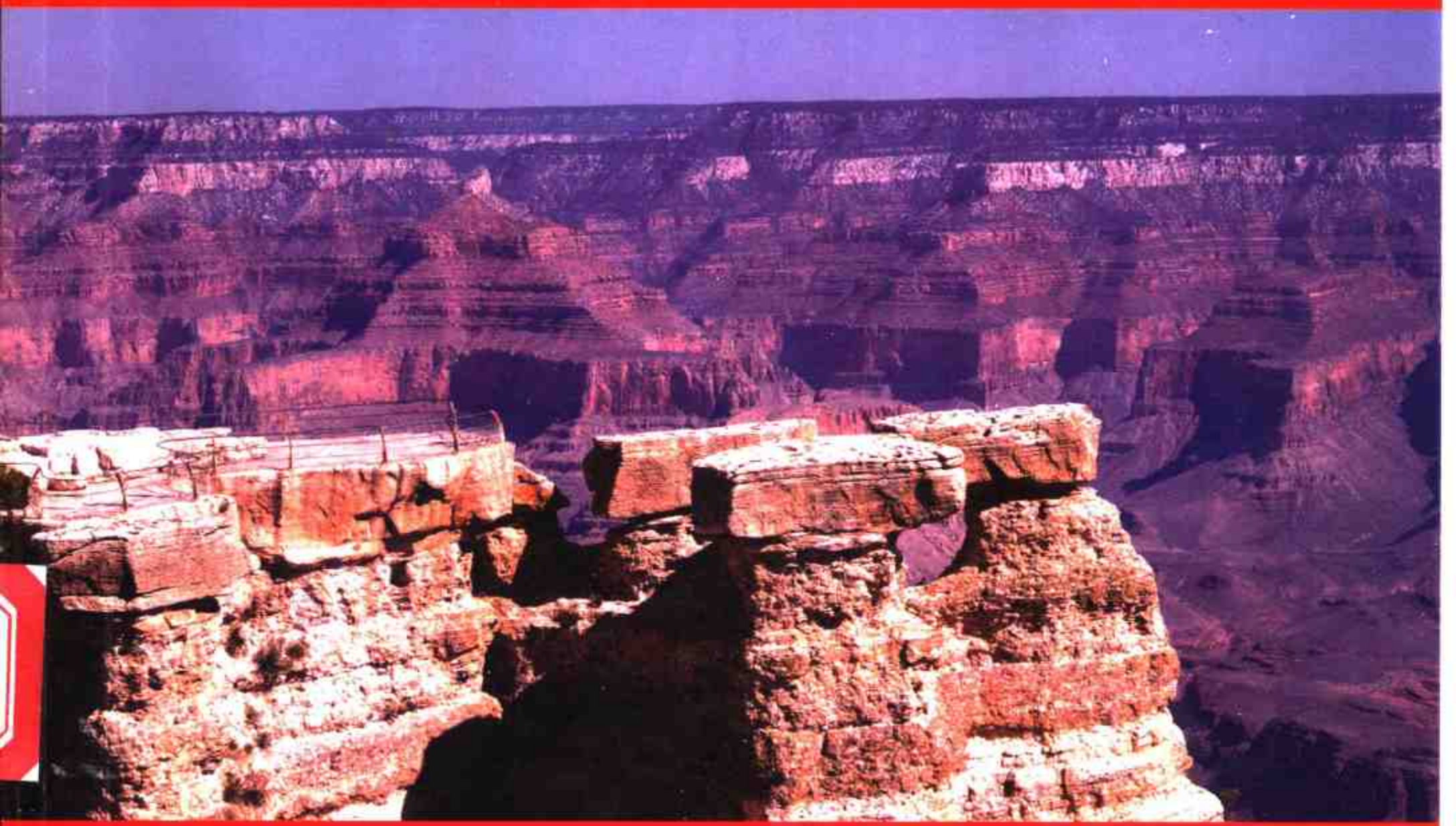
深入 C++ 系列

Large-Scale C++ Software Design

大规模 C++ 程序设计

[美] John Lakos 著
李师贤 明仲 曾新红 刘显明 等 译



Addison Wesley

中国电力出版社
www.infopower.com.cn

Large-Scale C++ Software Design

大规模 C++ 程序设计

要想用 C++ 成功开发大规模的软件系统，不仅需要很好地理解大多数关于 C++ 程序设计的书中所覆盖的逻辑设计问题，而且需要掌握物理设计的概念。这些概念与开发技术联系紧密，甚至很熟练的软件开发人员对其所涉及的内容也可能经验很少或者没有经验。

这是一本为所有从事软件开发工作（例如数据库、操作系统、编译程序及框架）的 C++ 软件专业人员而写的权威著作。它是第一本实际演示如何开发大型 C++ 系统的书，并且是一本少有的面向对象设计的书，尤其侧重于 C++ 编程语言的实践方面。

在本书中，Lakos 介绍了将大型系统分解成较小且较好管理的组件层次结构（不是继承）的过程。这种具有非循环物理依赖的系统的维护、测试和重用从根本上比相互紧密依赖的系统更容易且更经济。此外，本书还说明了遵从好的物理设计和逻辑设计规则的动机。Lakos 给读者提供了一系列用来消除循环依赖、编译时依赖和连接时（物理）依赖的特殊技术。然后，他将这些概念从大型系统扩展到了超大型系统。本书最后讨论了一种容易理解的、适用于单个组件的自顶向下设计方法。附录包含以下内容：一种用来在最小化物理依赖时避免胖接口的有价值的设计模式“协议层次结构”，实现一个兼容 ANSI C 的 C++ 过程接口的细节，以及一整套用于获取和分析物理依赖的类似于 UNIX 工具的完整规范。另外，实用的设计规则、指导方针和原则也收集在附录中。

John Lakos 在 Mentor Graphics 公司工作。该公司编写的大规模 C++ 程序比大多数其他公司要多，并且是首先尝试真正的大规模 C++ 项目的公司之一。Lakos 从 1987 年起就一直使用 C++ 进行专业编程，并于 1990 年在哥伦比亚大学开设了面向对象编程方面的研究生课程。

ISBN 7-5083-1504-9

9 787508 315041 >

责任编辑 / 闫　宏
封面设计 / 王红柳

ISBN 7-5083-1504-9
定价：72.00 元

深入 C++ 系列

深圳大学出版基金资助

Large-Scale C++ Software Design

大规模 C++ 程序设计

[美] John Lakos 著
李师贤 明仲 曾新红 刘显明 等 译

中国电力出版社

Large-Scale C++ Software Design(ISBN 0-201-63362-0)
John Lakos.

图书在版编目（CIP）数据

大规模 C++程序设计 /（美）勒科著；李师贤等译. —北京：中国电力出版社，2003
（深入 C++系列）
ISBN 7-5083-1504-9
Ⅰ.人… Ⅱ.①勒…②李… Ⅲ.C 语言—程序设计 Ⅳ.TP312
中国版本图书馆 CIP 数据核字（2003）第 033472 号

责任编辑：闫宏
丛 书 名：深入C++系列
书　　名：大规模C++程序设计
编　　著：（美）John Lokes
翻　　译：李师贤等
出版发行：中国电力出版社
地址：北京市三里河路6号 邮政编码：100044
电话：（010）88515918 传真：（010）88423191
印　　刷：北京地矿印刷厂
开　　本：787×1092 1/16　　**印　　张**：40　　**字　　数**：750千字
版　　次：2003年9月北京第一版
印　　次：2003年9月第一次印刷
定　　价：72.00 元

前　言

作为 Mentor Graphics 公司 IC 分部的成员，我有幸与许多富有才智的软件工程师一起工作，共同开发了一些非常大型的系统。

早在 1985 年，Mentor Graphics 公司就是最早用 C++开发实际的大型项目的公司之一。那时，无人知道如何做，也没有人预料到项目会出现这样的问题——费用失控、偏离计划、可执行代码庞大、性能低劣以及令人难以置信的昂贵的开发周期，这是不成熟的开发方法不可避免的结果。

许多有价值的经验都是经过痛苦的历程获得的。没有什么书可以帮助指导这种设计过程，也从未有人在这样的规模上尝试使用面向对象设计。

十年后，由于积累了大量有价值的经验，Mentor Graphics 公司已用 C++开发了数个大型软件系统，同时也为其他人在做同样的工作时不用再付出高昂代价开辟了道路。

在我 13 年的 C 语言（后来转为 C++）计算机辅助设计（CAD）软件开发生涯中，我多次看到了这样的情况：提前计划好总能生产出高质量、易维护的软件产品。在 Mentor Graphics 公司，我一直强调要从一开始就确保质量，要把质量作为设计过程中一个必不可少的部分。

1990 年我在 Columbia 大学开设了一门名为“面向对象设计与编程”的研究生课程。从 1991 年起，作为这门课程的老师，我有机会将我们在 Mentor Graphics 公司获得的许多经验传授给学生，这些经验是我们在从事工业化软件的开发过程中取得的。来自数百个研究生和专业程序员的提问和反馈信息，帮助我明确了很多重要的概念。本书是这些经验的总结。据我所知，这是第一本指导开发大型 C++项目的书，也是第一本针对大型 C++项目中出现的质量问题的书。我希望这些信息在读者的工作中有用，就像在我的工作中所起的作用一样。

读者对象

本书是专为有经验的 C++软件开发者、系统设计师、前摄的软件质量保证人员等编写的。本书尤其适合于那些从事大型软件开发工作（如数据库、操作系统、编译程序和框架）的人员阅读。

用 C++开发一个大型的软件系统，需要精通逻辑设计问题，在大多数有关 C++程序设计

的书中都包括了这些问题。若要进行有效的设计，还要求掌握物理设计概念，这些概念虽然与开发的技术方面紧密联系，但是其中有些方面即便是软件开发专家也可能经验很少或者没有经验。

本书提出的多数建议也适用于小型项目。我们开发项目时通常是先从小型项目开始，然后才开发更大更具挑战性的项目。一个特定项目的范围经常会扩展，因而开始时是小项目，后来就变成大项目了。但是，在大型项目中忽略好的策略所造成的直接后果，比在较小型项目中要严重得多。

本书将高层设计概念与特定C++编程细节结合起来，以满足下面两个需求：

（1）一本面向对象设计的书，尤其侧重于C++编程语言的实践方面。

（2）一本C++程序设计的书，描述如何使用C++编程语言来开发非常大型的系统。

请别弄错，这是一本高级别的书。本书既不适用于从头开始学习C++语法，也不适合于用来学习未曾掌握的C++语言细节，这是一本教你如何使用C++的全部功能去开发超大型系统的书。

简言之，如果读者认为自己C++掌握得很好，但想更多地学习如何使用C++语言去有效地开发大型项目，那么这本书就是为你所写的。

正文中的例子

多数人通过例子来学习。通常，我们提供说明现实世界设计的例子，避免那些只说明某一点问题而在设计的其他方面暴露出错误的例子；也避免那些只说明语言的细节但没有其他意义的例子。

除非特别指出，本书正文的所有例子都表示“好的设计”。因此，前几章的例子与整本书所介绍的所有策略一致。这种方法的不足之处是：读者看到示例代码与自己以往所见的代码不一致，但不能准确地知道为什么会这样。作者认为，若能利用书中所有的例子作为参考，就能弥补这个不足。

对于这种策略有两个显著的例外：注释和包前缀（package prefix）。本书正文中的许多例子的注释，为节省篇幅简单地省略了。有些地方虽然有注释，但也是少而精的。但是，这是一处要求读者“按我说的做，而不是按我做的那样做”的地方——至少在本书中如此。笔者在实践中编写接口时会当即非常仔细地添加注释，而不是事后再加注释，这一点读者可以放心。

第二个例外是，书中前面几个例子中的包前缀的使用不一致。在大型项目环境中需要包前缀，但是开始时，包前缀不便使用，需要一段时间适应。笔者选择先不使用注册的包前缀，在第7章正式提出之后再使用，以便能专注于介绍其它重要的基础内容。

当举例说明设计中的功能时，为了文字的简洁，在例子中使用了内联函数。在正文中多次说明了这种情况。由于本书的大部分内容与如何组织的问题（如，什么时候内联）有关，

所以笔者的倾向是在例子中避免使用内联函数。如果一个函数被声明为“inline”，肯定有其正当的理由，而不只是为了表示方便。

用C++开发大型系统是一系列的工程方法的折衷，几乎没有绝对。用“从不”或“总是”这样的词语来陈述是很吸引人的，但这样的词语只能用来对内容进行简单的描述。对于那些我希望将来阅读本书的C++程序员来说，这样总括性的陈述会引起异议——事实确实如此。为了避免陷入这种局面，我会在申明什么是（几乎是）肯定正确的同时，为例外的情况提供脚释或线索。

目前，有多种流行文件扩展名，用于区别C++头文件和C++实现文件。例如：

头文件扩展名：　.h　.hxx　.H　.h++　.hh　.hpp

实现文件扩展名：.c　.cxx　.C　.c++　.cc　.cpp

在所有的例子中，我们都使用.h扩展名来标识C++头文件，使用.c扩展名来标识C++实现文件。在本书中，我们会频繁地称头文件为.h文件，称实现文件为.c文件。最后需要说明的是，本书正文中的所有示例都在SUN SPARC工作站的CFRONT 3.0（SUN版本）上编译过并校验过语法；同时也在HP700系列机的白备C++编译器上进行过上述测试。当然，出现的任何错误都只能由作者负全责。

阅读导航

本书包含很多内容。不是所有的读者都有着相同的知识背景。所以我在第1章中提供了一些基本的（但却是必备的）知识以铺平学习道路。专业的C++程序员可以略过这一部分，或者需要的话简单地参考一下。第2章包含了一些基本的软件设计规则，我希望每一位有经验的开发人员都能够很快地认可。

第0章：引言

概述大型C++软件的开发人员所面临的问题。

第1部分：基础知识

第1章：预备知识

回顾了基本语言信息、一般设计模式以及本书的文体惯例。

第2章：基本规则

介绍了在任何C++项目中都应该遵循的重要设计经验。

后面的内容分为两大部分。前一部分题为“物理设计概念”，介绍了一系列与大型系统物理结构相关的重要论题。这些章节中的内容（第3~7章）集中在编程方面，对许多读者来说将是全新的，并且只精选了适合于大型程序设计的内容。这部分的论述是“自底向上”的，每一章的叙述都承接了前一章的内容。

第2部分：物理设计概念

第3章：组件

介绍一个系统的基础物理构筑模块。

第 4 章：物理层次结构

阐述建立组件层次结构的重要性，这些组件在物理上没有循环依赖，便于测试、维护和重用。

第 5 章：层次化

减少连接时依赖的特殊技术。

第 6 章：绝缘

减少编译时依赖的特殊技术。

第 7 章：包

将上述技术扩展到更大型的系统。

后一部分题为“逻辑设计问题”，研究逻辑设计与物理设计相结合的传统课题。这些章节（第 8~10 章）叙述了如何将一个组件作为一个整体来设计，总结了大量有关合理接口设计的问题，并研究了在大型项目环境中的实现问题。

第 3 部分：逻辑设计问题

第 8 章：构建一个组件

全面设计组件需要考虑的重要问题的总括。

第 9 章：设计一个函数

详细探讨生成一个组件的功能接口的问题。

第 10 章：实现一个对象

针对在大型项目环境中实现对象的若干组织问题。

附录中的主题是整本书的参考。

正文页下注中给出的书目的有关信息可再参阅书后的参考文献。

感谢

如果没有我在 Mentor Graphics 公司的许多同事的共同努力，这本书是不可能出现的。他们对于公司的划时代建设和发展作出了贡献。

首先，我要感谢我的朋友、同事和大学同学 Franklin Klein 对本书所作出的贡献。他实际上审读了本书每一页手稿。Franklin 对许多概念提供了解释——对大多数软件开发人员来说，这些都是新的概念。Franklin 的智慧、学识、社交能力、领会有效交流的细微差别的能力都远远超过我。他对内容修订、叙述顺序和表达风格等方面进行了详细的评审。

在排版期间，数位有天分和献身精神的专业软件人员检查了本书的大部分内容。他们愿意花费宝贵的时间审阅本书，对此我感到很幸运。我要感谢 Brad Appleton、Rick Cohen、Mindy Garber、Matt Greenwood、Amy Katriel、Tom O'Rourke、Ann Sera、Charles Thayer 和 Chris Van Wyk。他们花费大量的精力帮助我尽可能提高本书的价值。我要特别感谢 Rick Eesley，因为

他提供了丰富的评论和实际的建议——尤其是他建议在每一章的结尾作一个小结。

许多专业软件开发人员和质量保证工程师审阅了个别的章节。我要感谢 Samir Agarwal、Jim Anderson、Dave Arnone、Robert Brazile、Tom Cargill、Joe Cicchiello、Brad Cox、Brian Dalio、Shawn Edwards、Gad Gruenstein、William Hopkins、Curt Horkey、Ajay Kamdar、Reid Madsen、Jason Ng、Pete Papamichael、Mahesh Ragavan、Vojislav Stojkovic、Clovis Tondo、Glenn Wikle、Steve Unger 以及 John Vlissides，他们在技术上作出了贡献。我也要感谢 Mentor Graphics 公司的 Lisa Cavaliere-Kaytes 和 Tom Matheson，他们为书中的一些图表提出了宝贵的建议。另外我还要感谢 Eugene Lakos 和 Laura Mengel 所作的贡献。

自从本书首次印刷以来，我要感谢以下读者帮助我排除了一些我要负全责的不可避免的错误：Jamal Khan、Dat Nguyen、Scott Meyers、David Schwartz、Markus Bannert、Sumit Kumar、David Thomas、Wayne Barlow、Brian Althaus、John Szakmeister 以及 Donovan Brown。

如果不是在哥伦比亚大学收到了一封推销信，提供给我一份免费的有关 Rob Murray 的著作的评论拷贝，这本书也许永远也不会出现。因为我只在春季学期才教课，所以在我寄回所附表格的时候，要求将那本书寄到 Mentor Graphics 而不是哥伦比亚。之后不久，我接到了 Pradeepa Siva（她是 Addison-Wesley Corporate & Professional Publishing Group 的）打来的电话，要确定我这个不寻常要求的原因。在使她相信了这个要求的合理性（也许有一些是没有理由的夸大）之后，她说："我想我的老板可能会和你谈谈。"几天之后，我见到了她的老板——出版商。我一直很欣赏该出版社制作的"专业计算机丛书"的卓越品质，正是这种声誉最终使我答应为那套丛书撰写本书。

我非常感谢 Addison-Wesley Corporate & Professional Publishing Group 的成员。出版商 Join Wait，耐心地教给我许多关于人和交流的见识，这些东西将使我终生受益。从阅读大量书籍、与很多专业软件技术人员进行讨论、到站在书店观察潜在读者的购买习惯等方面，John Wait 一直尽力把握行业的脉搏。

以 Marty Rabinowitz 为首的产品人员在各方面都是优秀的。尽管与其他出版商联系的学术界的作者们对我表示担心，但是我仍然高兴地对待 Marty 所提出的在表达作者思想时应做到技术上准确、容易使用以及美学上有魅力的提炼与加工。我特别要感谢 Frances Scanlon，为了排印这本书，她付出了艰苦不懈的努力。

Brian Kernighan，本系列的技术编辑，在风格和实质上都提供了有价值的贡献，并发现了一些印刷错误和没有被其他人发现的不一致性。他的知识的广度和深度，与简洁的写作风格结合起来，对本系列丛书的成功作出了很大的贡献。

最后，因为对其他书中基础逻辑概念和设计原则的引证，我还想感谢本系列中的其他作者。

译 者 序

C++是当前的主流技术之一。随着我国社会与经济信息化工作的发展和深入，计算机应用水平和层次正在逐步提高，应用软件系统日益复杂化和大型化，我国软件业将面对越来越多的大型软件开发问题。然而，目前国内这一方面的书籍极少，而直接基于 C++大型软件开发的资料尤其缺乏。针对这一情况，我们翻译了《大规模 C++软件设计》一书，以满足读者的需求。

在实施大型和超大型 C++软件工程时，会出现许多意想不到的严重问题。若方法不当，轻则费用失控、进度延迟、执行代码臃肿、低效，重则可能导致开发项目完全失败。作者在总结多年来从事 C++大型工程的经验基础上，提出了物理设计和逻辑设计的一些新概念和新理论，阐明了在从事大型和超大型 C++软件工程时应遵循的一系列物理设计和逻辑设计原则，讨论了设计具有易测试、易维护和可重用等特性的高质量大规模 C++软件产品的方法。在说明这些概念、理论、方法和原则时，既有定性描述又有定量分析，还有许多生动的实例，并几对如何设计复杂系统进行了示范。书中的概念、理论、方法和原则对于大型软件工程的开发具有极强的现实指导意义，有利于提高软件的质量，保证大型项目的成功。作者 John Lakos 曾在 Mentor Graphic 公司（最早用 C++成功开发大型软件的公司之一）长期承担用 C++开发 CAD 软件的工作，书中积聚了他十余年的实践功力。作者从 1990 年起在美国哥伦比亚大学开设名为“面向对象设计与编程”的研究生课程，使本书中提及的方法和实例经受了课堂实践检验，并得到了进一步的总结提炼和理论升华。

本书虽然主要面向有经验的 C++程序员、系统设计师、软件质量保证人员等中、高级程度的读者，但书中对于 C++面向对象程序设计的重要概念、设计模式与良好的编程风格的精彩再现，以及对于在任何 C++项目中都应该遵循的重要设计原则和指导方针的深刻揭示，无疑亦可以使初级程度的读者大受裨益。即使是针对大规模软件设计的一些理念和方法，亦无妨为中、小型软件开发所参考或借用。也许正是由于上述缘故，本书尽管定位于大规模软件设计这一高端问题，但并未因此而曲高和寡，其英文版问世至今，读者甚众，短短数年，重印已达九次之多。

参加本书翻译的有李师贤、明仲、曾新红、刘显明、王志刚、李皓、李智、梅晓勇、李

宏新、杜云梅、邱璟、孙恒、徐晶、舒忠梅、章远和熊春玲等。李师贤对全书进行了译校和审定。明仲初译了前言及偶数章节。曾新红初译了奇数章节。限于译校者水平所限，译文中的谬误之处在所难免，恳请读者批评指正。

译　者

深入 C++ 系列

侯捷译著

- 影响深远的经典著作
- 35 条专家经验
- 条条精彩

- 最具权威性的 STL 书籍
- 侯捷评语："另辟捷径"

经典著作　不容错过　循序渐进　可收宏效

- C++ 最佳入门书籍
- 快速掌握 C++ 的全新方法
- 单刀直入 C++ 核心部分

- C++ 最新教程
- C++ 最佳参考书
- 名著名译
- 相得益彰

- comp lang c++ 精华荟萃
- C++ 程序员进阶必备

- 斯坦福大学教授的呕心之作
- 与《Art of Computer Programming》齐名的算法巨著

- 经典权威的 C++ 进阶图书
- C++ 设计思想的精彩汇总
- 原汁原味的理论接触

- 著名网络专栏 Guru of the Week 的精华荟萃
- 47 条专家经验
- 条条精彩

目　录

第 2 部分 物理设计概念

第 3 部分 逻辑设计问题

0 引言

开发出好的 C++程序并非易事。当项目很大时，用 C++开发高可靠、易维护的软件则更加困难，并且将引出许多新概念。就像建筑师建造摩天大楼时不能使用从建造单个家庭房屋取得的经验一样，许多从 C++小型项目中学到的技术和方法也不能简单地用于大型项目的开发。

本书讨论的是如何设计非常大型的、高质量的软件系统，也是专门为有经验的软件开发者而编写的。本书可以指导他们构造出易维护、易测试的软件结构。本书不是一本有关程序设计的理论方法的书，而是引导开发者走向成功的完全的、实际的指导。它是精通 C++的程序员开发大型多节点系统的多年经验的总结。我们将示范如何设计复杂系统。这些复杂系统需要数百个程序员参加，有数千个类并且其源代码可能有数百万行。

本章将讨论在使用 C++开发大型项目时会遇到的一些问题，并为在前几章中我们必须做的基础工作提供环境知识。引言中用到的几个术语还没有定义，大多数术语应根据上下文理解。在后面的章节中，会给出这些术语的更精确的定义。真正的高潮将出现在第 5 章，我们将应用特殊的技术来减少 C++系统中的耦合（即相互依赖的程度）。

0.1 从 C 到 C++

在控制大型系统的复杂性方面，人们已充分认识了面向对象风范的潜在优点。在写作本书时，每 7 到 11 个月，C++程序员的数量就翻了一番[①]。在有经验的 C++程序员的手中，C++是人类发挥技能和工程才干的强有力的工具。然而，如果认为在大型项目中，只要使用 C++就能确保成功，那就完全错了。

① **stroustrup94**，7.1 节，163~164 页。

C++并不只是 C 的扩充：C++支持一种全新的风范。由于面向对象风范比对应的面向过程风范需要更多的努力和悟性，所以其名声并不好。C++比 C 更难掌握，而且有无数情形令程序员陷入困境。常常会出现这样的情况，即程序员会忽略一些严重的错误，等到他们发现了想要弥补这些错误，并使项目仍能满足进度要求时，却已经太晚了。即使相当小的失误，例如不加区分地使用虚函数或通过值来传递用户自定义类型，在完全正确的 C++程序中，也可能导致其运行速度比完成同样功能的 C 程序慢十倍。

许多程序员在开始接触 C++时，会经历这样的过程，在这个过程中，编程效率大大降低甚至陷入停顿，因为似乎要探索无数的设计方案。在此过程中，传统的程序设计员会深感不安，因为他们想要竭力掌握“面向对象”概念。

尽管对于最有经验的专业 C 程序员来说，C++的规模和复杂度在开始时都有点难以承受，但一个能干的 C 程序员要写出一个小的但不是微不足道的 C++程序，并不需要花太长的时间。然而，这种用 C++创建小型程序的未经训练的技术，来完成大型项目是完全不能胜任的。也就是说，C++技术的不成熟应用不适合大型项目，不谙此道所造成的后果甚多。

0.2 用 C++开发大型项目

就像 C 程序一样，一个很差的 C++程序可能会非常难以理解和维护。如果接口没有完全封装，则很难对其实现进行调整或加强。不良的封装会妨碍重用，也可能导致可测试性方面的优势完全消失。

与主流观点相反，从根本上说，最普通形式的面向对象程序要比对应的面向过程的程序更难测试和校验[1]。通过虚函数改变内部行为的能力可能使类不变式（class invariant）无效，而对于程序的正确性来说，类不变式是必要的。此外，一个面向对象系统的控制流路径的潜在数量可能十分巨大。

幸运的是，我们没有必要写这样霸道的（也是不好测试的）面向对象程序。可靠性可以通过限制只使用面向对象风范中的一个更容易测试的子集来获得。

当程序变得更大时，一种本质不同的力量会起作用。下面的小节会介绍一些我们可能会遇到的问题的具体例子。

0.2.1 循环依赖

作为一个软件专业人员，可能面临过这样一个局面：第一次看一个软件系统，似乎找不到一个合理的起点或者自身有完整意义的系统片段。不能理解或不能单独使用系统的任何

① **perry**，13~19 页。

部分就是一种循环依赖设计的征兆。C++对象有一种显著的互相混杂的倾向。图 0-1 描述了这种潜在而危险的紧密物理耦合形态。电路（circuit）是元件（elements）和电线（wires）的集合。因此，类 Circuit 知道 Element 和 Wire 的定义。Element 知道它所属的 Circuit，而且能分辨出是否与特定 Wire 相连。因此类 Element 也了解 Circuit 和 Wire。最后，一根电线可以连接到一个元件或一个电路的末端。为了完成这种连接，类 Wire 必须访问 Element 和 Circuit 的定义。

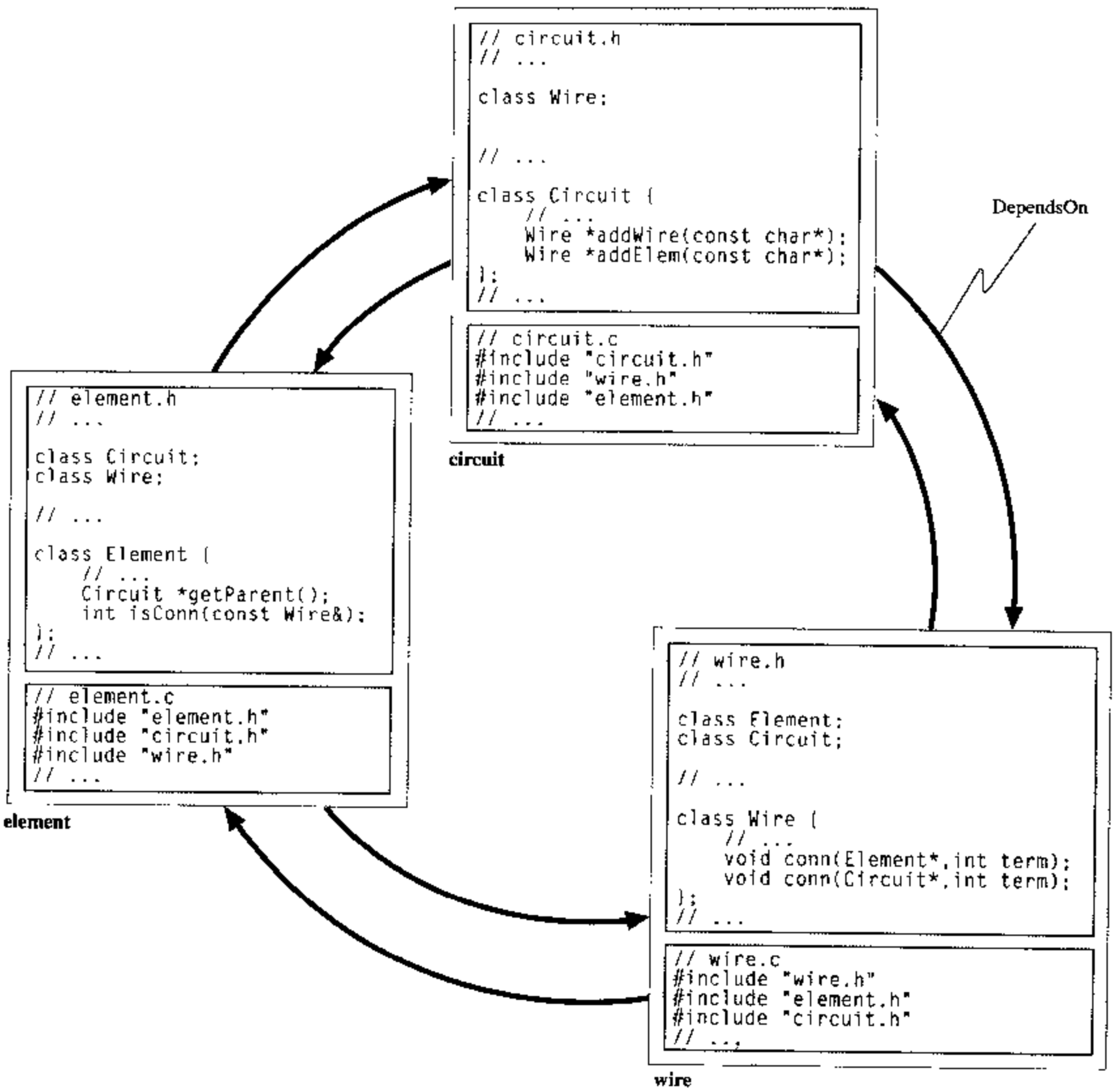


图 0-1　循环依赖组件

这三种对象类型的定义存在于单独的物理组件（编译单元）中，以提高模块化程度。虽然这些单个类型的实现被它们的接口完全封装，然而，每个组件的.c 文件都必须包含其他两个组件的头文件。这三个组件的最终依赖图是循环的。就是说，没有一个组件可以在没有其他两个组件的情况下单独使用甚至单独测试。

草率构建的大型系统会因为循环依赖而变得紧密耦合，从而强烈地抗拒分解。支持这样的系统可能是一场噩梦，通常不可能对其进行有效的模块测试。

一个适当的案例是一种电子设计数据库的早期版本。那时，它的作者没有意识到在物理设计中避免循环依赖的必要性，其结果是创建了一个相互依赖的文件集合，其中包含了数千个函数的数百个类，最终导致没有办法使用甚至测试，而只能把它当作一个单一的模块。这个系统的可靠性非常差，并且不适于扩展和维护，最终不得不抛弃它而重新编写。

比较起来，层次化物理设计（即没有循环相互依赖）相对更容易理解、测试和重用。

0.2.2 过度的连接时依赖

如果读者尝试过连接库中的少量功能程序，就会发现连接时间相对于所希望带来的利益来说已经不成比例地提高了，于是我们可能会尽量重用“重量级的组件”而非“轻量级的组件”。

使用对象的好处之一是在需要时很容易将遗漏的功能加进到对象中。面向对象风范的这种儿近诱惑的特性，使得许多谨慎的开发者将精练的、审慎考虑过的类变成巨大的“恐龙”，将数量巨大的代码合为一体，而绝大多数客户不会使用其中的大部分代码。图 0-2 描述了扩充一个简单类 String 的情形，即将简单类 String 扩充为满足所有客户需要的类。每一个客户加进一个新的特性，潜在地增加了实例体积、代码容量、运行时间以及物理依赖，从而增加了其他客户的开销。

C++程序常常比必需的大。如果不谨慎，一个 C++的可执行程序的大小可能要比同样功能的 C 程序大得多。将外部依赖忽略不计，雄心勃勃的类开发者已创建了一些复杂的类，这些类直接或间接地依赖数量巨大的代码。一个用精心制作的 String 类编写的“Hello World”程序竟产生了 1.2MB 的执行容量！

像 String 类这样的“超重”类型不仅会增加可执行程序的大小，而且会使连接过程过度缓慢和痛苦。如果连接 String（连同它所有的实现依赖）所需要的时间大于连接子系统所用的时间，那么人们就不太可能费力地去重用 String。

幸好现在已经有了避免各种形式的不必要的连接时依赖的技术。

```
// str.h
#ifndef INCLUDED_STR
#define INCLUDED_STR

class String {
    char *d_string_p;
    int d_length;
    int d_size;
    int d_count;
    // ...
    double d_creationTime;

  public:
    String();
    String(const String& s);
    String(const char *cp);
    String(const char c);
    // ...
    ~String();
    String &operator=(const String& s);
    String &operator+=(const String& s);
    // ...
    // (27 pages omitted!)
    // ...
    int isPalindrome() const;
    int isNameOfFamousActor() const;
};

// ...

#endif
```

```
// str.c
#include "str.h"
#include "sun.h"
#include "moon.h"
#include "stars.h"
// ...
// (lots of dependencies omitted)
// ...
#include "everyone.h"
#include "theirbrother.h"
String::String()
: d_string_p(0)
, d_length(0)
, d_size(0)
, d_count(0)
// ...
// ...
// ...
```

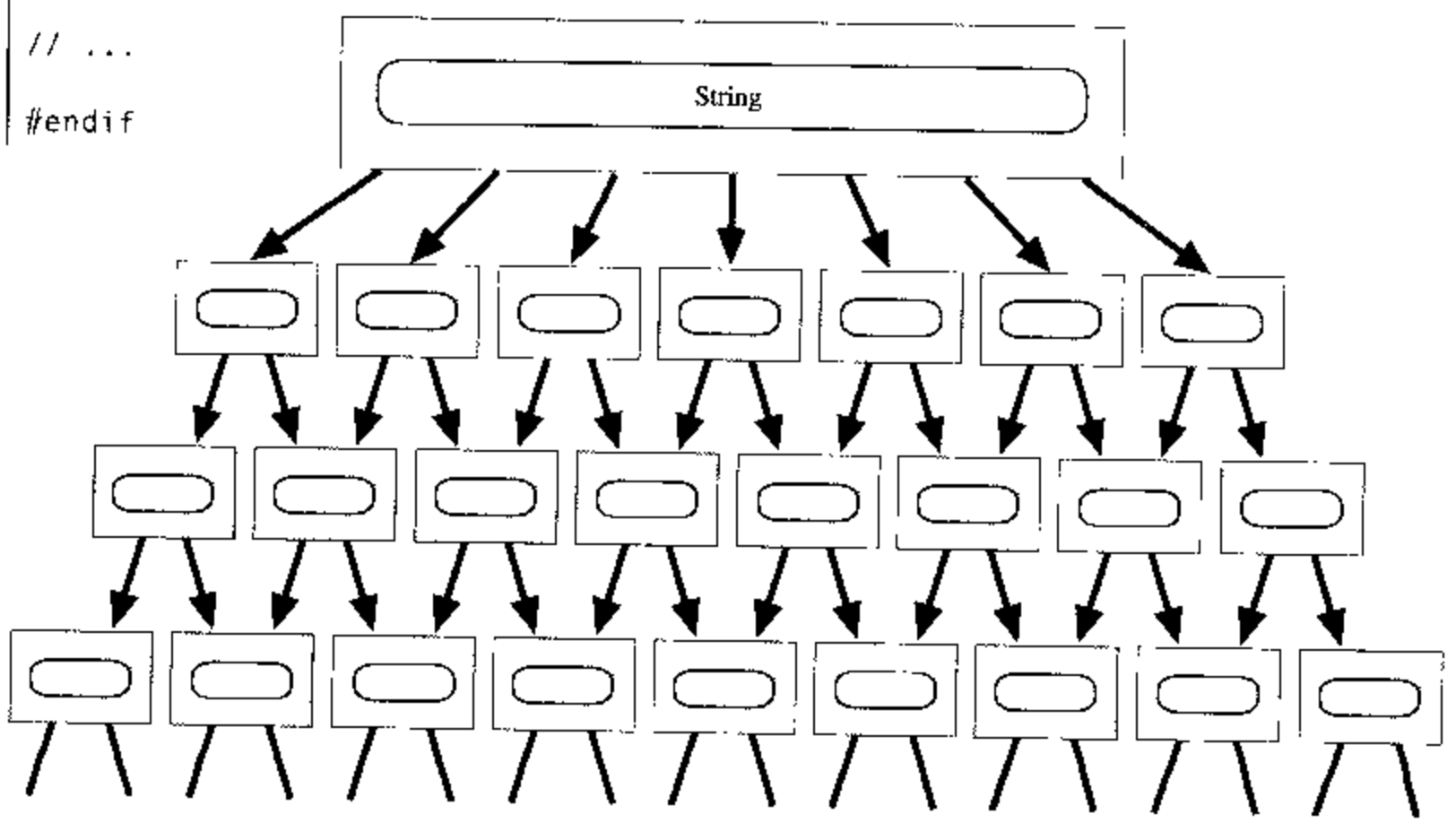


图 0-2　特大的、重量级的、不可重用的 String 类

0.2.3 过多的编译时依赖

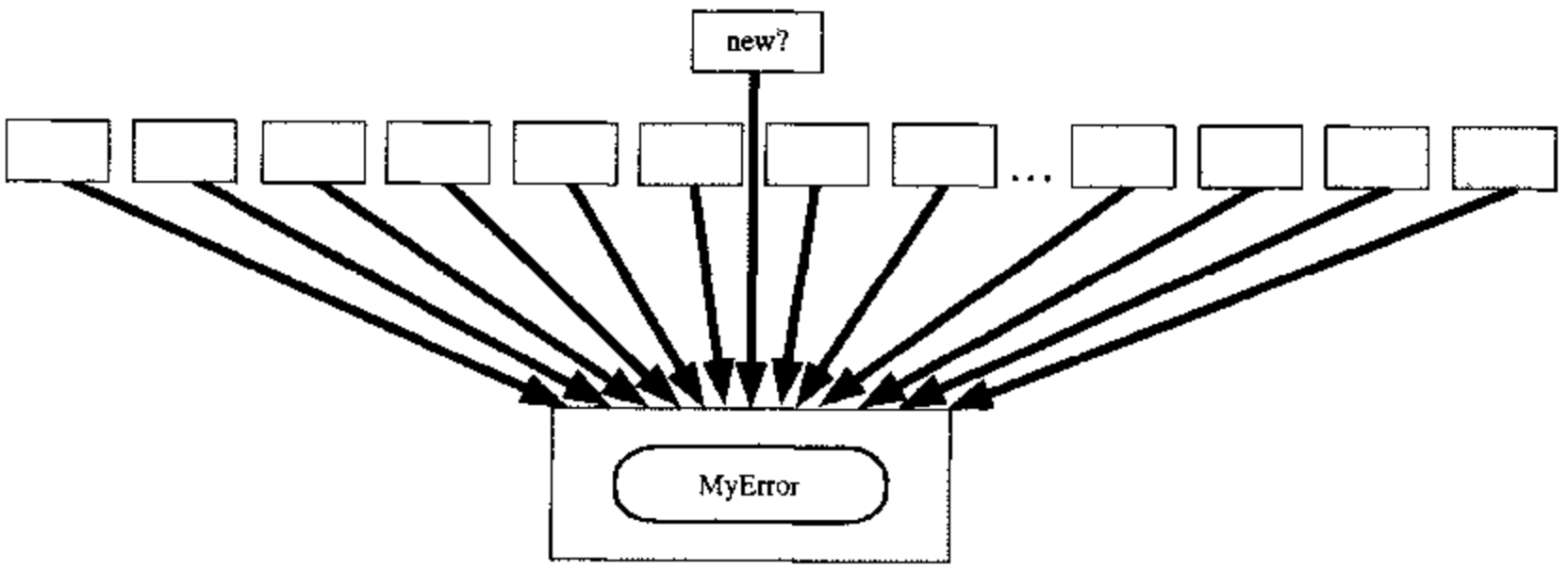
如果读者曾经尝试过用 C++开发一个多文件程序，那么就会知道改变一个头文件可能会潜在地引起多个编译单元要重新编译。在系统开发的最开始阶段，若一个修改会导致整个系统重新编译，则意味着增加毫无意义的负担。然而，当继续开发系统时，改变一个低级头文件的想法会变得越来越令人厌烦。不仅重新编译整个系统的时间会增加，甚至编译单个编译单元的时间也会增加。我们迟早会因为重编译的消耗太大而完全拒绝修改低级类。如果这些事情我们觉得熟悉，那么我们可能已经经历了过度的编译时依赖。

过度编译时耦合，对于小型项目毫无影响，对于较大型的项目却可能成为支配开发时间的重要因素。图 0-3 列举了一个过度编译时耦合的普通例子：开始时好像是一个好主意，随着系统容量的增长而变得糟糕。“myerror”组件定义了一个 MyError 结构，其中列举了所有可能的错误代码。每一个加进来的新组件都自然地包含这个头文件。但是，每一个新组件都可能有它自己的错误代码，这些代码并未标识在主列表中。

```
// myerror.h
#ifndef INCLUDED_MYERROR
#define INCLUDED_MYERROR

struct MyError {
    enum Codes {
        SUCCESS = 0,
        WARNING,
        ERROR,
        IO_ERROR,
        // ...
        READ_ERROR,
        WRITE_ERROR,
        // ...
        // ...
        BAD_STRING,
        BAD_FILENAME,
        // ...
        // ...
        CANNOT_CONNECT_TO_WORK_PHONE,
        CANNOT_CONNECT_TO_HOME_PHONE,
        // ...
        // ...
        MARTIANS_HAVE_LANDED,
        // ...
    };
};

#endif
```

图 0-3 潜伏的编译时耦合源代码

当组件的数量更多时，我们将自己的错误信息代码加入到清单中的愿望就没有那么强烈了。我们对重用已有的错误信息代码感兴趣，因为这些代码也许只要大概适合就可以，而不必修改 myerror.h 文件。最终，我们将放弃任何添加新的错误信息代码的想法，只是简单地返回 ERROR 或 WARNING 而不是修改 myerror.h 文件。到这个时候，我们就会陷入这样一个境地：所作的设计变得不可维护并且实际上已毫无用处了（参见下图）。



还有许多其他原因会引起不必要的编译时依赖。一个大型 C++程序会比同样的 C 程序有更多的头文件。一个文件包含不必要的头文件，是造成 C++中过多耦合的常见原因。例如，在图 0-4 中，在 simulator 头文件中本来没有必要包含对象的定义，只是因为 simulator 类的客户可能会发现这些定义有用处才这样做。这就迫使客户程序在编译时依赖于组件，不管这些组件实际用到与否。过多地包含（伪）指令，不仅会增加编译客户程序的开销，而且可能增加由于在较低的层次上改变了系统而必须重新编译客户程序的可能性。

如果忽略编译时依赖，有可能使得系统中的每一个编译单元包含系统中的几乎每一个头文件，从而使编译速度降低到跟爬行一样。最早真正大型的 C++项目（准确点说，数千人年的工作量）之一是 Mentor Graphics 公司开发的 CAD 框架产品。开发者最初不知道“编译时依赖”是如何阻碍他们的工作的。即使使用大型的网络工作站重新编译整个系统，也需要花一周的时间。产生这个问题的原因是组织的细节问题。图 0-4 中显示的 simulator 组件部分地说明了这个细节问题。“装饰（cosmetic）”技术可以缓解这个问题，但是真正的解决方案是消除不必要的编译时依赖。

就像连接时依赖一样，也有若干消除编译时依赖的特定技术。

0.2.4 全局名称空间

如果读者曾参加过多人合作的 C++项目，则一定知道软件集成常常会出现意想不到的事。特别是增加全局标识符很成问题。其中一个明显的危险是这些名字可能会冲突。结果造成分别开发的系统部件不经修改就无法集成。对于有数百个头文件的更大的项目来说，甚至很难

找到全局变量名的声明。

```
// simulator.h
#ifndef INCLUDED_SIMULATOR
#define INCLUDED_SIMULATOR
#include "cadtool.h"                    // required by "IsA" relationship
#include "myerror.h"                    // bad idea (see Section 6.9)
#include "circuitregistry.h"            // unnecessary compile-time dependency
#include "inputtable.h"                 // unnecessary compile-time dependency
#include "circuit.h"                    // required by "HasA" relationship
#include "rectangle.h"                  // unnecessary compile-time dependency
// ...
#include <iostream.h>                   // unnecessary compile-time dependency

class Simulator : public CadTool {      // mandatory compile-time dependency
    CircuitRegistry *d_circuitRegistry_p;
    InputTable& d_inputTable;
    Circuit d_currentCircuit;           // mandatory compile-time dependency
    // ...
  private:
    Simulator(const Simulator &);
    Simulator& operator=(const Simulator&);
  public:
    Simulator();
    ~Simulator();
    // ...
    MyError::Code readInput(istream& in, const char *name);
    MyError::Code writeOutput(ostream& out, const char *name);
    MyError::Code addCircuit(const Circuit& circuit);
    MyError::Code simulate(const char *outputName,
                           const char *inputName,
                           const char *circuitName);
    Rectangle window(const char *circuitName) const;
    // ...
};

#endif
```

图 0-4 不必要的编译时依赖

例如，我曾经使用过由数千个头文件组成的对象库。我曾想在文件作用域中把类型 TargetId 的定义寻找出来，因为 TargetId 看起来像是类定义（但实际不是）：

```
TargetId id;
```

我记得曾用“grep”[①]在数千个头文件中寻找该定义。但只收到了一个信息，大意是“有

① “grep”是 Unix 中的查找工具程序。

太多的文件”。最终，我在 shell 命令文件中使用嵌套的“grep”命令，基于首字母将头文件分拆，以便将问题简化为 26 个大小可控制的子问题。我最终找到了我要寻找的那个“类”，实际上它并不是类。它也不是 struct 或 union！正如图 0-5 说明的那样，搜索出来的结果是，类型 TargetId 实际上是在文件作用域内为一个 int 所作的 typedef 声明！

该 typedef 声明给全局名称空间引入了一个新的类型名。而没有任何迹象表明它是 int 类型，也没有暗示在哪里能找到其定义。

```
// upd_system.h
#ifndef INCLUDED_UPD_SYSTEM
#define INCLUDED_UPD_SYSTEM

typedef int TargetId;           // bad idea!
class upd_System {
    // ...
  public:
    // ...
};

#endif
```

图 0-5　不必要的全局名称空间污染

如果 typedef 的声明已嵌入到一个类的内部（如图 0-6 所建议的那样），那么可以通过该类来引用 typedef 声明（或者可以继承该声明），可直截了当地写为如下形式：

```
upd_System::TargetId id;
```

遵循上面所建议的简单规则，可以使冲突的可能降到最低，同时使得在大系统中寻找逻辑实体更加容易。

```
// upd_system.h
#ifndef INCLUDED_UPD_SYSTEM
#define INCLUDED_UPD_SYSTEM

class upd_System {
    // ...
  public:
    typedef int TargetId;       // much better!
    // ...
};

#endif
```

图 0-6　在类作用域内的 typedef 是容易找到的

0.2.5 逻辑设计和物理设计

大多数有关 C++的书仅阐述逻辑设计。逻辑设计是指那些属于类、运算符、函数等语言结构的设计。例如，一个特殊的类该不该有一个拷贝构造函数，是一个逻辑设计问题。决定一个特定操作（例如：operator==）应该是一个类的成员函数还是一个自由函数（也就是说，非成员函数），也是一个逻辑问题。甚至选择一个类的内部数据成员的类型，也属于逻辑设计。

C++支持完全的逻辑设计。例如，**继承**是面向对象必不可少的成分。另一个成分是**分层**（layering），包括用更原始的对象来构造类型，并经常直接嵌入到类的定义中。但是，在许多需要使用分层技术的时候，有很多人使用了继承。例如：一个 Telephone 不是一种 Receiver、Dial 或 Cord；而应由这些原始部件构成（或“分层于其上”）。

这种对事物的错误判断可能导致时间和空间上的低效率，并可能使结构的语义模糊，从而使整个系统更难以维护。有经验的 C++程序员知道何时使用或不使用某种特定的语言结构，这是他们成为很有用的人的原因之一。

逻辑设计并不涉及“将一个类定义放置在什么地方”的问题。从纯逻辑观点来看，所有在文件作用域中的定义都存在于单一的名称空间的同一层次上，这些定义之间没有界限。相对于类成员的定义来说，类在何处定义和是否支持自由运算符的问题与逻辑设计并不相干。重要的是这些逻辑实体如何组织在一起形成一个能工作的程序。由于将整个程序看作一个单元，所以没有单独的物理依赖的概念。程序作为一个整体只依赖于自己。

有若干关于逻辑设计的好书（参见参考文献）。但是，也有不少这些书未涉及的问题，这些问题只有当程序变得很大时才出现。这是因为与成功的大型系统设计相关的很多内容不属于逻辑设计的范畴，本书把它们作为物理设计。

物理设计涉及的问题包括与系统的物理实体有关的问题（如文件、目录和库等）以及组织问题（如物理实体之间的编译时依赖和连接时依赖等）。例如，使类 Telephone 的成员函数 ring()成为内联函数，将迫使 Telephone 的所有客户不仅要查看 ring()的声明，而且要查看它的定义，才能完成编译。无论 ring()是否声明为内联函数，其逻辑行为都是一样的。受影响的是 Telephone 与其客户之间的物理耦合的程度和特性，从而使得维护使用 Telephone 的程序的开销也要受到影响。

好的物理设计不仅仅是被动地决定系统的现有逻辑实体应如何划分。物理设计常蕴涵要支配逻辑设计的输出。在逻辑领域中类之间的关系，例如 IsA、HasA、Uses，在物理领域中压缩成组件之间的单个 DependsOn 关系。另外，一个好的物理设计的依赖关系是一种没有循环的图。所以，我们要避免在组件之间暗含循环依赖关系的逻辑设计。

同时满足逻辑设计和物理设计的约束条件，有时是一种挑战。实际上，有些逻辑设计可能必须重做甚至替换掉，以满足物理设计的质量标准。然而，在我的经验中，总是有充分地满足两个领域的解决方案，尽管开始时要费一些时间来发现它们。

对于小项目来说，一个文件目录就可满足要求，因而不必太重视物理设计。但是，对于

大型项目来说，一个好的物理设计的重要性就大大提升了。对于非常大型的项目来说，物理设计是项目成功的决定性因素。

0.3 重用

面向对象设计将易重用作为自己的优点。但是，正如其他风范一样，要获得好处，就要付出代价。重用意味着耦合，而其本身的耦合是我们不希望看到的。如果若干程序员企图使用同一组件，并且不要求功能上的修改，则重用是合理和正当的。

但是，考虑这样的情形，有若干客户分别编写不同的程序，每个人都想重用一个公共组件以达到不同的目的。如果另外一些独立客户也在积极寻求增强支持，他们会发现重用的结果彼此间不一致：某一客户程序的加强可能会毁坏其他人的程序。更糟糕的是，我们最终可能得到一个对谁都无用处的“超重”的类（如图 0-2 所示的 String 类）。

重用经常是好的方案。但是为了成功地重用一个组件或一个子系统，该组件或子系统不应该与一大块不需要的代码绑在一起。也就是说，只有那些与系统其他部分没有必然联系的部分才有可能重用。

不是所有的代码都能重用。试图实现过多的功能或者为实现对象进行完全彻底的错误检查，都会增加不必要的研制和维护开销，也会增加可执行代码的大小。

实现者的以下两方面的知识对大型项目有益：何时重用代码和何时使代码能重用。

0.4 质量

质量可以从多方面来衡量。**可靠性**是传统的质量定义（也就是说，“它有错误吗？”）。一个产品容易使用并且在大多数情况下运行正常，人们通常认为它是合适的了。但是，对于某些应用来说，如航空航天领域、医学领域以及财政金融领域，软件错误的代价会非常巨大。通常，软件不能只是通过测试这一种手段来达到可靠性要求。等到我们能测试软件的时候，软件的内在质量早已形成了。不是所有的软件都能够有效地测试。要想让软件能够有效测试，必须从一开始就本着这个目标来对它进行设计。

为了易于测试而设计，虽然很少成为小项目最关注的事情，却是成功构造大型和超大型系统的最重要的事情。易测试性与质量本身一样，不可能是事后产生的想法：它必须在编写第一行代码之前就预先考虑到。

除了易测试性外，保证质量还有许多其他方面需要考虑。例如，**功能性**是指一个产品是否能完成客户所期望的工作。有时一个产品可能被用户拒收，因为它没有具备客户所期望的足够的性能。更糟糕的是，一个产品可能得“零分”：如果客户希望购买一把螺丝刀，则世界上最好的锤子也不可能通过功能性测试。在开发开始之前就有一个符合市场需求的功能规格

说明，是保证软件具有合适功能的重要的第一步。但是，本书讨论的技术是如何设计和建立一个大型系统，而不是讨论设计什么样的大系统。

可用性是质量的另一种衡量标准。一些软件产品在正确性方面做得很好，在软件的有效使用方面却做得不够。如果一个软件产品对于具有代表性的预期客户来说，使用起来太复杂、太困难、太笨拙或者令人痛苦，那么，就不会有人去使用该软件。我们所说的用户通常是指系统的最终用户。但是，在一个大型的、设计成层次结构的系统中，组件的客户很可能是别的组件。来自客户（包括别的开发者）的早期反馈对于保证可用性是非常重要的。

可维护性是衡量支持一个系统工作相对开销的指标。支持工作包括追踪错误、移植到新平台以及扩充产品的性能以满足客户期望的（甚至没有预料到的）未来需要。用 C++（或者其他语言）编写的质量低劣的系统设计，使维护的开销非常昂贵，甚至扩充软件的开销也很大。大型的可维护设计决不是碰巧得到的，它们是遵循确保可维护性原则进行项目设计的结果。

性能是衡量产品的速度和大小的指标。尽管大家都知道面向对象设计在可扩展性和重用方面有优势，但是，面向对象风范的某些方面，如果使用不当，可能会使程序的运行速度更慢，并且需要更多的内存。如果我们的代码运行速度比竞争对手的产品慢，并且需要更多的内存，我们的产品就会销售不出去。例如，在一个文本编辑器中，将每一个字符都建模为一个对象，尽管在理论上很吸引人，但，如果我们希望达到最佳的时间/空间性能[①]，这就是一个不恰当的设计。试图用一个用户自己定义的类（例如 BigInt 类）去替代很常用的类型（例如 int），必然会降低性能。如果我们一开始没有强调性能目标，我们就可能采纳没有达到性能目标的系统结构或编程方法，除非我们重写整个系统。是否知道哪些地方可以不必太精细以及如何进行性能折衷，是区别软件工程师和单纯程序员的标准。

质量的各个方面对于产品的整体成功都很重要。质量的各方面良好性能的获得有一个共同点：必须在项目的一开始就考虑质量的各个方面。设计一旦完成，就无法再提高质量了。

0.4.1 质量检测

质量保证部（Quality Assurance，QA）一般是一个公司内负责测试产品质量已达到某种标准的部门。获得高质量软件的一大障碍是 QA 经常只在软件开发过程的后期，即已经造成了损害以后才参与。QA 通常不能影响软件产品的设计。QA 很少参与低层次的项目设计决策。QA 所执行的测试通常是在最终用户层次上的，并且依赖于开发者自己来进行任何低层次的回归测试。

在这种平淡无奇的过程模型中，由工程师开发出最初的软件，然后把它丢给 QA。这样的

① 参见 Flyweight pattern （**gamma**，第 4 章，195~206 页），那儿对这个特殊种类的性能问题有一个聪明的解决方案。

软件往往文档质量低劣，难以理解和测试，且不可靠。现在，QA 希望以某种方式一点点地提高软件质量。但是，如何提高呢？多次实践表明，在大型项目中，用上述模型来获得高质量软件的效率很低下。我们现在建议用另一个不同的模型。

0.4.2 质量保证

QA 必须成为开发过程中必不可少的部分。在这个过程模型中，开发者有责任确保质量。也就是说，软件已具备了高质量，测试工程师只是去验证而已。

在这个过程模型中，QA 与开发的界限很模糊，对 QA 人员和开发人员的技术资格要求本质上是一样的。某天，一个工程师写了一个接口，可由另一个工程师检查该接口的一致性、明确性和可用性。第二天，这两个人的角色可以互换。为真正有效，必须集体配合——在软件开发时，每个成员都帮助别的成员确保设计出高质量的软件。

提供一个完整的过程模型是一个规模宏大的任务，并且远远超过了本书的范围。但是，如果要获得高质量的软件，系统结构工程师和软件开发人员在整个开发过程中都必须把质量放在首位。

0.5 软件开发工具

大型项目可以受益于许多种工具，包括浏览器、增量式连接程序和代码生成器。即使简单的工具也可能很有用。附录 C 提供了一个简单的依赖性分析器的详细描述，我曾认为该工具在我的工作中没有用处。

一些工具可以帮助减轻低劣设计的“症状”。类浏览器可以帮助分析令人费解的设计并发现逻辑实体的定义，否则这些定义会被隐藏——淹没在一个大型项目中。带有增量式连接器和程序数据库的高级编程环境，有助于封装可以完成的设计，即使它含有低劣的物理设计。但是没有工具能解决根本性问题：即固有的设计质量问题。没有一个快捷和容易的方法可获得好的质量。工具本身不能解决由低劣设计引起的基础性问题。尽管工具可以推迟这些“症状”的发作，但是没有保证质量的专门工具，也没有工具能确保设计是完全按照规格说明进行的。从根本上说，是经验、智力和规程产生高质量产品。

0.6 小结

C++是一个整体而不是 C 的一个扩充。编译单元之间的循环“连接时依赖”会使程序难以理解、测试和重用。不需要的或过多的“编译时依赖”会增加编译开销并且不利于维护。当项目很大时，采用无组织的、无规程的或幼稚的 C++开发方法，最终一定会出现这些问题。

大多数有关 C++设计的书只阐述逻辑问题（如类、函数和继承），而忽略物理问题（如文

件、目录和依赖关系)。然而，在大型系统中，物理设计的质量将影响许多逻辑设计决策输出的正确性。

重用不是无开销的。重用蕴涵耦合，而耦合不是我们所希望的。没有保障的重用应该避免。

质量的衡量标准有多个方面：可靠性、功能性、可用性、可维护性以及性能。每一方面都会影响大型项目的成功或失败。

获得高质量是一项工程的责任：从一开始就应积极追求质量。质量不是在项目大体上完成之后可以加入的东西。若要使 QA 部门有效率，则应将它作为整个设计过程中必不可少的部分。

最后，良好的工具是开发过程的重要组成部分。但是，在大型 C++系统中，工具不能弥补固有的设计质量问题。本书主要讨论如何进行软件设计以保证质量的问题。

第1部分

基础知识

本书包含了许多与面向对象设计和C++编程有关的内容。因为不是所有的读者都有相同的背景，所以，在本书的第1部分，我们讨论基础知识，以达到进行进一步讨论的一个共同起点。

第1章回顾了C++语言、基本的面向对象设计原则和概念，以及本书中所采用的标准编码和文档编制惯例。本章的目的是帮助读者拉平基础知识。作者认为这些内容的大部分对于许多读者来说是熟悉的。这里没有介绍任何新的内容。熟练的C++程序员可以选择略过本章或只在需要时参考一下。

第2章描述了一般意义的设计惯例的精简集合。大多数富有经验的软件开发者已经发现了这些惯例。坚持这里所介绍的基本规则是成功的软件设计不可缺少的部分。这些规则也可以作为本书中所介绍的高级而微妙的原则和指导方针的框架。

1 预备知识

这一章回顾 C++程序语言和面向对象分析的一些重要的方面，这些知识对于大型系统设计来说是基本的。这里没有介绍革命性的东西，但有一些材料可能是读者不熟悉的。首先，我们仔细分析了多文件程序、声明与定义，以及在头（.h）文件和实现（.c）文件上下文中的内部连接和外部连接，然后研究了 typedef 声明和 assert 语句的使用。在介绍了一些有关命名习惯和类成员设计的风格问题之后，我们又研究了**迭代器**（iterator）——最常用的面向对象设计模式之一。在本章结尾，我们充分讨论了用于整本书的逻辑设计符号。对继承和分层也进行了简单的讨论。最后是一个关于在接口中进行最小化的建议。

1.1 多文件 C++程序

对于所有（除了最小的）程序来说，将整个程序都置于单个文件中既不明智也不实用。首先，每次修改了程序的任何一部分，都必须重编译整个程序。也不能在另一个程序中重用这个程序的任何一部分，除非把源代码拷给另一个文件。这种复制很快就会成为令维护者头疼的事。

把一个程序中紧密关联的各部分的源代码分别放在单独的文件中，可以使程序更有效地编译，同时也可以使它的局部能够在其他程序中重用。

在这一节中，我们回顾与那些从多个源文件生成的程序有关的 C++语言结构的一些基本性质。这些概念会频繁地在本书中使用。

1.1.1 声明与定义

一个声明就是一个定义，除非[①]：

① **ellis**，3.1 节，14 页。

- 它声明了一个没有详细说明函数体的函数；
- 它包含一个 extern 定义符并且没有初始化函数或函数体；
- 它是一个包含在一个类定义之内的静态类数据成员的声明；
- 它是一个类名声明；
- 它是一个 typedef 声明。

一个定义就是一个声明，除非：

- 它定义了一个静态类数据成员；
- 它定义了一个非内联成员函数。

> **定义：一个声明将一个名称引入一个程序；一个定义提供了一个实体（例如，类型、实例、函数）在一个程序中的惟一描述。**

一个声明将一个名称引入一个作用域。声明与定义的区别在于：在 C++中，在一个给定的作用域中重复一个声明是合法的。与之相比，在程序中使用的每一个实体（例如，类、对象、枚举值或函数）却只能有一个定义。例如：

```
int f(int,int);
int f(int,int);
class IntSetIter;
class IntSetIter;
typedef int Int;
typedef int Int;
friend IntSetIter;
friend IntSetIter;
extern int globalVariable;  // bad idea (global variable declaration)
extern int globalVariable;  // (see Section 2.3.1)
```

都是声明，在单个的作用域内可以重复任意次。另一方面，下面这些在文件作用域内的声明同时也是定义，所以在一个给定的作用域内出现不能超过一次，否则就会引起编译错误：

```
int x;                                  // bad idea (global variable)
char *p;                                // bad idea (global variable)
extern int globalVariable = 1;          // bad idea (global variable)
static int s_instanceCount;
static int f(int, int) { /* ... */ }
inline int h(int, int) { /* ... */ }
enum Color { RED, GREEN, BLUE };
enum DummyType {};
enum { SIZE = 100 };
enum {} silly;
const double DEFAULT_TOLERANCE = 1.0e-6;
class Stack { /* ... */ };
struct Util { /* ... */ };
union Rep { /* ... */ };
template<class T> void sort(const T** array, int size) { /* ... */ }
```

我们应该注意到，函数和静态数据成员声明是例外的，它们虽然不是定义，但是在一个类定义中也不可以重复：

```
class NoGood {
    static int i;      // declaration
    static int i;      // illegal in C++
  public:
    int f();           // declaration
    int f();           // illegal in C++
};
```

1.1.2 内部连接和外部连接

当一个.c 文件编译时，C 预处理器（cpp）首先（递归地）包含头文件，形成一个含有所有必要信息的单个源文件。然后这个中间文件（称为“编译单元”）被编译生成一个与主文件名相同的.o 文件（目标文件）。连接把在不同的编译单元中产生的符号联系起来，构成一个可执行程序。有两种截然不同的连接：内部的和外部的。连接所用的类型会直接影响到我们如何将一个给定的逻辑结构合并进我们的物理设计中。

> **定 义**：如果一个名称对于它的编译单元来说是局部的，并且在连接时不可能与其他编译单元中的同样的名称相冲突，那么这个名称有**内部连接**。

内部连接意味着对这个定义的访问被局限于当前的编译单元中。也就是说，一个有内部连接的定义对于任何其他编译单元来说都是不可见的，所以在连接过程中不能用来解析未定义的符号。例如：

```
static int x;
```

虽然定义在文件作用域内，但是关键词 static 决定了连接是内部的。内部连接的另外一个例子是一个枚举类型：

```
enum Boolean { NO, YES };
```

枚举类型是定义（不仅仅是声明），但是它们自己决不会将符号引进.o 文件。要想让有内部连接的定义影响程序的其他部分，它们必须放置在头文件中，而不是在.c 文件中。

有内部连接定义的一个重要例子是一个类的定义。类 Point（见图 1-1）的描述是一个定义，不是声明；因此，它不能在同一作用域内的一个编译单元中重复出现。如果类要在单个编译单元之外使用，那么它们必须定义在一个头文件中。内联函数定义（就像显示在图 1-1 底部的对 operator==的定义）是有内部连接定义的另一个例子。

> **定 义**：在一个多文件程序中，如果一个名称在连接时可以和其他编译单元交互，那么这个名称就有**外部连接**。

```
class Point {
    int d_x;
    int d_y;

  public:
    Point (int x, int y) : d_x(x), d_y(y) {}          // internal linkage
    int x() const { return d_x; }                     // internal linkage
    int y() const { return d_y; }                     // internal linkage
    // ...
};                                                    // internal linkage

inline int operator==(const Point& left, const Point& right)
{
    return left.x() == right.x() && left.y() == right.y();
}                                                     // internal linkage
```

图 1-1 一些有内部连接的定义

外部连接意味着该定义不仅仅局限于单个编译单元中。有外部连接的定义可以在.o 文件中产生外部符号，这些外部符号可以被所有其他的编译单元访问，用来解析它们未定义的符号。这种外部符号必须在整个程序中是惟一的，否则这个程序不能被连接。

非内联成员函数（包括静态成员）有外部连接，非内联函数、非静态自由函数（即nonmember 函数）也一样。有外部连接的函数的例子见图 1-2。

```
// non-inline member function:
Point& Point::operator+=(const Point& right)
{
    d_x += right.d_x;
    d_y += right.d_y;
    return *this;
}                                                 // external linkage

// non-inline free function:
Point operator+(const Point& left, const Point& right)
{
    return Point(left.x() + right.x(), left.y() + right.y());
}                                                 // external linkage
```

图 1-2 一些有外部连接的函数的定义

注意，我们谈到非成员函数时都是指**自由函数**，而决不是指**友元函数**。一个自由函数不必是任何类的**友**元；无论如何它都应该是一个实现细节（见 3.6 节）。

在可能的地方，C++编译器会把一个内联函数的函数体直接替换成函数调用，不将任何符号引进.o 文件。有时编译器会选择放下一个内联函数的静态拷贝（因为各种原因，例如递归或动态绑定）。这个静态拷贝只将一个局部符号引入当前的.o 文件，这个符号不能与外部符号交互。

因为声明只对当前的编译单元有用，所以声明本身并不将任何东西引进.o 文件。考虑以下这些声明：

```
/* 1 */        int f();          // bad idea (see Section 2.3.2)
/* 2 */        extern int i;     // bad idea (see Section 2.3.1)
               struct S {
/* 3 */            int g();      // fine
               };
```

这些声明本身没有影响到最终的.o 文件的内容。每一个都只是命名一个外部符号，使当前的编译单元在需要的时候可以访问相应的全局定义。实际上是符号名称的使用（例如，调用一个函数）而不是声明本身导致了一个未定义的符号被引入到.o 文件。正是这个事实允许构建早期的原型：只要所缺的功能不是所需的，那么部分完成的对象可以用在运行程序中。

在前面这个例子中，三个声明中的每一个都激活对一个外部定义函数或对象的访问。我们也许会很随便，说这些“声明”有外部连接。但是还有其他种类的声明不能用来激活对外部定义的访问。我们常常会称这类声明有“内部”连接。例如：

```
typedef int Int;                        // internal linkage
```

是一个 typedef 声明。它既不能将任何符号引进.o 文件，也不能通过它来访问一个有外部连接的全局对象；它的连接是内部的。一种重要的恰好有内部连接的声明是类的声明：

```
class Point;                            // internal linkage
struct Point;                           // internal linkage
union Point;                            // internal linkage
```

这些声明在把名称 Point 作为某种用户自定义类型引入时都有相同的作用；特殊的声明类型（如，class）不必和实际的定义类型（如，union）相匹配：

```
class Rep;
// ...
union Rep {
    // ...
};
```

这些声明潜在引用的定义也有内部连接；这个特性把类声明与前面所举例子中的外部声明区分开来。类声明和类定义都对.o 文件没有贡献，只是为当前的编译单元所用。

另一方面，静态类数据成员（在类定义内部声明）有外部连接：

```
class Point {
    static int s_numPoints;  // declaration of external object
    // ...
};
```

