第5章 重用设计

要在程序中重用库和其他代码，这是一个重要的设计策略。不过，这只是重用策略的前一半。另一半则是设计和编写可以在程序中重用的代码。你可能已经发现了，在设计得好和设计得不好的库之间存在显著的差异。设计得好的库很好使用，而设计不好的库可能会让你心生嫌恶，以至于干脆放弃，自己动手来编写代码。不论你是在编写一个专门设计要由其他程序员使用的库，还是仅仅确定一个类层次体系，设计代码时都应该把重用谨记心头。你无法知道在以后的某个项目中是否还需要一个同样的功能。

第2章介绍了重用的设计原则。第4章解释了如何在设计中结合库和其他代码来应用这种原则。本章则讨论重用的另一面：如何设计可重用的代码。这里以第3章所述的面向对象设计原则为基础，还将介绍一些新的策略和指导原则。

读完本章之后，你将了解到：

- 重用方法论：为什么要设计可供重用的代码
- 如何设计可重用的代码
  - 如何使用抽象
  - 建立代码以供重用的三种策略
  - 设计可用接口的六种策略
- 如何协调一般性和易用性

5.1 重用方法论

应当把代码设计为你和其他程序员都能重用。这条原则不仅适用于你特意希望其他程序员使用的库和框架，还适用于为程序所设计的任何类、子系统或组件。一定要谨记这样一条座右铭："编写一次，经常使用"。采用这种设计方法有许多原因：

- 代码很少只用于一个程序中。在写代码时，你可能没有打算这些代码会得到重用，不过几个月或者几年之后，你会发现自己或者你的同事可能在一些类似的项目中采用了这个程序的一些组件。由此可以了解到，代码可能会以某种方式再次得到使用，所以开始的时候就应该考虑充分，正确地做出设计。
- 设计可重用的代码能够节省时间和金钱。如果设计代码时根本不考虑将来的使用，以后当遇到需要实现一种类似功能的时候，你或你的同事肯定要花时间再做一次同样（或类似）的设计。即使没有完全杜绝重用，但如果你提供的接口不好，或者缺少功能，将来使用这段代码时就需要花费额外的时间和精力。
- 同组的其他程序员必须能够使用你所写的代码。你的代码可能只适用于当前的特定程序，即使在这样一些情况下，你有可能并非孤身作战来完成项目。如果你花时间为你的同事提供了设计得当、功能完备的库和代码段以便其使用，他们肯定会非常感激的。你很清楚，如果别人写了不好的接口或者欠考虑的类，在使用这些接口和类时是多么的苦恼。设计可重用的代码还可以称为协

作编码（cooperative coding）。应当适当地编写代码，使得除了当前项目中的程序员可以由此得到好处，其他项目（包括将来项目）中的程序员也能由此获益。

- 你将成为自己所做工作的最大受益人。有经验的程序员从来不会把代码扔掉。经过一段时间以后，他们就能建立起自己的一个发展工具库。你无法知道将来是否还需要同样的功能。例如，本书作者之一最早是在上研究生的时候学过网络编程课程，他写了一些通用网络例程来创建连接、发送消息和接收消息。从那以后，每次开发涉及到网络的项目时他都会参考这些代码，而且在许多不同的程序中重用了其中的一些代码段。

> 如果你作为公司员工设计或编写了一些代码，那么拥有知识产权的是你的公司，而不是你。如果你已经退出了这家公司，仍然保留你所做设计或代码的副本往往是非法的。

5.2 如何设计可重用的代码

可重用的代码要满足两个主要目标。首先，它要有足够的一般性，可以用于稍有不同的多种用途，或者可以在不同的应用领域中使用。如果程序组件带有特定应用的有关细节内容，将很难在其他程序中重用。

可重用的代码还要易于使用。不需要花太多时间来理解其接口或功能。程序员必须能够很轻松地将其纳入到他们的应用中。

> 可重用代码要有一般性，而且要易于使用。

你提供的可重用代码集并不一定非得是正式的库。它可以是一个类、一个函数集，或者是一个程序子系统。不过，与第4章一样，这一章也使用“库”这个词泛指你编写的所有代码集。

需要注意的是，本章使用“客户”一词表示使用你的接口的程序员。不要把这里的客户与运行你的程序的“用户”混为一谈。本章还使用“客户代码”一词表示为使用你的接口而编写的代码。

设计可重用代码时，最重要的策略就是抽象。第2章通过真实世界中的电视对抽象做了对照分析，你可以通过电视提供的接口来使用电视，而无需了解其内部工作原理。类似地，在设计代码时，应当将接口与实现清楚地分离。这种分离使得代码更易于使用，这主要是因为客户要使用此功能时，不必了解其内容的实现细节。

