的方式呈现出来，以便于我们理解，而不仅仅是我个人的理解。这些定义中有些是重叠的，因为虽说它们针对的是不同的概念，但都是 C++ 实践者的字典里的一部分。这里的“C++ 实践者”也包括我所说的不完美主义实践者。

基本类型和复合类型

基本类型（C++-98: 3.9.1）包括整型（char、short、int、long（long long/__int64），以上这些类型的有符号和无符号的版本[注5]，以及bool）、浮点型（float、double以及long double）以及void类型。

[注5]注意，char有三种“型号”：char、unsigned char以及signed char。我们会在第13章看到，这也会带来一些问题。

复合类型（C++-98: 3.9.2）基本上就是表示除基本类型之外的其他所有类型，包括：数组、函数、指针（任何类型的指针，包括指向非静态成员的指针）、引用、类、联合（unions）以及枚举。

我倾向于不使用“复合类型”这个术语，因为我觉得它的名字好像是在说它是由其他东西所“组成”的一样，但实际上对于指针和引用来说根本就不是这么一回事。

对象类型

对象类型（C++-98: 3.9; 9）包括任何“不是函数类型、不是引用类型、也不是void类型”的类型。我同样不喜欢使用这个术语，因为按照这种说法它的范畴并不仅仅包括“类类型（class types）的实例”，可是实际上人们却往往会这么想。所以我在全书中凡涉及到这种地方一律用“实例”这一术语。

标量类型和类类型

标量类型（Scalar types）（C++-98: 3.9; 10）包括“算术类型、枚举类型以及指针类型”。类类型（C++-98: 9）即是指以class、struct或union这三个关键字之一所声明的东西。

结构是一个由struct关键字进行定义的类类型，其成员和基类的访问控制缺省情况下均为public。联合是一个通过union关键字定义的类类型，其成员缺省情况下也是public的。类则是由class关键字定义的类类型，其成员和基类的访问控制在缺省情况下均为private。

聚合体

标准（C++-98: 8.5.1; 1）这样来描述聚合体（aggregate）：“一个数组，或一个无用户自定义构造函数、无private或protected的非静态数据成员、无基类并且无虚函数的类”。作为一个数组或一个类类型，它意味着将多个东西聚合进一个东西中，故有“聚合体”一说。

聚合体的初始化方式可以是一对包含有初始化子句的大括号，像这样：

```
struct X
{
 int i;
 short as[2];
} x = { 10, { 20, 30 }};
```

尽管聚合体通常由 POD 类型（见下一节）组成，但未必非得如此。在上面的代码中，X 的成员变量 i 也可以是类类型的，只要它具有一个接受单个整型参数的非显式构造函数（non-explicit constructor）（见 2.2.7 节）和一个可用的拷贝构造函数即可。

POD 类型

POD 意思是“plain-old-data”（C++-98: 1.8; 5），它是 C++中的一个非常重要的概念，但通常会被人们误解，而且虽说它是个“小东西”，但却相当重要。其定义也相当糟糕。标准给出了两个线索。（C++-98: 3.9; 2）说：“任何完备的（complete）[译注 1]的 POD 类型 T……必须满足以下条件：将组成它的一个对象的各字节拷贝到一个字节数组中，然后再将它们重新拷贝回原先的对象所占的存储区中，此时该对象应该仍具有它原来的值。”然而在（C++-98: 3.9; 3）处，我们又被告知：“对于任何 POD 类型 T，如果有两个指针分别指向不同的 T 对象 obj1 和 obj2，此时如果使用 memcpy()将 obj1 的值拷贝进 obj2 中，obj2 就应该跟 obj1 具有相同的值。”

[译注 1]完备的（complete）与非完备的（incomplete）相对。一个类类型如果只有声明没有定义的话，就称它是非完备的，反之则是完备的。一般而言，一个类型要成为完备的，前提是其内存布局必须是已知的。

唔，这些说法相当华而不实，不是吗？值得庆幸的是，为了充实这个定义，C++标准在它 776 页篇幅的内容中另有若干处（准确地说，是 56 处）零星地提到了关于 POD 类型的一些信息。

在 [Comp-POD]中，Greg Comeau 指出，大多数 C++书籍根本不提 POD，并且说“大多数 C++书籍都不值得购买”。带着为了提高销量这个“愤世嫉俗”的企图，我打算尽量把 POD 描述得到位一些。这应该不会太难，因为 Greg 在他的文章中已经提供了 POD 结构的所有本质特征，那我就全当搭顺风车啦。Let's GO!

标准（C++-98: 3.9; 10）对 POD 类型的总体性定义如下：“标量类型、POD 结构类型、POD 联合类型，以上这些类型的数组，以及这些类型以 const/volatile 修饰的版本。”上面这个定义，除了 POD 结构和 POD 联合

有待澄清外，其余都非常清楚。

POD结构（C++-98: 9;4）是“一个聚合体类，其任何非静态成员的类型都不能是如下任一种：指向成员的指针、‘非POD’结构、‘非POD’联合，以及以上这些类型的数组或引用，同时该聚合体类不允许包含用户自定义的拷贝赋值操作符和用户自定义的析构函数。”POD联合的定义类似，只不过它是一个联合而不是结构而已。注意，聚合体类不仅可以是结构（struct），也可以是类（class）。

到目前为止一切都没问题，但POD类型到底有什么重要之处呢？呃，POD类型允许我们与C函数进行交互，它们是C与C++之间互通的手段，因此推而广之就成了C++与外部世界沟通的桥梁（见7、8、9三章）。因此，POD结构或POD联合的存在就允许我们“知道一个普通的结构或联合在C里面是什么样子的”[Como-POD]。POD最好的助记办法就是将它看成是一种与C兼容的类型，当然同时你也不能忘记有关POD的方方面面。

POD其他重要的方面包括：

- 宏offsetof()所作用于的类型应该是“一个POD结构或一个POD联合”（C++-98: 18.1; 5），此外用在任何其他类型（见2.3.2节和第35章）身上都是未定义的。

- POD类型可以被放在联合中。这个特性被我用来制造一个约束（见1.2.4节），该约束的作用是约束一个类型为POD类型（见1.2.4节）。

- 如果一个POD类型的静态变量是以常量表达式来初始化的话，那么其初始化发生于执行流进入它所在的语句块之前，而且也是在任何（不管是POD类型还是其他类型）需要动态初始化的对象的初始化之前（见第11章）。

- 指向成员的指针不是POD类型，这一点跟其他任何指针类型恰恰相反。

- POD结构或POD联合类型可以具有静态成员、成员typedef、嵌套类型和方法[注6]。

[注6]当然，这些东西对于任何C代码来说都应该是不可见的，因此，如果你的代码必须是C/C++兼容的话，你就应当使用条件编译将它们排除在外，只有当处于C++编译环境中时才让它们成为可见的。

第 1章 强制设计：约束、契约和断言

在我们设计软件时，我们希望软件根据设计而进行使用。这并非一句空话。在大多数情况下，很容易发生以意料之外的方式来使用软件，而这么做的结果往往是令人失望的。

大多数软件的文档几乎都是不完整，甚至是过时的，我坚信你也有这方面的经验。这并非单纯的错误或缺失，“如果还有比没有文档更糟的情形，那就是文档是错误的”[Meye1997]。如果被使用的组件比较简单，使用得当，或者说是标准的或被普遍使用的，那么没有文档倒也不是什么大问题。例如，如果许多程序员需要一而再、再而三地查找 C库函数 malloc()的用法，那可真算是奇闻怪谈了。然而，这种情况毕竟很少。我曾经遇到一些程序员，他们非常有经验，但对 malloc()的兄弟 realloc()和 free()之间的细微差别却并不那么熟悉。

对于这些问题，解决的方式很多。其一是通过增强的参数验证，使软件组件具有更强的抵抗错误的能力，但这种方式通常不那么有吸引力，因为它会损及性能，还倾向于滋生坏习惯。制作良好的文档并让它们保持更新当然是解决方案的一个重要组成部分，然而这种做法是远远不够的，因为它是“非强制性”的。此外，要想写出好文档也是一件极其困难的事情[Hunt2000]。软件越复杂，其原始作者要想把自己摆在对该软件懵懂无知的处境从而便于写出更好的说明文档就越不可能，而独立的技术作者要想抓住其所有的细微之处就更加困难了。因此，当情况不再单纯时，一个确保正确使用代码的更好的方式显然是必不可少的。

如果编译器能够为我们找出错误那就更可取了。事实上，本书中的相当一部分内容都是关于如何驱使和利用编译器，令它在碰到糟糕的代码时卡壳，从而便于我们在编译期及早抓住错误。但愿你能够意识到花几分钟来安抚编译器比花上几个小时和调试器纠缠要好得多。正如 Kernighan和 Pike在《The Practice of Programming》[Kern1999]中所说的那样，“无论你喜欢与否，调试是一门我们经常要实践的艺术……如果不产生 bug就好了，所以我们尝试在第一时间就把代码写正确，从而尽量避免 bug的产生。”由于我并不比其他软件工程师更勤快，因此我总是尽量让编译器帮我做事情。从长远来看，“苦行僧”式的编程是比较容易的选择。然而，并非所有的错误都能够在编译期查出来。在这种情况下，我们需要求助于运行期机制。一些语言（例如 D和 Eiffel）提供了内建的机制，即通过“契约式设计（Design by Contract）”来确保软件按照其设计而被使用。此项技术的先驱是 Bertrand Meyer[Meye1997]，它源于程序的形式验证。契约式设计要求为组件指定“契约”，这些契约会在程序运行过程中的某些特定点被强制执行。契约在很多方面都可以作为文档的替代品，因为它们无法被忽略，并且是自动进行验证的。此外，通过遵循特定的语法约定，它就可以和自动化文档工具合作，我们将会在 1.3节对此进行讨论。

实施“强制（enforcement）”的机制之一是断言（assertion），包括广为人知的运行期断言，以及较少为人知然而甚至更为有用的编译期断言。本书中两者均得到了大量的使用，因此我们将会在 1.4节对这种重要的工具进行详细的观察。

1.1 鸡蛋和火腿

我并不怀疑我很可能是在班门弄斧，然而有些事情很重要，在此不得不说。因此，请各位允许我唠叨片刻：

- 在设计期捕获 bug比在编码 /编译期捕获好。 [注 1]
- 在编码 /编译期捕获 bug比在单元测试中捕获好。 [注 2]
- 在单元测试中捕获 bug比在调试中捕获好。
- 在调试中捕获 bug比在预发行 /beta版中捕获好。
- 在预发行 /beta版中捕获 bug比让你的客户捕获好。
- 让你的客户捕获 bug（以具有亲和力的方式）比没有客户好。

[注 1]我并非瀑布模型的拥护者，所以编码期和编译期对于我来说都是一样。不过，纵然我喜欢单元测试，并且体验过一些快速结对编程（pair-programming）合作，我仍然不认为我是一个 XP（极限编程）[Beck2000] 热衷者。

[注 2]这假定你做了单元测试。如果你没有，那么你需要开始这么做——现在就开始！

这些都是相当明显的东西，尽管客户可能并不赞同最后一条。最好把那条留给我们自己。

实施强制有两种方式：在编译期和在运行期。这些正是本章要讲述的内容。

1.2 编译期契约：约束

本章讲述编译期强制，通常它也被称为“约束（constraints）”。遗憾的是，C++并不直接支持约束。

Imperfection： C++并不直接支持约束。

C++是一门极其强大和灵活的语言，因此很多支持者（甚至包括一些 C++权威）都会认为本节描述的约束实现技术已经足够了。然而，作为 C++和约束的双重拥护者，我必须提出我的异议（由于一些很平常的原因）。虽然我并不买其他语言鼓吹者的账，然而我同样认为阅读因违反约束而导致的编译错误信息是很困难（甚至极其困难）的。它们通常令人眼花缭乱，有时甚至深奥无比。如果你是“肇事”代码的作者，那么问题通常并不严重，然而，当面对的是一个多层模板实例化的情形时，其失败所导致的错误信息近乎天书，难以理解，甚至非常优秀的编译器都是如此。本节后面我们会看到一些约束，以及违反它们所导致的错误信息，还有可以令错误信息稍微更具可读性的一些措施。

1.2.1 must_have_base()

这个例子是从 comp.lang.c++.moderated新闻组上几乎原样照搬过来的，是 Bjarne Stroustrup发的帖子，他称之为“ Has_base”。 [Sutt2002]对此亦有描述，不过换了个名字，叫做 IsDerivedFrom。而我则喜欢将约束的名字以“ must_”开头，因此我把它命名为 must_have_base。

程序清单 1.1

```
template< typename D
        , typename B
        >
struct must_have_base
{
  ~must_have_base()
  {
    void(*p)(D*, B*) = constraints;
  }
private:
  static void constraints(D* pd, B* pb)
  {
    pb = pd;
  }
};
```

它的工作原理如下：在成员函数 constraints()中尝试把 D类型的指针赋值给 B类型的指针（D和 B都是模板参数)， constraints()是一个单独的静态成员函数，这是为了确保它永远不会被调用，因此这种方案没有任何运行期负担。析构函数中则声明了一个指向 constraints()的指针，从而强迫编译器至少要去评估该函数以及其中的赋值语句是否有效。

事实上，这个约束的名字有点不恰当：如果 D和 B是同样的类型，那么这个约束仍然能够被满足，因此，它或许应该更名为 must_have_base_or_be_same_type，或者类似这样的名字。另一个替代方案是把 must_have_base进一步精化，令它不接受 D和 B是同一类型的情况。请将你的答案写在明信片上寄给我。

另外，如果 D和 B的继承关系并非公有继承，那么该约束也会失败。在我看来，这是个命名方式的问题，

而不是约束本身的能力缺陷问题，因为我只需要对采用公有继承的类型应用这个约束。 [注 3]

[注 3]这听起来像是循环论证，不过我敢保证它不是。

由于“ must_have_base”在其定义中所进行的动作恰好体现了约束本身的语义，所以如果该约束失败，错误信息会相当直观。事实上，我手头所有编译器（见附录 A）对此均提供了非常有意义的信息，即要么提到两个类型不具有继承关系，要么提到两个指针类型不能互相转换，或与此类似的信息。

1.2.2 must_be_subscriptable()

另一个有用的约束是要求一个类型可以按下标方式访问（见 14.2节），实现起来很简单：

```
template< typename T>
struct must_be_subscriptable
{
  . . .
  static void constraints(T const &T_is_not_subscriptable)
  {
    sizeof(T_is_not_subscriptable[0]);
  }
  . . .
```

为了提高可读性， constraints()的形参被命名为 T_is_not_subscriptable，这可以为那些违反约束的可怜的人儿提供些许线索。考虑下面的例子：

```
struct subs
{
public:
  int operator [](size_t index) const;
}
struct not_subs
{};

must_be_subscriptable<int[]>    a; // int*可以按下标方式访问
must_be_subscriptable<int*>     b; // int*可以按下标方式访问
```

```
must_be_subscriptable<subs>     c; // subs可以按下标方式访问
must_be_subscriptable<not_subs> d; // not_subs不可以按下标方式访问，编译错误
```

Borland 5.6给出了令人难以置信的信息：“ 'operator+' not implemented in type '<type>' for arguments of type 'int' in function must_be_subscriptable<not_subs>::constraints(const not_subs &)”。而当模板实例化深度多达 15层时，不绞尽脑汁简直是不可能搞清楚这些错误信息的含义的。

相较而言， Digital Mars更为准确一些，不过仍然不够好：“ Error: array or pointer required before '['; Had: const not_subs”。

另外一些编译器的错误信息都包含了变量名“ T_is_not_subscriptable”。其中最优秀的当数 Visual C++给出的错误信息：“ binary '[' : 'const struct not_subs' does not define this operator or a conversion to a type acceptable to the predefined operator while compiling class-template member function 'void must_be_subscriptable<struct not_subs>::constraints (const struct not_subs &)”。

1.2.3 must_be_subscriptable_as_decayable_pointer()

在第 14 章中，我们会深入考察数组和指针之间的关系，并了解有关指针偏移的一些诡秘的特性，它致使“ offset[pointer]”这种语法成为跟“ pointer[offset]”一样合法且等价的形式（这也许使 Borland编译器关于 must_be_subscriptable的莫名其妙的编译错误看起来不是那么毫无意义，不过它对于我们追踪并理解约束是如何被违反的确实有很大的帮助。） 由于这种颠倒形式对于重载了“ operator []”的类（class）类型是无效的，因此 must_be_subscriptable还可以被进一步精化，从而把它作用的类型约束到仅限于（原生）指针类型。

程序清单 1.2

```
template <typename T>
struct must_be_subscriptable_as_decayable_pointer
{
  . . .
  static void constraints(T const &T_is_not_decay_subscriptable)
  {
    sizeof(0[T_is_not_decay_subscriptable]); // 译注：offset[pointer形式，
                                             // 仅仅作用于原生指针。
  }
```

```
  . . .
};
```

很明显,所有可以通过 offset[pointer形式进行索引访问的 pointer也都可以接受 pointer[offset操作,因此不需要把 must_be_subscriptable合并到 must_be_subscriptable_as_decayable_pointer中去。不过,尽管这两种约束有着不同的结果,但利用继承机制将二者结合起来也是合适之举。

现在我们可以区分(原生)指针和其他可进行索引访问的类型了:

```
must_be_subscriptable<subs>                          a; // ok
must_be_subscriptable_as_decayable_pointer<subs> b; // 编译错误
```

1.2.4 must_be_pod()

你会在本书中多次看到 must_be_pod()的使用(见 19.5、19.7、21.2.1和 32.2.3节)。这是我编写的第一个约束,那时我根本不知道约束是什么,甚至连 POD 是什么意思都不清楚(见"序言")。must_be_pod()非常简单。

C++98标准 9.5节第 1条说明:"如果一个类具有非平凡的(non-trivial)构造函数,非平凡的拷贝构造函数,非平凡的析构函数,或者非平凡的赋值操作符,那么其对象不能作为联合(union)的成员。"这恰好满足我们的需要,并且,我们还可以设想,这个约束和我们已经看到的那些模样差不多:有一个 constraints()成员函数,其中包含一个 union:

```
template <typename T>
struct must_be_pod
{
  . . .
  static void constraints()
  {
    union
    {
      T   T_is_not_POD_type;
    };
  }
  . . .
```

遗憾的是，这是编译器容易发生奇怪行为的领域，所以真正的定义没有这么简单，而是需要许多预处理器操作（见 1.2.6节），但效果还是一样的。

在 19.7节中，我们将会看到这个约束和一个更专门化的约束 must_be_pod_or_void()一起使用，目的在于能够检查某个指针所指的是否为非平凡的类类型。为此，我们需要对 must_be_pod_or_void模板进行特化 [Vand2003]，而其泛化的定义则与 must_be_pod相同：

```
template <typename T>
struct must_be_pod_or_void
{
  . . . // 和 must_be_pod一样
};
```

```
template <>
struct must_be_pod_or_void<void>
{
  // 该特化的定义是空的，里面什么也没有，所以不会找编译器的麻烦。
};
```

同样，编译器对于违反 must_be_pod / must_be_pod_or_void约束所生成的信息也是各式各样的：

```
class NonPOD
{
public:
  virtual ~NonPOD();
};

must_be_pod<int>      a; // int是 POD（见“序言”）
must_be_pod<not_subs> b; // not_subs是 POD（见“序言”）
must_be_pod<NonPOD>   c; // NonPOD不是 POD，编译错误
```

这一次，Digital Mars一惯的简练风格却给我们带来了麻烦，因为我们能得到的错误信息只是“Error: union members cannot have ctors or dtors”，指向约束类内部引发编译错误的那行代码。如果这是在一个大项目里的话，那么很难追溯到错误的源头，即最初引发这个错误的实例化点。而 Watcom对于这么一个极小的

错误给出的信息则是最多的：“ Error! E183: col(10) unions cannot have members with constructors; Note! N633: col(10) template class instantiation for ''must_be_pod<NonPOD>' was in: ..\constraints_test.cpp(106) (col 48)”。

1.2.5 must_be_same_size()

最后一个约束 must_be_same_size()也在本书的后续部分被使用到（见 21.2.1节和 25.5.5节）。该约束类使用静态断言“无效数组大小”技术来确保两个类型的大小是一致的，我们很快就会在 1.4.7节看到该技术。

程序清单 1.3

```
template< typename T1
        , typename T2
        >
struct must_be_same_size
{
  . . .
private:
  static void constraints()
  {
    const int T1_not_same_size_as_T2
                        = sizeof(T1) == sizeof(T2);
    int       i[T1_not_same_size_as_T2];
  }
};
```

如果两个类型大小不一致，那么 T1_not_same_size_as_T2就会被求值为（编译期）常数 0，而将 0作为数组大小是非法的。

至于 must_be_pod_or_void，则是在两个类型中有一个或都是 void时会用到它。然而，由于 sizeof(void)是非法的表达式，因此我们必须提供一些额外的编译期功能。

如果两个类型都是 void，则很容易，我们可以这样来特化 must_be_same_size类模板：

```
template <>
struct must_be_same_size<void, void>
{};
```

然而，如果只有一个类型是 void，要使其工作可就没那么直截了当了。解决方案之一是利用局部特化机制[Vand2003]，然而并非所有目前广为使用的编译器都支持这种能力。进一步而言，我们还得同时为这个模板准备一个完全特化和两个局部特化版本（两个局部特化分别将第一个和第二个模板参数特化为 void）。最后，我们还得构思某种方式来提供哪怕是“有点儿”意义的编译期错误信息。我没有借助于这种方法，而是通过令 void成为“可 size_of的”来解决这个问题。我的方案实现起来极其容易，并且不需要局部特化：

程序清单 1.4

```
template <typename T>
struct size_of
{
  enum { value = sizeof(T) };
};
template <>
struct size_of<void>
{
  enum { value = 0 };
};
```

现在我们所要做的就是在 must_be_same_size中用 size_of来替代 sizeof：

```
template< . . . >
struct must_be_same_size
{
  . . .
  static void constraints()
  {
    const int T1_must_be_same_size_as_T2
                  = size_of<T1>::value == size_of<T2>::value;
    int       i[T1_must_be_same_size_as_T2];
```

```
  }
};
```

现在我们可以对任何类型进行验证了：

```
must_be_same_size<int, int>   a; // ok
must_be_same_size<int, long>  b; // 依赖于硬件架构或编译器
must_be_same_size<void, void> c; // ok
must_be_same_size<void, int>  d; // 编译错误：void的“大小”是 0!
```

正如前面的约束所表现出来的，不同的编译器提供给程序员的信息量有相当大的差别。Borland和 Digital Mars在这方面又败下阵来，它们提供的上下文信息极少甚至没有。在这方面，我认为 Intel做得最好，它提到“ zero-length bit field must be unnamed”，指出出错的行，并且提供了两个直接调用上下文，其中包括 T1的实际类型和 T2的实际类型，加在一起一共四行编译器输出。