静态的类数据成员 s_numPoints（如上所示）只是一个声明，但在.c 文件中，它的定义有

“外部的”连接：

```
// point.c
int Point::s_numPoints;        // definition of external object
                               // (initialized to 0 by default)
```

注意，按照这种语言的规范，每一个静态类数据成员都必须在最终程序的某处准确地定义一次[1]。

最后，C++语言对待枚举类型和类的方式是不同的：

```
enum Point;                    // error
```

在 C++中不可能未经定义就声明一个枚举类型。就像我们将要看到的，类声明常常用来代替预处理器的#include 指令，以便可以未经定义就声明一个类。

1.1.3 头（.h）文件

C++中，将一个带有外部连接的定义放置在一个.h 文件中几乎都是编程错误。如果这样做了，还把这个头包含在不止一个编译单元中，那么把它们连接在一起时就会出错，且会出现下面这样的出错信息：

```
MULTIPLY DEFINED SYMBOL.
```

在 C++中，在一个头文件的文件作用域内放置带有内部连接的定义是合法的，但这种做法并不是人们所希望的。不仅因为这些文件作用域内的定义会污染全局名称空间，而且在有静态数据和函数的情况下，它们会在每一个包含有这个头的编译单元中消耗数据空间。甚至在文件作用域中将数据声明为 const 也会引起相同的问题，尤其是在这个常量的地址已经获得的情况下。将一个文件作用域常量（带有内部连接）和一个静态常量类成员（带有外部连接）进行比较：在整个程序中将只能有一个类作用域常量的拷贝。图 1-3 提供了一些应该和不应该归入头文件的（定义）例子。

双重的非成员数据定义这种冗余不仅会影响到程序的大小，还会影响运行性能，因为它破坏了主机的高速缓存机制。但是，有时也会有正当的理由在头文件的文件作用域中放置一个用户自定义对象的静态实例。特别是，这样一个对象的构造函数可以用来确保一个特殊的全局工具（例如 iostream）在使用前已被初始化[2]。虽然这种解决方案对于中小型系统来说是很好的，但对于非常大型的系统来说就可能有疑问。我们会在 7.8.1.3 小节中继续讨论这个问题。

---

① **ellis**，9.4 节，179 页；18.3 节，405 页。

② **ellis**，3.4 节，21~22 页。

```
// radio.h
#ifndef INCLUDED_RADIO
#define INCLUDED_RADIO

int z;                                  // illegal: external data definition
extern int LENGTH = 10;                 // illegal: external data definition
const int WIDTH = 5;                    // avoid: constant data definition
static int y;                           // avoid: static data definition
static void func() {/*...*/}            // avoid: static function definition

class Radio {
    static int s_count;                 // fine: static member declaration
    static const double S_PI;           // fine: static const member dec.
    int d_size;                         // fine: member data definition
    // ...
  public:
    int size() const;                   // fine: member function declaration
    // ...
};                                      // fine: class definition

inline int Radio::size() const
{
    return d_size;
}                                       // fine: inline function definition

int Radio::s_count;                     // illegal: static member definition

double Radio::S_PI = 3.14159265358;     // illegal: static const member def.

int Radio::size() const { /*...*/ }     // illegal: member function definition

#endif
```

图 1-3　应该和不应该归入头文件的定义

1.1.4　实现（.c）文件

有时候我们会选择定义一些函数和数据用于我们自己的实现，不希望这种实现暴露于我们的编译单元之外。有内部连接（而不是外部连接）的定义可以出现在一个.c 文件的文件作用域中，不会影响全局（符号）名称空间。在.c 文件的文件作用域中要避免的定义是：没有声明为静态的数据和函数。例如：

```
// file1.c

int i;                                          // external linkage
int max(int a, int b) { return a > b ? a : b }  // external linkage
```

上面的这些定义有外部连接，可能会与全局名称空间中的其他相似的名称之间存在潜在

冲突。因为内联和静态自由函数有内部连接，这些种类的函数可以在.c文件的文件作用域定义，不会污染全局名称空间。例如：

```
// file2.c

inline int min(int a, int b) { return a < b ? a : b }           // internal

static int fact(int n) { return n <= 1 ? 1 : n * fact(n - 1); } // internal
```

枚举类型的定义、声明为static的非成员对象以及（通过默认的）const数据定义等也有内部连接。在.c文件的文件作用域中定义所有这些实体都是安全的。例如：

```
// file3.c
#include <math.h>
class Link;                                                     // internal

enum { START_SIZE = 1, GROW_FACTOR = 2 };                       // internal

const double PI_SQ = M_PI * M_PI;                               // internal

static const char *names[] = { "Ntran", "Ptran", "NPN", "PNP" }; // internal

static Link *s_root_p;                                          // internal
Link *const s_first_p = s_root_p;                               // internal
```

其他结构，如typedef声明和预处理器宏，不会将输出符号引进.o文件。它们也可以出现在.c文件的文件作用域中，不会影响全局名称空间。例如：

```
typedef int (PointerToFunctionOfVoidReturningInt *)();

#define CASE(X) case X: cout << "X" << endl; // Classic C preprocessor

#define CASE(X) case X: cout << #X << endl;  // ANSI C preprocessor
```

typedef和宏在C++中的用处有限，如果滥用的话可能有害。我们将在1.2节和2.3.3节讨论typedef的危险性，在2.3.4节讨论宏的危险性。

1.2 typedef（类型别名）声明

一个typedef声明为一个已存在的类型创建了一个别名，而不是一个新的类型。因此，一个typedef只是提供了类型安全的假象。所以接口中的typedef可以轻而易举地做更多有害的事而不是好事。

思考一下图1-4中所示的类Person。我们已决定将typedef声明嵌入Person类中，以避免影响全局名称空间，并使它们更容易查找到。成员函数setWeight定义为接受一个以“Pounds”为单位的一个重量实参，同时方法getHeight返回以“Inches”为单位的高度。

```
// person.h
#ifndef INCLUDED_PERSON
#define INCLUDED_PERSON

class Person {
    // ...
  public:
    typedef double Inches;
    typedef double Pounds;
    //...
    void setWeight(Pounds weight);
    Inches getHeight() const;
    //...
};

#endif
```

图 1-4 typedef 不等于类型安全

很不幸，一个嵌套的 typedef 并不比一个在文件作用域中声明的 typedef 提供更多的类型安全：

```
void f (const Person& person)
{
    Person::Inches height = person.getHeight();
    person.setWeight(height);                    // ok ??
};
```

两个类型名称 Inches 和 Pounds 在结构上是相同的，因此是完全可以互换的。这些类型别名绝对不提供编译时类型安全，也使人不容易知道实际的类型。

但是，在定义复杂的函数参数时类型别名可以起作用。例如：

```
typedef int (Person::*PCPMFDI)(double) const;
```

把 PCPMFDI 声明为一个类型：指针，指向一个 const Person 成员函数，其参数为 double 类型，并返回一个 int。在跨越不同的编译器和计算机硬件时，有些数据成员的大小必须保持不变，在定义这样的数据成员时，类型别名也是很有用的（见 10.1.3 节）。

1.3 assert 语句

标准 C 库提供一个名为 assert 的宏（见 assert.h），用以保证一个给定的表达式求值为一个非零值；否则就会输出一个出错信息并且结束程序执行[①]。assert 语句使用方便，并且是开

① **plauger**，第 1 章，17~24 页。

发者的一个强有力的实现级文档工具。assert 语句就像活动的注释——它们不仅使假定清楚而准确，而且如果违反了这些假定，它们实际上会对此做某些工作。

要在运行时捕捉程序的逻辑错误，使用 assert 语句可能是一种有效的方法，并且它们很容易从产品代码中过滤出来。一旦开发结束，只需通过在编译过程中定义预处理器符号 NDEBUG，就可以消除这些为检测代码错误而进行的冗余测试的运行时开销。但是一定要记住，放在 assert 中的代码在产品版本中将被省略掉。请思考下面的一个 String 类的部分定义：

```
class String {
    enum { DEFAULT_SIZE = 8 };
    char *d_array_p;
    int d_size;
    int d_length;

  public:
    String();
    // ...
};
```

如果 assert 宏的表达式参数影响了软件的状态（正如以下代码那样），那么产品版本就会展示完全不同的行为。

```
String::String()
: d_size (DEFAULT_SIZE)
, d_length(0)
{
    assert(d_array_p = new char[d_size]);    // error
}
```

通过确保 assert 中的代码完全独立于正常的对象操作，就可以避免出现这个问题：

```
String::String()
: d_size (DEFAULT_SIZE)
, d_length(0)
{
    d_array_p = new char[d_size];
    assert(d_array_p);                       // fine
}
```

这种处理容错技术的通用做法是抛出像 CodingError 这样的异常信息了事[1]。甚至较高层次的软件都使用这种方式来捕捉和处理该问题。在没有错误处理程序时，默认的行为就简化为 assert 行为[2]。

---

① **murray**，9.2.1 节，208~210 页。

② 要想知道更多关于有意使用异常的信息，见 **ellis**，15.1 节，355 页。

1.4 有关风格的一些问题

当程序员聚集在一起开始一个项目时，常常要讨论采用什么编码标准。这些标准很少能够对产品的质量作贡献。他们经常关心诸如以下这些问题：

我们应该缩进 2、4 或 8 个空格吗？

我们应该在一个 if 语句的右圆括号与左括弧之间像下面这样加一个空格：

```
if (exp) {
```

还是应该像下面这样不加空格：

```
if (exp){
```

在一个大项目的开始阶段，往往要花费数个星期的时间来讨论标准问题。我们得出的结论是：虽然标准化有好处，标准的列表还是越小越好，并且每一条标准都应该由清晰的工程原则来推动。上述的两个例子都不符合这些规则。

我们要学习的另一件事情是什么时候要加强标准，有两个领域：接口和实现。好的接口比好的实现要重要得多。接口会对客户程序产生直接的影响，并且有全局影响力。实现只影响作者和代码维护者。

有明确的理由要求在设计接口时上实行严格的标准，尤其是在大型项目中。修补接口通常要比修补实现更困难和更昂贵。假如有一个封装得很好的接口，那么扔掉一个糟糕的实现，用一个更好的实现来取代它，通常并不会太困难。

1.4.1 标识符名称

下面这些编码惯例已经被无休止地争论过，但经受住了痛苦的考验。本书提出的大多数的建议集中在影响接口方面，在这些方面可以最强烈地感受到它们的好处。此外，这其中大部分只是个人爱好问题。如果说有一个规则的话，那就是保持一致。

1.4.1.1 类型名称

C++的语法是复杂的。有关其结构性质的精细线索总是受欢迎的。一个相当标准并被广泛接受的惯例是以特殊的考虑来处理类型名称。在本书中我们会始终把类型名称的首字母置为大写字母，而其他非类型名称则以小写字母开头。

对于我们的目的来说，类型就是那些既非数据亦非函数的实体：

- 类（Classes）
- 结构（Structures）
- 联合（Unions）
- 类型别名（Typedefs）

- 枚举（Enumerations）
- 模板（Templates）

以下是一些说明这种编程风格的声明：

```
class Point;
struct Date;
union Value;
enum Temperature { COLD, WARM, HOT, VERY_HOT } temp;①
typedef Temp Temperature;
template class Stack<int>;
int Point::getX() const;
void Point::setX(int xCoord);
```

用词典编纂的方式来区分类型名称和其他名称是一种客观的可验证的标准，对于客户和实现者来说同样可以提高可读性。如果一致地使用，这个惯例可以使接口更容易理解，代码也更容易维护。

1.4.1.2 多词标识符名称

给标识符命名的人有两种类型——一种人提倡用下划线字符（“_”）来分隔单词，另一种人则提倡第二个及以后的单词首字母大写：

```
this_is_a_very_long_identifier    与    thisIsAVeryLongIdentifier
```

两种方式都有争论。我刚开始是在下划线阵营，但是为了遵从大多数人的意见不得不作出改变。如今我意识到其实并没有什么区别：这只是一个你习惯于什么的问题。也许使用大写字母的方式更好一点，因为名称可以更短，你习惯了以后它们就会变得更容易阅读。使用大写字母也可以把下划线应用于其他的用途（见 6.4.2 节和 7.2.1 节）。重要的是要在整个产品线中始终保持一致。

在同一个产品中一批类使用一种命名习惯而另一批类使用另一种命名习惯，这种做法显得不专业，且有可能会令人困惑，尤其是在考虑到付费顾客会（或有朝一日会）直接访问基本 C++类的情况下。但是，有一些程序员可能只是简单地把这些不一致性当作一种风格问题来处理。

在这本书里我采用了“大写字母标准”。然而，无论你采用了何种方式，我都要强烈地忠告：要保持一致，尤其是在接口中。

1.4.1.3 数据成员名称

如果人们记得给他们的类的数据成员始终加上前缀（如，d_），易读性和可维护性将会大

① 本文本中枚举类型和（静态）常量的名称都是大写的，并使用下划线来分隔单词。

大增强。考虑下面的类 Shoe：

```
class Shoe {
    double d_temperature;
    int d_size;
    // ...
  public:
    // ...
    void expand(double calories);
    // ...
    void setSize(int size);
    // ...
};
```

在成员函数内部，局部（自动）变量所拥有的值只是临时的；当成员函数返回之后它们就不存在了。另一方面，类成员数据定义了对象的状态，它们存在于成员函数调用之间：

```
void Shoe::expand(double calories)
{
    const double FACTOR = 42.57;        // Always initialized to same value
    // ...                              // (probably belongs at file scope).

    double factor = calories * FACTOR; // short lived automatic variables
    d_temperature += FACTOR / d_size;  // use of "state" variables
}
```

使用 d_的主要目的是要在一个独立上下文中以一种机械的方法突出类数据成员。因为这两种数据类型有非常不同的用途，使用词典编纂的方式来区分类数据成员名称和局部变量名称，可以使对象实现更易于理解。

经常可以看到这样的成员函数，把一个实例变量（如，d_size）设置成包含单个的赋值表达式：

```
inline
void Shoe::setSize(int size)
{
    d_size = size;
}
```

在数据成员前面放置 d_，也可以避免为操纵函数的参数臆想出来的怪异名称（如，sz）：

```
void Shoe::setSize(int sz)
{
    size = sz;
}
```

前缀 d_的选择是很任意的。我们之所以不单单使用一个下划线字符（_）来作前缀，是

因为以一个下划线字符开始的标识符是留给 C 编译器用的[①]。有些人更喜欢使用后下划线来达到这个目的：

```
void Shoe::setSize(int size)
{
    size_ = size;
}
```

我发现把加后缀留给其他目的是有用的（例如用_p 标识一个指针数据成员）[②]。你也可以使用不同的前缀（例如用 s_标识静态类数据）。无论是在一个类中还是在文件作用域中，非常量静态数据都潜在地包含独立于实例的状态信息。正如将在 6.3.5 节中讨论的，静态类数据成员可以移动到一个.c 文件的文件作用域中，以帮助避免编译时耦合。因为这两种数据类型具有非常相似的特性和可互换性，所以用 s_来标识.c 文件中的状态变量也是有意义的。始终遵循这种命名习惯可以方便查找一个组件中的所有独立于实例的状态变量。

静态类或文件作用域常量数据若没有状态则毫无用处。只需将其名称全部使用大写字母就可以标识这种数据的性质和生存期。对于一个类作用域的常量数据来说，一个诸如 S_DEFAULT_VALUE 或简单如 DEFAULT_VALUE 的名称可以工作得同样好。在这本书里我们选择用 S_DEFAULT_VALUE 来标识类作用域常量静态数据，用以提醒我们需要将它保持私有（见 2.2 节）。

比较而言，一个非静态常量数据成员有更有限的寿命，它的值不必在每一个对象的化身中都相同。因此，它的名称可以以小写字母出现和以前缀 d_开始。

```
class Set { /* ... */ }
class SetIter {
    Set *const d_set_p;        // d_set_p is a const pointer to a Set.
    const double D_PI;         // bad idea (should be static)
    // ...
};
```

我们整个公司都采用加 d_的惯例，没有人抱怨。

1.4.2 类成员布局

在使用一个不熟悉的对象时，要断定在哪儿能找到东西可能会比较困难。尽管在类中成员函数的顺序明显是一个风格的问题，但从客户的观点来看它可以帮助保持前后一致。对成员功能进行分类的一个基本方法就是看它是否潜在地影响了对象的状态。

在图 1-5 中描述了一个对开发者和客户都有用的组织。这个组织具有通过功能（几乎每个

① **ellis**，2.4 节，7 页。

② 标识符后缀的另一个用法见 6.4.2 节。

C++类都会提供某种功能）种类进行分组的优势。这个组织也独立于将被实现的特定的抽象。

```
class Car {
    // ...
  public:
    // CREATORS
    Car(int cost = 0);
    Car(const Car& car);
    ~Car();

    // MANIPULATORS
    Car& operator=(const Car& car);
    void addFuel(double numberOfGallons);
    void drive(double deltaGasPedal);
    void turn(double angleInDegrees);
    // ...

    // ACCESSORS
    double getFuel() const;
    double getRPMs() const;
    double getSpeed() const;
    // ...
};
```

图 1-5　创建函数/操纵函数/访问函数成员组织

创建函数（CREATOR）使对象开始存在和停止存在。注意 operator=不是一个创建函数，但却是（习惯上）第一个操纵函数。**操纵函数**（MANIPULATOR）只是非常量成员函数；**访问函数**（ACCESSOR）是常量成员函数。这种纯粹的对象分组使得其很容易在匆匆一瞥中得到校验：所有的访问函数都声明为类的常量函数，所有的操纵函数都不声明为类的常量函数。但主要的好处还在于为解剖一个不熟悉的类的基本功能提供一个公共的起点。对于更大型的类来说，在每一节中把成员按字母顺序排序可能是有用的。对于像包装器（wrapper，在 5.10 和 6.4.3 节讨论）这样的非常大型的类来说，其他组织方式可能更适用。

有些人会设法把成员函数分组为 get/set 对，如图 1-6 所示。对某些用户来说，这种风格是概念误导的结果，认为一个对象不过就是一个有数据成员的公共数据结构，每一个数据成员都必须既有一个“get”函数（访问函数）又有一个“set”函数（操纵函数）。这种风格本身可能（有时候）会阻碍真正封装接口的产生，在真正封装的接口中，数据成员没有必要透明地反映在对象的行为中。

最后，还有在哪里放置数据成员的问题。完全封装好的类没有公共数据。从逻辑角度来看，数据成员只是类的实现细节。因此，许多人都愿意把类的实现细节（包括数据成员）放在类定义的末尾，如图 1-7 所示。

```
class Car {
    double d_fuel;
    double d_speed;
    double d_rpms;

  public:
    Car(int cost = 0);
    Car(const Car& car);
    Car& operator=(const Car& car);
    ~Car();

    double getFuel() const;
    void setFuel(double numberOfGallons);

    double getRPMs() const;
    void setRPMs(double rpms);

    double getSpeed() const;
    void setSpeed(double speedInMPH);

    // ...
};
```

图 1-6 get / set 成员组织

```
class Car {
  public:
    Car(int cost = 0);
    Car(const Car& car);

    // ...

  private:
    double d_fuel;
    double d_speed;
    double d_rpms;
};
```

图 1-7 尾部数据成员组织

虽然这个组织对于不成熟的客户来说可能更具有可读性，把实现细节藏在类定义末尾的尝试掩饰了它们并未被隐藏的事实。在头文件中实现细节的存在会影响编译时耦合的程度，这种耦合不会因为在类定义中重新放置这些细节就简单地消失。

因为本书要研究物理的和组织的设计问题，我们始终把头文件中的实现细节放在公共接口的前面（部分原因是要强调它们的存在）。在第 6 章中，我们会讨论像这样的实现级的混乱如何才能被整个地从一个头文件中移走，从而确实对客户隐藏。

1.5 迭代器

也许在面向对象设计中最普通的模式就是迭代器（iterator）模式[①]。一个迭代器就是一个与某种原始对象密切耦合的对象，并且这个原始对象被提供给了这个迭代器；它的用途是允许客户程序顺序地访问原始对象的部件、属性或子对象。

对象常常会表现为其他对象的集合，这种对象一般称为**容器**（containers）。集合、列表、栈、堆、队列、哈希表（hash tables）等等就是典型的容器对象。请注意在有关的地方，我们常常会用一段前导注释来标识代码体源文件。例如：

```
// stack.h
#ifndef INCLUDED_STACK
// ...
```

```
// stack.c
#include "stack.h"
// ...
```

例如，我们考虑一下图 1-8 所示的实现一个整数集合的简单类。我们可以从它的头文件中看到，IntSet 使用 IntSetLink 对象来实现，但那是一个封装好的类的实现细节。在这个最小的实现中，我们已经作了这样的选择：通过使这些另外自动产生的函数成为私有函数，来防止用户建立一个 IntSet 拷贝或赋值给一个 IntSet 对象。（注释 NOT IMPLEMENTED 指示此功能不存在，即使是私有的也一样。）只允许 IntSet 的用户建立一个空集合，加进整数，检查成员关系，以及删除它。

```
// intset.h
#ifndef INCLUDED_INTSET
#define INCLUDED_INTSET

class IntSetLink;
class IntSetIter;
class ostream;

class IntSet {
    // DATA
    IntSetLink *d_root_p;  // root of a linked list of integers

    // FRIENDS
    friend IntSetIter;

  private:
    // NOT IMPLEMENTED
    IntSet(const IntSet&);
    IntSet& operator=(const IntSet&);
```

① **gamma**，Iterator，第 5 章，257~271 页。

② **stroustrup**，5.3.2 节，160 页；7.8 节，243 页；8.3.4 节，267 页。

```
public:
  // CREATORS
  IntSet();
      // Create an empty set of integers.
  ~IntSet();
      // Destroy this set.
  // MANIPULATORS
  void add(int i);
      // Add an integer to this set.  If the given integer is
      // already present, this operation has no effect.

  // ACCESSORS
  int isMember(int i) const;
      // returns 1 if integer i is a member of the set,
      // and 0 otherwise.
};

#endif
```

图 1-8 一个简单的整数集合类

图 1-9 中显示了一个用来测试这个有限功能的微小的测试驱动程序。注意，本书中的驱动程序用文件名后缀.t.c 来标识。

```
// intset.t.c
#include "intset.h"
#include <iostream.h>

main()
{
    IntSet a;

    a.add(1); a.add(2); a.add(3); a.add(2); a.add(4); a.add(6);

    for (int i = 0; i < 10; ++i) {
        cout << ' ' << i << '-' << (a.isMember(i) ? "yes" : "no");
    }

    cout << endl;
}

// Output:
//      john@john: a.out
//       0-no 1-yes 2-yes 3-yes 4-yes 5-no 6-yes 7-no 8-no 9-no
//      john@john:
```

图 1-9 测试 IntSet 功能的微小的驱动程序

假设我们要找出这个集合中存在什么成员，然后把它们打印出来。理论上，我们可以写我们自己的输出函数（如图 1-10 所示），但是这种实现的性能可能会稍显不足。

```
#include <limits.h>     // defines INT_MIN and INT_MAX
ostream& operator<<(ostream& o, const IntSet& intSet)
{
    o << "{ ";
    for (int i = INT_MIN; i <= INT_MAX; ++i) {
        if (intSet.isMember(i)) {
            o << i << ' ';
        }
    }
    return o << '}';
}
```

图 1-10　IntSet 输出运算符的行不通的实现

一个显而易见的解决办法就是把 operator<<函数设为 IntSet 类的一个友元，以便利用其内在表示法。我们可以这样做，但是如果客户不喜欢这个运算符的实现所提供的格式怎么办？如果后来我们发现需要访问内部成员又会发生什么？比方说，要比较两个 IntSet 对象。

我们可以持续加入新的成员和友元，但是每次我们这样做，都会把我们的客户和我们自己置于增加类的复杂度的危险之中。不断地修改和扩充对象功能是一种众所周知的把程序错误引进软件的方式。同样，除非计划支持多个版本，否则不关心这些新功能的其他客户将被迫承担它们。

我们可以提供一种普遍而有效的方法来访问集合的单个成员，从而一次性地解决大多数的不足，而不是一次一个地处理这些不足。假设我们决定把这种能力直接加入类 IntSet 本身，就像图 1-11 描述的那样。这时一个客户程序就有可能对 IntSet 类的一个实例进行迭代，并以任何需要的格式打印出那个对象的所有内容。图 1-12 说明了迭代器的一些功能。如果不考虑集合的实现可能如何变化的话，客户的代码将不受影响。

```
class IntSet {
    // ...
  public:
    // ...

void reset();
    // Reset to beginning of sequence of integers.  The Current
    // integer will be invalid only if the set is empty.

void advance();
    // Advance to the next integer in the set.  If the current
    // integer was the last in the set, the current integer
    // will be invalid after advance returns.  Note that the
    // behavior is undefined if the current integer is already
    // not valid.

int current() const;
    // Return the current integer in the sequence.  Note that the
    // behavior is undefined if the current integer is not valid.
```

```
    int isCurrentValid() const;
        // Return 1 if the current integer is valid, and 0 otherwise.
        // Note that the current integer is valid if the set is not
        // empty and we have not advanced beyond the last integer
        // in the set.
    }
```

图 1-11 尝试给容器自身增加迭代能力

```
ostream& operator<<(ostream& o, const IntSet& intSet)
{
    o << "{ ";
    for (intSet.reset(); intSet.isCurrentValid(); intSet.advance()) {
        o << intSet.current() << ' ';
    }
    return o << '}';
}
```

图 1-12 IntSet 输出运算符的另一种实现

但是，图 1-12 中的设计仍然存在问题。对于一个给定的对象，在任何时候都至多只能有一个迭代器在运行。假设我们要为 IntSet 实现一个比较函数，为了调试目的，我们决定打印出比较迭代过程中的集合内容。打印例程可能会有破坏比较函数迭代状态的有害副作用。问题是 IntSet 要分配足够的空间以便为每一次迭代保留状态信息。无论该迭代是否是活动的，被分配的空间都要保留。如果因为某种原因我们需要有一对嵌套的 for 循环，它们在同一个集合的元素上迭代，我们就必须复制整个集合。

这个问题可以通过让客户程序保持内部状态或保留一些其他形式的位置占有来解决。如果客户程序动态地分配状态，客户必须记着删除状态以避免内存泄漏。

如果位置占有是一种整数索引的形式，那么可能会有一些对集合底层实现的附加实际约束。例如，若集合实现为一个链表（而不是数组），那么在每次迭代中都会有潜在的二次指数时间复杂性（即：$O(N^2)$），因为 for 循环的每次迭代都会导致不得不遍历链表。

标准的方法是为每一个容器类都提供一个迭代器类（在同一头文件中）。迭代器声明为容器的友元，因此可以访问它的内部组织。把迭代器和容器类定义在同一个头文件中，是为了避免有关“远距离”友元关系的问题（在 3.6 节中讨论）。具体容器（如 IntSet）的迭代器一般创建在程序堆栈中；这样当迭代器离开作用域时其状态会自动删除。迭代器对象可以更具有空间效率，因为每次迭代的空间只需在迭代过程（本身）中存在。同样，在一个给定的容器中，任意数量的迭代器在任何时间都可以独立地活动，不会互相干扰。

作为一个实践问题，对迭代器来说通常都假设它们所操作的对象在迭代过程中不会被修改或删除。在迭代中对象出现的顺序通常也是依赖于实现的，并且容易遭受擅自的改变。理想的做法是，迭代器开发者应明确声明是否定义迭代的顺序。为安全起见，迭代器的客户程序不应该假定一个顺序，除非指定了一个顺序。

图 1-13 描述了本书中所使用的标准迭代器模式的设计。这个迭代器对象将使用 for 循环。该迭代器的语法相当简洁。运算符的使用很不明显，尤其是对于以前从未见过这种用法的人来说。人们很容易认为这种风格是运算符重载的滥用，因为易读性降低了。然而，这种做法还有其他理由。

```
class IntSetIter {
    // DATA
    IntSetLink *d_link_p;     // root of linked list of integers

  private:
    // NOT IMPLEMENTED
    IntSetIter(const IntSetIter&);
    IntSetIter& operator=(const IntSetIter&);

  public:
    // CREATORS
    IntSetIter(const IntSet& IntSet);
        // Create an iterator for the specified integer set.

    ~IntSetIter();
        // Destroy this iterator (an unnecessary comment).

    // MANIPULATORS
    void operator++();
        // Advance the state of the iteration to next integer in set.

    // ACCESSORS
    int operator()() const;
        // Return the value of the current integer.

    operator const void *() const;
        // Return non-zero value if iteration is valid, otherwise 0.
};
```

图 1-13 IntSet 容器的一个标准迭代器

因为在大型设计中迭代器可能会也确实会频繁出现，开发者需要考虑的最重要的事情一定是一致性问题。如果我们不使用运算符重载而使用函数，那么每一次使用相同的函数名称就很重要；否则我们将发现自己会无意识地给这些函数取错名，并总是要为寻找语法细节而查看头文件。图 1-14 中列出了一些具有代表性的可能是等价的函数名称。

it	it.more()	it.isMore()	it.valid()	it.notDone()
++it	it.next()	it.getNext()	it.advance()	it.getMore()
it()	it.item()	it.getItem()	it.element()	it.value()

图 1-14 我们应该用哪些名称

经验表明，每一个标准迭代方法都采用图 1-14 的左栏中的运算符，形成了超越具体迭代类型的一致的、易用的、很快就会被熟悉的和易于公认的习惯用语。无论你决定用什么，都

要保证在整个产品线中保持一致。IntSet 输出运算符的最后实现如图 1-15 所示：简明扼要的迭代器符号提供了一种简洁的实现。

```
ostream& operator<<(ostream& o, const IntSet& intSet)
{
    o << "{ ";
    for (IntSetIter it(intSet); it; ++it) {
        o << it() << ' ';
    }
    return o << '}';
}
```

图 1-15 使用简洁的迭代器实现的 IntSet 输出运算符

在图 1-15 中，使用前加 1（++it）而不是后加 1（it++）是经过深思熟虑的；后加 1 这种版本需要第二个形式参数并且不是总能得到[1]。此外，当应用到基本类型时，迭代器加 1 的语义更近似于前加 1 的语义（见 9.1.1 节）。

1.6 逻辑设计符号

面向对象设计需要一个丰富的符号集合[2]。这些符号的大多数用来表示设计的逻辑实体之间的关系。

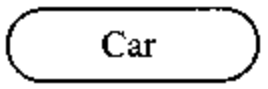
定义：

符号	含义
X	X 是一个逻辑实体（如，类）
x	x 是一个物理实体（如，文件）
B —IsA→ A	B 是 A 的一种
B —Uses-In-The-Interface— A	B 在其接口中使用 A
B —Uses-In-The-Implementation— A	B 在其实现中使用 A

在本书中，我们始终用一个椭圆形的框来标识逻辑实体（例如：类、结构、联合）；

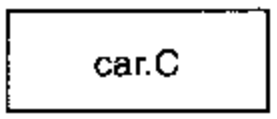
① 见 **ellis**，13.4.7 节，338~339 页。

② **booch**，第 5 章，171~228 页。



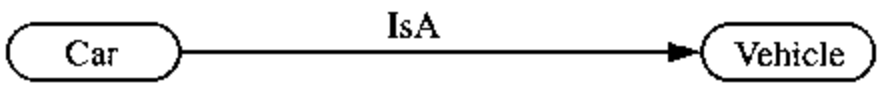

```
class Car {
    // ...
};
```

而用长方形来标识物理实体：

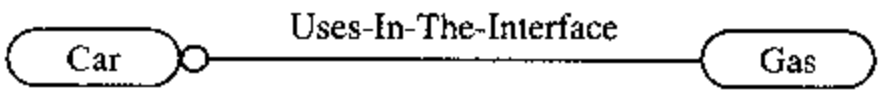



```
// car.c
#include "car.h"
// ...
```

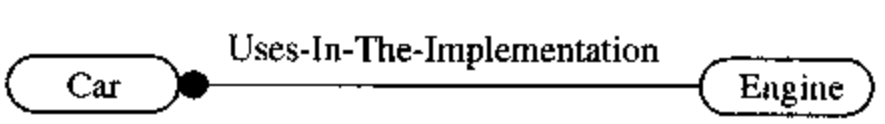
对于我们的目标，三种逻辑符号就足够了：




```
class Car : public Vehicle {
    // ...
};
```




```
class Car {
    // ...
  public:
    void addFuel(Gas *);
    // ...
};
```




```
class Car {
    Engine d_motor;
    // ...
};
```

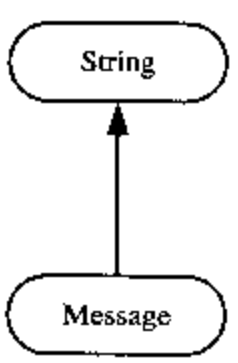
如果还需要另外的逻辑符号，一个清楚地标识关系的加了标签的箭头就足够了。

1.6.1 IsA 关系

假设 Message 是一种 String，也就是说，Message 类型的对象可以用在需要 String 对象的任何地方。

```
class String {
    // ...
  public:
    // ...
};

class Message : public String {
    // ...
  public:
    // ...
};
```

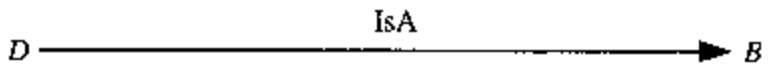
（a）省略的类定义



（b）符号表示法

图 1-16 IsA 关系

正如我们从图 1-16（a）的定义中可以看到的，Message 类继承 String 类，在图 1-16（b）中用一个箭头来表示这种关系：

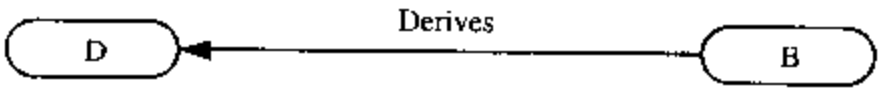


也就是说，*D*——▶*B* 的意思就是“*D* 是一种 *B*”和“*D* 继承 *B*”。

箭头的方向是很重要的，它指向隐含依赖的方向。类 *D* 依赖 *B*，因为 *D* 由 *B* 派生而来，*B* 必须先出现，因为 *D* 要把 *B* 作为一个基类命名：

```
class B { /* ... */ };
class D : public B { /* ... */ };
```

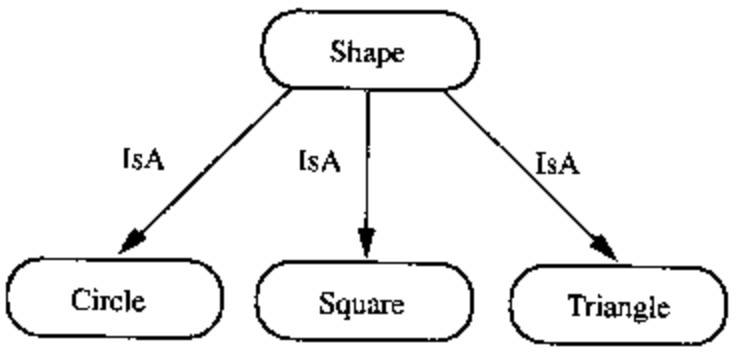
我们常常会看到箭头指向相反的方向，这可能会令人误解。箭头表示两个实体间的一种不对称关系，关系名称由标签指出（在此例中是“IsA”）。如果把箭头画成其他的方式，我们在逻辑上将不得不改变关系名称，例如“派生（Derives）”或“是...的基类（Is-A-Base-Class-Of）”：



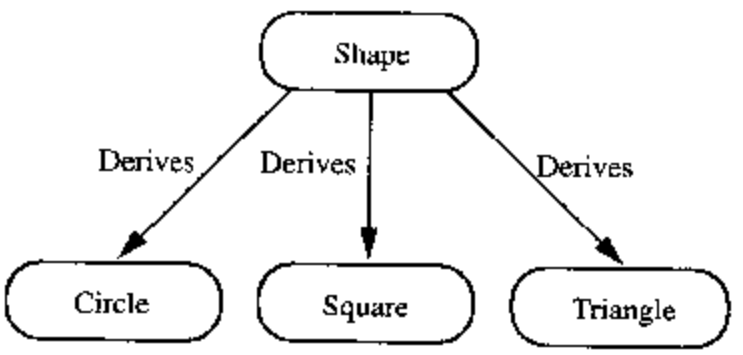
（很少用）

这种可替代的符号很不受欢迎，因为箭头的方向与隐含依赖的方向相反。

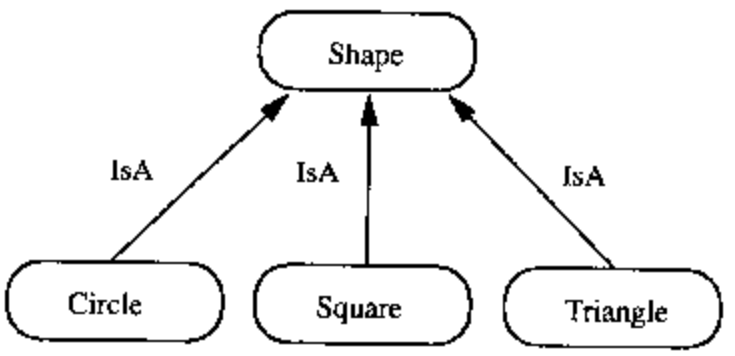
因为分析物理依赖对于好的设计来说是必需的，所以我们采用了使用 IsA 标签和指向隐含依赖方向的箭头的符号。图 1-17 用经典的图形示例展示了继承表示法。



（a）不正确的符号



（b）很少使用的符号



（c）正确的符号

图 1-17　用于标识派生的符号

1.6.2 Uses-In-The-Interface 关系

无论何时一个函数在它的参数表中命名了一个类型或把一个类型命名为一个返回值，就称这个函数在它的接口中使用了那个类型。也就是说，如果一个类型名称是函数返回类型或基调（signature）的一部分，该函数的接口就使用了该类型[①]。

> **定义**：如果在声明一个函数时提到某个类型，那么就是**在该函数的接口中使用了该类型**。

例如，自由函数

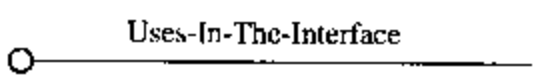
```
int operator==(const IntSet&, const IntSet&);
```

在它的接口中明显地使用了类 IntSet。该函数碰巧返回一个 int，所以 int 也将被认为是这个函数接口的一部分。然而，基本类型是普遍存在的，在实践中可以忽略基本类型。

> **定义**：如果在一个类的（公共）成员函数的接口中使用了某类型，那么就是**在这个类的（公共）接口中使用了该类型**。

在 C++中对类的逻辑访问有三个级别：公共的（public）、保护的（protected）和私有的（private）。一个类的公共接口定义为那个类的公共成员函数的接口的并集。一个类的保护接口的定义与此类似。换句话说，当类 B 的一个（公共）成员函数在它的接口中使用了类 A，我们就说类 B 在 B 的(公共)接口中使用了类 A[②]。例如，类 IntSetIter 的构造函数 IntSetIter(const IntSet&)在它的接口中使用了类 IntSet；因此在 IntSetIter 的接口中使用了 IntSet。

“Uses-In-The-Interface（在接口中使用）”关系是最常见的关系之一，表示为



也就是说，*B* ○——*A* 的意思是“*B* 在 *B* 的接口中使用了 *A*”。我们有时会随便些，说“*B* 在它的接口中使用了 *A*”，但是我们的意思始终是 *B* 在 *B* 的接口中使用了 *A*，而决不是 *B* 在 *A* 的接口中使用了 *A*。

你可以把符号○——想像成一个箭头，它的尾巴在○上，头不见了（或者想像成正指着一个管弦乐队成员的乐队指挥棒）。隐含的箭头方向很重要——它指向隐含依赖的方向。也就是说，如果 *B* 使用了 *A*，那么 *B* 依赖 *A*，但反之则不然。（我们会在 3.4 节中更多地讨论隐含依赖）。

图 1-18 显示了 intset 组件的逻辑视图，包括其中定义的逻辑实体（类和自由运算符函数）间的“Uses-In-The-Interface”关系。图中反映出 IntSetIter 和两个自由运算符都在它们各自的

---

① 不包括 typedef 的可能使用，它只是同义名。

② 友元关系和 Uses-In-The-Interface 关系的交互在 3.6.1 节讨论。

接口中使用了 IntSet。

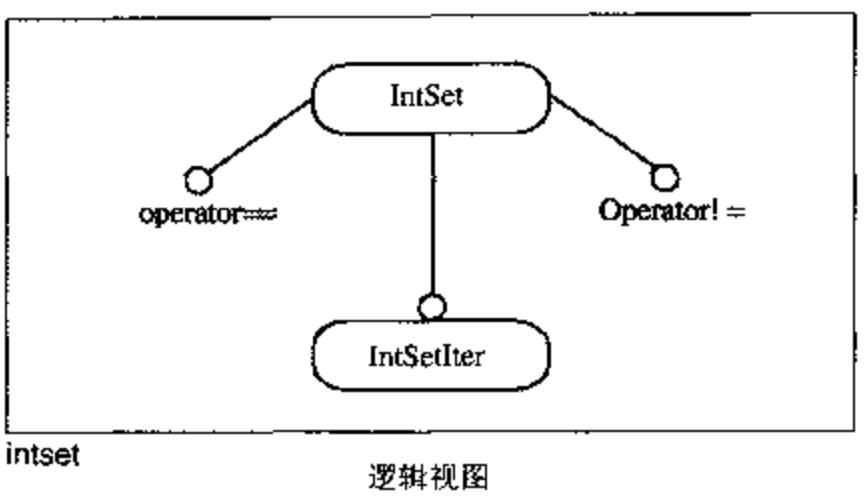


图 1-18 在 intset 组件内部的“Uses-In-The-Interface”关系

“Uses-In-The-Interface”关系是一种对逻辑设计和物理设计都颇有价值的工具。当要将逻辑实体（类和自由运算符）限制在文件作用域内时，这个符号最有用。自由运算符经常会从逻辑图表中省略，以减少符号的混乱。

一个类的实际的逻辑接口可能相当大而且复杂。通常我们最有兴趣展示的特性是一个本质的依赖关系而不是详细的使用方法。在一个类的接口中所使用的类型集合，比被任何特定的成员函数所使用的类型集合都更稳定（即在开发和维护过程中不大可能改变）。所以，与单独的成员函数的使用特性相比，作为整体的类的使用特性越抽象，对在逻辑接口上的小变化就越具有弹性。

1.6.3 Uses-In-The-Implementation 关系

“Use-In-The-Implementation（在实现中使用）”关系增加了设计者抽象表达逻辑依赖的能力。一个逻辑实体将在它的实现中使用另一个逻辑实体（即使没有在它的接口中使用）的表示法，在分析一个设计的基础结构时可能非常有用。和“Uses-In-The-Interface”关系一样，“Uses-In-The-Implementation”关系暗示了两个逻辑实体之间的一种物理依赖。当体系结构设计者们提取高层设计并把它们分布到分散的物理组件中时，可以很好地利用这个信息。

> **定 义**：如果在某函数的定义中提到了某类型，那么在这个函数的实现中就使用了该类型。

考虑下面的自由函数 operator==的实现，它**假设**迭代器总是以**相同的顺序**返回等价的 IntSet 对象的成员：

```
int operator==(const IntSet& left, const IntSet& right)
{
    IntSetIter lit(left);
    IntSetIter rit(right);
```

```
        for (; lit && rit; ++lit, ++rit) {
            if (lit() != rit()) {
                return 0;
            }
        }
        // At least one of lit and rit now evaluates to 0.
        return lit == rit;
    }
```

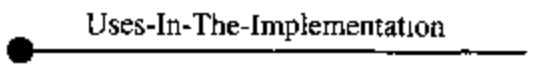
在上面的实现中，创建了两个迭代器：为每个 IntSet 参数都创建一个。只有当两个迭代器都指向有效集合元素时才会进入 for 循环体。随着每个迭代经过循环，集合里的相应位置的整数被比较。若任何一次这样的比较失败，则集合立刻被认为是不相等的。要从 for 循环退出，以下两个条件都必须为真：

（1）至少有一个迭代器已经到达了集合的末端并且当前是无效的。

（2）没有发现集合的相应条目是不相等的。

当且仅当两个迭代器当前都无效时，两个 IntSet 对象才会相等。

注意 operator==(const IntSet&, const IntSet&) 不是类 IntSet 的友元。因此这个运算符的任何有效的实现都必须利用类 IntSetIter。在 operator==与类 IntSetIter 之间的使用关系产生了一种 operator==对类 IntSetIter 的隐含依赖关系。因为在这个运算符的实现中（而不是在它的逻辑接口中）使用了 IntSetIter，所以我们使用一种稍微不同的符号来表示这种关系：

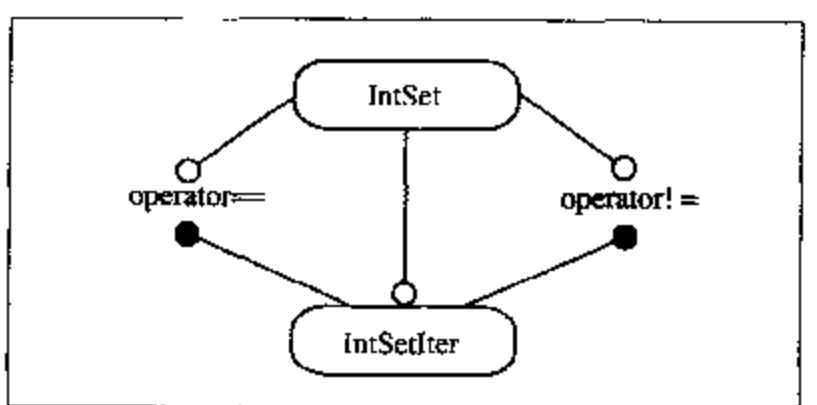


也就是说，*B* ●——*A* 的意思就是在 *B* 的实现中使用了 *A*。

图 1-19 再次为我们展示了带有两种使用关系的 intset 组件的逻辑视图。特别是我们可以看到

```
int operator==(const IntSet&, const IntSet&)
```

在它的接口中使用了类 IntSet，在它的实现中使用了类 IntSetIter。虽然 operator!=显示为与 operator==对称实现，但实际上 operator!=有可能根据 operator==来实现。



逻辑视图

图 1-19　intset 组件中的两种使用关系

如果在一个函数的接口中使用了一个对象，就自动地认为在那个函数的实现中使用了该对象。所以我们看到符号●——就可以推断它所指示的使用不是用在接口中。例如，我们可以从图 1-19 直接推断，operator!=没有在它的接口中使用 IntSetIter。

> **定义**：如果一个类型（1）被用在某个类的一个成员函数中，（2）在某个类的一个数据成员的声明中被提到，或者（3）是某个类的一个私有基类，那么这个类的实现中就使用了这个类型。

一个类可以以多种形式在它的实现中使用另一个类型。正如我们将在 3.4 节提到的那样，类使用一个类型的特定方式，不仅会影响到类依赖那个类型的方式，而且会影响到被迫依赖于那个类型的类的客户范围。我们暂时只简单地展示类在它的实现中可以使用一个类型的方式：

> **定义**：
>
> "Uses-In-The-Implementation"关系的特定种类：
>
> | 名称 | 含义 |
> |---|---|
> | Uses | 该类有一个成员函数命名了该类型。 |
> | HasA | 该类嵌入了该类型的一个实例。 |
> | HoldsA | 该类把一个指针（或引用）嵌入了该类型。 |
> | WasA | 该类私有继承于该类型。 |

1.6.3.1 使用（Uses）

如果一个类的任何成员（包括一个私有成员）函数在它的接口或它的实现中命名了一个类型，就认为在该类的逻辑实现中使用了该类型。

图 1-20 举例说明了因为在类 Crook 的一个成员函数（bribe）体中命名了类型 Judge，所以在 Crook 中的实现中使用了 Judge。换句话说，类 Crook 使用了 Judge。

```
class Crook {
  private:
    void bribe();
    // ...
};

class Judge;

void Crook::bribe() {
    Judge *bad = 0;
    // ...
};
```

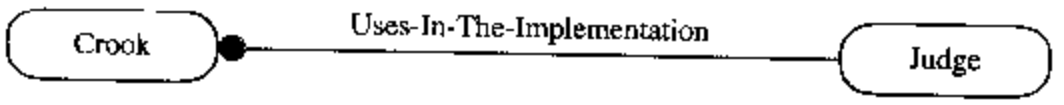


图 1-20 Crook 使用了 Judge

1.6.3.2 HasA 和 HoldsA

当一个类 X 嵌入了类型 T 的一个（私有）数据成员，就出现了另一种使用形式。这种内部使用法通常称为 **HasA**。即使类 X 包含了一个数据成员，其类型只是从 T 派生而来（在 C 语言意义上）（例如：T*或 T&），仍然认为在 X 的逻辑实现中使用了 T。我们偶尔会把这种内部使用法称为 **HoldsA**。

图 1-21 显示了类 Tower 的定义和类 Cannon 的声明。在类 Battleship 的实现中使用了这两个类型。具体说来，就是 Battleship **HasA** Tower 和 Battleship **HoldsA** Cannon。我们没有在其所用的符号上进行区别：HasA 和 HoldsA 都用通常的 ●——符号来表示。

```
class Tower { /* ... */ };
class Cannon;                          // declaration only

class BattleShip {
    Tower d_controlTower;
    Cannon *d_replaceableForwardBattery_p;
    Cannon& d_fixedAftBattery;
    // ...
};
```

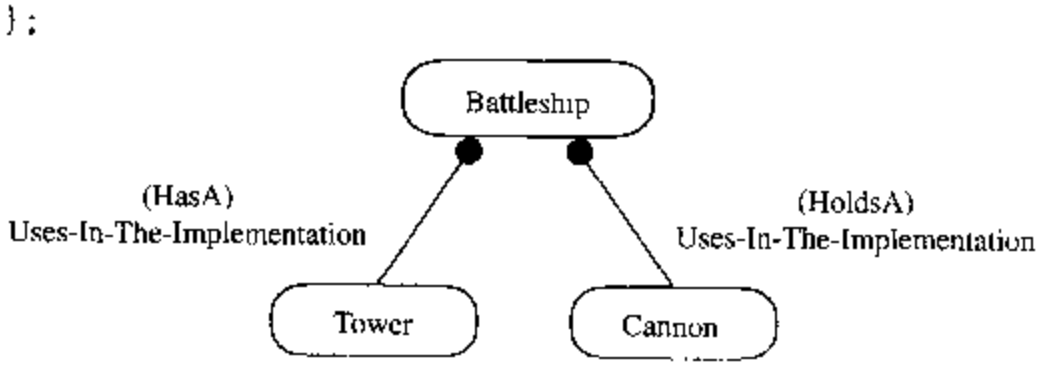


图 1-21 Battleship HasA Tower 和 HoldsA Cannon

1.6.3.3 WasA

私有继承一个类型也是在一个类的逻辑实现中使用该类型的一种方式。私有继承是一个派生类的实现细节。从逻辑角度来看，一个私有的基类（就像一个私有的数据成员）对客户程序是不可见的。私有继承是一种技术，它只可以用来传播其基类属性的一个子集。这种很少使用的关系已经被生动地赋予了一个术语 WasA，如图 1-22 所示。

图 1-22 显示了类 Battleship 的定义，它充当 ArizonaMemorial 的一个私有基类。一旦处在激活服务状态，战舰 Arizona 就是 1941 年珍珠港轰炸中的损失战舰之一。Arizona 目前是一个有礼物商店和展览品的博物馆。

虽然私有继承是一种实现细节，公共继承和保护继承却不是。继承加大了与基类型兼容的类型集合。因此非私有继承会引入信息（客户程序通过编程可以访问这些信息）。公共继承和保护继承的独特性质使它们值得拥有自己的符号，如 1.6.1 节中所示。

```
class Battleship { /* ... */ };
class Shop { /* ... */ };
class Exhibit;                          // declaration only

class ArizonaMemorial : private Battleship {
    Shop d_giftShop;
    Exhibit *d_current_p;
    Exhibit& d_default;
    // ...
};
```

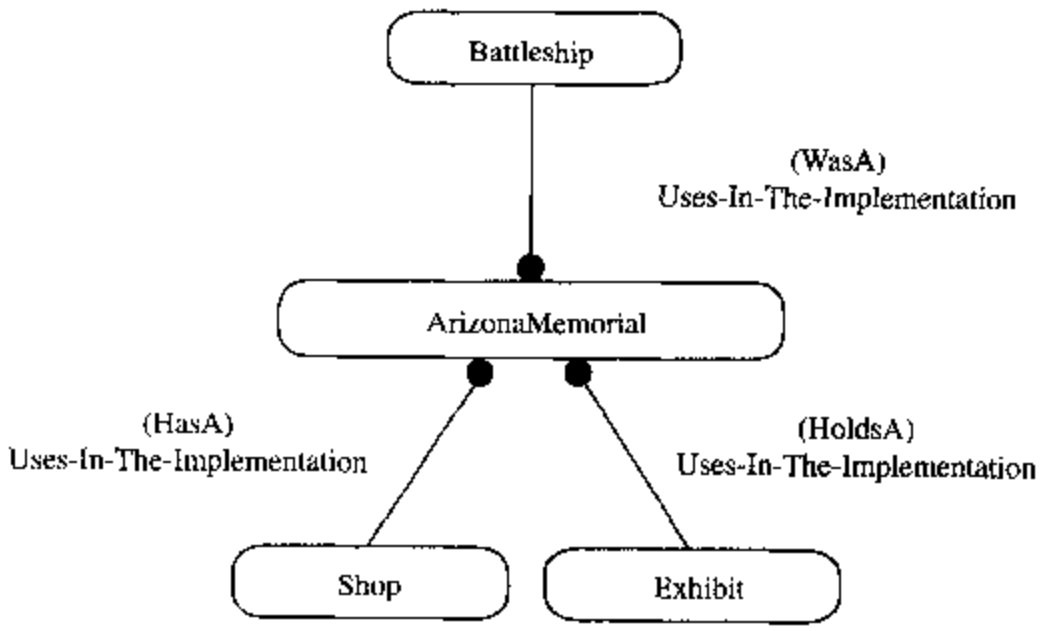


图 1-22 ArizonaMemorial WasA Battleship

现在我们已经回顾了所有的逻辑符号，我们需要开始认真考虑严肃的物理设计问题了。设计的逻辑和物理方面是紧密结合的。每一种逻辑关系——IsA、Uses-In-The-Interface 以及 Uses-In-The-Implementation——都隐含了逻辑实体间的一种物理依赖。正如我们将在第 3 章介绍的，最终正是这些逻辑关系决定了系统内的物理相互依赖关系。

1.7 继承与分层

在面向对象设计的上下文中，当有人提到“层次结构”这个词时，许多人就会想到“继承”。继承是逻辑层次结构的一种形式——分层则是另一种形式。到目前为止，在面向对象设计中更常见的逻辑层次结构形式产生于分层。

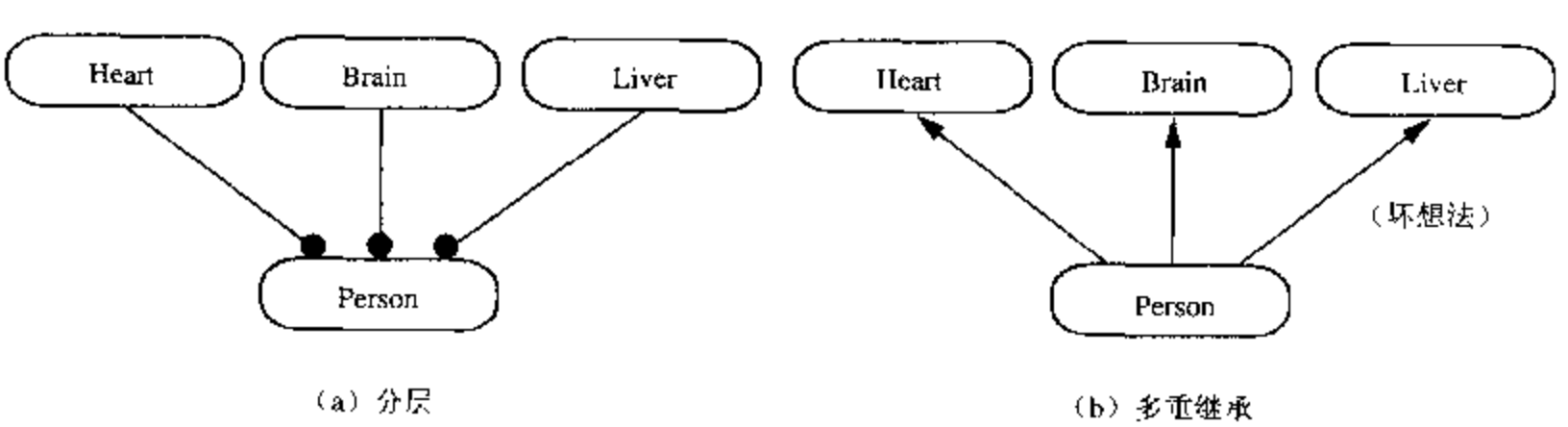
> **定义**：如果某个类在它的实现中实质地使用了某个类型，则该类分层于该类型之上。

分层是把更小、更简单或更原始的类型建成更大、更复杂或更精密的类型的过程。分层经常通过组合（例如 HasA 或 HoldsA）来进行，但任何形式的实质使用（即任何导致物理依赖的使用）都具备分层的资格。

通常，客户不能通过更高层对象的接口来编程访问分层类型的实例。隐含的意思是基本类型是在一个较低级的抽象层次上。例如，一个人有一个心脏、一个大脑、一个肝脏等等，但这些分层的器官对象不是大多数健康人的公共接口的一部分。像一个列表这样简单的对象经常作为一个连接集合来实现，但 Link 类本身不会用在写得最好的 List 类的接口中。

继承，连同动态绑定，可以用来区别面向对象语言（如 C++）和基于对象语言（如 Ada），后者支持用户自定义类型和分层，但不支持继承[1]。继承的语义与分层有很大的不同。例如，基类和派生类的公共功能都可以被客户访问[2]。对继承来说，越特殊越具体的类依赖越一般越抽象的类。对分层来说，在较高抽象层次上的类依赖较低抽象层次上的类。

分层是面向对象设计者“兵工厂”中的一种重要而常常配置不足的武器。程序员新手在需要分层的地方试图使用继承的情况并不少见。图 1-23 显示了逻辑层次结构的两个例子。在两个案例中，Person 为了尽其职责都隐含依赖 Heart、Brain 和 Liver。在这里分层显然是正确的方法，因为一个 Person 不是一个 Heart、一个 Brain 或一个 Liver。应改为，一个 Person 有（has）一个 Heart、一个 Brain 和一个 Liver。而且，这些器官不应该暴露在一个 Person 的接口上。有了分层，客户不必屈从于这些内部细节的接口。



（a）分层　　（b）多重继承

图 1-23　分层与多重继承

1.8　最小化

有些类作者想让他们的类满足所有人的所有需要。这样的类已经被亲切地称为温尼贝戈（Winnebago）类。这种很常见和似乎高贵的愿望令人忧虑。作为开发人员，我们必须记住，只因为一个客户要求增加功能并不意味着对所有类都是适当的。假设你是一个类的作者，10 个客户中的每一个都请求你进行不同的增加。如果你同意，会发生两件事情：

（1）你将不得不实现、测试和存档 10 个新特征，你开始并没有认为这些新特征是你要实现的抽象的一部分（这本身就是问题的一个症状）。

① 见 **booch**，第 2 章，39 页。

② 注意：私有继承是分层的一种形式。

（2）你的 10 个客户的每一个都会得到 9 个他们并没有请求和可能不必要或不想要的新特征。

每次你增加一个特征去取悦一个人，你就扰乱和潜在地骚扰了你的客户库的其他人。曾经发生过这样的事情：原本是轻量级的和很有用的类，经过一段时间后变得过于臃肿，不但不能做好每件事情，而且毫不夸张地说，它们已经变得每件事情都做不好了。

注意，在 1.5 节中，我们选择了通过将其成员函数声明为私有明确地禁止对 IntSet 和 IntSetIter 的实例进行初始化或赋值。为一个集合做一个拷贝可能导致并非微不足道的开发工作量，而这样的迭代器功能在实践中很少需要。我们可以推迟多余功能的实现和测试，除非或直到对那个功能的需求出现。推迟实现也是保持我们的选择权的一种方法。这样做不仅实现、测试、存档和维护软件所需做的工作更少，而且通过特意不过早提供功能，我们既不用为对它的性能负责，也不用对它的实现负责。事实上，不实现功能可以改善可用性。例如，使拷贝构造函数私有可以阻止通过值不经意地传递一个对象——一种用于输入输出流软件包（iostream package）的技术[1]。

这种只要组件足够但不必完备的最低限要求方法适用于正在开发的大型项目，在这种项目中，组件的用户是“内部的”或组件的用户处在一个一旦需要即可快速请求和接收额外功能的位置。最极端的情况是组件高度专业化并且作者是惟一有意向的用户。在这种情况下，实现任何不必要的功能都可能是没有根据的。当然，若一个功能实现对一个抽象来说是本质的，则省略该功能实现将没有意义，比方说，对于一个商业组件库，其用户是付费顾客，他们会期望强壮而且完全的功能对象。这个问题并不是黑白分明的，在这两个极端之间存在一个范围，它对应着一个组件将被广泛使用的程度。在进行这种权衡时，记住要考虑到功能总是更容易增加而不容易删除。

1.9 小结

大型 C++程序分布在不止一个源文件中。把程序分割成单独的编译单元可以使重编译更有效，并且更有可能重用。

虽然大多数 C++声明可以在一个给定的作用域中重复，但每一个用在 C++程序中的对象、函数或类都只能有一个定义。

把有内部连接的定义限制在单个的编译单元中，不能影响其他的编译单元，除非它放在一个头文件中。这样的定义可以存在于 .c 文件的文件作用域内，不会影响全局（符号）名称空间。

有外部连接的定义在连接时可以用来解析其他编译单元中的未定义符号。把这样的定义放在头文件中几乎肯定是一个编程错误。

---

① 通过值传递用户自定义类型是引起不必要的性能降级的一个常见原因（见 9.1.11 节）。

Typedef 声明只是类型的别名，并不提供额外的编译时类型安全。

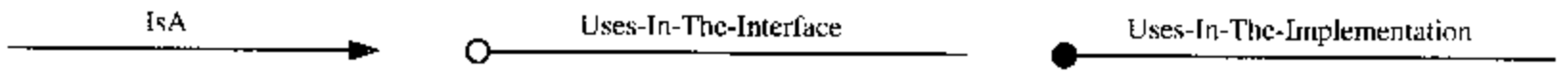
在开发时可以用 assert 语句来有效地发现代码错误，不会影响程序的大小或在一个产品版本中的运行时性能。

在本书中我们采用以下风格惯例：

- 类型标识名以一个大写字母开头。
- 函数和数据以小写字母开头。
- 多词标识名的第二个及以后的词的首字母大写。
- 常量和宏都用大写字母（用一个下划线分隔单词）。
- 类数据成员以 d_为前缀（或者静态成员以 S_为前缀）。
- 类成员函数将按照创建、操纵和访问分类来组织。
- 在类定义中私有细节将放在公共接口之前（主要是强调它们在头文件中的存在）。

迭代器设计模式用于顺序访问某个原始对象的部件、属性或子对象。迭代器被声明为该原始对象的友元，并且它的定义所驻留的头文件应该与该对象的头文件相同。本书中所使用的迭代器表示法是一种简洁的 for 循环模型。

面向对象设计需要一个丰富的逻辑符号集合。但是在本书中，我们只使用三种符号：



每个符号的方向（这里所示的是从左到右）应该和它的标签及所指出的隐含依赖的方向保持一致。一些特殊种类的 Uses-In-the-Implementation 关系有一些特定名称（Uses、HasA、HoldsA 和 WasA），但是，用来表示这些变体的符号都是一样的。

继承和分层是逻辑层次结构的两种形式。到目前为止，分层更常用。它通常包含一个只在实现中的依赖。当问题中的类不能被明显地认为是一种所建议的基类时，分层，尤其是合成比派生好。最后，为了满足若干客户程序的需求而扩展单一类的功能，常常会导致类超重而不受欢迎。对于那些不被广泛使用的类来说，实现过分完全的功能会不必要地增加开发时间、维护开销和代码容量。延缓实现还不需要的功能可减少开发时间，同时预留了选择实现版本的机会。另一方面，用户则希望商用组件库功能完善并且健壮。

2

基本规则

本章将介绍基本设计规则的精简集合。实践证明这些规则是很有用的，并且可以作为框架来讨论本书后面将要介绍的围绕更高级规则的素材。这些基本规则描述基本的规程，例如，限制成员数据访问和减少在全局名称空间中的标识符数量等。特别是，我们将讨论在一个头文件的文件作用域内可以安全放置的结构类型。讨论了为什么需要内部包含卫哨和外部包含冗余卫哨。本章以什么可构成充分文档（例如显式标识未定义的行为）的讨论结束，随后是一个关于标识符命名惯例的简短清单。

2.1 概述

任何精美艺术的美感不仅来源于创造，而且来源于规范。编程也是如此。C++是一种大型语言，有充足的空间进行创造。但是，由于设计空间太大，以致于没有约束——也就是说没有设计结构上的一些适当的约束——大型项目很容易变得难以管理和维护。本书将以设计规则、指导方针和原则的形式介绍这些约束。

设计规则：经验告诉我们，某些编程习惯虽然在 C++中完全合法，但是决不能简单地用于大型项目环境中。直截了当地禁止或毫无例外地要求某种惯例的建议在本书中称为**设计规则**。检验是否遵守了这些规则的过程不能是一种主观过程。设计规则必须足够准确、详尽和良好定义，以便可以客观地检验是否遵守了这些规则。为了效果更好，设计规则必须适合于进行非人为的、借助自动工具的机械验证。

指导方针：经验也告诉我们，有一些习惯应尽可能地避免，这种具有更抽象特性的建议规程称为指导方针，这样的规程有时允许一些合法的例外。指导方针就像必须遵守的经验法则，除非是别的更紧迫的工程原因要求遵守其他方针。

原则：有一些观察和事实在设计过程中经常被证明是有用的，但必须在特定设计的上下文中评估。这些观察和事实称为原则。

确保独立的程序员遵守一致的软件编码标准，是一件很有挑战性的事情。每一个程序员都有自己的习惯扩展集。我给自己强加了许多规则，这些规则比我可能与读者共享的要多得多，但是，大多数只涉及风格，而不是实质。如果我们同意这些规则中的 10%（能给我们带来 90%的实际利益），那么我们就确实做得很好了。

本书包含许多建议。在本章中，我们介绍了一个非常基本的设计规则的集合，称为基本规则，本章解释并证明了（我希望是）每一个规则。读者开始可能不同意所有规则，但是，这些规则经过时间检验，已被证明对非常大型的项目来说是既可行又有效的。

我们把设计规则分成两个不同的类：主要的和次要的。主要的设计规则是指那些必须一直遵守的规程。偏离主要的设计规则很可能不仅影响所涉及的组件的质量，而且会影响系统中其他组件的质量。甚至偶尔违反这些规则就可能动摇一个大型项目成功的基础。贯穿本书，我们假设决不违反主要的设计规则。通常来说，never 决不意味着 NEVER。如果特殊的环境和常识要求违反一个或者多个主要的设计规则，那么设计者有义务完全理解和评估他们的行为所隐含的意义和可能的后果。次要的设计规则是指那些我们强烈推荐但对于一个项目的整体成功不是必不可少的关键因素——例如，只在实现中使用的结构所涉及的问题，不可能影响别的开发者的结构所涉及的问题，以及相对地包含在孤立的实例中并且容易修复的结构所涉及的问题。严格遵守次要规则的要求没有达到至关紧要的地步，因为每个次要规则的违反可能只是增加项目的开销（不像遵守主要规则会影响项目的成功基础）。

因为我们不希望以工程方面的理由来违反设计规则（主要的和次要的），所以对任何禁止一种方法的设计规则，都提供了一个合适的、在所有情况下都能工作的可选方案。

2.2 成员数据访问

封装是一个术语，用于描述在过程接口后面隐藏实现细节的概念。类似的术语有**信息隐藏**和**数据隐藏**。直接访问一个类的数据成员违反封装原则。

主要设计规则

保持类数据成员的私有性。

考虑图 2-1 中类 Rectangle（长方形）的定义。Rectangle 是由标识其左上角和右下角的两个点对象来定义的。因为在 Rectangle 的这个特定实现中，是在内部存储这些 Point 的值，所以我们可以尝试将数据成员变成公共的，以避免为每个类提供操纵函数（即 set）和访问函数（即 get）。

```
// rectangle.h
#ifndef INCLUDED_RECTANGLE
#define INCLUDED_RECTANGLE

class Rectangle {
  public:
    Point d_lowerLeft;     // bad idea (public data)
    Point d_upperRight;    // bad idea (public data)

  public:
    // CREATORS
    Rectangle(const Point& lowerLeft, const Point& upperRight);
    Rectangle(const Rectangle& rect);
    ~Rectangle();
    // MANIPULATORS
    Rectangle& operator=(const Rectangle& rect);
    void moveBy(const Point& delta);
    // ...
    // ACCESSORS
    int area() const;
    // ...
};

// ...

inline
void Rectangle::moveBy(const Point& delta)
{
    d_lowerLeft += delta;
    d_upperRight += delta;
}

// ...

#endif
```

图 2-1 糟糕的（未封装的）Rectangle 类接口

现在考虑，当我们发现 Rectangle 是经常移动的对象时对客户程序的影响。为了改进性能，我们可以试着改变 Rectangle 对象的表示方式。例如，不再存储右上角的绝对位置，而是通过存储其相对于左下角的位置来隐含地表示右上角的位置：

```
class Rectangle {
  public:
    Point d_lowerLeft;                  // same purpose as in Figure 2-1
    Point d upperRightOffset            // new "relative" representation
```

用这种新的表示法，moveBy 成员函数可以在一行中实现（而不是两行），因为右上角与

左下角的相对位置不受移动的影响。

```
inline
Rectangle::moveBy(const Point& delta)
{
    d_lowerLeft += delta;
}
```

右上角的位置不再存储在 Rectangle 对象中，因此当需要它时必须通过计算来得到：

```
void client(const Rectangle& rect)
{
    Point upperRight = rect.d_lowerLeft + rect.d_upperRightOffset;
    // ...
}
```