基于抽象，代码会分离为接口和实现，所以设计可重用代码时所强调的主要就是这两个领域。首先，必须适当地建立代码结构。你要使用哪些类层次体系？是否应当使用模板？应当怎样把代码划分为子系统？

其次，必须设计接口（interface），这是库或代码的“入口”，程序员就是使用这样一些入口来访问你提供的功能。要注意，接口不一定是一个正式的API。这个概念涵盖了你提供的代码与使用它的其他代码之间的任何边界。类的公共方法、函数原型的一个头文件等等就是很典型的接口。

“接口”一词可以表示一个访问点，如一个单个的函数或方法调用，或者可以表示一个完整的接口集，如一个API、类声明或头文件。

5.2.1 使用抽象

第2章已经学习了抽象的原则，在第3章中对抽象在面向对象设计中的应用有了更多了解。要遵循

抽象原则，应当为代码提供一些接口来隐藏底层的实现细节。在接口和实现之间应当有一个清晰的界线。

使用抽象不仅对你有好处，对使用你代码的客户也有好处。客户会受益，这是因为他们不必操心实现细节就可以充分利用你提供的功能，而不必理解这些代码具体是如何工作的。你会受益，原因在于你能修改代码而不用修改代码的接口。因此，无需客户修改其使用代码，你也能提供升级和做出修正。利用动态链接库，客户甚至不需要重新生成其可执行程序。最后一点，你们都会受益，这是因为作为一个编写库的程序员，可以根据你希望提供什么样的交互以及想要支持什么样的功能来指定接口。接口和实现之间如果能清楚地分离，就可以避免客户采用你不期望的方式不适当地使用库，如若不然，这可能会导致预想不到的行为和bug。

假设你在编写一个随机数库，而且希望为用户提供某种方法来指定随机数的范围。一种不好的设计是设置公共的全局变量或类成员，以供随机数生成器实现在内部使用，从而影响（改变）范围。这种设计得不好的库要求客户代码直接设置这些变量。好的设计则会隐藏内部实现所用的变量，而提供一个函数或方法调用来设置范围。如此一来，客户就不必理解内部算法。另外，由于实现细节未公开，你就能修改算法而不会影响客户代码与库的交互。

有时，库要求客户代码保留从一个接口返回的信息，以便将这些信息传递至另一个接口。这种信息有时称为一个句柄（handle），通常用于跟踪一些特殊的实例，这些实例需要记住调用之间的状态。如果你的库设计需要一个句柄，就不要暴露它的内部细节。要将该句柄置于一个不透明（opaque）的类中，程序员无法访问此类的内部数据成员。不要让客户代码调整这个句柄内部的变量。作为一个不佳设计的例子，本书作者之一就确实曾经用过一个不好的库，这个库要求他在一个原以为不透明的句柄中设置一个结构的特定成员，如此才能打开错误日志记录。

> 在编写类时，C++没有提供相关机制来支持好的抽象。必须将私有数据成员和方法声明放在公共方法声明所在的同一个头文件中。第9章介绍了一些技术，可以用这些技术绕过这些限制来提供清晰的接口。

抽象如此重要，它应当指导你的整个设计。在做每一个决策时，要问问自己你的选择是否满足抽象的原则。要从客户的角度来考量，并确定是否需要了解接口中的内部实现。一般不要违背这个原则。

5.2.2 适当地建立代码结构以优化重用

必须从设计的一开始就考虑重用。以下策略有助于你适当地组织代码。需要注意，这里所有策略都强调了代码的一般性。设计可重用代码的第二个方面是要易于使用，这与接口设计关系更大，将在本章后面介绍。

避免将无关或逻辑上分离的概念混杂在一起

在设计一个库或框架时，要使它集中于一个任务或一组任务。不要将无关的概念（如随机数生成器和XML解析器）混杂在一起。

即使你并非专门设计重用的代码，也要牢记这个策略。很少会完全地重用整个程序。与此不同，程序中的部分或子系统会直接结合到其他应用中去，或者有所调整以适应稍有不同的用途。因此，应当适当地设计程序，从而将逻辑上分离的功能划分到不同的组件，这些组件可以在不同的程序中重用。

相对于真实世界中可互换的离散部件，上述编程策略正是对此设计原则的建模。例如，可以把一台旧车上的轮胎卸下来，换到另一种新车上使用。轮胎就是可以分离的组件，没有与车的其他部件固定在一起。卸轮胎时不用把发动机也一并拆下来！