1.2.6 使用约束

我比较喜欢通过宏来使用我的约束，宏的名字遵循“ constraint_<constraint_name>”的形式 [注 4]，例如 constraint_must_have_base()。这从多方面来说都是非常有意义的。

[注 4]正如 12.4.4节所描述的原因，把宏命名为大写形式总是好习惯。之所以我对约束的宏没有采用同样的命名习惯，是因为我想要和约束类型保持一致（小写）。事后我才发现这么做令它们看起来不怎么显眼。你自己编写的约束当然可以采用大写风格。

首先，容易无歧义地查找到它们。正因为如此，我才为约束保留了“ must_”前缀。也许有人会争论说这个需求已经被满足了。然而使用宏的做法更具有“自描述”性。对观者而言，在代码中看到“ constraint_must_be_pod()”，其意义更加明确。

其次，使用宏的形式更具有（语法上的）一致性。尽管我没有写过任何非模板的约束，然而并没有任何理由限制其他人那么做。此外，我发现尖括号除了导致眼睛疲劳外，没有什么其他好处。

再次，如果约束被定义在某个名字空间中，那么在使用它们时必须加上冗长的名字限定，而宏则可以轻易地将名字限定隐藏起来，免得用户去使用淘气的 using指令（见 34.2.2节）。

最后一个原因更实际。不同的编译器对相同的约束的处理具有细微的差别，为此需要对它们耍弄一些手段。

例如，针对不同的编译器， constraint_must_be_pod()被定义成如下三种形式之一：

```
do { must_be_pod<T>::func_ptr_type const pfn =
                    must_be_pod<T>::constraint(); } while(0)
```

或者

```
do { int i = sizeof(must_be_pod<T>::constraint()); } while(0)
```

或者

```
STATIC_ASSERT(sizeof(must_be_pod<T>::constraint()) != 0)
```

利用宏，客户代码变得更加简洁优雅，否则它们会遭受大量的面目丑陋的代码的干扰。

1.2.7 约束和 TMP

本书的一位审稿人提到部分约束可以通过 TMP（template meta-programming，模板元编程）[注 5]实现，他说的很对。例如，must_be_pointer约束就可以通过结合运用静态断言（见 1.4.7节）和 is_pointer_type（见 33.3.2节）来实现。如下：

```
#define constraint_must_be_pointer(T) \
  STATIC_ASSERT(0 != is_pointer_type<T>::value)
```

[注 5]我故意让这本书尽量少地涉及模板元编程技术，因为它本身就是一个相当大的主题，而且和我所讨论到的那些“ imperfections”并没有直接的关系。你可以在 Boost、 RangeLib、 STLSoft库（包含在配书光盘中）或其他一些书 [Vand2003，Alex2001]中找到许多模板元编程例子。

我之所以不采用 TMP有几方面原因。首先，约束的编码看上去总是那么直观，因为一个约束只不过是对被约束类型所应该具备的行为的“模拟”。而对于 TMP traits就不能这么说了，有些 TMP traits可能会相当复杂。因此，约束较之模板元编程更易于阅读。

其次，在许多（尽管并非全部）情况下，要想“说服”编译器给出容易理解的信息，约束比 traits（和静态断言）更容易一些。

最后，有些约束无法通过 TMP trait来实现，或者至少只能在很少一部分编译器上实现。甚至可以说，约

束越是简单，用 TMP traits来实现它就越困难，对此 must_be_pod就是一个极好的例子。

Herb Sutter在 [Sutt2002]中曾示范了结合运用约束和 traits的技术，当你进行自己的实际工作中时，没有任何理由阻止你那么做。对我来说，我只不过更喜欢保持它们简洁且独立而已。

1.2.8 约束：尾声

本章所讲述的约束当然并非全部，然而，它们可以告诉你哪些东西是可实现的。约束的不足之处和静态断言一样（见 1.4.8节），那就是产生的错误信息不是特别容易理解。根据实现约束的机制的不同，你可能会看到明白如“ type cannot be converted”之类的错误信息，也可能会看到令人极其困惑的“ Destructor for 'T' is not accessible in function <non-existent-function>”之类的信息。

只要有可能，你应该通过为你的约束选择合适的变量或常量名字来改善错误信息。我们在本节已经看到了这方面的例子： T_is_not_subscriptable、 T_is_not_POD_type以及 T1_not_same_size_as_T2。记住，要确保你选择的名字反映了出错的情况。想想违反了你的约束却看到诸如“ T_is_valid_type_for_constraint”之类的信息的可怜人儿吧！

最后，关于约束，还有一个值得强调的重要方面：随着我们对编译期计算和模板元编程的认识越来越清晰，我们可能会想去更新某些约束。也许你已经从本书中描述的某些组件中看出我并非模板元编程方面的大师，不过重点是，通过良好地设计用于表现和使用约束的方式，我们可以在以后学到更多的技巧时再来“无缝”地更新我们的约束。我承认我自己有好多次就是这么做的，我不觉得这是什么丢人的事情，不过，我早期在编写约束方面的尝试如果拿出来现眼倒的确令人赧颜（我没有在附录 B中提到它们，因为它们还没有差劲到那个地步，远远没有！）。

1.3 运行期契约：前条件、后条件和不变式

“如果例程的所有前条件（precondition）已经被调用者满足了，那么该例程必须确保当它完成时所有后条件（postconditions）（以及任何不变式）皆为真。”

——Hunt and Thomas, The Pragmatic Programmers [Hunt2000].

如果我们无法执行编译期强制，那么还可以采用运行期强制。运行期强制的一个系统化的实现途径是指定函数契约。函数契约精确定义了在函数被调用之前调用者必须满足哪些条件（前条件），以及在函数返回之时哪些条件（后条件）是调用者可以期望的。契约的定义以及它们的强制实施是 DbC（Design by Contract，契约式设计） [Meye1997]的基石。

前条件是指函数履行其契约所必须满足的条件。满足前条件是调用者的责任，而被调用者则假定它的前条件已经被满足，并且仅当它的前条件被满足时才负责提供正确的行为。这一点非常重要，在 [Meye 1997] 中被强调指出。倘若调用者没有满足前条件，则被调用者做出任何事情都是完全合理的。事实上，通常这会引发一个断言（见 1.4节），进而可能导致程序终止。这听起来似乎颇令人恐慌，刚接触 DbC的程序员通常会对此感到很不舒服，直到你问起他们“如果一个函数的（前提）条件都不能被满足，那还能指望它有什么样的行为呢？”才哑口无言。事实上，契约越严格，违反它所导致的后果越严重，从而软件的质量就会越好。当转到 DbC上时，要理解这一点是最为困难的。

后条件在函数执行完毕时必须为真。确保后条件被满足是被调用者的责任。当函数返回控制时调用者可以假定后条件已经得到了满足。然而，在现实中，有些时候有所保留（不要把赌注全部押在被调用者身上）还是必要的，例如，当调用应用服务器中的第三方插件时就是如此。然而，我认为前面所讲的原则仍然是对的。事实上，对违反契约的插件的合理反应之一是将它卸载掉，并给公司经理以及第三方插件厂商发一封电子邮件。既然我们对于违反契约的行为可以作出任何反应，那么有什么理由不这么做呢？

前条件和后条件可以被应用到类的成员函数，也可以被用到自由函数身上，这对于 C++（更一般地说，面向对象编程）来说很有益处。事实上，还有另外一个与 DbC相关的东西，它只能依附于类而存在，那就是类不变式（class invariant）。类不变式是指一个或一组条件式，它们对于一个处于良好定义状态的对象总是为真。根据定义，类的构造函数负责确保类的实例进入一个符合该类的不变式的状态中，而类的（public）成员函数则在它们完成之际确保类的实例仍然处在该状态中。仅当处于构造函数、析构函数或其他某个成员函数的执行过程中时，类不变式才不一定要为真。

在某些场合下，将不变式的作用范围定义为比“单个对象的状态”的范围更广可能更合适一些。原则上，不变式可以被应用到操作环境的整个状态上，然而，在实践中，这种情况是极其少见的，类不变式则很常见。因此，在本章以及本书剩余的篇幅中，如果提到不变式，均是指类不变式。

对部分或根本没有进行封装的类型提供不变式是可行的（见 3.2节和 4.4.1节），这个不变式是由与该类型相关的 API函数（以及该函数的前条件）来强制实施的。事实上，当使用这种类型时，不变式是极好的主意，因为它们缺乏封装性的特质提高了滥用的风险。不过这种不变式相当容易被“绕过”，这也说明了为什么通常应该避免使用这种类型。事实上，[Stro2003]中某种程度上提到：如果存在一个不变式，则公有数据简直毫无意义。封装既是关于隐藏实现又是关于保护不变式的。至于“属性”（第 35章），可能是为了结构上的一致性（见 20.9节）而引入的，只不过为我们提供公有成员变量的表象而已，它仍然具有不变式。

对于违反前条件、后条件或者不变式，你所采取的行动完全由你来决定。你可以把信息记录到日志文件中，也可以抛出异常，或者给你家里那口子发一封 SMS，告诉她今夜你将 debug到很晚，无法与她枕边絮语。不过，通常我们采取的行动是引发一个断言。

1.3.1 前条件

在 C++中，前条件测试相当简单。在这本书中我们已经看到了好几个例子。它和使用断言一样简单：

```
template< . . . >
typename pod_vector<. . .>::reference pod_vector<. . .>::front()
{
  MESSAGE_ASSERT("Vector is empty!", 0 != size());
  assert(is_valid());
  return m_buffer.data()[0];
}
```

1.3.2 后条件

这一块是 C++容易产生磕磕碰碰的地方。这里的挑战是在函数的退出点捕获返回值和“输出”参数（译注：即用于向外界返回东西的函数参数，例如指向待填充的缓冲区的指针）。当然了，C++提供了特别有用的 RAII（Resource Acquisition Is Initialization，资源获取即初始化）机制（见 3.5节），该机制保证当执行流程退出某个作用域时栈上对象的析构函数都会得到调用。这就意味着我们可能借助这一点实现一个可行方案，至少该机制具备这个潜力。

我们的选择之一是声明监视器对象，它持有对输出参数和返回值的引用。

```
int f(char const *name, Value **ppVal, size_t *pLen)
{
  int              retVal;
  retval_monitor   rvm(retVal, . . . policy . . . );
  outparam_monitor opm1(ppVal, . . . policy . . . );
  outparam_monitor opm2(pLen, . . . policy . . . );
  . . . // 函数体
  return retVal;
}
```

一些策略会被用来检查变量是否为 NULL，或者是否位于一个特定的区间内，或者是一组数值中的一个，等等。尽管实现这些东西都有困难，这里仍然存在两个问题。第一， rvm的析构函数会对它所持有的指向函

数返回值变量 retVal引用来施行约束。如果函数的其他任何部分返回了一个不同的值（或一个常量），那么 rvm无可避免地会报告一次失败。为了能够正确工作，我们不得不强制让所有函数都通过单个变量来返回，这肯定不符合一些人的口味，在某些场合下也是不可能的。

然而，最主要的问题还在于各个后条件监视器之间是没有关联的。大多数函数的后条件是复合型的，个体输出参数和返回值仅当符合某种一致的关系时才有意义，例如：

```
assert(retVal > 0 || (NULL == *ppVal && 0 == *pLen));
```

我不打算建议你如何将这三个个体监视器对象以这样的方式结合起来，以便强制实施各种各样的后条件状态，这类事情对于模板元编程爱好者可能是一个令人激动的挑战，不过对于其他人，它所带来的复杂性不值得我们付出代价。

Imperfection: C++对后条件未提供合适的支持。

在我看来，唯一合理的（虽然看起来很平凡）解决方案是通过一个转发函数将（待调用）函数和对它的（后条件）检查分离开来，就像在程序清单 1.5中展示的那样：

程序清单 1.5

```
int f(char const *name, Value **ppVal, size_t *pLen)
{
  . . . //进行 f()的前条件检查
  int retVal = f_unchecked(name, ppVal, pLen);
  . . . //进行 f()的后条件检查
  return retVal;
}
int f_unchecked(char const *name, Value **ppVal, size_t *pLen)
{
  . . . // f的语义
}
```

在实际代码中，你可能希望在不需要执行 DbC的地方省略掉所有的检查，为此我们需要使用预处理器：

程序清单 1.6

```
int f(char const *name, Value **ppVal, size_t *pLen)
#ifdef ACMELIB_DBC
{
  . . . // 进行 f()的前条件检查
  int retVal = f_unchecked(name, ppVal, pLen);
  . . . // 进行 f()的后条件检查
  return retVal;
}
int f_unchecked(char const *name, Value **ppVal, size_t *pLen)
#endif /* ACMELIB_DBC */
{
  . . . // f的语义
}
```

这完全算不上优雅，不过它可以工作，并可以很容易合并到代码生成器中。当处理被重写的（overridden）类成员函数时，问题可能要稍微复杂一点，因为你要面对是否实施父类的前条件和后条件的问题。这得条分缕析后才能决定，已经超出了我们的讨论范围。 [注 6]

[注 6]在这一点上，我承认我有点胆小自私，不过我有很好的借口。即便是在成熟运用 DbC的语言中，对于继承体系中的层与层之间的关联契约的用处（事实上是机制）仍然是模棱两可的。此外，为 C++加入 DbC的提议直到本书的撰写时仍然不过是纳入考虑而已 [Otto2004]，因此，我认为在这儿过多地在细节上饶舌没有什么好处。

1.3.3 类不变式

在 C++中，实现类不变式几乎和实现前条件一样简单。我个人的做法是为类定义一个名为 is_valid()的方法，像这样：

```
template<. . . >
inline bool pod_vector<. . .>::is_valid() const
{
  if(m_buffer.size() < m_cItems)
  {
    return false;
```

```
  }
  . . . // 这儿进行进一步的检查
  return true;
}
```

然后，该类的每个公有方法都把它放在断言里进行调用，在进入方法时断言一次，退出方法前再来一次。我喜欢在紧接着前条件检查之后进行类不变式的检查（见 1.3.1节）：

```
template< . . . >
inline void pod_vector<. . .>::clear()
{
  assert(is_valid());
  m_buffer.resize(0);
  m_cItems = 0;
  assert(is_valid());
}
```

作为一个替代策略，我们可以将断言放在不变式函数自身之中。然而，除非你手头拥有的是一个“久经考验”的断言（见 1.4节），否则这会令你不得不选择提供关于“肇事”的条件或方法的断言信息（文件+行+消息）。我倾向于后者，因为违反不变式毕竟是非常少见的情况。不过，你可能会选择前者，如果是那样的话，你可能希望将断言放到 is_valid()成员函数中。

事实上，对此存在一个合理的折中方案，我通常在具有良好的日志/跟踪界面的环境中使用这种策略（见 21.2节），具体做法是在 is_valid()成员函数中记录违反不变式的细节，并且让“肇事”成员函数（译注：而非不变式函数）来触发该断言。

与输出参数和返回值检查不同，使用 RAII（见 3.5节）来使类不变式的检查自动化还是相当容易的（这种检查也作为方法退出前的后条件验证的一部分），像这样：

```
template< . . . >
inline void pod_vector<. . .>::clear()
{
  check_invariant<class_type> check(this);
  m_buffer.resize(0);
  m_cItems = 0;
```

}

缺点是，强制会在 check_invariant模板实例的构造函数和析构函数中被实施，这意味着使用预处理器来获悉 __FILE__和 __LINE__信息的简单的断言可能会给出误导信息。然而，要想实现一个可以正确显示断言失败位置的“宏-模板”的断言形式并不算是很大的挑战，甚至可以结合运用非标准的 __FUNCTION__预处理符号（当然，对于那些支持它的编译器而言）。

1.3.4 检查？总是进行！

在 [Stro2003]中， Bjarne Stroustrup 做了一个非常重要的观察：不变式只对那些具有方法的类才是必要的，而对于仅仅作为变量聚合体的简单结构而言是没有必要的（例如，我们将会在 4.4.2节看到的 Patron 类型就不需要不变式）。在我看来，这话还可以这么说：任何具有方法的类都应该具有类不变式。不过，在实践中对此有一个下限。如果你的类持有一个指向某些资源的指针，那么它要么是 NULL，要么不是 NULL。除非你的类不变式方法可以使用非空指针所指向的有效的外部资源，否则你的类不变式将无事可干。在这种情况下，是否使用一个“存根（stub）”类不变式取决于你自己，或者你也可以干脆什么都不干。但如果你的类将来会不断升级，那么在里面放上一块有待以后扩充的“存根”方法可以令后续的精化工作变得容易一些。如果你使用了某种代码生成器的话，我建议你总是用它来生成类不变式，并生成对所生成的类不变式的调用。

类不变式较之散落在类实现周围的一堆断言而言，好处是非常明显的。类不变式使你的代码更容易阅读，并且在不同的类的实现之间具有一致的外观，以及具有更好的可维护性，这是因为对于每个类你都把类不变式定义在了某个单一的地方。

1.3.5 DbC还是不 DbC?

到目前为止，我所描绘的关于运行期契约的蓝图其实隐含了一个假定，那就是：在进行适当的测试后，人们会对他们的系统进行一次构建（build）（译注：对程序进行编译和连接的过程），在这次构建中， DbC元素都被预处理器消去（译注：其实通常就是发行版（release）的构建，其中 assert(exp)会展开为空）。

事实上，关于“是否任何构建（build）都应该不实施 DbC”这个问题 [Same2003]，仍然颇有争议。一个论据是（借用 [Same2003]里的逻辑） DbC 里的契约实施就好比电力系统中的保险丝，任何人都不应该在部署一个成熟的电力设备之前把它里面的所有保险丝都拔掉。

断言和保险丝之间的区别在于前者涉及运行期测试，而测试的代价明显不为零。尽管保险丝中的合金成分的电阻可能与它所在系统中的其他部分的电阻略有差别，然而这跟断言引入的代价相比仍然无法相提并论。

我的看法是，这需要仔细分析才能求得一个良好的平衡。这就是为什么本节的例子代码中包含了 ACMELIB_DB 这个符号的缘故。我没有使用 NDEBUG（或者 _DEBUG），因为 DbC 的使用不应该直接和“调试版/发行版（debug/release）”的二进制概念耦合起来。究竟何时使用它，何时消除它，取决于你自己。[注 7]

[注 7]在 ISE Eiffel 4.5中，你无法去掉前条件，大概是因为前条件可以在程序变成未定义状态之前进行反馈，从而对于程序捕获违反前条件的异常并继续执行是有意义的。

1.3.6 运行期契约：尾声

尽管我们已经看到 C++在后条件方面是有缺陷的，然而进行前条件和类不变式的测试仍然是合理的。在实践中，将这两者结合使用往往能发挥 DbC大部分的威力。对返回值和输出参数的后条件测试的能力缺失虽然令人遗憾，但也并非十分严重的事情。如果你必需这种能力的话，你可以求助于预处理器，就像在 1.3.2 小节中看到的那样。

如同约束一样，对于不变式，我们可以通过使用一个间接层让日子好过一些。这个间接层对于约束来说是一个宏，而对于不变式来说则是一个成员函数。正因为如此，提供对新的编译器的支持或者修改某个类的内部实现也变得更为容易了，并且，我们还把该机制不爽的那一面全部隐藏到了类不变式方法中。

1.4 断言

在我看来，断言并非一个良好的报错机制，因为它们通常在同一个软件的调试版和发行版中的行为有着极大的差异。虽说如此，断言仍然是 C++程序员确保软件质量的最重要的工具之一，特别是考虑到它被使用的程度和约束、不变式一样广泛。任何关于报错机制的文档如果没有提到断言的话肯定不能算是完美的。

基本上，断言是一种运行期测试，通常仅被用于调试版或测试版的构建，其形式往往像这样：

```
#ifdef NDEBUG
# define assert(x)  ((void)(0))
#elif /* ? NDEBUG */
extern "C" void assert_function(char const *expression);
# define assert(x)  ((!x) ? assert_function(#x) : ((void)0))
#endif /* NDEBUG */
```

断言被用于客户代码中侦测任何你认为绝不会发生的事情（或者说，任何你认为永远不会为真的条件式）：

```
class buffer
{
  . . .
  void method1()
  {
    assert((NULL != m_p) == (0 != m_size));
    . . .
  }
private:
  void    *m_p;
  size_t  m_size;
};
```

buffer类中的断言表明类的作者的设计假定：如果 m_size不是 0，那么 m_p也一定不是 NULL，反之亦然。

当断言的条件式被评估为 false时，断言就称为被“触发”了。这时候，或者程序退出，或者进程遇到一个系统相关的断点异常，如果你处于图形界面操作环境中的话，你往往还会看到弹出了一个消息框。

不管断言是如何被触发的，将失败的条件表达式显示出来总是很好的，并且，既然断言大部分时候是针对软件开发者而言的，那么最好还要显示它们的“出事地点”，即所在的文件和行号。大多数断言宏（assertion macros）都提供这个能力：

```
#ifdef NDEBUG
# define assert(x)  ((void)(0))
#elif /* ? NDEBUG */
extern "C" void assert_function( char const *expression
                               , char const *file
                               , int        line);
# define assert(x)  ((!x)
                    ? assert_function(#x, __FILE__, __LINE__)
                    : ((void)0))
#endif /* NDEBUG */
```

因为断言里的表达式在发行版的构建中会被消去，所以确保该表达式没有任何副作用是非常重要的。如果不遵守这个规矩的话，你往往会遇到一些诡异而令人恼火的情况：为什么调试版可以工作而发行版却不能

呢？

1.4.1 消息！消息！

断言所采取的行动可能五花八门。然而，大多数断言实现都使用了其条件表达式的字符串形式。这种做法本身没什么不对，不过可能会令可怜的测试者（可能就是你）陷入迷惘，因为你所能得到的全部信息可能简洁得像这样：

```
"assertion failed in file stuff.h, line 293: (NULL != m_p) == (0 != m_size));"
```

不过，我们还是可以借助这个简单的机制使我们的断言信息变得更丰富一些。例如，你可能会在 switch语句的某个永远不可能到达的 case分支中使用断言，这时候你可以通过使用一个值为 0的具名常量来显著改善断言的信息，像这样：

```
switch(. . .)
{
  . . .
  case CantHappen:
   {
      const int AcmeApi_experienced_CantHappen_condition = 0;
      assert(AcmeApi_experienced_CantHappen_condition);
      . . .
```

现在当这个断言被触发时，它所给出的信息要比下面所示的更具有描述性：

```
"assertion failed in file acmeapi.cpp, line 101: 0"
```

还有一个办法可以用来提供更丰富的信息，同时还可以免除前一种方法中的变量名称中有大量下划线的不爽。因为 C/C++会把指针隐式地解释（转换）为布尔值（见 15.3节），所以我们可以借助于“字面字符串常量可被转换为非零指针并进而被转换为 true”这个事实，把一则易于阅读的信息和断言的测试表达式进行逻辑与运算：

```
#define MESSAGE_ASSERT(m, e)  assert((m && e))
```