任何以前直接访问过 d_upperRight 数据成员的客户程序现在都被迫要重新编码。组件重用使这个问题更严重。如果一个定义公共数据的类在可执行程序中是共享的，那么，对单个类的数据表示方式的改变，可能会使得有必要修改所有独立程序的源代码。

> **定义**：若不能通过某个类的逻辑接口编程访问或检测到其包含的实现细节（类型、数据或函数），则称这些实现细节被该类封装了。

封装是面向对象设计的一个重要工具[①]。封装意味着我们将低层次的信息集合在一起，使它们以一种紧密耦合的密切方式潜在地交互。而信息隐藏则用于限制外部世界与某些细节交互，这些细节与类要帮助实现的抽象无密切关系。

使所有的数据成员保持私有，并提供适当的访问函数和操纵函数，如图 2-2 所示，这样我们就可以自由地改变内部表示而不会迫使客户程序重新编码。getUpperRight()的实现可能已经被修改来计算右上角值了，但没有改变其逻辑接口。

除了可维护性外，不要含有公共数据成员还有一些其他原因。例如，在一个类中的数据成员的值很少是独立的。直接（可写的）访问数据（如图 2-2 中的 d_area）很容易使对象处于不一致的状态。只提供一个函数接口就可以授予类作者必要的控制级别，以确保其对象的完整性。提供操纵函数和访问函数也赋予了开发者插入临时代码的机会（例如，为了调试的打印语句，为了性能调整的参考计数，以及为了可靠性的 assert 语句）[②]。

注意，对自身整个被隐藏（不是私有地隐藏在另一个类中就是局部地隐藏在一个.c 文件中）的结构（或类）的数据成员的公共访问是一件不同的事情，不适用上述规则（见 6.4.2 节和 8.4 节）。当数据成员并非私有时，通过使用关键字 struct 而不是 class 来表示认为不需要封装的结构更合适。

---

① **booch**，第 2 章，49~54 页。

② 对于为什么要避免公共数据的进一步的讨论，见 **meyers**，Item 20，71~72 页。

```
// rectangle.h
#ifndef INCLUDED_RECTANGLE
#define INCLUDED_RECTANGLE

class Rectangle {
    Point d_lowerLeft;     // Yet another representation!
    int d_width;           // Fortunately, these data members are private.
    int d_height;
    int d_area;            // Store this redundantly to improve performance.

  public:
    // CREATORS
    Rectangle(const Point& lowerLeft, const Point& upperRight);
    Rectangle(const Rectangle& rect);
    ~Rectangle();

    // MANIPULATORS
    Rectangle& operator=(const Rectangle& rect);
    void moveBy(const Point& delta);
    // ...

    // ACCESSORS
    int area() const;
    Point getLowerLeft() const;
    Point getUpperRight() const;
};

// ...

inline
void Rectangle::moveBy(const Point& delta)
{
    d_lowerLeft += delta;
}

// ...

inline
Point Rectangle::getUpperRight(const Point& delta) const
{
    return d_lowerLeft + Point(d_width, d_height);
}

// ...
```

图 2-2 更好的（封装的）Rectangle 类接口

有人主张使用保护的数据以方便来自派生类的任意访问。但是从维护的角度看，保护访问与公共访问是一样的，因为任何想要得到保护数据的人，只要稍微增加一点派生一个类的

工作即可。和友元关系不同（友元关系明确地表示谁可以访问私有细节），使类数据成为保护的数据会导致封装出现一个大裂口。

适用于公共接口的观点同样也可应用于保护接口。通过将保护接口和公共接口看成是独立但同等重要的，基类作者可以保持可维护性。保持所有数据成员的私有性并提供合适的保护函数，将使得对基类实现的改变独立于任何派生类。

2.3 全局名称空间

对于中等规模的项目来说，如果参加的开发者不止一人，当将开发者各自独立开发的部件集成到一个程序时，就会有命名冲突的危险。问题的严重性会随着系统规模的扩大而呈指数级增长。若冲突是由于第三方提供的集成软件引起的，则情况会更加恶化。

有许多污染全局名称空间的方式，其中一些方式比其他方式麻烦。在大型系统环境中，所有这些方式都是起反作用的。我们现在分别研究几个这样的问题，并以一个设计规则作为本节的结论，该规则描述什么类型的声明和定义可以安全地存在于C++头文件的文件作用域中。

2.3.1 全局数据

有人将全局变量比喻为癌症：不能与之一起生存，但是一旦建立了，则很难将它消除。在一个新的 C++项目中，我们总是可以避免使用外部全局变量而获得成功。这条规则的例外可能涉及两种访问：一种是通过全局变量进行通讯的结构复杂的程序访问（如 Lex 或 YACC）；另一种可能是在嵌入式系统中的访问。

主要设计规则

避免在文件作用域内包含带外部连接的数据。

文件作用域中的带有外部连接的数据，存在与其他编译单元中的全局变量冲突的危险（这些编译单元的作者是以自我为中心的，他们认为自己拥有整个全局域）。但是“名称污染”只是全局变量破坏程序的许多方式之一。全局变量将对象和代码绑在一起，这种方式使得在别的程序中实际上不可能有选择地重用编译单元。在大型项目中，调试、测试，甚至理解大量使用全局变量的系统的开销也可能变得非常惊人。

假如我们不是被迫使用一个已要求在其接口上使用全局变量的系统，则有两种简单的变换方式能将这些变量非全局化：

（1）将所有全局变量放入一个结构中。

（2）然后将它们私有化并添加静态访问函数。

假如我们有下列全局变量：

```
int size;
double scale;
const char *system;
```

通过将这些变量放入一个结构中并使它们成为该结构的静态成员，可以把它们从全局名称空间中删除[①]：

```
struct Global {
    static int s_size;                  // bad idea (public data)
    static double s_scale;              // bad idea (public data)
    static const char *s_system;        // bad idea (public data)
};
```

当然，要记住在相应的.c 文件中定义这些静态数据成员。现在不要用

```
size、scale 或 system
```

来访问全局变量，而是改为分别使用

```
Global::s_size、Global::s_scale 或 Global::s_system
```

来访问。命名冲突的概率现在减少到只可能会与一个单一类名相冲突（即使是这种概率的冲突，应用 7.2 节讨论的技术也容易解决）。

尽管我们已经解决了全局名称空间问题，我们仍未做完我们应做的事。经验表明，就像直接访问非静态成员数据（即特定实例）一样，直接访问静态成员数据（即特定类）会使得大型系统非常难以维护。如果我们要把一个成员(如 s_size)的输出数据类型从 int 变为 double，那么这种改变将是接口的改变；所有的客户程序都会受到影响（不管我们采取了什么措施）。但是我们可以决定把 s_size 的实现改变为一个基于其他更原始的值（如 s_width 和 s_height）而计算得到的值。倘若利用静态函数成员访问（和操纵）静态数据成员，我们就可以进行上述局部修改而不会扰乱整个作用域的客户程序。

消除公共数据的下一步措施是使 Global 成为一个类，并且提供静态的操纵和访问方法，如图 2-3 所示。类 Global 现在所起的作用是作为在程序的任何地方都可以访问的逻辑模块。由于所有的接口函数都是静态的，所以没有必要实例化一个对象来使用该类。将默认构造函数声明为私有并且不具体实现它，可以加强这种使用模型。

为了获得灵活的设计，我们应该谨慎小心，以免滥用全局状态信息。我们希望只有一个对象的单个实例这一事实，并非是使它成为模块（而不成为一个可实例化的类）的充分理由。当全局可访问模块相当于固有的惟一实体（例如系统控制台）时，或者是对于一个没有被特定应用程序所控制的系统范围的常量（如在 limits.h 中可找到的）（见 6.2.9 节）来说，全局可访问模块是有意义的。如果有别的更局部化的（如基于对象的）实现足以胜任，则最好避免

① meyers，Item 28，93~95 页。

使用全局模块[1]。

```
class Global {
    static int s_size;
    static double s_scale;
    static const char *s_system;

  private:
    // NOT IMPLEMENTED
    Global(); // prevent inadvertent instantiation

  public:
    // MANIPULATORS
    static void setSize(int size) { s_size = size; }
    static void setScale(double scale) { s_scale = scale; }
    static void setSystem(const char *system) { s_system = system; }

    // ACCESSORS
    static int getSize() { return s_size; }
    static double getScale() { return s_scale; }
    static const char *getSystem() { return s_system; }
};
```

图 2-3 包含全局状态信息的逻辑模块

2.3.2 自由函数

自由函数也会对全局名称空间形成威胁，尤其是参数基调中不包含任何用户定义类型时。如果一个自由函数在一个.h 文件中定义为有内部连接或者在一个.c 文件中定义为有外部连接，那么在程序集成过程中它可能会与有相同名称和（基调）的另一个函数定义相冲突。运算符函数例外。

主要设计规则

在.h 文件的文件作用域内避免使用自由函数（运算符函数除外）；在.c 文件中避免使用带有外部连接的自由函数（包括运算符函数）。

幸好，自由函数总能分组到一个只包含静态函数的工具类（结构）中。这样产生的内聚性不一定是最佳的，但它减少了全局名冲突的可能性。下面是一个例子：

```
int getMonitorResolution();                                  // bad idea
void setSystemScale(double scaleFactor);                     // bad idea
int isPasswordCorrect(const char *usr, const char *psw);     // bad idea
```

① 见 singleton 设计模式，**gamma**，第 3 章，127~134 页。

该自由函数总是可由下面的静态方法代替：

```
struct SysUtil {
    static int getMonitorResolution();
    static void setSystemScale(double scaleFactor);
    static int isPasswordCorrect(const char *usr, const char *psw);
};
```

惟一有冲突危险的符号是类名 SysUtil。

但是，自由运算符函数不能嵌入类中。这不是一个严重的问题，因为自由运算符要求至少有一个参数是用户自定义类型，因此自由运算符冲突的可能性很小，而且这种冲突在实践中一般不成问题。

2.3.3 枚举类型、typedef 和常量数据

枚举类型、typedef 和（默认的）文件作用域常量数据都有内部连接。人们经常在头文件的文件作用域内声明常量、枚举类型或用户自定义类型，这是一个错误。

主要设计规则

在.h 文件的文件作用域内避免使用枚举类型、typedef 和常量数据。

因为 C++完全支持嵌套类型，因此可以在一个类的作用域中定义枚举类型（或者声明 typedef）而不会在全局名称空间中与别的名称冲突。选择一个更受限的作用域，在该作用域中定义一个枚举类型，我们可以确保所有的该枚举类型中的枚举元素的作用域相同，并且不会与在该作用域之外定义的别的名称冲突。

考虑下列枚举类型：

```
// paint.h
enum Color { RED, GREEN, BLUE, ORANGE, YELLOW };      // bad idea

// juice.h
enum Fruit { APPLE, ORANGE, GRAPE, CRANBERRY };       // bad idea
```

这两个枚举类型可能不是同一个开发者开发的，但是却很可能在某一天包含在同一个文件中，导致 ORANGE 有二义性，无法解析！

```
// picture.c
#include "picture.h"
#include "paint.h"
#include "juice.h"
```

如果将这两个枚举类型改为在不同的类中定义，则我们可以很容易地使用作用域标识符

来解决二义性问题：Paint::Orange 或 Juice::Orange。

因为同样的原因，typedef 和常量数据也应该放在头文件的类作用域内。大多数常量数据是整型的，并且嵌套枚举类型能很好地在类的作用域中提供整型常量。其他的常量类型（如 double、String）必须是类的静态成员，并且在.c 文件中初始化。

```
// array.h
#ifndef INCLUDED_ARRAY
#define INCLUDED_ARRAY

class String;

class Array {
    enum { DEFAULT_SIZE = 100 };
    static const double DEFAULT_VALUE;
    static const String DEFAULT_NAME;

    // ...
};

#endif
```

```
// array.c
#include "array.h"


#include "str.h" // class String



double Array::DEFAULT_VALUE = 0.0;
String Array::DEFAULT_NAME = "";

// ...
```

在大型项目中，除了全局名称冲突外，还有在文件作用域中寻找枚举类型、typedef 和常量等非常现实的问题。将一个 typedef 嵌套在一个类中将迫使名称被完全界定（或声明被继承），这样的名称相对容易发现。同样的推理也适用于枚举类型，但是对于在类中的枚举类型嵌套，我们已经给出了更强有力的论据。

2.3.4 预处理宏

在 C++中几乎不需要宏。它们对包含卫哨（guard）是有用的（见 2.4 节），并且只有在少数情况下，在一个.c 文件中它们的好处才超过它们的问题（最特别的是，当用于为移植或调试获得条件编译时）。但是，一般来说，预处理宏对软件产品是不合适的。

主要设计规则

除非是作为包含卫哨，否则在头文件中应避免使用预处理宏。

预处理器不是 C++语言的一部分：它的基本原则是完全不改变原文（即不对原文进行编译），这使得宏非常难以调试。尽管宏可以使代码易于编写，但是它们的自由形式经常使得代码更难阅读和理解。考虑下列代码片段：

```
#define glue(X,Y) X/**/Y
glue(pri,ntf) ("Hello World");
```

在源代码层次上，我们如何告诉调试者、浏览器或其他的自动工具去处理上述代码呢？

和在.c 文件中包含宏一样糟糕，甚至有更充足的软件工程理由要求头文件不能包含宏。这样的情况是允许的：在一个头文件中使用#define 来定义预处理器常量。因为宏不是 C++语言的一部分，它们不能置于一个类的作用域中。任何包含一个带有#define 的头文件的文件将具有该预处理器常量的定义。

假设 theircode.h 定义了一个常量值 GOOD 作为一个预处理器常量：

```
// theircode.h
#ifndef INCLUDED_THEIRCODE
#define INCLUDED_THEIRCODE

// ...

#define GOOD 0  // bad idea

// ...

#endif
```

```
// ourcode.c
#include "ourcode.h"
#include "theircode.h"

// ...

int OurClass::aFunction()
{
    enum { BAD = -1, GOOD = 0 } status = GOOD;

    // ...

    return status;
};

// ...
```

当编译 ourcode.c 文件时，编译器首先调用预处理器。即使 GOOD 在函数的保护作用域内定义，它对于预处理器也是不安全的，预处理器将毫不留情地将枚举值 GOOD 置为精确的整数 0：

```
// ...

int OurClass::aFunction
{
    enum { BAD = -1, 0 = 0 } status = 0;

    // ...

    return status;
};
```

当编译器遇到枚举类型时，会弹出 Syntax Error（语法错误），但是直到我们经历了一个持续的“grepping”，在所有的.h 文件中寻找谁定义（#define）了我们的一个枚举值后，才能知道出现错误消息的原因。注意，如果在文件作用域中预处理器符号由 const 或 enum 代替，这个问题将不会发生（根据 2.3.3 节的讨论，这也违反了设计规则）。

```
// theircode.h
#ifndef INCLUDED_THEIRCODE
#define INCLUDED_THEIRCODE

// ...

const int GOOD = 100;   // bad idea
    // file-scope constant data

// ...

#endif
```

```
// theircode.h
#ifndef INCLUDED_THEIRCODE
#define INCLUDED_THEIRCODE

// ...

enum { GOOD = 100 };   // bad idea
    // file-scope enumerated value

// ...

#endif
```

在丢失或没有充分实现 C++语言特性的情况下，预处理器宏也可以用于实现模板。如果宏用于此目的，那么宏函数将出现在头文件中。有一些解决这个问题的方法（不借助于宏），可能更适合大型项目。无论如何，与模板相关的问题应在开发过程的早期解决。

2.3.5 头文件中的名称

在头文件的文件作用域中声明的名称，可能潜在地与整个系统中任何一个文件的文件作用域名称冲突。即使在一个.c 文件的文件作用域中声明的带有内部连接的名称也不能保证一定不与.h 文件的文件作用域名称冲突。

主要设计规则

只有类、结构、联合和自由运算符函数应该在.h 文件的文件作用域内声明；只有类、结构、联合和内联函数（成员或自由运算符）应该在.h 文件的文件作用域内定义。

我们希望只能在一个头文件的文件作用域中找到类声明、类定义、自由运算符声明和内联函数定义。在类作用域中嵌入所有其他的结构，可以消除大多数与名称冲突相关的麻烦。

为了辅助说明这个规则，图 2-4 中提供了一个在其他方面没有意义的带有注释的头文件。这个头文件包含了几个结构。注意，一个用户自定义类型的静态实例是一个特殊情况，将在 7.8.1.3 中讨论。现在，把避免在.h 文件中使用这些静态用户自定义对象看作是指导方针而不是规则。

```
// driver.h                                  // fine: comment
#ifndef INCLUDED_DRIVER                      // fine: internal include guard
#define INCLUDED_DRIVER                      // fine: (see Section 2.4)

#ifndef INCLUDED_NIFTY                       // fine: redundant include guard
#include "nifty.h"                           // fine: CPP include directive
#endif                                       // fine: (see Section 2.5)
```

```
#define PI 3.14159265358                    // AVOID: macro constant
#define MIN(X) ((X)<(Y)?(X):(Y))            // AVOID: macro function

class ostream;                                // fine: class declaration
struct DriverInit;                            // fine: class declaration
union Uaw;                                    // fine: class declaration


extern int globalVariable;                  // AVOID: external data declaration
static int fileScopeVariable;               // AVOID: internal data definition
const int BUFFER_SIZE = 256;                // AVOID: const data definition
enum Boolean { ZERO, ONE };                 // AVOID: enumeration at file scope
typedef long BigInt;                        // AVOID: typedef at file scope


class Driver {
    enum Color { RED, GREEN };                // fine: enumeration in class scope
    typedef int (Driver::*PMF)();             // fine: typedef in class scope
    static int s_count;                       // fine: static member declaration
    int d_size;                               // fine: member data definition
private:
  struct Pnt {
      short int d_x, d_y;
      Pnt(int x, int y)
      : d_x(x), d_y(y) {}
    };                                        // fine: private struct definition
    friend DriverInit;                        // fine: friend declaration
  public:
    int static round(double d);               // fine: static member
                                              //       function declaration
    void setSize(int size);                   // fine: member function declaration
    int cmp(const Driver&) const;             // fine: const member
                                              //       function declaration
};                                            // fine: class definition

static class DriverInit {
    // ...
} driverInit;                               // special case (see Section 7.8.1.3)

int min(int x, int y);                      // AVOID: free function declaration

inline
int max(int x, int y)
{
    return x > y ? x : y;
}                                           // AVOID: free inline
                                            //        function definition


inline
void Driver::setSize(int size)
{
    d_size = size;
```

```
}                                       // fine: inline member
                                        //       function definition

ostream& operator<<(ostream& o,
                    const Driver& d);   // fine: free operator
                                        //       function declaration

inline
int operator==(const Driver& lhs,
               const Driver& rhs)
{
    return compare(lhs, rhs) == 0;
}                                       // fine: free inline operator
                                        //       function definition
inline
int Driver::round(double d)
{
    return d < 0 ? -int(0.5 - d)
                 : int(0.5 + d);
}                                       // fine: inline static member
                                        //       function definition

#endif                                  // fine: end of internal include guard
```

图 2-4　头文件作用域中各种各样的结构

2.4　包含卫哨

即使我们遵守了上面的建议：只在头文件的文件作用域中定义类、结构、联合和内联函数，如果相同的头文件在一个编译单元中被包含两次，仍然会有问题。如图 2-5 所示，这个问题可能出现在一个简单的包含图中。

当编译组件 c 的.c 文件时，预处理器首先包含相应的头文件 c.h，依次地，c.h 文件包含 a.h，c.h 包含 b.h，触发 b.h 又第二次包含 a.h。如果 a.h 有任何定义（在 C++中它几乎肯定有），则编译器会抱怨有多种定义。

主要设计规则

在每个头文件的内容周围放置一个惟一的和可预知的（内部）包含卫哨。

解决这个问题的传统方法是，在每个头文件的内容周围加上一个内部的保护包装器。不管包含图是什么样的，包装器都能确保类和内联函数在一个给定的编译单元中只出现一次。注意，我们不是在企图阻止循环包含（这可能是一个设计错误）；我们要阻止的是重复包含，这种重复包含源自一个非循环包含图中的再收敛。对前面例子中存在的编译问题的一个解决方案如图 2-6 所示。注意，我们仍然没有加入冗余（外部）包含卫哨（在 2.5 节中讨论）。

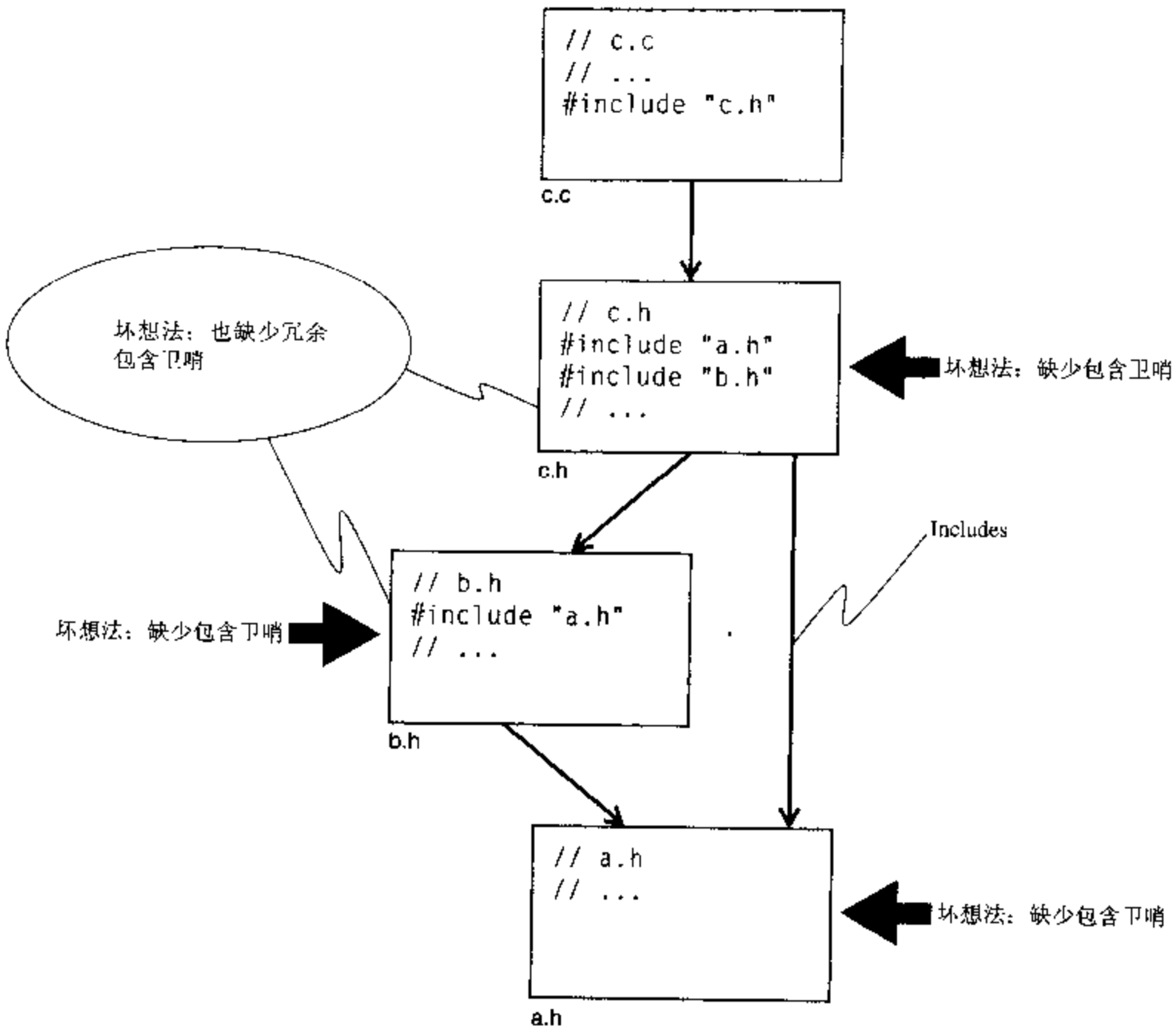


图 2-5 再收敛（reconvergent）包含图引起编译时错误

例如，当 a.h 被包含在一个编译单元中时，预处理器首先检查预处理器符号 INCLUDED_A 是否已定义。如果没有定义，将定义一次卫哨符号 INCLUDED_A，并且是为整个编译单元定义的，然后通过读包含在头文件其余部分中的定义，继续进行预处理。第二次（或更多次地）包含头文件时，预处理器将忽略#ifndef 条件从句中的内容（即文件的其余部分）。

用于包含卫哨的实际符号并不重要，只要它不与整个系统中的其他符号冲突就行。因为包含卫哨是与特定的头文件绑定在一起的，而该头文件名必须在整个系统中是惟一的，所以将头文件名称并入卫哨符号可以保证没有哪两个卫哨符号会是一样的。

预处理器不知道 C++的作用域规则，因此我们必须确保包含卫哨符号不与其他系统中的名称冲突——甚至不能与在一个.c 文件中定义的函数名称冲突。

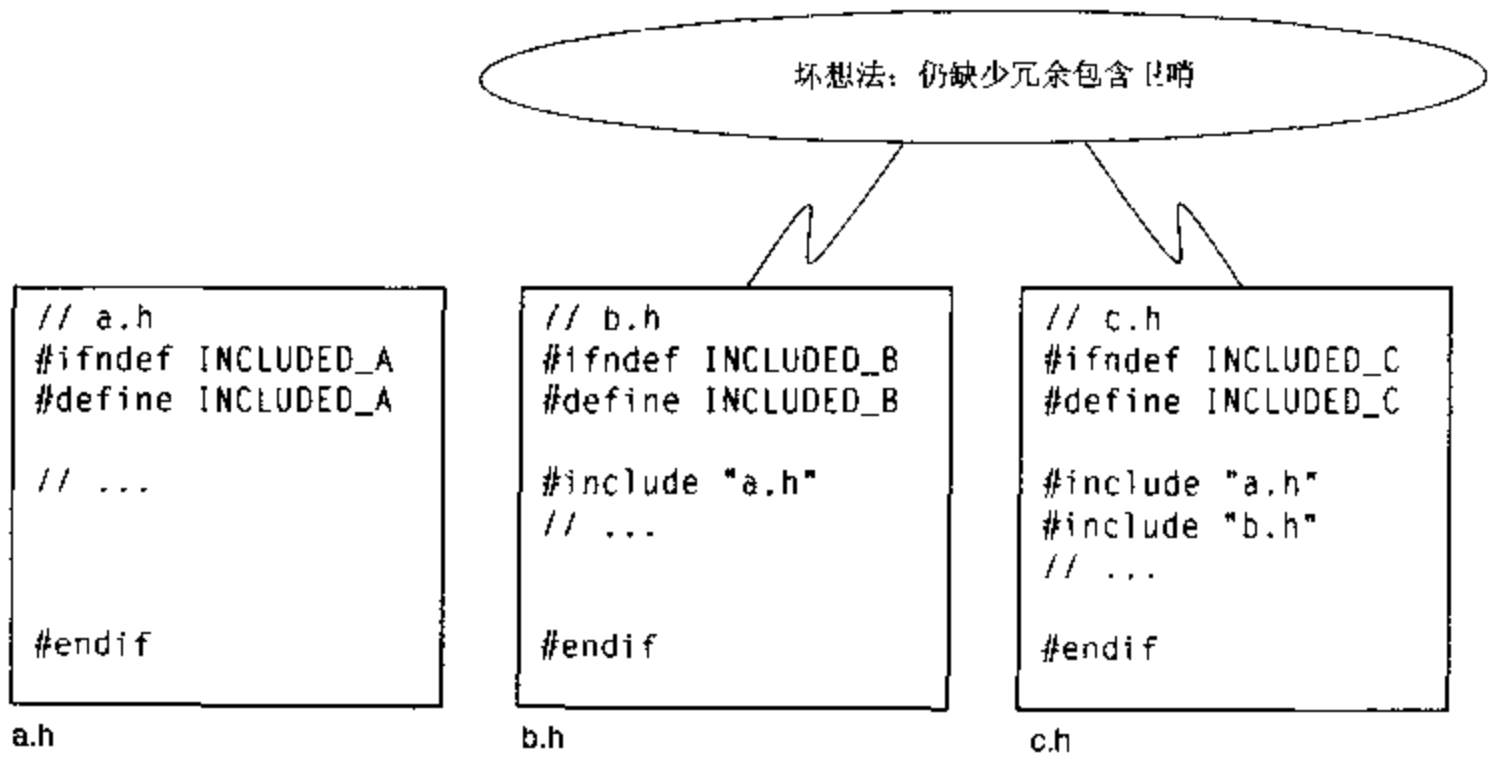


图 2-6　使用包含卫哨提供重复的包含

采用标准的命名习惯，即在头文件的大写根名称（如，STACK）前加上全局保留前缀（如，INCLUDED_），可以确保卫哨名称是惟一的并且可预知的：

```
// stack.h
#ifndef INCLUDED_STACK
#define INCLUDED_STACK
// ...
#endif
```

```
// iccad_transistor.h
#ifndef INCLUDED_ICCAD_TRANSISTOR
#define INCLUDED_ICCAD_TRANSISTOR
// ...
#endif
```

对可预知性的需求将在 2.5 节中讨论。

2.5　冗余包含卫哨

可行的并不总是恰当的，下面要介绍的情况就是一个例子。理论上讲，惟一的内部包含卫哨就足够了。但是，对于大型项目而言，若不做进一步的考虑，则开销可能会很大。

一个设计良好的系统由多层抽象构成。只要可能，我们总是希望先创建少量的基本对象类型，然后在更高的抽象层次上将这些对象组织成新的对象。一个科学的应用可以将各种不同种类的原子建模为对象。在元素周期表中有 100 多种原子。这些相对较少的基本类型的实例以不同的方式和比例通过分层构成宇宙中各种不同类型的分子。

分层设计的另一个例子可能是一个面向对象的窗口系统。假设我们有一个 N 个基本窗口部件的集合（如按钮、刻度盘、滑动开关、显示控件等）。为简洁起见，我们将这些基本窗口部件类命名为 W1、W2、…、W*n*。每个窗口部件 Wi 存在于自己的独立编译单元 wi.c 中，带

有相应的头文件 wi.h。可以通过组织各种类型的部件对象来创建新的屏幕类型。我们将这 M 个屏幕类称为 S1、S2、…、S*m*，每一个类 Si 都存在于自己的独立编译单元中，带有头文件 si.h。

一般来说，每个屏幕会使用相当数量的可用窗口部件。为讨论方便，假设每个屏幕类型都充分使用了所有的（或者绝大部分的）基本类型，这样就促使实现者在每个 s1.h 文件中包含所有的 w1.h、w2.h、…、w*n*.h 文件。一个典型的屏幕头文件 s13 如图 2-7 所示。

```
// s13.h
#ifndef INCLUDED_S13
#define INCLUDED_S13

#include "w1.h"
#include "w2.h"
#include "w3.h"
// ...
#include "wn.h"
#include <math.h>

class S13 {
    W1 d_w1a;
    W1 d_w1b;
    W2 d_w2;
    W3 d_w3;
    // ...
    Wn d_wn;
};

#endif
```

图 2-7 由许多部件组成的典型的屏幕（Screen）

读者是否已经看出了一个潜在的问题？让我们继续讨论。假设读者研制了许多 screen，并且在系统的某个编译单元 ck.c 中，读者需要包含所有的 screen 头文件（假定要创建它们）。一个窗口应用程序的包含图如图 2-8 所示，该图有 *N*=5 个窗口部件和 *M*=5 个 screen。

当预处理器看到 ck.c 已包含 s1.h 时，预处理器会包含 w1.h 到 w5.h。当遇到 s2.h 时，在寻找结束符#endif的整个过程中，部件的头文件仍然必须重新打开并逐行地重新处理（只是要查明没有其他事情需要做）。这种冗余的处理对于 s3.h、s4.h 以及 s5.h 都会发生。尽管该程序也能被编译并正常地工作，但是在只用 5 个部件头文件就能工作的情况下，我们却不得不等候处理 25 个部件头文件。

除非特别小心，否则 C++倾向于拥有大型的、高密度的包含图（比 C 多很多）。尽管继承和分层有助于解决该问题，但是根本的原因是部分 C++程序员经常被这样的信念所误导：通过在客户的头文件中包含每一个客户可能需要的其他头文件的方式来帮助客户。

```
// ck.c
#include "ck.h"
#include "s1.h"
#include "s2.h"
#include "s3.h"
#include "s4.h"
#include "s5.h"
```

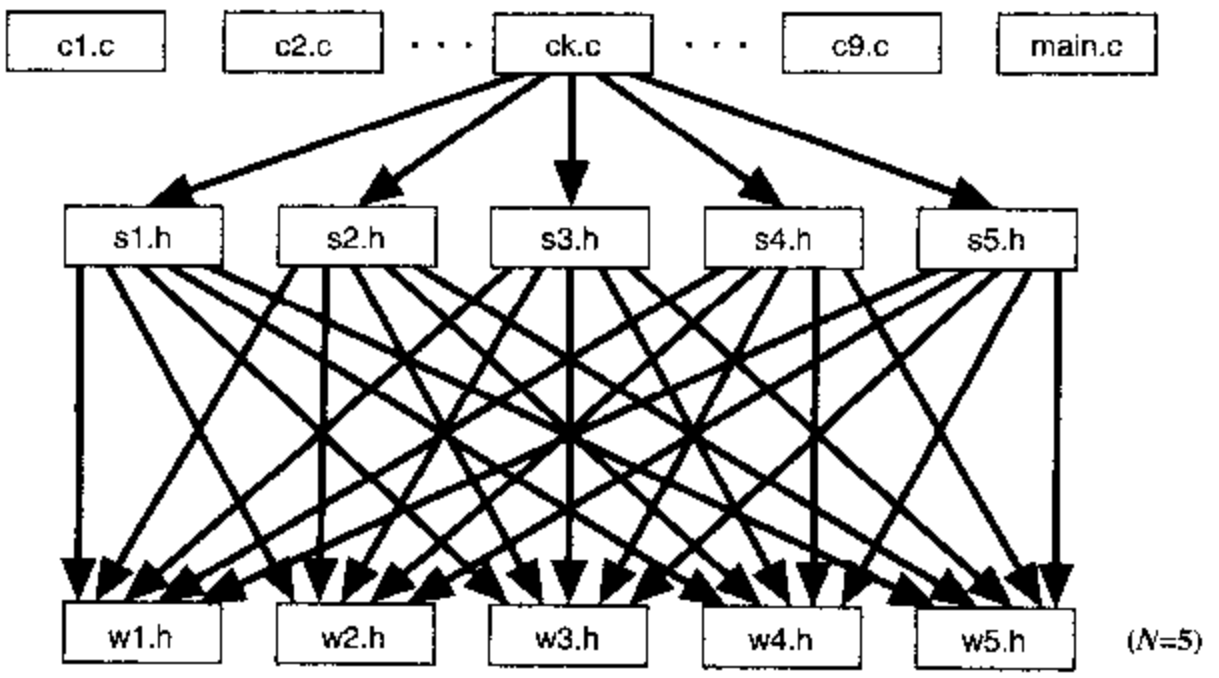


图 2-8　规模 N=5 的窗口系统中的一个组件的包含图

避免高密度的包含图是“绝缘”论题的一部分，将在第 6 章中讲述。下面介绍一个规程，即使在一个不良设计中，遵守该规程也可以减少这样的再收敛包含的影响。注意，有一些开发环境非常智能，可以跟踪记录以前包含的头文件，但是，大多数普通的环境则不能。如果可移植性是一个问题，则最好先保证安全而不要留下遗憾。

次要设计规则

在每个头文件预处理器的包含指令周围，放置一个冗余（外部）包含卫哨。

在每一个出现在头文件中的预处理器包含指令周围放置一个冗余（外部）包含卫哨，这项技术应用于一个典型的 screen 头文件的情况如图 2-9 所示。第一次处理文件 S13.h 时仍然要包含 w1.h、w2.h、…、wn.h。但是，包含另一个 screen 时，不会引起部件头的任何冗余的解析。

注意，数学标准库的头文件的冗余包含卫哨与其他的不同。尽管 math.h 确实有自己的内部包含卫哨，但是它可能不遵守我们的标准。由不同的编译器提供的运行时库，很可能对它们所使用的包含卫哨应用不同的命名规则，并且这些包含卫哨的名称可能不总是一致的。由第三方供应商提供的组件还可能使用另一种规则。对于所有不能保证遵守我们的包含卫哨命名规则的组件，有必要在相应的包含指令之后（如 math.h 中使用的那样）添加一行，用以定义适当的包含卫哨符号。

使用冗余包含卫哨确实令人不快。使用冗余包含卫哨时，在一个头文件中包含一个文件头不只需要一行代码，而是要占三行。若被包含的文件头来自影响范围之外，则要占用四行。冗余包含卫哨不仅使头文件更长，而且使头文件难以阅读。使用冗余包含卫哨还要求遵守一致和可预知命名规则，值得吗？

具有高密度包含图的真正大型项目的开发经验表明，上述问题的回答完全是肯定的。由几百万行 C++源代码构成的项目最初结构，使用大型工作站网络来进行编译要用一星期的时间。插入冗余包含卫哨后，在没有对代码进行本质修改的情况下，能显著地减少编译时间。

```
// s13.h
#ifndef INCLUDED_S13
#define INCLUDED_S13

#ifndef INCLUDED_W1
#include "w1.h"
#endif

#ifndef INCLUDED_W2
#include "w2.h"
#endif

#ifndef INCLUDED_W3
#include "w3.h"
#endif

// ...

#ifndef INCLUDED_WN
#include "wn.h"
#endif

#ifndef INCLUDED_MATH
#include <math.h>
#define INCLUDED_MATH          // extra line
#endif

class S13 {
    W1 d_w1a;
    W1 d_w1b;
    W2 d_w2;
    W3 d_w3;
    // ...
    Wn d_wn;
};

#endif
```

图 2-9 带有冗余包含卫哨的典型的 Screen 组件

一般来说，我们刚刚讨论的问题对于小型甚至中等规模的系统都不是问题。但是，如果我们正在处理相当于包含有数百个基本部件和数百个基本 screen 的问题时，会发生什么事呢？为了提供量化信息以说明使用冗余包含卫哨的好处，我们进行了如下试验。

我们将 *N* 作为部件和 screen 的数量。然后生成子系统并且对单个编译单元测量编译时间（该编译时间主要是C预处理器时间），包括所有带有和不带有冗余包含卫哨的screen头文件。我们对 10 行的头文件和 100 行的头文件进行了试验，将加速因子（speedup factor）定义为：没有带冗余包含卫哨的编译时间除以带冗余包含卫哨的编译时间。结果如图 2-10 所示。

	10 行/头文件			100 行/头文件		
	CPU 秒		加速	CPU 秒		加速
N	没有	有	因子	没有	有	因子
1	0.2	0.2	1.00	0.2	0.2	1.00
2	0.2	0.2	1.00	0.2	0.2	1.00
4	0.3	0.3	1.00	0.4	0.3	1.33
8	0.5	0.3	1.67	0.7	0.4	1.75
16	0.7	0.4	1.75	1.7	0.5	3.40
32	1.5	0.5	3.00	5.8	0.9	6.44
64	5.8	1.1	5.27	22.1	2.0	11.05
128	25.9	3.5	7.40	89.5	5.2	17.21
256	126.5	13.6	9.30	376.5	17.1	22.02
512	702.3	61.6	11.40	1697.4	68.6	24.74
1024	4378.5	306.6	14.28	8303.8	330.6	25.12

图 2-10　带与不带冗余包含卫哨的预处理器时间

对于少于 8 个部件和 8 个 screen 的系统，要么不存在加速，要么加速很小。但是，若总的编译时间小于 1CPU 秒，则无关紧要。

在 C++中，头文件很少只有 10 行；100 行的也还算是小的，但是比较典型。对于一个有 32 个部件的系统来说，在作者的机器上花费在 C 编译器上编译每个客户端组件的时间可以缩短，加速因子超过 6（从 5.8 到 0.9CPU 秒）。对于一个有 64 个部件的系统来说，加速因子超过 11!冗余包含卫哨很难看，但是没有真正的危害。而不使用冗余卫哨则要冒编译时间复杂度为二次方（即 $O(N^2)$）的危险。

注意，冗余卫哨对于.c 文件不是必要的。如果在.c 文件中未能谨慎地复制#include 命令，对于 *N* 个不同的.h 文件，（不合理的）最坏情况下的性能为 $2N$，仍然是线性的（即 $O(N)$）。

本节中的数据反映的是运行于基于 Unix 工作站上的 CFRONT。其他开发环境可能有所不同。在第 6 章中，读者将了解到在头文件中使用嵌套的#include 命令不仅不是所希望的，而且经常是不必要的。冗余包含卫哨的丑陋性提醒我们，应该避免将#include 指令放在头文件中（在这样做有意义的任何时候）。

2.6 文档

本书中的例子并不是为了给产品代码树立什么是足够注释的好榜样（否则，本书就不是一本，而是二本了）。但是，注释，尤其是接口中的注释，是开发过程中的一个重要组成部分。

指导方针

为接口建立文档以便其他人可以使用，至少请另一个开发者检查每个接口。

想要理解为什么让别的开发者检查接口是有价值的，可以假设你自己就是试图理解你的类的客户或测试工程师。你自己非常了解如何使用接口——毕竟是你自己设计的！你用来给成员函数命名的简洁名称是“明显的”和“自解释的”。但除非你已花时间让别人检查了你的接口和文档，否则一定有很多需要改进的地方——特别是在它的可用性方面。

可用性主要是指拿到一个不熟悉的头文件就可以开始使用它。在实践中，头文件注释经常是与接口相关的仅有的文档（或者至少是惟一的更新过的文档）。如果客户必须被迫查看实现以便能够领会到如何使用组件，那么该文档就是不合适的。

指导方针

明确地声明条件（在该条件下行为没有定义）。

文档的另一个重要方面是明确地确定行为没有定义的条件。考虑下面的声明：

```
struct MathUtil {
    // ...
    static int fact(int n);
        // Returns the product of consecutive integers between 1 and n.
};
```

你认为函数 fact 的注释怎么样？我们可能猜测 fact 应该是普通数学函数 factorial($n!$)，fact(0)实际上是 1 而不是 1·0=0 或者没有定义。但是，这不是该注释的意思。该注释没有说明这样的意思：当 n 是非正数时，情况会如何。

阶乘对于负整数值是没有定义的。在这种情况下，我们的特定实现将返回 0。当 n 的值是负数时，fact(n)返回什么值是一种实现技巧，而不是规范的一部分。必须明确告知客户不能依靠这种行为。另一种替代该实现的实现很容易对 n 的负值（包括引起程序崩溃的值）提供不同的行为。

除非明确地在注释中声明，否则客户和测试工程师一般来说没有办法区分什么是特意设计的或必需的行为，什么是仅由特定实现选择引起的巧合行为。一个较好较实用的接口如下：

```
struct MathUtil {
    // ...
    static int factorial(int n);
        // Returns the product of consecutive integers between 1 and n
        // for positive n.  If n is 0, 1 is returned.
        // Note that the behavior is not defined for negative values
        // of n nor for results that are too large to fit in an int.
};
```

没有明确地规定未定义行为的条件，会不经意地使得软件支持无关的行为，这种行为会影响性能或限制实现的选择。如果一个测试工程师不是很内行，读者可能会发现自己的实现选择产生的无关行为会被一套可以明确测试那些行为的回归测试无情地击中。更糟糕的是，通过不适当地（或无意识地）使用，客户可能会依赖这种巧合的行为。

原　则

使用 assert 语句有助于为程序员实现编码时的假设建立文档。

对系统的每一层次进行错误检查，以便找出逻辑错误，这样做代价很高，对于大型系统来说尤为如此。好文档是编写额外代码的可行的替代方案。例如，有些软件开发者认为有必要处理进入函数的每一个指针，即使那个指针为空。如果某个函数是一个广泛使用接口的一部分，那么支持鲁棒性将被证明是一种好的决策。另一个选择是，只要使客户清楚传递一个空指针将导致无定义就够了，可以在函数实现的开始用一个 assert 语句来实现：

```
// stdio.c
#include <stdio.h>
#include <assert.h>
/* ... */
int printf(const char *format ...)
{
    assert(format);
    /* ...
    */
}
```

文档和 assert 语句的有效使用可以使我们得到更简练但仍十分有用的代码。如果有人误用了某个函数，这是他们自己的错——而且他们很快就能发现这个错误！

如果每个开发者总是很清楚函数的指针参数不能为空，这是值得称赞的。但是，负责任的客户不应该假设指针参数可以为空，除非结果行为是明确声明的。

2.7 标识符命名规则

维护一个大型系统时，能以一种一致的、可客观证实的方式区分数据成员、类型和常量

名称与其他标识符可能是一种十分重要的优点。1.4.1 节给出了一个命名规则的集合，在这里我们将用三个设计规则和两个指导原则简洁地将其区分开。

标识类数据成员的词典编纂原则可以简洁地表达出来，它的价值不仅仅是风格问题。因此，将这种原则作为设计规则来介绍。

次要设计规则

使用一种一致的方法（例如加前缀 d_）来突出类成员函数。

读者也可选择使用 s_来区分静态数据和实例数据。上述规则是一条次要规则，因为客户程序决不是一定要处理这类问题（因为根据 2.2 节，数据成员应该总是私有的）。

次要设计规则

使用一种一致的方法（例如第一个字母大写）来区别类型名称。

上述原则被作为一条规则而不是一个指导方针，因为它是一个已被广泛接受并且可客观检验的标准，一般可提高可读性，它使得接口更容易理解，代码更容易维护。它是一个次要规则，因为一个单独的失误并不代表世界末日。

次要设计规则

使用一种一致的方法（例如所有的字母都大写以及使用下划线）来标识不变的值，如枚举值、const 数据和预处理器常量。

上述规则可以帮助我们把常数（因而是“无状态的”）变量与局部变量和成员（状态）变量区别开来。它被作为一个设计规则而不是一个指导方针，因为它有助于提高可维护性，它是客观可验证的，并且它要求没有例外。

指导方针

标识符名称必须一致；使用大写字母或下划线（但不能同时用两种）来分隔标识符中的单词。

上述原则是客观可验证的，但不是每个人都能确信它的优点，并且它很大程度上是一种风格问题。它的用处是可使标识符名称在某种程度上更易于记忆，并可对多数客户展示一种更专业化的形象。在这里它是作为一条指导方针提出来的（尤其对于接口），但在实现中也容许存在某种程度的个性。（在本书中，我们采用大写字母标准）。

指导方针

以相同方式使用的名称必须一致；特别是对于递归设计模式（recurring design patterns）（例如迭代），要采用一致的方法名称和运算符。

在一个大型系统的整个接口中获得一致性，可以增强可用性，但要达到此目的也可能会遇到出人预料的困难。在大型项目中，授权一个顶级开发小组担当“接口工程师”已被证明是有效的，可以跨越开发小组获得一致性。容器类以及它们的迭代器也有助于模板的实现（见10.4节），实现模板可有效地加强跨越其他无关对象的一致性。

2.8 小结

C++是一种大型语言，为更大的设计空间开辟了道路。本章我们描述了基本设计规则和指导方针的一小部分，这些规则和指导方针在实践中被证明是非常有用的。

主要设计规则被认为是绝对不能违反的。甚至偶尔的违反也可能危及大型系统的完整性。在本书中，我们自始至终遵守了所有的主要设计规则。

次要设计规则也被认为是要遵守的，但也许不必严格地遵守。在一个隔离的实例中违反一个次要规则不大可能产生严重的全局性影响。

指导方针是作为经验法则提出来的，因此必须遵守，除非有强制性的工程方面的原因要求遵守别的原则。

把一个类的数据成员暴露给其客户程序违反了封装原则。提供对数据成员的非私有访问意味着表示上的局部改变可能迫使客户重新编写代码。此外，由于允许对数据成员进行可写访问，无法阻止偶然误用导致数据处在不一致的状态。保护的成员数据就像公共成员数据一样，无法限制因数据改变而可能影响到的客户的数量。

全局变量会污染全局名称空间，而且会歪曲设计的物理结构，使得实际上不可能进行独立的测试和有选择的重用。在新的C++项目中没有必要使用全局变量。我们可以通过将变量放置在一个类的作用域中作为私有静态成员、并提供公共静态成员函数访问它们的方法来系统地消除全局变量。但是，对这种模块的过度依赖是一种不良设计的症状。

自由函数，特别是那些不在任何用户自定义类型上操作的函数，在系统集成时很可能与别的函数冲突。将这样的函数嵌套在类作用域中作为静态成员基本上可以消除冲突的危险。

枚举类型、typedefs以及常量数据也可能威胁全局名称空间。通过将枚举类型嵌套在类作用域中，任何二义性都可以通过作用域解析来消除。一个在文件作用域中的typedef看起来有点像类，但是在大型项目中极难发现。通过将typedef嵌套在类作用域中，它们就变得相对容易追踪。一个在头文件中定义的整数常量，其最好的表达方式通常是通过在类作用域中的一个枚举值来表达。其他常量类型可以通过使它们成为某个类的静态常量成员来限定其范围。

预处理宏对于人和机器来说都难以理解。由于宏不是 C++的一部分，所以宏不遵守作用域约束，并且，如果将宏放置在一个头文件中，宏可能与系统中的任何文件的任何标识符冲突。因此，宏不应该出现在头文件中，除非是作为包含卫哨。

总的看来，我们应该避免在一个头文件的文件作用域中引入除了类、结构、联合和自由运算符之外的任何东西。当然，我们允许在头文件中定义内联成员函数。

一个定义被包含两次会引起编译时错误。因为大多数 C++头文件包含定义，我们有必要防止再收敛包含图的可能性。在一个头文件中，用内部包含卫哨围绕定义可以确保每个头文件的内容在任何一个编译单元中最多被加入一次。

冗余（外部）包含卫哨虽然不是一定必需的，但是它可以确保我们避免编译时的二次包含行为。通过用冗余卫哨围绕头文件中的包含指令，我们可以确保每个编译单元最多两次打开一个头文件。

良好的文档是软件开发必不可少的一部分。缺少文档将降低可用性。文档的一个重要部分是声明什么是没有定义的。否则，客户可能会依赖巧合的行为，这种行为只能来自特定的实现选择。

不是所有的代码都必须是鲁棒的。在系统每个层次上的冗余的运行时程序错误检查，可能对性能产生无法接受的影响。文档和断言（assertion）的结合可以达到同样的目的，但在最终产品里可以获得更优越的运行时性能。

最后，提供一个一致的命名规则的集合来区别数据成员、类型和常量，可以提高跨越开发小组的可读性。我们建议对数据成员使用一个“d_”前缀（如果是静态的再加上“s_”）；用第一个字母大写来表示一个类型，而用小写的字母表示一个变量或函数。用所有的字母都大写（包括下划线）表示枚举值、const 数据和预处理器常量；用第一个字母大写来分隔多单词标识符。我们也建议为递归设计模式（如迭代器）使用一致的名称。

第2部分

物理设计概念

用 C++开发大规模软件系统需要的不仅仅是对逻辑设计问题的可靠理解。逻辑实体，例如类和函数，就像一个系统的皮肉。构成大型 C++系统的逻辑实体分布在许多物理实体中，例如文件和目录。物理体系结构是系统的骨架——如果它是畸形的，没有任何美容疗法可以减轻它令人讨厌的症状。

一个大型系统的物理设计质量，将决定其维护开销以及它所具有的让其子系统独立重用的潜能。要进行有效的设计需要彻底掌握物理设计概念，尽管这些概念与许多逻辑设计问题紧密联系在一起，但是对于这些物理设计概念的某一方面，甚至老练的专业软件开发人员都可能缺少经验或者没有经验。本书的第 2 部分全面介绍了优秀物理设计的基本概念。

第 3 章介绍了作为设计基本单位的组件。本章还介绍了若干物理设计规则，以确保我们所有的设计都具有可靠的、重要的和合乎需要的特性。许多逻辑设计关系（例如，IsA、HasA、Uses）都压缩为一种物理关系：**依赖（DependsOn）**。我们会介绍逻辑设计决策如何能够潜在地影响物理依赖。我们也会介绍怎样从一个已存在的组件集合中有效地提取物理依赖。

第 4 章描述了物理层次结构（即，分层）对于开发、维护和测试的重要性。在这一章中我们要探讨如何根据物理依赖来刻画单个的组件、子系统以及整个系统的特性。我们还会介绍如何开发可靠物理设计的层次结构，通过隔离的、增量式的和分层次的测试，以较低的开销来获得较高的可靠性。本章还定量分析了系统中的物理依赖在连接时间和磁盘空间方面对维护开销和回归测试的影响。

第 5 章研究了许多过度连接时依赖的常见原因。这一章罗列了减少一个系统中的连接时依赖的若干技术和变换——一个在本书中称为**层次化（levelization）**的过程。我们使用了许多选自不同的应用程序的实例来说明这些技术。

第 6 章研究了与过度编译时耦合有关的维护开销。这一章展示了若干常用语言结构，这些语言结构会迫使客户依赖于封装好的实现细节；介绍了减轻或消除对个别细节的编译时依赖的技术，以及使客户远离实现的一整套技术——一个在本书中称为**绝缘（insulation）**的过程。最后，以与绝缘有关的运行时开销作为度量标准，讨论了不适当使用绝缘的情形。

第 7 章把层次化的概念扩展到超大型系统。需要超出单独组件物理结构之外的额外物理结构来支持这种系统的复杂功能。**包（package）**代表了一种在物理上内聚的可协同运作组件的集合，并且提供的物理抽象的层次比单独使用组件时所能达到的物理抽象层次更高。在本章中，在把软件包作为一个整体的上下文中我们再次用到了层次化和绝缘的概念。我们也谈到了属于开发和发布一个超大型系统的稳定快照过程的问题。最后，我们讨论了面向对象系统中 main()的角色以及不同初始化策略的相对优点。

3 组件

本章介绍物理设计（与逻辑设计对应）的概念，以组件作为基本的设计单位来介绍。随后研究物理规则的一个子集，它能在大型设计中确保重要的合乎需求的特性。紧接着讨论组件之间的 DependsOn 关系，以及在设计时如何从抽象的逻辑关系中推断出这种关系（DependsOn 关系）。我们还将介绍怎样通过检查组件之间的#include 图表来有效地跟踪物理依赖。最后，我们研究在组件内部和外部对友元关系进行授权的微妙物理含义。

3.1 组件与类

逻辑设计强调系统内部定义的类和函数的交互作用。从纯粹的逻辑角度来看，一个设计可以看作是类和函数的海洋，那里没有物理分区存在——每一个类和自由函数都驻留在一个单一的无缝空间里。交互式的面向对象语言如 Smalltalk 和 CLOS 有丰富的适合于单个开发者的运行环境，无疑已经助长了这种类和自由函数都驻留在单一空间的观点。

但是逻辑设计只考虑了设计过程的一个方面。逻辑设计不考虑像文件和库这样的物理实体。编译时耦合、连接时依赖以及独立重用等都不是仅仅通过逻辑设计就能解决的。例如，一个函数无论是否声明为内联（inline）都不会影响它所要完成的任务，但是可能会极大地影响到它的一些可显式测量的特性，如运行时间、编译时间、连接时间和可执行程序的大小。如果不从物理角度来考虑设计，就不可能考虑到在设计超大型系统时变得很重要的组织问题。

原　则

逻辑设计只研究体系结构问题；物理设计研究组织问题。

物理设计集中研究系统中的物理实体以及它们如何相互关联的问题。在大多数传统的 C++程序设计环境中，系统中每一个逻辑实体的源代码都必须驻留在一个物理实体中，一般

称之为一个文件。基本上每一个 C++程序都可以描述为一个文件的集合。这些文件中有一些是头文件（.h），而有一些则是实现文件（.c）。对于小型程序，这种描述足够了。但对于大一些的程序，我们需要利用额外的结构来生成可维护、可测试和可重用的子系统。

定义：一个组件（component）就是物理设计的最小单位。

组件不是类，反之亦然[1]。从概念上来讲，一个组件包含一组逻辑设计的子集，这些逻辑设计使组件作为一个独立的、内聚的单位存在时是有意义的。类、函数、枚举等等都是构成这些组件的逻辑实体。特别地，每个类定义都严格地只驻留在一个组件中。

在结构上，组件是一个不可分割的物理单位，没有哪一部分可以独立地用在其他的组件中。一个组件的物理形态是标准的，并且独立于它的内容。一个组件严格地由一个头文件（.h）和一个实现文件（.c）构成[2]。

一个组件一般会定义一个或多个紧密相关的类和被认定适合于所支持的抽象的任何自由运算符。像 Point、String 和 BigInt 这样的基本类型，每一个都会在包含一个单一类的组件中实现，如图 3-1（a）所示。像 IntSet、Stack 和 List 这样的容器类，一般会在至少包含原理类（principle class）和它的迭代器（iterator）的组件中实现，如图 3-1（b）所示。涉及多种类型的更复杂的抽象（如 Graph）可以在单个的组件中具体化为若干个类，见图 3-1（c）所示。最后，为整个子系统提供包装器（wrapper）的类（见 5.10 节）可以构成薄薄的、由一个或多个原理类和许多迭代器组成的封装层，如图 3-1（d）所示。

图 3-1 中的每一个组件（像每一个其他的组件一样）都既有物理视图也有逻辑视图。物理视图由.h 文件和.c 文件组成，.h 文件被包含在.c 文件中，作为.c 文件的第一行有效语句。组件的物理实现在编译时总是依赖它的接口。这种内部物理耦合要求把这两个文件当作一个单一的物理实体来对待。

原 则

一个组件就是设计的适当的基本单位。

一个组件（不是一个类）是逻辑和物理设计的适当的基本单位，至少有三个理由：

（1）一个组件把便于管理的一定数量的内聚功能（常常跨越多个逻辑实体，例如类和自由运算符）塞进一个单个的物理单位中。

（2）一个组件不仅把捕捉的整个抽象作为单一的实体，而且允许不通过类级设计来考虑物理问题。

---

① 组件的概念在 **stroustrup**（12.3 节，422~425 页）中介绍。在本章中，我们在那个讨论上进行了扩展，介绍了在 C++中使一个组件的定义具体化的物理设计概念。

② 我们将忽略可能允许一个组件有不止一个.h 文件或.c 文件的特殊环境。

图 3-1 若干组件的逻辑与物理视图

（3）一个设计适当的组件（是一个不像类的物理实体）可以作为一个单一的单位从系统中提出来，不必重写任何代码就可以在另一个系统中有效地重用。贯穿本书，这种既考虑物

理设计问题又考虑逻辑设计问题的要求会变得越来越明显。

作为一个具体的实例，图 3-2 显示了组件 stack 的头文件，该组件包含了两个在文件作用域定义的类，即 Stack 和 StackIter。我们还可以看到有两个自由（即不是成员）运算符函数实现了两个 Stack 对象之间的“==”和“!=”。稍微看一下实现，我们就会发现 operator==使用了 StackIter，而 operator!=是根据 operator==来实现的。图 3-3a 显示了组件 stack 的文件作用域中的完整的逻辑实体集合。物理实体（stack.h 和 stack.c）连同它们规范的物理关系在图 3-3（b）中描述。

我们选择了一个简单的堆栈以保证应用程序功能不会遮盖我们要阐明的要点。在这个例子中，几乎每一个成员都有注释（这是为了使产品代码最小化）。堆栈是一种容器。访问一个堆栈的非头元素通常不会认为是堆栈抽象的一部分。我们已经提供了迭代器来保证在这个 stack 组件中定义的功能，在保持封装的同时能被客户更广泛地扩展（见 1.5 节）[①]。我们没有提到最大堆栈容量，因为一个堆栈抽象没有最大容量。提供诸如 isFull 或一个返回状态（该状态反映由不规范实现强加的人为限制）这样的功能，不仅会干扰抽象，而且会使其应用复杂化。这种意外的、基于实现的限制最好视之为异常。但有时候我们会允许客户“帮助”一个对象预见未来的事件，潜在地改进性能。为了避免暴露一个特定的实现选择，这样的“帮助”——像 C 中的 register 或 C++中的 inline——应该只是一个提示而且无法通过编程检测到（见 10.3.1 节）。

> **定义**：一个组件的逻辑接口就是可被客户通过编程访问或检测到的东西。

组件的逻辑接口是定义在头文件中的类型和功能的集合，它们可以被该组件的客户通过编程访问。因为组织原因而驻留在.h 文件中的私有实现细节将被封装，不被认为是逻辑接口的一部分。

> **定义**：一个组件的.h 文件的文件作用域中定义的任何类或声明的任何自由（运算符）函数，其公共（或保护）接口里如果使用了一个类型，则称在这个组件的接口中使用了这个类型（Used-In-The-Interface）。

一个类的公共接口由那个类的公共成员的接口的并集构成（1.6.2 节），同样的道理，一个组件的“公共”接口也是由该组件的.h 文件中所声明的所有公共成员函数、typedef、枚举类型和自由（运算符）函数的集合构成。

例如，Stack 和 StackIter 的公共成员函数都属于组件 stack 的逻辑接口。自由运算符函数

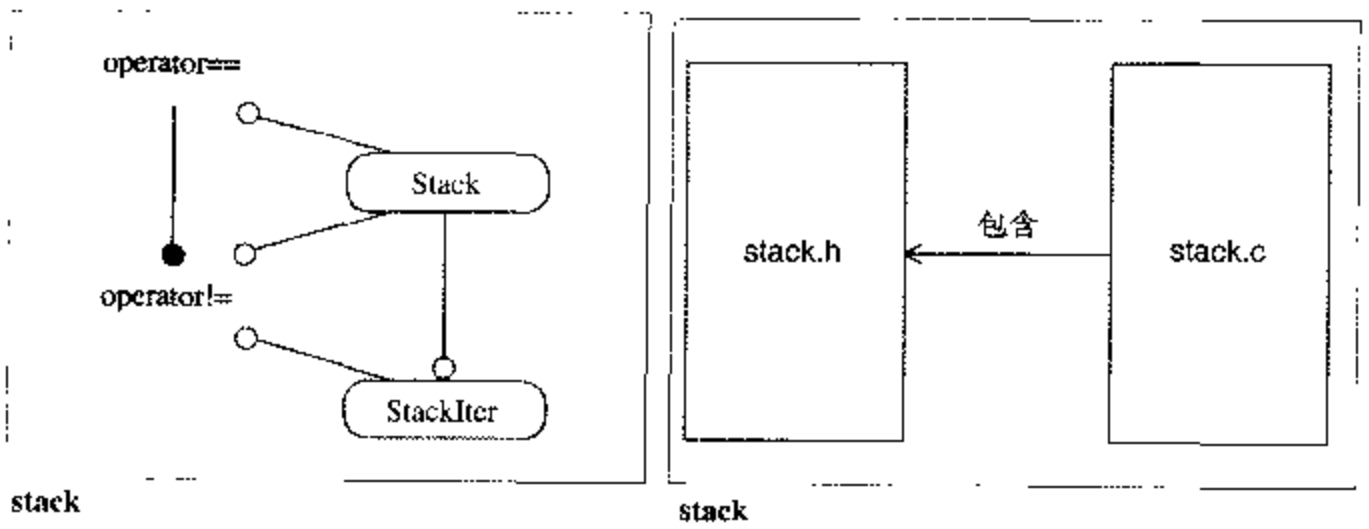
```
operator==(const Stack&, const Stack&)
```

---

① 在所有定义容器类的组件中需要一个迭代器，这是在 **meyer**（2.3 节，23~25 页）中讨论的 open-closed 原理的一个直接结果。

```
// stack.h
#ifndef INCLUDED_STACK
#define INCLUDED_STACK

class StackIter;

class Stack {
    int *d_stack_p;                          // pointer to array of int
    int d_sp;                                // stack pointer (index)
    int d_size;                              // size of current array of int
    friend StackIter;                        // (no comment needed)

  public:
    // CREATORS
    Stack();                                 // create an empty Stack
    Stack(const Stack& stack);               // (no comment needed)
    ~Stack();                                // (no comment needed)

    // MANIPULATORS
    Stack& operator=(const Stack& stack);    // copy Stack from Stack
    void push(int value);                    // push integer onto this Stack
    int pop();                               // pop integer off this Stack
                                             // undefined if Stack empty
    // ACCESSORS
    int isEmpty() const;                     // 1 if empty else 0
    int top() const;                         // integer on top of this Stack
};                                           // undefined if Stack empty

int operator==(const Stack& lhs, const Stack& rhs);
    // 1 if two stacks contain identical values else 0

int operator!=(const Stack& lhs, const Stack& rhs);
    // 1 if two stacks do not contain identical values else 0

class StackIter {                            // iter order: top to bottom
    int *d_stack_p;                          // points to orig. stack array
    int d_sp;                                // local stack pointer (index)
    StackIter(const StackIter&);             // not implemented
    StackIter& operator=(const StackIter&); // not implemented

  public:
    // CREATORS
    StackIter(const Stack& stack);           // initialize to top of Stack
    ~StackIter();                            // (no comment needed)

    // MANIPULATORS
    void operator++();                       // advance state of iteration
                                             // undefined if done
    // ACCESSORS
    operator const void *() const;           // non-zero if not done else 0
    int operator()() const;                  // value of current integer
};                                           // undefined if done

#endif
```

图 3-2 组件 stack 的头文件 stack.h



（a）逻辑视图 （b）物理视图

图 3 3 组件 stack 的两种视图

不是 Stack 的成员，因此不被认为是类 Stack 的逻辑接口的一部分。但是，这个运算符会扩展组件 stack 中定义的可编程访问的函数集合，因而也会扩展组件的逻辑接口。这个有关友元关系的微妙的问题在 3.6 节讨论。

图 3-4 显示了一个小型驱动程序 stack.t.c，它创建了一个 Stack 对象，在其中压入了一些整数，然后按照从头到尾的顺序把它的内容打印出来。这个驱动程序可以访问 stack 组件的逻辑接口，但不能访问它的组织结构。然而，stack 组件的.h 文件中有比可编程访问的信息更多的信息。

```
// stack.t.c
#include "stack.h"
#include <iostream.h>

main()
{
    Stack stack;
    stack.push(111);
    stack.push(222);
    stack.push(333);

    for (StackIter it(stack); it; ++it) {
        cout << it() << endl;
    }
};

// Output:
//      333
//      222
//      111
```

图 3-4 组件 stack 的驱动程序 stack.t.c

定 义：一个组件的**物理接口**就是它的头文件中的**所有东西**。

一个组件的物理接口由.h 文件中可利用的所有信息构成（不考虑访问特权）。组件的.h 文件中包含的信息越多，对组件实现的修改就越有可能影响组件的客户程序而导致它们要重编译。

任何程序员看到 stack.h 就可以断定这是一个基于数组的堆栈（它还未实现，例如，实现为一个链表）。一个编译器要完成工作，在整个过程中都必须查看 stack.h。编译器甚至必须考虑私有信息（例如，d_stack_p），从逻辑角度来看，这些信息确实是一个实现细节。这种物理暴露的结果是，一个未触及 stack 组件接口逻辑视图的实现变化仍然可能强制包含 stack.h 的所有客户重新编译。

作为程序员，我们观察到在 stack.h 中没有函数声明为内联（inline）。修改这个组件中的任何函数体都不会因此改变物理接口而迫使客户程序重新编译。不利的方面是对于像 Stack 这样的轻量级对象，删除内联函数可能导致运行时性能方面的一个数量级的损失（见 6.6.1 节）。

定 义：如果在一个组件的任何地方通过名称引用了一个类型，则称在**这个组件的实现中使用了这个类型（Used-In-The-Implemention）**。

从逻辑角度看，在组件的实现中有或没有使用什么都是封装细节，并不重要。从物理角度看，这样的使用可能隐含着对其他组件的物理依赖。在大型系统中正是这些物理依赖会影响可维护性和可重用性。

优秀的设计要求开发者既懂得逻辑设计知识又懂得物理设计知识。逻辑设计是天然的出发点。我们必须考虑哪些逻辑实体应该天然地在一起或者充分相互依赖，因此不能被合理地分割。我们也必须考虑在物理接口上需要暴露多少实现细节。而且我们还必须决定我们的组件会依赖于哪些其他的组件，在这些组件中有哪些变化会对我们自己的组件及其客户程序产生影响。所有这些问题都已经研究解决之后，才正确地设计了一个组件。

3.2 物理设计规则

本节介绍物理设计的基本规则。如果要使我们其他的规程和技术生效，这些规则是必要的。若一个大型设计从一开始就没有从本质上遵循这些规程，那么我们事实上不可能改正它。

主要设计规则

在一个组件内部声明的逻辑实体不应该在该组件之外定义。

这似乎是显而易见的，但是这条规则应该明确地陈述一次。对一个可重用的组件来说，它必须合理地自我包含。一个组件可以有对其他组件的依赖，但是，一个组件在它的头文件

中声明的任何逻辑结构（类声明除外）——如果定义了——则应该全部定义在该组件内部。

图 3-5 是一个如何不把逻辑实体分割成物理单位的例子。类 Stack 已经定义在组件 stack 中，但是它的实现没有限制在组件 stack 内。Stack::push 定义在 intset.c 中，而 Stack::pop 却定义在 main.c 中！

```
// intset.h
#ifndef INCLUDED_INTSET
#define INCLUDED_INTSET
class IntSet {
    // ...
  public:
    // ...
    // ...
    // ...
};
#endif
```

Intset.h

```
// stack.h
#ifndef INCLUDED_STACK
#define INCLUDED_STACK
class Stack {
    // ...
  public:
    // ...
    void push(int i);
    int pop();
};
#endif
```

stack.h

```
// intset.c
#include "intset.h"
// ...
Stack::push(int i)
{
    // ...
}

// ...
```

Intset.c

```
// stack.c
#include "stack.h"
// ...
// ...
// ...
// ...
// ...

// ...
```

stack.c

```
// main.c
#include "intset.h"
#include "stack.h"
int Stack::pop()
{
    // ...
}

// ...
```

main.c

图 3-5 逻辑实体的不合理的物理分割

不能严格遵守上述设计规则除了引起维护麻烦，还可能导致一个设计的许多合乎需求的物理性质的丢失（特别是在其他程序中获得和重用编译单元的能力）。小心谨慎地遵循这条规则将会改善任何规模项目的模块性和可维护性。