在程序设计中，可以在宏观的子系统级和微观的类层次体系两方面应用这种逻辑划分策略。

把程序划分为逻辑子系统

要将子系统设计为可以独立重用的离散组件。例如，如果你在设计一个网络游戏，就应该把网络部分和图形用户界面部分放在不同的子系统中。这样一来，就可以分别重用这两个组件而不必在使用一个组件时牵扯上另一个组件。例如，你可能想编写一个非网络游戏，在这种情况下，可以重用图形用户界面子系统，而不需要网络部分。类似地，可以设计一个端到端（对等）文件共享程序，此时可以重用网络子系统，而不需要图形用户界面功能。

要确保每个子系统都遵循抽象原则，将子系统的接口与其底层实现清晰地分离。可以把每个子系统认为是一个小型库，必须为之提供一个一致而且易于使用的接口。即使只有你一个人使用这些小型的库，如果接口与实现设计得当，能将逻辑上不同的功能加以分离，也能获得很大好处。

使用类层次体系来分离逻辑概念

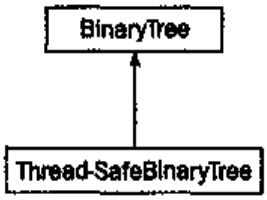

图 5-1

除了将程序划分为逻辑子系统，还要避免在类层次上将无关的概念混杂在一起。例如，假设你想为一个多线程程序编写一个平衡二叉树结构。你认为这个平衡二叉树数据结构一次应当只允许一个线程来存取或修改结构，因此，你在数据结构本身当中结合了加锁机制。不过，如果想在另外一些程序中使用这个二叉树的话，而这些程序刚好只是单线程程序，又会怎么样呢？在这种情况下，加锁只会浪费时间，而且要求程序必须与库链接，对于单线程程序，这本来是可以避免的。而且更糟糕的是，树结构可能在另一个平台上不能编译，因为加锁代码可能不是跨平台的。对此的解决方案就是创建一个类层次体系（见第3章的介绍），其中有一个线程安全的二叉树作为通用二叉树的一个子类。这样一来，就可以在单线程程序中使用二叉树超类，而不会不必要地引入加锁的开销，或者在一个不同的平台上使用二叉树超类，而无需重新编写加锁代码。图5-1显示了这个类层次体系。

如果存在两个逻辑概念，如线程安全和二叉树，这种策略能很好地作用。如果有三个或更多概念，就会变得更为复杂。例如，假设你想同时提供一个 n 叉树和一个二叉树，它们可以是线程安全的，也可以不是。逻辑上讲，二叉树是 n 叉树的一个特例，因此应当是 n 叉树的一个子类。类似地，线程安全结构应当是非线程安全结构的子类。通过一个线性体系无法提供这些逻辑概念分离。一种可能的做法是让线程安全方面作为一个混合类，如图5-2中所示。

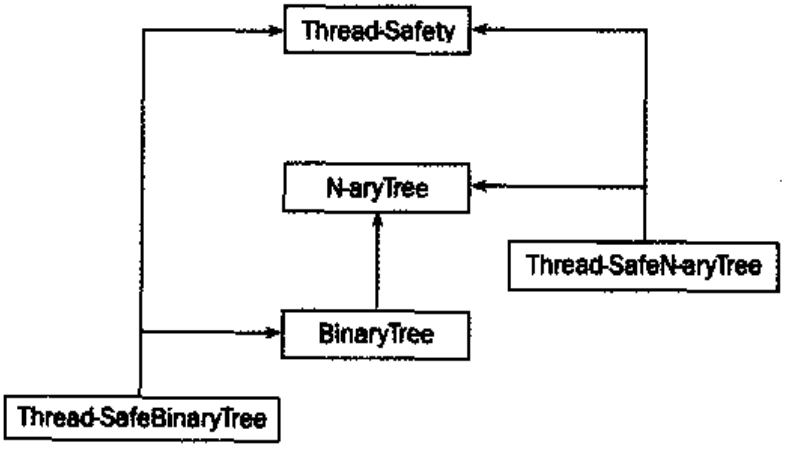

图 5-2

在这个层次体系中，尽管需要编写5个不同的类，不过由于能把功能清楚地分离，所以这是值得的。

还可以使用类层次体系将通用功能与更为特定的功能相分离。例如，假设你在设计一个操作系统，它支持用户级多线程。你可能想要编写一个包括多线程支持的进程类。不过，如果这些用户进程不想成为多线程的，怎么办呢？一个更好的设计是，创建通用进程类，并建立多线程进程作为它的子类。

使用聚集来分离逻辑概念

聚集（Aggregation）在第3章曾做过介绍，这是对*has-a*关系建立的模型，对象包含其他对象来完成其功能的某些方面。在继承不适用时，可以使用聚集来分离无关的功能，或者尽管相关但不同的功能。