像这样使用它：

```
MESSAGE_ASSERT("Inconsistency in internal storage. Pointer should be null when size is 0, or non-null when size is non-0", (NULL != m_p) == (0 != m_size));
```

这下我们所能得到的失败信息可就丰富多了。另外，因为那个字符串本身就是条件表达式的一部分，所以在发行版的构建中会被消去。也就是说，你可以随心所欲地给出任何附加的信息！

1.4.2 不恰当的断言

断言对于调试版构建中的不变式检查是有用的。只要你谨记这一点，你就不会错得太离谱。

唉，我们看到太多把断言误用在运行期查错中的情形了。一个典型的例子是把它用在检查内存分配失败中（这可能会出现在大学一年级的程序查错材料中）：

```
char *my_strdup(char const *s)
{
  char *s_copy = (char*)malloc(1 + strlen(s));
  assert(NULL != s_copy);
  return strcpy(s_copy, s);
}
```

你可能会觉得没有人会这么干。如果你是这么认为的，我建议你使用 grep（译注：一种可在文件内进行字符串查找的工具）去你最喜爱的一些库里查一查，其中你会看到用断言检查内存分配失败、文件句柄以及其他运行期错误的代码。

这么做错也就错了罢，不幸的是中间偏偏还有个“半吊子”，也就是说，有不少人会将断言和正确处理错误情况的代码放在一起使用：

```
char *my_strdup(char const *s)
{
  char *s_copy = (char*)malloc(1 + strlen(s));
  assert(NULL != s_copy);
  return (NULL == s_copy) ? NULL : strcpy(s_copy, s);
}
```

我实在无法理解这种做法！考虑到大部分人都是在拥有虚拟内存系统的桌面硬件上做开发的，在这种环境下调试，除非你被限制在一个低内存量的配置机制下，或者你被规定运行时库的调试 API 具有低内存量的行为，否则你几乎不可能感受到内存异常的存在（译注：即几乎不可能遇到内存分配失败或内存不足的情况）。

不过，若是把“内存分配”换成其他更容易“闯祸”的举动，则事情会看得更明白一些。例如，若用于文件句柄，这就是在提醒你：把你的测试文件放到正确的地方，而不要试图把错误反馈功能武装到坚不可摧。几乎可以肯定地说，错误反馈能力到了实际部署中总会不够用。

还有，如果问题在于一个运行期的失败条件，那么为什么你要在一个断言中捕获它呢？如果你在下面的运行期错误处理的编码上出了错误，难道你不想让程序崩溃掉从而更能体现发行模式的真实行为吗？退一步说，即便你手中握着一个“超级智能”的断言 [Robb2003]，这仍然只会为你自己以及评审你的小组的人树立一个糟糕的例子。

[译注]作者这里讲的相当简略。具体的意思是：如果你在前面使用一个断言捕获了错误，那么即使下面又有运行期错误处理的代码，也将得不到调用（调试期）。也就是说，即便你在后面的错误处理的编码上出了错误，也会因为在调试时执行流被上面的断言“截断”而无法发现（错误的编码只要不被执行当然也就不会露出马脚了）。然而这种错误到了发行版又会立即显出狰狞的面目，因为在发行版中断言会失效。总的来说，这就导致了两个问题：第一，你的编码错误在调试期被前面的断言掩盖了起来；第二，调试版和发行版的错误处理行为不一。这两点都可能会带来相当大的困惑。

在我看来，将断言应用到运行期的失败条件式身上，即便后面跟着发行模式下的处理代码，也最多只会分散注意力而已，说得严重一点，这是错误的编程实践。不要那样做！

建议：使用断言来断言关于代码结构的条件式，而不要断言关于运行期行为的条件式。

1.4.3 语法以及 64位指针

另一个问题 [注 8]是有关在断言中使用指针的。在 int是 32位而指针是 64位的环境中，如果把原生指针用在断言中的话，根据 assert()宏的定义，将会导致一个截断警告 [注 9]：

```
void *p = . . .;
assert(p); // 警告：截断
```

[注 8]我知道，我把什么东西都一股脑儿塞到这一部分来了，不过我觉得这些东西值得你去了解。

[注 9]以前用 Dec Alpha的时候我曾遇到过这个问题，并且，我在新闻组上看到有人在其他平台上有过类似的经验。

当然，这也是我在 17.2.1节的话题之一，并且实际上也是引起我对有关布尔表达式的恼人问题“过敏”的原因。答案是在你的代码中明确地表达意图：

```
void *p = . . .;
assert(NULL != p); // 现在漂亮多了！
```

1.4.4 避免使用“verify()”

不久前我和别人谈论关于自己的断言宏的定义问题，他们想把它命名为 verify()，以避免跟标准库宏冲突。唉，这么做有两个问题：

首先，VERIFY()是 MFC中一个广为人知的宏，其用途和 assert()大致相同，不过它的条件式不会被消去，从而在任何场合下都会执行。它的定义如下：

```
#ifdef NDEBUG
# define verify(x)  ((void)(x)) /* x仍然存在 */
#elif /* ? NDEBUG */
# define verify(x)  assert(x)
#endif /* NDEBUG */
```

如果把 verify()宏的行为定义成跟 assert()一样的话，那些习惯于已经建立的“verify”的行为的人们肯定会对此感到相当困惑：为什么在调试期还好端端的代码，一到发行模式下就失败了呢？他们也许需要愣一阵子才能意识到 verify宏在发行模式下不起作用，不过一旦回过神来，他们可能就会晃着扳手从停车场那边向你追过来。

第二个问题是，单词“assert”只可以为“断言”所用。这是因为你可以使用 grep和类似的工具几乎无歧义地查找到你的断言，也是由于它对程序员而言是如此扎眼。当在代码中看到它们时，人们立刻就会意识到“某些不变式（见 1.3.3节）正被测试”。现在如果将断言冠以另外一个名字来使用的话，只会把原来相对清晰的概念搞成一团糊涂浆。

我以前曾经定义过自己的 verify()宏，它的语义和 MFC的 VERIFY()相同，现在想起来觉得这是个危险的举

动。交给断言的表达式必须没有任何副作用，而坚持这一点并不算太困难，我已经好几年没有犯这个错误了。不过，如果你混合使用了不同语义的断言宏，其中有些必须有副作用，而有些必须没有副作用，那么你很容易就会被搞混淆，并且非常难以建立一个稳当的习惯。现在我已经不再使用任何形式的 verify()宏了，我建议你也要这样。

1.4.5 为你的断言命名

既然命名问题在上一节被提了出来，那么让我们现在就来解决它。正如我曾提到的，一个断言宏首先应该包含单词“ assert”。我见过或用过的就有： _ASSERTE()、 ASSERT()、 ATLASSERT()、 AuAssert()、 stlsoft_assert()、 SyAssert()，等等。

C和 C++中的标准断言宏被称为 assert()。在 STLSoft库中，我有自己的 stlsoft_assert()以及其他一些宏，所有这些都是小写的。在 Synesis库中，断言宏被命名为 SyAssert()。在我看来，这些做法都是不合适的。

根据惯例，所有的宏都应该是大写的，这是个非常好的习惯，因为这么一来，宏就可以跟函数和方法醒目地区分开来。尽管将 assert()写成一个函数是完全可行的：

```
// 假定只被 C++编译器所用
#ifdef ACMELIB_ASSERT_IS_ACTIVE
extern "C" void assert(bool expression);
#else /* ? ACMELIB_ASSERT_IS_ACTIVE */
inline void assert(bool )
{}
#endif /* ACMELIB_ASSERT_IS_ACTIVE */
```

之所以不考虑这么做，是因为它不具有目前的断言宏的许多优点。首先，如果这么做的话编译器就无法将断言表达式优化掉。好吧，严格一点说，在某些情况下这种优化还是可能的，不过优化不能完全进行，即便编译器有最好的优化能力也不行。不管编译器和项目之间存在什么精确的调整，原则上这总是会带来大量的垃圾代码。

另一个问题是某些类型不能被隐式地转换为 bool或 int，或转换为被你选作表达式类型的类型。因为标准的 assert()宏可能会把接受到的表达式放到 if/while 语句或 for 循环的条件表达式或三元条件操作符（?:）中，进而所有通常的隐式布尔转换（见 13.4.2节和第 24章）都会参与进来。这和将表达式传给接受 bool或 int的函数有相当大的差别。

最后一个原因是：如果将 assert()实现为函数，那么就无法将表达式作为断言的错误信息的一部分于运行期显示出来，这是因为“字符串化”能力是预处理器的重要能力之一，而不是 C++（或 C）语言的。

目前断言是以宏的形式存在的，并且可能一直以这种形式存在下去。所以，它们应该是大写的。这并不仅仅因为那是个一致的编码标准，也是因为大写更醒目，从而可以使我们的编码生活轻松一些。

1.4.6 避免使用 #ifdef _DEBUG

在 25.1.4节我提到，由于性能上的原因，default条件被排除在 switch语句之外，而使用一个断言来代替它的位置。我所尊敬的一个审稿人具有和我差异相当大的背景，他对此表示怀疑，并建议我应该做得更简洁一些：

```
switch(type)
{
  . . .
#ifdef _DEBUG
  default:
    assert(0);
    break;
#endif // _DEBUG
}
```

这说明，即使是最具经验的程序员也容易成为身处的开发环境的牺牲品。这段代码有几个小错误。首先，assert(0)所给出的错误信息可能会相当贫乏，这取决于编译器对断言的支持。这个问题容易解决：

```
. . .
default:
  { const int unrecognized_switch_case = 0;
  assert(unrecognized_switch_case); }
. . .
```

不过，在大多数编译器中，这跟最初的冗长形式相比信息量仍然不够：

```
assert( type == cstring || type == single ||
```

```
        type == concat || type == seed);
```

使用 _DEBUG的最主要问题还在于，它并非指示编译器去生成断言的明确符号。首先，根据我的经验，_DEBUG只在 PC编译器上流行。对于许多编译器而言，调试模式是缺省的构建（build）形式，只有定义了 NDEBUG才会导致编译器进入发行模式，并且将断言消去。显然，正确的途径是使用一个编译器无关的抽象符号来控制构建模式，例如：

```
#ifdef ACMELIB_BUILD_IS_DEBUG
  default:
    assert(0);
    break;
#endif // ACMELIB_BUILD_IS_DEBUG
```

然而即便是这些也不能算是问题的全部。通常，在产品的预发布版中保留一些调试功能是完全合理的。因此，你可能需要使用自己的、独立于 _DEBUG、NDEBUG甚至 ACMELIB_BUILD_IS_DEBUG的断言。

1.4.7 DebugAssert()和 int 3

尽管这是特定于 Win32+Intel架构上的东西，不过还是值得注意的，因为它非常有用，并且让人吃惊地鲜为人知。Win32 API函数 DebugBreak()会在调用它的进程中引发一个断点异常。这种能力允许一个独立的进程被调试，或者使你的 IDDE（译注： Integrated Development and Debugging Environment，集成开发与调试环境）中的当前调试进程暂停运行，从而允许你去查看调用栈（call stack），或体验其他调试方面的迷人功能。

在 Intel架构上，该函数只是简单地执行“ int 3”机器指令，这在 Intel处理器上会引发一个断点异常。

一个小小的遗憾是，当控制流程转到调试器时，执行点落在 DebugBreak()内部，而不是友好地落在引发异常的代码上（译注：作者的意思大概是，没有落在调用 DebugBreak()的用户代码上）。解决这个问题的一个简单方式是在 Intel架构下采用内嵌汇编。Visual C++运行时库提供了 _CrtDebugBreak()函数作为它的调试基础设施的一部分，该函数在 Intel架构中定义如下：

```
#define _CrtDbgBreak() __asm { int 3 }
```

使用“ int 3”指令意味着调试器会精确地停在它被需要的点上，也就是说，精确地停在“肇事”的那行代码上。

1.4.8 静态/编译期断言

到目前为止我们只看过了运行期断言。然而，如果可能的话，最好还是在编译期就把错误抓住。在本书中的许多部分，我们都提到了静态断言，它也被称为编译期断言，因此现在正适合把它们详细地描述一下。

基本上，静态断言提供了一个编译期对表达式进行验证的机制。不消说，为了能够在编译期得到验证，表达式当然必须能够在编译期进行求值。这就缩小了可以被编译期断言所作用的表达式的范围。例如，你可以使用编译期断言来确保你对 int和 long的大小的期望对于当前编译器而言是正确的：

```
STATIC_ASSERT(sizeof(int) == sizeof(long));
```

不过要注意，它们不能对运行期表达式进行求值：

```
. . . Thing::operator [](index_type n)
{
  STATIC_ASSERT(n <= size()); // 编译错误！
  . . .
```

触发静态断言的结果是无法通过编译。由于静态断言跟大多数 C++（和 C）的现代特性一样，本身并不是语言特性，而是其他语言特性的某种副作用，所以错误信息的含义不会那么明显直白。我们马上就会看到它们会怪异到什么地步。

通常，静态断言的机制是定义一个数组，并将表达式的布尔结果作为数组大小。由于在 C/C++中， true可以被转换成整数 1， false则转换为 0，因此表达式的结果可被用于定义一个大小为 1或 0的数组，而大小为 0的数组在 C/C++中是不合法的。因此，如果表达式的布尔结果为 false，则编译便不能通过。考虑下面的例子：

```
#define STATIC_ASSERT(x)   int ar[x]
. . .
STATIC_ASSERT(sizeof(int) < sizeof(short));
```

int的大小永远不会小于 short（C++-98:3.9.1;2），因此表达式 sizeof(int) < sizeof(short)的评估结果恒为 0。从而，上面的那行 STATIC_ASSERT()被求值为：

```
int ar[0];
```

而这在 C/C++中是非法的。

很明显，上面的实现存在诸多问题。数组 ar被声明了，却并没有被使用，这将会导致大部分的编译器给出一个警告，阻止你的构建（build）过程[注 10]。再者，在同一个作用域中使用 STATIC_ASSERT()两次或两次以上将会导致 ar被重复定义的错误。

[注 10]你确实把警告级别设成“高（high）”，并把它们当成错误来对待了，对吗？

为了解决这些问题，我把 STATIC_ASSERT()定义成这样：

```
#define STATIC_ASSERT(ex)  \
        do { typedef int ai[(ex) ? 1 : 0]; } while(0)
```

这在大多数编译器中都工作得不错。然而，有些编译器对大小为 0的数组却姑息纵容，因此需要一些条件编译来处理这些情况：

```
#if defined(ACMELIB_COMPILER_IS_GCC) || \
    defined(ACMELIB_COMPILER_IS_INTEL)
# define STATIC_ASSERT(ex)  \
        do { typedef int ai[(ex) ? 1 : -1]; } while(0)
#else /* ? compiler */
# define STATIC_ASSERT(ex)  \
        do { typedef int ai[(ex) ? 1 : 0]; } while(0)
#endif /* compiler */
```

无效数组大小并非实现静态断言的唯一途径。我知道还有另外两种有趣的机制[Jagg1999]，尽管这里我并没有使用它们。

第一个机制依赖于这么一个事实：switch语句的每个 case子句都必须对应于不同的值：

```
#define STATIC_ASSERT(ex) \
    switch(0) { case 0: case ex:; }
```

第二个机制则依赖于“位域（bitfield）必须具有非零长度”的事实：

```
#define STATIC_ASSERT(ex) \
    struct x { unsigned int v : ex; }
```

以上三种形式的静态断言在触发时给出的错误信息都很让人费解。你会看到诸如“case label value has already appeared in this switch”或“the size of an array must be greater than zero”之类的信息，因此处于嵌套模板的情形时，你得花上一会儿才能弄明白是哪儿出了问题。

为了改善这种混乱的状况，Andrei Alexandrescu 在 [Alex2001] 中描述了一种技术，用于提供更好的错误信息，并且在现有语言的限制下竭尽所能发展这种技术。[注 11]

[注 11] 你应该查看一下该技术，它非常迷人。

对于我自己而言，我倾向于避开那么做带来的复杂性，这是基于三个原因。第一，我比较懒，总是倾向于尽量避开复杂性 [注 12]。第二，我写的 C 代码和 C++ 代码一样多，我更喜欢对于这两种语言都有效的设施。

[注 12] 我还认为你提供的技术越简单越好，但由于本书中有许多东西颇费脑细胞，所以我没法把这作为一个严格的理由，只是因为我懒而已。

最后，静态断言是由于在编码时误用了某个组件而触发的。这就意味着它们不但少见，而且局限于导致它们被触发的程序员圈子内。因此，我认为开发者只需少量的时间就可以找到错误之所在（虽然不一定花上少量的时间就可以找到解决的办法），他们很少会介意这一丁点儿代价。

在我们结束这个条款之前，还有一件值得注意的事情，那就是：在实现静态断言时，“无效数组大小”和“位域”技术具有一个“switch”技术所不具备的优点：前二者可以被用在函数之外，而后者则不能（运行期断言同样不能）。

1.4.9 断言：尾声

本节描述了断言的基础知识，当然还有更多断言可做的有趣事情，不过它们已经超出了本书的讨论范围。另外，还有两个相当不同但同样有用的技术，即 SMART_ASSERT [Torj2003] 和 SUPER_ASSERT [Robb2003]，我建议你去研读一下关于它们的资料。

第 7章 ABI

7.1 共享代码

假定你实现了一个包含有一些有用的类的库，希望将它提供给其他程序员使用。此外，让我们假定这些“其他程序员”中的某些人正使用着其他操作系统和（或）编译器——不同于你开发该库所使用的操作系统和编译器。那么，你应该如何怎么做呢？

最明显的（尽管可能并不是最恰当的）解决方案是将所有的源代码都分发出去，让需要使用该库的那些人自己去构建（build）。从理论上说，这种方式很不错，因为所有编译器都应该能够处理那些 C++代码，不是吗？然而遗憾的是，正如你可能已经经历过的那样，或者至少在本书中看到过的那样，C++是如此复杂，以至于在各个不同的编译器之间存在着许多“地方性的差异”。这就导致了一个结果：如果你想让你的源代码能够在其他编译器上顺利地通过编译，通常你需要做相当数量的辅助工作（译注：例如预编译处理），尤其当涉及到模板元编程（template meta-programming，TMP）这类较新的技术时。如果你的代码将被用于其他操作系统之上，那么你几乎无可避免地需要针对新的平台进行代码移植工作，包括修改各种调用、算法、甚至整个子系统等等，因此二进制兼容性的问题就摆上了议事日程。不过，让我们暂且先将这些问题统统放在一边，并“规定”源代码级的共享从原则上说是相对直截了当的方式，即使在现实中有时并非如此。[注 1]

[注 1]现实是残酷的。我的一个朋友所在的开发小组就曾花了整整四个月，将能够在三种 UNIX变种上正确工作的代码移植到第四个变种上。靠！

仅提供二进制形式的好处不仅在于它能够保护知识产权，还在于你能够保留对代码的任何修改或升级的控制权。这种方式的缺点在于，使用者为了从你的升级或修改中获益必须先获取改进后的版本，然后对项目重新进行连接。另一方面，当你提供的是源代码时，没有任何办法阻止你的客户进行他们自己的修改和改进（也可能是改的还不如你写的版本）。除非他们将所做的修改告知你，并且你将这些修改合并到新版本中，否则下次当他们升级你的库时就会发现他们以前所做的修改都被覆盖掉了，又得重做！

不管是由于偏执、知识产权保护、方便性、害羞还是其他什么原因，你也许并不乐意将库的内部实现暴露给外界，而希望最好能以二进制的形式提供出来。在理想情况下，你可以将你的实现文件编译成二进制库，然后将它跟合适的头文件一起提供给客户。你的客户只需在他们的客户代码中 #include这些头文件，并且连接你的二进制库就可以了。还有比这更简单的做法吗？

唉，可惜这个乐观的画面在现实中并不存在。要实现这一点，就必须对你提供的头文件中所描述的类型、函数和类的编译后的形式和机制有一个统一的表示。这种东西就是所谓的应用程序二进制接口

（Application Binary Interface），简称 ABI，可惜 C++里却没有！

Imperfection: C++标准中并没有定义 ABI，这导致在不同平台上的不同编译器之间存在着大量的（二进制）不兼容。

近来在 C++ ABI 标准化方面也有一些进展，其中较为突出的是 Itanium C++ ABI [Itan-ABI]，但从总体上来说，这个缺陷仍然存在。

因为 C++是一门“编译为机器代码”的语言，所以在某个平台上编译的二进制可执行体并不需要拿到其他平台上还能执行，因此，说“C++在既定的平台上缺乏 ABI”更为精确一点，而“平台”的概念则是架构和操作系统的组合。本章我们将要考察的是在 Win32-Intel x86平台上碰到的问题。从某种意义上来说，微软为 Windows应用程序开发提供了一个事实上的标准，因为它上面的编译器至少必须能与 Win32系统库中的 C函数进行交互。但这不能算是真正的 ABI，而且这个平台上的众多编译器之间存在着大量的不兼容。

对于一门像 C++这样复杂的语言来说，若要想支持 ABI，必然有许多方面需要标准化，包括对象布局（包括基类）、调用约定与名字重整（name mangling）、虚函数机制、静态对象实例化、C++语言支持（例如 set_unexpected()、operator new[]()等）、异常处理与运行时类型信息（RTTI，C++-98: 18.5），以及模板实例化机制。另外，还有一些不那么明显的问题，比如运行时库内存调试设施，虽然从实质上而言它本身并非语言的一部分，但对编程实践却有着重大的影响。

在得不到指导和控制的情况下，不同的编译器厂商无可避免地在 C++（和 OO）运行时的形式、行为以及基础设施的实现上有所不同。不管是出于自然的分歧，抑或是深思熟虑的商业策略，这都不在本书讨论范围之内。然而，无论出于什么原因，有一点是肯定的：这是个大问题。我的一个好朋友 [注 2]这样总结道：

[注 2]呜呼，他已经不再是 C++方面的朋友了，就是因为这一点以及 C++的其他一些“重大”缺陷，我们分道扬镳了。

> 在我看来，C++最大的缺陷就在于缺乏一个有关 ABI 的标准。造成直接后果之一是，使用 GCC编译的 C++代码无法与使用 MSVC编译的 C++代码合作，除非你为它们提供“C”接口（即使用 extern “C”连接属性）。这么一来，倘若 C++开发者希望使代码更具一般性和复用性，就必须在某些地方借助于“C”连接了。

某种程度上他是正确的。如果我提供了一个采用某种编译器构建（build）的二进制库，那么很可能它与另一种编译器编译的代码无法兼容。换句话说，只分发库的一种二进制版本是行不通的，我可能需要对潜在的客户可能使用的所有编译器都提供一个版本。

自然，某些编译器厂商的商业统治带来了某种程度上的兼容性。举个例子，我能够在 CodeWarrior和 Intel架构上编译 C++库，然后将编译输出的二进制形式文件连接到 Visual C++创建的可执行文件中。但是，一个部分的解决方案等于没有解决方案，除非你愿意将自己限制在单个或很小的一组编译器中。