次要设计规则

组成一个组件的.c 文件和.h 文件的根名称应该严格匹配。

严格匹配一个组件文件的根名称，对于可维护性来说是重要的。例如，知道 stack.c 和 stack.h 组成一个组件，不仅方便手工维护，而且为简单的面向对象设计自动工具打开了大门（见附录 C）。

但是，一些已存在的对象代码档案文件在对象文件名称上设置了相对较低的字符数限制

（例如：f3）。因此组件的.c 文件的名称不是总能与它的主要类的名称对应。更糟糕的是，一些操作系统限制文件名称只允许有八个字符（加上三个字符的后缀），在开发超大型系统时这可能会是一个相当大的负担。

主要设计规则

每个组件的.c 文件都应该将包含它自己的.h 文件的语句作为其代码的第一行有效的语句。

我们必须把组件的.h 文件包含在其.c 文件中，因为编译器必须先看到一个类成员的声明才能编译它的定义。这个惯例是 C++语言以及许多常用的独立分析工具所要求的。把#include 指令放在文件顶端的原因则有些微妙。

原　则

通过确保一个组件自己分析自己的.h 文件——不要外部提供的声明和定义，可以避免潜在的使用错误。

把包含.h 文件的语句作为.c 文件的第一行，可以确保组件物理接口的固有信息之关键段不会从.h 文件中遗漏（或者，如果有的话，在你试图编译.c 文件时会马上发现它）。

考虑下面组件 wildthing 的头文件：

```
// wildthing.h
#ifndef INCLUDED_WILDTHING
#define INCLUDED_WILDTHING

class WildThing {
    // ...
  public:
    WildThing();
    // ...
};

ostream& operator<<(const ostream& o, const WildThing& thing);
    // Note: uses class ostream in the interface

#endif
```

注意，我们已经重载了左移运算符（<<），所采用的方法对于数据流输出是正常的，也是惯例。下面考虑实现：

```
// wildthing.c
#include <iostream.h>
#include "wildthing.h"
// ...
```

```
ostream& operator<<(const ostream& o, const WildThing& thing)
{
    // ...
}
```

我们尝试编译这个实现，它编译得很好。下一步我们为 wildthing 创建一个测试文件：

```
// wildthing.t.c
#include <iostream.h>
#include "wildthing.h"

int main()
{
    WildThing wild;

    // ...
    // ...

    cout << wild << endl;

    return 0;
}
```

文件 wildthing.t.c 编译和连接了。程序运行得很好，然后我们去告诉所有的朋友我们完成了。但是，这里面有一个错误并且是一个物理错误！下面的程序不能通过编译，为什么？

```
// product.c
#include "wildthing.h"
#include <iostream.h>

int main()
{
    WildThing wild;

    // ...
    // ...

    cout << wild << endl;

    return 0;
}
```

问题出在我们试图在 operator<<（在 wildthing.h 中声明的）的接口中使用类 ostream 时没有事先声明它。客户代码中#include 指令的顺序颠倒了，现在头文件自身不能进行语法分析，因为 ostream 标识符还没有声明。我们怎样处理这个问题呢？

一旦找出了错误，修正就很简单：在第一次使用 ostream 之前将声明“class ostream”[①]

① 不是预处理器指令#include <iostream.h>（如同 6.3.7 节所解释的那样）。

加入 wildthing.h 的文件作用域：

```
// wildthing.h
#ifndef INCLUDED_WILDTHING
#define INCLUDED_WILDTHING

class ostream;          // was missing before, oops!

class WildThing {
    // ...
  public:
    WildThing();
    // ...
};

ostream& operator<<(const ostream& o, const WildThing& thing);

#endif
```

更重要的问题是我们怎样预防此类问题？答案同样简单。始终让每个组件的.c 文件在包含或声明任何别的东西之前包含该组件的.h 文件。通过这种方法，每个组件都能确保它自己的头文件对于编译来说都是能自我满足的。

指导方针

客户程序应该包含直接提供了所需类型定义的头文件；除了非私有继承，应避免依赖一个头文件去包含另一个头文件。

一个头文件是否应该包含另一个头文件是一个物理问题，不是逻辑问题。为了编译，头文件本身需要另一个头文件中的一个定义，在这样的情况下（见 6.3.7 节），把适当的#include 指令（当然，要被冗余外部包含卫哨所包围，如 2.5 节所述）放在那个头文件中是正确的。

但是，除了公共和保护的继承之外，包含一个类型的定义而不是预先在头文件中声明它，这种需求几乎总是由封装的逻辑实现细节来决定。

例如，如果类 Stack 在它的实现中使用了类 MyType，可能有必要在 mytype.h 中包含 stack.h，以确保 mytype.h 的编译。如果决定用 List 代替 Stack 来改变 MyType 的实现，则不再需要 mytype.h 中的#include "stack.h"指令。任何依赖 mytype.h 包含 stack.h 的客户程序现在将不得不修改成直接包含 stack.h。

我们如何将一个类型在另一个类型上进行分层，这将影响编译时耦合的程度。逐渐减少编译时耦合是 6.3 节的主题。例如，使用 MyType HasA（嵌入）Stack 还是 HoldsA（指向）Stack 决定了是 mytype.h 包含 stack.h，还是简单地事先声明类 Stack（见 6.3.2 节）。如果改变了 MyType 的实现，用 HoldsA（代替 HasA）Stack，那么#include 指令可能不再需要写在 mytype.h 中。如果删除了该条指令，那么客户程序也将被迫作出改变（这些客户程序依赖于在 MyType

的实现中使用 Stack 的方式）。即使 MyType 的逻辑接口中使用了 Stack，mytype.h 仍然可能不需要包含 stack.h（见 6.3.7 节）。这就要求每一个实质上使用 Stack 的用户程序都直接包含它的定义。

继承是一个例外，因为继承总是隐含一个编译时依赖，并且是派生类的逻辑接口的一部分。改变继承层次结构（采用允许组件作者从组件头文件中删除#include 指令的任何方式来改变），也将改变逻辑接口，同时将迫使客户程序被再次访问（不考虑物理问题）。因此，客户程序只包含一个派生类的定义，并依赖这个派生类的头文件来包含基类的定义是合理的。

因为类似的原因，客户程序依赖某个组件的头文件来事先声明一个只用在该组件的逻辑实现中的类是不明智的。

主要设计规则

在一个组件的.c 文件中，避免使用有外部连接并且没有在相应的.h 文件中明确声明的定义。

为了分析、维护，尤其是测试，确保某人（或某工具）只看到一个组件的物理接口就能了解那个组件的全部逻辑接口是很重要的。要求一个组件在它的头文件中声明它的完整的逻辑接口有助于提高：

（1）可用性——使客户程序只从接口就可以全面了解一个组件所支持的整个抽象。

（2）可重用性——确保组件提供的所有支持功能对所有客户都是同样可访问的。

（3）可维护性——避免不支持的“后门”接口（它会干扰组件支持的抽象）。

假设某人在组件 foo 的.c 文件中定义了一个外部自由函数（或变量），在 foo.h 中，没有把它作为外部函数（或变量）进行声明。另一个刚好和 foo 相连接的组件 bar，通过局部创建适当的外部声明可以获准访问那个函数（或变量）。图 3-6 中描述了这个不幸的场景。请注意这个例子描述的是一个糟糕的设计（我已尽量避免在本书中展示这种例子）。

如图所示，bar 的.c 文件依赖于 foo 的物理实现提供的定义，但它独立于 foo 的物理接口。这里有一个 foo 的“后门”用法和一个 bar 对 foo 的隐含物理依赖很难被发现。只考虑#include 图的 makefile 的自动依赖产生器（mkmf、gmake 等等）很难找出这种微妙依赖的线索。此外，这种代码的维护者也没有这两个组件耦合的直接证据。可是，当我们去重用 bar 时，在连接阶段会失败，因为函数 f（以及全局变量 size）的定义将会丢失。

如果在 foo.h 中指定完整的接口，客户组件 bar 就可以简单地在它自己的.c 文件中包含 foo.h 文件，使 bar 的实现对 foo 的接口的依赖显性化。这种新的稍有改进的实现在图 3-7 中描述。但是，一个外部全局变量或一个外部自由函数的使用仍然违背了在 2.3.1 节和 2.3.2 节介绍的设计规则。

完全定义在一个组件.c 文件的文件作用域中的类，可能会很容易违背这条规则，因为非内联类成员函数和静态成员数据有外部连接。如果我们对完全定义在一个.c 文件中的类施加

某种约束，我们就可以避免创建外部定义，从而避免违背这条规则。这种约束与 C++语言本身对局部类定义（即完全定义在一个单一函数内的类）[1]所施加的约束是一样的。

坏想法：自由函数和全局变量违反了设计规则

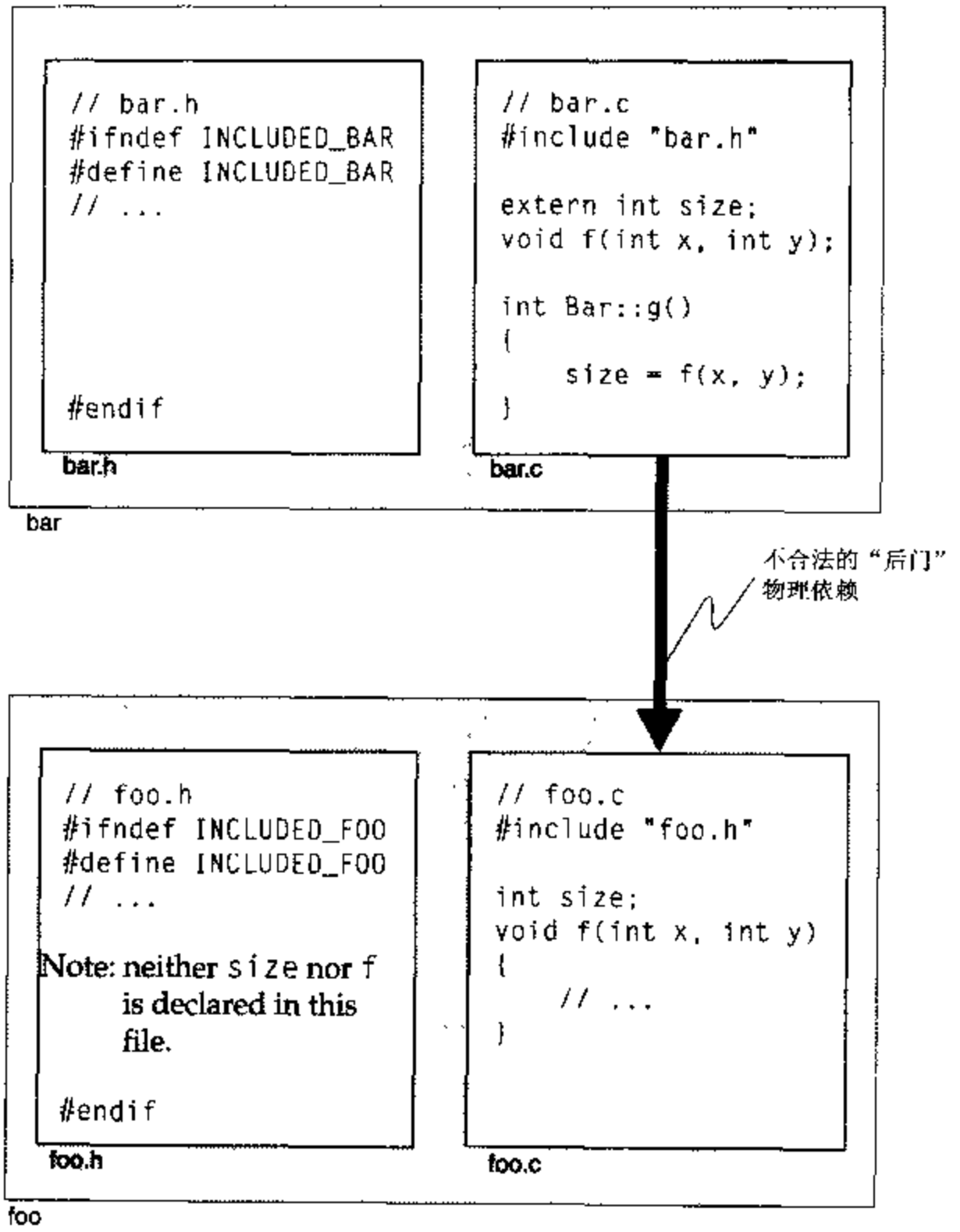


图 3-6　bar.c 直接依赖 foo.c 的糟糕的物理设计

① **ellis**，9.8 节，188~189 页。

坏想法：自由函数和全局变量仍然违反了设计规则

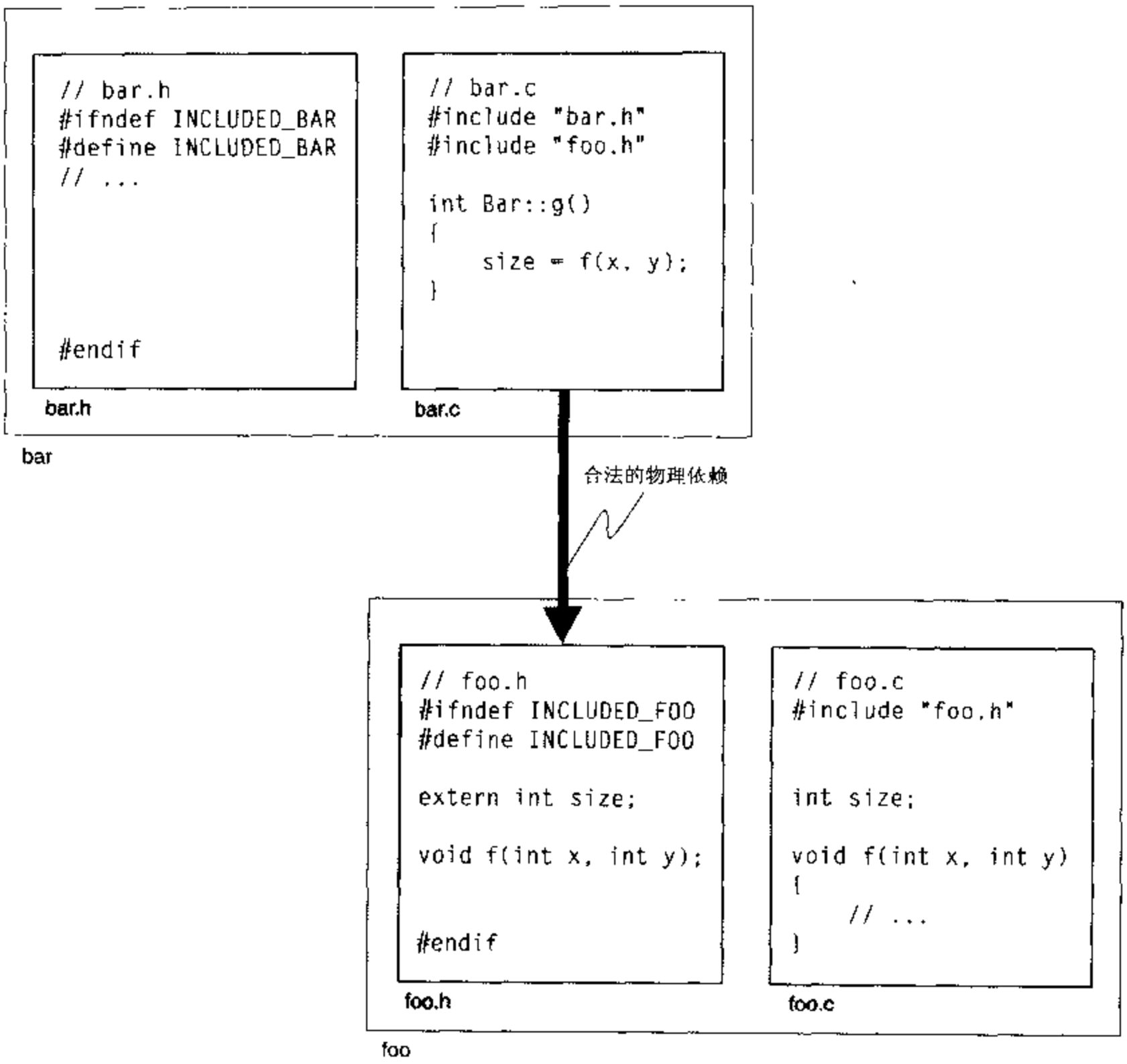


图 3-7 bar.c 依赖 foo.c 的（稍微）好些的物理设计

在一个.c 文件内完整地定义一个类，虽然在技术上违背了一条规则，但是在实践中是相对无害的，因为名字冲突趋向于阻止人们尝试直接使用外部符号。惟一真正的危险是外部定义可能会与一些其他的同样的定义相冲突（如果那个类定义在它自己的单独的组件中仍会是

同样的情形）。避免在一个.c 文件内完整定义类的一个更重要的理由也许是这种类随后不能被直接测试（见 8.4 节）。

避免后门用法对于好的物理设计和有效的重用来说是关键的。只要求组件作者这样做是不够的。要堵上所有的漏洞，我们必须制定一种对大家都有好处的要求：客户程序不能试图通过局部声明来使用任何带有外部连接的结构。相反，客户程序要包含一个组件的.h 文件，以便访问组件所提供的任何定义。

主要设计规则

避免通过一个局部声明来访问另一组件中带有外部连接的定义，而是要包含那个组件的.h 文件。

遵循这条规则的理由主要是要让对其他组件中的外部定义的依赖显性化。

用包含头文件来取代提供一个局部函数声明，对客户也有好处。有时候文件头会改变。你的局部声明将如何改变来反映这些文件头的变化？一个声明错误的有 C++连接的函数至少在连接时就可以被发现[1]，但是来自标准 C 库的不正确的函数局部声明（带有 C 连接），可能直到运行时才能发现。

例如，下面的程序可以编译和连接：

```
// foo.c
#include "foo.h"
extern "C" double pow(double, int); // bad idea: local extern declaration

double Foo::func(double x, double y)
{
    return pow(x, y) + pow(y, x);
}
```

然而，我们在运行时会得到错误的结果，因为局部 extern 声明与目前的 pow 定义不匹配[2]：

```
extern "C" double pow(double, double)
{
    /* ... */
}
```

不匹配的声明将导致 pow 的第二个参数变得混乱不清。我们可以通过包含.h 文件来避免这样的问题并使依赖显性化：

① 见 **ellis** 的 Type-Safe Linkage，7.2c 节，121~126 页。

② **plauger**，第 7 章，138 页。

```
// foo.c
#include "foo.h"
#include <math.h>      // pow()

double Foo::func(double x, double y)
{
    return pow(x, y) + pow(y, x);
}
```

通过包含头文件，与具有任何一种连接特征的函数的不一致之处都将在**编译时**捕获，这样比在连接时或运行时捕获要好得多。

有一件重要的事要说明，以下形式的类声明：

```
class QueueLink;        // forward declaration of class QueueLink
```

是完全不同的问题，因为类定义有内部连接。这样的声明不仅是普遍的而且是合乎需要的，尤其是在它们可以消除头文件中的预处理器#include 指令的地方。类声明的这种用法与连接时依赖有关，将在 5.10 节中讨论；与编译时依赖有关的用法，将在 6.4.3 节中讨论。

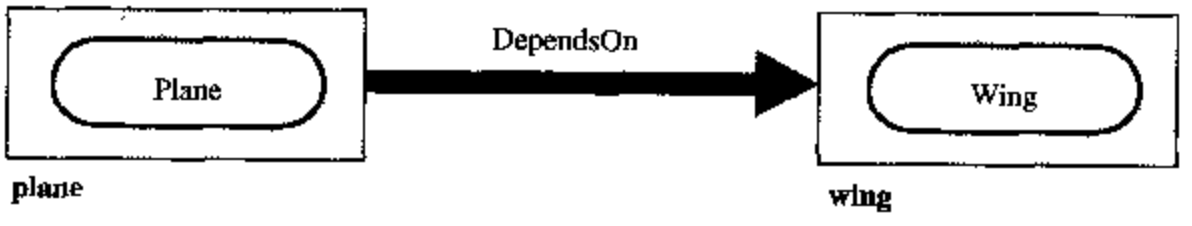
3.3 依赖（DependsOn）关系

一个系统的组件之间的物理依赖，将影响系统的开发、维护、测试和独立重用。类和自由（运算符）函数之间的逻辑关系，隐含了它们所驻留组件之间的物理依赖。如果编译和连接一个函数体时需要一个组件，那么通过说那个函数依赖于那个组件，我们可以宽松地为函数定义实现依赖。我们也可以用相似的方法为类定义实现依赖。更普遍地，我们可以精确地定义组件间的重要而纯粹的物理关系。

> **定 义**：**如果编译或连接组件 y 时需要组件 x，则组件 y DependsOn 组件 x。**

DependsOn 关系非常不同于我们已经介绍的关系。IsA 和 Uses 是逻辑关系，因为它们应用于逻辑实体，与逻辑实体驻留的物理组件无关。DependsOn 是一种物理关系，因为它应用于作为一个整体的组件，组件本身是物理实体。

用来表现一个物理单位依赖另一个物理单位的符号是一个（肥大的）箭头。例如：



意味着组件 plane 依赖组件 wing。也就是说，除非组件 wing 也是可用的，否则不能使用

组件 plane（即，它不能连接进一个程序或甚至不可能编译）。

逻辑实体用椭圆形表示，而物理实体用长方形表示，这已是我们的惯例了。注意，用来指出物理依赖的箭头画在组件之间而不是单个的类之间。永远不要将用来表示物理依赖的（肥大的）箭头符号和用来表示继承的箭头符号相混淆。继承箭头总是在两个类之间（类是逻辑实体）；DependsOn 箭头则连接物理实体（例如文件、组件和包）。

为了动态地描述 DependsOn 关系，考虑下面的一个串组件的框架头文件。顺便说一句，不要试图把组件命名为“string”；在标准 C 库头文件 string.h 存在的情况下它也许不能很好地工作。

```
// str.h
#ifndef INCLUDED_STR
#define INCLUDED_STR

#ifndef INCLUDED_CHARARRAY
#include "chararray.h"
#endif

class String {
    CharArray d_array;  // HasA
    // ...
  public:
    // ...
};

// ...

#endif
```

这里刚好有足够的可见信息使我们可以了解到，类 String 有一个 CharArray 类型的数据成员。我们从 C 可以知道，如果一个 struct 有一个用户自定义类型的实例作为数据成员，那么即使对 struct 的定义进行语法分析，也必须知道那个数据成员的大小和布局。

> **定 义**：如果编译 y.c 时需要 x.h，那么组件 y 展示了对组件 x 的**编译时**依赖。

更明确地说，如果不首先包含 chararray.h，就不可能编译任何需要 String 定义的文件。为此我们将#include "chararray.h"连同伴随物——冗余包含卫哨——一起嵌入组件 str 的头文件是正确的。

图 3-8 描述了组件 str 对组件 chararray 的物理依赖。一个组件的.c 文件在编译时必须总是依赖它的.h 文件。因为 str.c 没有 str.h 就不能编译，而 str.h 没有 chararray.h 不能编译，所以 str.c 有一个对 chararray.h 的间接的编译时依赖。请再次注意用于表示物理依赖的箭头是画在两个物理实体（在此例中是文件）之间的。一种组件级的更抽象的表示法如图 3-9 所示。

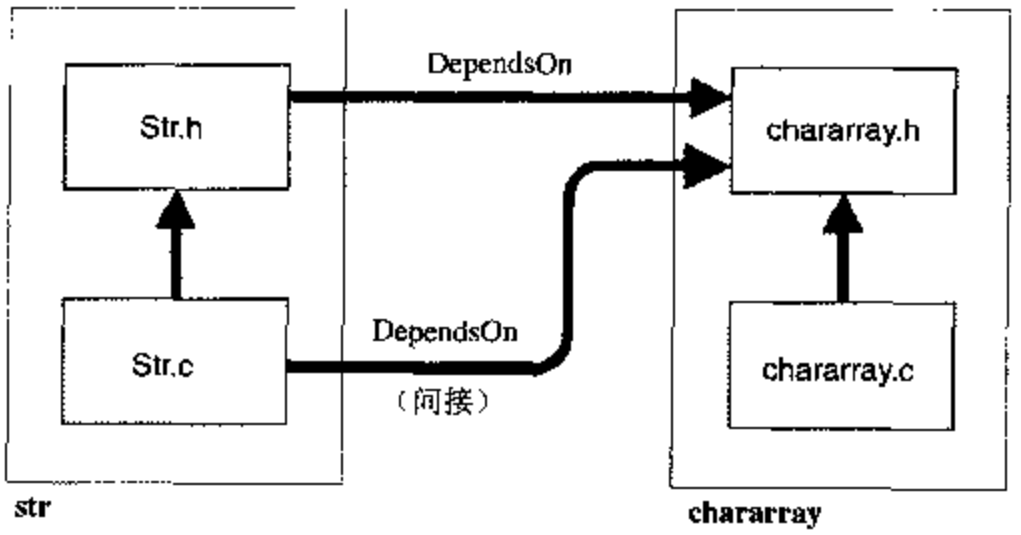


图 3-8 str.c 对 chararray.h 的间接的编译时依赖

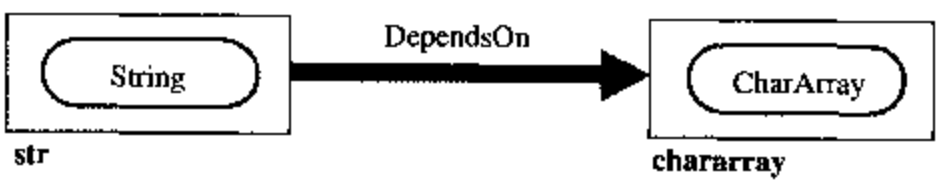


图 3-9 组件依赖的抽象表示法

一个组件可能在编译时不必依赖另一个组件，而在连接时要依赖它。考虑下面的组件 word 的实现和组件 str 的候补实现：

```
// word.h
#ifndef INCLUDED_WORD
#define INCLUDED_WORD

#ifndef INCLUDED_STR
#include "str.h"
#endif

class Word {
    String d_string;  // HasA
    // ...
  public:
    Word();
    // ...
};

#endif
```

```
// str.h
#ifndef INCLUDED_STR
#define INCLUDED_STR


class CharArray;


class String {
    CharArray *d_array_p;  // HoldsA
    // ...
  public:
    String();
    // ...
};

#endif
```

```
// word.c
#include "word.h"


// ...
```

```
// str.c
#include "str.h"
#include "chararray.h"

// ...
```

编译 chararray.c 当然需要 chararray.h。编译 str.c 时需要 str.h 和 chararray.h。最后，编译 word.c 时需要 word.h 和 str.h。注意编译 word.c 时不需要 chararray.h。组件 word 对组件 chararray 没有编译时依赖。但是，word 仍然展示了对 chararray 的物理依赖，一旦我们将一个测试驱动程序连接到 word，这种依赖就会变得明显。

> **定义**：如果对象文件 y.o（通过编译 y.c 产生）包含未定义的符号，因此可能在连接时直接或间接地调用 x.o 来辅助解析这些符号，那么就说组件 y 展示了对组件 x 的一种**连接时依赖**。

回顾一下，除了内联函数，在 C++中所有的类成员函数和静态数据成员都有外部连接。对于所有的实际用途，我们都可以说如果一个组件为了编译需要包含另一个组件，那么它将在连接时依赖那个组件在对象代码级解析未定义的符号。

原　则

一个编译时依赖几乎总是隐含一个连接时依赖。

正如图 3-10 所示，word.o 依赖定义在 str.o 中的外部名称。即使 word.o 不直接使用定义在 chararray.o 中的名称，它却使用了定义在 str.o 中的名称。用在 str.o 中解析这些未定义符号的名称仍将可能引入新的未定义的名称，它们的定义必须由 chararray.o 提供。综上所述，我们得出了一个有趣而重要的结论。

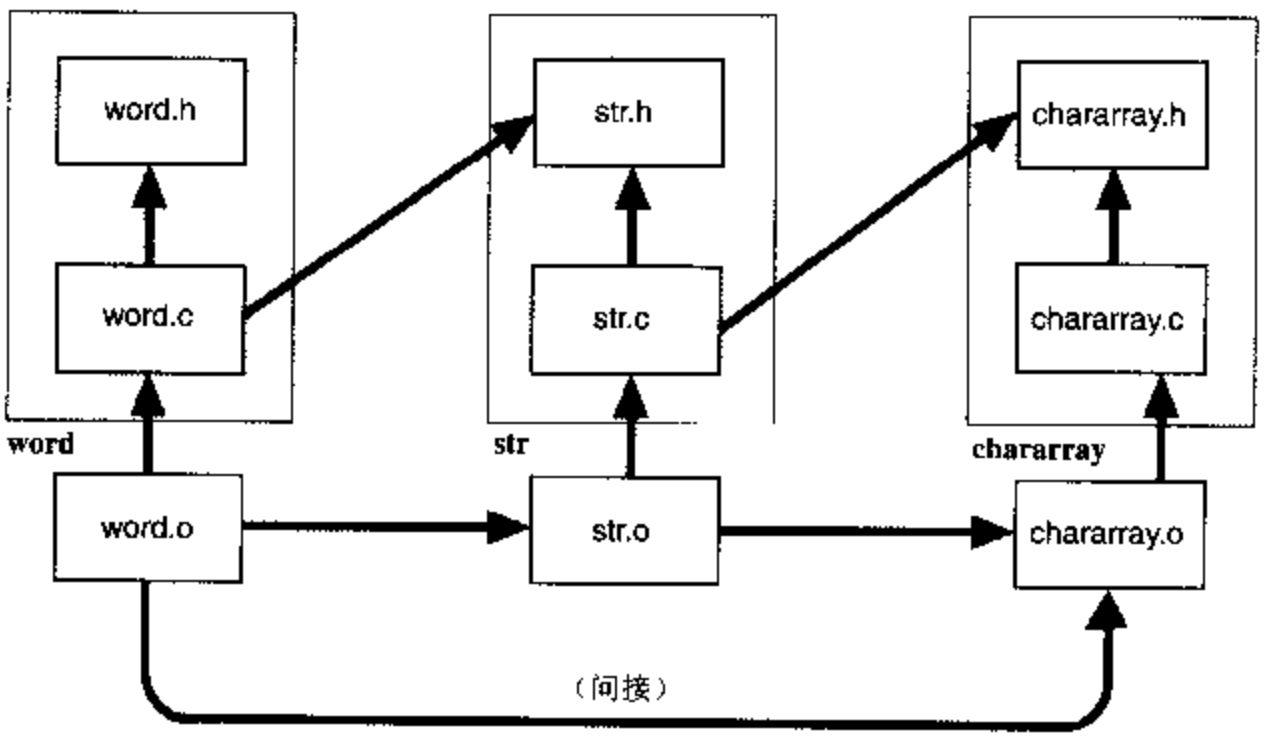


图 3-10　word 对 chararray 的连接时依赖

原　则

组件的 DependsOn 关系具有传递性。

例如，假设 x、y 和 z 是组件。如果 x 依赖 y，而 y 依赖 z，那么 x 依赖 z。组件之间的这种传递性质并没有涉及一个组件中的哪一个文件依赖另一个组件中的哪一个文件。任何这样的文件级依赖都足以使作为一个整体的组件产生实现依赖。

前面这个例子的抽象的、组件级依赖关系显示在图 3-11 中。word 对 str 的编译时依赖和 str 对 chararray 的编译时依赖已经产生了 word 对 chararray 的间接（连接时）依赖。

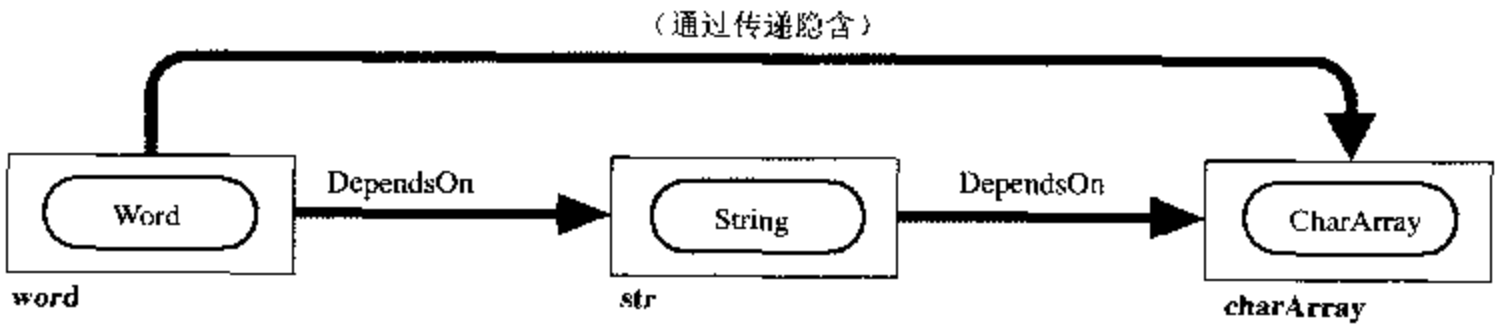


图 3-11 抽象的、组件级依赖关系

DependsOn 关系对物理设计来说是很重要的，因为它指明了在维护、测试和重用时由一个给定的组件所提供的功能所需要的所有组件。我们刚刚已经介绍了如何从源代码本身来推断物理依赖。正如我们将在 3.4 节介绍的，直接从像 IsA 和 Uses 这样的抽象逻辑关系来推断物理依赖是可能的。在设计阶段推断物理依赖有助于我们在开发过程的早期就获得一个可靠的物理体系结构。

3.4 隐含依赖

逻辑设计隐含一定的物理特性。在我们的逻辑设计实现**之前**，我们期望能够充分利用已知的逻辑关系来预测它的物理隐含。读者可能经常因为逻辑设计导致了不合需要的物理特性而被迫修改甚至完全重做逻辑设计。在这一节，我们从 Uses 关系开始集中讨论逻辑设计中有关物理依赖的隐含关系。

原 则

定义了某个函数的组件，通常会物理依赖于定义了某个类型（那个函数使用了该类型）的任何组件。

除非另外指定，否则我们将假设，如果一个函数使用了一个用户自定义类型，那么它会采用一种“实质的”方式来这样做。为解释“实质的”是什么含义，让我们暂时假设，若一个函数在它的接口中使用了一个类型，则定义了该函数的组件就必然要包含定义了该类型的组件的.h 文件。

图 3-12 描述了我们的假设：如果函数 Two::getInfo 在它的接口中使用了类 One，那么它可能会在它的实现中用 One 做某种事情，这就要求看到 One 的定义。在这个例子中，Two::getInfo

调用了类 Onc 的 const 成员函数 info，这就要求编译器在编译 two.c 时要找到 One 的定义。

```
// two.c
#include "two.h"
#include "one.h"
// ...
int Two::getInfo(const One& one)
{
    return one.info();
}
// ...
```

```
// one.h
#ifndef INCLUDED_ONE
#define INCLUDED_ONE
// ...
class One {
    // ...
    int info() const;
    // ...
};
// ...
#endif
```

图 3-12　Uses 关系常常隐含一个编译时依赖

Uses 关系隐含一个编译时依赖的假设太强了。但是，这个假设相当精确地预测了物理实现依赖。Uscs 关系没有必要为了减少一个间接的物理依赖而引起一个编译时依赖。为了了解一个间接的连接时依赖怎样才能出现，可以考虑在前一个例子中再加上另一个组件 three 和另外两个文件 two.h 和 three.c。

如图 3-13 所示，three.c 定义了一个成员函数 x2info，x2info 在它的接口中使用了 One。但是，x2info 的参数是通过引用传递的，x2info 在把它的参数传递给 Two 的静态成员函数 getInfo 之前，并没有实质地使用 One 的定义。getInfo 也通过引用接受了一个 One 对象。组件 three 中的 x2info 函数把类 One 看成是不透明的，除了知道它是一个类、结构或联合之外，不知道有关 One 的任何事情。

```
// three.c
#include "three.h"
#include "two.h"
// ...
int Three::x2info(const One& one)
{
    return 2 * Two::getInfo(one);
}
// ...
```

```
// two.h
#ifndef INCLUDED_TWO
#define INCLUDED_TWO
class One;
// ...
class Two {
    // ...
  public:
    static int getInfo(const One& one);
    // ...
};
// ...
#endif
```

图 3-13　Uses 关系几乎总是隐含一个连接时依赖

假设类 Three 中没有其他的函数（实质地）使用了 One，而且组件 three 对组件 one 没有其他的编译时依赖。也就是说，编译 three.c 只要包含 three.h 和 two.h 就足够了，即使 three

的接口中使用了 one。但是函数 x2info 是间接依赖 One 的。如果我们要测试 three，我们必须先编写和编译 two.c 才能够连接。要那样做，two.c 就必须包含 one.h。

如图 3-14 所示，由于使用一个对象而引起的实现依赖具有传递性。就是说，如果 Three 用了 Two 而 Two 用了 One，那么定义了 Three 的组件（几乎总是）DependsOn 定义了 One 的组件。在这个案例中，组件 three 中的函数 x2info 使用了 One（一般地），但也使用了定义在组件 two 中的函数 getInfo。函数 getInfo 又在它的接口中使用了 One，但这次 getInfo 在它的接口中实质性地使用了 One。

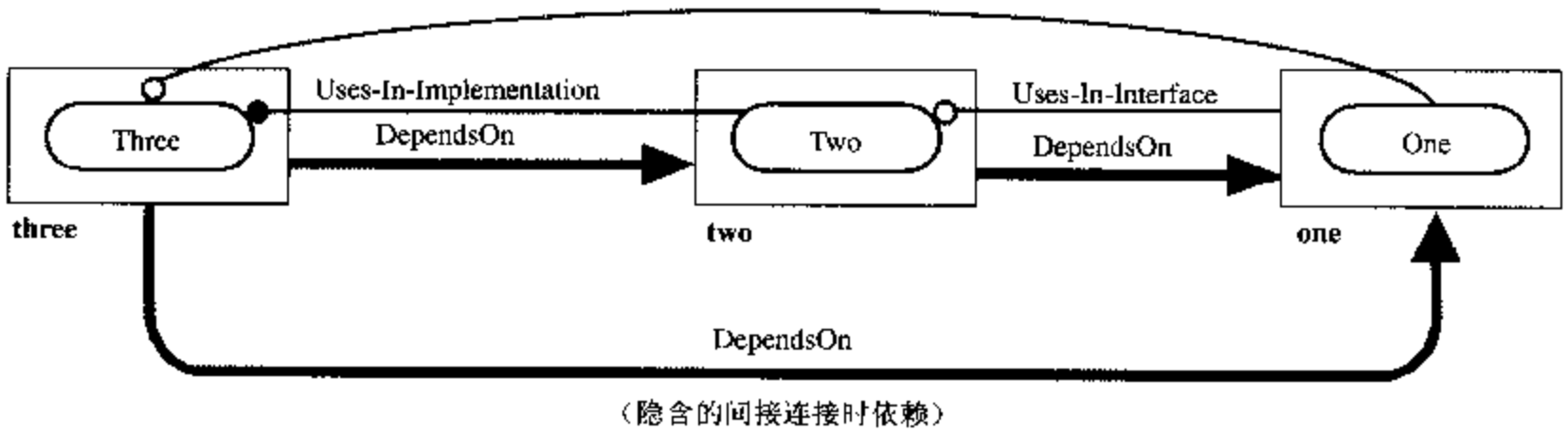


图 3-14 逻辑使用（Logical Uses）关系隐含组件依赖

有这种可能：使用了一个对象而没有引起对该对象的编译时依赖或连接时依赖。在实践中，这种受限的使用有时会在设计中出现，但几乎永远不会碰巧出现。我们将在 5.4 节中探讨这种设计技术。

原 则

如果一个组件定义了某个类，且该类 IsA 或 HasA 用户自定义类型，那么那个定义了类的组件总是在编译时依赖那个定义了类型的组件。

某些逻辑关系有很强的物理隐含。例如，从一个类型派生（IsA）或嵌入一个类型的实例（HasA）总是隐含一个类将在编译时依赖那个类型。事实上，这些逻辑关系隐含的编译时依赖不仅针对类本身，也针对这个类的任何客户程序。

图 3-15 描述了图 3-11 例子中的 IsA 和 HasA 的物理隐含。这一次 Word 被重新实现为一种 String，而 String 有一个 CharArray 数据成员[①]。为了编译 word.c，String 和 CharArray 的定义都必须是可访问的。此外，每一个 Word 的客户程序要想通过编译也都需要 String 和 CharArray 的定义。实质使用了一个类型的私有派生和内联函数拥有这些同样强烈的物理隐

① 从逻辑设计角度来看，这种结构的继承方式是潜在不可靠的。假设 Word 的语义要求它保存由 String 基类支持的任意数据的一个适当子集（例如非空格、标点或控制符），如果使用了公共继承，就没有办法阻止基类功能被客户使用，从而违反了这个要求（见 **meyers**，Item37，130~132 页）。

含。在所有这些案例中，我们都应该把需要的#include 指令嵌入组件的.h 文件中。

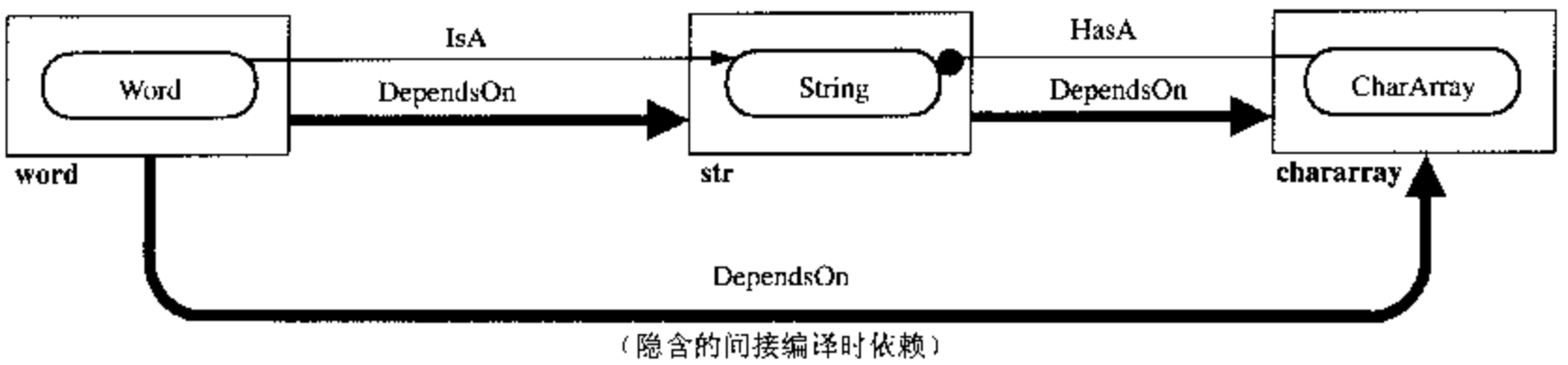


图 3-15　逻辑的 IsA 和 HasA 关系隐含组件依赖

如果一个类 HoldsA 一个类型（就是说如果它指向或引用那个类型作为一个数据成员），或者这个类型被实质地用在一个非内联函数体中，那么就不一定隐含这种强烈的物理耦合。这种用法是不正当的，因为它迫使组件的用户程序在编译时依赖它的实现类型，正如把#include 指令嵌入组件的文件头所导致的结果那样。在第 6 章中，这些微妙而重要的区别将被用来减少编译时耦合。

迄今为止我们一次只涉及了两个或三个类。现在我们要从一个给定的抽象逻辑表示，推断出一个稍微大些的组件集合之间的物理依赖。图 3-16 中的图描述了一个用于支持在线词汇表的小型子系统。

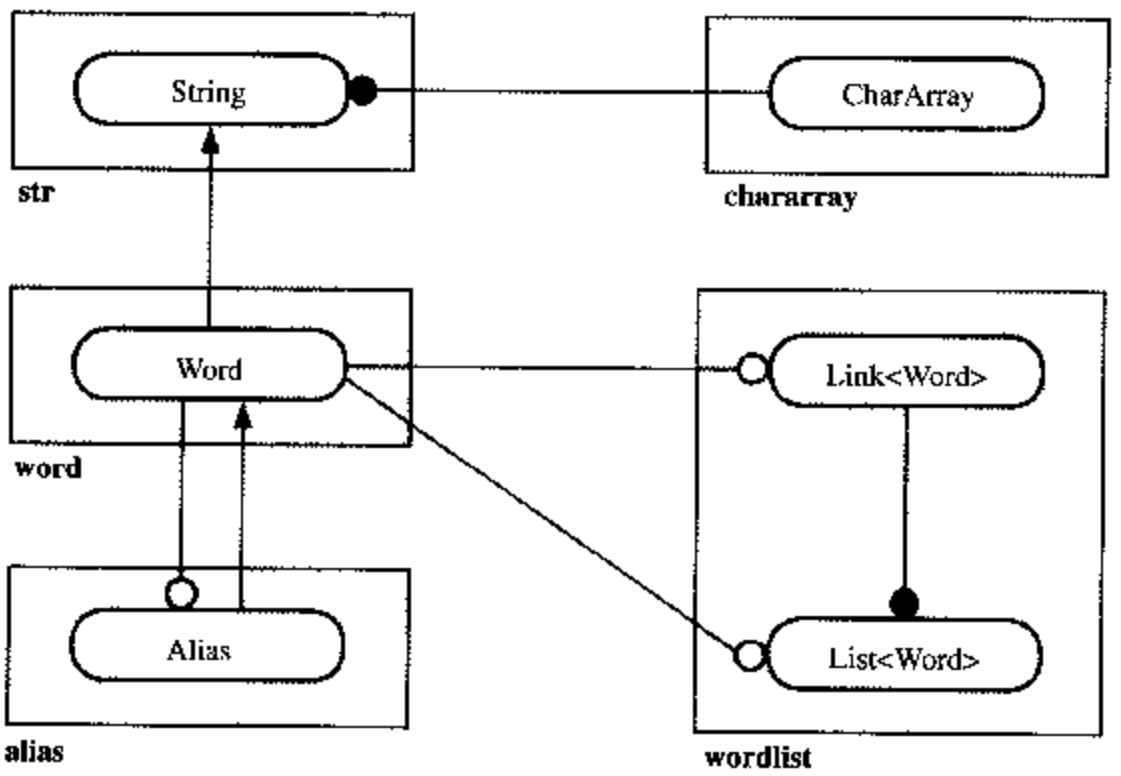
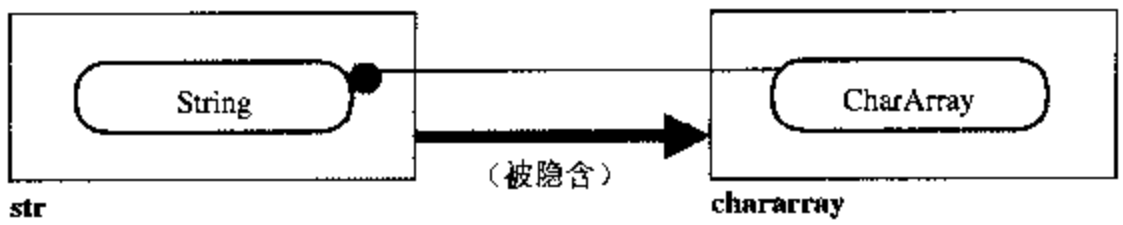
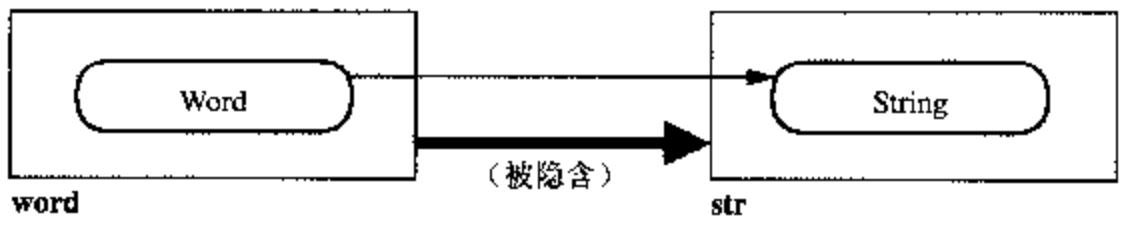


图 3-16　组件之间的逻辑关系

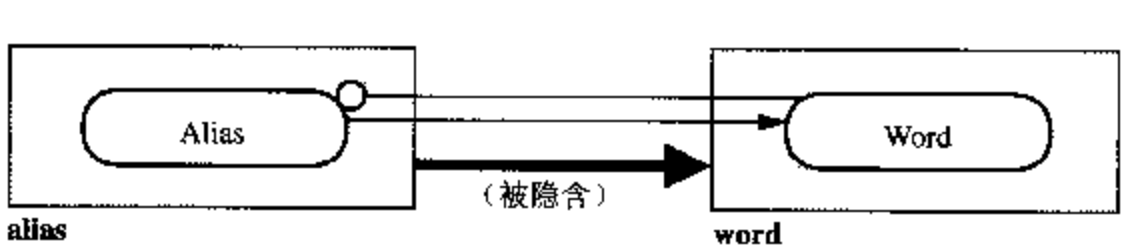
在图 3-16 的右上角，可以看到类 CharArray 在它自己的单独的组件中。类 String（在它的左边）在其实现中使用了类 CharArray，这样我们推断出组件 str 对组件 chararray 的一个可能的物理依赖：



在这个设计中，一个 Word 实现为一种 String（也许并不适当）。一个从类 Word 到类 String 的箭头被用来表示这种关系。我们也知道 IsA 关系总是隐含着定义组件之间的一个编译时物理依赖（和继承箭头相同的方向）。因此 word 明确地依赖 str。



正如我们可以从图 3-16 中看到的，Alias 不仅是一个（IsA）Word 而且在其接口中使用了（Uses）Word。但是要注意，Uses 关系的隐含依赖和表示 IsA 关系的箭头指向同一方向（从 Alias 到 Word）。因此，在 Word 和 Alias 之间没有隐含的循环依赖。所以有可能在一个没有包含或连接到 alias 的程序中使用 word。



现在考虑图 3-16 中的 wordlist 组件，它定义了两个假定的模板类 Link<Word>和 List<Word>。在组件 wordlist 内，我们可以看到 Link<Word>和 List<Word>之间有一个逻辑的“Uses-In-The-Implementation”关系。因为这些类已经定义在同一个组件内，它们之间的逻辑关系不会影响物理依赖。

Link<Word>和 List<Word>都在它们各自的接口中使用了 Word。这些逻辑关系的任何一个都足以使我们推断出一个可能的整个 wordlist 组件对 word 的物理依赖。请再次注意组件 word 可以存在于一个不包含或连接到 wordlist 的程序中，但反之则不然。

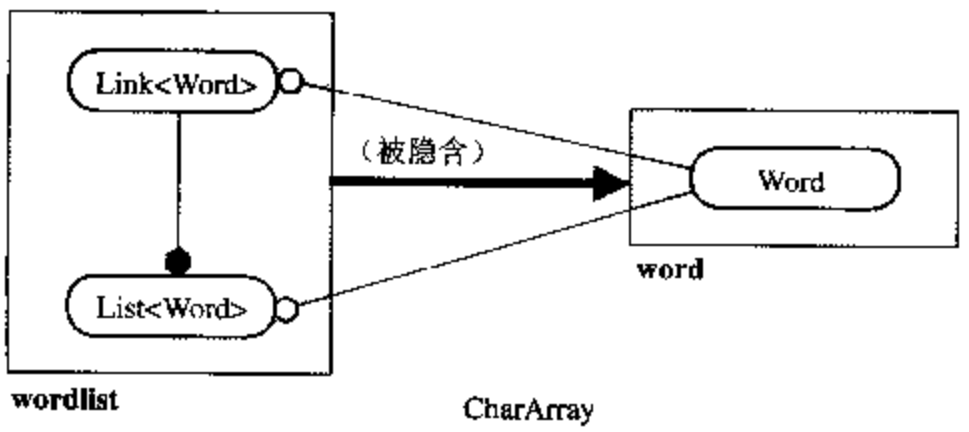


图 3-17 显示了从图 3-16 中的抽象逻辑关系推断物理依赖而产生的一个组件依赖图。实际上，这个图没有显示出完整的组件依赖关系。例如，它没有明确显示 word 对 chararray 或 wordlist 对 str 的间接依赖。如果我们把每个间接依赖都视同直接依赖那样画出来，就可以得到完整的依赖图。这样的图称为一个**传递闭包（transitive closure）**。

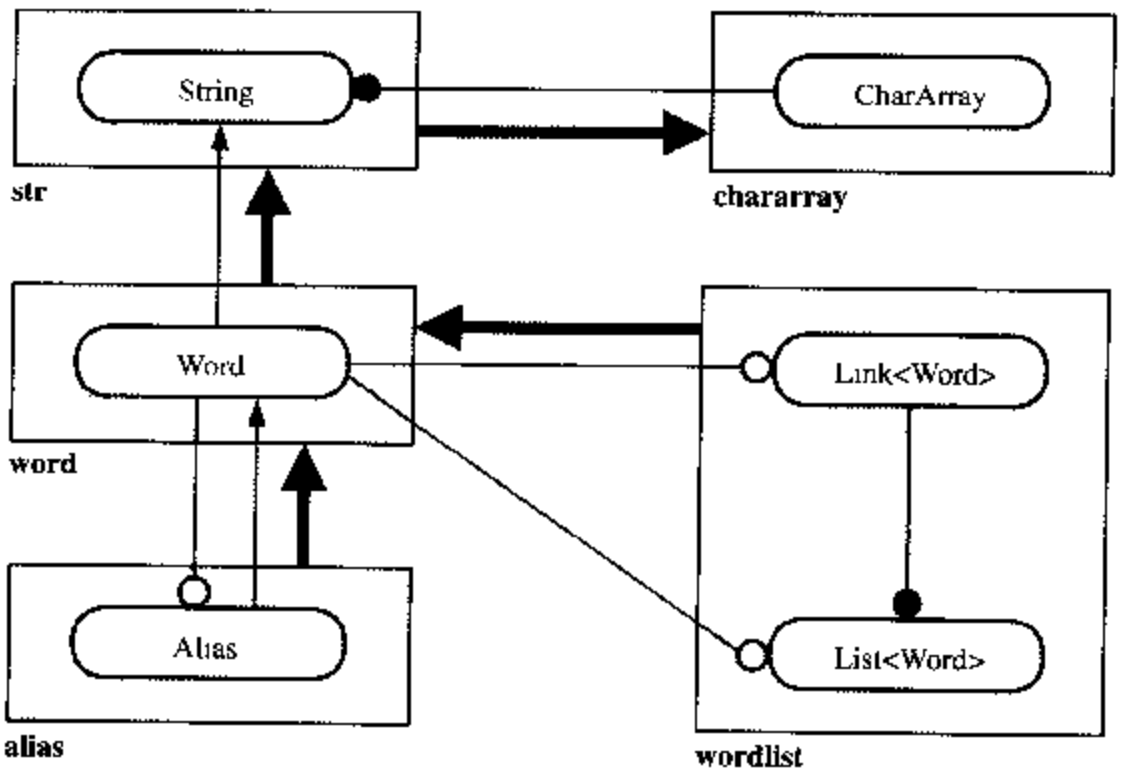
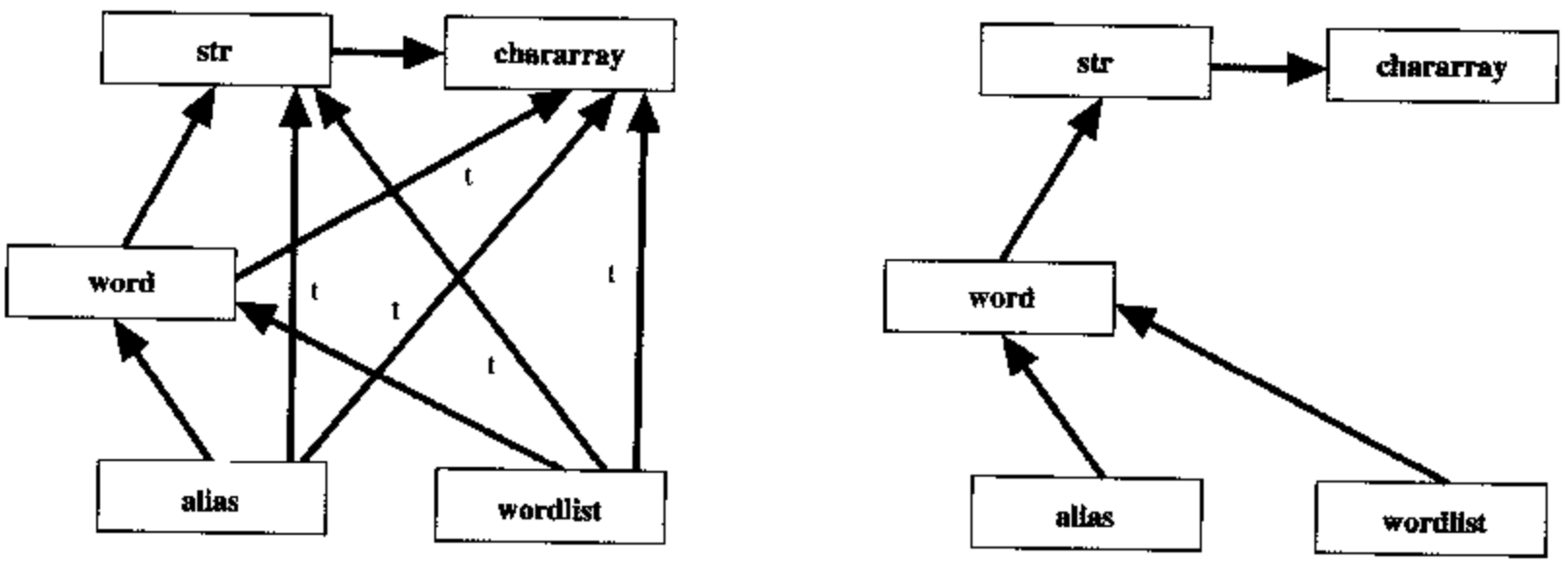


图 3-17 只显示直接依赖的组件图

图 3-17 的依赖图的传递闭包显示在图 3-18（a）中。在这张图中所有标识为 t 的边被称为**传递边（transitive edges）**，因为其他代表“直接”依赖的边隐含了它们的存在。删除这些冗余的传递边不会损失基本的信息，但是它们的存在可以减少混乱，使图更容易理解。



（a）完整的图　　（b）删除了冗余边的图

图 3-18 直接依赖图的传递闭包

从图 3-18（b）中很容易知道 word 间接依赖 chararray，而 wordlist 间接依赖 word。一般来说，当且仅当依赖图中有一条从 x 到 y 的路径时，组件 x 才依赖（DependsOn）组件 y。

扼要重述：逻辑关系隐含物理依赖。像 IsA 和 HasA 这样的逻辑实体之间的关系，在跨越组件边界实现时将总是隐含编译时依赖。像 HoldsA 和 Uses 这样的关系可能隐含跨越组件的连接时依赖。通过在设计阶段考虑隐含的依赖，我们可以远在编写任何代码之前就评价出我们的体系结构的物理质量。在第 4 章我们将讨论既可以提高易测性又可以促进重用的组件依赖的特性。在下一节中，我们将讨论如何更有效地从源代码中提取实际的物理依赖。

3.5 提取实际的依赖

现在假设我们根据隐含依赖设计一个大型的项目。在大部分设计工作完成之后，开发正在进行，我们很想有一种能够提取我们的组件之间的实际物理依赖的工具，以便我们可能跟踪实际的组件依赖，并且将它们和我们最初的设计期望进行比较。

虽然通过语法分析整个 C++程序的源代码来确定准确的组件依赖图是可能的，但这样做既困难又相对较慢。假如我们已经遵守了在 3.2 节介绍的设计规则，就可能通过只分析 C++预处理器#include 指令直接从组件的源文件中提取组件依赖图。这个过程相对较快，因为有许多标准的、公共领域的依赖分析工具（如 gmake、mkmf 和 cdep）可以做这个工作。

原 则

假如系统编译成功的话，仅凭由 C++预处理器#include 指令产生的包含图，就足以推断出系统内部的所有物理依赖。

想要了解为什么这个论断是正确的，考虑下面的推理。如果组件 x 直接实质地使用了组件 y，那么为了编译 x，编译器就必须查看 y.h 提供的定义。惟一能做到这一点的方法就是组件 x 直接或间接地包含 y.h。作为 3.2 节的设计规则的一个结果，任何这样的直接实质使用都等同于一个编译时依赖。

假如 x 通过了编译，则相反的说法（即如果 x 不包含 y.h，那么 x 就没有对 y 的编译时依赖）无疑也是正确的。

从另一个角度来说，组件 x 会合理地包含组件 y 的头文件的惟一的理由是：组件 x 事实上确实直接实质地使用了组件 y。否则包含物本身将是多余的，并且会引进不必要的编译时

耦合。

相反的说法（即如果x没有一个对y的本质的编译时依赖，那么x不包含y.h）应该总是正确的——但是有时候，由于人们的疏忽，也可能不正确。

指导方针

只有当组件x直接实质地使用了定义在y中的一个类或自由运算符函数时，x才应该包含y.h。

无论先前是否已经有一个编译时依赖存在，一个组件包含了另一个组件的文件头这个事实都会再强加一个编译时依赖。如果我们假设一个组件中的所有的#include指令都是必要的，那么将有一个连接时依赖伴随着编译时依赖（我们已经知道它具有传递性）。换句话说，“实质的使用”应该等同于“头文件包含”，实质的使用几乎总是隐含一种传递的物理依赖。

一个组件集合的#include图恰好相当精确地反映了组件间的依赖关系。如果我们把“x包含y.h（直接或间接）”解释为“x直接DependsOn y”，那么#include图展示的关系精确地反映了编译时物理组件依赖。

设计规则规定，所有的对一个组件的实质的使用都必须通过包含它的头文件（而不是通过局部的外部声明）来标志，以保证Include关系的传递闭包能指明组件之间所有实际的物理依赖。

这些提取的依赖有时候会因为太机械而犯错。以这种方式提取的依赖图可能会包括由不必要的#include指令（应该被删除）引起的额外的、假的依赖。但是如果所有的主要设计规则都遵守了，依赖图就决不会遗漏一个实际的组件依赖。

从一个潜在的大型组件集合快速而精确地提取实际物理依赖的能力，使我们在整个开发过程中都能够检验这些依赖是否与我们整体的体系结构计划相一致。附录C中描述了一种物理依赖提取/分析工具。

3.6 友元关系

我们现在偏离本章主题来讨论有关友元关系以及授权的友元关系如何影响一个类和一个组件的逻辑接口。友元关系和物理设计之间的交互程度强烈得令人惊讶。虽然表面上是一个逻辑设计问题，友元关系却会影响我们把逻辑结构收集进组件的方式。避免友元关系穿越组

件边界的愿望，甚至可能导致我们要重建我们的逻辑设计。我们将在整本书中经常提到本节中介绍的内容。

指导方针

避免把（远距离的）友元关系授权给定义在另一个组件中的一个逻辑实体。

根据《Annotated C++ Reference Manual》（注释C++参考手册），“友元是一个类的接口中的和一个成员差不多的部分[①]”。在作这个论断时，有一个隐含的假设，即友元不可分离地与一个授予它友元关系的对象结合在一起。

从纯粹的逻辑角度来看，如果一个类作了一个友元关系的声明，那么，按照封装的定义（2.2节），该声明就不是这个类的一个封装细节。任何人只要定义了一个函数并且其声明严格与一个类中的一个友元声明相匹配，那么假如没有与定义在同一程序内的友元声明相匹配的其他函数，他就可以获得编程访问那个类的私有成员的权利。在那种非常严格的意义上，友元声明本身是类接口的一部分——实际的函数定义却不是。

原 则

一个组件内部的友元关系是该组件的一个实现细节。

通过把组件（而不是类）视为设计的基本单位，我们得出了一种完全不同的观点。只要友元关系被局部授权（即只要它只授权给定义在相同组件内的逻辑实体），那么，这些友元事实上就与授权友元的对象不可分离地结合在一起了。

定 义：若通过一个组件的逻辑接口不能通过程序访问或探测到该组件包含的一个实现细节（类型、数据或函数），则称这些实现细节被该组件**封装**了。

根据该定义，一定可以通过程序判定什么在逻辑（公共）接口中。请思考定义在组件stack中的等式运算符：

```
int operator==(const Stack&, const Stack&);
```

如果这个运算符突然被声明为类Stack的一个friend（友元），那么按照上面的说法，把运算符本身放在Stack的（公共）接口中，应该能通过程序发现这个变化——对吧？但是，假如运算符是定义在该组件内的，则授予运算符友元状态对该组件的逻辑接口绝对没有影响。事实上，从任何客户的角度来看，operator==是不是类Stack的友元是被这个组件所封装的一个实现细节！

为了进一步说明这一点，请简单地思考一个String类，它（在其他东西中间）定义了成

① **ellis**，11.4节，248页。

员 operator+=来实现了自加功能。

```
String {
    //...
  public:
    //...
    String(const String& string);                    // copy constructor
    //...
    String& operator+=(const String& rhs);           // concatenate to me
};
```

现在我们可以选择在同一组件内实现非破坏性的连接（+），而不必使运算符成为友元：

```
String operator+(const String& lhs, const String& rhs)
{
    return String(lhs) += rhs;
}
```

如果分析之后我们发现有必要改善operator+的性能，我们可以通过声明operator+为String的一个友元来扩展类String的封装：

```
String {
    //...
    friend String operator+(const String&, const String&);
  public:
    //...
    String(const String& string);                    // copy constructor
    //...
    String& operator+=(const String& rhs);           // concatenate to me
};
```

声明operator+为String的友元，可以得到一个更有效的实现，却会潜在地增加维护开销，但不会影响组件的逻辑接口：

```
String operator+(const String& lhs, const String& rhs)
{
    // clever, more efficient implementation using private members
}
```

没有简单的编程方法可以让组件的客户知道，一个给定的在组件内定义的逻辑实体是否被同一组件内定义的一个类声明为友元[①]。

原　则

为定义在同一个组件内的类授予（局部的）友元关系不会破坏封装。

① C++语言不区分类中的友元声明所处的位置。但是，将声明放在类的一个私有区域，反映了该组件的有关局部友元关系的语义。

授予局部的友元关系不会威胁到暴露一个对象的私有细节给未经授权的用户。因为被声明为友元的类被定义（局部地）在同一组件的头文件内，任何试图使用该对象的授权友元关系的人都将拥有信任他们的所有友元类的有效的定义。重定义这些友元类的任何企图都将被编译器制止，它会迅速发出错误警告：

```
MULTIPALLY DEFINED CLASS.
```

原　则

在组件内定义一个容器类的同时定义一个迭代器类，可以在保持封装的同时，使组件具有用户可扩展性、改进可维护性和加强重用性。

在单个组件内，应该在组件（作为一个整体）必需获得适当封装的地方局部地授予友元关系。当使用容器/迭代器模式时，我们总是希望让逻辑实体一直能够访问我们的实现，该实现在物理上紧密反映高度的逻辑内聚性。在一个组件之外对逻辑实体授予友元关系，在本书中称为远距离友元关系（long-distance friendship），则是完全不同的另一回事。

原　则

对一个定义在系统的单独物理部分的逻辑实体，授权（远距离）友元关系，会破坏授予该友元关系的那个类的封装。

对系统的另一个物理片段授予私有访问权，会导致封装存在漏洞，很可能会被通过插入一个假冒的组件来获得访问而滥用。例如，假设 3.1 节的 StackIter 类定义在组件 stackiter 中，与类 Stack 分离。于是没有什么可以阻止一个 stack 组件的用户定义一个定制的 StackIter 来替代他或她自己的组件，从而获得对 Stack 类的私有访问权。一旦这种事情发生，授权远距离友元关系的类就不能保护它的私有成员不受访问——它的封装已经遭到了破坏。

除了是封装的破坏者，远距离友元关系也是糟糕结构设计的一个症状。让物理上独立的逻辑实体相互紧密依赖，极易降低可维护性。特别是远距离友元关系允许对私有实现细节进行局部修改，影响系统中物理上较远的部分，因而会降低模块化。

甚至局部友元关系的过度使用也会影响可维护性。授权友元关系会扩大一个类本身的“接口”。有权访问对象实现的封装细节的函数越多，当修改实现时需要回访（可能需要重做）的代码也就越多。直接访问私有信息的代码行越少，就越容易试验可选择的实现。

3.6.1 远距离友元关系与隐含依赖

尽管我们不鼓励使用远距离友元关系，但是在一个组件外授予友元关系的可能性却导致我们必须明确，在决定涉及某个类的 Uses 关系时，是否应该考虑与该类的一个友元声明相匹配的函数。对这个问题的回答是重要的，但仅限于在友元关系超出了单个组件并且要基于 Uses

关系来推断出物理依赖的情况下。

原　则

友元关系影响访问特权但不隐含依赖。

一个类是一个不可分割的逻辑单位。一个自由函数是一个不同的逻辑单位。自由函数是否是一个类的友元决不会影响系统中的任何隐含物理依赖。

考虑定义在其自己的组件 barop 中的自由运算符函数

```
// barop.h
class Bar;
int operator==(const Bar&, const Bar&);
```

图 3-19 显示了这个运算符，以及类 Stack 和类 StackIter（现在也显示在单独的组件中）。这个自由运算符既不是 Stack 的成员也不是它的友元，因此它显然不会扩大类 Stack 的接口。但是当我们把这个运算符声明为 Stack 的友元时会确切地改变什么呢？