还是看前面的操作系统例子，你可能希望将准备就绪进程存储在一个优先队列中，不要把优先队列结构与ReadyQueue类相集成，而应当编写一个单独的优先队列类。这样ReadyQueue类就可以包含并使用一个优先队列。按面向对象的术语来说，就是ReadyQueue有一个优先队列（has-a）。利用这种技术，优先队列可以更容易地在另一个程序中重用。

对通用数据结构和算法使用模板

只要可能，就应当为数据结构和算法使用一种通用设计，而不是将特定程序的特定方面编写到代码中。不要编写一个只存储“书”对象的平衡二叉树结构。要让它作为一个通用结构，可以存储任何类型的对象。如此一来，你不仅可以将其用于一个书店、一个音乐商店、一个操作系统，而且只要是需要平衡二叉树结构的地方都可以使用。这种策略正是第4章所讨论的标准模板库（STL）的基础。STL提供了可用于任何类型的通用数据结构和算法。

由STL可见，C++为这种通用编程模板提供了一个绝好的语言特性。如第2章和第4章所述，利用模板，就能编写用于任何类型的数据结构和算法。第11章提供了利用模板编写代码的详细内容，不过，这一节只讨论其中一些重要的设计方面。

为什么模板优于其他通用编程技术

模板并非惟一一种编写通用数据结构的机制。在C和C++中也可以存储void*指针而不是一个特定类型，并以这种方式来编写通用结构。客户只需将某个类型强制转换为一个void*，就可以使用这种结构来存储他们需要的任何东西。不过，这种方法存在一个主要问题，即它不是类型安全的，容器无法检查或保证所存储元素的类型。你可以将任何类型强制转换为一个void*，从而存储在此结构中，而且从这个数据结构中删除指针时，必须再把它们强制转换回你所认为的类型。由于在此不涉及任何检查，其结果可能是灾难性的。可以假想有这样一种情况，一个程序员在一个数据结构中存储了指向int的指针，并且首先将其强制转换为void*，但是另一个程序员可能认为这是Process对象的指针。第二个程序员会很自然地把void*指针强制转换为Process*指针，想把它们用作为Process*。不用多说，程序肯定不能像他们预想的那样工作。

还有一种方法，即为特定类编写数据结构。通过多态，该类的任何子类都可以存储在这个结构中。Java就把这种方法用到了极致，它指定每个类都直接或间接派生自Object类。Java容器存储Object，因此这些容器可以存储任何类型的对象。不过，这种方法不是真正类型安全的。从容器删除一个对象时，必须记住它实际上是什么，还要向下强制转换为适当的类型。

与此不同，模板如果使用得当则是类型安全的。模板的每个实例化只有一个类型。如果试图在同一个模板实例化中存储不同类型，程序就无法通过编译。

模板存在的问题

模板也不是十全十美的。首先，模板的语法很复杂，特别是以前从没有用过模板的人很可能摸不着头脑。其次，解析很困难，并非所有编译器都完全支持C++标准。另外，模板要求同构的数据结构，在一个结构中只能存储相同类型的对象。也就是说，如果你编写了一个模板化的平衡二叉树，可以创建一

个树对象来存储 Process 对象，再创建另一个树对象存储 int。但是不能在同一个树中同时存储 int 和 Process。这种限制是模板类型安全性的一个直接结果。尽管类型安全很重要，但是有些程序员认为这种同构需求是一个严重的限制。

模板存在的另一个问题是，它们会导致代码膨胀（code bloat）。在创建一个树对象存储 int 时，编译器实际上会“扩展”模板来生成代码，就好像只为 int 编写了一个树结构一样。类似地，如果创建一个树对象来存储 Process，编译器也会生成代码，就好像只为 Process 编写了一个树结构。如果为多种不同类型实例化了模板，最后就会得到超大的可执行文件，因为在此会对应所有类型生成不同的代码。

模板 vs 继承

程序员有时会发现很难确定究竟该使用模板还是继承。下面是一个技巧，可以帮助你做出决定。

如果想为不同类型提供相同的功能，就可以使用模板。例如，如果想编写一个通用的排序算法，从而可以用在任何类型上，就应使用模板。如果想创建一个可以存储任何类型的容器，也可以使用模板。关键在于，模板化的结构或算法会对所有类型同等对待。

如果想为相关类型提供不同的行为，则应使用继承。例如，如果想提供两个不同但相似的容器，如队列和优先队列，此时就可以使用继承。需要注意，继承和模板可以结合使用。你能编写一个模板化的队列存储任何类型，而且有一个模板化优先队列作为其子类。第 11 章将介绍模板语法的有关详细内容。

提供适当的检查和防护

一定要将程序设计得尽可能地安全，以便其他程序员使用。这个原则最重要的方面就是要在代码中完成错误检查。例如，如果随机数生成器需要一个非负整数作为种子，不要寄希望于用户一定会正确地传入一个非负的整数。要检查传入的值，如果是非法值则应拒绝此调用。