只有当统一的平台 ABI（例如 Itanium-ABI项目）成为标准时，我们才不用面对我那位博学的朋友所指出的痛苦的问题。本章以及下一章承认问题的确存在，但并不认为它们像我的朋友所说的那样属于彻底的失败。我们将会看到缓解的措施。

7.2 C ABI需求

由于 C++几乎是 C的一个完全的超集，所以任何 C++ ABI都包含 C ABI。因此，在考察 C++ ABI时，第一步自然是考察 C相关的议题。

和 C++相比， C是一门简单得多的语言，这也正符合 C的精神（见序言）。 C里面没有类、对象、虚函数、异常处理、运行时类型信息以及模板。

然而，前面指出的问题中仍然有几个是与 C相关的。 C里面有结构，而编译器可以按照自己的标准自由地对结构的内存布局进行对齐，除非我们通过编译器相关的选项和 pragma给出自己的 pack指令（见 8.2节）。

即便是运行期调用函数的方式或者连接（编译）期为函数命名的方式也是五花八门的。所有 C函数都服从一种或多种调用约定，而调用约定在不同的编译器之间可能也可能不兼容，具体取决于平台。类似地，编译后的函数名字，即符号名（ symbol name），不同的编译器也各不不同。在 Win32操作系统上的编译器之间，这两个问题则尤为明显，甚于其他系统，如 UNIX。这也正是本章我选择 Win32编译器作为讨论对象的缘由之一。那些具有纯粹的 UNIX背景的人可能会长吁一口气，因为在他们的操作系统上这些问题比较小。让我们关心一下那些在 Win32下试图实现 C++（和 Q）的二进制互操作性的可怜的程序员吧。

尽管 C比 C++要来得简单得多，然而 C在它的语言支持上也存在一些问题，例如 atexit()的可重入语义 [Alex2001]。这就意味着 C ABI会依赖于不同编译器对此类库功能的公共解释。

7.2.1 结构布局

C中的内存布局问题理解起来相当直观。考虑下面的结构：

```
struct S
```

```
{
  long  l;
  int   i;
  short s;
  char  ch;
};
```

该结构的大小完全是由实现定义的。在实践中，它依赖于特定编译器的 pack约定， pack的算法总是很简单的。如果该结构按 1字节对齐（ alignment）来 pack的话，那么四个成员将会连续地紧跟在一起。假定 long、 int、 short、 char的大小分别是 8、 4、 2、 1字节，那么该结构的大小就是 15字节。然而，如果按 2字节对齐来 pack的话，该结构大小就变成了 16字节。如果按 4字节对齐，就是 24字节， 8字节（或大于 8字节）对齐则是 32字节。

如果两个编译器使用了不同的 pack对齐惯例，那么它们创建的二进制组件将不是二进制兼容的。事实上，pack对齐可以由编译选项来指定，因而即便是同一个编译器编译的二进制组件之间也很可能存在不兼容性。

7.2.2 调用约定、符号名以及目标文件格式

即使编译器必须或被指定使用相同的结构 pack对齐方式，你仍然还需要面对两个问题：当把若干二进制组件连接到一起时的符号名和调用约定问题。

早期 C在 UNIX[Lind1994]上的时候，所有编译器都使用相同的调用约定，就是为人所熟知的“ C调用约定”，即参数从右至左被 push到栈上去，并且栈由调用方来清理 [注 3]。后来其他编译器，特别是 Windows平台上的编译器提供了其他的调用约定，例如 stdcall（ standard call，标准调用）、 pascal以及 fastcall等等。在这类环境中， C调用约定被称为 cdecl。如果两个编译器使用了不同的调用约定，那么它们生成的二进制组件也是不兼容的。

[注 3]这正是该调用约定之所以能够支持可变参数列表的原因，例如 int printf(char const*,...); 因为调用上下文“知道”有多少实参被传递了，而函数本身通常对此一无所知（译注：所以堆栈由调用方来清理）。

一个相关的问题是关于目标文件中符号的名字。目标文件是指编译单元经编译而生成的二进制文件（见序言 0.4）。符号名是指具有外部连接的函数和变量在编译生成的目标文件中的二进制名字，为的是在程序被连接时，不同的符号可以被定位并连接在一起。古典的做法是将函数名本身作为其对应的符号名 [Ken1999]。许多编译器厂商都为 C++定义了他们自己的符号命名模式，这也是导致 C++在二进制兼容性方面表现不佳的

缘由之一（7.3.3 节）。此外，某些编译器还会基于构建（build）的线程模型而提供不同的符号名，从而进一步使情况复杂化。（译注：比如 MSVC）

最后，目标文件还可能具有不同的格式，这也是 Win32平台上不兼容性的另一来源。存在着好几种不同的目标文件格式，如 OMF、COFF等等，它们通常都是不兼容的，尽管大多数编译器都支持不止一种格式。

7.2.3 静态连接

在实践中，有些编译器能够共享同样的静态库。表 7.1展示了一些 Win32编译器在 C静态库的构建和使用方面的兼容情况。如你所见，它们中有些能够兼容，但从整体来看差强人意。在收集这些信息时我对每个编译器用的都是它们的缺省编译选项。尽管你还可以采用其他一些手段来提高兼容性，但是不管怎样都无法确保完全的互操作性。

表 7.1 Win32上的静态连接兼容性（C代码）。#表示一个可兼容的组合。

用来生成库的编译器	使用库的编译器					
	Borland	CodeWarrior	Digital Mars	GCC	Intel	Visual C++
Borland	#					
CodeWarrior		#				
Digital Mars			#			
GCC		#		#	#	#
Intel		#		#	#	#
Visual C++		#		#	#	#

显然，ABI的缺乏导致了提供单一版本的静态连接库成为不可能，不仅是在 Win32平台上，其他平台上也存在类似的不兼容的情况，尽管 Itanium ABI的标准化意味着 GCC和 Intel可以在 Linux上合作。对于这个问题，有一个索然无趣的解决方案：对于你的客户可能使用的任何一种编译器，你也需得到它或与它兼容的其他编译器，并使用它们分别创建目标库。因此，你可能需要创建 mystuff_gcc.a（对于 Win32则是 mystuff_gcc.lib）和 mystuff_cw.a（对于 Win32则是 mystuff_cw.lib）等等。实际的障碍是你不大可能得到你所需要的所有编译器，特别是当你需要支持几个不同的操作系统的时候。即使你得到了，维护所有的 makefile/project文件也是件够呛的工作，更多的像是在干苦力而不是软件工程。对于小项目，这种方式可能还算合适，但是对于大项目来说则是不可接受的：你能想象以这种方式来构建操作系统吗？

7.2.4 动态连接

现代操作系统以及许多现代应用都使用了所谓的动态连接（dynamic linking）技术，我们将会在第 9章详细考察这种技术。使用动态连接时，你采用和静态连接类似的方式连接到库，但库的代码并不会被拷贝到最终的可执行文件中。可执行文件中所用到的动态连接库中的各函数的进入点被记录下来（译注：实际上，是记录在该可执行文件中的一个所谓的导入表（import table）中。“导入”意即从动态连接库中导入符号），当该可执行文件被操作系统加载时，它所依赖的动态库也一一被自动（隐式）加载到进程的地址空间中（译注：确切地说，这种动态连接称为装入时连接，还有一种运行时连接方式），那些原本“悬空”的进入点则被替换为实际动态库中的对应代码段地址（译注：这种替换是由加载器来完成的，被称为 fixup，意即把待决议的引用动态库的函数的进入点，替换为实际的进入点）。在 Win32系统上，动态连接库的创建通常伴随着一个小型静态库的创建，这个小型静态库称为导入库（import library），它包含了应用程序用来在动态库被加载时修补“悬空”地址的代码。可执行文件被连接到导入库的方式跟连接到常规静态库的方式相同。

从库这一边来说，可以被用于动态连接的函数称为导出函数（export functions）。一个库中的所有函数都可以是导出的，或者仅导出那些被你以某种方式明确标记了的函数，具体取决于编译器和操作系统。标记一个函数为“导出的”有各种不同的方式，具体也取决于编译器和操作系统。例如在 Solaris上使用 mapfile来选择导出函数，而 Win32上的编译器则使用 __declspec(dllexport)修饰符将函数标记为导出函数。

动态连接的优点是可以节省磁盘空间和缩减操作系统工作集大小，因为它可以消除不同的可执行文件或并发执行的进程之间的重复代码块 [Rich1997]。使用动态连接还意味着对 bug修补和对库的改进不需要对依赖该库的可执行文件进行重新构建（build）。事实上，如果某个库是操作系统的一部分，这类更新就可以在程序厂商甚至用户毫无觉察的情况下进行，一般作为安装或更新其他软件（或操作系统本身）所导致的副作用即可。

当然，使用共享库也有缺点，所谓的“DLL Hell”[Rich2002]指的就是动态库的更新版本如果包含了 bug则会破坏先前（使用老版本库的）工作良好的程序。“DLL Hell”的另一面也是个很常见的问题：对库的修补破坏了依赖于修补前的 bug的程序。具体的例子我就不举了，各位应该知道的。尽管存在这些非常实际的问题，但是通常其优点还是盖过了缺点，如今我们很难想象倒退回纯粹静态连接的旧社会了。

动态连接对兼容性的影响是显著的，从表 7.2中可以看出这一点。其兼容性的测试方式是这样的：对于某个给定的编译器，每个客户程序被连接到该编译器创建的导入库，然后将库的 DLL文件替换为由其他编译器创建的版本，替换完成后再尝试执行该客户程序。

表 7.2 Win32上的动态连接兼容性（C代码，cdecl）。#表示一个可兼容的组合

用来生成库的编	使用库的编译器

译器	Borland	CodeWarrior	Digital Mars	GCC	Intel	Visual C++
CodeWarrior	#*	#	#	#	#	#
Digital Mars	#*	#	#	#	#	#
GCC	#*	#	#	#	#	#
Intel	#*	#	#	#	#	#
Visual C++	#*	#	#	#	#	#
CodeWarrior	#*	#	#	#	#	#

从表中看出，这里几乎不存在不兼容的情况，除了 Borland使用其他编译器编译的库的时候（加了 *）。这是由于 Borland在动态连接库中对每个使用了 C调用约定的符号都使用了前导下划线，这是 Win32对静态连接的使用约定。大多数其他的 Win32编译器都不像 Borland这么做，这可能是因为 Visual C++也不这么做的缘故。（在 Borland上弥补这个问题比较棘手，因此我加了 *号。我将这个难题留给读者。提示：试试 –u- 选项，以及 IMPLIB和 IMPDEF工具。）在 UNIX系统上则不存在这种前缀命名问题。

如果你略微动动脑筋，就会意识到上面展示的完全兼容性是非常有意义的。不管你是在 Win32还是 Solaris上，你总得能够与动态系统库交互吧。如果你做不到这一点，你的编译器就不能为所在的系统生成（二进制）代码，那么制造这种编译器有什么意义呢？

7.3 C++ ABI需求

由于 C++的好些方面的原因，在 C++中实现统一的 ABI比在 C中要困难得多。 C++的类和 C中的结构虽说有很多共同点，但是却复杂得多，因为在 C++中，一个类可以有一个或多个基类。

C++通过虚函数机制提供运行期多态能力，尽管编译器大多使用一个共同的机制，即虚函数表，但具体实现的模式往往互不兼容。另外，与 C相比， C++编译器使用了复杂得多、也专有得多的符号命名模式，其复杂和专有程度几乎令 C++客户代码无法调用由其他编译器创建的二进制库。

除了 C里面的符号命名问题之外， C++编译器为了支持函数重载以及类型安全策略还使用了名字重整（name mangling）机制。

因为 C++支持静态对象，所以必须支持全局和函数作用域内的静态对象的构造和析构。我们将会在第 11章看到不同的编译器使用的不同的方案，从而导致全局对象的初始化顺序并不相同。显然这是实现 C++ ABI的另一个绊脚石。

C++还支持异常处理、运行时类型信息（RTTI），以及五花八门的 C++标准语言功能，所有这些都要求必须

具有相同或兼容的格式，并且它们的运行期行为必须保持一致。

7.3.1 对象布局

在 C++中，所有 C结构的布局问题（见 7.2.1节）仍然存在，而派生类、模板以及虚继承 [Lipp1996]的使用则为这个问题平添了更多的复杂性。派生类的布局与 C中内嵌结构的 pack问题有许多共同之处，而虚继承则是另一个“由实现定义（implementation-defined）”（C++-98: 9.2;12）问题，从而进一步加剧问题的复杂性。

模板的实现方式也带来了一定的影响。 12.3节和 12.4节讨论了模板继承会在哪些方面对对象布局产生影响。

7.3.2 虚函数

C++标准描述了虚函数机制的效果， C++运行期多态就是建立该机制之上，但是 C++标准并没有规定该机制应该以哪种方式来实现。现实的情况是，所有现代的商业编译器都使用了虚函数表机制 [Stro1994]。具体来说，对于任何定义了虚函数（或继承自某个定义了虚函数的类）的类，其实例包含有一个隐藏的指针，即 vptr，它指向一个函数指针表，即 vtable，该表中的每一项都指向一个虚函数。虽然这种“共同性”让人感到乐观，然而不同的厂商却使用了不同的约定。在第 8章我们会详细探讨这个问题。

除了 vtable的格式外，编译器还必须在“何时某个给定类的对象应该具有 vtable”或者“何时复用基类的 vtable”这类问题上达成共识。

7.3.3 调用约定和名字重整

关于 C++ ABI一个最为明显的问题就是名字重整（name mangling），它被用于支持 C++的基础机制之一：重载，因此名字重整对于 C++来说是必不可少的。考虑程序清单 7.1中的代码：

程序清单 7.1

```
class Mangle
{
public:
  void func(int i);
  void func(char const *);
```

```
};

int main()
{
  Mangle  mangle;
  mangle.func(10);
  mangle.func("Hello");
   return 0;
};
```

我故意省掉了 Mangle 的成员函数的定义，这样我们才得以“偷窥”到名字重整机制的内部。如果你使用 Visual C++ 6.0来构建（build）这个项目的话，你会发现编译能够通过，但是连接器会报两个错误：

```
error: unresolved "void Mangle::func(char const *)" (?func@Mangle@@QAEXPBD@Z)
error: unresolved "void Mangle::func(int)"          (?func@Mangle@@QAEXH@Z)
```

你应该留意到错误信息中的搞笑的名字，它们是为了支持函数重载。对于 C来说，同一个可执行文件中的某个给定的函数只能够有单个的定义，从而它的二进制名字，即符号名也只能有一个，这个符号名是函数名字经过简单的改变而来的，并且是系统相关的。但在 C++中，函数（包括自由函数、类方法以及实例方法）可以被重载。这就是说，可以定义多个同名的函数，为此必须存在某种方式用于将各个重载版本的符号名（二进制名字）区分开来。同样，不同的类中的同名方法、不同名字空间中的同名类中的同名方法的符号名（二进制名字）都需要某种方式来加以区分。结果导致所有编译器都采用了所谓的“名字重整”机制，这是一个很贴切的术语，因为为了确保符号名唯一，这种机制将原先可读的函数名字“重整”为难以辨认的一团乱麻。

由于即使是 C++，其加载符号也必须借助于操作系统的 API，而操作系统 API却是为了处理 C兼容的 API而写的，所以需要被加载的 C++符号必须被转换为“单一名字”的形式。若无这样的策略，连接器就无法知道要将哪个重载版本连接到客户代码中的调用去，程序也就无法运转。可以想象，如果存在一个专为 C++构建的集成环境，其连接器能够理解重载，那些令人眼花缭乱的符号也就没有必要了。不过仍然会需要进行某种形式的编码，只需到 JNI（Java Native Interface，Java本地接口）[Lian1999]中瞧一瞧，你就会发现仍然有许多东西，不管它们的实现有多“纯粹”，其可读性还是很差的。无论如何，一个集成的 C++操作环境需要一个 C++加载器，我们几乎已经陷入循环论证的怪圈了。

但是，既然我们决定让连接器来处理这件事情，那么我们还担心什么呢？名字重整策略就其自身来说是完全合理的，那么问题何在呢？很简单，不同的编译器会使用不同的名字重整策略，因而想要从一个编译器

编译的代码中动态加载并调用另一个编译器创建的 C++库几乎是不实际的。我们将会在 9.1节看到这方面更多的讨论。

7.3.4 静态连接

现在，如果我们考察由不同的 Win32编译器分别创建的一个简单的 C++库和一个简单的客户程序之间的静态连接的兼容性 [注 4]，我们会发现（见表 7.3） C++比 C还惨。

[注 4]现在在 Linux上， Intel编译器和 GCC达成了静态 C++连接兼容。

表 7.3 Win32上的静态连接兼容性（C++代码）。 #表示一个可兼容的组合。

用来生成库的编译器	使用库的编译器					
	Borland	CodeWarrior	Digital Mars	GCC	Intel	Visual C++
Borland	#					
CodeWarrior		#				
Digital Mars			#			
GCC				#		
Intel		#			#	#
Visual C++		#			#	#

7.3.5 动态连接

我们已经讨论过，静态连接名字重整机制的冲突问题，可以通过为客户可能使用的编译器相应提供一个单独的库的版本来避免，但是对于动态连接而言，这种方案是行不通的。因为动态库可能由多个可执行文件在运行期所共享，而这些可执行文件可能是也可能不是由相同的编译器创建的，如果动态库中的符号名所采用的名字重整约定无法被创建进程的编译器所理解，那个该进程就无法加载该动态库。

再一次，实际情况是几家 Win32编译器厂商都使用了兼容的策略，包括名字重整约定，从表 7.4中可以看出这一点。 [注 5]

[注 5]现在在 Linux上， Intel编译器和 GCC达成了动态 C++连接兼容。

表 7.4 Win32上的动态连接兼容性（C++代码）。 #表示一个可兼容的组合。

用来生成库的编译器	使用库的编译器					
	Borland	CodeWarrior	Digital Mars	GCC	Intel	Visual C++
Borland	#					
CodeWarrior		#	#		#	#
Digital Mars		#	#		#	#
GCC				#		
Intel		#	#		#	#
Visual C++		#	#		#	#

很明显，动态连接的兼容性要比静态连接的强得多。然而，同样明显的是，仍然存在相当程度的不兼容。这也是 C++在 Win32平台上不存在令人满意的 ABI的明证。

再一次，我们可能会考虑创建多个动态库，每一个使用不同的名字重整策略。我们可以假想提供动态库的“编译器特定”的版本，比如 libmystuff_gcc.so、mystuff_cw_dmc_intel_vc.dll（由于上节讨论的静态连接的兼容性问题，我们还得创建更多的导入库，不过相对而言这算是小问题）。

然而，这个方案存在一些问题。首先，它令 DLL的两大意图全部落空了。使用这种方案会导致磁盘和内存中的库的数量都会增加，而且对该库的更新必须被一致地创建并统一安装。此外，有些动态库并非纯粹是代码仓库。动态库也可以包含静态数据（在下一章我们会进一步讨论这个问题），这些静态数据可以被用于提供程序逻辑，例如，一个自定义的内存管理设施或者一个套接字池（socket-pool）。如果对于逻辑上相同的库存在多个编译器特定的版本的话，事情会变得险象环生。

因此，虽说在有限尺度上可以通过为同一个动态库提供多个编译器特定的版本的方式来“解决”动态库的 C++ ABI 问题，但是这条路却危机四伏，并且我从未听说过有哪个系统实际采用了这种方法。看起来，对于一个给定的操作系统而言，如果要达成库的可移植性，我们必须依靠 C连接方式。这么做的显著缺点是，由于 C并没有名字重整，所以我们无法导出/导入重载函数或者任何类方法。

到目前为止，我们似乎一直是在同意我的朋友所指出的“厄运的征兆”，不过幸运的是确实存在一些补救措施，我们只需稍作一点回顾。

7.4 I Can C Clearly Now

在考察了 C和 C++的动态/静态连接之后，可以被称为“编译器无关”的东西很明显就是通过 C接口来进行动态连接了。

现在我们就来考察如何实现编译器无关性，不过首先让我们对 C和 C++的兼容性进行一次快速的回顾。

7.4.1 extern "C"

C++作为一门近乎 C的超集的语言，自然存在着和 C函数交互的方式。由于 C++会对每一个函数进行名字重整，目的在于对付可能的函数重载的情形，因此，我们需要一种机制来告诉 C++编译器别去对某个用 C实现的函数进行名字重整。这可以通过 extern “ C” 来实现，如下：

```
void cppfunc(int);
extern "C" void cfunc(int);
```

现在你可以在你的 C++源代码中使用 cfun()了，并且编译器和连接器都会确保你在 C++源代码中对该名字的引用被决议成一个未经重整的符号（例如 _cfunc）。而对 C++函数 cppfunc()的引用，在与 Visual C++兼容的编译器 /连接器中则会被决议为符号 “ ?cppfunc@@YAXH@Z”。

自然，一旦某个函数被声明为具有 C连接，就无法以另一个具有 C连接的同名函数来对它进行重载了：

```
void cppfunc(int);
extern "C" void cfunc(int);
extern "C" void cfunc(char);  // 错误：不止一个 “ cfunc” 具有 C连接
```

但是，只要其同名函数没有被声明为 extern “ C”，你就可以随心所欲地对某个被声明为 extern “ C” 的函数进行重载。这令许多参加面试的人很吃惊 [注 6]。然而这种能力确实能够带来便利。设想你有一组重载的 C++函数，其中除了第一个之外其他所有函数都是为了使用便利而存在的转发函数（转发至第一个函数），见程序清单 7.2：

[注 6]以及一些考官！

程序清单 7.2

```
// ConnApi.h
struct conn_info_t *conn_handle_t;
conn_handle_t CreateConnection( char const *host
                                , char const *source
                                , int flags, unsigned *pid);
```

```
conn_handle_t CreateConnection(char const *host, int flags);
conn_handle_t CreateConnection( char const *host, int flags
                              , unsigned *pid);

// ConnApi.cpp
conn_handle_t CreateConnection( char const *host
                              , char const *source
                              , int flags, unsigned *pid)
{
  . .
}
conn_handle_t CreateConnection(char const *host, int flags)
{
  return CreateConnection(host, host, flags, NULL);
}
conn_handle_t CreateConnection(char const *host, int flags
                              , unsigned *pid)
{
  return CreateConnection(host, host, flags, pid);
}
```

为了将这个 API 置于动态库中，你可以简单地将第一个函数声明为 extern “ C”，并将其余的都声明为只在 C+编译单元中存在的重载版本，见程序清单 7.3:

程序清单 7.3

```
// ConnApi.h
struct conn_info_t *conn_handle_t;
#ifdef _ _cplusplus
extern "C" {
#endif /* _ _cplusplus */
conn_handle_t CreateConnection( char const *host
                              , char const *source
                              , int flags, unsigned *pid);
```

```
#ifdef _ _cplusplus
} /* extern "C" */

inline conn_handle_t CreateConnection( char const *host
                                     , int flags)
{
  return CreateConnection(host, host, flags, NULL);
}
inline conn_handle_t CreateConnection( char const *host
                                     , int flags
                                     , unsigned *pid)
{
  return CreateConnection(host, host, flags, pid);
}
#endif /* _ _cplusplus */

// ConnApi.cpp
conn_handle_t CreateConnection(char const *host, char const *source, int flags, unsigned *pid)
{
  . .
}
```

这么做的一个良好的副作用是，现在 C程序员也可以使用这个库了。当前 C程序员仍然有相当多的数量，能够提高你的代码的潜在使用人数总是件好事。

显然，这种方案得以通过的前提是：你的库使用了函数，并且它所涉及的所有类型都是可以用 C来表达的，或者至少在给定操作系统上，所有可能的 C++编译器中具有与 C相同的布局。

“可由 C来表达的类型”本质上意味着 POD类型（见序言），C++中定义 POD类型是为了允许 C和 C++之间的互换性。当使用 C连接的时候，人们通常倾向于将参数类型限制为 POD类型。然而 extern “ C”确实只是意味着“不加任何名字重整”，而并非意味着“仅 C（C only）”。因此，定义下面的代码是完全可行的：

```
class CppSpecific
{
  int CppSpecific::*pm;
```

```
};
extern "C" void func(CppSpecific const &);
```

尽管将共享库建立在操纵 C++类、具有 C连接的函数之上是完全可行的，但这么做很危险，因为许多编译器都具有不同的对象布局模型 [Lipp1996]，我们将在第 12章看到这一点。在实践中，坚持使用 POD类型是明智之举。

然而，关于“可移植的 C++”，这还不是故事的全部，本章的后续部分还有更多的讨论。

7.4.2 名字空间

前面当我提到我们无法以可移植的方式来导出重载函数集或类方法时，我没有提到名字空间。这并非疏忽。

在 [Stro1994]中，Bjarne Stroustrup 讨论了这样一个限制：在同一个连接单元中，一个名字最多只能对应一个 extern“C”函数，并且这种限制与名字空间上下文没有任何关系。他将这种限制称为“兼容性 hack”，事实也的确如此。然而，我们应该对这个 hack手段的存在感到高兴，因为当我们定义可移植的函数时，该手段能够为我们提供一定程度的灵活性。

基本上，如果一个 extern “C”函数被定义在某个名字空间中，那么该名字空间对该函数的符号名没有任何影响。因此，下面的这个函数的符号名仍然是 ns_func：

```
namespace X
{
  extern "C" void ns_func();
}
```

这就是为什么 C++标准库可以将 C标准库函数放在 std名字空间内，同时不会破坏 C++程序和它们之间连接的原因。

我们也可以利用这个优势：一方面，遵从作为 C++的“好市民”的原则，我们将可移植的函数定义在名字空间中，另一方面，我们仍然可以从 C代码中来使用这些函数。然而，这是一把双刃剑，因为一个具有 C连接的函数只能有一种二进制格式。换句话说，如果你在两个地方定义了同样的可移植的函数，并且将它们连接到了同一个二进制文件中，那么你会得到一个连接冲突，即便你将这两个函数置于不同的名字空间中也不行。

在实践中，我从未遇到过这类问题，但这并不代表问题不存在。所以，最好采用某些 C风格的消除二义性的手法，遵循“ API_Function”形式的命名习惯，例如 Connection_Create()。

7.4.3 extern "C++"

与 extern “ C”相对的是 extern “ C++”，这个古怪的形式很少被使用。它将一个函数或一个包含一组函数的区块声明为具有 C++连接。由于只在 C++代码中才有效，所以你可能想知道到底什么时候才需要用到它。

考虑如下的头文件：

```
// extern_c.h
#ifdef _ _cplusplus
extern "C" {
#endif /* _ _cplusplus */
  int func1();
  int func2(int p1, int p2, int p3 /* = -1 */);
#ifdef _ _cplusplus
} /* extern "C" */
  inline int func2(int p1, int p2)
  {
    return func2(p1, p2, -1);
  }
#endif /* _ _cplusplus */
```

fun2()的第三个参数在未被指定任何有意义的值时应该缺省为 -1。但由于 C并不支持缺省参数，所以合适的做法是提供一个仅限于 C++的重载版本。我们通过在 C++编译区块（第二个 #ifdef __cplusplus区块）中重载该函数来达到这个目的。

现在考虑另一个头文件，这个头文件纯粹是供客户代码使用的：

```
// no_extern_c.h
int func1();
int func2(int p1, int p2, int p3);
```

如果这个文件是为某个 C编译的源文件提供声明的话，这两个函数就具有 C连接，并且编译后的目标文件 /

库文件中的符号名是未经名字重整的。因此，在 C++中使用它们将会引发一个连接错误，因为编译后的代码告诉连接器去寻找重整后的名字（而实际上目标文件 /库文件中的符号名却是未经重整的）。为了避免出现这种情况， [Stro1994]中建议将 #include语句用 extern “ C” 包围起来，像这样：

```
// cpp_src.cpp
extern "C"
{
#include "no_extern_c.h"
}
int main()
{
  return func2(10, 20, -1);
}
```

遗憾的是，如果你对 extern_c.h这样的文件也这么干的话，就会被告知具有 C连接的函数 func2()不能有两个重载版本。因此，将仅用于 C++的重载版本简单地声明在 extern "C '区段之外的这种做法仍然不够好，你还得将它们放置在 extern "C++'区段之中，这就明确地告诉编译器它们具有 C++连接（也就是说，让编译器使用名字重整），参见程序清单 7.4：

程序清单 7.4

```
// extern_c.h
#ifdef _ _cplusplus
extern "C" {
#endif /* _ _cplusplus */
  int func1();
  int func2(int p1, int p2, int p3 /* = -1 */);
#ifdef _ _cplusplus
  extern "C++"
  {
    inline int func2(int p1, int p2)
    {
      return func2(p1, p2, -1);
    }
  } /* extern "C++" */
```

```
} /* extern "C"" */
#endif /* _ _cplusplus */
```

人们不可能将任何纯 C++头文件（有时用 .hpp或 .hxx扩展名来表示，或者像标准库头文件这样的无扩展名头文件）里的代码都用 extern "C++"包围起来，那太不体面了。但是，混合性的头文件通常都具有 .h扩展名，所以如果你将 C++代码放在了 .h头文件之中，你就需要将它们（使用 extern "C++"）保护起来。使用这种方式可以写出同时兼容于 C和 C++的健壮的头文件。

有时候，你需要在某些特殊的上下文中声明甚至定义 C++函数，这种上下文会自动由 extern "C"包围起来。一个很好的例子是在使用 COM IDL时。有时候，需要将一些辅助性的 C++函数或简单的类直接定义在 IDL头文件里，这么样做可以令你的代码与它所使用的类型和接口之间的隔离最小化。这种情况下，将代码包裹在一个条件定义（#ifdef __cplusplus）的 extern "C++"之中可以避免 MIDL编译器的扰乱，因为 MIDL编译器会将接口定义文件所转换成的 C/C++头文件，一股脑地用 extern "C"包围起来。

在结束该主题之前，我还需要做一些注明，有几款老旧的编译器对于以 extern "C"声明的函数中的模板实例化会报错，而且这个错误信息很让人糊涂：

```
extern "C" void CreateSomeObject(SomeObject *pp)
{
  *pp = new Concrete<SomeObject>(); // 错误：模板 Concrete不能被定义为 extern "C"
defined extern "C"
}
```

一个简单的解决办法是提供一个转发函数：

```
SomeObject *makeSomeObject()
{
  return new Concrete<SomeObject>();
}
extern "C" void CreateSomeObject(SomeObject *pp)
{
  *pp = makeSomeObject();
}
```

7.4.4 获得 C++类的句柄

到目前为止，我们已经获得了相当程度上的可移植性，但这需要以极大地牺牲 C++的表达力为代价。幸运的是，还没有到山穷水尽的地步。我们中很少有人会将客户代码用 C来实现。但是为客户保留这个选择的权利是件好事，因为有大量的 C程序员可能想要使用我们的库，尽管这个库是用那些“效率低下的”、“华而不实的”、“面向对象的”东西写成的。但是，就算仅冲着 RAII（见 3.5节），我们也希望采用 C++编写客户端代码。那么该做些什么使我们的可移植代码对 C++更为友好呢？

正如我们可以将类的公共接口拆解为一组 C API一样，我们同样可以在客户端以外覆类（wrapper class）的形式来对它进行重建。外覆类的优点在于可以借助 RAII机制方便地为我们进行资源管理。当然，这么做会付出效率的代价，但是通常这种牺牲是值得的。外覆类同样还可以用于处理库的初始化和释放，因而它完全可以是自包含的（独立的）。

在有些场合，你可以将这种技术用到导出类上，虽然有些“拐弯抹角”，但很实用。让我们来看 Synesis库中的 BufferStore类，这是我最喜爱的类之一，其实现采用了我随后将会提到的技术。逻辑上，它的形式如下：

```
class BufferStore
{
public:
  BufferStore(size_t cbBuffer, unsigned cBuffers);
  ~BufferStore();
public:
  unsigned Allocate(void **ppBuffers, unsigned cBuffers);
  unsigned Share( void const **ppSrcBuffers
                , void **ppDestBuffers, unsigned cBuffers);
  void Deallocate(void **ppBuffers, unsigned cBuffers);
};
```

我创建了一组可被共享的缓冲区，它们可以被分配、归还以及高效地共享。在实现网络服务时这种缓冲区类非常理想。通过将它隐藏在一个 C API之后，我令它成为了可移植的，如下：

```
// MLBfrStr.h
_ _SYNSOFT_GEN_OPAQUE(HBuffStr); // 生成一个唯一的句柄
HBuffStr BufferStore_Create(Size siBuffer, UInt32 cBuffers);
void     BufferStore_Destroy(HBuffStr hbs);
```

```
UInt32   BufferStore_Allocate( HBuffStr hbs, PPVoid buffers
                              , UInt32 cRequest);
UInt32   BufferStore_Share( HBuffStr hbs, PPVoid srcBuffers
                           , PPVoid destBuffers, UInt32 cShare);
void     BufferStore_Deallocate( HBuffStr hbs, PPVoid buffers
                                , UInt32 cReturn);
. . .
```

该 API是通过将 HBuffStr句柄转换为一个指向内部类 BufferStore_[注 7]的指针来实现的，像这样：

[注 7]可能将它命名为 BufferStoreImpl更为合适，但是我是个下划线迷。你可别学我！

```
// In MLBfrStr.cpp
UInt32 BufferStore_Allocate( HBuffStr hbs, PPVoid buffers
                           , UInt32 cAllocate)
{
  BufferStore_ *bs = BufferStore_::HandleToPointer(hbs);
  return bs->Allocate(buffers, cAllocate);
}
```

编写这种代码属于体力活，而且还需要付出一些效率的代价。但是，它令我们能够用 C++来实现这个类，同时还能维持可移植的 C接口。我们同样也可以用 C++的形式来使用它，因为其头文件中还包含了程序清单 7.5所示的代码：

程序清单 7.5

```
// MLBfrStr.h
#ifdef _ _cplusplus
extern "C++" {
#endif /* _ _cplusplus */

class BufferStore
{
  . . .
  void  Deallocate(PPVoid buffers, UInt32 cReturn)
```

```
  {
    BufferStore_Deallocate(m_hbs, buffers, cReturn);
  }
  . . .
private:
  HbuffStr m_hbs;
};

#ifdef _ _cplusplus
} /* extern "C++" */
#endif /* _ _cplusplus */
```

现在，我们可以在双方（库的实现和客户的使用）都使用 C++，而通讯则通过一个可移植的 C接口完成。实际上这正是所谓的“ Bridge模式 ”[Gamm1995]。因此，虽然实现这种东西很费时，但是你可以说服自己——它所带来的摩登效果值得你这样做。

由于从外部的持有句柄的类到内部的类之间要经过一个间接层，所以需要付出少量的时间上的代价。然而，考虑到这种技术主要是为重量级的类所准备的，所以这点微小的代价算不上什么（我承认这可能是个自我实现的预言，我得承认，在过去的十年里， ABI问题渗入了我对 C++的许多思考当中）。

7.4.5“ 由实现定义 ”的隐患

到目前为止，情况还算乐观，但是仍然存在一些事情可能会搅乱我们的盛宴。

首先，在某些操作系统上可能存在着几种不同的调用约定。对这些调用约定的完整讨论超出了本书的范围，但是有一点必须明确：如果一个函数可能具有其他二进制名字，或者可能以其他方式使用调用堆栈，那么它将会破坏我们已经开发的 ABI技术。因此，在必要时，函数调用约定也必须成为 ABI规范的一部分。因此，你可能会看到类似程序清单 7.6这样的代码：

程序清单 7.6

```
// extern_c.h
#ifdef WIN32
# define MY_CALLCONV  _ _cdecl
#else /* ? operating system */
```

```
# define MY_CALLCONV
#endif /* operating system */

#ifdef _ _cplusplus
extern "C" {
#endif /* _ _cplusplus */
  int MY_CALLCONV func1();
  int MY_CALLCONV func2(int p1, int p2, int p3 /* = -1 */);
#ifdef _ _cplusplus
} /* extern "C" */
. . .
#endif /* _ _cplusplus */
```

当涉及到你的 ABI 函数所使用的类型的大小时，类似的情况也会出现。你无法确保某个类型的大小在不同的编译器之上是相同的。如果你使用了一个类型，比如 long，而它在相同操作系统上的不同编译器之间的解释是不同的，那么你就遇到麻烦了。这也正是固定大小的类型（fixed-sized types）（第 13章）大显身手的地方。我的对策是，对 ABI函数总是使用固定大小的类型。

在实践中，这种问题很少发生在整型身上，但对于在浮点数和字符类型上有差异的编译器来说却很普遍。例如，对于 long double类型的大小，不同的编译器就有着相当不同的意见。一些编译器（例如 Borland、Digital Mars、GCC和 Intel）遵从 IEEE754标准 [Kaha1998]，将它的大小定义为 80位，而其他的编译器则将其定义为 64位（从而跟 double的大小一样）。在没有任何保证的情况下，我强烈建议你采取防患于未然的策略。

与此相关的还有另外一个你更容易遇到的问题，即结构的 pack问题。因为不同的编译器缺省的 pack行为不同，所以对于我们的 ABI所共享的任何结构，我们都必须显式地指定它们的 pack大小。和调用约定一样，这也可能引入许多琐碎的预处理工作，以及对常见但并非标准的 #pragma packing 指令的使用，见程序清单 7.7：

程序清单 7.7

```
#if defined(ACMELIB_COMPILER_IS_ABC)
# pragma packing 1
#elif defined(ACMELIB_COMPILER_IS_IJK)
# pragma pack(push, 1)
```

```
#elif . . .
  . . .
struct abi_struct
{
  int   i;
  short s;
  char  ar[5];
};
#if defined(ACMELIB_COMPILER_IS_ABC)
# pragma packing default
#elif defined(ACMELIB_COMPILER_IS_IJK)
# pragma pack(pop)
#elif . . .
```

如果所有编译器在 pack语义上都比较一致，那么存在一个不错的方法可以用于改善这种情况：将 push和 pop pragma指令定义在各自的头文件之中，这种方式可以提高可维护性和可读性：

```
#include <acmelib_pack_push_1.h>
struct abi_struct
{
  . . .
};
#include <acmelib_pack_pop_1.h>
```

在实现我们的 ABI的时候，当然还会存在许许多多实际的问题，但是我们作为不完美主义的实践者，不会为一点小小的困难所吓倒。在下一章，我们将会看到如何以可移植的方式来对 C++进行更多的支持。

第 10章 线程

多线程主题可是个大块头，它本身就值得写好几本书 [Bute1997, Rich1997]。简单起见，我将多线程编程的挑战全部归于对资源的同步访问。这里的资源可以是指单个变量，一个类，甚至是由某个线程生产另一个线程消费的东西。关键在于，如果两个或多个线程需要访问同样的资源的话，它们的访问必须是安全的。唉，C和 C++面世时多线程还不像现在这般流行。因此：

Imperfection: C和 C++对线程只字未提。

这意味着什么呢？嗯，在 C++标准中你找不到任何地方有关于线程（threading）的任何描述 [注 1]。那么，这是不是意味着你无法使用 C++来写多线程的程序呢？当然不是。但这确实意味着 C++对多线程编程没有提供任何（语言级别）的支持。这一点所导致的实际后果是相当可观的。

[注 1]在 C++98标准中，“thread”一词出现的唯一的地方（C++-98: 15.1;2）是在讨论异常处理中的“控制流程（threads of control）”时。

在编写多任务系统时，人们必须知道的两个经典概念是 [Bute1997, Rich1997]竞争条件和死锁。竞争条件是在两个不同的执行中的线程同时访问相同的资源时产生的。注意，这里我使用了“执行中的线程”这个术语，实际上包括同一个系统中的进程，以及同一个宿主系统中相同或不同进程中的线程。

为了避免竞争条件，多任务系统使用了同步机制，例如互斥体、条件变量、信号量 [Bute1997, Rich1997]等，来防止对共享资源的并发访问。当某个执行中的线程获取了一个资源时（也被称为锁定（lock）某个资源），其他需要该资源的线程都被锁在外面，并进入等待状态直到该资源被释放（即解锁（unlock））。

当然，两个或多个执行中的独立线程之间的交互通常具有潜在的高度复杂性，并且可能出现这样的情况：两个资源分别被两个线程所获取，而这两个线程都需要同时拥有这两个资源才能继续执行，但目前的状态却是各持其一，并等待对方释放另一个。这种情况就是所谓的死锁（deadlock）。另一个不那么常见、但也是致命的情况，是活锁（livelock），即一组进程中的每一个都不断地根据该集合中的其他进程的状态的改变而不断改变自身的状态，从而可能导致每个进程都停滞不前的情况。

竞争条件和死锁是难于预见或测试的，这是多线程编程的实际挑战之一。虽然死锁很容易发现：你的可执行文件停止执行了，但是它们很难诊断，因为其时你的进程（或其中的某个线程）已经挂起了。

10.1 对整型量的同步访问

由于每次当进程遇到线程切换时，处理器寄存器的内容都会被保存到线程上下文当中，因此同步的最基本形式就是确保对单个内存位置的访问是被序列化的。如果被读取的内存位置的大小是一个字节，那么处理器就能够原子地访问它。实际上，依照给定架构的规则，同样的逻辑也可以被运用于对更大数值的读取。例如，一个 32位的处理器能够确保对 32位值的访问是序列化的，前提是这个 32位的值必须对齐到 32位边界上（译注：即按 4字节对齐）。序列化对未对齐的数据的访问可能也可能不被某个处理器支持。很难想象一个可用的架构会不支持这种原子操作。

如果你希望原子地读写平台相关大小的整型值，处理器的这种保证已经够好，但除此之外你可能还想让实际上由几个操作构成的一个整体的操作也成为原子的。然而由于这类操作本身是由多个操作构成的，所以并非原子的。最经典的一个例子是递增或递减变量。下面的这条语句：

```
++i;
```

其实只是以下语句的简写形式：

```
i = i + 1;
```

对 i递增会导致如下的几个步骤：从内存中获取其值（译注：一般至寄存器中），将该值加一，最后再将新值保存到 i的内存位置中。这个过程被称为“读-改-写（RMW, Read-Modify-Write）”[Gerb2002]。由于这个过程分为三步，所以如果任何其他线程试图并发操纵 i的话，就可能导致这两个线程之一或全部被破坏。如果两个线程都试图递增 i，那么就可能产生步骤“脱节”的情况：

线程 1	**线程 2**
从内存获取 i	
	从内存获取 i
	值增 1
值增 1	
	将新值存储至 i中
将新值存储至 i中	

两个线程都从内存中获取同样的值，当线程 1保存递增后的值时，会覆盖线程 2刚保存的结果，从而导致两个递增操作之一被作废了。

在实践中，不同的处理器对这些步骤会生成不同的操作指令，例如，在 Intel处理器上的实现可能像这样：

```
mov eax,dword ptr [i]
inc eax
mov dword ptr [i], eax
```

或更简洁，如下：

```
add dword ptr [i],1
```

即使是对于只含单个指令的后一种形式，对于多处理器的系统来说它也不能保证原子性，因为从逻辑上它跟第一种是等价的，另一个处理器上的线程照样可以按照前面描述的方式介入该操作。

10.1.1 操作系统函数

由于原子递增和递减是许多重要机制（包括引用计数）的基石，因此提供某些设施以便以线程安全的方式来引导这些操作是非常重要的。正如我们将会在后面看到的，通常只需要原子整型操作能力就可以实现较复杂的组件线程安全性 [注 2]，这种做法可以节省可观的性能开销。

[注 2]我曾在第 7章提到的 Synesis的 BufferStore组件就是一个例子，它避免使用任何内核对象作为同步机制，从而获得了较高的性能。

Win32提供了 InterLockedIncrement()和 InterLockedDecrement()这两个系统函数，它们看起来像这样：

```
LONG InterlockedIncrement(LONG *p);
LONG InterlockedDecrement(LONG *p);
```

这两者分别实现了前置递增（++i）和前置递减（--i）语义。换句话说，其返回值反映了修改后的新值，而非旧值。 Linux也提供了类似的函数 [Rubi2001]：atomic_inc_and_test()和 atomic_dec_and_test()。特定的操作系统对此会提供特定的函数。

现在，使用这些函数，我们就可以以彻底的线程安全的方式来重写先前的递增语句：

```
atomic_inc_and_test(&i); // ++i
```

这种函数在 Intel处理器上的实现简单地利用了 LOCK指令前缀，像这样 [注 3]：

[注 3]真实的指令并非是这样，这里不过是为了简洁起见而已。

```
lock add dword ptr [i], 1
```

LOCK指令前缀会导致总线上出现一个 LOCK#信号，阻止其他任何线程在该 ADD指令的执行其间对 i的内存位置作任何举动。(当然，实际上比这要复杂得多，具体涉及了缓冲区行（cache line）和各种各样的鬼蜮伎俩，但从逻辑上说，它使得该指令成为原子的，不管是对于其他线程还是处理器而言 [注 4]。

[注 4]在多 CPU或超线程 /多核 CPU的机器上，线程可以真正地并行运行，而在单 CPU上，线程只是看起来同时执行而已。在单 CPU机子上，中断可能会中断指令（原子指令除外）。

使用锁语义的缺点是你需要付出速度上的代价。实际代价根据不同的架构有所不同。在 Win32上使用锁的代价大体上是不使用的 200-500%(在本章的后续部分我们会看到这一点)。因此，简单地让每个操作都成为线程安全的做法是不妥当的。事实上，多线程的全部要义在于允许独立无关的处理业务并发地执行。