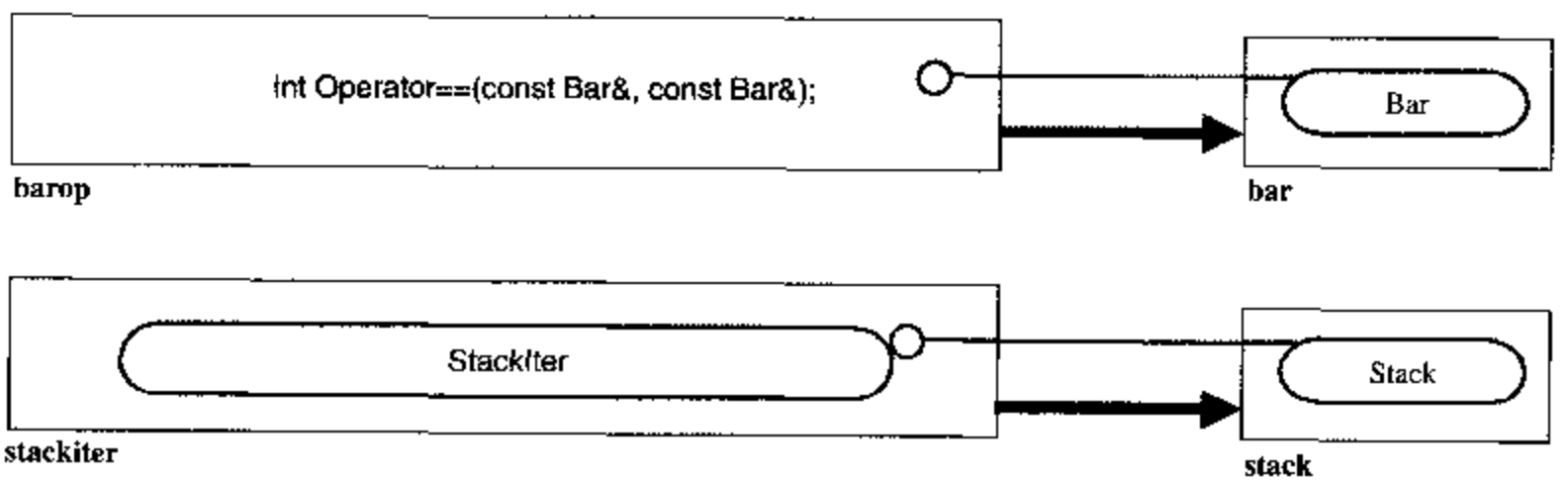


图 3-19　在不同的组件中的相关的逻辑实体

现在 operator==(const Bar&, const Bar&)会被认为是类 Stack 的接口的一部分吗？如果会，那么 Stack 在它的接口中使用了 Bar，并且有一个错误的、组件 stack 对组件 bar 的隐含依赖，如图 3-20 所示。

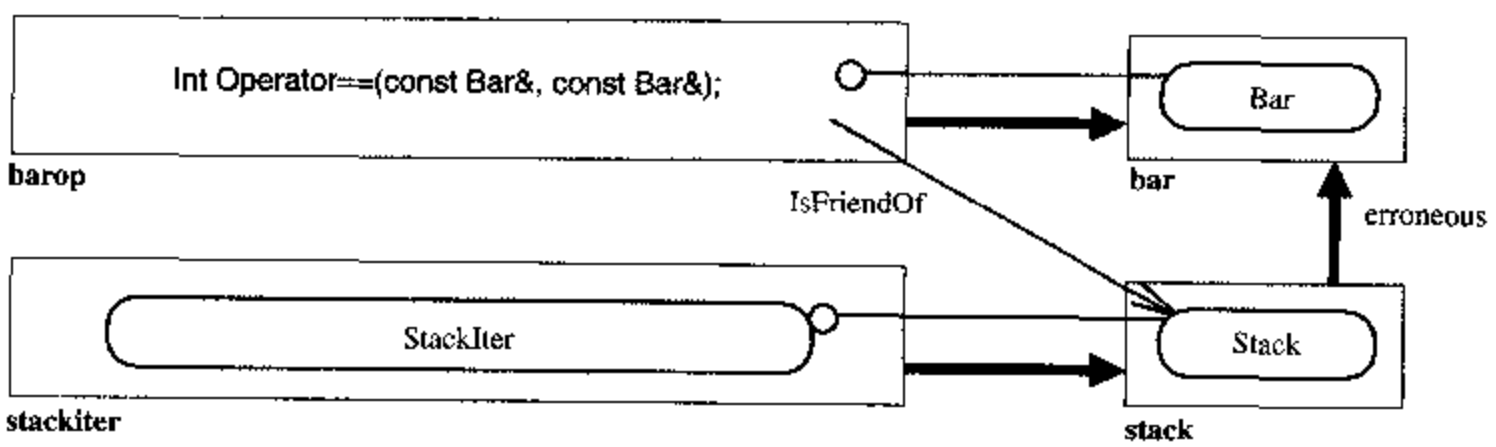


图 3-20　错误推断的 stack 对 bar 的依赖

只授权运算符友元关系不会使得 stack 突然物理依赖于 barop。使用一个类型会隐含一个对它的**所有**成员的依赖，但**不**一定会隐含对它的**任何**友元的依赖。特别地，operator==(const Bar&, const Bar&)是 Stack 的一个友元，StackIter 使用了 Stack，但这决不意味着 StackIter 直接或间接地使用了 operator==(const Bar&, const Bar&)。

注意图 3-20 中用于指示 IsFriendOf 关系的箭头的方向。这个箭头表示 operator==(const Bar&, const Bar&)现在被允许以一种比以前更紧密的方式依赖 Stack，但它并不保证任何实际的依赖。无论什么在箭头的相反方向都不会有物理依赖——把 operator==(const Bar&, const Bar&)看成是 Stack 的逻辑接口的一部分时，就会隐含上述观点。总而言之，授权友元关系改变的只是访问特权而不是物理依赖。

下面，通过一对用于比较 Stack 类型和 Foo 类型（对称地）的对象的自由运算符来举例说明这条原理的重要性：

```
int operator==(const Stack&, const Foo&)
int operator==(const Foo&, const Stack&)
```

我们不必浏览 Stack 或 Foo 的头文件就可以知道，这些运算符不是成员，因而不是这两个类的逻辑接口的一部分。因为这些运算符不是类的一部分，我们可以把它们定义在一个完全独立的组件中，从而使其可以被客户程序只在需要的时候包含。若不考虑访问特权，Uses-In-The Interface 关系指向一个方向，即从运算符到类，如图 3-21 所示。

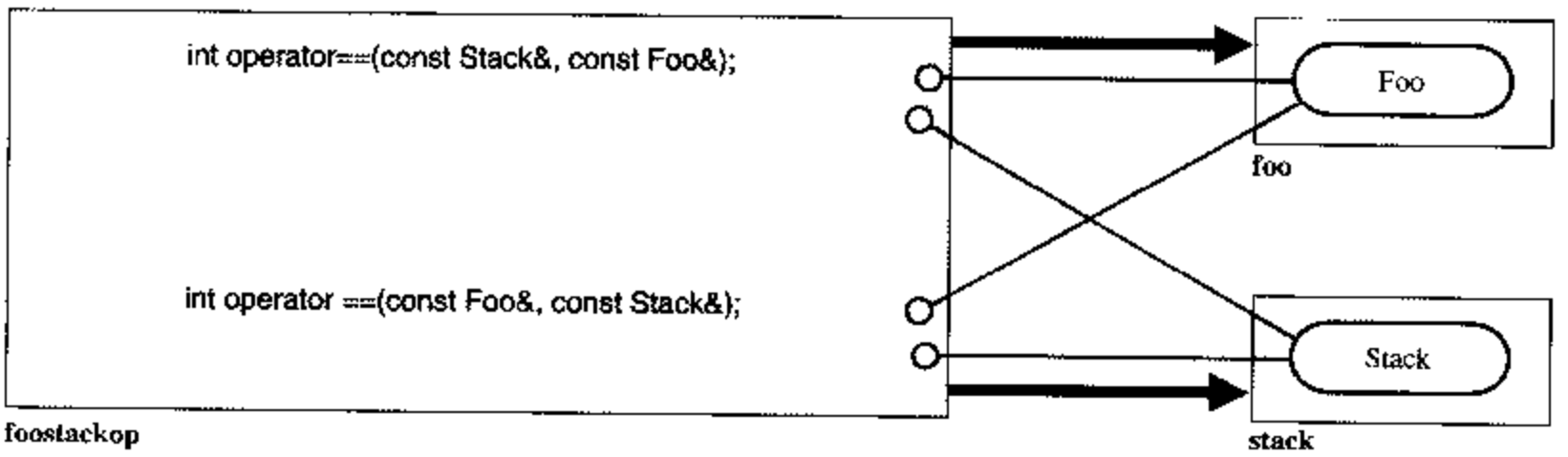


图 3-21 自由运算符对类的非循环依赖

现在考虑这个非常可疑的决定：改为加上以下两个 operator==**成员**函数：

```
int Stack::operator==(const Foo&   rhs) const;
int   Foo::operator==(const Stack& rhs) const;
```

作为成员，这些运算符函数显然是它们各自类的接口的一部分。每一个成员运算符都在它的接口中使用了另一个类。这些运算符的存在，在 Foo 和 Stack 之间引入了一个不合乎需要的、循环的 Uses-In-The-Interface 关系，如图 3-22 所示。当这些运算符是自由的并定义在一个独立的组件中时，不会导致这样的循环依赖。加入自由（运算符）函数绝对不会影响任何

类的逻辑接口（不考虑访问权），因为自由运算符（不像成员）不是任何类的固有的部分。（注意：使 operator==成为一个成员，按照纯粹的逻辑设计考虑，是一个糟糕的决定，正如将在 9.1.2 节中所讨论的那样）。

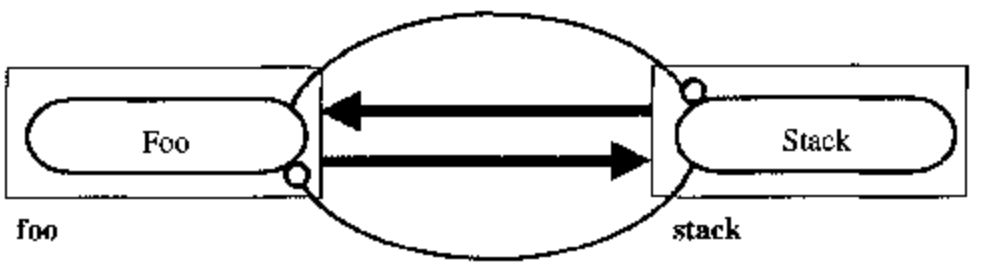


图 3-22　成员运算符对类的循环依赖

虽然授权友元关系在本质上绝不会直接影响隐含依赖，但是友元关系可能间接影响物理耦合。在试图避免这些与远距离友元关系有关的问题时，我们可能发现自己正在把若干紧密依赖的逻辑实体聚合进一个单个的组件内，从而在物理上使它们建立了耦合关系（见 5.8 节）。

3.6.2　友元关系与“骗局”

对于大型系统（可能跨越若干管理层次以及若干地理位置）来说，保护某一实现不被未经授权地使用是很重要的。在这种情形下，只是说“我留了一个漏洞，但请不要进来！”是没有用的。已经在源代码级访问了你的代码的人（尤其是顾客）将做一些他们需要做的事情来使他们的程序运转。如果使用你的一个私有数据成员能够解决他们的问题，即使只有一半的机会他们也可能使用它。如果用户能够直接访问你的实现，那么在你将来试图改进实现的时候，你可能会遇到不必要的阻力。

一个不谨慎的开发者可以简单地通过局部地（在文件作用域）定义友元类来获得访问私有细节的权利。于是这个开发者可以经由内联函数来使用这些细节，内联函数没有外部连接，因而不会与合法的函数定义发生冲突（即使它们被连接进了程序）。同样的原因，把一个单独的非内联自由（运算符）函数声明为一个友元——即使是局部地——也不能免于经由内联置换的“骗局”。人们实际上在产品代码中已经这样做了。你已经被警告了！

图 3-23 描述了一个非常值得怀疑的实践：通过使用远距离友元关系来故意利用封装上遗留的漏洞。类 Jail 定义了一个私有成员 release()，并且视一个定义在 jail 组件之外的名为 JailKey 的类为友元。经授权的 JailKey 被定义在组件 jailkey 内，被连接进程序中。一个恶意的 visitor 组件声明了类 JailKey 的一个局部版本，整个隐藏在 visitor.c 文件内。因为这个 JailKey 的违法版本没有带外部连接的成员，它可以静静地共存在一个文件中，依然利用 Jail 提供的友元关系。定义在 main()中的名为“bugsy”的 Visitor 对象的构造函数，创建了一个它自己的 JailKey 的一个实例，它在构造时调用 Jail 的私有的 release()方法。泄漏是不可避免的了。

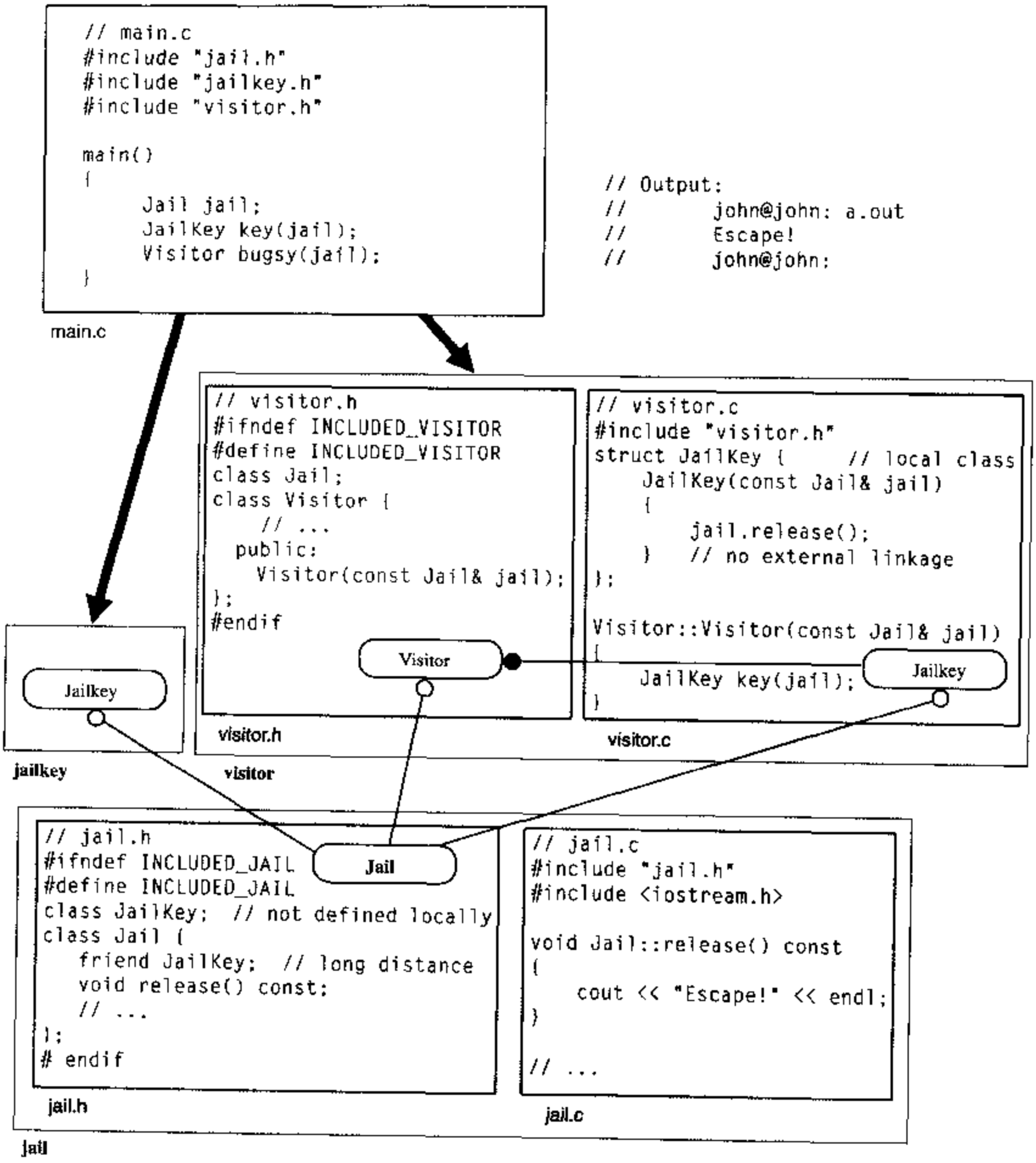


图 3-23 滥用友元关系的例子

令人悲哀的是，甚至还有更容易和更可恨的方法可以破坏封装：

```
// felon.c
#define private public        // capital offense
```

```
#include "jail.h"

void Felon::breakOut(Jail *jail)
{
    jail->release();
}

// ...
```

然而，像这样写头文件

```
// jail.h
#if !defined(INCLUDED_JAIL) && !defined(protected) && !defined(private)
#define INCLUDED_JAIL

class Jail {                    // maximum security
// ...

#endif
```

就可能走得太远了。

3.7 小结

开发可维护、易测试和可重用的软件需要全面的物理设计和逻辑设计的知识。物理设计研究组织的问题，超出了逻辑领域的范畴，物理设计很容易影响可测量的特性，例如运行时间、编译时间、连接时间以及可执行文件大小。

一个组件是由一个.c文件和一个.h文件组成的物理实体，它具体表达了一个逻辑抽象的具体实现。一个组件一般包含一个、两个甚至多个类，以及需要用来支持全部抽象的适当的自由运算符。一个组件（而不是一个类）是逻辑设计和物理设计的适当单位，因为它能够

（1）让若干逻辑实体把一个单一的抽象表现为一个内聚单位；

（2）考虑到物理问题和组织问题；

（3）在其他程序中选择性地重用编译单元。

一个组件的逻辑接口仅限于指能够被客户程序通过编程访问的部分，而物理接口则包括它的整个头文件。如果在一个组件的物理接口中使用了一个用户自定义类型T，即使T是一个封装的逻辑细节，也可能迫使那个组件的客户程序在编译时依赖T的定义。

组件是自我包含的、内聚的和潜在可重用的设计单位。在一个组件内部声明的逻辑结构不应该定义在那个组件之外。一个组件的.c文件应该直接包含它的.h文件，以确保.h文件可基于自己进行语法分析。对于每一个需要的类型定义，都始终包含其头文件，而不是依赖一个头文件去包含另一个，这样，当一个组件允许一个#include指令从其头文件中被删除时不会出现问题。想要改进可用性、可重用性和可维护性，如果某个带有外部连接的结构没有在一

个组件的.h 文件中声明，那么我们应该避免把该结构放在这个组件的.c 文件中。同样的原因，我们应该避免使用局部声明去访问有外部连接的定义。

DependsOn 关系标识组件之间的物理（编译时或连接时）依赖。一个编译时依赖几乎总是隐含一个连接时依赖，而且组件间的 DependsOn 关系具有传递性。

我们可以从一个跨越组件边界的逻辑 IsA 或 HasA 关系推断出一个确定的编译时依赖。在这样的情形下，逻辑 HoldsA 和 Uses 关系隐含了一个可能的连接时依赖。通过利用抽象逻辑关系来推断我们的设计决策的物理衍生物（依赖关系），我们可以在编写任何代码之前预知和修正物理设计缺陷。

我们希望在整个开发过程中跟踪实际的物理依赖，以确保与我们初始的设计保持一致。分析一个大型 C++系统的所有源代码太耗费时间。但是，倘若我们已经遵守了本书中的主要设计规则，从组件的包含图就可推断出它们之间的所有物理依赖。附录 C 中提供了对这样一个工具的描述。

最后，友元关系虽然表面上是一个逻辑问题，却会影响物理设计。在一个组件内部，（局部的）友元关系是那个组件的一个封装的实现细节。为了改进可用性和用户扩展性，一个容器类常常会把同一组件内的一个迭代器视作友元，不会破坏封装。

因跨越了组件边界，（远距离）友元关系变成了一个组件的接口的一部分，并且会导致该组件的封装被破坏。远距离友元关系还会通过允许对一个系统的物理上较远的部分进行密切访问而进一步影响可维护性。

友元关系直接影响访问权限但不隐含依赖。但是，我们避免远距离友元关系的愿望会间接地迫使我们把紧密相关的逻辑实体打包进一个单个的组件内，从而使它们在物理上耦合。忽略这些物理考虑会导致客户去使用封装漏洞，这种封装漏洞可由所有的远距离友元关系引起，甚至可由局部的友元关系对单个的非内联自由（运算符）函数作用所致。

4

物理层次结构

由 DependsOn 关系所定义的组件之间的物理层次结构，与分层所隐含的逻辑层次结构很相似。对于有效理解、维护、测试和重用来说，避免循环物理依赖是中心问题。设计良好的接口应该是短小的、易于理解和易于使用的，但是，这种接口会使用户级的测试代价很大。

本章我们将探讨如何利用物理层次结构来简化“好”接口的有效测试。根据组件间的物理依赖关系，我们引入层次号的概念来帮助表达组件的特点。我们将使用一个复杂的例子来说明隔离测试、分层测试以及增量式测试的价值。最后，我们导出了一种客观度量方法，该方法可以用于量化一个任意子系统中的物理耦合程度。通过使物理设计质量这个概念更客观并且更具体后，这种测量方法就可以帮助我们评估各种设计选择方案的效果。

4.1 软件测试的一个比喻

当一个顾客对一辆小汽车进行测试驾驶时，她（或他）主要注意小汽车作为一个整体的的性能如何——小汽车的操纵、急转弯、刹车等方面怎么样。顾客对主观可用性也感兴趣——小汽车的外观是否漂亮，座位是否舒适，内饰是否豪华，总的来说就是拥有该车的满意度。一般的客户不会去测试安全气囊、球窝接头（ball-joints）或引擎安装，以了解这些部件在所有的条件下的运行是否和所期望的一样。当顾客从一家著名的生产商那儿购买一辆新的小汽车时，会想当然地相信这种重要的低层可靠性。

小汽车若要功能正常，重要的是汽车赖以工作的每个对象也都能很好地工作。顾客不会单独地测试小汽车的每一个部件——但有人会这么做。对小汽车进行质量检测不是顾客的责任。顾客为质量好的产品付款，而质量好的一个方面就是顾客对小汽车的正常工作性能满意。

在现实世界中，小汽车的每一个部件都被设计成带有定义良好的接口，并且在极限条件

下进行了隔离测试，以保证部件在被集成进一辆小汽车之前能满足指定的耐用期限。为了对汽车进行维护，机械师必须随时能够访问各种部件以便诊断和修理。

复杂的软件系统就像小汽车。所有低层次的部件都是具有良好定义接口的对象。每个部件或组件都可被隔离进行重点测试。通过分层技术，这些部件可以集成到一系列日益复杂的子系统中——对每个子系统都有一套测试，以保证这种增量式的集成能够正常地进行。这种分层体系结构使得测试工程师能访问在更低抽象层次上实现的功能，而不会将低层次的接口暴露给产品的客户。最终的产品也要进行测试，以确保产品能满足顾客的期望。

总的来说，一个设计良好的汽车是用分层的部件建造的，制造商对这些部件进行了彻底的测试：

（1）在隔离的条件下；

（2）在一系列部分集成的子系统中；

（3）作为一个完全集成的产品。

一旦装配完成，机械师很容易查看部件，以便进行正确的测试和维护。在软件中，这些概念本质上是一样的。

4.2 一个复杂的子系统

作为软件子系统的一个具体例子，考虑一个计算机辅助电子设计应用程序中的点对点路径问题。该子系统以一种相对简单的描述解决了一个相当复杂的问题。

问题声明

假定：

（1）一个封闭的区域（描述为一个简单的封闭的多边形）。

（2）在封闭区域中的一组障碍物或“洞”（每一个都被描述为一个封闭的多边形），这些洞不彼此重叠（但可以彼此邻接），也不与该封闭区域的周界线重叠（但可以邻接）。

（3）一个起点（表示为一个点）。

（4）一个终点（也表示为一个点）。

（5）宽度（表示为一个整数）。

确定在指定的起点和终点之间是否有一条指定宽度的直线路径存在。如果有，则（有选择地）返回最短直线路径（作为一个开放的多边形）的中心线。

图 4-1 说明了该点到点路径问题的一个实例[①]。这个封闭区域包含 3 个洞，有一条成功的路径可以到达但不重叠。起点由 *s* 表示，终点由 *e* 表示。指定宽度的多种可能的最短直线路径之一由中心线定义，在图中显示为 *s* 和 *e* 的连接。

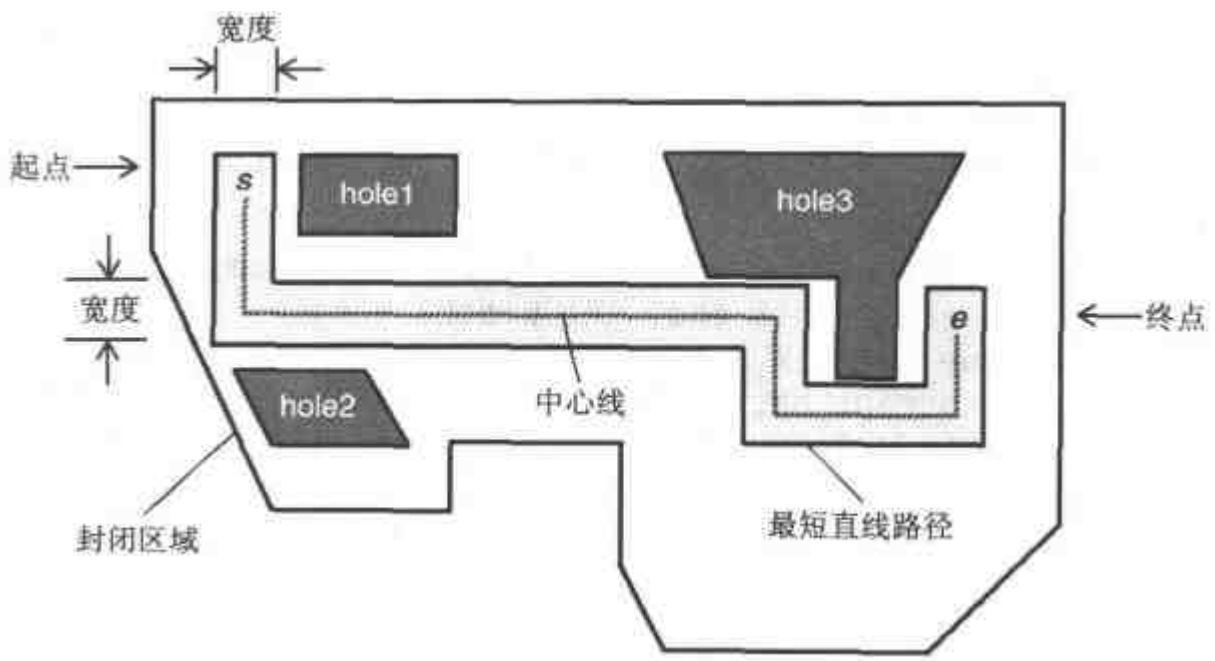


图 4-1 一个点到点路径问题的例子

解决这个复杂问题的一个组件的逻辑接口可能会令人误以为很简单。描述点到点路径子系统的客户程序接口的头文件 p2p_router.h 的全貌如图 4-2 所示。（注册的）类前缀 p2p_表示该组件属于 p2p 软件包，同时消除了属于不同包的类之间的标识符名称冲突的可能性（见 7.2 节）。

```
// p2p_router.h
#ifndef INCLUDED_P2P_ROUTER
#define INCLUDED_P2P_ROUTER

class geom_Point;
class geom_Polygon;
class p2p_RouterImp;

class p2p_Router {
    p2p_RouterImp *d_data_p;

    // NOT IMPLEMENTED
    p2p_Router(const p2p_Router&);
    p2p_Router& operator=(const p2p_Router&);

  public:
    // CREATORS
    p2p_Router(const geom_Polygon& enclosingRegion);
        // Create router for specified enclosing region.
```

① 我们提供了这个真实例子的所有细节。但想从后面的讨论中获益并没有必要理解这个例子的每一个方面，粗略的阅读就足够了。

```
        // The region must be a simple, closed polygon.
    ~p2p_Router();

    // MANIPULATORS
    int addObstruction(const geom_Polygon& hole);
        // Add obstruction; obstruction must be a simple, closed polygon.
        // If obstruction overlaps another obstruction or the perimeter
        // of the enclosing shape, return non-zero with no effect and 0
        // otherwise.  Note: Regions are allowed to touch but not overlap.

    // ACCESSORS
    int findPath(geom_Polygon *returnValue, const geom_Point& start,
            const geom_Point& end, int width) const;
        // Determine whether a rectilinear path of specified width exists
        // in the current obstructed region between specified start and
        // end points.  Return 1 if such a path exists and 0 otherwise.
        // If a path exists and returnValue is not 0, store the center
        // line of any shortest path in (*returnValue).
};

#endif
```

图 4-2 p2p_Router 的完整的头文件

该点对点路径子系统的逻辑接口中使用了两个用户自定义类型。这些类型(geom_Polygon 和 geom_Point）是公共几何类型的软件包（geom）的一部分，该软件包在整个系统中广泛使用。为方便参考，图 4-3 粗略地显示了 geom_Point 和 geom_Polygon 各自的接口。

```
class geom_Point {
    // ...
  public:
    geom_Point(int x, int y);
    geom_Point(const geom_Point& point);
    ~geom_Point() {};
    geom_Point& operator=(const geom_Point& point);
    void setX(int x);
    void setY(int y);
    int x() const;
    int y() const;
};
```

```
class geom_Polygon {
    // ...
  public:
    geom_Polygon();
    geom_Polygon(const geom_Polygon& pgn);
    ~geom_Polygon() {};
    geom_Polygon& operator=(const geom_Polygon& pgn);
    void appendVertex(const geom_Point& point);
    // ...
    int numVertices() const;
    const geom_Point& vertex(int vertexIndex) const;
    // ...
};
```

图 4-3 类 geom_Point 和 geom_Polygon 的接口概貌

该子系统的一个实际实现包含 5000 行 C++源代码（不包括注释），但使用点对点路径组件却是非常容易的。图 4-4 完整地显示了一个简单明了的、运行图 4-1 中例子的驱动程序。注意，这种长的、线性风格的优点是其简单性。这是一个在开发和测试中实际使用的典型的驱动程序。

```
// p2p_router.t.c
#include "p2p_router.h"
#include "geom_polygon.h"
#include "geom_point.h"
#include <iostream.h>

main()
{
    geom_Polygon enclosingRegion;
    enclosingRegion.appendVertex(geom_Point(0, 1000));
    enclosingRegion.appendVertex(geom_Point(0, 600));
    enclosingRegion.appendVertex(geom_Point(700, -100));
    enclosingRegion.appendVertex(geom_Point(2100, -100));
    enclosingRegion.appendVertex(geom_Point(2100, 100));
    enclosingRegion.appendVertex(geom_Point(3000, 100));
    enclosingRegion.appendVertex(geom_Point(3000, -200));
    enclosingRegion.appendVertex(geom_Point(3200, -400));
    enclosingRegion.appendVertex(geom_Point(4500, -400));
    enclosingRegion.appendVertex(geom_Point(5000, 100));
    enclosingRegion.appendVertex(geom_Point(5000, 1000));
    enclosingRegion.appendVertex(geom_Point(0, 1000));

    geom_Polygon hole1;
    hole1.appendVertex(geom_Point(800, 900));
    hole1.appendVertex(geom_Point(800, 700));
    hole1.appendVertex(geom_Point(1400, 700));
    hole1.appendVertex(geom_Point(1400, 900));
    hole1.appendVertex(geom_Point(800, 900));

    geom_Polygon hole2;
    hole2.appendVertex(geom_Point(600, 300));
    hole2.appendVertex(geom_Point(800, 100));
    hole2.appendVertex(geom_Point(1600, 100));
    hole2.appendVertex(geom_Point(1400, 300));
    hole2.appendVertex(geom_Point(600, 300));

    geom_Polygon hole3;
    hole3.appendVertex(geom_Point(2600, 900));
    hole3.appendVertex(geom_Point(2900, 600));
    hole3.appendVertex(geom_Point(3800, 600));
    hole3.appendVertex(geom_Point(3800, 300));
    hole3.appendVertex(geom_Point(4200, 300));
    hole3.appendVertex(geom_Point(4200, 600));
    hole3.appendVertex(geom_Point(4500, 900));
```

```
    hole3.appendVertex(geom_Point(2600, 900));

    p2p_Router router(enclosingRegion);
    router.addObstruction(hole1);
    router.addObstruction(hole2);
    router.addObstruction(hole3);

    geom_Polygon centerLine;
    geom_Point start(400, 800), end(4600, 500);
    int width = 400;

    if (router.findPath(&centerLine, start, end, width)) {
        cout << centerLine << endl;
    }
    else {
        cout << "Could not find path." << endl;
    }
}

// Output:
// john@john a.out
// { (400, 800) (400, 500) (3400, 500) (3400, 200) (4600, 200) (4600, 500) }
// john@john
```

图 4-4 点对点路径问题的简明的驱动程序

4.3 测试“好”接口时的困难

面向对象技术的一种实际有效的应用是把极大的复杂性隐藏在一个小的、定义良好的、易于理解和易于使用的接口后面。但是，正是这种接口（如果被不成熟地实现）会导致开发出来的子系统测试起来极其困难。

例如，p2p_router 组件（图 4-2）只包含四个公共函数：

（1） 一个建立封闭区域的构造函数。

（2）一个析构函数。

（3） 一个在封闭区域内收集障碍物的函数。

（4） 一个决定区域内任意两点之间（不包含目前已收集的障碍物）指定宽度的最短直线路径的函数。

图 4-4 末尾的输出告诉我们，该组件产生了一个答案。现在让我们停一下，并把自己想像为一个委派给该项目的质量保证测试工程师。我们会如何去彻底测试这样一个接口呢？

首先考虑，一般来说对这类问题的实例会有许多等价的好的解决方案。检验一个解决方案是否是连接两点之间（在一个有障碍物的区域内）的给定宽度的最短直线路径，不是微不足道的事，但也可能并不需要付出特别的气力就能完成。要检验该问题的一个解决方案是否是最佳的，一般来说，与发现这个解决方案一样困难。

我们可以通过尝试几种测试实例并进行手工检查来检验输出结果。尽管费时，手工检查在开发过程中可能是很有效的。考虑当开发阶段结束、子系统进入维护/调试阶段时会发生什么。认为我们或者开发者愿意或能够手工检查每一个版本的每个子系统的输出是不现实的。

> **定义**：回归测试指的是这样的规程：运行一个程序（该程序被给定了一个有固定期望结果集合的特定输入），比较其结果，以便检验程序从一个版本升级到另一个版本时是否能够继续如所期望的那样运行。

一种通常用于帮助自动回归测试的方法是在系统顶层运行大量的测试用例并捕捉运行的结果，然后手工检查一遍这些结果以确定它们的准确性。在每个版本发布之前获得新的结果，并将新结果与原来的结果进行比较。可以这样推测，如果新的输出结果与旧的输出结果完全一样，则该子系统是正确的。

对于许多复杂的问题来说（包括这个问题），回归测试的一个明显缺陷是可能有多重正确的解决方案。尽管点对点路径子系统的每个组件都可以有完全可预知的特性，但规范中仍然有空间允许开发者改变 p2p_Router 的实现，这种实现对一个给定的输入会产生不同的（但同样好的）最终结果。

在一个小得多的规模上，考虑一个在某个集合上的简单迭代器的规范。一般来说对元素的出现顺序没有限制，集合里的每个元素只能出现一次。验证一个迭代器在隔离情况下工作正常并不困难。但是当迭代器嵌入到一个复杂子系统（如以 p2p_router 组件为首的子系统）的实现中时，有效测试迭代器可能很难。

尽管点对点路径系统在最优路径存在的情况下一定会产生一个最优的结果，但很多复杂问题在合理的时间内很难用最优的方案解决。在这些情况下，可以使用启发式（heuristic）方法来产生好的（但不一定是最优的）解决方案。启发式技术经常采用一种智能的 trial-and-error 策略的形式，它们本质上是不可预测的，要用实验来决定哪个启发式技术倾向于产生最优的解决方案。依赖于启发式方法的软件抵制高层的回归测试，因为在启发式技术中的任何改进都会使回归数据无效。

在高层的接口上测试复杂的、基于启发式技术的软件更加复杂，因为在较可预测的基础组件中的失败，可能不会引起整个子系统完全失败，但这些潜在的错误会悄悄地降低子系统输出结果的质量。因为不是总能验证结果的最优性，所以这种质量的降低可能很难探测到。

比基于启发式系统的伪随机行为[①]更糟的是，那些与使用异步通信的系统有关的完全不可预测的行为。这样的系统产生的结果一般来说是不可重复的。在这种情况下，高层回归测试实际上是无用的。

在设计中最小化“表面区域”（即，提供足够的但最小的接口）是优秀软件工程的基石。但是也有非常出人预料的事情：我们非常努力得到的接口却对传统的测试技术设置了可怕的

① 要想更多地了解伪随机功能，见 **plauger** 中的 rand()（第 13 章，337 页）。

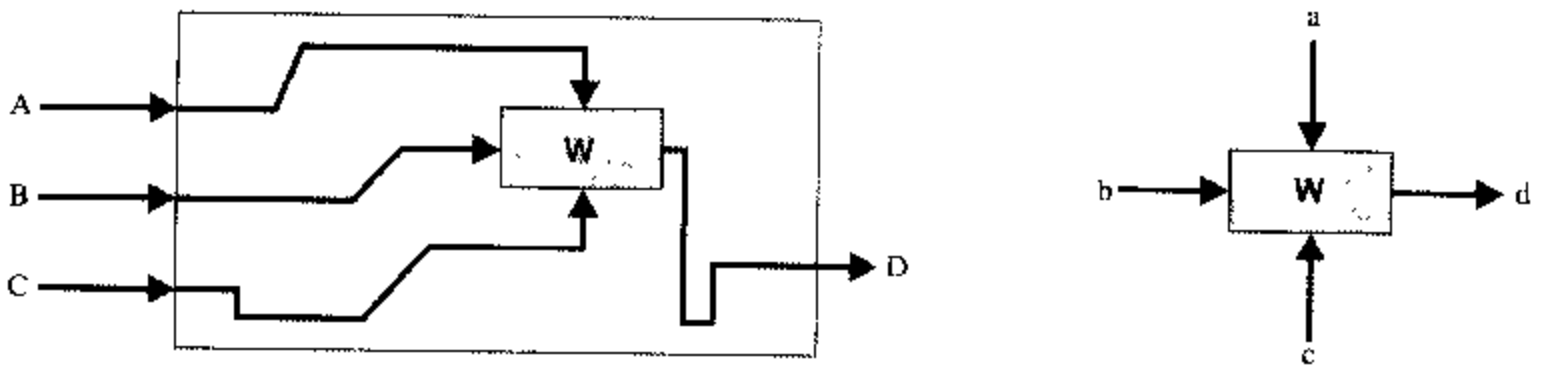
障碍。幸好我们有可以用来解决这些测试问题的技术。以下谚语用在这里很恰当：一盎司预防相当于一磅治疗。

4.4 易测试性设计

质量设计的一个主要部分是易测试性设计（Design For Testability，DFT）。DFT 的重要性在集成电路（IC）工业界是公认的。在许多情况下，只从外部管脚来测试 IC 芯片是不现实的，一些芯片有超过一百万个的晶体管。

当一个 IC 芯片制成之后，它就充当了一个“黑盒子”，可以只通过外部输入和输出（管脚）进行测试。图 4-5（a）描述了只使用提供给芯片本身的普通客户的接口来测试一个硬件子系统 w 的过程。为了测试 w，不仅有必要弄清楚 w 的一个好测试组合的组成，而且也要知道如何穿过芯片传送该测试组合给 w 的输入。如果不出错的话，w 产生的每个结果必须从 w 的输出传送到芯片本身的某个输出，以便观察和检验 w 的行为是否正确。要确保这种信息的传送，需要了解有关整个芯片的详细知识——即那些与 w 的功能毫无关系的知识。

IC 芯片的 DFT 有一种形式称为 SCAN，SCAN 使用那些只用于测试目的的额外的管脚和附加的内部电路来完成。使用这些特殊部件，测试工程师能够隔离芯片内部的各种子系统。在这样做的过程中，他们能够直接访问内部子系统的输入和输出并且直接测试它们的功能。换句话说，这种 DFT 方法试图授权测试者直接访问子系统，从而消除通过整个芯片传送信号的开销。使用这种方法，子系统的全部功能都可以有效地探测到，如图 4-5（b）所示（没有考虑在更大的系统中如何使用该子系统的细节）。



（a）从系统层次测试一个组件　　（b）直接测试一个组件

图 4-5　在集成电路中的易测试性设计

第一次应用 DFT，可以极大地提高质量。但是，IC 设计者们不喜欢这些额外的设计要求。这不仅需要额外的考虑，而且会使得它们的设计更大因而导致成本更高。许多设计者感到沮丧，认为这种严格的方法是对他们的创造力的侵犯。

目前 DFT 是 IC 工业的一种标准。没有哪个称职的硬件工程师会在考虑设计一个复杂芯片时不直接考虑易测试性问题。比较起来，大型软件系统的功能可能比在最复杂的集成电路中的功能复杂好几个数量级。令人吃惊的是，很多软件系统在设计过程中并没有适当的计划

来确保软件是可测试的。在过去的十年中，要求软件具备易测试性的努力经常遇到在 IC 工业界中遇到的同样的挫折。通常是人而不是技术本身对解决一个技术问题形成巨大的挑战。

原　则

对于测试来说，软件中的一个类类似于现实世界中的**实例**。

就像 IC 设计一样，面向对象软件涉及相对少量的类型的创建，然后，这些类型被重复地实例化以形成一个工作系统。例如，在许多软件系统中，一个 String 类是一个基本类型。在一个典型的系统调用中，可以创建该类的许多实例。

两个设计规范都要求对这些类型中的功能进行彻底的测试，以确保实例化时的行为正确。但是，软件设计不像 IC 设计那样要求对类型的每个实例都必须进行物理故障测试，软件对象对于这样的故障具有免疫力。如果一个类的实现是正确的，那么，由定义可知，该类型的所有的实例也都被正确地实现。

从测试的观点来看，每个软件类型都类似于现实世界的实例。如果直接对它进行操作，而不是试图将它作为一个更大系统的一部分来测试，那么测试一个 String 类的功能是最简单和最有效的。并且，和 IC 测试不同，我们自动地拥有对该软件子系统的接口（String 类）的直接访问权。

原　则

对整个设计的层次结构进行分布式测试，比只在最高层接口进行测试有效得多（就每美元测试费用而言）。

换种方式来说，假设我们只有 X 美元可花在测试上，那么如果我们把分布式测试遍及到整个系统，我们就可以获得比只从最终用户接口进行测试所能获得的更完全的覆盖。

让我们再次考虑图 4-2 中的 p2p_router 组件例子。即使假设整个行为都是可预测的，试图从最高层次来对该组件进行整体测试的效率也是很低的，尤其是在所给定的接口极小的情况下。这就像在 IC 测试中（见图 4-6）只通过两个管脚来测试一个有一百万个晶体管组成的微处理器一样！[①]

软件测试本质上比硬件测试要容易，因为在系统中产生的类的实例与同一个类在系统之外独立产生的类的实例**没有什么不同**。如果一个复杂的软件子系统真的类似于 IC 芯片，其实

① 其他种类的 IC 测试策略，例如集成自测试（Build-In Self Test，BIST），在芯片上放置了附加的电路，该电路激活后可校验芯片是否正常工作，而不必向接口传递特定的信息。BIST 有点类似于软件中使用的 assert 语句。添加公共的功能（例如 testMe()）后则更加相似。但是，我们的软件体系结构中的物理层次结构使得我们可以获得同样的结果，而不必在一个组件的接口中添加任何专门用于测试的功能。

现将整个地驻留在一个单一的物理组件中。如果在 p2p_router.h 中声明的功能全部在 p2p.router.c 中实现，那么我们将有可能被迫违反封装的原则，在公共接口上提供额外的功能——仅仅是为了能够进行有效的测试。

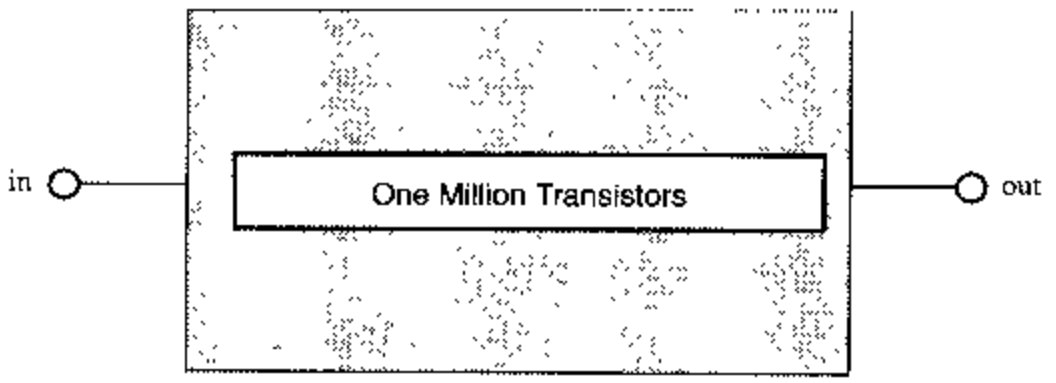


图 4-6 虚构的只有两个管脚的、高度抗拒测试的芯片

幸好，点对点路径的实现并非驻留在一个单一组件内。该实现被特意分散到整个组件物理层次结构中。即使一个 p2p_Router 对象的客户不能编程访问构成路径实现的分层对象，但测试工程师仍然可能识别有可预测行为的子组件，对这些子组件在隔离的情况下进行测试和检验会更有效。

4.5 隔离测试

在一个设计良好的模块化了系统中，可以对许多组件隔离测试。考虑一个涉及点对点路径了系统的非常真实的情况，该子系统最终将支持所有角度的几何形状。该系统目前暂时还处于原型阶段并且只能处理曼哈顿（90 度）角。该点对点路径系统是基于对象的，因此它在许多对象之上进行分层，这些对象目前大多数支持所有的角度。由于有些组件还没有升级为支持所有的角度，所以该 p2p_router 本身只能接受曼哈顿测试案例。

原 则

独立测试降低了一部分与软件集成相关的风险。

考虑如图 4-7 所示的 p2p_router 的物理体系结构。通过设计 p2p_router 使它的每个子系统都可以单独地开发和测试，我们可以确保它们升级后的每一个功能都是合适的，即使直到将来某一天它们才能通过完全的路径接口来检验。如果发生了程序方面的错误，可以并行地检测和修复这些错误。

一个可供选择的、不很严格但被广泛使用的软件集成的“方法”是，一直等到所有的软件都已经到位了再来试用它。这种方法通常称为大爆炸（big bang）方法。这个名字多少有些误导——预期的“巨响”经常只是轻微的嘶嘶声。

集成测试会检测出大多数规范中的错误。当集成系统不能按预期执行时，开发小组必须

尽快诊断出问题。他们不可避免地会发现很多代码错误，这些错误本质上与系统集成本身没有关系。独立测试至少能使这些代码错误在开发过程的更早期就被检测出来并进行修复。

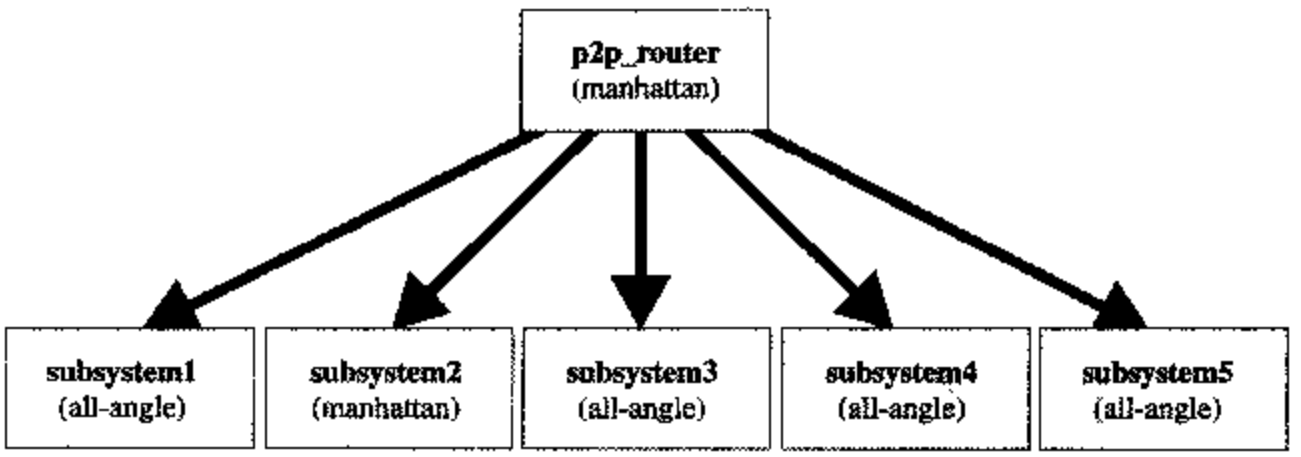


图 4-7 一个 p2p_router 实现的物理依赖关系

定义：隔离测试是指这样的规程：独立于系统的其他部分对单个组件或子系统进行的测试。

原 则

隔离测试组件是确保可靠性的有效办法。

在一个复杂系统的最底层，组件经常被深度优化，这增加了细微错误出现的可能性以及对详细的回归测试的需求。例如，仔细设计的特定对象内存管理经常可以使运行时性能提高一倍。但是，常规的内存管理程序很容易出错，并且这些错误最难检测和修复。在一个隔离的组件测试驱动程序中设置全局操作符 new 和 delete，可以确保内存管理方案在各种条件下正常工作，包括那些在实践中很少碰到的情况。

不是所有的程序都可使用可重用组件中的所有功能。例如，如果程序没有调用 Stack 类的 pop()成员函数，就没有办法只通过测试那个程序来测试 pop()。即使有一个特殊的程序调用了每一个函数，也可能有这样的状态：对象应该可以正确运转，但是环境软件不允许它完成。

考虑一个 String 类，该类被开发为一个解释器的一部分。该解释器从未见过长度为 0 的标识符，因此它决不会产生一个空 String 来表示该标识符。（通过为 string 组件特别设计的一个完全测试，这个边界条件肯定可以处理好。）随着系统的演化，我们可能在将来的某个时候在同一系统的其他部件中重用 String 类，但以新的方式（例如，拥有 String 变量）。此时一个空 String 的实例**可能**出现在本系统中。这种增强可能是在系统的相当高的层次上进行的，但是潜在的错误存在于最低的层次——在 String 类中——该类已在相当长的时期内工作得很“完美”！

在大型项目中，String 类的作者与有效地增强该类并暴露出问题的人可能不是同一个人。检测并修复这样的错误，暂且不提随后出现的失败，比从一开始就通过早期的组件级隔离测试来避免这样的错误代价要高得多。

对每个使用库程序（如，iostream）的系统都进行测试，以验证所需要的 iostream 功能是否工作正常，这是一种多余的、不必要的高昂代价。人们已经假定 iostream 如所预期的那样工作。对于大型项目，可能会有很多内部开发的应用程序库。没有一个单个的可执行程序会利用所有这些功能，但是所有这些程序都必须进行完全的隔离测试。

我们可以通过对组件本身的测试进行分组来避免冗余。在这样做的时候，我们扩展了面向对象设计的概念，作为一个单独的单元，它不仅包括组件，而且包括支持测试和支持文档。此外，对精心编写的组件级测试，可以通过给预期的用户提供一整套小但易于理解的例子来重用。现在可以在一个地方对每个组件提供的功能进行完全的测试，依赖于这些组件的客户可以合情合理地认为这些组件是可靠的。

隔离测试对于找出由增强引起的低层次问题是很理想的，并且对将一个系统移植到一个新的平台特别有用。这些低层测试可确保维持基本功能，并使测试人员很容易发现矛盾之处。偶尔也会有一些缺陷逃过了局部检测，而被更高层的测试捕捉到。应该及时更新低层次组件测试，以便在缺陷修复**之前**暴露错误行为。这样通过使组件的测试独立于任何特定的客户，既可以方便修复又可以保护模块性。

对于隔离测试有一个关键点，过了这个关键点，隔离测试的回报就会减少。例如，将一个简单的 List 对象的 Link 类定义放在一个不同的组件中，以便可以隔离测试它，这样做是荒谬的，原因有两个：

（1）一个 List 对象的正常的操作会完全使用 Link 的功能。

（2）附加的组件会不必要地增加系统的物理复杂性，使系统更难以理解和维护。

应该基于对开销/利益的分析客观地决定组件级隔离测试的这个点，而不是只凭某个开发者的好恶来测试。

4.6 非循环物理依赖

对于一个能被有效测试的设计来说，一定能够将其分解成复杂性可以控制的功能单元。组件是实现此目的的理想部件。考虑图 4-8 中描述的组件 c1、c2 和 c3 的头文件。注意我们已经在组件头文件 c2.h 和 c3.h 中声明了类 C1 但没有提供其定义，因为没有必要定义一个出值返回的类[1]来声明那个函数。更多有关内联函数的内容见 6.2.3 节。

读者可以观察到（见 3.4 节）c1 没有隐含依赖于任何其他的组件。类 C2 在其接口上使用类 C1。因此，组件 c2 很可能依赖于组件 c1，但是，我们希望它不依赖于 c3。类 C3 在其接口上使用 C1 和 C2，因此，c3 很可能依赖于 c2 和 c1。该系统中的隐含依赖形成了一个有向的非循环图（Directed Acyclic Graph，DAG），如图 4-9（a）所示。

① 对于通过值传递的用户自定义类型也一样，但是要参见 9.1.1 节。关于内联函数使用用户自定义类型的情况请见 6.2.3 节。

```
// c1.h
#ifndef INCLUDED_C1
#define INCLUDED_C1



class C1 {
    // ...
  public:
    C1 f();
};

#endif
```

C1.h

```
// c2.h
#ifndef INCLUDED_C2
#define INCLUDED_C2

class C1;


class C2 {
    // ...
  public:
    C1 g();
};

#endif
```

C2.h

```
// c3.h
#ifndef INCLUDED_C3
#define INCLUDED_C3

class C1;
class C2;

class C3 {
    // ...
  public:
    C1 h(const C2& arg);
};

#endif
```

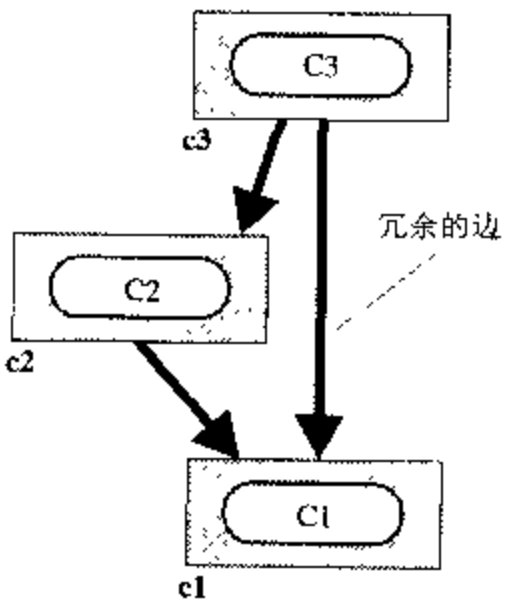
C3.h

图 4-8　带有非循环隐含依赖的组件

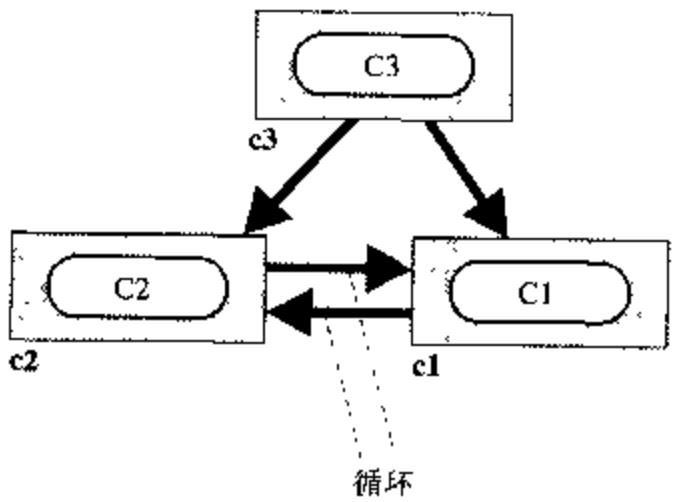
不含循环的组件依赖图非常有利于实现易测试性，但不是所有的组件依赖图都是非循环的。为了弄清楚原因，考虑这样的情况：如果我们将 C1::f 的返回类型从 C1 改变为 C2（如下所示），会产生什么结果？

```
class C1 {
    //...
  public:
    // C1 f();          // old
    C2 f();             // new
};
```

现在 C1 在其接口上使用 C2 并且（可能）依赖于它。这个修改后的系统的隐含组件依赖图现在有一个**物理**循环，如图 4-9（b）所示。



（a）非循环依赖



（b）循环依赖

图 4-9　非循环的和循环的物理依赖

有非循环物理依赖的系统［如图 4-9（a）所示的这个］远远要比那些带有循环的系统更容易进行有效的测试。一个系统中的组件依赖只要是非循环的，就（至少）存在一个测试该系统的合理顺序。由于组件 c1 不依赖于任何别的组件，应该首先编写在隔离的条件下验证其功能的测试程序。接下来我们看到组件 c2 只依赖于组件 c1。因为我们能够为 c1 编写有效的测试程序，我们可以假设 c1 的功能是正常的，因此可以给对由 c2 添加的功能编写测试程序。我们不需要重新测试其中的 c1 部分，因为其功能已经被覆盖了。然后我们来看依赖于 c1 和 c2 的 c3。因为我们已经编写好了验证由 c1 和 c2 提供的功能的测试程序，所以我们只需要对付由 c3 实现的额外功能。

4.7 层次号

在本节中我们将介绍一种方法，将组件（基于它们的物理依赖）划分成等价的类（称为层次）。每个层次都与一个非负整数位标相联系，在本书中称为**层次号**。接下来的几个段落会描述层次号的起源以及它们如何被用于它们最初的上下文中。然后我们将这些精心建立的概念应用于一个新的上下文中——软件工程。

4.7.1 层次号的起源

层次号的概念是借用数字电路、门层次、零延迟电路仿真领域的概念[1]。在这里，一个 **gate**（门）实现一个低层次布尔功能块。每个门有两个或者更多的称为 **terminal**（接头）的连接点。一个 **circuit**（电路）由一个相互连接的门的集合构成。像一个门一样，一个电路也有输入接头和输出接头。初级输入（primary input）是到电路本身的输入。通过成对的称为 **wire**（电线）的接头连接器，这些输入与一些电路中的门的输入相连接。这些门的输出又通过电线与别的门的输入相连接，如此一直下去。一个有四个初级输入（a、b、c 和 d）的简单电路如图 4-10（a）所示。

仿真一个电路涉及用逻辑值设置其初级输入，然后依次求出每个（分层的）门的值。但是在一个特定的门的值能求出之前，我们必须证实其输入是有效的，方法是确保输入到这个特定门的所有门的值已经求出。

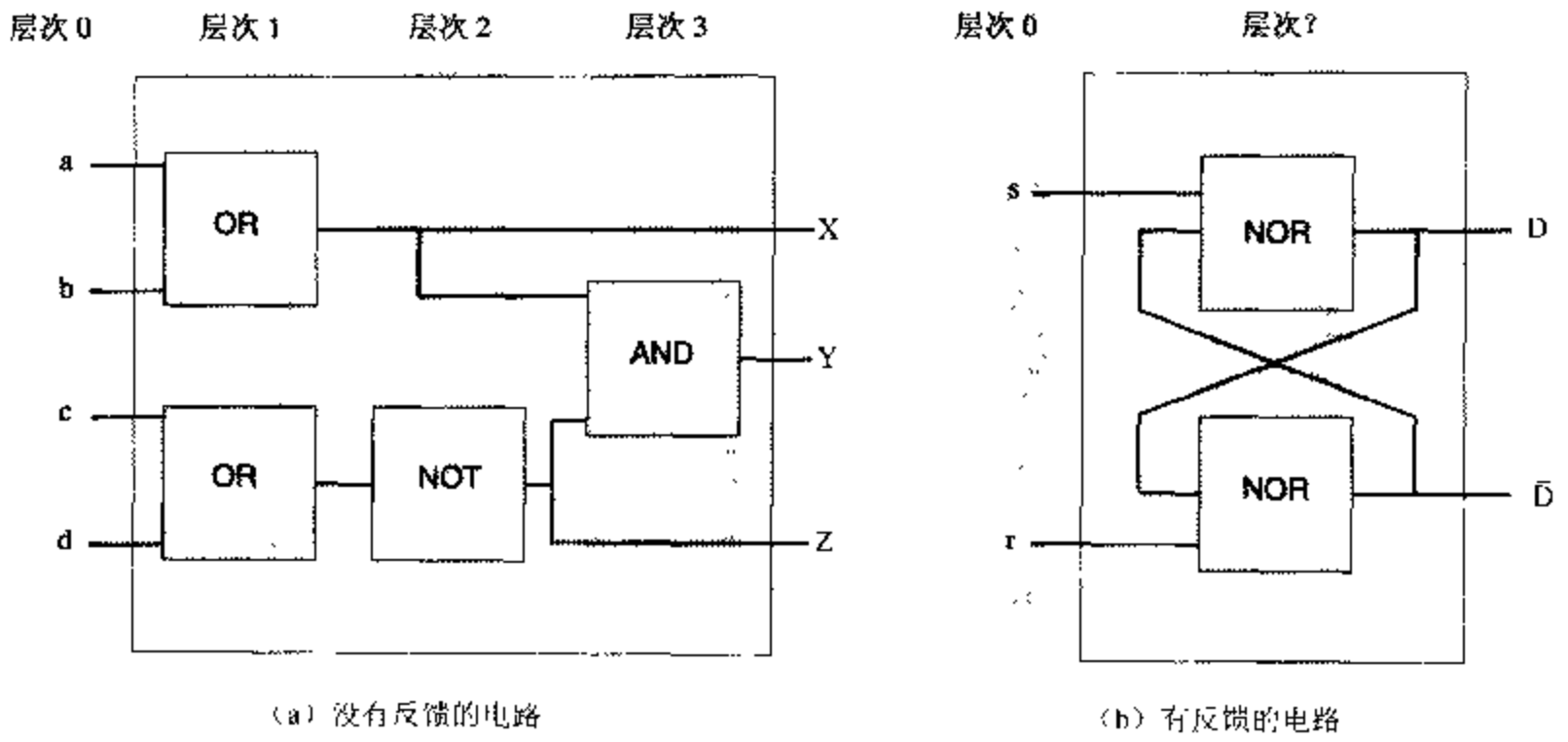
一个电路是一种图。这里，门和初级输入作为图的节点，而电线作为（有向）边[2]。在这个上下文中的层次号表示的是特定的门到初级输入的最长路径。初级输入被定义为第 0 层。

---

① 零延迟渐进法主要用在一种特殊的电路仿真器中（称为 fault simulator）。这种硬件和软件相似性的发现，部分源于作者在哥伦比亚大学与 Stephen H. Unger 教授所进行的博士学位的研究工作。

② 门本身影响边的方向，该方向反映了门对其输入源（例如，要么是一个初级输入，要么是电路中另一个门的输出）的依赖。

通过按照递增层次的顺序计算这些门的值，我们可以保证每个门的输入都是有效的。



（a）没有反馈的电路　　（b）有反馈的电路

图 4-10　有与没有反馈的逻辑电路

初级输入值是设定的，并不需要计算。在仿真期间，第 1 层的门只接受初级输入。这些门首先被计算，计算的顺序是任意的。接着计算第 2 层的门。由于第 2 层的门只接受一个或多个第 1 层的门（也可能接受初级输入）的输入，所以我们可以保证，在这个时候第 2 层的门的所有输入都已被求值。由于在第 N 层的门只依赖[0…*N*–1]层作它的输入，所以按照层次顺序计算门的值可以保证得到一个成功的仿真。

在图 4-10（a）中，一个在第 1 层上的 OR 门是 NOT 门的惟一输入，使得 NOT 门成为第 2 层的门。而 AND 门的输入包括第 1 层的 OR 门和第 2 层的 NOT 门。从 AND 门到初级输入的最长路径是 3（通过 NOT 门到初级输入 c 或 d）。AND 门属于最高层次（3），并且最后被求值。

原　则

每个有向非循环图都可被赋予惟一的层次号；一个有循环的图则不能。

注意，图 4-10（b）中一对交叉耦合的 NOR 门，从任一门到任一初级输入（r 或 s）的最长路径是无约束的。该电路不能被层次化——也就是说，它不能被赋予惟一的层次号。使一个电路可层次化的特性是它没有反馈。没有反馈使得电路在性质上更易于理解、开发、分析和测试。正是由于这些原因，反馈在大型系统中只是在非常严格的环境下才使用。出于完全相似的原因，“没有反馈”也正是我们希望软件设计所拥有的特性。

定　义：一个可被赋予惟一的层次号的物理依赖图称为是**可层次化的**（levelizable）。

4.7.2 在软件中使用层次号

让我们回到软件工程。如果软件系统中的组件依赖正好形成一个 DAG 图，那么我们可以给每个组件定义**层次**。

> 定义：
> 层次 0：一个在我们的软件包之外的组件。
> 层次 1：一个没有局部物理依赖的组件。
> 层次 *N*：一个组件，它在物理上依赖于层次 *N*–1 上（但不是更高层次上）的一个组件。

在这个定义中，我们假设所有在我们当前软件包之外的组件[①]（例如：iostream）都已经测试好了并且会正确地发挥作用。这些组件被看作“初级输入”并具有 0 层次号。一个没有局部物理依赖的组件被定义为层次 1。另外，一个组件被定义为比它所依赖的组件的最高层次高一个层次。

图 4-11 显示的是 3.4 节的图 3-17 的组件依赖图，该图恰好没有任何循环，因此是可层次化的。层次号显示在每个组件的右上角。组件 chararray 在本地并没有依赖任何其他组件，但是依赖于标准库组件（这种组件被设定为层次 0），因此 chararray 是层次 1。一个层次号为 1 的组件（如 chararray）如果只依赖于编译器提供的库，这样的组件就称为叶子组件。总是可以对叶子组件进行隔离测试。

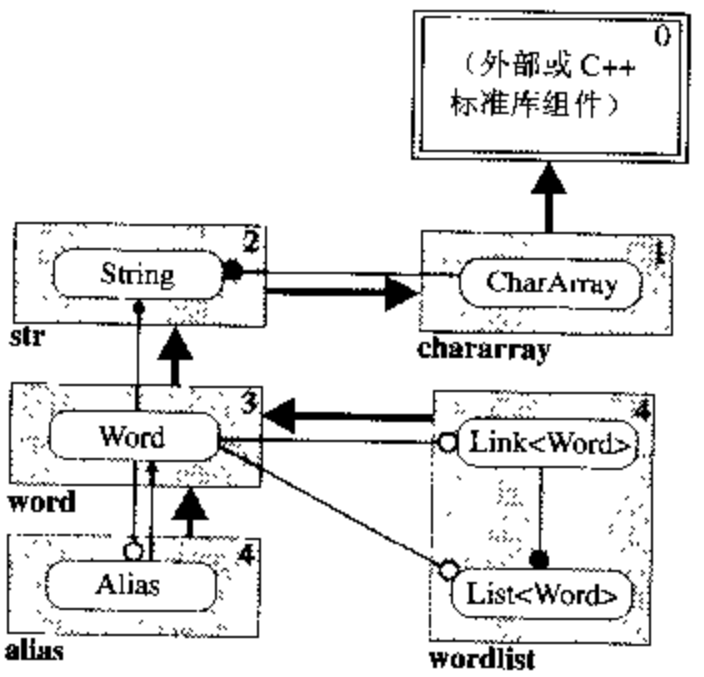


图 4-11 层次化的组件依赖图

① 假设现在 package 的意思是当前项目目录。

组件 str 只依赖于 chararray。str 的层次号为 2，比 chararray 的层次号多 1。组件 word 依赖 str（并且间接地依赖 chararray），由于 str 的层次号为 2，所以 word 的层次号为 3。因为 word 在第 3 层，而且 alias 直接依赖的惟一组件是 word，所以 alias 在第 4 层。wordlist 组件也直接依赖于 word 但不依赖于 alias，所以 wordlist 也是在第 4 层。

> **定 义**：一个组件的层次是一个最长路径的长度，该路径是指从那个组件穿过（局部）组件依赖图到外部（或由编译器提供的）库组件的集合（可能为空）的路径。

由于有了层次化的组件图，所以很容易指出系统中哪些组件是可隔离测试的。在图 4-11 中只有一个可独立测试的组件：chararray。通过从最底层开始（即层次 1）并在移到下一个更高层次之前测试当前层次的所有组件，我们可以保证当前组件依赖的所有组件都已经测试过了，在图 4-11 的例子中，我们可以最后测试 wordlist 或者 alias，但是其余的测试顺序则由层次号决定。

原 则

在大多数真实情况下，如果大型设计要被有效地测试，它们必须是可层次化的。

注意，层次化这一术语适用于物理实体而不是逻辑实体。虽然一个非循环的逻辑依赖图也可能隐含着有一个可测试的物理划分存在，但是（物理）组件的层次号以及我们的设计规则，隐含了有效测试的一个可行顺序。此外，图 4-11 标识出了哪些子系统可以被独立地重用。图 4-12（右栏）列出了每个组件重用时必须相应用到的其他一些组件。

为测试或重用	你还需要
$chararray_1$:	
$string_2$:	$chararray_1$
$word_3$:	$string_2$ $chararray_1$
$alias_4$:	$word_3$ $string_2$ $chararray_1$
$wordlist_4$:	$word_3$ $string_2$ $chararray_1$

图 4-12　可独立重用的子系统

可层次化设计的另一个显著优点是它们更容易被渐进地理解。理解一个可层次化设计的过程可以按照某种顺序（自上而下或自下而上）。不是所有的分层次设计的子系统都是可重用的。但是，如果要可维护，每个组件都必须有一个良好定义的接口，无论其适用性如何广泛，该接口都应该很容易理解。

当然，不是所有的设计都是可层次化的。有时一个设计是否可层次化不能从一个逻辑图中很明显地看出。考虑图 4-13，读者能否从该图中说出这个设计中的组件是否是可层次化的？

在这个设计中的逻辑关系并**没有**隐含任何组件中的循环物理依赖。事实上，我们的设计规则可以确保没有隐藏的物理依赖（例如，对外部全局变量的依赖）。图 4-14 显示了隐含的组件依赖以及最终得出的该设计的组件层次号。