作为对照，可以考虑一个帮你准备收入税费申报表的会计。当你雇用一位会计时，会向他或她提供你全年的财务信息。会计会使用这个信息来填写国家税务局（Internal Revenue Service）的表格。不过，这个会计并不会盲目地在表格中填入你的信息，而是会确保这些信息是有意义的。例如，如果你拥有一座房子，但是忘记指出你所交的财产税，会计就会提醒你补充提供这个信息。类似地，如果你说付的贷款利息是 $12000，但总收入只不过 $15000，会计可能会委婉地提醒你是不是数字给错了（或者至少建议你找一个更住得起的房子）。

可以把会计看作是一个“程序”，输入就是你的财务信息，输出是收入税费申报表。不过，会计的作用不只是会填写表格。你之所以会请一位会计，还因为他或她能够提供检查和防护。在编程中也是如此，类似地，应当在实现中提供尽可能多的检查和防护。

有许多技术和语言特性可以帮助你在程序中引入检查和防护。首先，可以使用异常来通知客户代码出现了错误。第 15 章将详细介绍异常。其次，可以使用智能指针（见第 4 章的讨论）和其他安全的内存技术（见第 13 章的讨论）。

5.2.3 设计可用的接口

除了抽象和适当地建立代码结构，要完成重用设计，还需要重视与程序员交互的接口。如果把一个丑陋的实现隐藏在一个很漂亮的接口里，没有人会知道。不过，如果你提供了一个很不错的实现，但其外部的接口很糟糕，那么你的库不会好到哪去。

需要注意，程序中每个子系统和类都应当有好的接口，即使你本来不打算在多个程序中使用这些子系统和类也不例外。首先，你无法知道什么时候可能会得到重用。其次，即使对于第一次使用，好的接口也很重要，特别是当你在一个团队中参与开发，而其他程序员必须使用你设计和编写的代码，好接口则更为重要。

接口的主要用途就是使代码易于使用，不过有些接口技术还可以帮助你遵循一般性原则。

设计易于使用的接口

你的接口应当易于使用。这并不是说这些接口要很平常，而是说在满足功能的条件下要尽可能地简单和直观。如果使用你的库的人要通读数页的源代码才能使用一个简单的数据结构，这就很不好，或者，要得到所需的功能，他们必须对代码做大幅调整，这也是应该避免的。本节对于如何设计易于使用的接口提供了4个具体的策略。

开发直观的接口

计算机程序员用直观（intuitive）一词来表示这种接口很容易领会，而无须太多讲解。“直观”这个词和短语“显而易见”在含义上是类似的，后者表示不用太多解释或分析就很明显。从定义几乎可以了解到，使用直观的接口会很容易。

要开发直观的接口，最好的策略就是遵循标准做法和人们最熟悉的做法。当人们遇到一个接口，而这个接口与他们以前使用过的某个接口很像，就能更好地理解、更容易地采纳，而且一般不太会使用不当。

例如，假设你在开发一辆车的操纵机制。对此可以有多种可能，可以是一个操纵杆、两个左移或右移的按钮、一个滑动水平杆或者是一个常规的方向盘。你觉得哪一个最容易使用呢？觉得用哪一个接口的车最好卖呢？顾客一般对方向盘很熟悉，所以这两个问题的答案自然都是方向盘。即使你开发过另外一种机制可以提供更好的性能和安全性，但是卖这种产品的时候你肯定会遇到困难，更不用说教人们学习如何使用这种产品了，可以想见难度会有多大。到底是遵循标准接口模型，还是另辟蹊径呢？在这二者之间做出选择时，最好还是采用人们熟悉的接口。

创新固然重要，但是应该强调底层实现中的创建，而不是接口中的创建。例如，顾客非常青睐某些车模里混合式的油电双用发动机。这些车在这个方面就卖得很好，因为这些车的接口与使用其他标准发动机的车没有什么不同。

如果应用到C++，这种策略则说明你开发的接口应当遵循C++程序员熟悉的标准。例如，C++程序员希望类的构造函数和析构函数会分别初始化和清除对象。在设计类时，就应当遵循这个标准。如果你要求程序员自行调用initialize()和cleanup()方法来完成初始化和清除，而不是把这些功能放在构造函数和析构函数中，使用这个类的每一个人都会搞糊涂。因为你的类与其他C++类表现不同，程序员学习如何使用这个类就要花费更长的时间，而且忘记调用initialize()或cleanup()的可能性更大（以至于使用不正确）。