在实践中，人们通常可以使用构建（build）设置来决定是否使用原子操作。在 UNIX上， _REENTRANT预处理符号通常可以被用来告诉 C和 C++代码该连接单元是为多线程环境而构建（build）的。在 Win32上则是 _MT或 __MT__，或类似的预处理符号，具体取决于编译器。自然，这种东西可以被抽象为一个平台 /编译器无关的符号，例如 ACMELIB_MULTI_THREADED，然后该符号被用于在编译期选择合适的操作。

```
#ifdef ACMELIB_MULTI_THREADED
  atomic_increment(&i);
#else /* ? ACMELIB_MULTI_THREADED */
  ++i;
#endif /* ACMELIB_MULTI_THREADED */
```

由于将这些预处理指令散布在代码之中实在是件不雅的事，所以通常我们采用的方式是将操作抽象到一个公共函数里面，让该函数将这些预处理指令封装起来。在一些被广泛使用的库中可以找到这方面的许多例子，例如 Boost和微软 ATL（Active Template Library，活动模板库）。

我应该指出，并非所有的操作系统都提供对整型量的原子操作，在这种情况下，你可能需要求助于操作系统的同步对象（例如互斥体（mutex））来锁定对原子整型量 API的访问，我们会在本章的后续部分看到这一点。

10.1.2 原子类型

前面我们看到，对于单线程上下文来说，我们可以对整型量简单地使用 - 和 ++。但是对于多线程，我们需要使用操作系统 /架构原语。这种方式的缺点是即使我们将差异性抽象到一个公共的函数，例如 integer_increment 中，我们也要时时记住对某个整型量的所有原子操作都必须以该公共函数来完成。然而忘记其中之一是很容易的事情，一旦出现了这种情况，你就可能在应用程序中遭遇一个竞争条件，并且这种东西非常难于诊断。

C++通过允许用户自定义类型和内建类型具有相同的“外观”，而支持语法的一致性。那么，为什么不让原子类型从任何方面看起来都跟内建类型一样呢（除了它们的操作是原子的之外）？没有理由不这样做，并且，实际上实现起来也相当简单 [注 5]：

[注 5] volatile修饰符用于此类变量的声明中，这种用法通常在多线程代码中较为常见，因为 volatile 可以防止编译器将该变量置于其内部寄存器中。编译器的这种优化行为会使得对该值的基于实际内存地址的同步失效。在 18.5节我们会更详细地考察它。

程序清单 10.1

```
class atomic_integer
{
public:
  atomic_integer(int value)
    : m_value(value)
  {}
// 操作
public:
  atomic_integer volatile &operator ++() volatile
  {
    atomic_increment(&m_value);
    return *this;
  }
  const atomic_integer volatile operator ++(int) volatile
  {
    return atomic_integer(atomic_postincrement(&m_value));
  }
  atomic_integer volatile &operator — () volatile;
```

```
  const atomic_integer volatile operator -- (int) volatile;

  atomic_integer volatile &operator +=(value_type const &value) volatile
  {
    atomic_postadd(&m_value, value);
    return *this;
  }
  atomic_integer volatile &operator -=(value_type const &value) volatile;

private:
  volatile int m_value;
};
```

这是 C++可以使线程编程更容易更简单的领域。然而，关于整型量的自然语义到底有多少应该被该类提供的问题仍然有待推敲。上面的代码展示了在给出原子整型操作的库函数的前提下实现递增/递减以及加减运算是多么得简单。然而，乘法、除法、逻辑操作、移位运算以及另外一些操作则要复杂得多，并且，大多数的原子整型库并不提供这些操作。如果你的代码确实需要它们的话，这就不关我的事了。这种时候，作者往往可以扬长而去，并堂而皇之地说，“这些东西作为一个练习留给读者”。

Boost的原子操作组件使用的正是这种途径：提供 atomic_count类型的平台相关的版本，atomic_count仅提供++和 --操作（atomic_increment/atomic_decrement）以及隐式转换（atomic_read），因此，如果你决定避开过于复杂的东西，大可心安理得，因为你并不是唯一这么做的人。

10.2 对（代码）块的同步访问：临界区

对于大多数同步需求来说，单个原子操作是不够的，它们往往需要对所谓的临界区的独占访问 [Bulk1999]。例如，如果你有两个变量需要原子地进行更新，你就必须使用一个同步对象来确保每个线程对该临界区的访问都是独占地进行的，像这样：

```
// 共享对象
SYNC_TYPE    sync_obj;
SomeClass    shared_instance;
. . .
// 被多线程调用的代码段
lock(sync_obj);
```

```
int i = shared_instance.Method1(. . .);
shared_instance.Method2(i + 1, . . .);
unlock(sync_obj);
```

Method1()和 Method2()这两个操作必须以一种不被中断的顺序进行。因此，调用它们的代码被包裹在获取和释放同步对象之间。通常，任何对此类同步对象的使用的代价都是高昂的，因而需要某些途径来使这些开销最小化甚至完全避免。

将同步对象用于保护临界区的做法之所以代价高昂，原因有二：第一，同步对象的使用本身可能代价高昂。例如，考虑表 10.1中的时间（以毫秒为单位），它们是基于对四个 Win32同步对象的一千万次“ 获取 释放 ”周期，以及由两个空函数调用所代表的控制场景。结果一目了然地显示出，使用同步对象的代价是相当可观的，可能高达常规函数调用开销的 150倍！

表 10.1

同步对象	单处理器	对称多处理器
无（控制）	117	172
CRITICAL_SECTION	1740	831
atomic_integer	1722	914
互斥体	17891	22187
信号量	18235	22271
事件	17847	22130

将同步对象用于保护临界区的第二个开销是由那些被拒绝在临界区之外的线程所导致的。临界区越长，就越是可能带来这种代价，因而最好尽量保持临界区的短小，或者将它们分解为多个“ 子临界区 ”，就像在 6.2节讨论的那样。然而，由于获取和 /或释放同步对象的函数调用的代价可能会非常高，所以在临界区分解和导致“ 被拒 ”线程的长时间等待之间必须求得一个微妙的平衡。只有根据实际情况对性能进行评测你才能得到一个确定性的答案。

10.2.1 进程间互斥体和进程内互斥体

互斥体是用于守护临界区的最常见形式的同步对象，并且，取决于操作系统，互斥体可以被分为两种：进程间的和进程内的。进程间互斥体是指可能被不止一个进程中的代码引用的互斥体，从而可以提供进程间的同步。在 Win32上，这类互斥体通常是通过调用 CreateMutex()（同时对其进行命名）来创建的。于是其他进程就可以通过互斥体的名字来对它进行访问（利用 CreateMutex()或 OpenMutex()） [注 6]。通过

PTHREADS[Bute1997]这个 UNIX POSIX标准线程库，一个互斥体可以通过一次进程分叉（译注： fork()，即在 UNIX系统上产生子进程的系统调用）传递给子进程，或者还可以通过内存映射来共享。

[注 6]匿名的互斥体句柄也可以通过其他 IPC机制被传递到子进程，但命名机制是最直接了当的机制。

相较之下，进程内互斥体只对同一进程中的线程是可见的。由于它们不必跨进程边界存在，因而它们的状态可以被维持在本进程空间中，因此可以部分乃至完全避免因陷入内核而导致的高昂开销。Win32 上有一个被称为 CRITICAL_SECTION的构造，这是一种轻量级的机制，用于将大部分的操作保持在内核之外，并且仅当（互斥体的）所有权需要转移给另一个线程时才进行必要的内核调用。通过使用进程内互斥体可以节省可观的性能损耗，表 10.1也说明了这一点，其结果是基于一个单线程的可执行体。后面我们会看到当应用程序中存在多线程时 CRITICAL_SECTION如何工作。

10.2.2 自旋互斥体

还有一种特殊的基于轮询的进程内互斥体，这通常是一种糟糕的实践。简单地说，轮询就是通过反复不断地测试来等待某个条件的改变，像这样：

```
int g_flag;

// 等待线程
while(0 == g_flag)
{}
. . . // 现在做我们想做的事吧！
```

这种东西会吞噬处理器时钟周期，因为跟那个改变标志变量从而允许轮询线程继续执行的线程相比，轮询线程通常[注 7]被给予同样的优先级。轮询是个糟糕的主意，通常标志着使用它的人是个刚接触多线程的菜鸟，或者等着拿解雇费的萎瓜。

[注 7]取决于线程相对优先级，以及其他线程可能正在等待的任何外部事件。

然而，某些场合则非常合适于采用自旋（spinning）。让我们看看下面自旋互斥体的实现，这是 UNIXSTL[注 8]中的 spin_mutex类（程序清单 10.2）：

[注 8]这是 STLSoft的一个子项目，它将 UNIX API映射到 STL。

程序清单 10.2

```
class spin_mutex
{
public:
  explicit spin_mutex(sint32_t *p = NULL)
    : m_spinCount((NULL != p) ? p : &m_internalCount)
    , m_internalCount(0)
  {}
  void lock()
  {
    for(; 0 != atomic_write(m_spinCount, 1); sched_yield())
    {}
  }
  void unlock()
  {
    atomic_set(m_spinCount, 0);
  }
// 成员
private:
  sint32_t  *m_spinCount;
  sint32_t  m_internalCount;
// 声明但不予实现
private:
    spin_mutex(class_type const &rhs);
    spin_mutex &operator =(class_type const &rhs);
};
```

自旋互斥体的工作机制相当简单。当 lock()被调用时，执行一次原子写操作，将自旋变量 *m_spinCount（整型）置为 1。如果它原先的值为 0，就意味着调用者是设置它的第一个线程，从而“获取”该互斥体，然后该方法（lock()）返回。但如果原先的值为 1，那就意味着调用者被另一个线程拒之门外（译注：另一个线程已经占用了该互斥体），无法获得该互斥体，于是接着调用 PTHREAD函数 sched_yield()，将执行机会让给其他线程。以后当它被再次唤醒时，它就会再次尝试获取该互斥体。这个过程不断重复，直到获取成功，从而该互斥体的所有权就得到了锁定。

当获取了互斥体的线程调用 unlock()时，自旋变量 *m_spinCount会被重置为 0，从而允许其他进程再次获取该互斥体。由于该类可以基于一个外部的自旋变量来创建，因而引入了成员 m_internalCount，使得构造函数看上去有一点复杂，不过这在某些特定的场合可能会非常有用（我们将在第 11章和第 31章中看到这一点）。

然而，当出现激烈的竞争情况时，自旋互斥体就不再是好的解决方案了。不过，在出现竞争的可能性很低并且获取 /释放同步机制的开销也不高的情况下，使用自旋互斥体是行之有效的方案。由于它们可能导致高昂的开销，所以我倾向于仅将它们用在初始化动作中，这种情况下竞争极其罕见，但理论上仍是可能的，你心里要有数。

10.3 原子整型的性能

在步入多线程扩展（10.4节）和线程相关的存储（10.5节）这两个主题之前，我们先来考察一下各种原子整型操作策略的性能的各个方面。

10.3.1 基于互斥体的原子整型

如果你的操作系统未提供原子整型操作，你可能需要求助于互斥体来对“对原子整型 API的访问”进行加锁，如程序清单 10.3所示：

程序清单 10.3

```
namespace
{
  Mutex s_mx;
}
int atomic_postincrement(int volatile *p)
{
  lock_scope<Mutex>  lock(s_mx);
  return *p++;
}
int atomic_predecrement(int volatile *p)
{
  lock_scope<Mutex>  lock(s_mx);
  return --*p;
```

```
}
```

这里的问题在于性能。你不但要付出因获取和释放互斥体对象而陷入内核态所导致的也许相当可观的开销，你还要面对若干进程试图同时执行各自原子操作而导致的竞争条件。你的进程中的每个单独的原子操作都引入一个单独的互斥体对象，这自然会造成性能上的瓶颈。

我曾亲眼目睹有人试图通过为每个原子函数提供一个单独的互斥体来减小开销。然而遗憾的是，这么做只是成功地缩短了线程的等待时间。另外，我估计你也猜到了，这在单处理器的 Intel 机器上通过了彻底的测试。但是，一旦这个应用程序到了多处理器的机子上，这种做法可就洋相百出了 [注 9]。由于每一个互斥体保护的是函数而非数据，所以当有些线程递增某个变量的同时另一个线程在递减该变量是可能的。这种做法所能确保的只是防止两个线程同一刻对同一整型变量做同样的事。正如多线程领域的其他东西一样，你只有在多处理器的机器上测试代码后才能确信它们是正确的。

[注 9]这在现代的超线程单处理器机器上也会遭遇同样的命运。

先不管这个可怜的失败，确实有办法在多个互斥体之间共享相同的竞争条件从而减轻的这个性能瓶颈。我们所需要的是基于被操纵的变量的某些属性来选择互斥体，这个属性就是变量的地址。(我们不能基于变量的值来选择互斥体，因为它随时可能改变)。

我们的做法看起来像这样：

```
namespace
{
  Mutex        s_mxs[NUM_MUTEXES];
};

int __stdcall Atomic_PreIncrement_By(int volatile *v)
{
  size_t             index = index_from_ptr(v, NUM_MUTEXES);
  lock_scope<Mutex> lock(s_mxs[index]);
  return ++*(v);
}
```

函数 index_from_ptr()提供从变量地址到一个位于 [0, NUM_MUTEXES-1]区间上的整型的确定性映射。简单地对地址进行取模运算（%）在这里并不适合作为映射手段，因为大多数系统都将数据对齐到 4 & 16或

32字节边界上。下面这种做法可能比较适合：

```
inline size_t index_from_ptr(void volatile *v, size_t range)
{
  return (((unsigned)v) >> 7) % range;
}
```

通过在我的 Win32机器上的测试，我发现使用 7比其他值具有更好的效果，但这种做法不大可能被原封不动地套用到其他平台上，因而你必须为你的平台进行特定的优化。

10.3.2 运行期按架构派发

下面我想为你展示在 Intel平台上对原子整型操作进行性能优化的一点小技巧。正如我们前面已经了解到的， Intel处理器能够不被中断地执行单个指令形式的 RMW（读-改-写）操作（例如 ADD, XADD），因而在 Intel 单处理器上我们不必对总线进行加锁就能实现该操作的原子性。相反，在多处理器机器上要想达到这个目的的话，你就必须对总线进行加锁。由于人们构建（build）并分发的通常是单一版本的代码（译注：对单处理器和多处理器机器而言），所以最好让代码只在必要时才付出因总线锁定而导致的性能开销。既然对总线加锁和不加锁两种情况下的指令数目都很少，那么我们就必须以一种高效的方式来完成这种优化，否则用于测试是否多处理器的代码的开销反而大过它带来的节省。解决方案的简化版本 [注 10]展示于程序清单 10.4中，它跟大多数现代的 Win32编译器都是兼容的（译注：由于这种技术是按照单/多处理器架构的不同来选择是否执行总线加锁的，因而被称为按架构分发技术，注意下文的这种称谓）：

[注 10]这些函数的完整实现见配书光盘。

程序清单 10.4

```
namespace
{
  static bool s_uniprocessor = is_host_up();
}

inline __declspec(naked) void __stdcall
  atomic_increment(sint32_t volatile * /* pl */)
{
  if(s_uniprocessor)
```

```
  {
    _asm
    {
      mov ecx, dword ptr [esp + 4]
      add dword ptr [ecx], 1
      ret 4
    }
  }
  else
  {
    _asm
    {
      mov ecx, dword ptr [esp + 4]
      lock add dword ptr [ecx], 1
      ret 4
    }
  }
}
```

即使你对 Intel汇编语言不熟悉，你也应该能够看出这种机制是何等得简单。 s_uniprocessor变量对于单处理器机器来说是 true，对于多处理器来说则是 false。如果它是 true的话，就可以不加锁地实现原子的递增操作，反之则需要对总线进行加锁。这里在 s_uniprocessor的实例化中可能存在的任何竞争条件都是无关紧要的，因为缺省情况下是进行加锁。

在下面的性能测试表中，这正是 Synesis库里的 Atomic_API和 WinSTL原子函数所采用的机制。

10.3.3 性能比较

前面我们对实现原子整型操作的种种机制已经讨论了很多，现在让我们来看一些事实。不过在这之前，我首先强调一下，本节的表格只反映单 /多处理器 Intel架构上的 Win32操作系统的性能，其他架构和操作系统可能具有不同的特征。

我曾考察过七种不同的策略，每一个都使用了一个公共的全局变量，该变量被线程递增或递减。第一种策略“未加守卫的（Unguarded）”不加锁，只是简单地通过 ++/--操作符来递增 /递减。接下来的两种策略（Synesis的 Atomic_*库函数和 WinSTL的内联函数）则使用“架构 -派发”技术。第四种策略调用了 Win32

的 Interlocked_*系统函数。最后三种策略则分别使用了 Win32的 CRITICAL_SECTION、WinSTL的 spin_mutex 和 Win32的 MUTEX内核对象作为同步对象来守护临界区，其中临界区由对变量的 ++或 --操作构成。测试的结果显示在表 10.2中，所显示的是 31个竞争线程一千万次操作的总时间。因为这些策略在不同的机器上进行了测试，因此我们获得的是一个相对性能比较表。

表 10.2

同步策略	单处理器		对称多处理器	
	绝对时间（ms）	与未加守卫的情形的百分比	绝对时间（ms）	与未加守卫的情形的百分比
未加守卫的 ++/--	362	100%	525（不正确）	100%
Synesis Atomic_* API	509	141%	2464	469%
WinSTL atomic_*内联函数	510	141%	2464	469%
Win32 Interlocked* API	2324	642%	2491	474%
Win32 CRITICAL_SECTION	5568	1538%	188235	35854%
WinSTL spin_mutex	5837	1612%	3871	737%
Win32 MUTEX	57977	16016%	192870	36737%

我故意给测试程序所生成的线程数目定了一个奇数，这样当所有线程都执行完毕，被操纵的变量就会具有一个跟迭代次数相同的较大的非零值。由此我们就可以迅速确定操作是否确实被原子地执行了。该表显示的所有情况，包括在单处理器上的“未加守卫的（Unguarded）”操纵，行为都是正确的。但是，在“未加守卫的/对称多处理器（Unguarded/SMP）”的情形，每次运行后变量的值都有相当大的差异，这就表示线程之间互相中断了其他线程的 RMW指令周期，也恰恰验证了我们对于多处理器机器上的某些单指令操作的非原子性的认识。

从性能方面来说，有几点可以很明显地看出来。首先，不管采用哪种机制，它们在多处理器上的开销相对于单处理器上来说总是较高的，这就意味着多处理器缓存（cache）不喜欢被中断。

其次，通过使用按架构派发技术，Synesis的 Atomic_API和 WinSTL的 atomic_*在单处理器系统上的表现都非常好，其开销只有 Win32 Interlocked_*系统库函数的 22%，或“未加守卫的（Unguarded）”++/--操作的 141%。而且，在多处理器上，除 LOCK之外的处理器测试所引入的额外开销非常小，仅为 1%。我想说的是，如果你写的程序既需要在单处理器架构上工作，又需要在多处理器架构上工作，并且你只想交付单一版本的程序，使用这种按架构派发的技术很可能会为你带来可观的收益。

这张表的结果印证了一个众所周知的事实：作为一种内核对象，将互斥体用于实现原子操作会导致很高的

开销，在 Win32系统上这么使用显得头脑不够冷静。

跟互斥体稍有区别的是 CRITICAL_SECTION。我不知道你怎么认为，但是就我在 Win32平台上学习多线程编程时所获知的大部分劝诫，都是提倡将 CRITICAL_SECTION作为替代互斥体的一个更好的方案。的确，在单处理器系统上，它比互斥体快上 10倍，然而在多处理器上，它跟互斥体的性能就不相上下了。再一次，你需要在多处理器系统上测试你的代码，从而验证你对所采用的机制的效率的假定。我得指出，CRITICAL_SECTION 并非实现原子操作的有效机制，和互斥体的情况不同，我曾在客户代码中看到许多对 CRITICAL_SECTION的使用。

你可能想知道为什么会有人使用自旋互斥体来实现原子操作。好吧，我来告诉你。Linux的 <asm/atomic.h> 中提供的原子操作只能保证 24位以内的操作。此外，在 Linux的一些变种上，具有正确语义（递增并返回原始值）的函数可能并不存在。通过使用概念上比较简单的写 / 置函数（write/set functions），支持读写更宽数据的原子操作仍然是可以实现的，同时并不会引入过高的性能开销。

我希望上表中的结果能够让你停下来思考一下你自己的实现。记住，这是在特定于 Win32平台的测试结果，在其他平台上性能可能有显著的不同。但是其主要作用是学习性能评测，并质疑 / 验证你的假设。

10.3.4 原子整型操作：尾声

在这一章中我花了很大的篇幅来讨论原子操作，对此我并不感到有何不妥。理由有三点：首先，我认为它们可以被直接合并到 C++之中，而其他多线程和同步构造则不能如此，因为它们之间具有极大的差异性，且缺乏广泛的可用性 [注 11]。

[注 11] 话虽如此，仍然有一些库致力于线程编程的统一化，例如优秀的 Pthread-Win32（见附录 A）库。

其次，它们是极其有用的构造，其威力与简单性成反比。仅仅使用原子整型操作，我们就可以在大量的 C++类中实现大多数乃至所有被要求的线程安全性，在后面的章节中我们将会看到这一点。

最后，退一步说，在这方面的著作中对原子操作的讨论还不够。我希望通过这里的讨论能够让你时常想起它们。

原子操作在许多平台上都是可用的，或者作为系统库函数，或者作为非标准库，甚至你还可以编写自己的汇编版本。（我曾见过一本书，其内容很大程度上展示高级 C++技术，但具有讽刺意味的是，该书的目的却是促进对汇编语言的运用。世界真奇妙！）

即使是在我们选择使用互斥体（通常是 PTHREADS）来实现原子操作时，同样有一些措施可以用于提高效率。需要注意的一点是，不要在这种情况下使用自旋互斥体。可能会出现这么一种情况：你使用了某个互斥体类的若干实例，该互斥体类是基于一个原子整型 API 实现的，而后者又是基于一个或几个全局互斥体来实现的。在这种情况下，你应该使用预处理来确保你选择的互斥体是（基于 PTHREADS的）简单互斥体，否则会给性能带来消极影响，因为这可不是你想要或期望的。

这是关于在多线程开发中发现更广泛的真理的一个绝好的例子。在实践中，我们确实需要考虑同步需求以及应用程序所运行的宿主系统设施的细节。要是 C++支持原子操作就好了，但是就我个人看来，提供一个标准化的高阶同步原语 [注 12]，同时在不同架构上仍维持最大的效率是非常困难的。通常，你只要了解并灵活运用手头的多线程工具就可以很好地完成任务（[Bute1997，Rich1997]）。