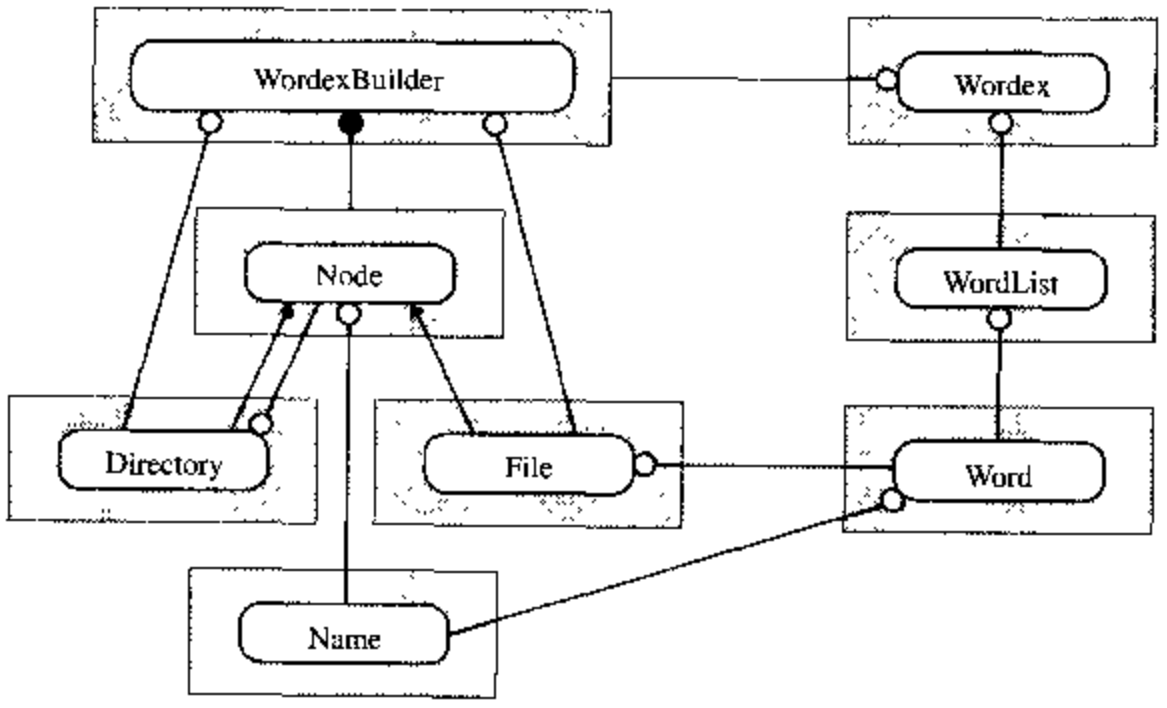


图 4-13 这个设计是可层次化的吗

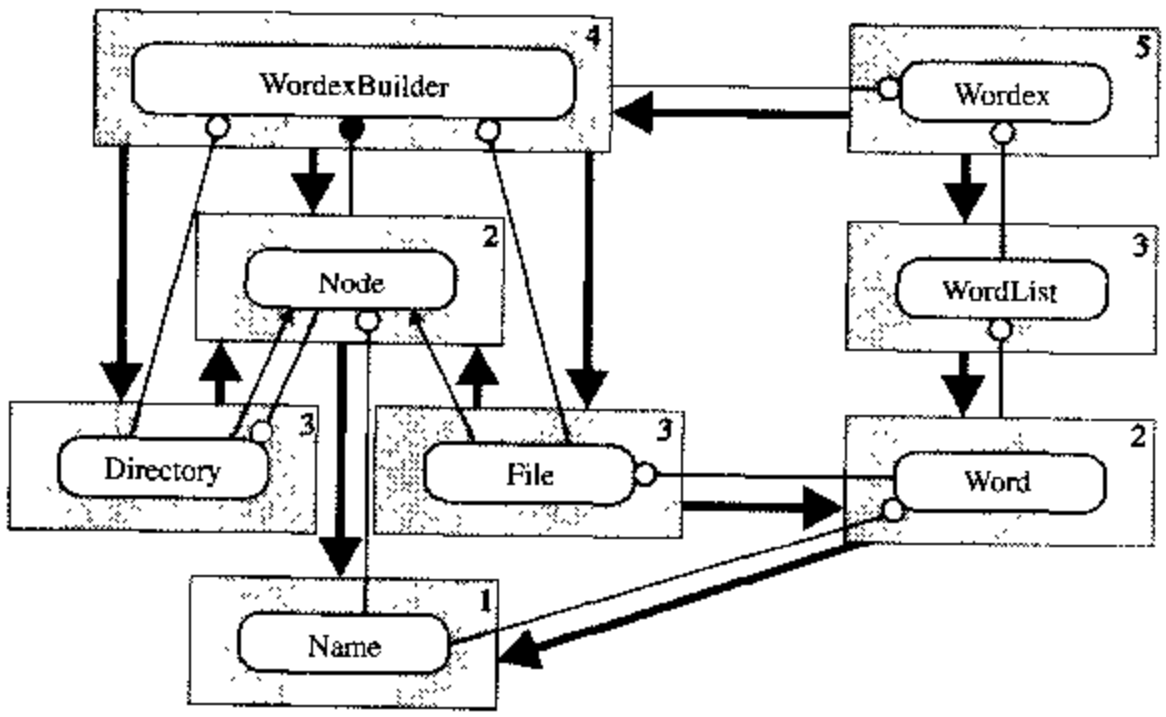


图 4-14 组件/类图

这个组件/类图是混乱的，并且包含的信息比理解系统的物理结构所需要的信息更多。如果我们重新安排组件的位置并且消除逻辑细节，我们可以得到非常清晰的组件依赖图，即图 4-15。

图 4-15 所示的依赖图中有一条冗余的边。组件 wordexbuilder 直接依赖组件 directory、file 和 node。正如我们在 3.3 节所介绍的那样，依赖关系具有传递性。因为 directory（和 file）依

赖 node，wordexbuilder 依赖于 node 的关系是隐含的，可以删除，不会影响层次号。图 4-15 所示的图很明显是非循环的，并且是典型的满足一个特定应用的子系统的图表。在这个抽象层次上，设计看来是合理的。

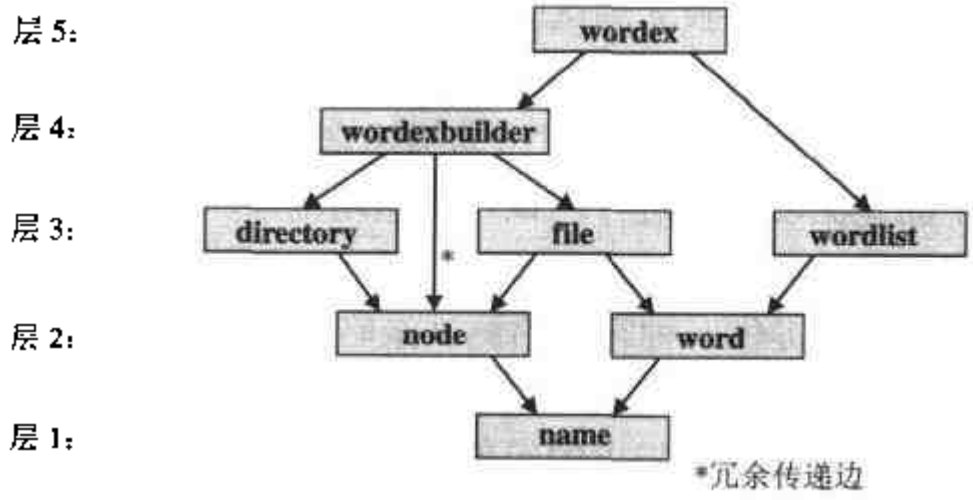


图 4-15　组件（直接）依赖图

这种分析的最大价值之一是，解开组件依赖图后，我们能够对物理设计的完整性作出实质性的定性评论，甚至不需要应用领域的最小的讨论。使这个过程自动化的简单工具很容易编写，并且实践已经证明它们对于大型项目的价值无法衡量。附录 C 描述了一个简单的组件依赖分析器。

4.8　分层次测试和增量式测试

组件是一个系统的基本构造部件。每一个组件都是不同的。每个都是*物理设计模式*的一个“实例”：“组件”。表面上，它们都有相同的基本物理结构——一个物理接口（.h 文件）和一个物理实现（.c 文件）。

在这个意义上，实现和测试一个软件系统就象建造一座房子。总体体系结构完成之后，砖（即**组件**而不是对象）一块一块地垒起来。每块砖的成功添加不仅依赖它自己的完整性，而且依赖于泥灰的完整性，泥灰被用来集成这块砖和这块砖所依赖的更低层次的砖。按照这个办法很容易检查每块砖的缺点。但是，一旦完成，这座房子经常很大也很复杂，为检查每个细节设置了太多的障碍。

> **定义**：分层次测试是指在每个物理层次结构上测试单个组件的惯例。

在建造房子的比喻中，一块砖代表一个惟一的组件（即，一个或者更多的类），不是单独的实例。在实践中，彻底的测试要求在装配每个组件之前测试该组件的完整性。在安装之前试运行每一个组件决不能排除后来在隔离的情况下进行更彻底检查的可能性。在物理层次结构的每个层次上测试组件接口，在本书中称为**分层次测试**（**hierarchical testing**）。

原　则

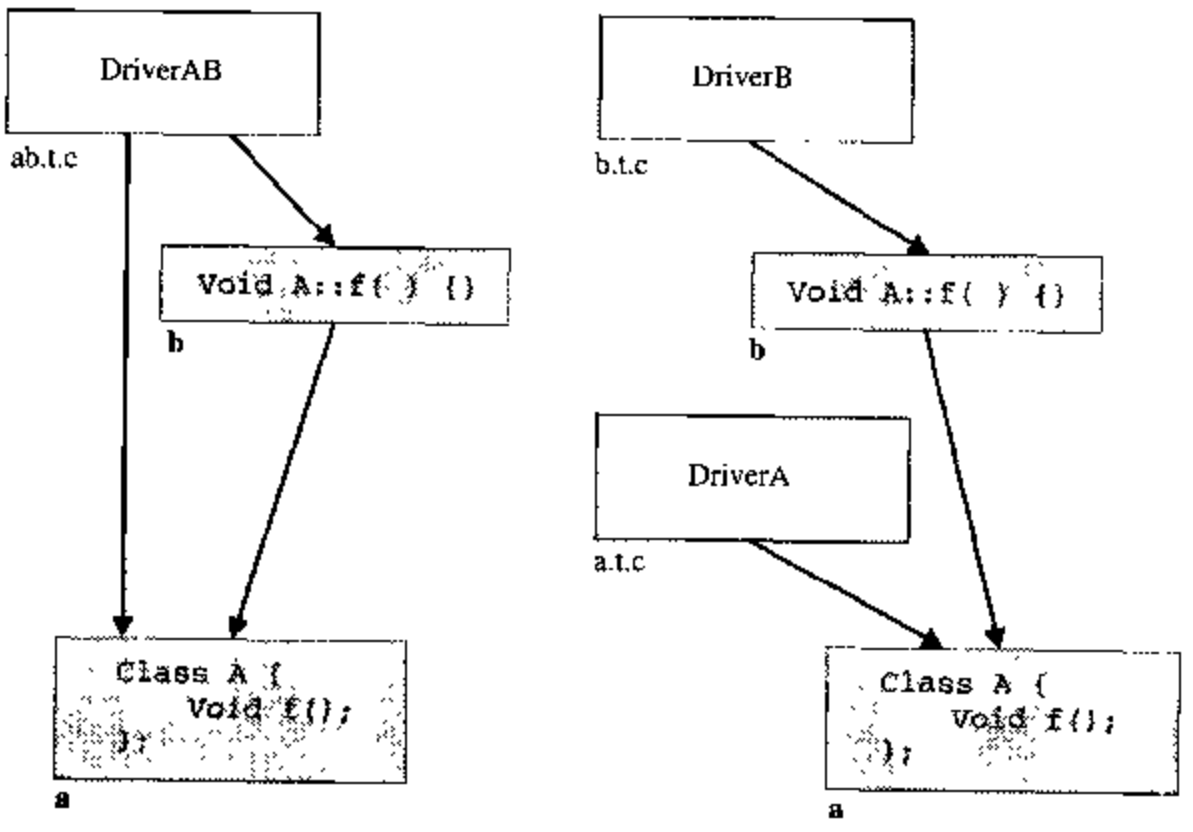
分层次测试需要为每个组件提供一个独立的测试驱动程序。

在这个方法中，为每个组件提供的独立测试驱动程序是由开发者建立的，同时用组件本身去试运行和验证在那个组件中实现的功能。这种测试驱动程序不仅在开发期间被广泛使用，而且以后还可用于质量保证（quality assurance），帮助描述它所验证的组件的预设行为。

每个组件都可使用单个测试驱动程序进行测试，该驱动程序试运行了在该特定的组件中实现的功能。物理依赖决定组件被测试、开发和运行的顺序。层次号可用于描述一个软件包中的局部组件的相对复杂性，还可提供一种客观的测试策略。

单独的驱动程序是保证遵守物理设计规则所必须的——否则我们将不能证明，在一个组件中声明的功能只能在由组件的依赖图所指出的组件子集中得到。为了说明为什么是这样，考虑一个违反设计规则的情况（显示在图 4-16 中），其中的组件 a 定义了一个有成员函数 f() 的类 A，还有一个组件 b（在 a 的上层），它非法地实现了 A::f()。

正如图 4-16（a）所示，一个单一的、同时连接到 a 和 b 的测试驱动程序不能检测出这种违反主要设计规则的情况。谁都可以从依赖图中知道，组件 a 是独立于组件 b 的，所以可以独立于组件 b 进行重用。如果某人试图独立于组件 b 重用组件 a，并且调用 f()，A::f()将在连接时作为一个没有定义的符号出现。



（a）用于许多组件的一个驱动程序　　（b）每个组件一个驱动程序

图 4-16　对单独的组件驱动程序的需求

在图 4-16（b）中，我们提供了不同的驱动程序来试运行每个组件中的功能。当连接组件

a 的驱动程序时，组件 b 被故意排除在连接过程之外。如果 a 的驱动程序是非常全面的（即，至少调用了每个函数一次），那么如果 A::f 没有定义，错误将在连接时被捕捉到——也就是说，甚至不必运行这个驱动程序。同样的技术也可以用来检测不可层次化的组件。

坚持编写单独驱动程序的另一个强制性的原因是，单个的组件一般都为一个测试驱动程序进行彻底测试提供了丰富的功能。在一个单个的驱动程序中对几个组件进行集中测试会导致过大的测试（或者，更可能的是，不足的测试）。

图 4-17 说明了分层次测试策略的抽象物理结构。在层次 1 上的每个组件可以只依赖外部组件（所有的外部组件都在第 0 层）。因此在第 1 层上的每个组件可以独立于所有其他（局部的）组件进行测试。

当我们继续深入到物理设计层次结构的更高层时，子系统的复杂性经常会呈指数级地增长。这种爆炸性的增长意味着我们不久会到达这样的境地：设计来覆盖一个高层次接口的全部行为的测试程序因太困难而无法编写，或者因花费时间太长而难以运行。

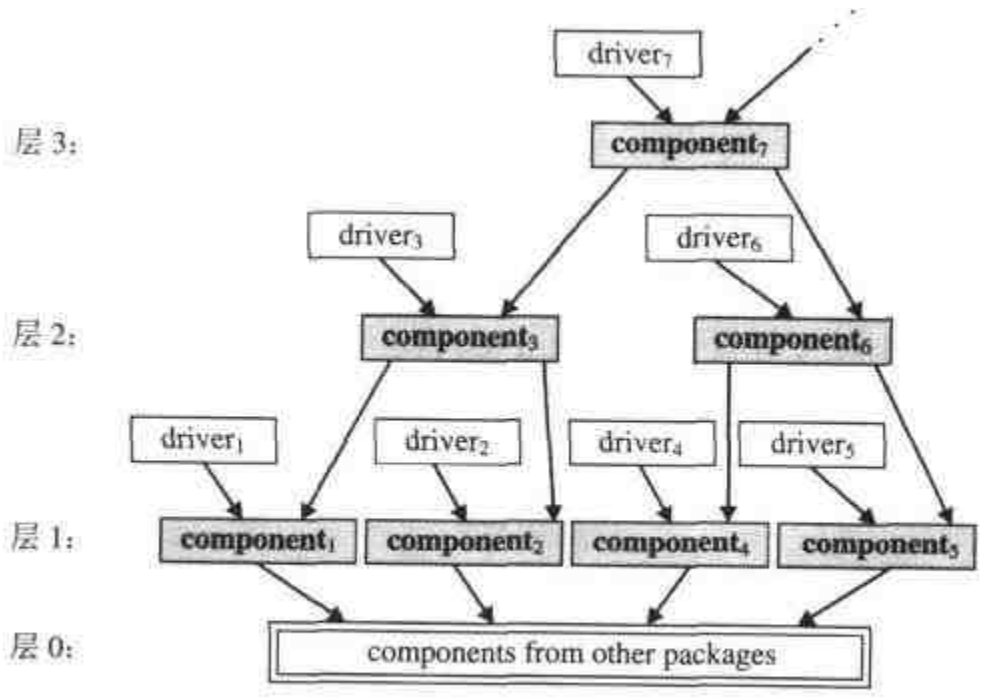


图 4-17　分层次测试策略

定义：增量式测试是指这样的测试惯例：只测试真正在被测试组件中实现的功能。

分层次方法使我们不必重新测试低层次组件的内部行为。如果我们只测试由一个特定组件添加的功能值，那么每个组件的测试复杂性就更可能保持在一个便于管理的水平上。只瞄准一个特定组件添加的新功能的测试方式，在本书中称为增量式测试（incremental testing）。

原　则

只测试在一个组件中直接实现的功能，能够使测试的复杂性与组件的复杂性相当。

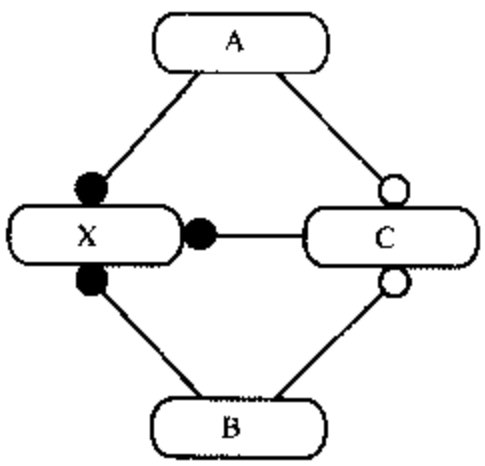
由于我们可以假设低层次的组件是工作正常的供应对象，所以增量式测试的任务经常简化为测试“低层次的对象联合起来形成高层次对象”的方式。编写增量式测试程序在实践中并不总是很容易，它需要编程者知道组件实现的内部知识。

例如，假设一个用户自定义类型 X 分层于 3 个其他类型（A、B 和 C）之上，这三个类型每个都在一个独立的组件中。图 4-18（a）显示了类 X 的部分定义。从这个局部的头文件中我们可以观察到图 4-18（b）的逻辑使用（uses）关系。现在假设每个类存在于一个独立的组件中，我们可以推断出其组件的依赖关系如图 4-18（c）所示。

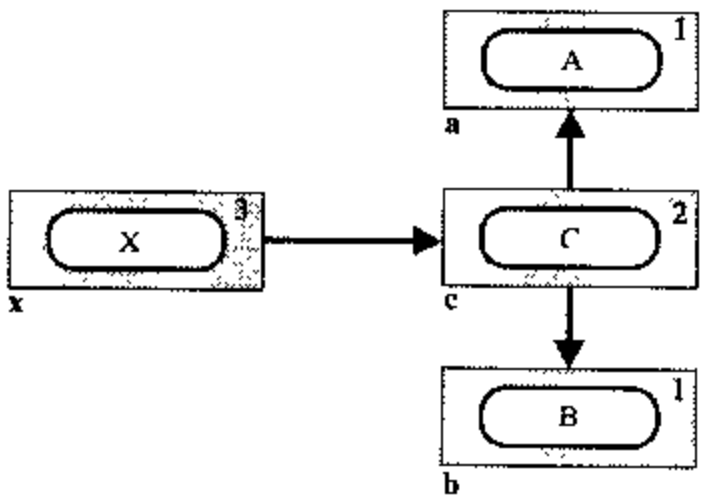
在这个高度简单化的例子中，测试类 X 的函数 f 和 g，实际上就是检验函数 X::f 和 X::g 是否各自与它们的适当的基础函数 C::u 和 C::v 正确地挂钩。由于组件 c 比组件 x 的层次更低，所以我们可以假设 c 已经被测试好了并且其内部也是正确的，在组件 x 的测试驱动程序中没有必要重新测试 C::u 和 C::v。与此相对照的是，X::h 的实现是实质性的，所以该组件的大多数测试工作应该集中于此。

```
class X {
    A d_a;
    B d_b;
    C d_c;
  public:
    //...
    int f() { return d_c.u(d_a); }
    int g() { return d_c.v(d_a, d_b); }
    int h();
};
```

（a）类 X 的定义



（b）类之间的逻辑关系



（c）组件之间的物理关系

图 4-18　为了测试目的分析一个分层对象

定义：白盒测试是指通过查看组件的底层实现来验证一个组件的期望行为。

查看组件实现是一种测试类型，称为白盒测试。通过小心选择试运行一个对象的所有内部功能的测试案例，白盒测试允许测试者用小得多的测试驱动程序测试几乎全部的内部代码。

白盒测试在帮助开发者排除低层次的程序错误方面是有效的，如简单的代码错误、甚至是经常导致内存泄漏或者强迫程序终止的基本算法错误等。因为白盒测试是依赖于实现的，一个基础对象若完全重新实现，将致使这样的测试无效。

白盒测试和百分之百代码覆盖度对于确保高质量的组件来说是必要的，但不是充分的。例如，如果作为一个开发者在分析一个问题时，遗漏了一个要求额外处理的特殊情况，那么不大可能只通过白盒测试来发现这种遗漏的情况。

> **定义：黑盒测试**是指仅基于组件的规范（即不必了解其基础实现）来检验一个组件的期望行为的惯例。

白盒测试检验代码工作情况是否与开发者的意愿相符，与此不同，黑盒测试检验组件是否满足其需求和是否符合规范。

黑盒测试由组件需求和规范直接驱动。黑盒测试对于大部分组件而言是独立于实现的。黑盒测试对于一个独立的测试员来说也是合适的，例如，QA 部门那些只能依靠文档来理解组件行为和正确用法的测试员。

正如图 4-19 所建议的那样，黑盒测试和白盒测试是有某种程度重叠的相互补充的技术。两种技术都很重要，各自侧重于质量的不同方面。白盒测试倾向于保证我们已经正确地解决了一个问题，而黑盒测试则帮助我们确认我们已经解决了正确的问题。

开发时往往趋向于利用白盒测试来确保可靠性。QA 可以使用白盒测试来确保覆盖度，但是也可以使用黑盒测试来检验伴随软件的规范和文档。另外，黑盒测试可以作为一种认**可测试**提供给客户来证实组件功能。而依赖于实现的白盒测试则可能作为内部测试。复杂组件的完全测试要有效地利用这两种策略。

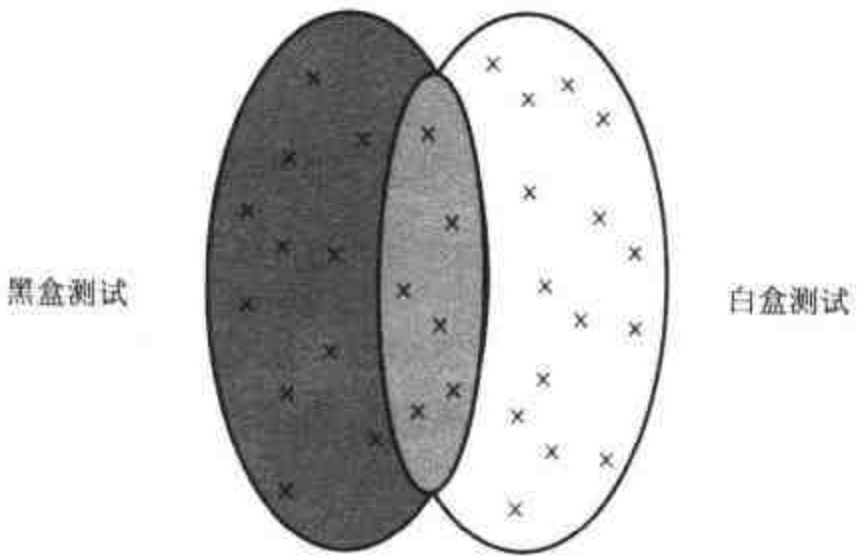


图 4-19　通过测试可发现错误

增量式测试的一个吸引人的特性是，测试任一特定组件的困难与组件本身添加的功能值大致成比例，而不是与组件所依赖的低层次组件的复合复杂性成比例。不管图 4-18 中的组件 a、b 和 c 的功能有多广泛，都有可能为组件 x 编写一个相对较短但却完全彻底的增量式测试程序，因为 X::f 和 X::g 只与一个有效的 C 子对象来回传递信息。

对本节总结如下：我们希望测试的复杂性与被测试的组件的复杂性相对应。我们希望在隔离的情况下测试所有的叶子组件。测试所有高层的组件时都假设它们所依赖的低层次的组件是内部正确的。这种增量式的、分层次的策略使得我们能将测试工作集中于它能做得最好的地方，并且可以避免冗余地重新测试已测试好的软件。

4.9 测试一个复杂子系统

让我们再次返回到图 4-2 中的点对点路径的例子。正如前面所讨论的，p2p_router 的接口很难有效地进行测试。正是这种接口最需要使用分层次测试来保证质量。

这个例子的一个实际实现分布于可层次化的等级结构中，如图 4-20 所示。这个子系统中的一些（但不是全部）组件被更高层的其他组件所重用。geom_point 和 geom_polygon 组件都属于一个独立的软件包 geom，并且被 p2p 软件包的实现者认为是内部正确的。这些可重用的库组件在 router 的实现中占有并非微不足道的部分。

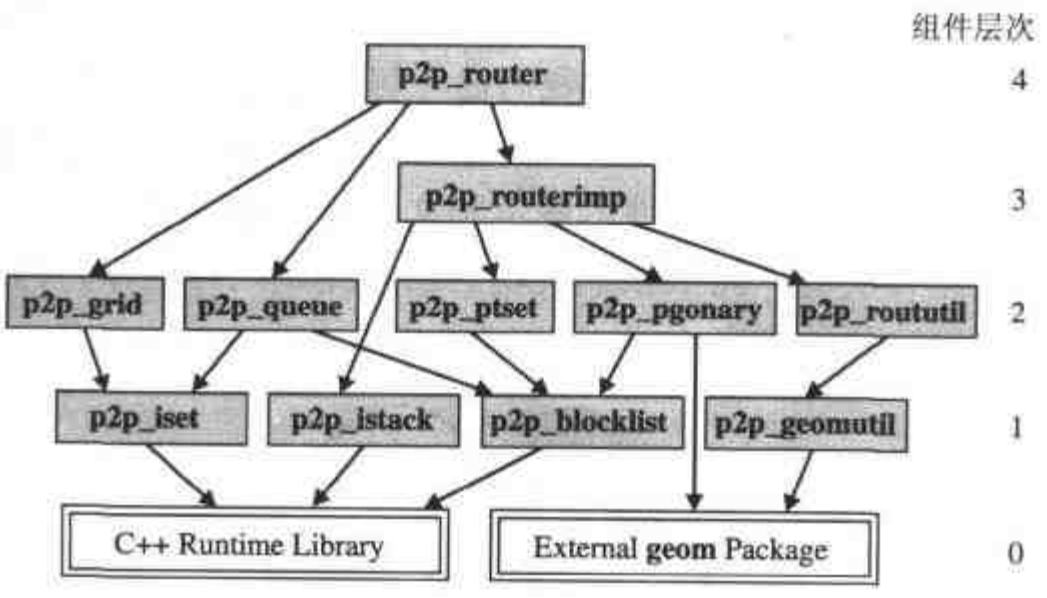


图 4-20 点对点路径的组件依赖图

让我们假设，在 p2p_router 子系统中，每个最低层次的组件都有可预测的行为并且在可测试方面很突出。层次 1 的组件仍然是可隔离测试的——独立于任何其他的 p2p 组件。在这个子系统中，层次 2 中的每个组件最多依赖于层次 1 中的两个组件。层次 2 中的每个组件实现了适当数量的附加功能，这些功能与已经测试好的低层次的功能复合在一起，不难理解和检验。

p2p_router 组件把 router 的客户与其实现的所有细节隔离开来，把许多实现推给了

p2p_routerimp 组件。反过来，p2p_routerimp 将把在 p2p_router.c 中另外定义的不可访问的子功能暴露给测试工程师。

在实际的实现中，p2p_router 只实现了少于百分之十的解决方案。其工作主要是协调在 p2p 子系统较低层次组件中实现的功能。通过小心地将这个复杂组件的功能的实现分布于有 10 个其他的局部子组件的层次结构中，我们已经提高了可维护性。每个子组件都有独立的、定义良好的接口，并且每个子组件都实现了可管理数量的可预测功能。

小结：通过将组件的实现分解成独立可测试（也可能是可重用）组件的一个可层次化的层次结构，我们已经为潜在地难以测试的组件的易测试性进行了设计。把测试工作分布到整个 router 子系统中，将指数级地减少在最高层次上要获得相同质量所需的回归测试的总量。人工检验代价高昂且容易出错，由于缺少时间和资源经常不进行这种检验。但是，层次化的结构使得子组件的可预测行为可以通过不需要人工干预的更健壮的方法来进行测试。

简言之，复杂子系统的分层次物理实现的测试与非层次化结构方案实现的测试相比，既可靠又节省开销。

4.10 易测试性和测试

易测试性（Testability）和测试（Testing）不是一码事。实际上，它们在很大程度上是质量的独立的两个方面。“可测试的”（testable），是指有一种有效的测试策略，这种策略使我们能检验由接口（以及支持文件）描述的功能是否已经实现了。“已测试的”（tested），是指产品已经被证实遵从了它的规范。**可测试性**是我们从设计一开始就努力追求的目标。**已测试**是一种我们的产品在交给客户之前必须达到的状态。**测试**（testing）是我们一直在做的事情。

原　则

完全的回归测试是昂贵的但也是必需的。建立彻底的回归测试的适当时间，与要测试的子系统的稳定性密切相关。

了解何时和花费多少进行测试是一种工程上的折衷。在实现过程中开发者对代码测试得越彻底，就越有可能避免不可预见的错误影响开发进度。

另一方面，开发彻底的测试程序非常费时并且会显著地增加前期的开发费用。这种额外的工作代价，仅通过减少花在维护、将来的功能增强甚至当前开发上的时间来补偿是不够的。

但是，在开发过程的早期阶段，许多组件的接口不可避免地会发生相当大的改变。一些组件会被分拆，另一些组件会被合并，还有一些组件会整个地消失。因此，在一个项目的早期就开发彻底的回归测试程序在有些情况下是不划算的。

随着项目的进展，各种组件将变得成熟。这些组件的接口也将变得更稳定——它们变化的频率会降低，比如少于一个月一次。正是在这个时候，QA 应该编写彻底的、系统的回归测

试程序来检验这些组件，并报告遗漏的或二义性的文档。

只要开发者设计的组件是可测试的，并提供了足够的和合适的文档，测试工程师应当可以十分直截了当地编写详细而系统的测试程序来检验每个组件提供的功能。[1]

如果开发人员在设计系统时不考虑易测试性，那么测试过程就可能不是简单明了的或有效的。为了方便进行有效的测试，必须远在对组件进行测试之前考虑一个系统的易测试性。

4.11 循环物理依赖

设计经常开始于非循环依赖，随着设计的演进，在系统功能的增强过程中，循环依赖会悄悄地混进来。例如，在4.6节的图4-8中的类C1中加入成员函数g()，通过值返回一个C2，如下所示：

```
class C1 {
    // ...
  public:
    C1 f();
    C2 g();     // new
};
```

成员函数引入了一个附加的依赖，导致了如图4-9（b）的循环。由于加入了这个函数，组件c1和c2必须彼此“了解”（即它们各自的组件必须包含彼此的头文件），因而变得相互依赖。此时不再可能在没有另一个的情况下单独测试或使用C1或C2。

原 则

组件之间的循环物理依赖限制了组件被理解、测试和重用。

组件之间存在循环物理依赖是不受欢迎的，不仅因为循环物理依赖使得组件难以测试且不可能独立重用，而且还因为循环物理依赖使得人们理解和维护这些组件更困难。一旦两个组件相互依赖，就需要同时理解两个组件，以便充分地理解其中的任何一个。

指导方针

避免组件之间的循环物理依赖。

紧密相关的类之间相互依赖很平常，但是，这些类将完全驻留在一个单个的组件中。如果我们发现两个（或更多）组件c1和c2相互依赖，我们有三种选择：

（1）对c1和c2重新打包以便它们不再相互依赖。

① 系统测试的全面论述见**marick**。

（2）将 c1 和 c2 从物理上组合成一个组件 c12。

（3）将 c1 和 c2 当作一个单一的组件 c12 来考虑。

最好的解决方案是在循环依赖发生之前纠正它；或者，如果它们确实已经混入了，那么就应该立即进行检测和纠正。第 5 章研究了一种技术，该技术可在保持预期特性的同时，对一个循环依赖的设计进行重新构造，以消除循环。

当复合抽象中的对象自然地紧密耦合在一起并且不需要考虑别的问题时，把组件合并为单一的组件是正确的解决方案。如果一个类是另一个类的友元，则进一步暗示这些类应归入同一个组件（见 3.6.1 节）。合并紧密耦合的内聚组件也有如下受欢迎的好处：减少了组件数量，从而降低了系统的物理复杂性，又不会进一步危及易测试性或独立重用。

有时单个的、紧密耦合的抽象会被认为太大了，不适合在一个组件中实现，它们将被分拆成相互依赖的组件。但是，大多数情况，抽象的紧密耦合的部分可能与实现的其他部分隔离开来，并被放在一个单个的组件中，这个组件反过来依赖其他独立的组件。这些独立的组件现在可以在隔离的情况下进行彻底的测试（见 5.9 节）。

如果没有别的解决方案，那么我们可以主观上把相互依赖的组件看成是一个大的组件，如图 4-21 所示。这种方法就目前来说是容易的，但是，从长远观点来看则是最不受欢迎的方案。这些物理上独立但是紧密耦合的组件必须被人为地看成是一个单一的物理单元，这种单元会降低可维护设计的一致性。尽管这样的依赖不是我们所希望的，但是只要这种“污渍”的数量和大小保持最小，系统的整体易测试性就不会丧失。

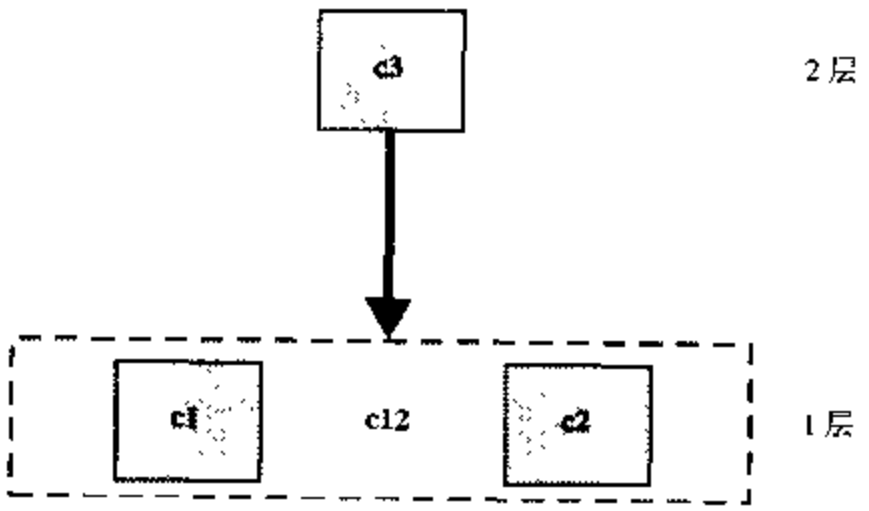


图 4-21　将两个相互独立的组件当作一个来看待

4.12　累积组件依赖

现在我们通过提供一种度量标准来形式化有关设计质量方面的讨论。这种度量标准在本书中称为一个子系统的**累积组件依赖**（cumulative component dependency，CCD）。CCD 与增量式回归测试的连接时开销紧密联系。更确切地说，CCD 提供了一种数字表示的值，表示与

开发和维护一个特定子系统有关的相对开销。

> **定 义**：**累积组件依赖**（CCD）就是在一个子系统内的所有组件 C_i 之上，对每个组件 C_i 在增量式地测试时所需要的组件数量进行求和。（即**累积组件依赖**就是为了增量式地测试子系统中所有组件所需的组件数量的总和）

连接大型程序要花费很长的时间。在创建组件及其测试驱动程序的过程中，开发人员一般需要多次连接一个组件。此后，无论何时运行回归测试程序，组件都必须与其驱动程序相连接。对于小型的项目，连接时间与单个组件的编译时间相当。当项目变大时，连接时间显著增长，甚至比最大的组件所需要的编译时间还要多得多。

我们的大多数开发时间消耗在低层次的组件上，主要是因为低层次的组件比高层次的组件多很多。系统的这些低层次的部件可能是错综复杂的，有时可以选择它们来调节性能。使开发、测试和维护低层次组件的过程简化并更有效率，这对我们是有好处的。

为了方便讨论，我们假定一个设计中的依赖形成了一个完美的二叉树。正好过半的组件在层次 1 上，并且能够在完全隔离的情况下进行测试。另外的四分之一组件，每个依赖于两个叶子组件。如果我们用 L 代表树中层次的数量，那么 2^L-1 个组件中只有一个组件会实际上依赖所有其他的组件。尽管实际的设计没有这样规整，但是测试非循环依赖的组件层次结构的优点仍然很清晰。

考虑与开发一组组件相关的开销。让我们暂时假设连接时间与被连接的组件的数量是成比例的[1]。例如，如果将一个组件连接到一个测试驱动程序要花费 1CPU 秒，那么连接 5 个组件大约会花费 5CPU 秒。

存在循环依赖时，可能有必要连接大多数或全部的组件，以便能测试其中的任意一个。一个完全相互依赖的设计，并不一定每个组件都直接依赖于每一个其他的组件[2]。假设我们的系统是非常紧密耦合的，并且每个组件都直接或间接地依赖所有其他的组件。如果我们用 N 代表系统中的组件数，将这些组件中的任何一个连接到其测试驱动程序的开销与 N 成正比。那么为这些组件建立全部 N 个测试驱动程序的单独的连接开销会与 N^2 成正比。这个事实解释了为什么大型系统中连接的开销常常是运行彻底的回归测试的开销的主要部分。

原 则

N 代表系统中组件的数量。

$\text{CCD}_{\text{循环依赖图}}(N)$ =（组件的总数）·（测试一个组件的连接时开销）$= N \cdot N = N^2$

① 这个假设当然只是一个粗略的估计，因为连接开销会明显受到组件规模变化以及函数调用层次结构的构造的影响。

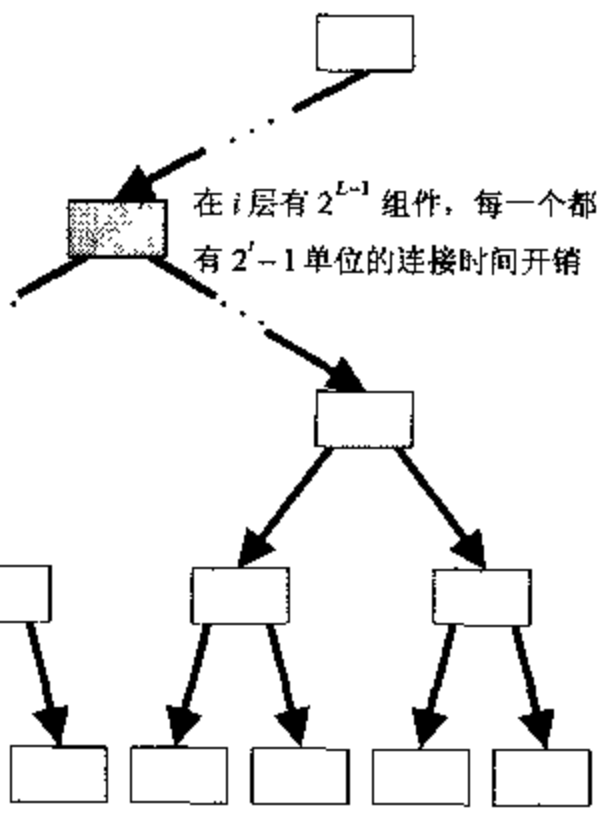
② 一个完全相互依赖的设计有一个直接依赖图，这个图是“强连接的（strongly connected）”，但不一定是“完全连接的（complete）”。这些不同术语的正式定义见 **aho**（5.5 节，189 页以及 10.3 节，375 页）。

现在考虑，如果依赖是非循环的并且形成了一个二叉树时会出现什么情况。现在，不是所有的组件的连接开销都相同。在层次 1 的组件在一个单位时间（例如，1CPU 秒）内可以被连接到各自的测试驱动程序。与组件测试相关的连接开销有至少一半实际上可以被消除。层次 2 上的每个组件依赖层次 1 上的两个组件，并且组成一个大小为 3 的子系统（它将消耗 3CPU 秒来进行连接）。也就是说，与连接相关的测试开销的 1/4 可以戏剧性地减少（乘以一个因子 $N/3$）。在这个假设的系统中只有一个组件，根（root），会需要 N CPU 秒的连接时间，而以前，N 个组件中的每一个都需要这么多时间。

在数学上我们可以表示出增量式地测试一个系统（其物理依赖形成一棵二叉树）的总的连接开销与 $N\log(N)$成正比，而不是与 N^2 成正比（见图 4-22）。例如，在有 15 个组件的情况下：

$$\text{CCD}_{\text{平衡二叉树}}(15) = (15+1) \cdot (\log_2(15+1) - 1) + 1 = 49$$

该层组件数量	连接需要的时间单位	层次
1	2^L-1	L
.	.	.
.	.	.
.	.	.
2^{L-i}	2^i-1	i
.	.	.
.	.	.
$\frac{2^L}{8}$	7	3
$\frac{2^L}{4}$	3	2
$\frac{2^L}{2}$	1	1



设 L 是系统中层次的数量（二叉树的深度）。

设 $N = 2^L - 1$ 是系统中组件的数量。

$$\text{CCD}_{\substack{\text{平衡}\\\text{二叉树}}}(N) = \sum_{i=1}^{L} \left(\begin{array}{c}\text{在 } i \text{ 层上的}\\\text{组件数量}\end{array} \right) \cdot \left(\begin{array}{c}\text{测试一个 } i \text{ 层上}\\\text{组件所需的连接}\\\text{时间开销}\end{array} \right)$$

$$= \sum_{i=1}^{L} 2^{L-i} \cdot (2^i - 1)$$

$$= \sum_{i=1}^{L} 2^L - \sum_{i=1}^{L} 2^{L-i}$$

$$= 2^L \cdot \sum_{i=1}^{L} 1 - \sum_{i=1}^{L} 2^{i-1}$$

$$= 2^L \cdot L - (2^L - 1)$$

$$= 2^L \cdot (L-1) + 1 \quad \{ \text{可用于与高度为 } L \text{（整数）的依赖关系二叉树进行比较}$$

$$= (N+1) \cdot (\log_2(N+1) - 1) + 1$$

$$= (N+1) \cdot \log_2(N+1) - N \quad \{ \text{可用于与大小为任意正数 } N \text{ 的理论上的二叉树进行比较}$$

$$= O(N \cdot \log(N)) \quad \{ \text{近似的连接时开销}$$

图 4-22 为一个依赖关系的二叉树计算连接时开销

原 则

非循环物理依赖可以明显减少与开发、维护和测试大型系统相关的连接时开销。

非循环依赖的好处很多。如果一个非循环设计的依赖关系是树型的层次关系，那么其单个测试驱动程序的平均连接时间是与组件数的 log 成正比的，而不是像循环设计的情况那样，与组件数本身成正比。

原 则

设 N 为系统中组件的数量。

CCD 循环依赖图 $(N) = (N+1) \cdot (\log_2 (N+1) - 1) + 1$

图 4-23 比较了当 N=1、3、7 和 15 个组件时，测试出的与循环的系统和层次结构的系统相关的连接时开销。显示在对应于依赖图中的每个组件的位置上的数字，表示与增量式测试该组件相关的连接开销。每个系统的 CCD 被计算并显示在依赖图的底部。每个树型系统的 CCD 按两种方式进行计算：一种是一层一层地计算，另一种是使用图 4-22 导出的方程进行计算。

假设我们正在开发一个有 63 个组件的系统，每个组件都有自己的测试驱动程序。在循环设计中，每个组件将花费 63 秒时间来重新连接以便于测试。将此系统与一个层次结构的设计相比（如图 4-24 所分析的那样），超过一半的组件可以在 1CPU 秒中完成连接，1/4 的组件在 3CPU 秒内完成连接，1/8 的组件在 7CPU 秒内完成连接，以此类推。63 个组件中只有一个组件花费了全部 63CPU 秒来进行连接。连接全部 63 个测试驱动程序的总的开销，在图 4-24 中用两种方法计算的结果都是 321CPU 秒（5.35CPU 分）。将这个开销与连接所有 63 个测试驱动程序到一个循环依赖的系统中所要花费的开销（63^2=3969CPU 秒(1.1CPU 小时)）比一比。

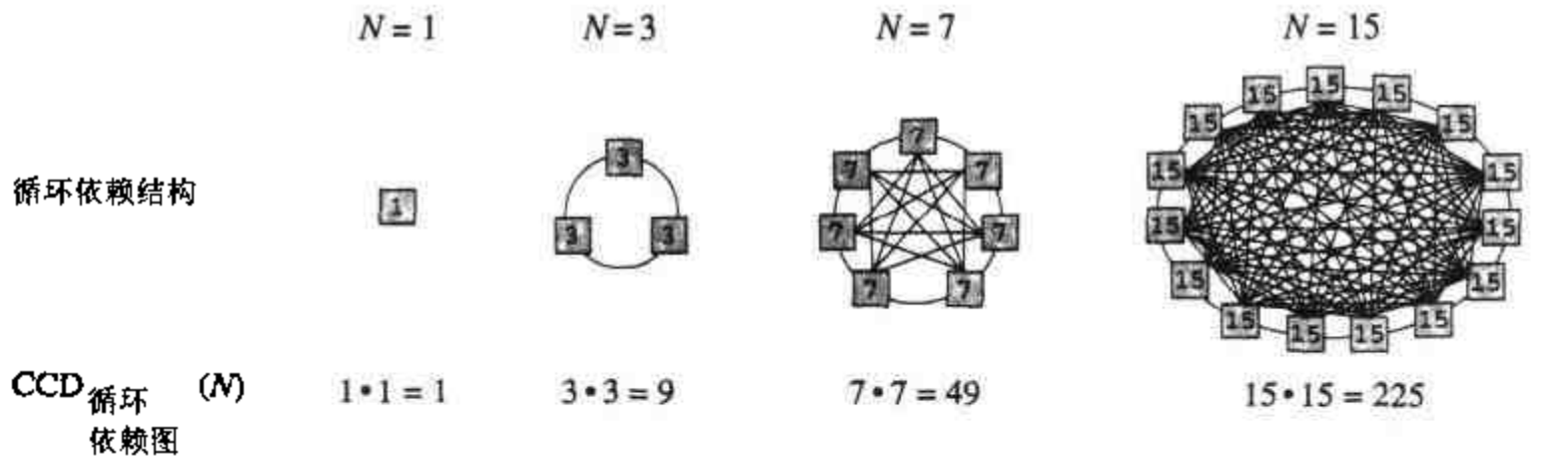


图 4-23 与增量式测试相关的相对连接开销

层次号	该层上的组件数量		连接该层上的一个组件所需的开销		该层上所有组件的连接开销
1	32	•	1	=	32
2	16	•	3	=	48
3	8	•	7	=	56
4	4	•	15	=	60
5	2	•	31	=	62
6	1	•	63	=	63
				总数 =	321

$$CCD_{\text{平衡二叉树}}(63) = (63+1) \bullet (\log_2(63+1) - 1) + 1$$
$$= 64 \bullet 5 + 1$$
$$= 321$$

图 4-24 N=63 的平衡二叉树层次结构的连接开销

如果组件的数目更大，例如为 1023 个组件，那么在一个树型设计中连接一个组件的平均开销将比循环设计的开销少 2 个数量级(连接每个组件的平均开销分别为 9CPU 秒和 1023CPU 秒)。我们可以使用在图 4-22 中导出的方程来预测这个系统的 CCD。在一个有 1023 个组件的

系统上建立组件回归测试的总的连接时间的范围，可以从层次化设计系统的 1024×9+1=9217CPU 秒（刚超过 2.5 小时）到循环依赖系统的 1023×1023=1046529CPU 秒（超过 12 天）。

一个单一的系统不太可能不被进一步分成多个软件包而由 1023 个组件直接构成。确保软件包之间的非循环依赖比确保单个组件中的非循环依赖更重要。（见 7.3 节）

CCD 也是增量式回归测试对累计磁盘空间需求的预测指标。当我们并行地增量式测试一个大型系统时，磁盘空间会变成一个需要考虑的重要因素。在磁盘上每个独立的可执行测试程序占用的磁盘大小，与测试驱动程序必须静态连接的组件的数量大致成正比。因此，循环依赖系统比层次设计需要多得多的磁盘空间。

小结：我们的目标是能够为每个组件建立一个测试驱动程序，该驱动程序与要测试的组件及其所依赖的（少量）组件相连接。CCD 是一种测量方法，它根据与增量式地测试每个组件相关的总的连接时间来量化一个系统的耦合程度。在增量式地测试组件所需的连接时间和所需要的磁盘空间方面，循环依赖的组件展示了二次方的特性。相反，形成一个非循环的（树形）层次结构的组件依赖关系则可以极大地减少增量式组件测试的连接开销。

4.13 物理设计的质量

在本节中，我们将基于物理依赖关系来刻画什么因素使得一个设计是可维护的。我们将继续讨论 CCD 并且用它来表示一个子系统的整体可维护性。我们还将讨论如何使用 CCD 来衡量物理设计质量上的增量式的改进。

设想我们加入了一家开发一个超大型系统的公司。分配给了我们一个有 150000 行 C++ 代码的子系统，并要求我们理解它是做什么的，还要就如何改进它提出建议。通过仔细检查，我们发现（大部分的）组件与第 2 章和第 3 章中提出的规则和指导方针一致。接着我们又发现系统中的大多数组件（直接或间接地）依赖其他的大多数组件。我们该怎么办呢？很不幸，这个故事没有令人愉快的结局。任何人能做的最好的事情是努力将整个设计装入自己的头脑，这也可能需要几个月。

如果同样的子系统着眼于用最小 CCD 的方法来设计，则大多数（如果不是全部的话）的循环依赖会被消除。因此有可能在隔离的情况下研究子系统的部件，以便测试、检验、调整甚至替换它们，而无论在主观上还是在物理上都不必涉及整个子系统。换句话说，有效地减少组件间的相互依赖（通过 CCD 来量化）可以提高可理解性，因而也提高了可维护性。

可理解性是几个难以量化的性质之一，但是它对于最小化组件之间的依赖非常有好处。有选择地重用是另一个这样的性质。考虑图 4-25 所示的子系统的体系结构。这个系统由 7 个组件组成，每个组件直接或间接地依赖系统中的其他每个组件。每个组件可以直接测试，但是没有一个组件可以在隔离的情况下测试或者独立于其他组件而重用。因为每个独立的测试驱

动程序被迫与整个系统连接，仅存储这些独立的驱动程序所需的磁盘空间的总量也是二次的。

现在假设图 4-25 中的设计的循环依赖被消除了，它是可层次化的了。尽管层次性（levelizability）是我们非常希望的，但是一些可层次化的体系结构比其他结构更易于维护和重用。考虑图 4-26 所示的结构设计，每个结构包含 7 个组件，并且每个都是可层次化的。图 4-26（a）显示了层次化的一个极端版本。这种设计被称为**垂直的**（**vertical**）。在这个系统中，每个组件依赖于在更低层次上的所有其他组件。垂直的子系统显示出高度的耦合程度，这抑制了独立重用。重用一个规模为 *N* 的垂直系统中的一个随机选择的组件，将导致平均要连接(*N*–1) / 2 个额外的组件。相应地，用于存放增量式测试驱动程序的平均磁盘空间也非常大。

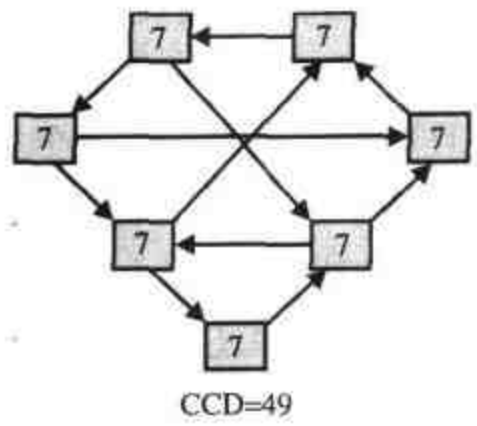


图 4-25　规模为 7 的循环依赖子系统体系结构

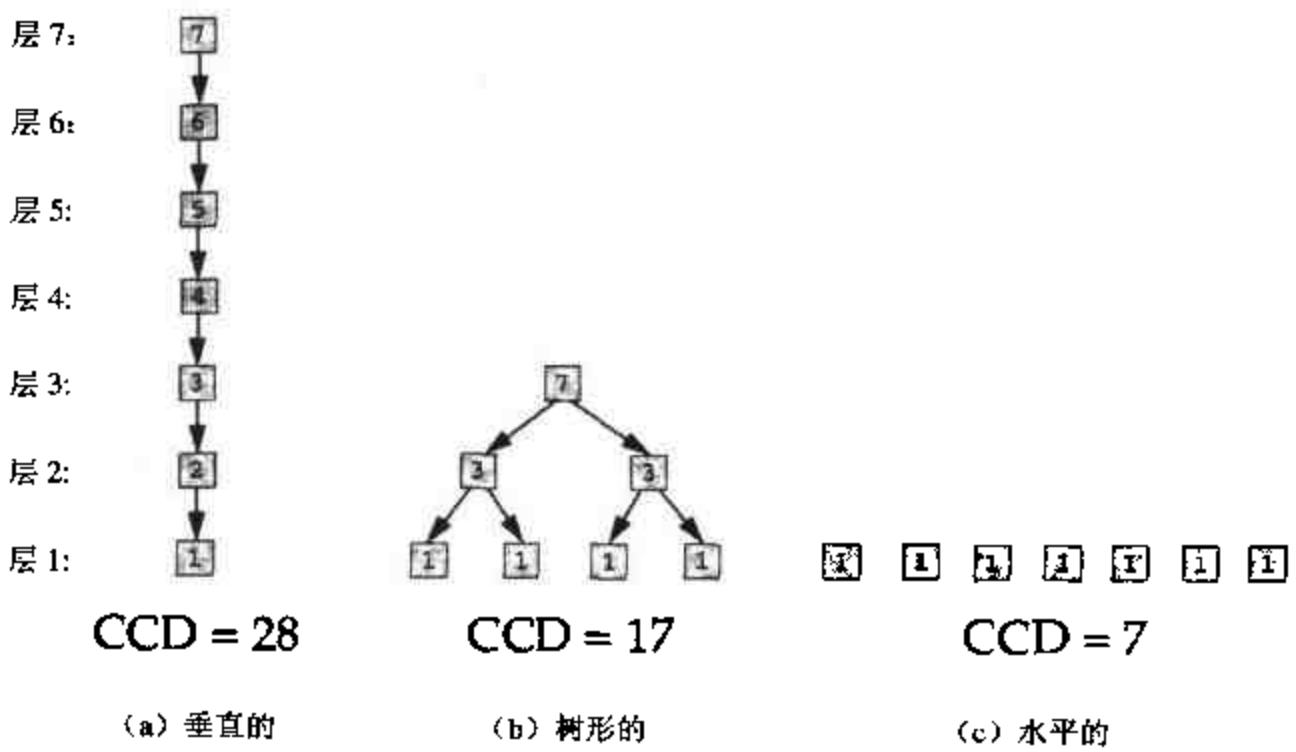


图 4-26　规模为 7 的各种组件体系结构的 CCD

垂直系统在测试和重用方面极不灵活。测试纯垂直系统的次序只有一种，这个次序完全由它的层次来决定。从连接时间来看，开发一个垂直的子系统也是相对昂贵的。这个系统的 28 个单位的总连接开销（CCD）比在图 4-25 中显示的循环依赖子系统的 49 个单位的总连接开销的一半还要多。另外，将一个垂直子系统划分为并行的开发工作并分配给多个开发者，也会相对比较困难。但是，一个垂直的子系统是非循环的，所以，从性质上看，它比循环依赖的系统更容易维护。

图 4-26（b）显示了一个二叉树形式的层次结构设计。正如我们所知道的，这个设计中有

超过一半的组件每个只在 CCD 中占一个单位。设计不会是完美的二叉树，但是一个二叉树的 CCD 为比较许多典型的应用提供了一个好的基准。树型设计的耦合程度更低，比垂直设计灵活得多并且更适合重用。在每个层次上，一般都有几个可以独立于系统的其他部分进行测试并可能重用的子系统。存储大多数增量式测试驱动程序所需要的磁盘空间也相对较小。

通过使依赖图更平而不是更高，我们增加了灵活性。设计越扁平，独立重用的潜力就越大。使依赖关系扁平还有助于减少理解和维护的时间。设计越扁平，就越有可能对一个单一的隔离组件或一个小的子系统进行错误追踪，因而就需要越少的磁盘空间来存储检查错误的测试驱动程序。

图 4-26(c)显示了层次化领域的另一极端。这种类型的设计被描述为**水平的(horizontal)**，因为所有的组件完全是独立的，并且彼此之间没有耦合。对属于纯水平子系统的组件可以以任何次序进行测试，并且可以以任何所希望的组合进行重用。每个增量式测试驱动程序所需的磁盘空间十分少。这种依赖特性在可重用组件库中是具有代表性的，但对于子系统来说则一般不具有代表性。

基于设计的 CCD，我们可以对一个给定大小的设计的可维护性和可重用性（但不一定是“长处”）做某种客观的和定量的描述。设计依赖形成了从循环到垂直、到树型、到水平这样一个连续区间。即使存在循环，每个设计也可以被赋予一个 CCD。在其他条件都相同的情况下，CCD 越低，开发和维护系统的代价就越低（就连接时间和磁盘空间而言）。

还有另外一个原因使我们努力追求一个具有最小 CCD 的分层次的系统。需求很少是一成不变的，在项目的开发过程中可能会有变化。通过将实现分布在一个由组件组成的层次结构中，可以使设计更能适应需求的变化。一个体系结构越扁平，软件规范的变化对整个系统产生影响的可能性就越小。这种规范的变化引起的预期开销，直接与系统中的平均组件依赖（Average Component Dependency，ACD）相关。

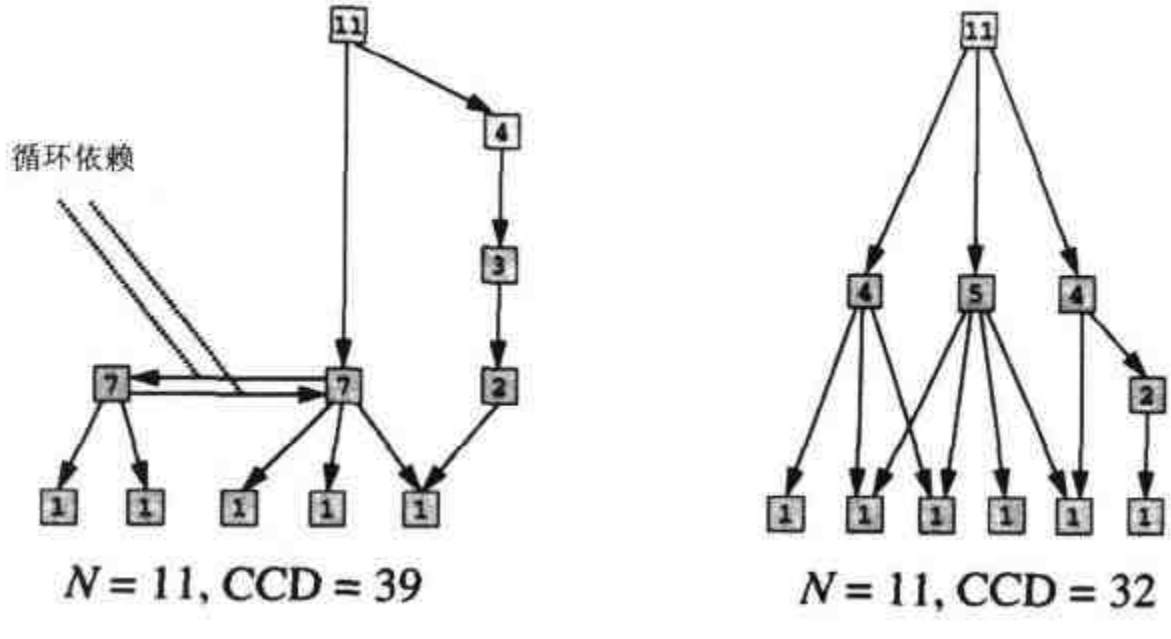
> **定义：平均组件依赖**是指一个子系统的 CCD 与系统中的组件数量 N 的比值：
>
> $$\text{ACD}(\text{子系统}) = \frac{\text{CCD}(\text{子系统})}{N_{\text{子系统}}}$$

例如，在一个完全水平化的系统中，一个单一组件的规范的改变只会引起一个组件变化。对于一个有 N 个组件的树形体系结构，平均需要改变大约 $\log(N)$个组件。而对于垂直结构，改变一个组件的接口，我们可能预期要重新访问$(N+1)/2$ 个组件。最后，对于一个完全循环依赖的设计，单个组件接口的变化可能会影响所有 N 个组件。

原 则

CCD 的主要用途是，对一个给定体系结构的较小改动引起的整个耦合结构的变化进行量化。

作为减少 CCD 的一个例子，考虑图 4-27 所示的具有相似依赖结构的两个系统。A 设计有两个组件循环依赖。测试这些组件中的任何一个，都需要连接这两个组件以及其中的任何一个组件所依赖的所有组件。这使得它们每一个都有一个单独的组件依赖（7）。还要注意 A 设计的右部，层次结构的一部分是纯垂直的。



（a）初始设计 A　　（b）修改后的设计 B

图 4-27　一个子系统的两种设计方案的依赖图

在第 5 章，我们将详细地介绍几种减少连接时依赖的技术。为了改进这种设计，我们首先应打破循环依赖，然后检查垂直部分，看能否使它不那么顺序化。在这种情况下，只有通过逐步将代码提升到更高的层次以及/或者提取出一个共享的资源，才有可能打破循环。至于垂直的那段，有可能可以将垂直链中的一个或多个组件移走并使它们变成叶子组件，从而独立于其他组件。做了这些修改后的结果是设计 B，如图 4-27（b）所示。我们的原始设计的 CCD 值 39 远远低于完全相互依赖的设计的 CCD 值 121。但是我们仍然能够把这个差不多分层的系统的 CCD 值从 39 降到 32——改进了大约百分之十八。

CCD 值是一种在系统中描述物理耦合的客观度量方法。CCD 可以用非常高的增量式开发和维护开销来标记子系统。例如，一个垂直的链是有最高 CCD（$N(N+1)/2$）的可层次化配置。因此，CCD 大于 $N(N+1)/2$ 就说明至少有一个循环依赖存在。但是，CCD 本身不是一个子系统的质量测量指标。

我们可以方便地使用图 4-22 导出的替换方程来确定规模与图 4-27 中所显示的设计一样的、（理论上）类似二叉树的体系结构的 CCD 值。图 4-28 证实了一个有 11 个组件的类似二叉树的体系结构有 32.02 的 CCD 值，这与设计 B 的 CCD 值差不多。

$$CCD_{\substack{平衡\\ 叉树}}(N) = (N+1) \bullet \log_2(N+1) - N$$

$$CCD_{\substack{平衡\\ 一叉树}}(11) = (11+1) \bullet \log_2(11+1) - 11$$

$$= 12 \bullet \log_2(12) - 11$$

$$= 32.02$$

图 4-28 计算规模为 11 的理论平衡树的 CCD

定义：标准累积组件依赖（NCCD）是指包含 N 个组件的子系统的 CCD 值与相同大小的树型系统的 CCD 值的比值。

$$NCCD(子系统) = \frac{CCD(子系统)}{CCD_{平衡二叉树}(N_{子系统})}$$

一个系统的 NCCD 可以用来描述该系统相对于同样规模的理论二叉树系统的物理耦合程度。返回去查阅图 4-27，设计 B 的 NCCD 是 32/32.02=1.00，设计 A 的 NCCD 为 39/32.02 = 1.21（而完全相互依赖的实现的 NCCD 是 121/32.02 = 3.78）。

如果 NCCD 的值小于 1.0，则可以认为是较“水平化的”或松散耦合的；这样的系统可能很少使用重用。如果 NCCD 的值大于 1.0，则可以认为是较“垂直的”和/或紧密耦合的；这样的系统可能正在大量地重用组件。如果 NCCD 的值远远大于 1.0，则表明在系统中可能有明显的循环物理耦合。

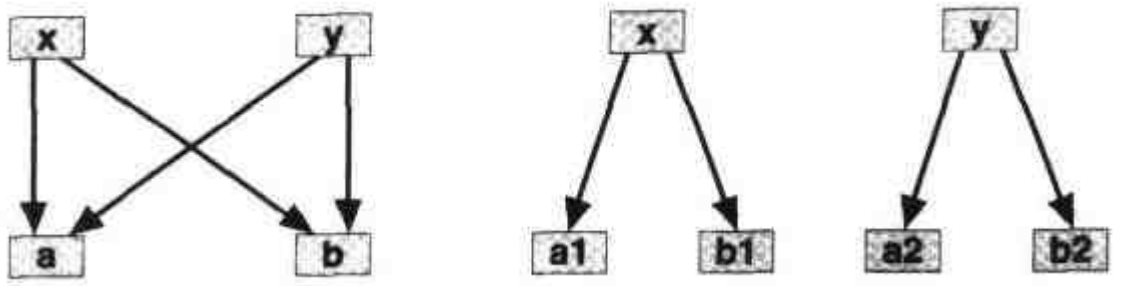
就我们可以获得的 CCD 值来说，可维护性的程度依赖于子系统的特性。我们不可能总是获得理想的树型的可维护性。对于水平的组件库，我们会希望有一个低得多的 CCD。对于大量使用重用的高度相互连接的拓扑结构，如 2.5 节中图 2-8 的窗口系统，其 CCD 将会更高。

NCCD 不是一个系统的相对质量的衡量指标。NCCD 只是一种描述一个子系统内的耦合程度的工具。增加在一个系统中的组件的数量可以人为地减少 NCCD。这样做的一种方法是：消除完全有效的重用；但这不大可能是一种改进。

图 4-29 显示了两个有等价功能的设计。设计 B 比设计 A 大 50%，CCD 也大 25%。另一方面，设计 A 通过重用显示出比设计 B 更多的物理耦合。虽然如此，设计 A 仍然非常可能是更精心策划的和更好维护的设计。

原 则

使一给定组件集合的 CCD 最小化是一个设计目标。



（a）设计 A　　（b）设计 B

图 4-29　举例说明冗余的影响

我们几乎总是希望减小给定规模系统的 CCD 值。减小一个系统的规模（组件数）也是我们所希望的，但是不能以引入循环依赖、不适当的合并组件或建立不便管理的大型编译单元为代价。当增加一个系统的组件数量实际上减小了 CCD 的值时，很可能因此提高了设计的整体质量。

总之，CCD 度量方法已被引进来显式地标识（我们想最小化的）依赖的种类。NCCD 提供给了我们一种定量方法，这种方法能够把子系统的物理依赖区分为水平的、树形的、垂直的或循环的。一个系统的准确的 CCD（或 NCCD）数值并不重要。重要的是要积极主动地设计系统，使每个子系统的 CCD 值始终不会比必要的更大。

4.14　小结

高质量的复杂子系统由许多分层组件形成的非循环的物理层次结构组成。在系统层次上进行完全测试不仅昂贵而且根本不可行——如果不是不可能——对“好”接口尤其如此。

一个“好”接口将实现的复杂性封装在一个简单的易于使用的外表后面。同时，这使得我们通过这个接口来测试其实现异常困难。

本章中的许多测试策略是由十多年前易测试性设计（DFT）的成功激发出来的。但是，和现实世界中的对象不同，定义在一个软件系统中的类的实例与定义在该系统之外的同一个类的实例并没有什么两样。我们可以利用这个事实来隔离地验证设计层次结构的各个部分，从而减少集成的部分风险。

隔离测试是确保复杂低层次组件的可靠性的一种很合算的方法。通过尽可能将测试推到设计层次结构的最底层，我们可以确保如果组件或子系统被增强、移植或在另一个系统中重用，它将独立于客户程序而保持其指定的特性。

层次号在一个子系统内基于组件对其他组件的物理依赖来描述组件。另外，层次号给有非循环依赖组件图的系统提供了能够进行有效测试的一种顺序。若某子系统的组件依赖形成

了一个直接的非循环依赖图（DAG），则称该子系统是**可层次化的**（**levelizable**）。一个层次化的组件依赖图使得一个系统的物理结构更容易理解，因而也更容易维护。

分层次测试是指测试物理层次结构中的每一层的组件。每个较低层次的组件都应该提供独立于更高层次组件的良好定义的接口，并实现可被测试、验证和重用的功能。

增量式测试是指使单个测试驱动程序只测试被测试组件中实际实现的功能。在物理层次结构的更低层次中实现的功能，此时假设为是内部正确的。因此，增量式测试反映的是被测试组件的实现的复杂性，而不是这个组件所依赖的组件的层次结构的复杂性。增量式测试是白盒测试的一种形式，它建立在了解组件实现的基础上，以提高可靠性。黑盒测试源于需求和组件规范，并独立于实现。这两种测试形式是相互补充的，它们都有助于保证整体质量。