一定要从使用接口的人的角度来考虑接口。它们有意义吗？这正是你所期望的吗？

C++提供了一种称为操作符重载（operator overloading）的语言特性，这有助于你为对象开发直观的接口。利用操作符重载，可以编写类，从而使用标准操作符也可以处理这些类，就好像处理诸如int和double等内置类型一样。例如，可以编写一个Fraction类，从而完成加法、减法和流操作，如下所示：

```
Fraction f1(3,4), f2(1,2), sum, diff;
sum = f1 + f2;
diff = f1 - f2;
cout << f1 << " " << f2 << endl;
```

与之对照，如果使用方法调用来完成同样的行为，则如下所示：

```
Fraction f1(3,4), f2(1,2), sum, diff;
sum = f1.add(f2);
diff = f1.subtract(f2);
f1.print(cout);
cout << " ";
f2.print(cout);
cout << endl;
```

可以看到，利用操作符重载，可以为类提供直观的接口。不过，要注意不要滥用操作符重载。有可能会重载＋操作符，使之实现减法；而把－操作符重载为实现加法。这些实现就与直观背道而驰。一定要遵循这些操作符本身的一般含义。有关操作符重载的详细内容请参见第9章和第16章。

不要遗漏必要的功能

这个策略有两个方面。首先，要在接口中包括客户可能需要的所有行为。乍一看这可能很显然。还是对照着车来分析，你肯定不会造一辆没有速度表的车（否则驾驶员就无法了解他或她的速度是多少了!）。类似地，你也不会设计未提供访问机制的 Fraction 类（也就是说，不允许客户代码访问这个分数的实际值）。

不过，其他行为可能没有这么明显。这种策略要求你预计到代码的所有用途，即客户会把你的代码用到哪里。如果以某个特定的方式来考虑接口，可能会遗漏一些功能，而客户以另一种方式使用代码时会需要这些功能。例如，假设你想设计一个游戏棋盘类。可能只考虑了典型的游戏，如象棋和跳棋，因此决定支持一种每一点上最多只有一个棋子的棋盘。不过，如果后来你又决定编写一个巴加门15子棋游戏，这样棋盘上每个点允许有多个棋子，又该怎么办呢？由于当初完全杜绝了这种可能性，就无法将前面写的棋盘用作为巴加门棋盘了。

显然，要想预计到库的每一种可能的用途，这往往很难，甚至是不可能的。不要有压力，不要因为想设计完美的接口而力图考虑到将来所有可能的用途，只要能充分考虑，并且尽你所能就可以了。

这个策略的第二个方面是在实现中尽可能多地加入功能。如果在实现中已经知道某些信息，或者另外采用适当的设计就可以从实现中知道有关信息，那就不要让客户代码来指定这些信息。例如，如果你的 XML 解析器库需要一个临时文件来存储结果，不要让使用这个库的客户来指定该文件的路径。这些客户并不关心你用的是什么文件，应当另找办法来确定一个合适的文件路径。

另外，不应要求库用户完成不必要的工作来得到合并的结果。如果随机数库使用了一个随机数算法，而它要单独计算随机数的低位和高位，就应该将其低位数和高位数先行合并，然后再提供给用户。

> 如果你能为用户完成某些任务，就不要让库的用户自己来完成。

提供简洁的接口

为了避免在接口中漏掉功能，有些程序员会走入另一个极端，他们会在接口中包括可能想到的每一个功能，并认为这样一来，使用接口的程序员肯定能找到完成任务的办法。遗憾的是，这种接口可能太过杂乱，以至于程序员根本无法了解该怎么做!

不要在接口中包括不必要的功能，要让接口保持清楚而简单。最初看来，这个原则与上一条策略（不要遗漏必要的功能）好像是矛盾的。为了避免遗漏功能，可以在接口中包括每一个可能想到的功能，尽管这也是一个策略，但是这不是一个健全的策略。正确的策略是：应该包括必要的功能，而去掉无用或无意义的接口。

可以再来考虑汽车的例子。开车时只会与几个组件打交道：方向盘、刹车和油门、变速器、后视镜和仪表盘上的其他一些仪表。下面假设一个汽车的仪表盘就像飞机的仪表盘一样，有数百个仪表、操纵杆、监控器和按钮。这就很没用。开车可比开飞机容易多了，所以接口可以简单得多，你不用查看你的高度，不用与控制塔通信，也不用控制飞机上的许多组件，如机翼、引擎和着陆装置等等。

另外，从库开发的角度看，小一些的库也更易于维护。如果你想让每一个人都满意，这往往会导致出错的可能性更大，如果实现太过复杂，所有一切都交织在一起，那么即使只有一个错误，也可能导致库无法使用。