[注 12]目前已经有一些努力致力于将它纳入标准的下一个版本中，我希望到那时前述的东西都变成无用之物，尽管我对此表示怀疑。

10.4 多线程扩展

既然我们已经讨论了与多线程有关的一系列议题，你现在大概已经开始认同语言对多线程操作提供内建支持是很有用的。事实上，有几门语言确实提供了多线程构造。C++的传统则是偏好提供新的库设施而非新语言元素。我们接下来考察一些领域，看看这些领域如何提供语言级别的多线程支持，以及我们如何使用库（外加一点预处理技巧）进行实现。

10.4.1 synchronized

D和 Java中都有 synchronized关键字，它可以被用于守护临界区，像这样：

```
Object obj = new Object();
. . .
synchronized(obj)
{
  . . . // 临界区代码
}
```

将 synchronized关键字合并到语言中的途径之一，是将如上的代码自动转换为下面的这种形式：

```
{ __lock_scope__<Object> __lock__(obj);
```

```
{
  . . . // 临界区代码
}
}
```

__lock_scope__从任何意义上说都跟 6.2节的 lock_scope模板类似。这实现起来相当简单，并且，使用一个关联的 std::lock_traits模板，可以令任何“可 trait”类型的实例通过这种方式进行同步，而并不一定要将它们转换成同步对象锁。

然而，这个理由还不足以说明语言扩展的必要性，因为通过一点点宏我们就可以实现同样的效果。基本上，我们所需要的仅仅是下面这两个宏：

```
#define SYNCHRONIZED_BEGIN(T, v) \
  {                              \
  lock_scope<T> __lock__(v);
#define SYNCHRONIZED_END() \
  }
```

这种做法只有一个小小的遗憾，那就是对象的类型不能被自动推导出来，我们必须手工给出，而且代码看起来有点不够优美 [注 13]：

[注 13]我可以辩称这儿有点丑陋的形式其实是个优点，因为它使得临界区同步状态的“轮廓”更加明显，这对任何阅读这段代码的人来说都是值得注意的一点。

```
SYNCHRONIZED_BEGIN(Object, obj)
{
  . . . // 临界区代码
}
SYNCHRONIZED_END()
```

如果你不喜欢 SYNCHRONIZED_END()部分，你可以耍一点小技巧，这样来定义 SYNCHRONIZED()宏：

```
#define SYNCHRONIZED(T, v)                          \
  for(synchronized_lock<lock_scope<T> > __lock__(v);   \
      __lock__; __lock__.end_loop())
```

类模板 synchronized_lock<>在这里只是用于定义一个状态 [注 14]，并结束这个 for循环，因为我们不能在这个 for循环内声明第二个条件变量（见 17.3节）。这是一个螺栓（bolt-in）类（见第 22章），大致如下：

[注 14]它并没有真的定义一个 operator bool()。在第 24章我们将会看到为什么不这么做，以及如何正确地完成这项任务。

程序清单 10.5

```
template <typename T>
struct synchronized_lock
  : public T
{
public:
  template <typename U>
  synchronized_lock(U &u)
    : T(u)
    , m_bEnded(false)
  {}
  operator bool () const
  {
    return !m_bEnded;
  }
  void end_loop()
  {
    m_bEnded = true;
  }
private:
  bool  m_bEnded;
};
```

这里还有另一个问题（当然了 !）。正如 17.3节描述的，编译器对 for循环的声明具有不同的反应。如果同一个作用域中出现了两个同步区段，那么有些老式的编译器就会发出抱怨。

```
SYNCHRONIZED(Object, obj)
{
  . . . // 临界区代码
}
  . . . // 非临界区代码
SYNCHRONIZED(Object, obj) // 错误："重新定义 __lock__"
{
  . . . // 其他临界区代码
}
```

因此，一个可移植的解决方案必须确保每个“ __lock__”都是不一样的，为此我们不得不面对“肮脏的”预处理代码 [注 15]：

[注 15]这里使用了两层宏来实现文字连接，希望你研究一下为什么这样。

```
#define concat__(x, y)            x ## y
#define concat_(x, y)             concat__(x, y)
#define SYNCHRONIZED(T, v)                    \
  for(synchronized_lock<lock_scope<T> >       \
      concat_(__lock__, __LINE__) (v);        \
      concat_(__lock__, __LINE__);            \
      concat_(__lock__, __LINE__)  .end_loop())
```

很丑陋，不是吗？但这种方案在我测试过的所有编译器上都能工作。如果你不需要考虑有关 for循环的历史遗留错误，你可以使用原先那个更简洁的版本。配书光盘中包含了这些宏和类的完整版本。

10.4.2 匿名 synchronized

由对象控制的临界区还有一种麻烦情况，那就是有时候没有对象可以被用作锁。在这种情况下，你可以在局部作用域中声明一个静态对象，或者，更好的做法是，在临界区所在的文件中的匿名名字空间内声明一个全局对象。你也可以基于 SYNCHRONIZED()宏使用的技术写一个 SYNCHRONIZEDANON()宏，它包含一个局部静态对象，但是这样的话你就可能会遇到一个可能的竞争条件，即两个或多个线程可能会同时试图执行该静态对象的“仅一次”的构造过程。这种情况并非没办法消除，我们将会在下一章讨论静态对象时看到解决这个问题的技术，但最好的做法还是避开这个问题。因此，使用名字空间范围内的对象在这里是最佳选择。

10.4.3 atomic

现在回到我最喜欢的同步议题，即原子整型操作上来。一个可能的语言扩展是引入 atomic关键字，以便支持如下形式的代码：

```
atomic j = ++i; // 等价于 j = atomic_preincrement(&i)
```

或者，还可以使用 XOR互换技巧 [注 16]

[注 16]这是老旧的 hack伎俩 [Dewh2003]，也是面试常问的问题之一。测试一下吧，你会发现它的确能起作用，尽管我认为这种做法并不能保证是可移植的！

```
atomic j ^= i ^= j ^= i;  // 等价于 j = atomic_write(&i, j);
```

编译器负责确保这些代码被转换成目标平台上的恰当的原子操作 [注 17]。遗憾的是，处理器指令集之间的差异意味着我们要么忍受不可移植的代码，要么就得面对只有少数操作适合被选作原子操作的现实。我们当然不愿意编译器除了在可能的时候使用轻量级的方案外，在实现其他操作的时候就一声不吭地使用共享的互斥体（来进行锁定和解锁）。对于后者，最好能在代码中显式地表达出来，就像我们现在所做的那样。

[注 17]注意，我这里的意思是 atomic 关键字是被运用到操作上的，而并非变量上。将一个变量定义为 atomic，然后让接下来 50行代码都依赖于这个 atomic行为的做法显然对可维护性是个伤害。相比之下，使用 atomic_*函数更可取，这样意图鲜明，且让代码维护者易于查找。

如果 C/C++中有一个 atomic关键字将是一件好事，它对应于所有架构都具有的（有限的）一组原子整型操作。然而，使用 atomic_*函数也不算难，并且与关键字形式相比或许还要更可读一些。其仅有的一个实际缺陷是这一组原子整型操作并非对所有平台都是强制性存在的。很可能 C/C++标准的下一代版本会引入它们。

10.5 线程相关的存储

到目前为止，本章所有讨论都集中于多线程对公共资源的同步访问上。然而，关于线程还有另一个方面，那就是线程相关的资源的提供，或者，用众所周至说法就是 TSS（Thread-Specific Storage，线程相关的存储） [Schm1997]。

10.5.1 重入

在单线程的程序中，在函数内部使用局部静态对象是提高函数易用性的一个合理方式。C 标准库中就有若干函数采用了这种技术，例如 strtok()，它基于一组字符分隔符将一个字符串标记化（tokenize）（译注：即将 str分解为由字符串 delimiterSet中的字符所分割的记号）。

```
char *strtok(char *str, const char *delimiterSet);
```

该函数在内部维护一个静态变量，它维护当前标记化点，从而后续的对该函数的调用（将 NULL传给 str）将会返回前字符串的一系列的标记（tokens）。

遗憾的是，在多线程情况下，这种函数意味着一个经典的竞争条件。当某个线程正在处理的途中，另一个线程可能会开始一个新的标记化过程。

跟其他竞争条件不同，这种情况下你不能使用同步对象来将访问串行化。那只会导致一个线程在使用该函数时其他线程都无法对内部标记化结构进行修改，而一个线程破坏另一个线程的标记化结果的情况仍然存在。

[译注]这里之所以加锁的办法不能解决问题，是因为问题的本质其实不在访问 tokenizer函数的线程同步上，而是在于其使用静态局部对象的本质决定了所有使用这个 tokenizer函数的线程都将使用/依赖其内部的同一个标记化结构，这样，即便每个线程在访问这个内部结构的时候都是互斥的，但它们对该结构产生的副作用仍然还是会影响到其他线程。

这里我们需要的并非对“线程全局”（译注：thread-global，即对于多线程来说从效果上等于全局变量：只有一份，且每个线程都能访问）变量的访问进行串行化，而是提供“线程局部”（译注：thread-local，即每线程一份，效果上相当于局部变量）变量[注 18]。这正是 TSS（线程相关存储）的由来。

[注 18]现代 C/C++运行时库使用了 TSS来实现 strtok()以及其他此类函数。

10.5.2 线程相关的数据/线程局部存储

PTHREADS和 Win32线程基础设施都提供了某种程度上的 TSS。PTHREADS[Bute1997]中的版本称为线程相关数据（TSD，Thread-Specific Data），Win32上的版本则被称为线程局部存储（TLS，Thread-Local Storage），它们的意思是一样的。

它们都提供某种方式来创建这样一个变量，该变量在进程中的每个使用它的线程中都具有不同的值。在PTHREADS里，该值被称为键（key），而在Win32上则被称为索引（index）。Win32把用于存放各线程中的键值的内存位置称为槽位（slot）。我喜欢键、槽位和值这三种说法。

PTHREADS的TSD基于如下的四个库函数：

```
int pthread_key_create(    pthread_key_t  *key
                       ,   void (*destructor)(void *));
int pthread_key_delete(    pthread_key_t  key);
void *pthread_getspecific( pthread_key_t  key);
int pthread_setspecific(   pthread_key_t  key
                       ,   const void     *value);
```

pthread_key_create()创建了pthread_key_t类型的一个键（不透明类型）。调用者还可以传递一个清理函数，待会儿我们会讨论这个函数。通过调用pthread_setspecific()和pthread_getspecific()可以线程相关地设置或获取值。pthread_key_delete()则被用来销毁不再被需要的键。

Win32的TLS API具有类似的“四大金刚”：

```
DWORD  TlsAlloc(void);
LPVOID TlsGetValue(DWORD dwTlsIndex);
BOOL   TlsSetValue(DWORD dwTlsIndex, LPVOID lpTlsValue);
BOOL   TlsFree(DWORD dwTlsIndex);
```

这些TSS API的常规用法是这样的：在主线程中，在任何其他线程活动之前，创建一个键，并将该键保存在一个公共地点（全局变量，或者通过某个函数返回），然后所有线程就可以通过从它们各自的槽位中存入或获取值，来操纵它们各自的TSS数据的拷贝。

遗憾的是，在这些模型中存在着若干不足，尤其是Win32的版本。

首先，这些API所能提供的键的数目有限。PTHREADS保证至少有128个，Win32则是64个[注19]。尽管实际上人们很少会超过这个限度，但考虑到软件的多组件特征日益明显，要求更多键的可能性并非一点儿都没有。

[注19]Windows 95和NT4提供了64个键。后来的操作系统则提供了更多（Windows 98/Me是80个，Windows

2000/XP则是 1088个），然而，对于必须能够在任何 Win32系统上执行的代码， 64是上限。

第二个问题是 Win32 API并没有提供任何在线程退出时清理槽位的能力。这意味着人们某种程度上必须能够拦截线程的退出，并借机清理与该线程的槽位中的值相关的资源。当然，对于 C++用户而言，语言本身提供了自动析构机制，可以使我们免受这方面的痛苦，而且在某些场合非这种方式不能如愿。

尽管 PTHREADS 提供了在线程终止时清理资源的手段，但是它仍然没有提供完整的简单正确的资源处理机制。从本质上说， PTHREADS提供了常性（ immutable） RAII（见 3.5.1节）。尽管这相对于 Win32缺乏任何 RAII是一个极大的改进，但有时候确实需要改变给定键对应的槽位值的能力。手工清理原先的值也是可行的，但是如果这种能力是自动的话要好得多。

第四个问题是， PTHREADS假定清理函数在清理点是可调用的。如果在某个线程退出时，它的清理函数将要（直接或间接）调用的 API已经被反初始化（ uninitialized）了，那么对该清理函数的调用就可能是无效的。类似的一个情况，甚至可算是实践中更常见的情况是，如果清理函数是位于动态库中的，那么此时该清理函数可能已经不再存在于进程的地址空间中了，而这则意味着进程崩溃。

10.5.3 declspec(thread)和 TLS

在我们处理这些问题之前，我想先描述一个 TSS机制，该机制被 Win32平台上的大多数编译器所支持，用于简化对 Win32 TLS函数的使用。编译器允许你对变量定义使用 __declspec(thread)修饰符，像这样：

```
__declspec(thread) int  x;
```

现在 x就是线程相关的了，每个线程都会得到一份自己的拷贝。编译器将此类变量置于目标文件的 .tls节中，连接器则将各目标文件中的 .tls节合并到一起。当操作系统加载进程时，会去寻找可执行文件的 .tls节并创建一个线程相关的内存块来存放该节。每当诞生一个线程时，操作系统都会进行这个步骤，为新生线程创建一个相关的内存块（ TLS块）。

遗憾的是，这种策略虽然极其高效 [Wils2003d]，但有一个巨大的缺陷，导致它只适合用于可执行文件，而不适合动态库。它可以被用于隐式连接的动态库，从而在进程的加载期被加载，因为操作系统可以对所有在应用程序加载时加载的连接单元分配线程相关的块。问题在于，如果一个包含了 .tls节的动态库在进程已执行一段时间后被显式加载，操作系统无法回去对所有现存线程的线程相关块进行扩展，因此你的库将会加载失败。

我认为最好在任何 DLL中都避免使用 __declspec(thread)，即使是那些你认为总是会被隐式连接的 DLL。

在现代的基于组件的世界里，完全存在这样一种可能：某个 DLL被隐式地连接到一个组件，但该组件却是由某个可执行文件来显式加载的，这个可执行文件又是由另一个编译器创建的，甚至是用另一门语言编写的，并且，它先前并没有（隐式）加载过你的 DLL。你的 DLL无法被加载，依赖于它的组件也无法被加载。

10.5.4 Tss库

我曾为上面提到的有关 PTHREADS和 Win32的 TSS机制的四个问题大伤脑筋，最后我捋起袖子自己写了一个库，该库提供了我需要的功能。它包含八个函数和两个辅助类。其主函数与 C和 C++都是兼容的，如程序清单 10.6所示：

程序清单 10.6

```
// MLTssStr.h - 函数被声明为 extern "C"
int     Tss_Init(void);    /* Failed if < 0. */
void    Tss_Uninit(void);
void    Tss_ThreadAttach(void);
void    Tss_ThreadDetach(void);
HTssKey Tss_CreateKey( void (*pfnClose)()
                     , void (*pfnClientConnect)()
                     , void (*pfnClientDisconnect)()
                     , Boolean bCloseOnAssign);
void    Tss_CloseKey(     HTssKey  hEntry);
void    Tss_SetSlotValue( HTssKey  hEntry
                        , void     *value
                        , void     **pPrevValue /* = NULL */);
void    *Tss_GetSlotValue(HTssKey  hEntry);
```

跟所有的良好的 API一样，它具有 Init/Uninit[注 20]方法，以便确保在任何客户使用它之前得到正确的初始化。它还含有另外两个函数，用于 attach（附加）到某个线程以及从某个线程 detach（脱离），我们待会儿再来讨论它们。

[注 20]给那些使用英式拼写的人一个小小的提示：你可以通过使用缩写来避免跟你的美国朋友在“ Initialise还是 Initialize”上进行无意义的争吵。

其中四个函数采用了操纵键的习惯做法。然而，这些函数提供了额外的功能。 Tss_CreateKey()的参数

pfnClose接受一个可选的回调函数，以便提供线程终止时的清理能力，如果你不需要，那么传递 NULL就行了。此外，如果你希望在槽位值被覆写的时候调用清理函数，就必须指定 bCloseOnAssign参数为 true。

pfnClientConnect和 pfnClientDisconnect两个参数接受可选的回调函数，用于防止代码过早地消失。你可以以任何恰当的方式来实现这两个回调函数，只要能够确保 pfnClose所指的函数在被需要时仍存在于内存中并可调用即可。我在使用该 API的过程中曾遇到过这样的情况：为其他 API指定 Init/Uninit函数，或者去锁定和解锁一个位于内存中的动态库，或者必要时二者兼而有之。

Tss_CloseKey()和 Tss_GetSlotValue()的语义跟你期望的一样。然而， Tss_SetSlotValue()跟它的 PTHREADS/Win32版本相比具有一个额外的参数： pPreValue，如果该参数是 NULL的话，原先的值就会被覆盖掉，并且按照该键创建时要求的清理方式被清理。如果该参数不是 NULL，任何清理操作都会被跳过，原先的值会被返回给调用者。这就允许对槽位值进行粒度更细的控制，同时在缺省情况下可提供强大的清理语义。

作为一个 C API，下一步很自然是将它封装到域守卫类中去，我提供了两个。第一个是 TssKey类，该类并不是特别引人注意，它只是用于简化接口，并且将 RAII用于关闭键，因此我只展示其公共接口：

程序清单 10.7

```
template <typename T>
class TssKey
{
public:
  TssKey( void (*pfnClose)(T )
        , void (*pfnClientConnect)()
        , void (*pfnClientDisconnect)()
        , Boolean bCloseOnAssign = true);
  ~TssKey();
public:
  void  SetSlotValue(T value, T *pPrevValue = NULL);
  T     GetSlotValue() const;
private:
  . . . // 成员，隐藏拷贝构造函数和赋值操作符
};
```

其实现中包含了用于确保 sizeof(T) == sizeof(void*)的静态断言（见 1.4.7节），这是为了防止任何想要将大对象按值保存到槽位中的错误企图。用户欲保存的值会被转型至参数化类型（即 T），节省了你在客户代码中的工作。

另一个类则要更有趣一些。如果你使用槽位值的目的是为了创建单个实体然后去复用它，那么你通常应该遵循程序清单 10.8中的模式：

程序清单 10.8

```
TssKey key_func(closeThing, . . .);
. . .
OneThing const &func(Another *another)
{
  OneThing *thing = (OneThing*)key_func.GetSlotValue();
  if(NULL == thing)
  {
    thing = new OneThing(another);
    key_func.SetSlotValue(thing);
  }
  else
  {
    thing->Method(another);
  }
  return *thing;
}
```

然而，如果该函数更复杂一些（大多数时候都是这样），那么其内部就会出现多处可能改变槽位值的地方。每一处都存在着资源泄漏的可能，这种资源泄漏是因在调用 SetSlotValue()之前过早返回而导致的。出于这个原因，我提供了域守卫类 TssSlotScope，见程序清单 10.9。我得承认我对该类有些过分的偏爱，因为它是纯粹的 RAII的体现。

程序清单 10.9

```
template <typename T>
class TssSlotScope
```

```
{
public:
  TssSlotScope(HTssKey hKey, T &value)
    : m_hKey(hKey)
    , m_valueRef(value)
    , m_prevValue((value_type)Tss_GetSlotValue(m_hKey))
  {
    m_valueRef = m_prevValue;
  }
  TssSlotScope(TssKey<T> key, T &value);
  ~TssSlotScope()
  {
    if(m_valueRef != m_prevValue)
    {
      Tss_SetSlotValue(m_hKey, m_valueRef, NULL);
    }
  }
private:
  TssKey  m_key;
  T       &m_valueRef;
  T const m_prevValue;
// Not to be implemented
private:
  . . . // 隐藏拷贝构造函数和赋值操作符
};
```

它的构造函数接受一个 TSS 键（TssKey<T>或 HtssKey）和一个对外部值变量的引用，然后它通过调用 Tss_GetSlotValue()将该外部变量设置为槽位值。

在析构函数中，外部变量的值跟槽位的原始值作比较，如果外部变量的值发生了改变，则通过 Tss_SetSlotValue()来更新槽位值。现在我们的客户代码更简单了，依赖于 RAII来对线程的槽位值进行必要的更新。

程序清单 10.10

```
OneThing const &func(Another *another)
{
  OneThing                  *thing;
  TssSlotScope<OneThing*> scope(key_func, thing);

  if( . . . )
    thing = new OneThing(another);
  else if( . . . )
    thing = . . .;
  else
    . . .

  return *thing;
} // scope的确定性析构确保 Tls_SetSlotValue()得到调用
```

现在我们已经看到如何使用 Tss 库了，但是它的工作原理是什么呢？我把推断它的实现的任务留给你 [注 21]，不过在此我们需要讨论一下线程通知是如何进行的。这涉及到我到目前为止还没提到的两个函数：Tss_ThreadAttach()和 Tss_ThreadDetach()。这两个函数应该在线程开始和结束时分别被调用。如果可能的话，你可以通过将它们挂钩到操作系统或运行时库的基础设施上来达到这个目的，否则你就得手工进行相应的调用。

[注 21]或者你也可以浏览一下配书光盘中的内容，因为我收录了该库的源代码。但是你得注意，里面都是些老旧的代码，并且没那么漂亮！它可能并不是最佳化的，因此你应该把它当成一个技术指南，而非 TSS 库的绝佳实现。

在 Win32上，所有 DLL都导出其进入点 DllMain()[Rich1997]，该函数在进程加载/卸载以及线程开始/结束时会收到通知消息。在 Synesis Win32库中，基本 DLL（MVCMNBAS.DLL）就是在其 DllMain()中接收到 DLL_THREAD_ATTACH 通知时调用 Tss_ThreadAttach()，并且在接收到 DLL_THREAD_DETACH 时调用 Tss_ThreadDetach()的。由于它是一个普通的 DLL，所以任何可执行文件的其他组成部分都可以使用 Tss 库，而不用考虑底层的设置动作，拿来就用，很简单！

程序清单 10.11

```
BOOL WINAPI DllMain(HINSTANCE, DWORD reason, void *)
{
```

```
switch(reason)
{
  case DLL_PROCESS_ATTACH:
    Tss_Init();
    break;
  case DLL_THREAD_ATTACH:
    Tss_ThreadAttach();
    break;
  case DLL_THREAD_DETACH:
    Tss_ThreadDetach();
    break;
  case DLL_PROCESS_DETACH:
    Tss_Uninit();
    break;
}
. . .
```

在 UNIX上，该库的函数 Tss_Init()中调用 pthread_key_create()，创建一个私有的、未被使用过的键，其唯一的意图是确保该库在每个线程结束时都能够接收到一个回调，并借此机会调用 Tss_ThreadDetach()。由于在 PTHREADS中并没有支持“每线程（per-thread）”初始化函数的机制，所以当它的 Tss库被请求从一个不存在的槽位中取数据时就会有“温和”的反应，并且当被请求保存一个槽位值而槽位不存在时就会新建一个。因此，Tss_ThreadAttach()可以被看作这样一种机制，它响应线程的诞生并高效地扩展所有活跃键，而并不是在线程执行的过程中逐次完成。