易测试性是一个设计目标。循环物理依赖抑制测试、理解和重用。累积组件依赖（CCD）提供了一种与增量式测试一个给定子系统相关的整体连接时开销的粗略数值度量。更一般地，CCD 是一个给定设计的相对可维护性的一个指示器。

循环依赖设计不是可层次化的。众所周知，这样的系统难以维护，并相应地有高 CCD 值。在设计中，层次结构越扁平，CCD 值就越低。使物理依赖（结构）更扁平有助于减少理解、开发和维护所需的时间，同时又提高了一个系统的灵活性、易测试性和可重用性。NCCD（标准 CCD）可以帮助我们将任意的设计的物理结构分类为循环的、垂直的、树形的和水平的。

5 层次化

在确立一个系统的全面物理质量时，系统内的连接时依赖（以 CCD 量化）扮演主要角色。质量的更传统方面，例如可理解性、可维护性、易测试性和可重用性，都紧密地依赖于物理设计的质量。如果不小心加以防范，循环物理依赖将使系统失去这种质量，使得它缺乏灵活性和难以管理。

甚至对于可修正的设计，维护和改进起来也可能要付出不必要的昂贵代价。对大型的、低层次子系统的被迫依赖，会给更高层次的子系统造成显著的开发负担。最小化这种依赖的影响有助于提高系统的物理质量。

在本章中，我们将研究消除循环依赖或其他过度连接时依赖的若干技术。升级（escalation）和降级（demotion）是相关的技术，它们把设计中的循环依赖部分移到物理层次的不同级别上。不透明指针（opaque pointer）和哑数据（dumb data）可用来消除概念上的依赖关系的物理隐含。冗余（redundancy）和回调（callback）也是我们要讨论的其他两种技术，它们可用于防止不必要的物理依赖。最后，将介绍一个管理类以及两种通用技术（分解封装和升级封装），以帮助编程人员创建可测试、可重用组件的、有效的、封装的层次结构。

贯穿本章，我们使用了许多取自若干应用领域的例子来阐释这些技术（在各种不同的上下文中）。偶尔我们会为了参考目的提供一段真实的源代码来使例子具体化。

5.1 导致循环物理依赖的一些原因

本节我们将讨论循环物理依赖在实践中可能出现的二种方式。为了说明这个问题的宽度，我们在单独的小节中介绍并讨论每一个例子（没有尝试去解决它们）。这些特定的问题以及许多其他的问题，在本章的其余部分介绍适当的技术时将得到解决。

5.1.1 增强

最初的设计通常都经过精心策划，常常是可层次化的。此时，未预见到的客户需求可能会使我们在增强系统时考虑不周，从而导致不必要的循环依赖。例如，我们有时候会发现相似的对象会因为这样那样的原因（例如：性能原因）而共存于一个系统中，但它们本质上包含的是同样的信息。

图 5-1 显示了一个简单但可说明问题的例子，它由两个类组成，每个类表示一种盒子。一个 Rectangle 由两个点定义，分别表示它的左下角和右上角。一个 Window 由一个中心点、一个宽度和一个高度所定义。这些对象有着截然不同的性能特征，但包含着相同的逻辑信息。

```
// rectangle.h
#ifndef INCLUDED_RECTANGLE
#define INCLUDED_RECTANGLE

class Rectangle {
    // ...
  public:
    Rectangle(int x1,
              int y1,
              int x2,
              int y2);

    // ...
    int lowerLeftX() const;
    // ...
};

#endif
```

```
// window.h
#ifndef INCLUDED_WINDOW
#define INCLUDED_WINDOW

class Window {
    // ...
  public:
    Window(int xCenter,
           int yCenter,
           int width,
           int height);

    // ...
    int width() const;
    // ...
};

#endif
```

图 5-1 一个盒子的两种表示法

这些对象中的每一个都将被用于帮助实施一个图形终端上的超大型交互设计；绘制速度是关键。因为性能原因，我们甚至不考虑使用虚函数，并且将大部分的函数都声明为内联。

原 则

允许两个组件通过#include 指令彼此“知道”隐含了循环物理依赖。

结果，客户程序将偶尔需要能够在这两类盒子之间转换，或许是为了获得另一类的性能特征。这是好的设计可能有时候开始变糟的一种方式。

考虑图 5-2 中前面的“解决方案”集合。我们已经给每一个类加了一个构造函数，该函数使用对另一个类的 const 引用作为它的惟一参数。我们现在可以传递一个 Window 对象给一个需要 Rectangle 的函数，反之亦然，转换将隐含地执行。你觉得听起来怎么样？

```
// rectangle.h
#ifndef INCLUDED_RECTANGLE
#define INCLUDED_RECTANGLE

#ifndef INCLUDED_WINDOW
#include "window.h"
#endif

class Rectangle {
    // ...
  public:
    // ...
    Rectangle(const Window& w);
    // ...
};

inline
Rectangle::Rectangle(const Window& w)
{
    // ...
}

// ...


#endif
```

```
// window.h
#ifndef INCLUDED_WINDOW
#define INCLUDED_WINDOW

#ifndef INCLUDED_RECTANGLE
#include "rectangle.h"
#endif

class Window
    // ...
  public:
    // ...
    Window(const Rectangle& r);
    // ...
};

// ...


inline
Window::Window(const Rectangle& r)
{
    // ...
}

#endif
```

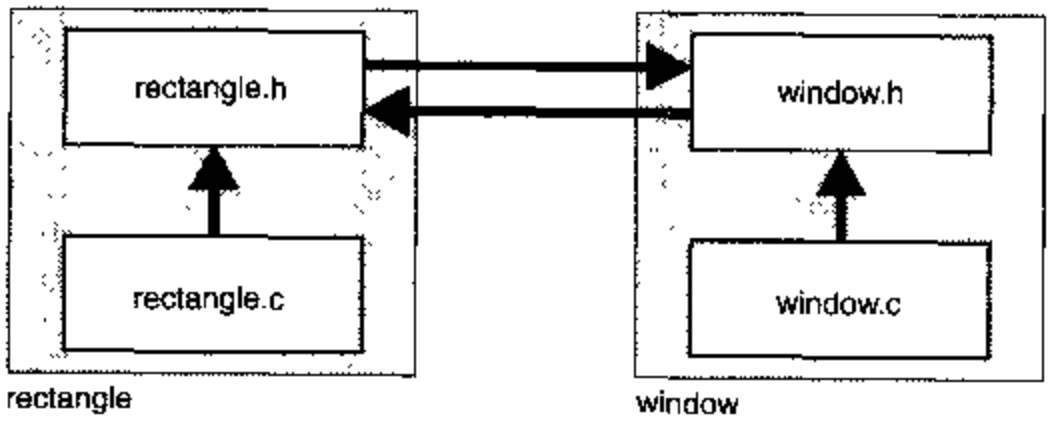


图 5-2 两个互相依赖的组件

如果你觉得这主意听起来不错，你并不是惟一这么想的人。但是它不是一个好的解决办法。首先，任何可能实现的速度优势都可能丧失，因为在进入一个函数时必须构造另一个类型的一个临时对象。由于转换是隐含和自动的，你的客户甚至可能没有意识到这个额外的临时对象的创建（并且会因为你的“缓慢的”类而责备你）。

更加重要的是，我们已经在这两个先前是独立的组件的头文件之间引入了一种循环物理依赖。这两个组件中的每一个现在都必须“知道”对方。如果没有另一个组件就不可能编译、连接、测试或使用两者中的任何一个组件。大多数客户不会关心这些类之间的性能特性方面

的细微差别，他们会选择使用其中的一个，但很少同时使用两个。这种不能层次化的增强却迫使他们两个都要使用。

我们可以把预处理器#include 指令从.h 文件移到.c 文件（如图 5-3 所示），但是这样做并不能消除物理耦合。两个组件在编译时仍然彼此依赖，并且每一个都在连接时潜在地依赖对方。我们需要做更激进一点的事情。

```
// rectangle.h
#ifndef INCLUDED_RECTANGLE
#define INCLUDED_RECTANGLE

class Window;

class Rectangle {
    // ...
  public:
    // ...
    Rectangle(const Window& w);
    // ...
};

#endif
```

```
// window.h
#ifndef INCLUDED_WINDOW
#define INCLUDED_WINDOW

class Rectangle;

class Window {
    // ...
  public:
    // ...
    Window(const Rectangle& r);
    // ...
};

#endif
```

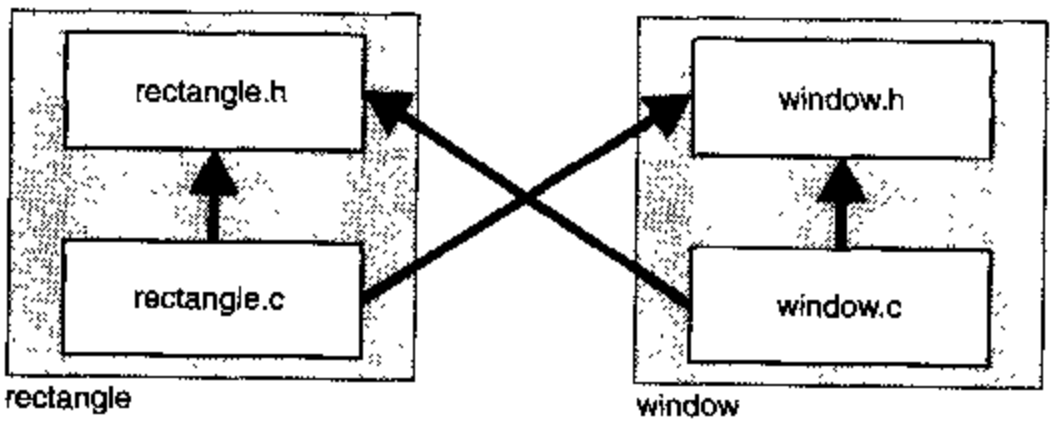


图 5-3 两个组件仍然互相依赖

定义：如果一个子系统可编译，并且单个组件（包括.c 文件）的包含指令隐含的依赖图是非循环的，则称这个子系统是可层次化的。

假设一个子系统由遵守了第 2 章和第 3 章所介绍的所有主要设计规则的组件构成。我们可以使用上述的可层次化的候选定义来帮助我们避免会导致组件变得物理耦合的增强。我们必须以某种方式找到一种方法来允许客户程序在 rectangle（长方形）和 window（窗口）之间转换，而不要求每个组件都包含对方。

5.1.2 便利方法

开发人员在使一个系统可用的过程中，常常会试图创作一些在结构上不可靠的设计。为了说明这个主题，我们引入了第二个例子——一个图形 shape 编辑器，它的设计被抽象地描述在图 5-4 中。类 Shape 是抽象的，它定义了一个协议，要求所有的具体 shape 都必须实现。每个 shape 都有一个位置坐标，我们假定现在必须尽可能快地操纵它（即，通过内联函数）。因为 Shape 类的一些功能已经实现了，所以 Shape 不仅可用来定义一个公共的接口，而且可用来分解实现的共有部分[①]。

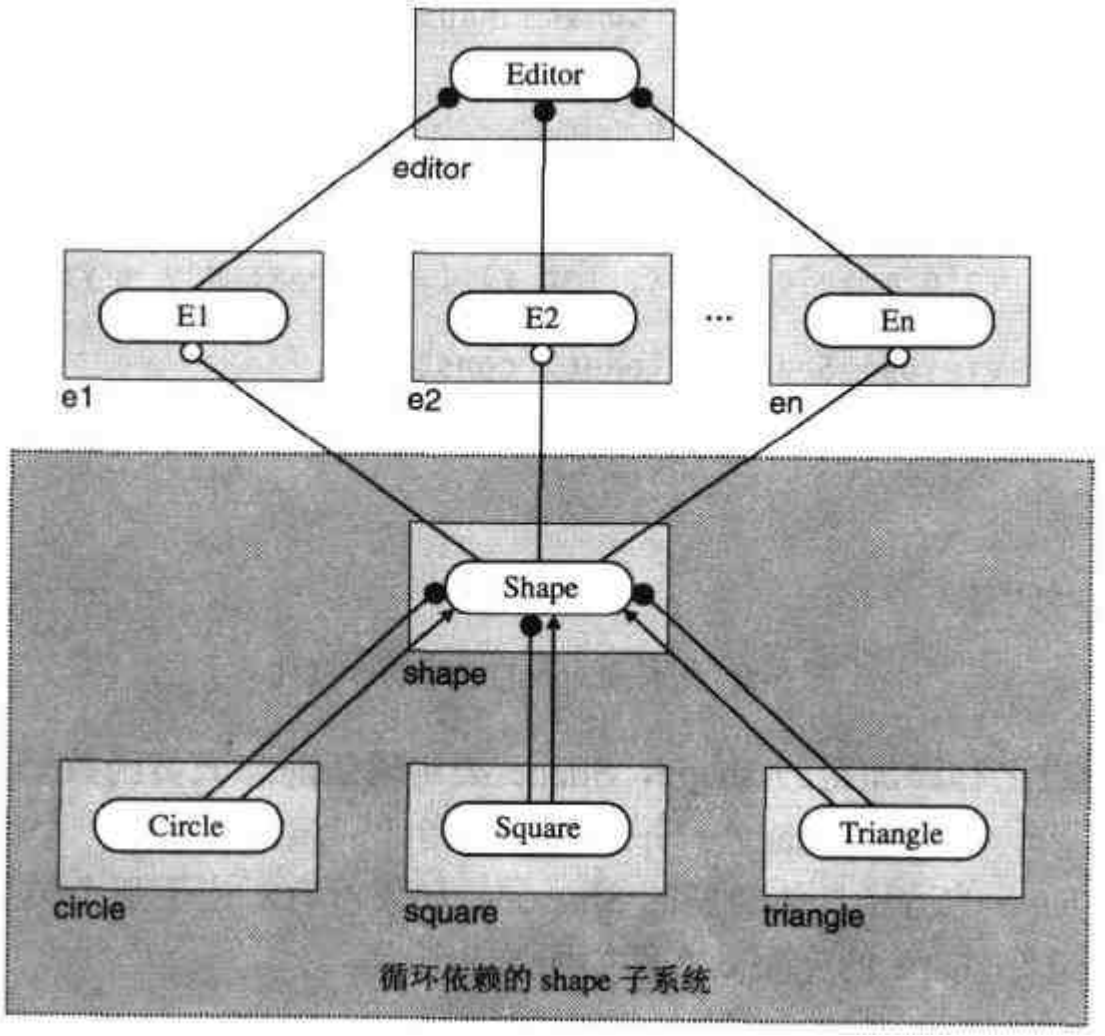


图 5-4 一个图形 shape 编辑器的没有层次化的设计

Shape 类能够潜在地定义大量的纯虚函数。shape 组件头文件的一个简略表示展示在图 5-5 中。Shape 类的客户需要能够建立真实的 shape，但是它们不需要直接与派生类接口交互。为了把 Shape 的客户与派生自 Shape 的具体类绝缘，创建特定种类 Shape 的能力被直接合并进了 Shape 的接口。

① 6.4.1 节描述了如果放宽对 moveTo 函数的速度要求，我们如何能够减少 Shape 接口的消费者与提供者之间的编译时耦合。

```
// shape.h
#ifndef INCLUDED_SHAPE
#define INCLUDED_SHAPE

class Screen;

class Shape {
    int d_xCoord;
    int d_yCoord;

  protected:
    Shape(int x, int y);
    Shape(const Shape& shape);
    Shape& operator=(const Shape& shape);

  public:
    static Shape *create(const char *typeName);
    virtual ~Shape();
    // ...
    void moveTo(int x, int y) { d_x = x; d_y = y; }
    // ...
    virtual Shape *clone() const = 0;
    virtual void draw(Screen *s) const = 0;
    // ...
};

#endif
```

图 5-5 组件 shape 的省略的.h 文件

为了更容易通过名称添加新的 shape，Shape 类实现了静态成员函数 create。该方法接受 Shape 的类型名称（一个 const char *）并返回一个指针，该指针指向一个动态分配的、重新构造的、派生自 Shape 的适当具体类型的 Shape[①]。如果没有对应于那个类型名称的 shape 存在，函数返回 0。组件 shape 的完整.c 文件如图 5-6 所示。

```
// shape.c
#include "shape.h"
#include "circle.h"
#include "square.h"
#include "triangle.h"
#include "screen.h"
#include "string.h"    // strcmp()
```

① 返回一个指向动态分配对象的指针容易产生错误，因为这样会把重新分配的责任留给客户。如果未能捕获一个异常可能很容易导致内存泄漏。句柄类（在 6.5.3 节讨论）可以用来减少内存泄漏的可能性。

```
Shape::Shape(int x, int y)
: d_xCoord(x)
, d_yCoord(y)
{}

Shape::Shape(const Shape& s)
: d_xCoord(s.d_xCoord)
, d_yCoord(s.d_yCoord)
{}

Shape& Shape::operator=(const shape& s)
{
    d_xCoord = s.d_xCoord;
    d_yCoord = s.d_yCoord;
    return *this;
}

Shape::~Shape() {}

Shape *Shape::create(const char *s)
{
    if (0 == strcmp(s, "Circle")) {
        return new Circle(x, y, 1);                    // unit radius
    }
    else if (0 == strcmp(s, "Square")) {
        return new Square(x, y, 1);                    // unit side
    }
    else if (0 == strcmp(s, "Triangle")) {
        return new Triangle(x, y, 1, 1, 1);            // unit side
    }
    else {
        return 0;                                      // unknown shape
    }
}
```

图 5-6 组件 shape 的完整的.c 文件

Editor 类本身在许多单独用在 Editor 的实现中的顾客类型（E1、…、En）之上分层。这些类型中的每一个都在其接口中使用了 Shape，以便执行不同的有关 shape 的抽象操作（例如：moveTo、scale、draw 等等）。只有一个实现组件 e1（实现加法命令）需要能够从一个类型名称创建一个 shape。其余的组件可以利用 Shape 的虚函数来访问一个特定 Shape 的功能，并且不需要直接依赖任何具体的 Shape。这听起来合理吗？

虽然这个设计从可用性的角度来看似乎是吸引人的，但它有一个设计缺陷，使得它维护起来比必要的代价会昂贵许多。Shape 的 create 成员函数使用了派生自 Shape 的每一个类的一个构造函数，这就在 Shape 和所有派生自 Shape 的类之间强加了一种相互的依赖关系。因此不可能独立于其他所有的 Shape 来测试一种特定的 Shape，从而显著地加大了在增量式测试过

程中所需的连接时间和磁盘空间。这个 shape 子系统在其他方面是水平的因而是高度可重用的，但它变成了一个要么全有要么全无的待解问题。

给这个子系统增加一种新的 shape 需要修改 Shape 基类，这可能导致其他独立派生类固有的功能出现错误。由于让一个基类“知道”它的派生类而导致的高度耦合，隐含了相当大的维护开销的增加和相当大的灵活性和重用方面的损失。

当我们考虑到只有组件 e1 需要创建每个 Editor 的具体 shape，因而只有 e1 需要依赖所有单个的具体 shape 组件时，维护方面的劣势就进一步加剧了。组件 e2、e3、…、en 只是通过抽象基类 Shape 的虚函数使用这些 shape。如果我们可以假设每个 shape 的功能都正在正确地工作，那么我们只需要测试出每一个编辑器子系统组件都正在正确地与 Shape 协议交互。可能会有几十个甚至几百个不同种类的 shape，始终用每一个类型的 shape 去测试每一个编辑器子系统组件，既无必要也不实际。但是，因为 shape 子系统中存在的耦合，无论何时想要测试任何一个编辑器实现组件，我们都被迫要连接所有的 shape。

为了提高这个系统的可维护性，我们必须找到一种方法来重新打包 shape 子系统，使它变成非循环的，从而可层次化。

5.1.3 本质的相互依赖

对象的相互连接网络给软件系统体系结构设计者们提出了一种工程挑战。这种高度的内在耦合（尤其是在接口中的）使得很难明显而直观地实现层次化。在这个最后的介绍性的例子中，我们将分析实现一个存在于最基本的对象网络中的图的困难。

原 则

相关抽象接口中的内在耦合使它们更抗拒层次分解。

考虑图 5-7 中的图。该图由节点和边的集合构成。这个图内的节点由有向边连接。通常，图中的边会形成循环[①]。每个节点由一些数据和一些信息（有关节点如何组织进图的信息）组成。在这个例子中节点的数据只是一个名称。连通性被简单地表示为一个边（从某个节点开始或到该节点为止的边）的列表。

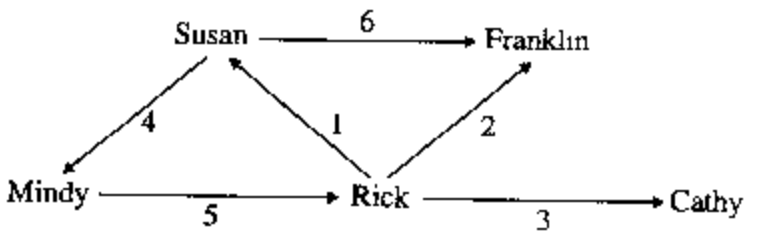


图 5-7 由节点和边组成的简单图

图 5-8 描述了与 node 组件有关的最小的功能。给定一个 Node，可能要请求它的名称，找出连接到它的边的数量，然后通过提供 0 到 *N*–1 之间的整数下标在这些边上进行迭代（这里

① 注意：这些是实例之间的循环，不是类之间的循环。

N 是由 Node::numEdges()返回的当前值）[1]。

```
// node.h
#ifndef INCLUDED_NODE
#define INCLUDED_NODE

class Edge;

class Node {
    // ...
    Node(const Node&);                          // not implemented
    Node& operator=(const Node&);               // not implemented

  public:
    Node(const char *name);
    ~Node();
    const char *name() const;
    int numEdges() const;
    Edge& edge(int index) const;
};

#endif
```

图 5-8　组件 node 的公共接口

图 5-9 描述了与 edge 组件有关的最小功能。在这个系统中，一个 Edge 用于连接节点。像节点一样，边也同时包含局部的和网络相关的功能。在这个例子中，与 Edge 有关的、独立于网络的信息只是它的权值，并且连通性信息也只是 Edge 连接的两个 Node 对象。

```
// edge.h
#ifndef INCLUDED_EDGE
#define INCLUDED_EDGE

class Node;

class Edge {
    // ...
    Edge(const Edge&);                          // not implemented
    Edge& operator=(const Edge&);               // not implemented

  public:
    Edge(Node *from, Node *to, double weight);
    ~Edge();
    Node& from() const;
    Node& to() const;
    double weight() const;
};

#endif
```

图 5-9　一个 edge 组件的公共接口

① 为迭代提供一个整数下标暗示着底层实现有可能是一个某种数组而不是边的连接链表。一个笨拙的链表实现会导致迭代过程中的二次运行时性能。

最初，我们面对的是图 5-10 中展示的毫无吸引力的设计。Node 在它的接口中使用了 Edge，反之亦然。按照现在的情况，似乎类 Node 和类 Edge 一定是相互依赖的——否则客户怎么可能遍历这个图呢？另外，还存在谁拥有这些对象的内存以及谁有权产生和取消 Node 和/或 Edge 的实例的问题。

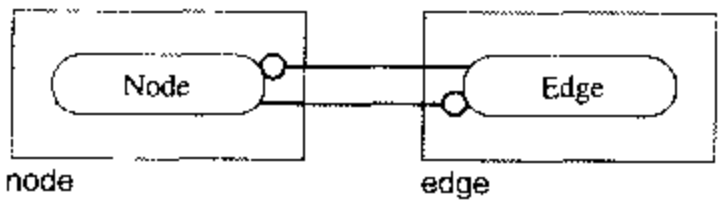


图 5-10 循环依赖的 node 和 edge 组件

回忆一下 3.6 节的介绍，友元关系本身不会引入物理依赖，但是为了保持封装，它可能间接地导致物理耦合的出现。为了避免出现封装缺口和由于远距离友元关系导致的缺乏模块性，可能有必要将若干可层次化的类归入一个单一的组件内（正如在 5.9 节的结尾所解释的那样）。这种耦合的常见例子实际上可以在每一个提供迭代器的容器组件内看到。迭代器永远是容器的一个友元，因而定义在相同的组件内。

以上介绍的只是实践中通常会出现的循环耦合种类的一些例子。本章的其余部分致力于论述各种技术和转换，以理清可能在其他方面挑战非循环物理实现的设计。

5.2 升级

现在让我们返回到包含两个循环依赖组件（rectangle 和 window）的例子（显示在图 5-1 中）。假设不再让 rectangle 和 window 彼此“知道”，而是由我们随机地决定 rectangle 比 window 更基本。我们可以把两个转换都移进 Window 类中。现在 Window “使用” Rectangle，但反之则不然，如图 5-11 所示。

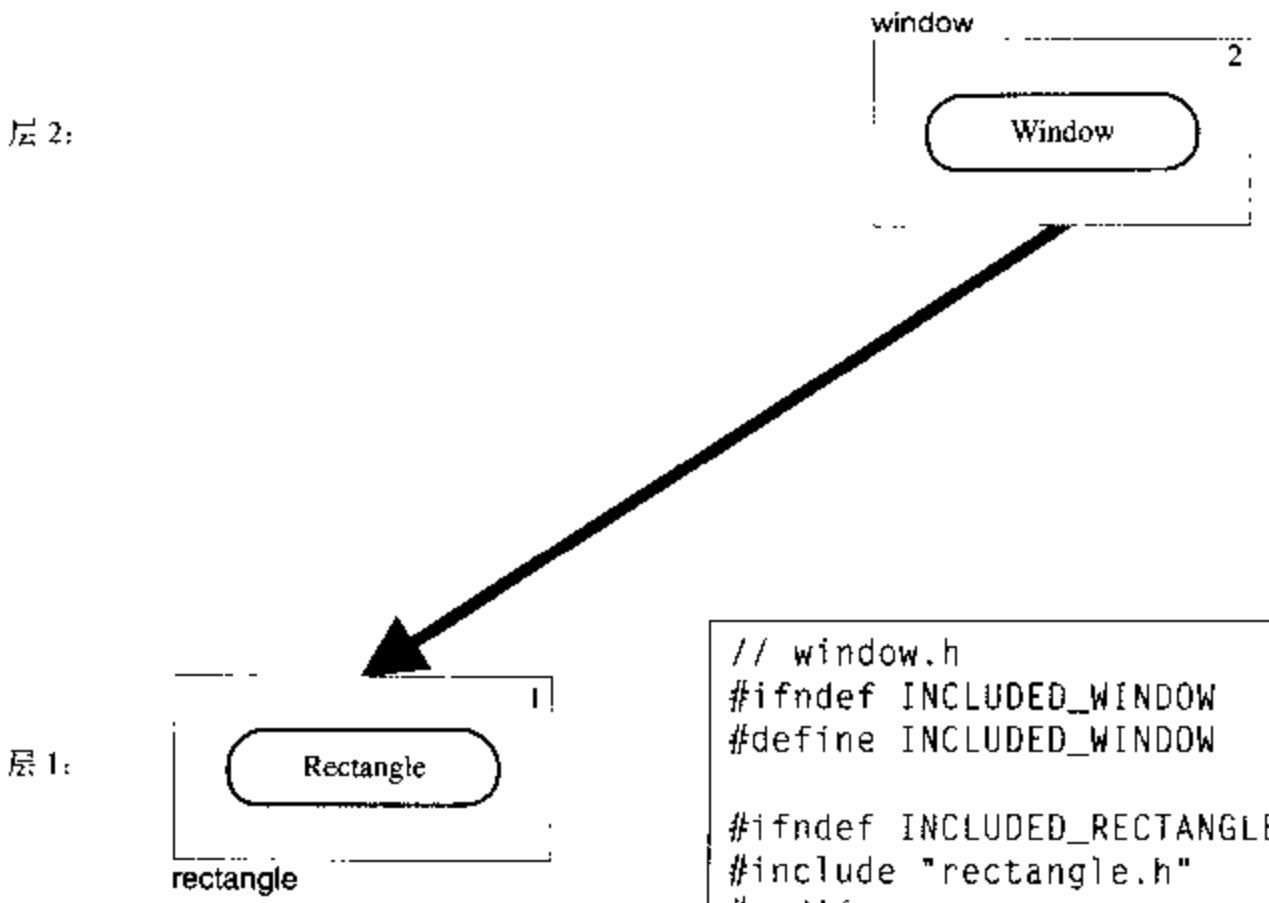
这种解决办法要求我们稍微改变一下我们的观点，因为 Rectangle 和 Window 类不再是对称的了。Rectangle 居于第一层，而 Window 现在定义在第二层。如果我们需要任何旧的盒子，我们可以重用 Rectangle 而不用担心 Window 或类之间的转换。然而，如果我们需要一个 Window，我们仍将不得不使用 Rectangle。

> **定义**：如果组件 y 处在比组件 x 更高的层次上，并且 y 在物理上依赖 x，则称组件 y 支配（**dominate**）组件 x。

支配是一种组件之间的属性，它和一个单一派生对象内的虚基类之间的同名属性大致相同[1]。我们现在介绍了组件之间的支配概念，并提到图 5-11 中展示的一个例子，其中组件

① **ellis**，10.1.1 节，204～205 页。

window 支配组件 rectangle。在后面的小节中我们会提到“支配”这个定义。




```
// window.h
#ifndef INCLUDED_WINDOW
#define INCLUDED_WINDOW

#ifndef INCLUDED_RECTANGLE
#include "rectangle.h"
#endif

class Window {
    // ...
  public:
    // ...
    Window(const Rectangle& r);
    // ...
    operator Rectangle() const;
    // ...
};

// ...

inline
Window::operator Rectangle() const
{
    // ...
}

#endif
```

```
// rectangle.h
#ifndef INCLUDED_RECTANGL
#define INCLUDED_RECTANGL

class Rectangle {
    // ...
  public:
    // ...
};

#endif
```

图 5-11 window 支配 rectangle

如图 5-12 所示，组件 u 支配组件 r 和组件 s。虽然组件 v 处在一个比组件 r 或组件 s 都更高的层次上，它却只支配组件 t。组件 w 支配所有的五个组件：r、s、t、u 和 v。

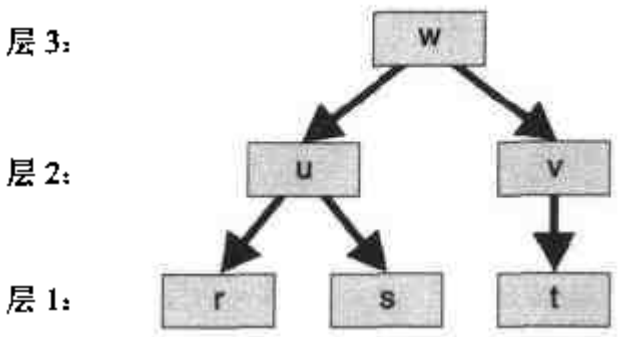


图 5-12 组件支配性质的阐释

支配的重要性在于它能够提供简单的层次号之外的额外信息。例如，增加一个高层次组件对低层次组件（如在图 5-12 中，u 对 t）的依赖决不会引入一个循环依赖或改变层次号[如图 5-13（a）所示]。

在同一层的两个组件之间（如在图 5-12 中，v 对 u）增加一个依赖也不会引入一个循环，但是会影响层次号，如图 5-13（b）所示。最后，甚至也有可能增加一个从低层次组件到高层次组件的依赖（如在图 5-12 中，t 对 u，没有引入一个循环依赖）。当且仅当组件 u 不是已经支配组件 t 时，才可能增加依赖而不引入循环。在这里，组件 u 没有支配组件 t，增加 t 对 u 的依赖的结果显示在图 5-13（c）中。

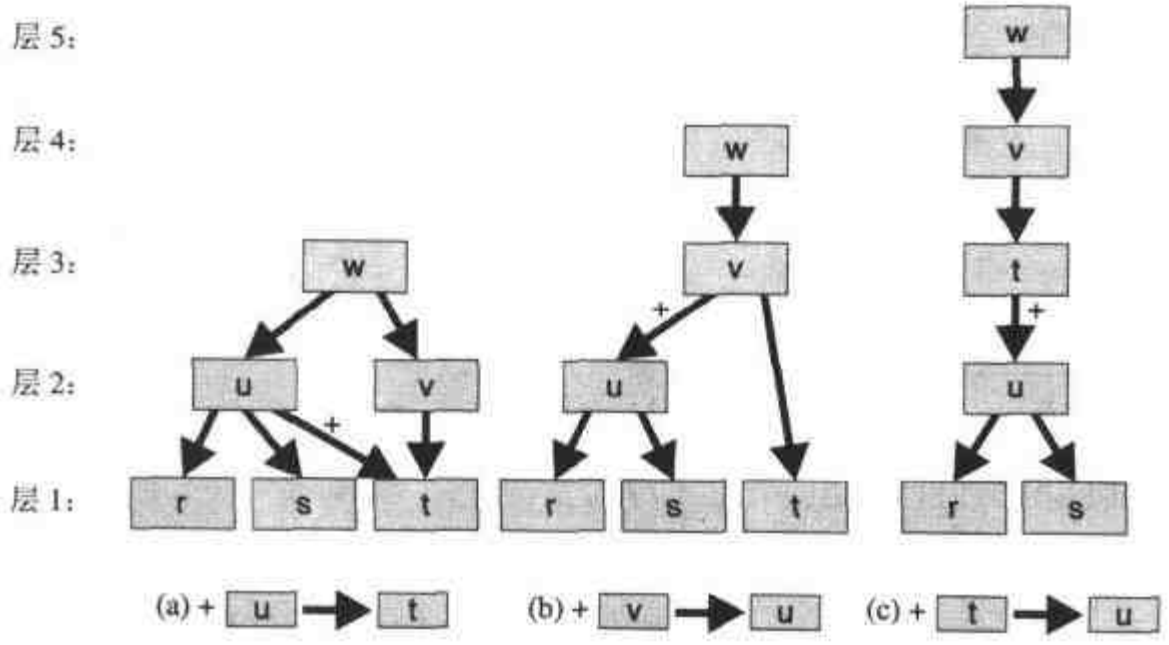
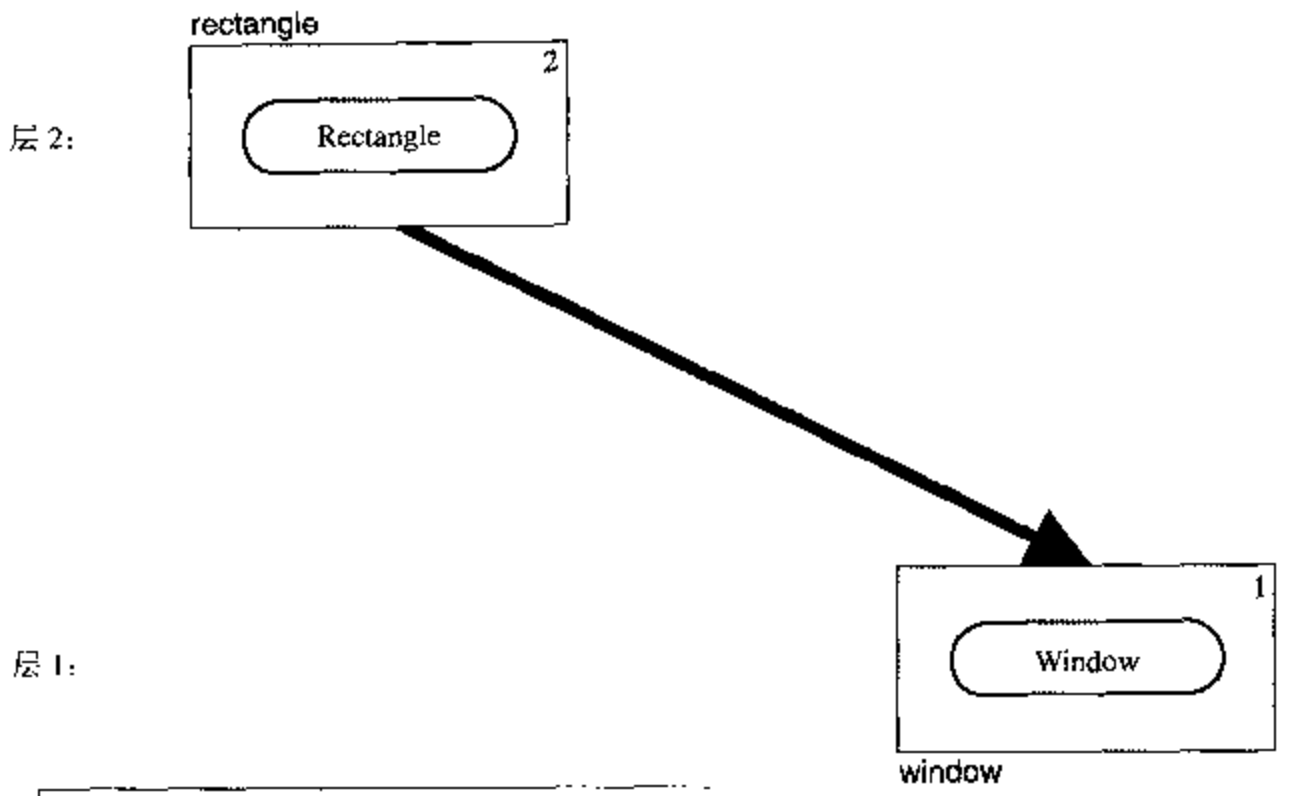


图 5-13 增加一个依赖可能引起层次号的改变

当然我们可能已经走到另一条道上去了，并使 Window 成了基本的对象。在那种情形下，rectangle 知道 window，但反之则不然。这种情形在图 5-14 中描述。请注意在这个例子中我们已经选择了将#include "window.h"指令移到 rectangle.c 文件中，这就意味着转换例程将不是内联的。




```
// rectangle.h
#ifndef INCLUDED_RECTANGLE
#define INCLUDED_RECTANGLE

class Window;

class Rectangle {
    // ...
  public:
    // ...
    Rectangle(const Window& w);
    // ...
    operator Window() const;
    // ...
};

#endif
```

```
// window.h
#ifndef INCLUDED_WINDOW
#define INCLUDED_WINDOW

class Window {
    // ...
  public:
    // ...
};

#endif
```

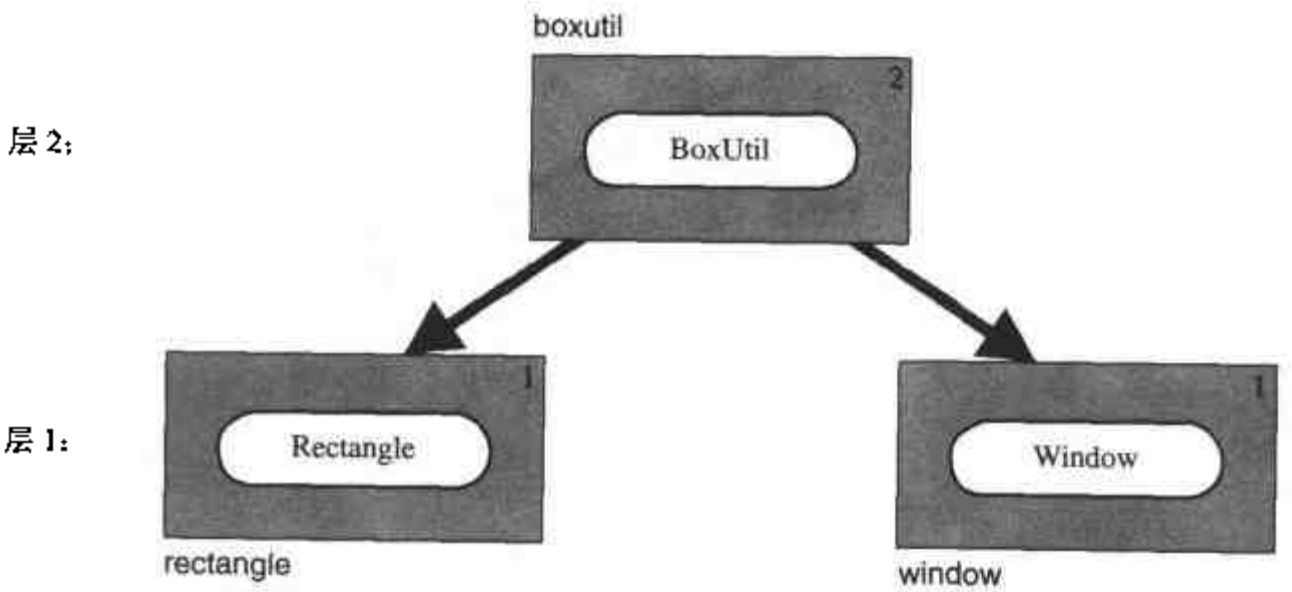
图 5-14　rectangle 支配 window

两种解决方法都意味着只有一个组件的使用能够独立于另一个组件。两种解决方案都是基于最初的循环依赖设计改进的，但我们还能做得更好一些。许多使用这些组件的客户都只需要其中的一个而不是同时需要两个。在那些确实需要使用两个组件的客户中，只有少许会需要在它们之间转换。为了使独立可重用性最大化，我们可以通过将引起循环的功能移到一个更高的层次上来避免让一个组件支配另一个组件——这种技术在本书中称为升级（escalation）。

原　则

如果同层次的组件是循环依赖的，那么就有可能把互相依赖的功能从每一个组件升级为一个潜在的新的更高层次组件（依赖于每一个初始的组件）的静态成员。

在公司里，如果两个雇员不能解决一个争论，通常的做法是将问题提交到更高层。在对象为了“支配”而竞争的情况下，同样的解决方法通常也是有效的。我们可以创建一个名为 BoxUtil 的工具类，它既知道 Rectangle 类也知道 Window 类，然后把这个类的定义放在一个完全独立的组件中，如图 5-15 所示。




```
// boxutil.h
#ifndef INCLUDED_BOXUTIL
#define INCLUDED_BOXUTIL

class Rectangle;
class Window;

struct BoxUtil {
    static Window toWindow(const Rectangle& r);
    static Rectangle toRectangle(const Window& w);
};

#endif
```

```
// rectangle.h
#ifndef INCLUDED_RECTANGLE
#define INCLUDED_RECTANGLE

class Rectangle {
    // ...
  public:
    // ...
};

#endif
```

```
// window.h
#ifndef INCLUDED_WINDOW
#define INCLUDED_WINDOW

class Window {
    // ...
  public:
    // ...
};

#endif
```

图 5-15 rectangle 和 window 都不支配对方

现在对 Rectangle 类或 Window 类有兴趣的客户可以自由地独立使用任何一个类。如果一个客户碰巧使用了两个类但不需要在它们之间转换，情况也是如此。如果还有其他客户需要转换程序，他们也可以得到。但是要注意，Rectangle 和 Window 之间的转换通常是隐式的，现在则必须显式地执行（要了解更多关于隐式转换的内容请看 9.3.1 节）。

注意，在前面的例子中，定义 BoxUtil 时我们选择了使用关键词 struct 来代替 class，暗示这个类型只是为公共嵌套类型和公共静态成员函数提供一个作用域。在这个约定中，一个 struct 的所有成员都是公共的，因此没有数据成员。虽然创建这样一个类型的一个实例是无意义的，但它不会造成真正的危害。如果我们抑制住将未实现的默认构造函数声明为 private 的冲动，我们就可以减少一些不必要的混乱。

现在让我们再来思考定义在图 5-4 的 shape 层次结构的基类中的静态 create 函数引起的物理耦合。假设我们通过引进一个新的工具类 ShapeUtil（其惟一的用途是创建 shape）将 create 在它的派生类层次之上升级。这个新的类将放在它自己的组件内，并且包含来自初始的 Shape 类的 create 函数，如图 5-16 所示。

```
// shapeutil.h
#ifndef INCLUDED_SHAPEUTIL
#define INCLUDED_SHAPEUTIL

class Shape;

struct ShapeUtil {
    static Shape *create(const char *typeName);
};

#endif
```

图 5-16　新的 shapeutil 组件的头文件

通过增加一个新的组件把 Uses 关系升级到一个更高的层次，我们已经消除了 shape 子系统中所有组件之间的循环依赖。新系统的层次化图如图 5-17 所示。

现在对每一个具体的 shape 都有可能进行隔离的测试。在 shape 组件的测试驱动程序中，通过从 Shape 派生一个具体的“桩（stub）”类，甚至能够对 Shape 类提供的部分实现进行模块化测试。现在每一个具体的 shape 在任何组合中都能独立于其他 shape 而重用。例如，一个系统现在可以重用 circle 和 square，而不必连接 triangle。

现在也有可能测试 E2、…、En 中的每一个，而不必连接每一个具体的 shape。因为这些组件只需要 shape 基类接口，所以，只需在所有可访问的具体 shape 中的一个有代表性的例子上，测试由每一个编辑器组件 e2、…、en 添加的增加值，可以认为这已经足够了。

这个基于最初设计的新设计的好处是减少了耦合，这种减少在扩大重用潜力时，将直接

转换成开发和维护开销的减少。当编辑器中的实现组件的数量，尤其是具体 shape 的数量较少时，也许很难意识到这种设计方法的重要性。真正的好处在于这种新的设计方法在更多的编辑器命令和新的 Shape 加进来时，在按比例扩展方面要比原来的方法好得多。

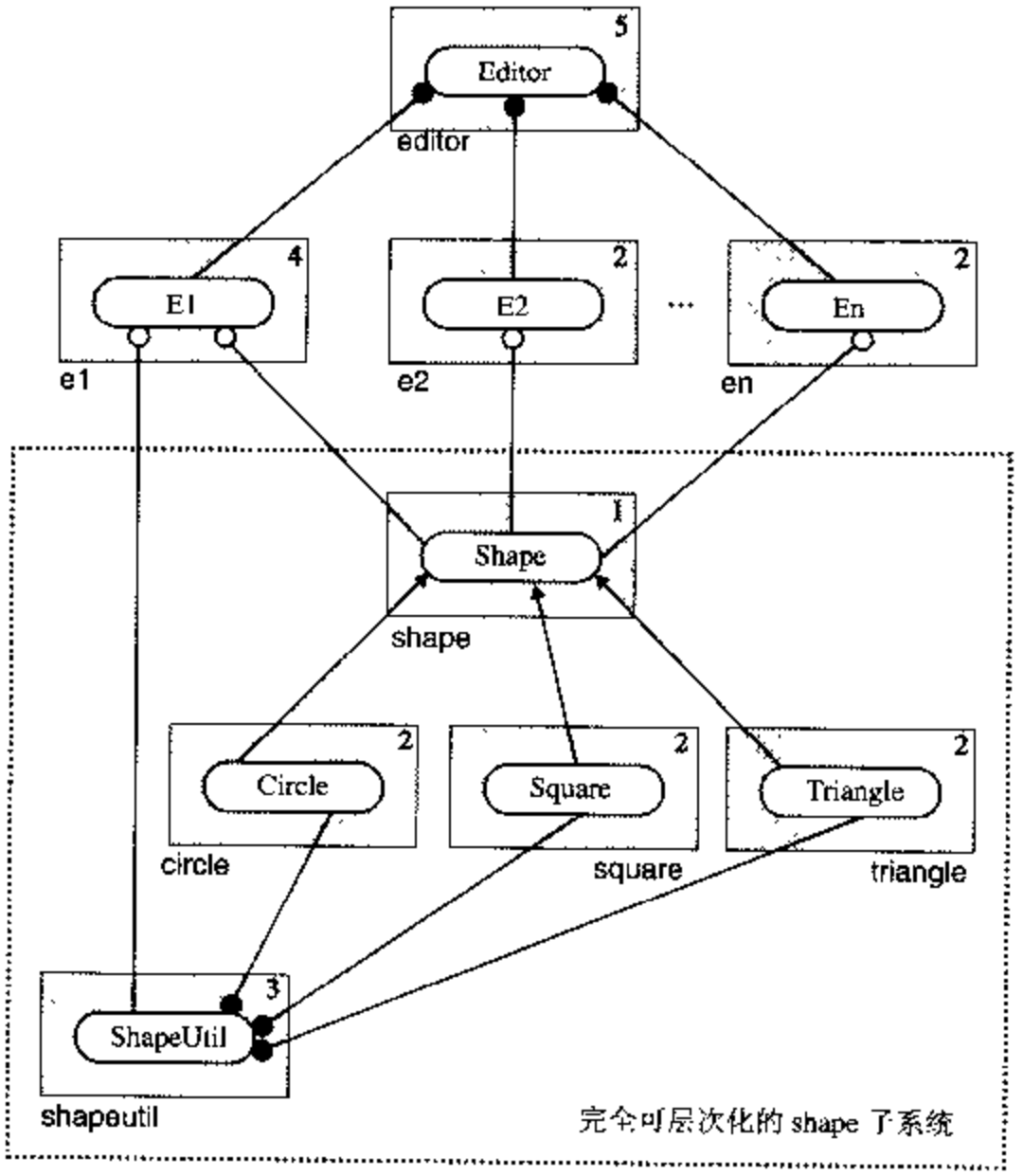


图 5-17 对 Shape Editor 的改进设计

作为对层次化给这个设计所带来改进的客观的、定量的估算，让我们来思考这个编辑器系统的四种不同的形式。在图 5-18（a）中的这个系统的“缩减”版本中，编辑器以及它所影响的 shape 的数量都是较少的（三种 shape 和三个编辑器实现组件）。rectangle 组件左上角的数字指出了为了增量式地测试该组件而必须连接的组件数量。

乍一看，这种新的设计似乎有不必要的复杂性，但事实上它简化了开发人员和客户的工作。甚至包括新设计中的额外组件，与分层次测试这个 shape 子系统有关的耦合（以 CCD 计算）也减少了整整 25%。与增量式测试编辑器子系统有关的耦合降低了 17.4%，以 CCD 计算总的减少了 20.5%。

图 5-18（b）描述了当编辑器子系统变得更大时（30 个实现组件，而不仅仅是 3 个）的

效果。现在编辑器子系统组件耦合的减少量接近 46%，将以 CCD 来计算的整体减少量提到 43.3%。

原　则

在庞大的低层子系统中的循环物理依赖将最大程度地增加维护系统的总开销。

在物理层次结构的较低层次的循环耦合可能会对维护客户程序的开销产生戏剧性的影响。正如在图 5-18（c）中可以看到的，当 shape 层次结构变得较大时（30 个具体类型，而不仅仅是 3 个），新设计的优势，以 CCD 来计算，不仅总计降低了 shape 子系统的超过 90%的耦合，而且降低了编辑器子系统超过 44%的耦合，整体减少将近 85%。当 shape 子系统和编辑器都很大时，整体降低耦合的百分比还会继续上升，如图 5-18（d）所示。

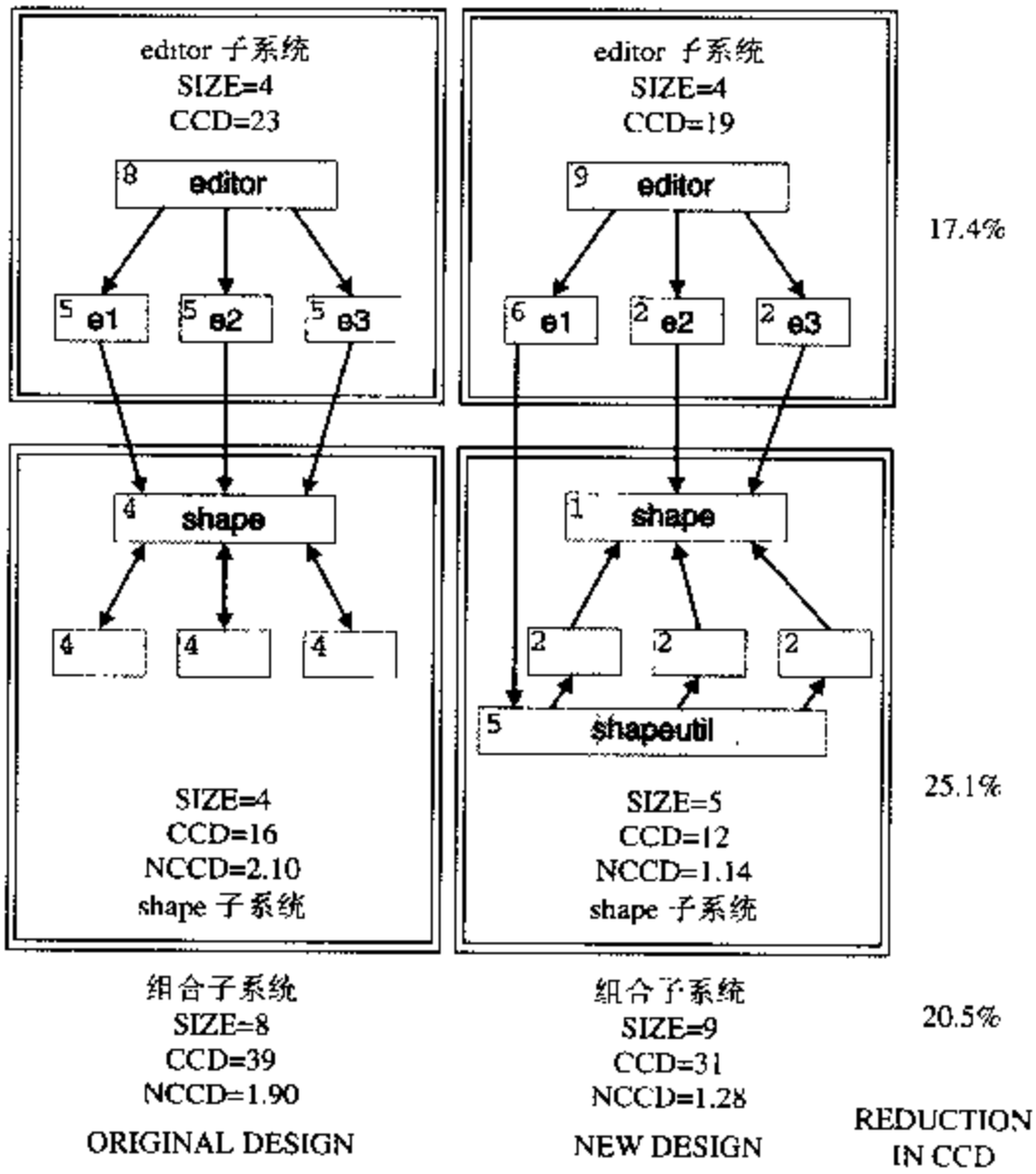


图 5-18（a）　小型 Shape 层次结构（3 个组件），小型 Editor（3 个组件）

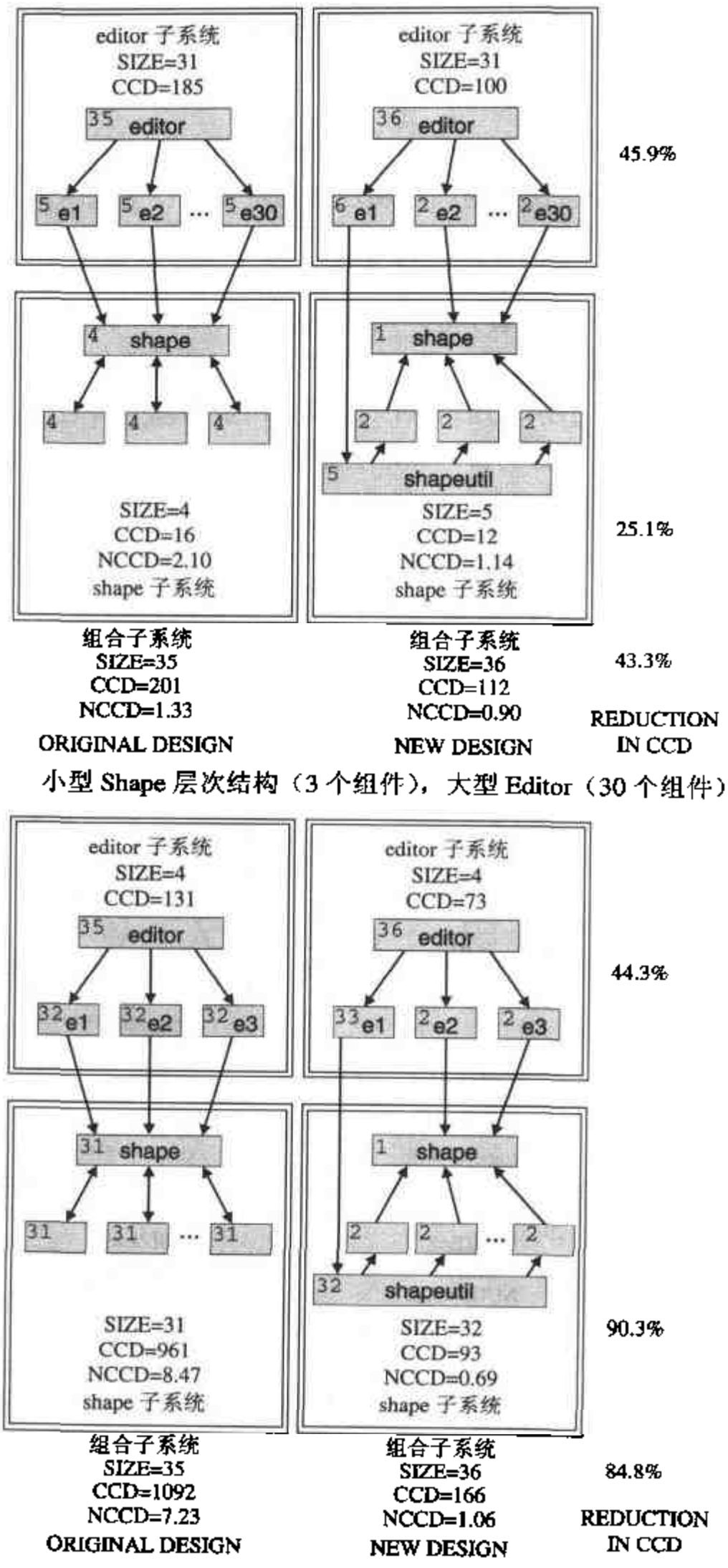


图 5-18（b） 小型 Shape 层次结构（3 个组件），大型 Editor（30 个组件）

图 5-18（c） 大型 Shape 层次结构（30 个组件），小型 Editor（3 个组件）

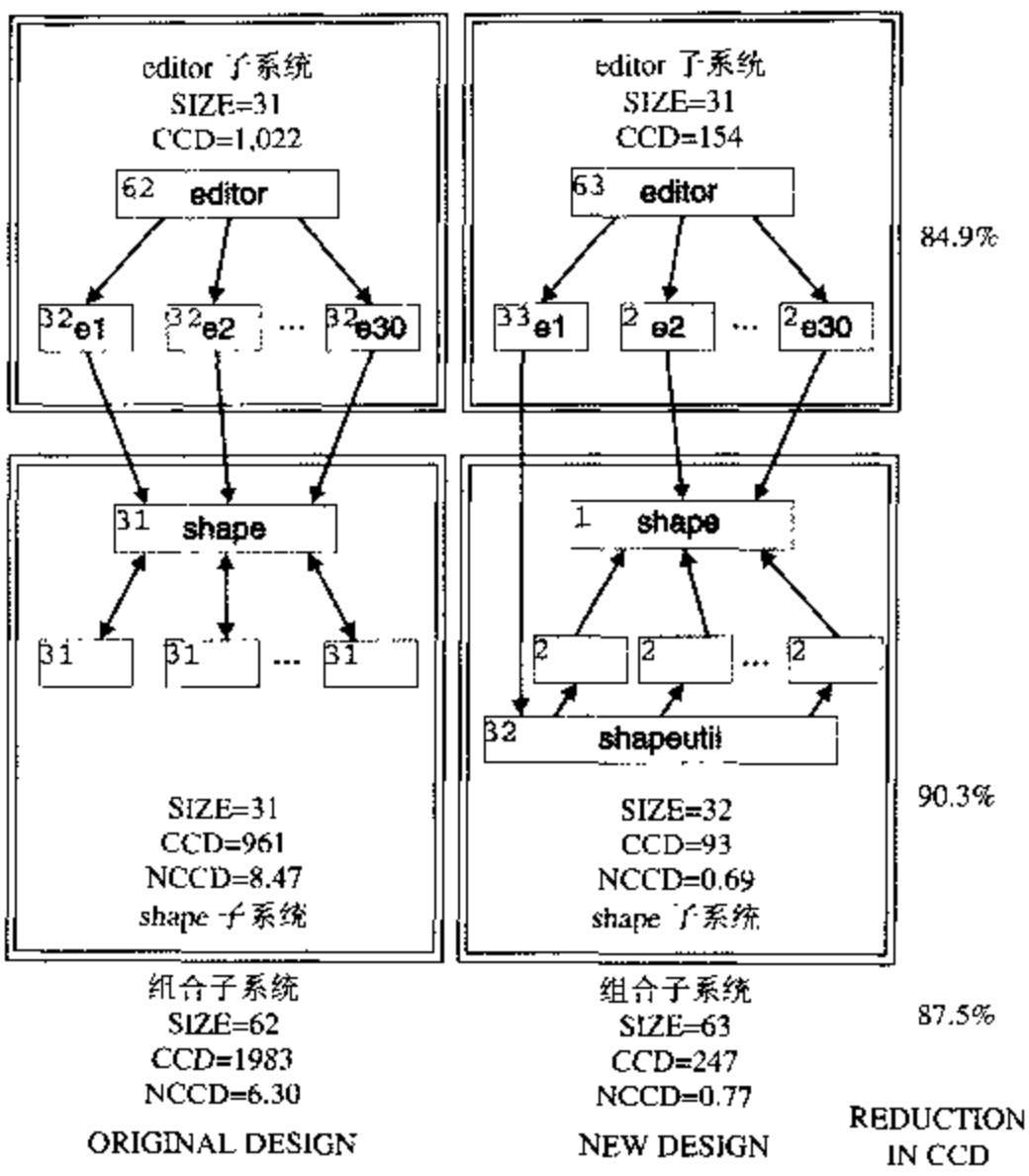


图 5-18（d） 大型 Shape 层次结构（30 个组件），大型 Editor（30 个组件）

从这个分析中得到的重要结论是，与低层次子系统有关的高度耦合可能会戏剧性地增加开发和维护较高层次客户程序和子系统的开销。

总结：可以通过把相互依赖升级到更高的层次将循环依赖转换成受欢迎的向下依赖。通过避免子系统本身内部组件之间不必要的依赖，可以显著降低子系统和它的所有客户程序的维护开销。同时，子系统也可以变得更灵活从而更可重用。这个改进设计的好处对于一个系统的较小版本也许并不显著。

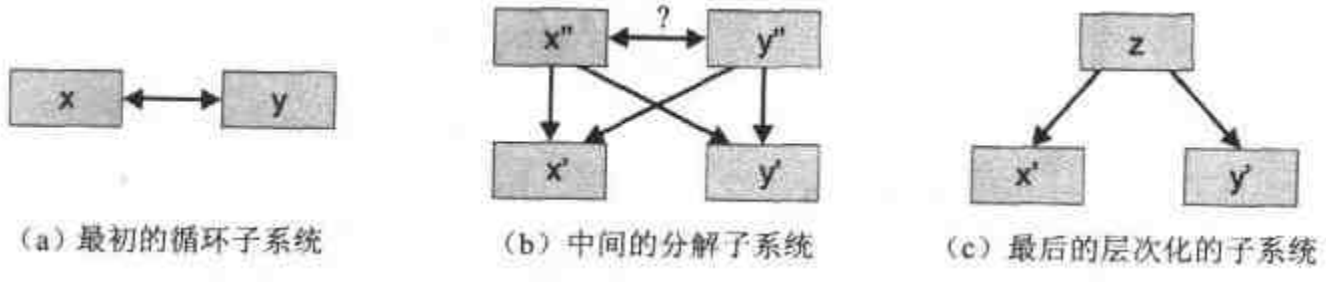
5.3 降级

到现在为止，我们已经努力通过把互相依赖的功能推到物理层次结构的更高层来消除循环依赖。在这一节中，我们要探讨把公用的功能推到物理层次结构的更低层的技术，在那里公共功能可以被共享甚至也许可以被重用。这种把公用的功能移到物理层次结构的更低层的技术在本书中称为**降级（demotion）**。

原　则

如果同一层次的组件是循环依赖的，那么就有可能把互相依赖的功能从每一个组件降到一个潜在的新的较低级（共享）组件中，每一个原来的组件都依赖于这个新组件。

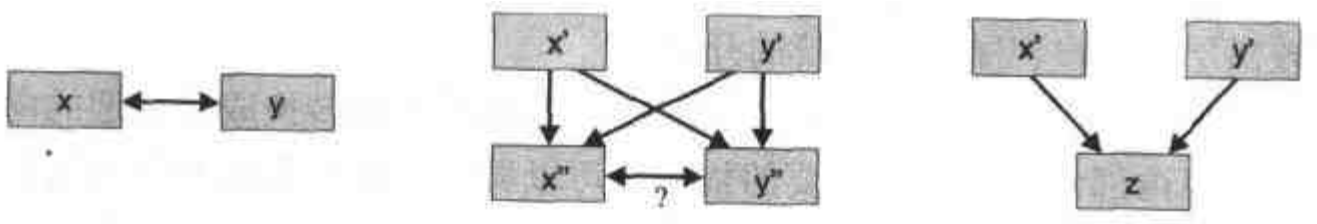
升级和降级是相似的，因为在这两种情况下，都是通过把循环依赖功能移到物理层次结构的另一层的方法来消除组件之间的循环依赖。让我们从分析在一个更通用的升级形式的过程中会发生什么开始。如图 5-19 所示，两个互相依赖的组件（a）被分解成四个组件（b），其中两个也许是相互依赖的而另外两个是独立的。如果有必要避免一个循环依赖，或者如果是内聚的，要降低物理复杂度，这两个较高层次的组件有可能被组合成（c）。



（a）最初的循环子系统　（b）中间的分解子系统　（c）最后的层次化的子系统

图 5-19　使用升级来解开循环依赖

现在，将此过程和常用的降级过程相比较。如图 5-20 所示，两个相互依赖的组件（a）再次被分解成四个组件（b），其中两个组件依赖于另外两个组件，这两个组件也可能是相互依赖的。然后如果有必要避免一个循环依赖，或者是内聚，为了降低物理复杂度，这两个较低层次的组件有可能被合并（c）。



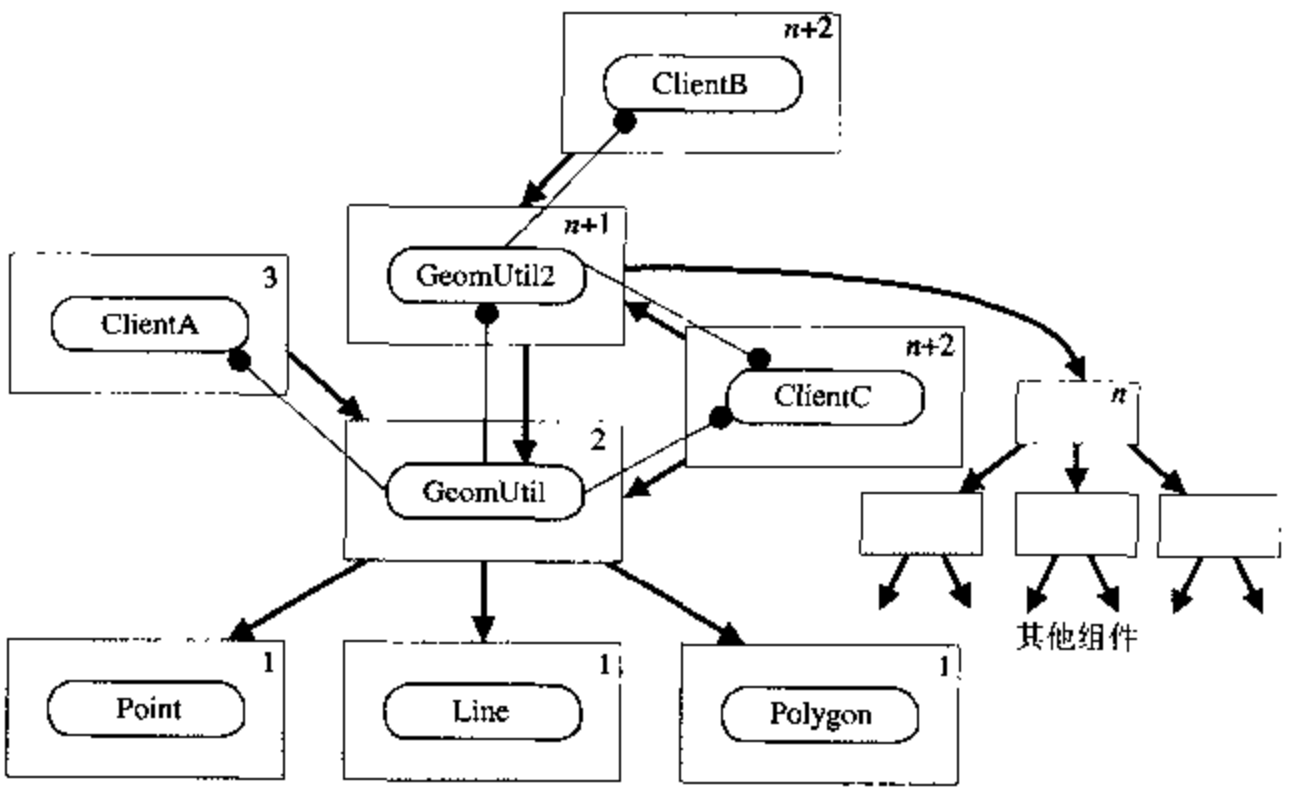
（a）最初的循环子系统　（b）中间的分解子系统　（c）最后的层次化的子系统

图 5-20　使用降级来解开循环依赖

考虑图 5-21 所示的情形，图中有两个几何工具类，GeomUtil 和 GeomUtil2。每一个工具都提供一套对点、线和多边形操作的函数。外部客户直接使用其中一个类或同时使用两个。和 geomutil 不同，geomutil2 是复杂的，它依赖许多其他的组件，甚至在它的接口中暴露了一些新的类型。那些只需要 GeomUtil 提供的基本几何功能的客户程序不需要和 geomutil2 组件连接。