遗憾的是，要设计简洁的接口，这种想法从理论上看很好，但是要付诸实践却极为困难。这个原则

往往存在主观性，你要确定哪些是必要的，哪些是不必要的。当然，如果你的选择有错，客户肯定会指出来。以下是务必记住的一些技巧。

- 消除重复的接口。如果一个方法以英尺为单位返回一个结果，另一个方法以米为单位返回一个结果，可以把这二者合并为一个方法，返回一个对象，从而既可以按英尺又可以按米提供结果。
- 确定采用何种方法提供所需的功能最为简单。去掉不必要的参数和方法，并适当地将多个方法合并为一个方法。例如，如果有一个方法可设置用户指定的初始化参数，就可以把它与一个库初始化例程合并。
- 适当地限制库的使用。要想满足每个人的要求，这是不可能的。不可避免，肯定有人会以某种你不希望的方式来使用库。例如，如果你提供了一个库来完成XML解析，有人可能会用它来解析SGML。这并非你的本意，所以对此不必有压力，不需要考虑还对SGML解析提供支持。

提供文档和注释

不论你的接口多么容易使用，多么直观，都应当提供相关文档来说明这个接口如何使用。除非你告诉了其他程序员该如何使用你写的库，否则不要寄希望他们总会正确地使用这个库。可以把你的库或代码认为是其他程序员消费的产品。你购买的所有有形产品（如DVD播放器）都会提供一组说明来解释其接口、功能、限制和有关问题。即使是椅子之类简单的产品通常也提供了说明，来指出如何正确地使用，尽管这个说明可能相当简单，也许只是“可以坐在本产品上。如果用其他方式使用本产品，可能带来人身伤亡”。但这确实是对产品功能的一个说明。类似地，你的产品也应该提供文档来解释如何正确地使用。

为接口提供文档有两种方法：在接口本身提供注释，以及外部文档。应当这两方面都做到才好。大多数公共API只提供了外部文档，许多标准UNIX和Windows头文件中很少有注释。在UNIX中，通常以在线手册的形式提供文档，这些在线手册称为手册页（man page）。在Windows中，集成开发环境会随附文档。

尽管大多数API和库在接口本身中都没有（或很少有）注释，但我们确实认为这种文档才最为重要。不应该发布一个只包含代码而没有任何注释的“裸”头文件。即使注释与外部文档中的说明是重复的，与查看只有代码的头文件相比，如果这个头文件中有友好的注释，就不会那么让人心生畏惧了。即使是最好的程序员，还是希望经常能看到文字性的说明。

有些程序员会使用一些工具从注释自动创建文档。这些工具会解析带有特殊关键字和格式的注释来生成文档，通常最终的文档采用超文本标记语言（Hypertext Markup Language，HTML）格式。Java程序设计语言提供了JavaDoc工具，使得这个技术进一步得到推广，不过对于C++也有许多类似的可用工具。第7章将更为详细地讨论这个技术。

在提供注释、外部文档或者二者都提供时，文档应当描述库的行为，而不是实现。行为包括输入、输出、错误条件和处理、本来用途和性能保证。例如，对于一个生成随机数的调用，描述此调用的文档应当指出它无需任何参数，返回一个指定范围内的整数，如果无法分配内存则抛出一个“内存不够”异常。这种文档不应解释具体生成随机数的线性算法细节。这个接口的客户并不关心算法，只要生成的数看上去是随机的就行！

在接口注释中提供太多实现细节可能是接口开发中最常见的一个错误。我们见过许多这样的例子，本来接口和实现得到了很好的分离，但却因为接口中的注释搞砸了，这些注释不应提供给客户，而应提供给库的维护人员才更合适。

> 公共的文档应当指定行为，而不是底层实现。

当然对内部实现建立文档是必要的，但不要把它作为接口的一部分公开。对于如何在代码中适当地使用注释，第7章提供了更详细的说明。

设计通用接口

接口应当足够的通用，以便用于各种不同的任务。如果在一个原本打算通用的接口中写入了某个应用的具体细节，它就无法用于任何其他用途了。以下是一些要记住的原则。

提供多种方法来完成同一功能

为了满足所有“顾客”的要求，有时提供多种方法来完成同一功能会很有帮助。不过，要明智地使用这个技术，因为过分使用很容易导致杂乱的接口。

再来考虑车的例子。如今大多数新车都提供了不用钥匙的远程开锁系统，只用按下遥控钥匙上的一个按钮就可以开锁。不过，这些车还是会提供一个标准的钥匙，可以用这个钥匙真正地锁车。尽管这两种方法是冗余的，但大多数顾客还是很高兴有这样两种选择。

在程序接口设计中，有时也有类似的情况。例如，假设某个方法要取一个字符串作为参数。可能希望提供两个接口：一个取C++ string对象，另一个取C风格的字符指针。尽管这二者之间可以转换，但是不同的程序员可能倾向于使用不同类型的字符串，所以同时提供这两种方法很有帮助。