如果你不使用 PTHREADS或 Win32，或者你不希望将你的库放在一个 Win32 DLL中，你就应该确保所有线程都调用了 attach/detach函数。然而，即使你不能或没有这么做，当最终对 Tss_Uninit()的调用到来之时，该库仍然会执行被注册的清理函数，以便清理所有键对应的所有槽位。

这是一个强大而完备的机制，唯一可能出现问题的情况就是：如果你的用于释放资源的清理函数必须在分配对应资源的线程中被调用起来，而且你又没法保证及时而无遗漏的线程 attach/detach通知，你就可能会遇到麻烦了。但话说回来，你干吗非得如此呢？你需要的是什么——我们只不过是原生质和硅！

[译注]由于 Tss库清理资源的机制是调用客户代码原先注册/挂钩过的清理函数（clean up），然后在 DLLMAIN里面收到 DLL_PROCESS_DETACH/DLL_THREAD_DETACH消息的时候再去回调那些挂钩的清理函数。这个机制就带来了本段描述的问题。想想看，假设你有一个线程 A，该线程为它的线程本地存储分配了一些资源或空

间，并注册了一个清理函数以便归还这些资源，但你有一个要求，即你的清理函数必须得在线程 A的上下文当中被调用。而基于 Tss库的实现机制，只有当它收到 DLL_THREAD_DETACH通知的时候才会去调用注册的清理函数，因此一旦不能保证及时而无遗漏的线程 attach/detach通知的话，这些清理函数就得等到 Tss库收到 DLL_PROCESS_DETACH通知的时候才会被调用起来了，而彼时的情况应该是除主线程之外所有其他线程都已结束了，换句话说，这些清理函数都在主线程上下文中被调用，从而不能满足你的要求。

10.5.5 TSS的性能

到目前为止我们还没有提及性能问题。当然，库的成熟和周全性，加上使用了互斥体来串行化对存储区的访问，这两个事实都意味着该库跟 Win32 的 TLS 实现相比代价要显著得多。Win32 TLS 确实非常快 [Wils2003f]。从某种意义上说，这不是问题，因为如果你需要这种功能的话，你就无可避免地要付出代价。其次，线程切换的代价是相当可观的，也许可达上千个时钟周期 [Bulk1999]，因此 TSS中某些实现的开销相比之下可能不足为虑。然而，我们也不能忽视这个问题。有一个手段可以被用来最小化 Tss库或任何 TSS API 的开销：将 TSS数据沿着调用链往下传递，而不是让每一层都自己去获取它们，后一种做法由于 TSS基础设施中的竞争条件从而可能会导致剧烈的上下文切换。显然这种开销的最小化在系统库函数或其他被普遍使用的函数中无法实现，但是在你自己的应用程序的实现中还是可能的。

此外，对于 Tss存储，使用 TssSlotScope模板是个不错的选择，因为它仅在槽位值需要被改变时才会去更新它。

一如既往，策略的选择依然取决于你自己，取决于你对性能、健壮性、编程难易程度以及所需的功能之间的权衡。

附录A 编译器和库

A.1 编译器

在探讨本书中的问题时，我使用了以下九款编译器：

- Borland C/C++ 5.5(1)、5.6、5.64
- CodePlay VectorC 2.06
- Comeau C++ 4.3.0.1、4.3.3
- Digital Mars C/C++ 8.30–8.38
- GCC 2.95、3.2
- Intel C/C++ 6.0、7.0、7.1
- Metrowerks CodeWarrior 7、8
- Microsoft's Visual C++ 6.0、7.0、7.1
- Watcom 12.0（也称为Open Watcom 1.0）

其中，以下的编译器是完全免费的：

- Borland 5.5(1)：可到http://www.borland.com下载
- GCC（所有版本）：有好多种下载来源，参见http://gcc.gnu.org/install/binaries.html
- Watcom：可到http://www.openwatcom.org/下载

以下编译器可以以某种形式免费获得：

- Digital Mars（所有版本），仅限命令行工具：http://www.digitalmars.com/
- Intel（最近的版本），仅限Linux版本：http://www.intel.com/
- Visual C++ 7.1，仅限命令行工具：http://www.microsoft.com/

以下是商业编译器：

- CodePlay VectorC：http://www.codeplay.com/
- Comeau：http://www.comeaucomputing.com/
- Metrowerks CodeWarrior：http://www.metrowerks.com/

这些编译器厂商即便不以任何形式免费开放他们的编译器，大多数至少也提供了免费的试用版。这里我并

不打算对它们进行任何比较，你到网上随便搜一搜就能看到足够多关于这方面比较的文章。你也可能已经在本书的讨论中看到了我对其中大部分编译器的印象。无论如何，我坚信我们应该在工作中使用多种编译器，而不是只用一个（见附录D）。这些编译器中的每一个都能够提供一些其他编译器所不能提供的有意义的警告信息。

我要感谢所有这些编译器厂商在我准备本书的过程中热心地提供编译工具。我很高兴地看到，从最小的编译器开发团队（Digital Mars的开发“团队”只有一个人：Walter Bright）到某些世界上最大的IT公司都表现出这种支持精神。非常感谢！

A.1.1 编译器优化

无论何时，只要我是在测量代码的性能，而不仅仅是验证代码的正确性，那么所有编译都是在最大化速度优化以及（尽量）针对宿主系统处理器的情况下进行的。我的宿主系统的处理器是奔腾4，在Win32上编译器的设置如下：

编译器	优化设定
Borland 5.6	-O2 -6
CodePlay VectorC 2.06	-nodebug -optimize 10 -max -target p4
Comeau 4.3.0.1 (Visual C++ 6.0 backend)	/O2 /Ox /Ob2 /G6
Digital Mars 8.38	-o+speed -6
GCC 3.2	-O7 -mcpu=i686
Intel 7.0	-Ox -O2 -Ob2 -GS -G7
Metrowerks CodeWarrior 8	-opt full -inline all,deferred -proc generic
Microsoft Visual C++ 6.0	-Ox -O2 -Ob2 -G6
Microsoft Visual C++ 7.1	-O2 -Ox -Ob2 -GL -G7
Watcom 12.0 (Open Watcom 1.0)	-oxtean -ei -6r -fp6

A.2 库

在本书中，只要有可能，我就尽量使用不依赖于任何特定库的例子。然而，作为STLSoft库的主要作者，我自然会从中选出许多现成的例子。不过，在大多数情况下，我尽量不提这些例子出自何人之手，这并不是惺惺作态的谦虚，而是因为我不喜欢看到一本外表道貌岸然实质上却是在推销作者自己的私货的书。为了理解每一个新论题，你不得不去理解一大堆实实在在、互相关联的组件，这尤其令人恼火。

我对库的不同实现进行性能比较时，总是尽量揭露它们的实质，因为我必须让各位读者朋友能够独立地验

证这些测试结果，否则这些结果就不那么可信。

其他一些例子则来自我所在公司 Synesis Software（http://synesis.com.au/）的库 [注 1]，还有就是来自 Boost 库（http://boost.org/）。Boost 是由 C++领域中的一些顶尖人物组成的开源社群。其余则是我在跟形形色色的客户的工作中目击到的一些错误，不过经过了重新整理（以免引起不满）。再剩下来的就是凭空捏造的了，目的仅在于阐明一些要点。

[注 1]自从我攻读博士学位（1992-1995）时，这个库就开始构建了，因此其中有许多令人不愉快的“神秘”代码。如果你对配书光盘中包含的或者本书中涉及的 Synesis 代码中有严重意见的话，请先不要忙着给我写错误报告，我或许已经知道了你说的这个错误，而且已经将它放到即将要做的事情清单中了，并且在后来的任何新生库（例如 STLSoft）中也已避免重复犯这些错误了。作为一个爱唠叨的家伙，我当然也希望从你那儿听取意见，但你当然也不想浪费你自己的时间和精力。

A.2.1 Boost

Boost 是近年来最为杰出的 C++库，它的背后有许多 C++顶尖人物在支持。Boost 库中包含了一些非常强大的组件，向其中提交各种组件的人们也是一个极其庞大的群体。Boost 已经被公认为“准标准库”。很可能下一代 C++标准库中的许多新特性就是来自 Boost。

配书光盘中也包含了 Boost 的一个发布版，你也可以从 Boost 站点（http://www.boost.org）免费获取最新的发布版。

A.2.2 STLSoft

STLSoft 库诞生于几年前，一开始只是一些 STL 式的序列式容器，用于封装文件系统的枚举 API 以及 Win32 的注册表 API。在最近几年中，它的内容不断扩充，到目前为止已经涵盖一定数量的技术并涉足多个操作系统领域。STLSoft 库中的许多部分都是起源于 Synesis Software 专有的代码，因此它的背后隐藏着人们许多年来的努力成果。STLSoft 关注的重点是效率、健壮性以及可移植性，并从始至终信守“只用头文件”的准则。当然，STLSoft 的成员没有 Boost 那么多。

我不太清楚 STLSoft 中是否会有特性被下一代 C++标准库接纳，但到目前为止我还并没有将这看成工作的重心。本书审稿人之一就是 C++标准委员会的成员，曾建议我将 STLSoft 库中的某些组件提交给标准库，所以……谁知道呢？

配书光盘中包含了 STLSoft 库的一个发布版本，你也可以从 STLSoft 站点（http://www.stlsoft.org/）免费

获得。

A.2.3 其他库

另外还有其他一些我认为值得一提的库：

Open-RJ（http://www.openrj.org/）是一个简单小巧的结构化文件读取库，遵循 Record-JAR[Raym2003]格式。Greg Peet跟我一起编写了这个库。其实现是用 C 编写的，但提供了到多种语言（库）的映射，包括 C++、D、Ruby 以及 STL。Record-JAR 是一种非常简单的格式（Open-RJ 库整个编译出来才 5KB），但其有用的程度令人吃惊[注 2]。

[注 2]Open-RJ 网站以及本书附带光盘的 HTML 内容就是通过 Ruby 脚本从 Open-RJ 格式的文件生成的。该格式还被我用作 Arturius 编译器多路分发器（见附录 C）的配置数据格式。

PThread-win32（http://sources.redhat.com/pthreads-win32/）是一个 GNU 库，它几乎提供了所有 PTHREADS 规范的 Win32 版本。如果你在 Windows 系统上编写 UNIX 程序的代码，PThread-win32 对你就是极其有用的了。当然，如果你的膝上电脑装有多系统，能够另外启动 Linux 的话，你可能并不需要它，但你仍然可以利用它来实现“单一代码服务两种系统”的目的。

RangeLib（http://www.rangelib.org）是区间概念的大本营，John Torjo 和我开发了这个库。区间这一概念仍然处于不断发展中，除了 Boost 和 STLSoft 中的实现外，它还可能在其他语言中实现，但我们需要耐心等待（见“尾声”）。

recls（http://www.recls.org/）（含义为 recursive ls）是我编写的一个平台无关的递归文件系统搜索库。我最初编写它是因为我对每次编写递归搜索时都得编写千篇一律的代码感到非常厌烦。它现在成了我在 C/C++ Users Journal 上的专栏“Positive Integeration”中的理想范例。该专栏描述的是如何将 C/C++ 与其他语言和技术整合在一起的技术。recls 库以 C++ 实现（使用 STLSoft 库），但它也向外界提供了 C 接口（见第 7 章和第 8 章）。它不断将枝叶伸展到其他一些递归搜索领域，并将其实现映射到 COM、D、JAVA、.NET、Ruby 以及 STL 中，而且这个列表还在不断变长。

zlib（http://www.zlib.org/）或许是免费的压缩库中最著名的一个。它使用起来非常简单，而且幸运的是，它还可以接受外部内存配置器（见 32.3 节），这正是我衷心拥护的做法。

A.3 其他资源

以下所有资源对于C++程序员来说都是有用的，同时也在本书的研究和写作过程中对我起到了帮助作用。

A.3.1 期刊

- BYTE：http://www.byte.com/
- C/C++ Users Journal：http://www.cuj.com/
- CVu：http://www.accu.org/
- Dr. Dobb's Journal：http://www.ddj.com/
- Overload：http://www.accu.org/
- The C++ Source：http://www.artima.com/cppsource/
- Windows Developer Network：http://www.windevnet.com/

A.3.2 其他语言

- D：http://www.digitalmars.com/d
- Java：http://java.sun.com/
- .NET：http://microsoft.com/net/
- Perl：http://cpan.org/；http://perl.org/
- Python：http://python.org/
- Ruby：http://ruby-lang.org/

A.3.3 新闻组

- comp.lang.c++.moderated
- comp.std.c++

附录 B “ 谦虚点儿，别骄傲！ ”

在尽我所能地逃避了一段时间的实际工作之后（这样我才能把更多的时间花在骑单车上！），我完成了我的博士学位，接着就一头扎进 “ 现实世界 ” 中找工作去了。我还记得那段时间我总是被那些可能会雇佣我的老板们弄得尴尬不已，因为他们对于我的简历上列出来的条件和资格无动于衷，也根本没有热情地满足我（自负的）薪水期望的打算 [注 1]。问题的关键似乎在于我缺乏他们所谓的 “ 经验 ”。我为他们的目光短浅感到无言以对，接连好几个星期我都是一边找工作一边抱怨着世道不公。难道他们就不能瞪大眼睛，看看我有多聪明，然后给我好多好多钱？

[注 1] 在接受我所选择的工作后，谈薪水时，我学到了非常重要的一课。经理说：“ 你对薪水的期望值是多少？ ”我使出了我所认为最婉转的说法，回答道：“ 大概在 20000 到 25000 英镑之间的某个数目吧 ”。结果还没等我缓过气儿来他就迅速给出了回答：“ 我们愿意以 20000 英镑的薪水提供给你这份工作。 ”后来我再也没有干过这种傻事。

当然，在经过了这些年艰苦的磨练之后，我完全体会到了经验的重要性。我曾与一些资历非常高但毫无实际经验的人共事过，也曾与一些只有很少或根本没有资历但才华横溢的工程师一起工作过，我也曾经不得已跟一些并不想用心做事、也对学习新东西丝毫不感兴趣的人共事过。如果说我们这个职业是一张纸，那么这些人就好比纸上的污点。软件开发业的低劣性并不能完全怪罪于那些无知的经理、狡猾的市场营销人员以及总是急不可耐的用户，实际上很大程度上要归咎于这个行业的某些从业人员，他们应该去从事一些即使玩忽职守也不会造成像在软件业里这样大的危害的行当，而不应该混迹于这个聚集着人类想象力的最复杂的创造性的行业。

正如我的母亲当年总不厌其烦地跟我说的那样：“ 谦虚点儿，别骄傲！ ”。所以，抱着谦虚的心态，或许还能够为你带来一点娱乐。下面我就讲一讲当年我从一个新手直到 “ 知道一些东西 ” 一路蹒跚走来途中犯的一些错误。如果你在我的经历中看到了你自己的影子，嘘！别作声，还是让我来挨枪子儿吧！

B.1 操作符重载

让 C++ 新手兴奋的事情之一就是操作符重载，有点讽刺意味的是，它可能是最容易被误用的语言特性之一。

我编写的第一个字符串类具有一个庞大的公共接口，下面只摘录了其中最糟糕的一部分：

```
class DLL_CLASS String // 不分青红皂白地导出所有成员
  : public BaseObject // 一个多态的字符串……？
{
```

```
. . .
// 将该串的长度置为 nInitLen，并用 cDef 来填充它
String &Set(int nInitLen, char cDef = '\0');
. . .
Bool   SetLowerAt(int nAt); // 如果 nAt 处元素被成功地设置为小写就返回 true
Bool   SetUpperAt(int nAt); // 如果 nAt 处元素被成功地设置为大写就返回 true
. . .
String &operator =(int nChopAt); // 将该串的长度截断为 nChopAt
. . .
String &operator +=(int nAdd); // 将该串增长 nAdd
. . .
String &operator -=(int nReduce); // 将长度减小 nReduce
// 将长度减至最后一个与 cChopBackTo 匹配的字符处的长度
String &operator -=(char cChopBackTo);
. . .
// 将串的内容循环左移 /右移
String &operator >>=(int nRotateBy);
String &operator <<=(int nRotateBy);
. . .
String &operator --();    // 将所有字符变成小写
String &operator --(int); // 将所有字符变成小写
String &operator ++();    // 将所有字符变成大写
String &operator ++(int); // 将所有字符变成大写
. . .
// 将整型值转换为字符串，并返回一个局部静态对象
static String NumString(long lVal);
. . .
operator const char *() const; // 隐式转换为 C 字符串
. . .
operator int() const; // 隐式转换，返回该字符串的长度
. . .
// 按值返回一个元素
const char operator [](int nIndex) const;
. . .
String &operator ^=(char cPrep);          // 前部添加字符
```

```
    String &operator ^=(const char *pcszPrep); // 前部添加字符串
    . . .
};
```

简直没有比这更糟的工作了，所有能干的糟糕事儿都干了。如果你对此的反应竟然不是强烈不满的话，我就会觉得更加悲哀了。不过，令人惊奇的是，我当时居然正确地定义了加法操作符：我将它们定义为自由函数，并使用 operator +=()进行实现（见 25.1.1节）。

```
String operator +(const String &String1,const String &String2);
String operator +(const String &String1,const char *pcszAdd);
String operator +(const char *pcszAdd,const String &String2);
String operator +(const String &String1,char cAdd);
String operator +(char cAdd,const String &String1);
```

我最近在常去的一个新闻组上居然还看到有几个人发帖子说，“ operator ++()意味着使所有字符变成大写 ”是个好主意，这把我吓了一大跳，但愿他们只是没经过大脑思考才这么说的。

B.2 后悔 DRY

在我编写上面展示的字符串类的那段时间里，我还编写了我的第一个 C++链表类。我很喜欢封装的概念。哎呀，只可惜我太喜欢 DRY（Don't Repeat Yourself，不要重复你自己）原则了 [Hunt2000]。该原则说同样的信息不应该出现在一个以上的地方，我在编写链表类时天真地误用了这个原则，因为我让 GetCount()成员函数每次被调用时都将链表从头到尾数一遍，这下时间复杂度可就不是 O(1)了。

我甚至都不敢再提自平衡树（self-balancing tree）了，在实现它时，我犯了类似的错误！

B.3 偏执式编程

软件工程领域常出现的偏执做法就是知识产权保护，我也曾中过这样的邪。下面就是当年的一个“ 劣迹 ”。

你们中的 Win32 GUI 程序员应该很清楚列表视图（ListView）控件的特性和缺点。我第一次尝试改进它是通过 MFC、多重继承以及一些邪门歪道来实现的。扩展的特性包括彩色子条目文本和背景、每子条目（per subitem）的用户数据和图像、二维遍历的编辑框等等，它们都是通过一个 CColouredListCtrl 类 [注 2]提供的。

[注 2]注意，这里我还天真地使用了 CClassName 命名习惯。我现在知道这个命名约定的意图是为了将 MFC 核

心类与用户自己创建的类区分开来，不过似乎并没有人向用户提起过这一点，包括那些写诸如“ 24小时学会使用MFC编写工业强度的应用 ”之类的书籍的家伙们。这种可怕的习惯至今仍然苟延残喘于某些领域。

```
class DLL_CLASS CColouredListView
  : public CListView
  , public CColouredList
{
  . . .

class DLL_CLASS CColouredListCtrl
  : public CWnd
  , public CColouredList
{
  . . .
```

姑且不说当时我根本不知道会有哪些用户想要使用这种东西，但我确实花了大力气去隐藏这个“如此智能”的改进控件的实现，当然了，借助于一些“令人愉快”的“技巧”：

```
class _CLCInfoBlock; // 前导声明
#define _CLC_RESBLK  (208)

class DLL_CLASS CColouredList
  : public COwnerDrawCtrl
{
  . . . // 许多、许多方法
// 成员
protected:
  HWND          &m_hWnd;          // HWND引用
  _CLCInfoBlock  &m_block;        // Info块
private:
  BYTE          m_at[_CLC_RESBLK]; // 保留
};
```

我相信你猜到了其余的情况：_CLCInfoBlock类是定义在CColouredList类的实现文件中的。在CColouredList的构造函数中，一个_CLCInfoBlock的实例被就地构造在m_at存储空间中：

```
CColouredList::CColouredList(HWND &hWnd)
   : m_hWnd(hWnd)
   , m_block(*new(m_at) _CLCInfoBlock)
{}
```

姑且不说字节数组并不能保证能为某个用户自定义类型的就地构造提供正确的对齐条件，单就 208这个常量就让人迷惑：究竟为什么要将 _CLC_RESBLK定义为 208？这种做法太糟糕了，不管以什么合理方式来衡量它都属于不当的行为，我想不会有人愿意购买像 CColouredList这样的东西的。

B.4 精神错乱！

几年前，我接到一个任务，是编写一个通用的库，用于替代某个现有的定制数据库子系统中原先使用的标准库 IOStream、string以及容器，并重写原有的解决方案，让它使用我们新编写的类，同时还不能对这个子系统的设计或公共接口作丝毫改变（尽管实际上这个子系统从来也没有迈出过它的开发团队半步）。

在我们的努力接近尾声的时候，我们被一件麻烦事儿缠住了：我们必须为每个标准库函数准备一个单独的包含文件（例如 fprintf.h）。还得为每一个“标准库组件—自定义库组件”对应关系各定义一个上下文相关的 typedef（见 18.2.2节）。

我犯了一个简单但致命的错误。我应该向老板提出那个子系统本身的设计是差劲的，其原本的实现也是差劲的，同样，只是出于性能方面的要求而去重写一大堆标准库组件也是个糟糕的主意，即便最终能够完成也得耗上好几个月。然而我没有这么做，而是只顾及我与那位系统最初的实现者之间的工作关系和私人关系（此君还是客户端开发领导），未作出正确的判断就一头扎进那个项目中去了。我做了我们这些程序员总是欣然去做的事情，即不分青红皂白，不管是否值得就跑上去迎接挑战了。这就好像你本该在家陪孩子吃匹萨的，却跑到健身房做了五个小时的运动！

我相信这个故事对你应该并不陌生。一项“应该”在三个星期内完成的工作拖了几乎三个月才完成了一半，到那时候我们的工作伙伴关系都已经变味儿了。我被这项工作本身弄得极度苦闷，并且严重丧失动力，而且我的同事之前树立起来的好名声都给毁掉了。我的合同被提交再审，项目被终止，我如释重负，因为再也不用去烦它了。我的那个同事不久也离开了他呆了好几年的公司，接连几个月都处于失业状态。

这个教训很简单：如果皇帝确实没有穿什么新衣服的话，看在上帝的份上，说点儿什么，不要一言不发，埋头傻干，否则你终将会后悔的！