需要注意，这种策略可以认为是接口设计中“简洁”原则的一个例外。在很多情况下这种例外是合适的，不过还是要更多地遵循“简洁”原则。

提供定制能力

为了提高接口的灵活性，需要提供定制能力。在没有定制能力的时候，人们对此往往有迫切的需要。例如，本书作者之一最近购买了一辆带防盗器的新车。如果用远程无钥开锁系统开锁，警报会自动解除。遗憾的是，如果开锁后30秒内车门没有打开，警报又会启动。在往车的后备箱放东西或拿东西时，有时这个特性很是烦人。如果只是要拿后备箱里的东西，打开车门会很不方便。不过，如果没有解除警报，关后备箱盖时又会让警报哗哗做响。这个问题最恼人的地方是，没有办法永久地解除防盗器。车的设计者肯定认为每个人都希望他们的防盗器有相同的功能，因此没有提供可以定制的机制。

定制能力可能很简单，只是允许客户打开或关掉错误日志记录。定制能力的基本前提是，可以为每个客户提供同样的基本功能，但是允许客户稍做调整。

通过函数指针和模板参数，可以提供更大的定制能力。例如，可以让客户设置自己的错误处理例程。这种技术就是装饰器模式（见第26章的介绍）的一个应用。

STL将这种定制能力策略用到极致，允许客户为容器指定自己的内存分配器。如果想使用这个特性，就必须编写一个遵循STL原则的内存分配器对象，并满足所需的接口。STL中的每个容器都取一个分配器作为它的一个模板参数。第23章将介绍有关的详细内容。

5.2.4 协调一般性和易用性

一方面想要易于使用，另一方面又想做到一般性，这两个目标有时好像存在着冲突。通常，引入一般性会增加接口的复杂性。例如，假设在一个地图程序中需要一个图结构来存储城市。为了做到一般性，可能会使用模板来编写一个适用于任何类型的通用地图结构，而不只是存储城市。这样一来，如果想在下一个程序中编写一个网络仿真器，就可以采用同一个图结构来存储网络中的路由器。遗憾的是，使用模板的话，接口可能不太漂亮，而且较难使用，特别是当客户不熟悉模板时更是麻烦。

不过，一般性和易用性并不是完全互斥的。尽管在某些情况下一般性的增加会降低易用性，但是设计出既一般又易于使用的接口并非没有可能。可以遵循以下两个原则。

提供多个接口

为了减少接口的复杂性，但仍然提供足够的功能，可以提供两个单独的接口。例如，可以编写一个通用的

网络库，其中包括两个单独的部分。其中一部分提供对游戏有用的网络接口，另一部分则提供对超文本传输协议（Hypertext Transport Protocol，HTTP）Web浏览协议有用的网络接口。

STL对其string类就采用了这个方法。正如第4章所提到的，string类实际上就是basic_string模板的一个char实例化。可以把类认为是一个接口，它隐藏了整个basic_string模板的复杂性。

优化常用功能

提供一个通用接口时，有些功能可能比其他一些功能用得多。应当让常用的功能易于使用，而仍然提供更高级的功能以便选择。再来看地图程序，你可能希望为地图客户提供一个选项，允许用户使用不同的语言指定城市的名字。由于英语如此普及，因此可以将英语作为默认语言，但是还提供一个额外的选项来改变语言。这样一来，大多数客户都不必担心语言的设置问题（也就是说，可以采用默认的英语），但是如果客户确实想采用其他语言，也可以达到目的。第4章中曾谈到要优化最常执行的部分代码，上述策略与第4章讨论的这个性能原则很接近。通过重点强调对设计中这些最常用部分进行优化，就能让大多数人得到最大好处。

5.3　小结

通过阅读本章，你学习了为什么要编写可重用的代码，以及如何编写可重用的代码。在此不仅了解了重用方法论（可以总结为“编写一次，经常使用”），并且知道了可重用代码应当既具有一般性，而且要易于使用。你还会发现，设计可重用代码有三个要求：使用抽象、适当地建立代码结构，而且要设计好的接口。

关于如何建立代码结构，本章给出了三个具体的技巧：要避免将无关或逻辑上分离的概念混在一起、使用模板建立通用数据结构和算法、提供适当的检查和防护。

本章还提供了6个设计接口的策略：开发直观的接口、不要遗漏必要的功能、提供简洁的接口、提供文档和注释、提供多种方法来完成同一功能、提供定制能力。最后对如何协调两个经常冲突的需求（一般性和易用性）给出了两点提示：提供多个接口、优化常用功能。

到本章为止，从第2章开始介绍的设计主题就结束了。第6章将讨论软件工程方法论，并以此结束本书第一部分有关设计的内容。第7章～第11章将深入到软件工程过程的实现阶段，并介绍具体的C++代码编写。