最初这两个组件是可层次化的，geomutil2 依赖 geomutil。但是，不是所有的开发人员都会小心地考虑他们工作中的物理隐含。开发人员有一天会发现，由于不小心的增强，这两个

```
// geomutil2.h
#ifndef INCLUDED_GEOMUTIL2
#define INCLUDED_GEOMUTIL2

class Line;
class Polygon;

struct GeomUtil2 {
    static int crossesSelf(const Polygon& polygon);
    static int doesIntersect(const Line& line1, const Line& line2);
    // ...
}

#endif
```




```
// geomutil.h
#ifndef INCLUDED_GEOMUTIL
#define INCLUDED_GEOMUTIL

class Point;
class Line;
class Polygon;

struct GeomUtil {
    static int isInside(const Polygon& polygon, const Point& point);
    static int areColinear(const Line& line1, const Line& line2);
    static int areParallel(const Line& line1, const Line& line2);
    // ...
}

#endif
```

图 5-21　两个几何工具组件：geomutil 和 geomutil2

几何工具组件已经变得相互依赖了。GeomUtil2::crossesSelf 现在依赖于 GeomUtil::areColinear，而 GeomUtil::isInside 现在依赖于 Geomutil2::doesIntersect。我们应该怎么办呢？

我们有几个选择。首先我们可以对功能重新打包，这样就可以恢复单向依赖，这也许是正确的答案。例如，我们可以把 doesIntersect 移到 GeomUtil，把 isInside 移到 Geomutil2。现在这些组件之间不再有循环依赖了，虽然这些组件的客户程序可能受影响。（一种通用的给组件重新打包的技术在本节结尾部分阐述。）

也可能有这样的情形，由于那些依赖它们的客户程序的要求，两个组件已具有截然不同的特征。在那种情况下，分解出共有的功能，并把它降级到物理层次结构的较低层也许更合适，如图 5-22 所示。就是说，我们可以把 doesIntersect 和 areColinear 函数都移到 GeomUtilCore 中去。

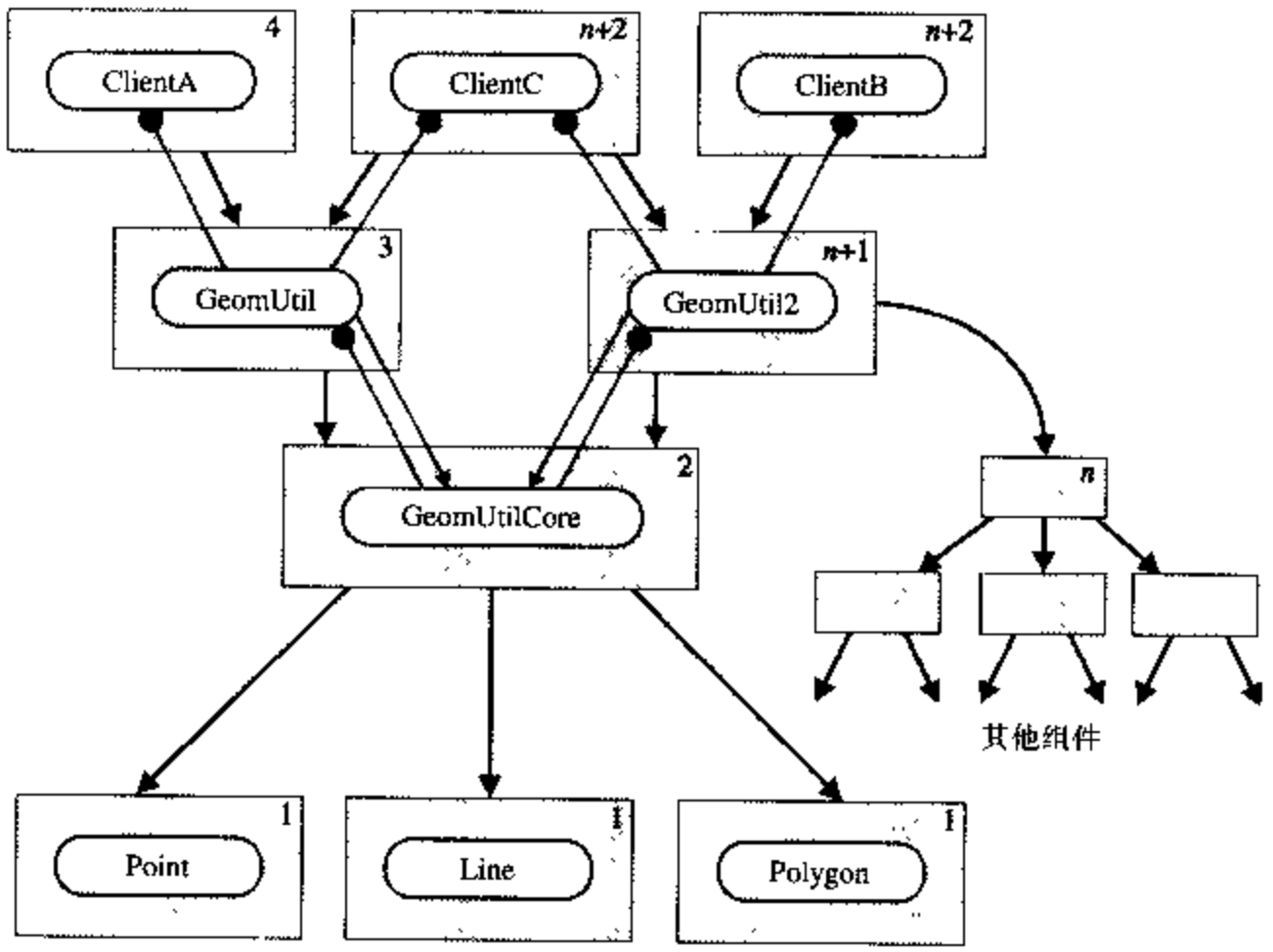


图 5-22 将（工具）类之间的共有功能降级

请再次注意这些工具类，只是在其中声明静态成员函数的作用域——绝对没有准备用它们来创建对象。如果通过变“戏法”让两个原有的工具都公共派生于一个公共核心，那么当一个或多个所使用的工具函数被降级时，原有工具的客户程序将不必修改他们的代码。

降级对于减少某些设计的 CCD（甚至在没有循环依赖时）来说是一种很有用的工具。假设一个 x 组件以高 CCD 只依赖于另一复杂组件 y 的一部分，如图 5-23（a）所示。如果我们能够把 y 的共有部分降级，我们也许能免除 x 的一些由 y 导致的物理依赖，见图 5-23（b）。

（a）降级前　　（b）降级后

图 5-23　使用降级减少 CCD

原　则

将共有代码降级可促成独立重用。

图 5-24 描述了这样一个情形：定义在子系统 A 中的枚举值被用于整个系统，但是子系统 B 是另外独立于子系统 A 的。

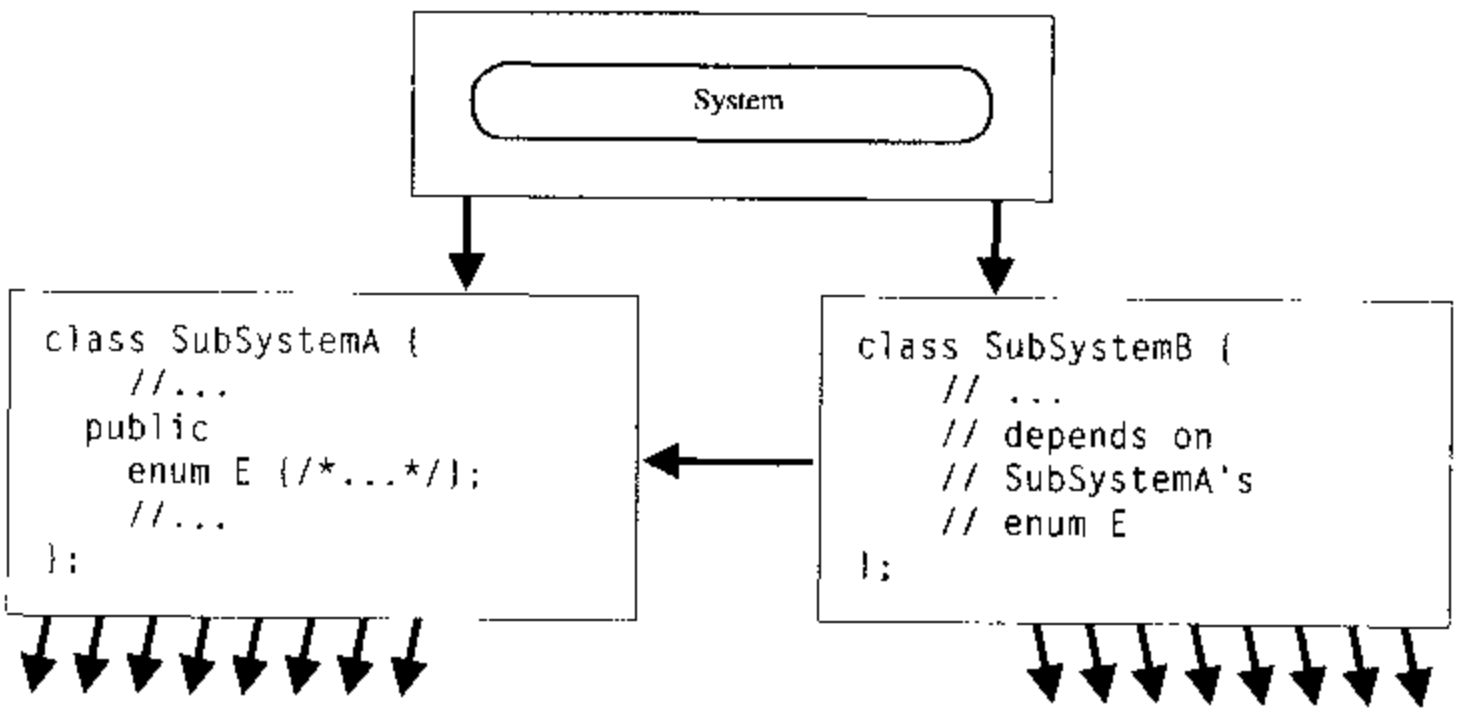


图 5-24　不良分解的系统结构

虽然目前没有子系统 B 对子系统 A 的连接时依赖，但这个事实仅通过检查提取出的包含图是搞不清楚的。包含图指出，最初的结构设计已经允许子系统 B 中的组件任意地依赖子系统 A 中的组件。日常维护将立刻不可避免地引起有更多实质性的、B 对 A 的连接时依赖，这又会影响维护子系统 B 的连接时间开销。

通过一开始就在 SubSystemA 中创建一个单独的类（或 struct）来限制枚举 E 的作用域，并将此作用域受限制的枚举移到一个单独的组件中，我们可以消除子系统 B 对子系统 A 的任何物理依赖。如图 5-25 所示，将枚举 E 降级会减少耦合和简化理解子系统 B 的任务，从而降低维护整个系统的开销。

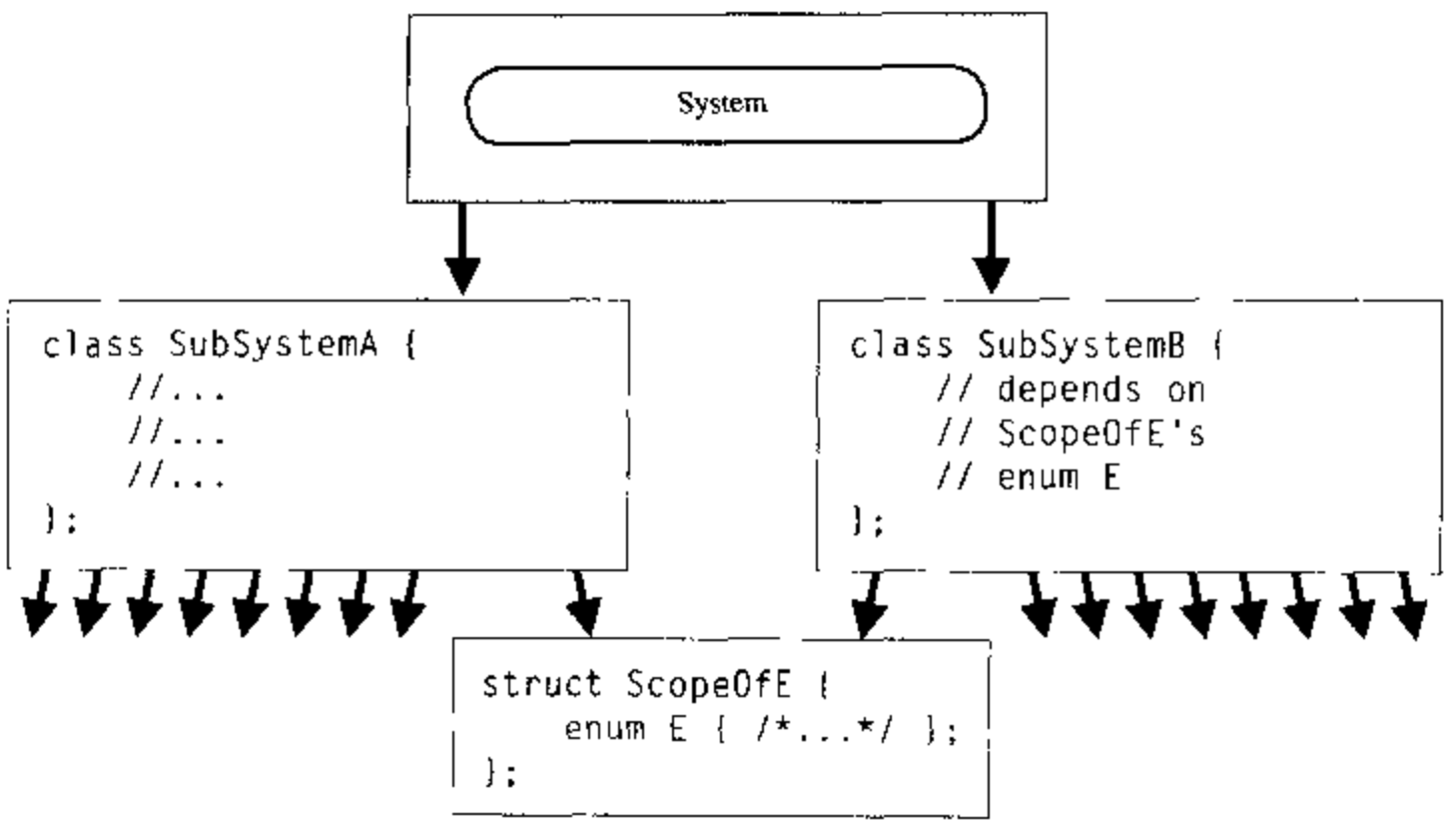


图 5-25 将枚举类型降级后的新系统结构

把单个的枚举置于它自己的类中似乎有点矫枉过正。在某些情形下是，但在这里不是。请注意，将这极少的代码放在它自己的组件中，已经把子系统 B 从不得不拖着整个子系统 A 到处跑的相当沉重的维护负担中解放出来了。

原 则

可以将升级策略与基础设施降级结合起来，以增强独立重用。

另外一个可能会有高 CCD 的常见的体系结构例子可以在这样一个系统中找到：它解析一个文本文件来建立一个运行时数据结构，然后对该数据结构进行操作，以执行一些需要的计算。

图 5-26 所示的体系结构中，在该系统层次结构底部的一个单个子系统中，解析器与运行时数据结构紧密耦合。因此，我们可能期望看到一个如下形式的成员函数：

```
RuntimeDB::Status RuntimeDB::read(const char *fileName);
```

其中 Status 是一个嵌套在类 RuntimeDB 内的枚举类型。大概这个 read 函数调用了一个解析器，用基于 fileName 指定的文件内容的信息来装载运行时数据结构。

在下一层，脱离运行时数据库进行操作的处理器被迫依赖于这个联合的解析器以及运行时数据库子系统。组件 system 相对较小，它同时管理运行时数据库的装载和处理。

虽然上面的体系结构是可层次化的，但却预示着一些有关维护和增强的潜在的严重后果。即使处理时并不需要一个解析器，一个处理器的开发也会既与解析器相耦合，又与运行时数

据库相耦合。随着系统的扩展，我们决定加入更多的处理器，在开发过程中每一个处理器都必须承担连接解析器的不必要的负担。

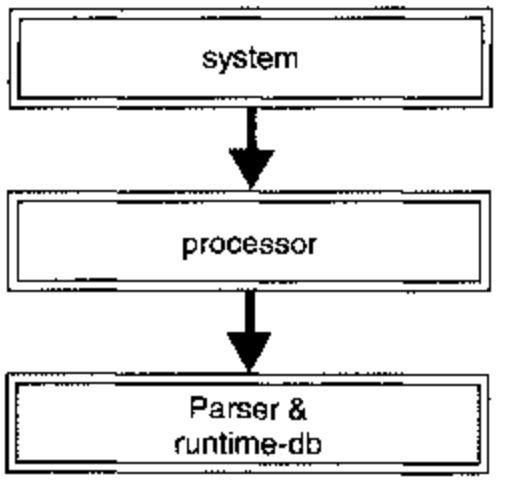


图 5-26　不良分解的运行时数据库体系结构

假设我们决定改变输入文件的格式或者（更糟糕）使用多重格式。现在，运行时数据库不再只支持单个的读命令，它必须支持若干个了：

```
RuntimeDB::Status RuntimeDB::readFormatA(const char *fileName);
RuntimeDB::Status RuntimeDB::readFormatB(const char *fileName);
RuntimeDB::Status RuntimeDB::readFormatC(const char *fileName);
```

这个结构将要求多个解析器连同运行时数据库共存于一个子系统中，如图 5-27 所示。

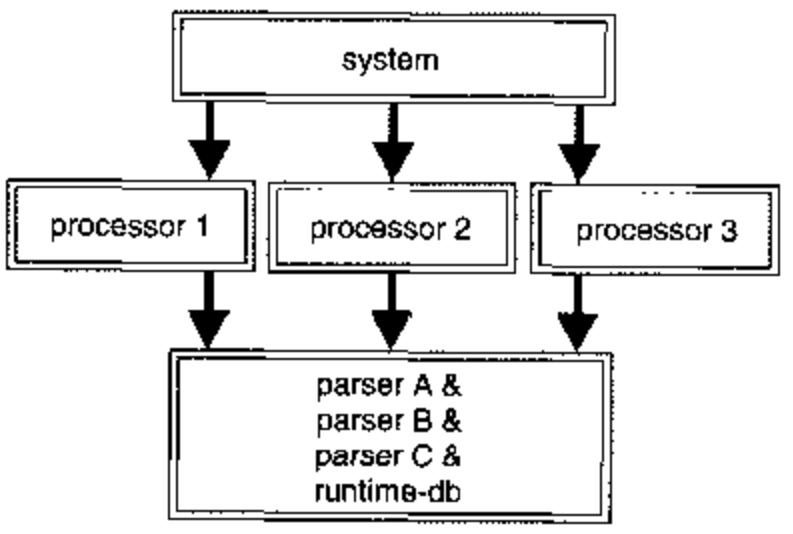


图 5-27　增强一个不良设计的结果

这个子系统体系结构的结果是，无论何时我们做以下事情都必须连接全部现有的解析器：

- 增强运行时数据库；
- 增强或开发一个新的解析器；
- 增强或开发一个新的处理器；
- 测试以上任意一项；
- 在一个处于独立状态的产品中重用运行时数据库。

在最初的体系结构中，数据库依赖解析器来装载信息。但是，进一步细查（见图 5-28）后，我们意识到在运行时数据库和解析器之间有（或应该有）一种几乎是非循环的关系。数据库是一个低层次的信息仓库，客户程序（例如解析器）存放信息进去，像处理器这样的客户又从那儿存取信息并（可能）操纵信息。每个解析器都依赖运行时数据库来存储被解析的信息。问题出在运行时数据库对一个解析器的无理由的“向上”依赖。

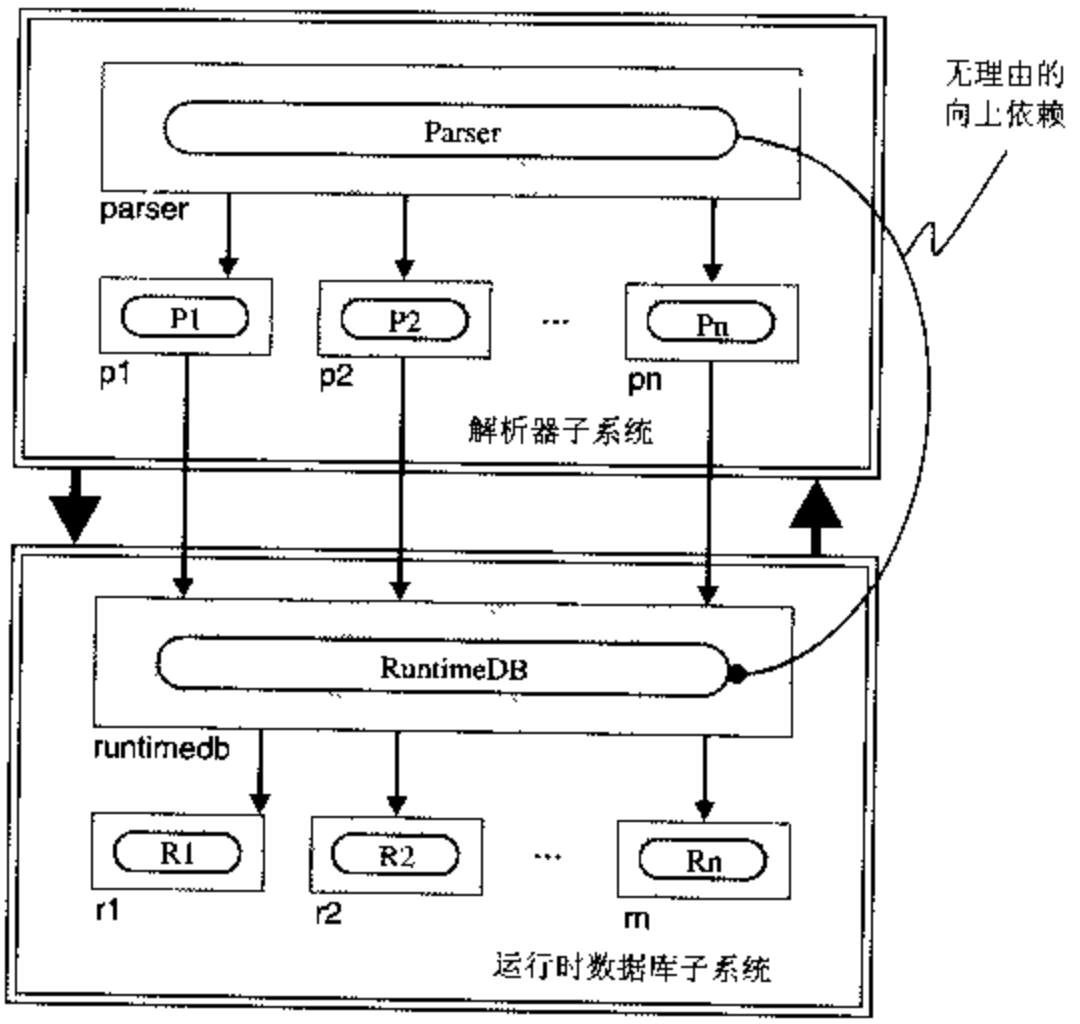


图 5-28 不良的解析器/数据库子系统体系结构的特写

升级和降级的结合为这个问题提供了一种有效的解决方案。我们可以重新构建原来的系统（见图 5-29），首先把解析器的 read 函数的调用从 RuntimeDB 升级到系统层次，然后将共有的运行时数据库子系统降级。通过把数据库变成一个“哑的”仓库（用一个过程接口来完成，该接口可以通过编程来装载和取回信息），每一个解析器就变成了数据库的“另一个客户”。现在系统只管理那些要被解析的文件，然后调用适当的解析器，传递给它一个指向要被装载的 RuntimeDB 对象的可写（非 const）指针：

```
ParserA::parse(RuntimeDB *db, const char *fileName);
```

如果后来的处理不会改变运行时数据库，而假定只是产生报告，系统可以通过给处理器传递一个只读的、到要被装载的 RuntimeDB 的（const）引用来确保数据库不被改写：

```
Processor1::quarterlyReport(ostrstream *ostr, const RuntimeDB& db);
```

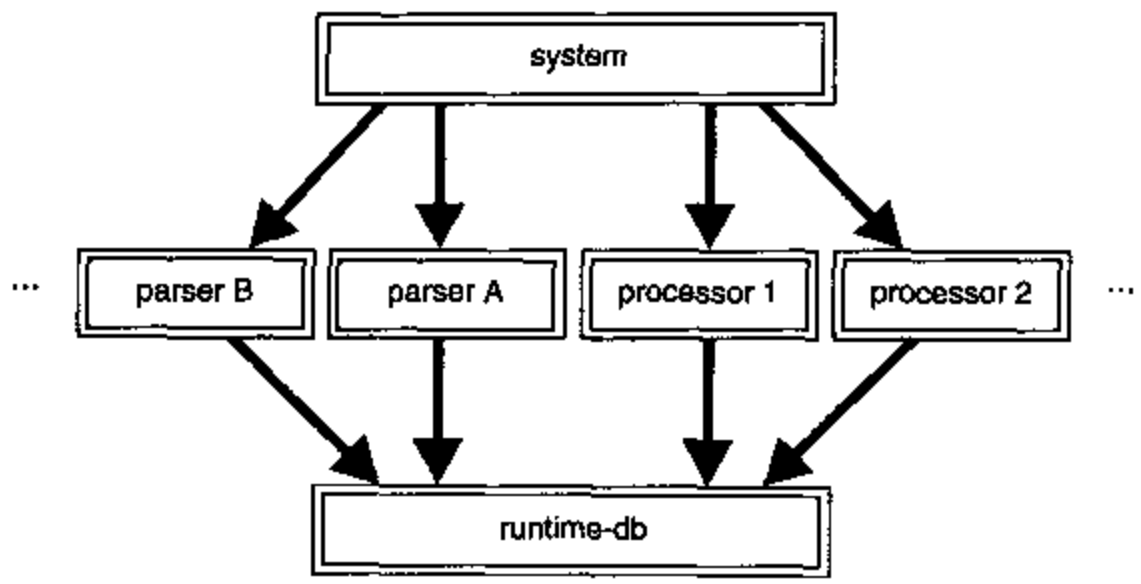


图 5-29　一个更好维护的系统体系结构

有了这个新的体系结构，任何数量的独立处理器都可以加入到系统中，它们不会依赖于任何解析器。同样地，解析器也可以被替换或增加，不会以任何方式影响运行时数据库、处理器或其他解析器。在这种体系结构下也不难想象，数据库、解析器和处理器可以在其他的处于独立状态的应用程序中，以不同的组合重用（例如：编译器、档案库存储器和浏览器）。

作为讲解降级威力的最后一个例子，考虑图 5-30 所示的子系统，其中的三个相关组件是循环依赖的。

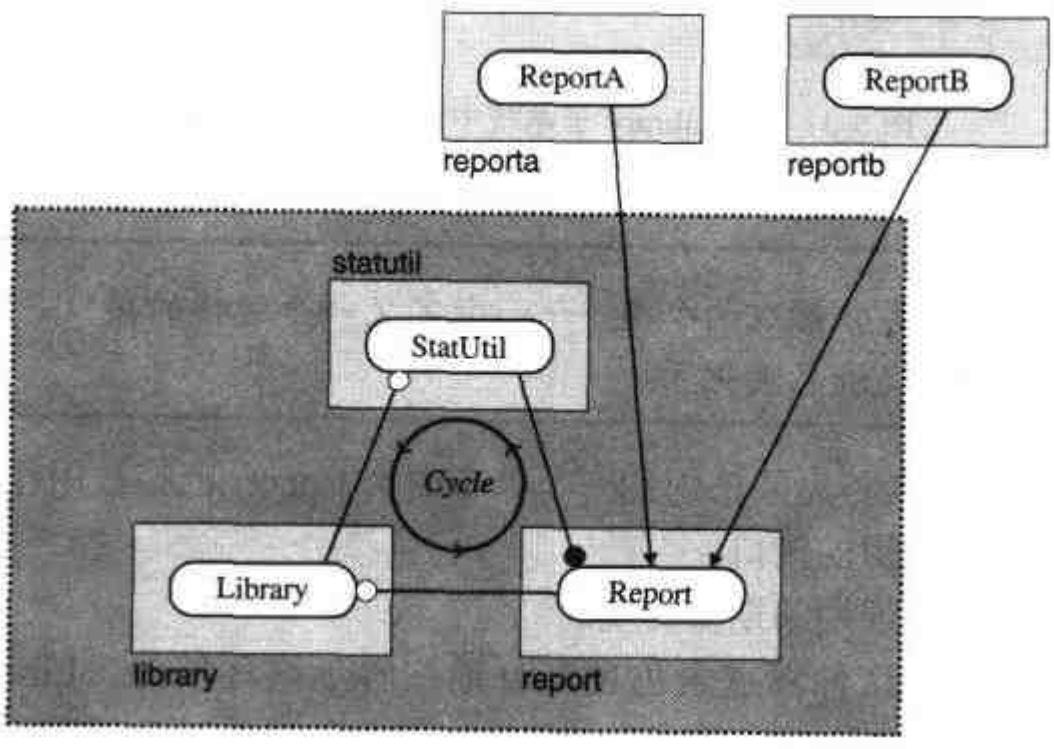


图 5-30　循环依赖的 Library 子系统

Library 包含了一个低层次信息数据库以及一个不同种类 Report 对象的集合。几乎所有种类的 Report 都提供依赖于聚集统计数字的信息，这些聚集统计数字是从存储在 Library 的低层

次数据计算出来的。一个统计工具类 Statutil 被用于帮助获得这个聚集信息[1]。在（抽象的）Report 基类中实现的共有功能使用了 StatUtil，StatUtil 因而依赖 Library，导致了 library、statutil 和 report 组件之间的循环依赖。

原　则

把一个具体的类分解为两个包含更高和更低层次功能的类可以促进层次化。

这个问题的出现部分是因为单个的 Library 类既用作低层次信息的仓库又用作（高层次）报告的集合。幸运的是有两个可供选择的解决办法。首先，通过把低层次仓库降级到子系统其余部分之下，我们可以消除循环耦合（如图 5-31 所示）。

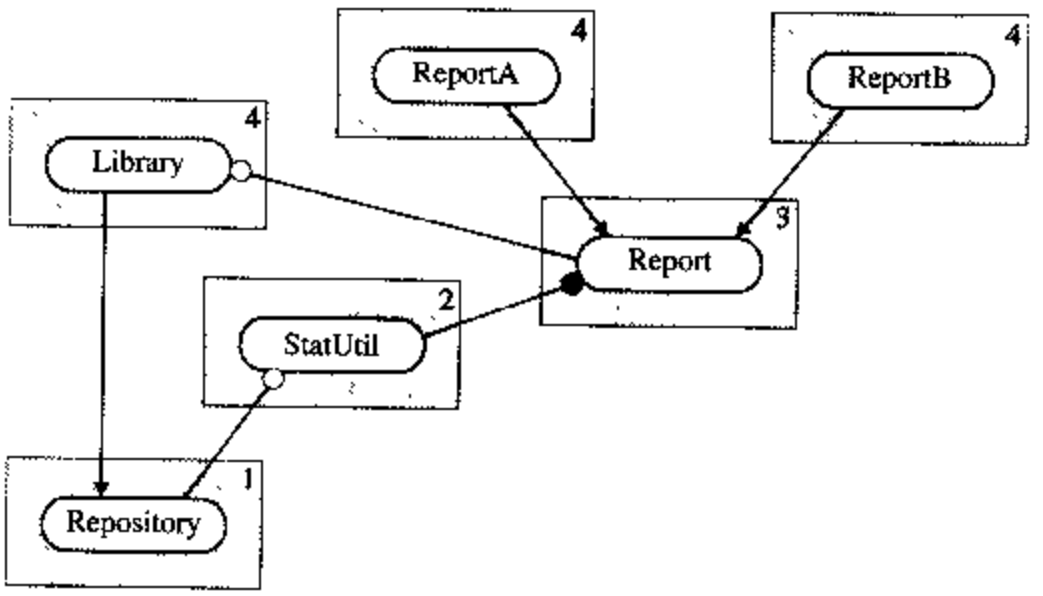


图 5-31　将 Library 子系统中的低层次信息降级

原　则

将一个抽象的基类分解成两个类——一个定义一个纯粹的接口，另一个定义它的部分的实现，可以促进层次化。

再看另一个解决办法，首先要认识到一个单个的类 Report 已经被用作两个不同的目的：

（1）提供所有报告通用的接口。

（2）提供所有报告都通用的期望实现。

让一个单个的类承担这种双重角色也部分地归咎于循环依赖。Library 直接依赖基类

---

① 通常，一个工具类要么只是一个为相关自由函数集合提供一个作用域的 struct，要么是一个模块（module）（即这个类只包含静态数据成员）。在以上任何一种情况下实例化这样一个类的实例都没有意义，因为它们不包含与一个特定实例相关的状态。见 **booch** 中的“Class Utilities”（第 5 章，186~187 页）。

Report 的接口，但只是通过使用虚函数间接依赖它的实现。

考虑一下，如果我们把 Report 分离成两个类会发生什么。第一个类将定义一个在原来的 Report 类中指定的接口，但不实现任何函数。也就是说，Report 类中的每一个函数现在都将被声明为一个纯虚[1]函数。第二个类，名为 ReportImp，将派生自 Report，通过覆盖适当的虚函数来提供类属的报告实现。

现在，通过只把定义在 Report 基类的接口降级到 Library 层次之下，就有可能解开原有系统（图 5-30）的循环依赖。实现共有功能并依赖 StatUtil 的 ReportImp 类，仍保留在物理层次结构的一个更高层次上，如图 5-32 所示。

我们考虑这些转换是升级还是降级并不重要。重要的是，我们能够用两种方法把单个的类分离成两个类，从而可以到达物理层次结构的不同层次。

哪一种解决方案更好些呢？第一种分解 Library 的方案对维护来说是理想的，因为低层次仓库和统计工具组件一样，可以独立于报告集合开发。第二种解决方案（图 5-32）对完整的未分解的 Library 施压，从而使低层次仓库和统计工具夹在接口和 Report 的部分实现之间。从这个角度来看，第一个解决方案更可取。但是还有另外的理由要求一个单个的基类应该定义在接口或者分解的实现中，但不能同时定义在这两个地方。从一个基类的（部分）实现中分离出接口，会在 6.4.1 节中详细讨论。

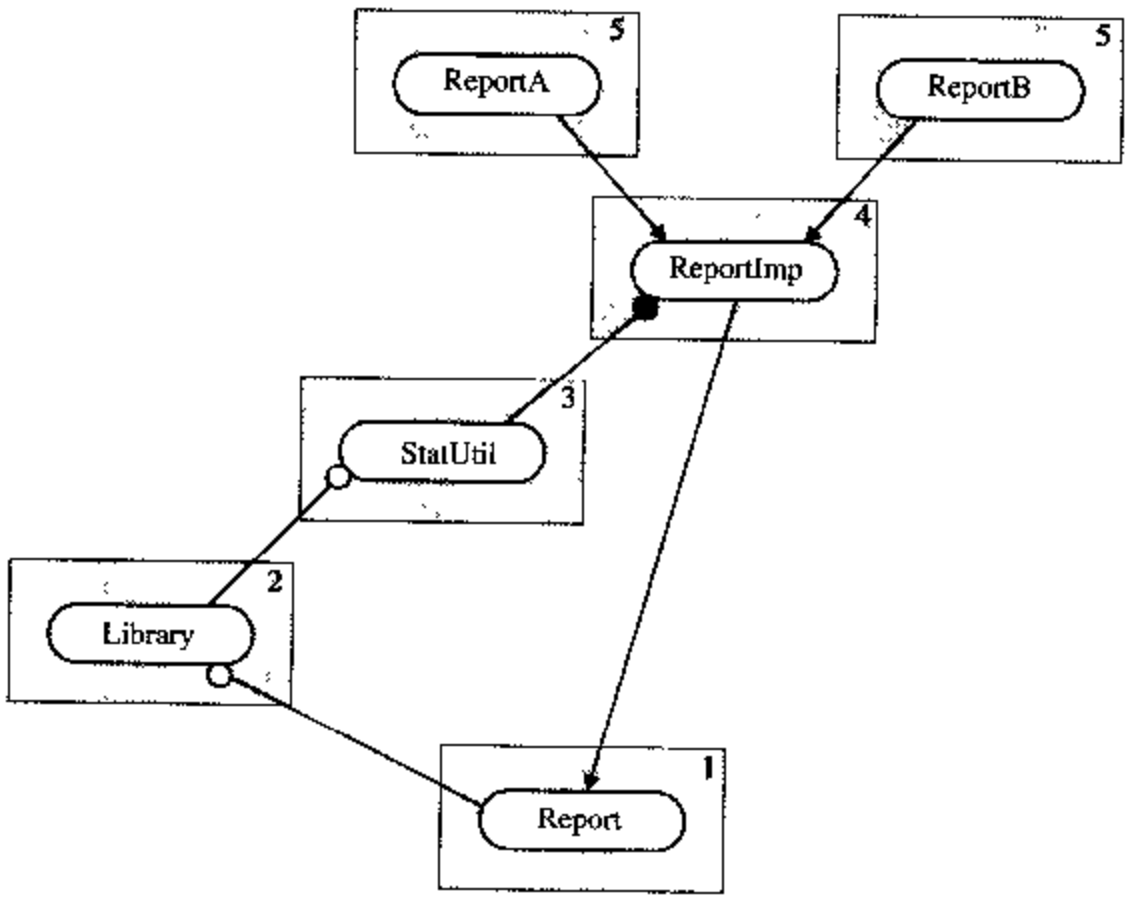


图 5-32 “只”降低 Report 基类接口的层次

① 析构函数也可以声明为虚函数，但不可以声明为纯虚函数（见 9.3.3 节）。

原　则

把一个系统分解成更小的组件，既可以使它更灵活，也可以使它更复杂，因为现在要与更多的物理部件一起工作了。

采用这两种转换后，可以产生一个更加灵活的体系结构。因为一个报告集合作为一个独立的抽象才有意义，所以可以通过允许报告集合独立于 Repository 被测试和重用来进一步改进这个体系结构。

在这个新体系结构中（图 5-33 所示），物理结构看起来比以前任何一个体系结构都更灵活。为了避免不必要的编译时耦合，我们必须把 Report 从它的部分实现中分离出来（在任何情况下）。这样做也使得我们能够创建一个很简单的、没有使用或依赖 StatUtil 的测试桩（ReportC）来测试 Report、Collection 和 Library（见图 5-34）。

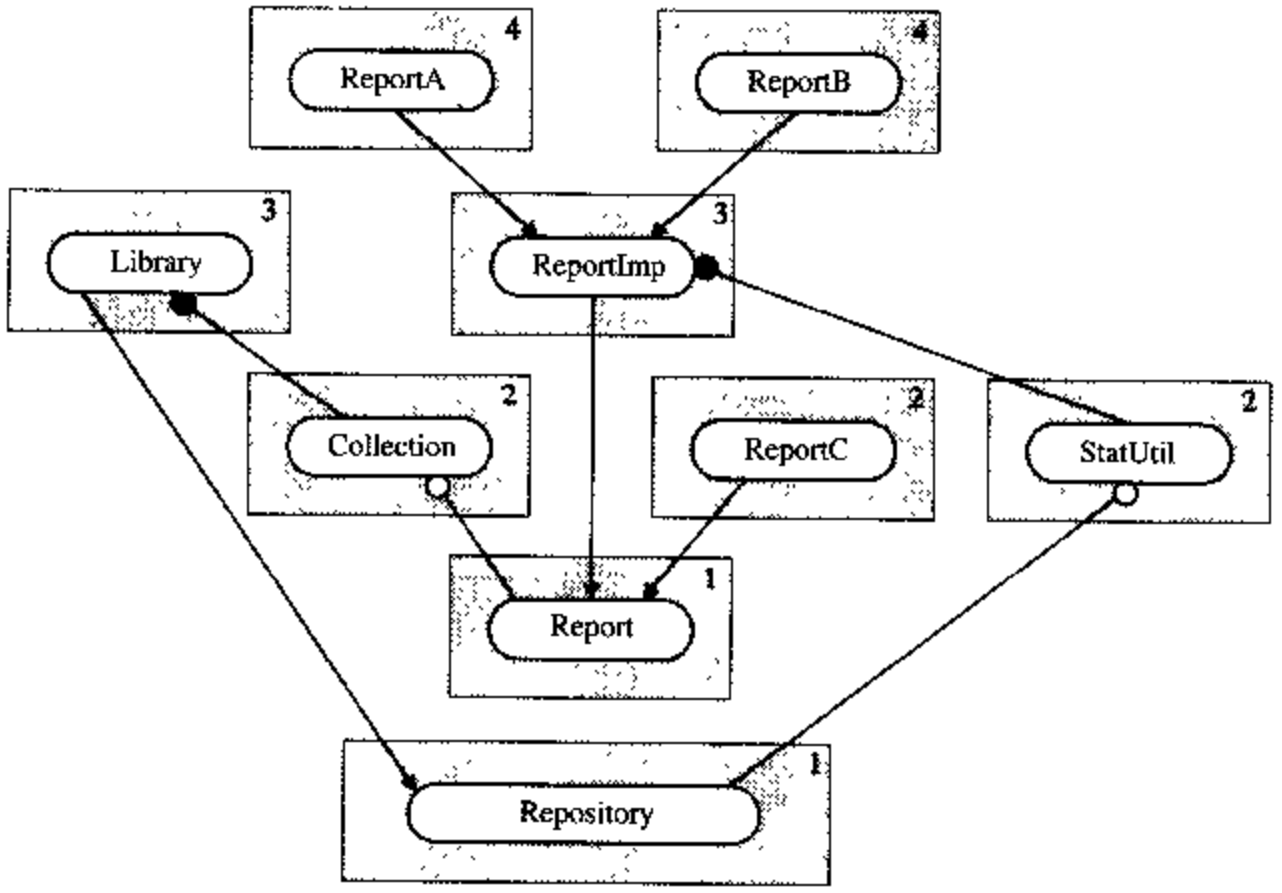
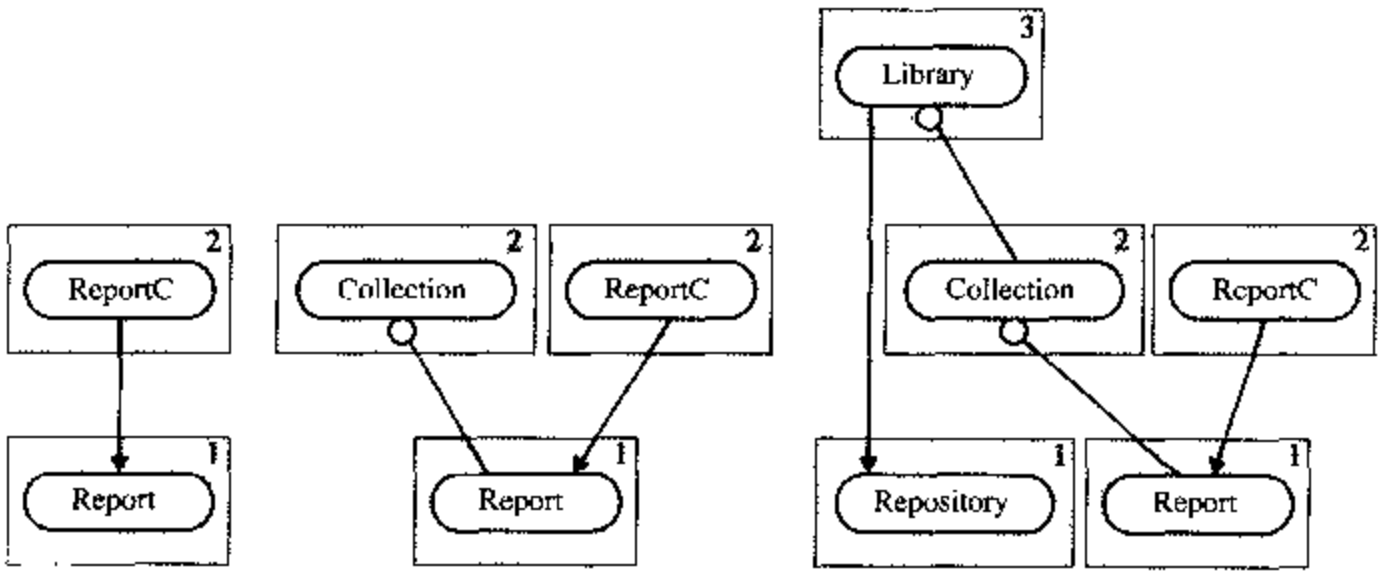


图 5-33　采用所有三个体系结构的改进

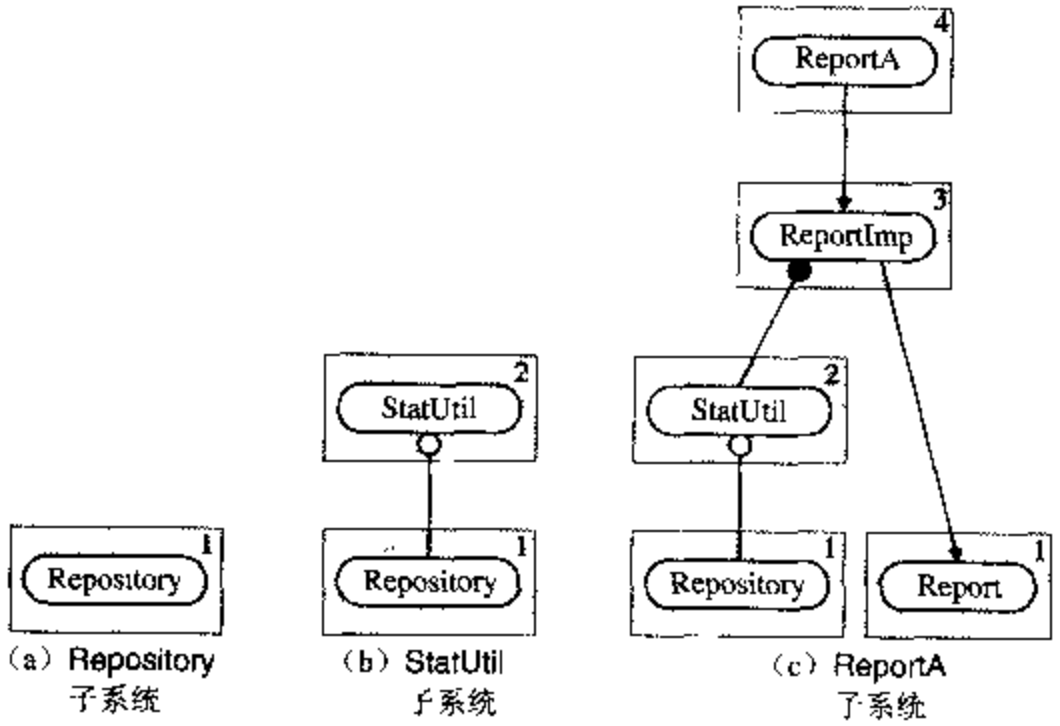
分解 library 组件是有利的，因为它进一步减少了子系统中的物理耦合。分割也特别适当，因为我们已经使 StatUtil 只依赖于 Repository，使 Collection 只依赖于 Report，所以给层次结构增加了相当大的灵活性。

降级 Repository 使其可以独立地被测试和重用[图 5-35（a）]，或者与 StatUtil 协同作用[图 5-35（b）]。单独的、复杂的报告（如，ReportA）可以被测试和重用，而不必依赖一个最后也许会也许不会保存它们的集合[见图 5-35（c）]。



（a）ReportC 子系统　（b）Collection 子系统　（c）Library 子系统

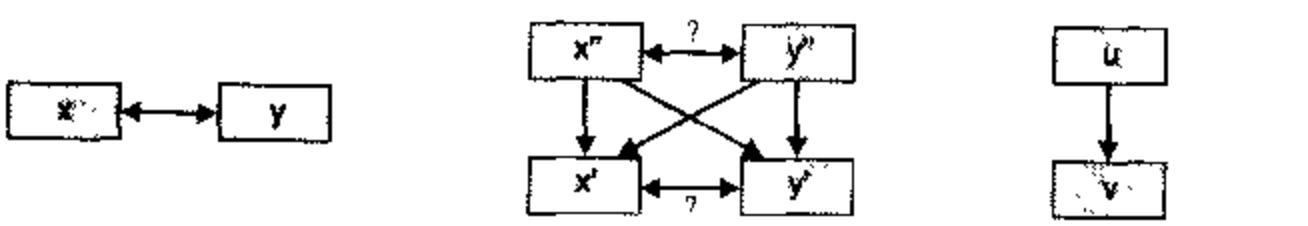
图 5-34　可独立测试和重用的子系统



（a）Repository 子系统　（b）StatUtil 子系统　（c）ReportA 子系统

图 5-35　更可以独立测试和重用的子系统

升级和降级紧密相关。升级与降级的本质区别只是功能移动的方向不同（这些被移动的令人讨厌的功能的数量相对来说较少）。事实上，升级和降级实际都只是图 5-36 中所描述的通用重打包技术的特殊情况。



（a）最初的循环子系统　（b）中间的分解子系统　（c）最后的层次化的子系统

图 5-36　通用重打包技术

在这里，两个互相依赖的组件（a）再一次分解成四个组件（b）。其中的两个组件，x'和y'，可能只是彼此依赖，而另外两个组件，x"和 y"，潜在地依赖其他三个组件的每一个。这两个分别的、也许互相依赖的组件对，现在可能组合成两个新的组件（c）。组件 u 现在依赖组件 v，而组件 v 是独立的。这种通用重打包技术被非正式地应用于本节一开始讨论的 geomutil 组件和 geomutil2 组件。

总结：我们有时可以通过分解出普遍需要的功能，并将它移到物理层次结构的更低层来消除组件之间的互相依赖。降级不仅对改进循环相互依赖设计有用，而且也对减少非循环体系结构的 CCD 有用。将共有子系统降级可同时改进可维护性和可扩展性。一个分解适当的系统会更加灵活，因为它的内部物理依赖允许它的组件以更多种类的有用方式被独立测试和重用。

5.4 不透明指针

通常，我们假设如果一个函数使用了一个 T 类型对象，那么它以一种需要知道 T 的定义的方式使用 T。也就是说，为了编译函数体，编译器需要知道它所用的对象的大小和布局。在 C++中，一个编译器获悉一个对象的大小和布局的方法就是让使用这个对象的组件包含含有该对象类定义的组件的头文件。

> **定义**：**如果编译函数 f 的函数体时要求提前看到类型 T 的定义，则称函数 f 实质（in size）使用了类型 T。**

如果一个函数体在只是看到类型 T 的声明（例如，classT;）的情况下就可以被编译，那么那个函数本身并不依赖于 T 的定义。实质使用一个类型的特点在于这样的用法会导致对定义 T 的组件的一种直接的编译时依赖。（避免不必要的编译时依赖是第 6 章的主题。）虽然函数 f 一般只在名称上使用而不是实质使用类型 T，但是如果 f 调用了其他组件中的一个或多个函数，这些函数依次地依赖 T 的定义，那么在这种情况下仍然有一个 f 对 T 的连接时依赖。

> **定义**：**如果编译函数 f 以及 f 可能依赖的任何组件时，不要求提前看到类型 T 的定义，则称函数 f 只在名称上（in name only）使用了类型 T。**

如果函数 f 和 f 依赖的所有组件在只看到了 T 的**声明**（而不是定义）的情况下就能够编译和连接，那么 f 就被认为**只在名称**上使用了 T。例如：

```
// util.h
#ifndef INCLUDED_UTIL
#define INCLUDED_UTIL

class SomeType; // used in name only

struct Util {
    SomeType *f(SomeType *obj);
}

#endif
```

```
// util.c
#include "util.h"

SomeType *Util::f(SomeType *obj)
{
    static SomeType *lastType=0;
    return obj ? lastType = obj : lastType;
}
```

说明函数 f 只在名称上使用了类型 SomeType。只在名称上使用一个类型的特点在于这样的用法没有隐含的物理依赖——即使是在连接时。没有物理依赖，耦合也就几乎全部消除了。

也可以对类建立相似的定义，即类**实质**或**只在名称上**使用了一个类型。更有用的是这些定义能够扩展应用于作为一个整体的组件。

> **定义**：如果编译组件 c 时要求必须提前看到类型 T 的定义，则称组件 c **实质使用了类型 T**。

> **定义**：如果编译组件 c 以及 c 可能依赖的任何组件时不要求提前看到类型 T 的定义，则称组件 c **只在名称上使用了类型 T**。

我们会在第 6 章使用第一个定义。现在，我们集中讨论这两个组件级定义中的第二个的定义。注意，如图 5-37 所示，组件 u 在名称上使用了一个 T 对象并且依赖于另一个组件 v，因为传递的原因，组件 u 实质使用了 T，而不仅仅只是在名称上使用 T。组件 u 物理依赖于组件 v，并且间接依赖于组件 t。

使用符号的虚线形式“○----”表示使用“只是名称上的”，并且强加了一个概念上的但不是物理上的依赖。

在这里，我们没有使用 Booch 提出的用于

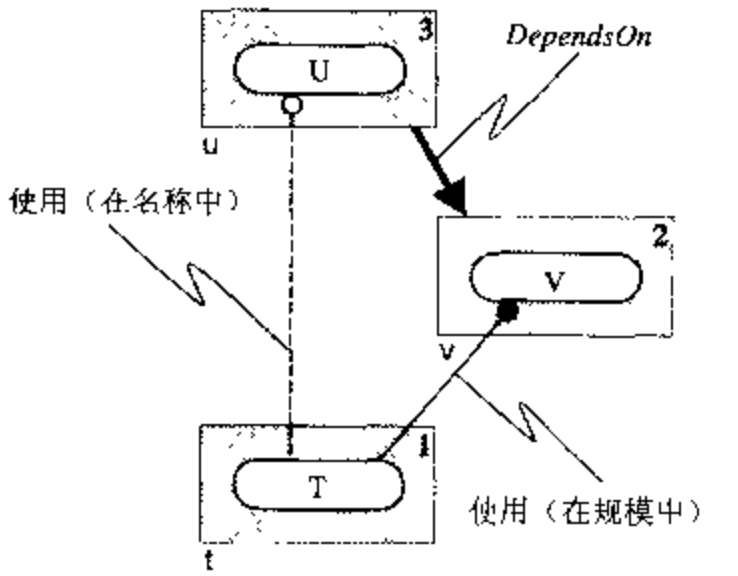
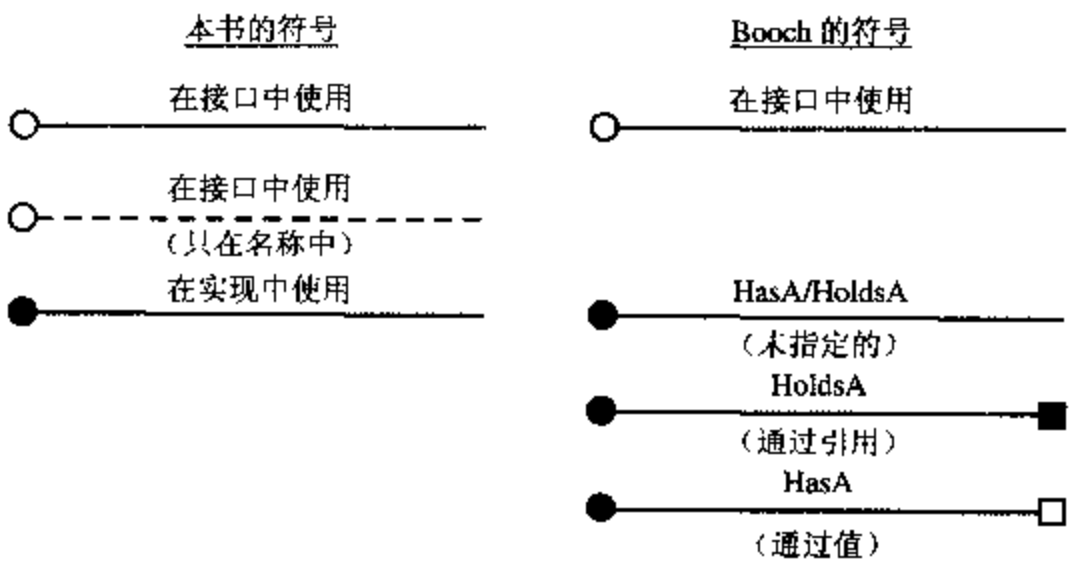


图 5-37 组件 u 不仅仅在名称上使用了类型 T

“通过引用（by-reference）”依赖的符号定义[①]，有两个原因：（1）这种新的 In-Name-Only 符号的逻辑意义和它的 In-Size 对应物是同样的——不同的只是**物理**隐含；并且（2）只在名称上使用（不像通过引用使用）清楚地否定了任何直接或间接的编译时或连接时依赖。对于我们的目标，有三种风格的使用符号就足够了：



原　则

只在名称上使用了对象的组件可以独立于被命名的对象被彻底测试。

涉及只在名称上使用一个类型的情况很少会自然出现；这样做只是为了避免不必要的物理依赖。当组件只通过指针或引用来“使用”对象，而决不以任何方式直接与对象交互（除了保存它的地址）时，只在名称上使用一个类型是可能的。

如果一个指针所指向的类型的定义不包含在当前的编译单元中，这个指针就被称为是**不透明的（opaque）**。图 5-38 显示了一个小例子，一个类拥有一个不透明指针，它指向某一名为 Foo 的类的实例。Handle 类的客户最终都将不得不包含定义了 Foo 的组件的头文件，以便产生一个 Foo 的对象。为了测试的目标，任何形式的 Foo 类都可以出现，只有一个类声明也可以，如图 5-39 所示。

```
// handle.h
#ifndef INCLUDED_HANDLE
#define INCLUDED_HANDLE

class Foo;

class Handle {
    Foo *d_opaque_p;
```

① **booch**，5.2 节，Figure 14，191 页。

```
  public:
     Handle(Foo *foo) : d_opaque_p(foo) {}
     void set(Foo *foo) { d_opaque_p = foo; }
     Foo *get() const { return d_opaque_p; }
};

#endif
```

图 5-38　只在名称上使用了 Foo 的句柄类

```
// handle.t.c
#include "handle.h"
#include <assert.h>

main()
{
    Foo *p1 = (Foo *) 0xBAD;
    Foo *p2 = (Foo *) 0xB0B;
    Handle handle(p1);
    assert(p1 == handle.get());
    h.set(p2);
    assert(p2 == handle.get());
}
```

图 5-39　handle 组件的微小测试驱动程序

这个例子的重要性在于说明了，在不包含或连接任何定义了 Foo 类的组件的情况下可以完全试运行 Handle 类的功能。这是一种决定性测试——测试是否有另外一个类型不仅被不透明地使用了，而且是只在名称上使用。

在开发一个具体的应用程序时，一个较高层次的对象常常会将信息存储于定义在物理层次结构的较低层次的对象中。如果信息以一种用户自定义类型的形式存在，就有可能导致下级对象依赖那个类型。只要这个下级不需要主动地对那个类型进行任何实质的使用，这个下级就没有必要包含该类型的定义。

原　则

如果一个被包含的对象拥有一个指向它的容器的指针，并且要实现那些实质地依赖那个容器的功能，那么我们可以通过以下方法来消除相互依赖：（1）让被包含类中的指针不透明；（2）在被包含类的公共接口上提供对容器指针的访问；（3）将被包含类的受影响的方法升级为容器类的静态成员。

假设 Screen 是 Widget 对象的容器，并且进一步假设每个 Widget 都拥有一个指针 d_parent_p，标识这个 Widget 属于的 Screen。现在考虑图 5-40 中给出的 widget 和 screen 组件

的接口，尤其是 Widget 类的访问成员函数 numberOfWidgetsInParentScreen。

这个函数允许一个只拥有一个 Widget 的客户程序找出这个 Widget 所属的 Screen 有多少其他的 Widget 对象。从纯粹易用性的角度来看，这个体系结构似乎是吸引人的；从维护角度来看，它是昂贵的。

```
// screen.h
#ifndef INCLUDED_SCREEN
#define INCLUDED_SCREEN

class Widget;

class Screen {
    Widget *d_widgets_p;
    // ...
  public:
    Screen();
    // ...
    void addWidget(const Widget& w);
    // ...
    int numWidgets() const;
    // ...
};

#endif
```

（a）容器组件 screen

```
// widget.h
#ifndef INCLUDED_WIDGET
#define INCLUDED_WIDGET

class Screen;

class Widget {
    Screen *d_parent_p; // Screen to which
    // ...              // this widget belongs
  public:
    Widget(Screen *screen);
    // ...

    // operations involving parent screen

    int numberOfWidgetsInParentScreen() const;

    // ...
};

#endif
```

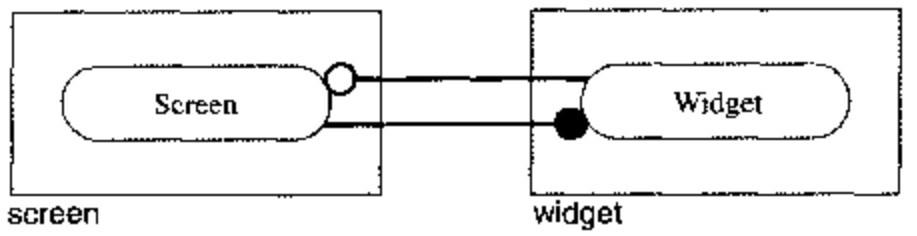
（b）被包含的组件 widget

图 5-40 Screen/Widget 设计导致循环依赖

这个设计的维护问题在于，为了实现 widget.c 文件中的 numberOfWidgetsInParentScreen 方法，我们必须“请求”父 Screen 来取得这个信息。请求 Screen 任何事情都意味着已经看到了它的定义，这要通过首先把 screen.h 包含在 widget.c 中来实现。但是这样做导致了图 5-41 中描述的不可层次化的情形。

这里的基本问题是 Widget 试图做得比它应该做的更多。一个 Widget 提供在它自己的上下文中有意义的功能，但它通常不可能不询问它的父 Screen 就知道其他的 Widget 对象。再考虑一个公司的类似情况，你可以问任何雇员“你正在干什么？”，而雇员应该能够告诉你。同样地，你也可以总是问雇员“谁是你的老板？”。但试着问问雇员为他（或她）的老板工作的雇员的数量，通常雇员不会知道答案，并且需要去老板那儿问。

实际上，了解有多少雇员为老板工作不是雇员的职责。考虑一个可替代的方法。假设你想知道有多少雇员为我的老板工作。不要问我这个问题，而是问我“谁是你的老板？”，我会告诉你，然后你可以自己去问她有多少雇员为她工作。如果她想告诉你，她就会告诉你。




```
// widget.c
#include "widget.h"
#include "screen.h"
// ...
int Widget::numberOfWidgetsInParentScreen() const
{
    return d_parent_p->numWidgets();
}
// ...
```

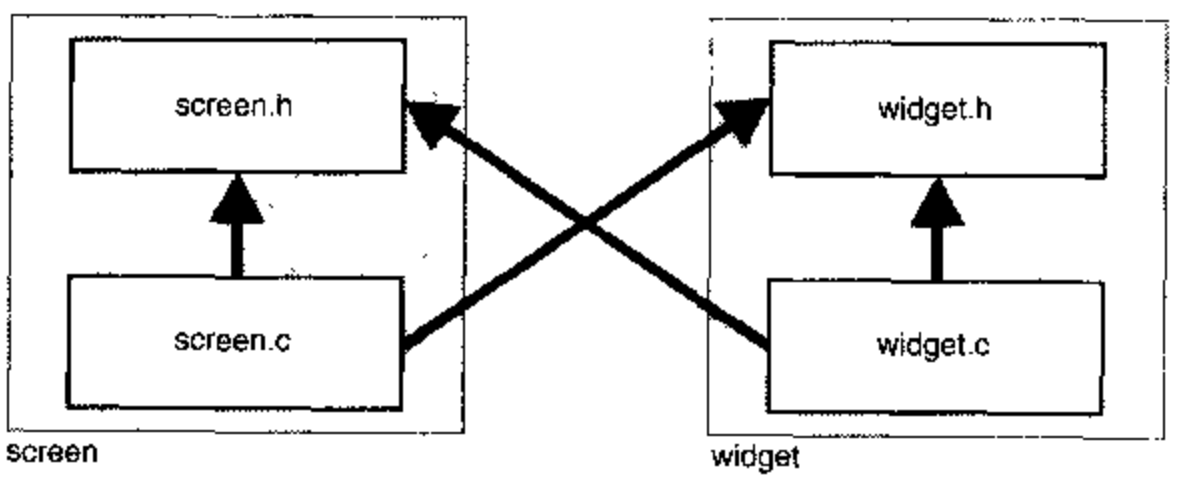


图 5-41　类 Widget“知道”容器类 Screen

使用不透明指针（只在名称上使用）可以用来打破不需要的循环组件依赖。返回到我们的设计实例，考虑图 5-42 所示的 widget 组件的可替代的定义。在这种使用模式中，可以问 Widget 来获得它的父 Screen。然后我们就能够问这个父 Screen 有关它的其他的 Widget 对象的事情（或有关那个问题的其他任何事情）。但是，这个模式的主要益处在于，组件 widget 在编译时或连接时都不再依赖组件 screen。此时 widget 对 screen 的依赖只是名称上的了。

组件 screen 和 widget 的新的组件依赖图显示在图 5-43 中。有了这个新的体系结构，就有可能独立于 screen 组件测试 widget 的所有功能。其他使用了 widget 但不关心 screen 的组件不需要包含 screen.h 或连接到 screen.o。

一个说明 widget 对 screen 的物理依赖的小型测试驱动程序如图 5-44 所示。

这个体系结构变化的一个直接结果是，客户程序必须执行两个操作（不再是一个操作）来获取父 Screen 中的 Widget 对象的数量：

```
widget.parentScreen()->numWidgets()
```

```
// widget.h
#ifndef INCLUDED_WIDGET
#define INCLUDED_WIDGET

class Screen;

class Widget {
    Screen *d_parent_p; // screen to which this widget belongs
    // ...
  public:
    Widget(Screen *screen);
    // ...

    // operations involving parent screen

    Screen *parentScreen() const;
};

#endif
```

```
// widget.c
#include "widget.h"
#include "screen.h" // no longer needed

// ...

Screen *Widget::parentScreen() const
{
    return d_parent_p;
}
```

图 5-42 组件 widget 的修改过的体系结构

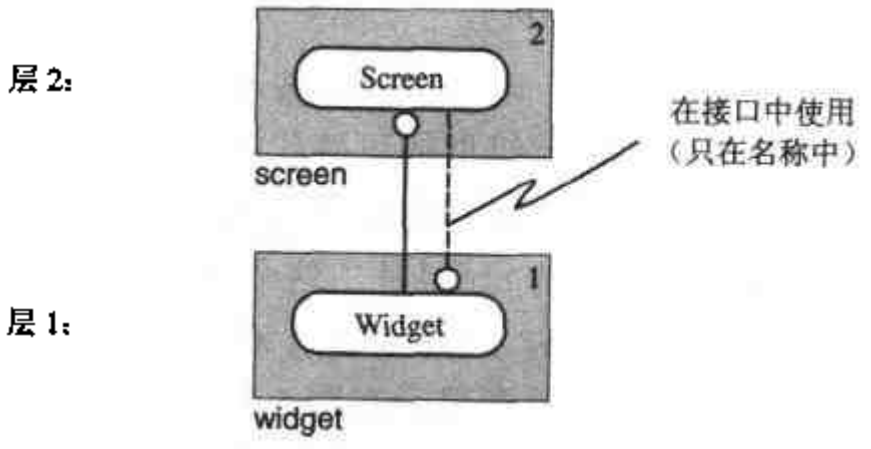


图 5-43 widget 和 screen 的可层次化的组件依赖

```
// widget.t.c
#include "widget.h"
#include <iostream.h>

class Screen;     // not necessary when including widget.h

main()
```

```
{
    Screen *const screen = (Screen *) 0xbad;

    const Widget widget(screen);

    if (screen != widget.parentScreen()) {
        cout << "Error!" << endl;
    }

    // ...
}
```

图 5-44 widget 组件的隔离测试驱动程序

为了方便，这两个操作可以合并成 Screen 的一个静态成员函数或其他的较高层次的类。不再用

```
widget.numberOfWidgetsInParentScreen()
```

而是要用

```
Screen::numberOfWidgetsInParentScreen(widget)
```

来获得这个值。在两种情况下，接口都强迫客户必须在 widget 组件的接口之外寻找才能获得他们所提问题的答案。

注意，当把功能从被包含对象移到容器时，新静态成员的第一个参数必须要么是一个 const 引用，要么是一个指向被包含对象的非 const 指针——取决于原来的成员是一个 const 还是一个非 const 函数。这种参数传递风格的基本原理将在 9.1.11 节继续讨论。

扼要重述：通过使 Widget 类的内部 Screen 指针不透明，以及把 Widget 中实质使用了 Screen 类的那部分移出 Widget，移入 Screen 类本身，我们能够获得一张非循环的组件依赖图。我们也在 Widget 的公共接口上暴露了 Screen 类型，并且使 widget 的客户必须在那个组件之外寻找的问题答案。但在这样做的时候，相互的物理依赖被概念上的协作所取代：较低层次的对象只同意保持在较高层次上使用的信息（只在名称上指定）。

5.5 哑数据

术语**哑数据（dumb data）**是对不透明指针概念的一种概括。哑数据是一个对象拥有但不知道如何解释的任何种类的信息。这样的数据必须用在另一个对象的上下文中，通常用在一个更高的层次上。

让我们考虑通过实现一个简化的子系统来建造一个赛马跑道模型时可能会涉及些什么。作为一个起点，我们期望能够问一些诸如图 5-45 所示的问题。顶层的组件应该提供在赛马比

赛（races）上进行迭代和通过名称来识别一匹马的能力。为了使这个例子更有趣，一个跑道（track）也应该可以接受赌注（bet）和赎回赌注（wager）。

```
void questions(const Track& track,
               const Race&  race,
               const Horse& horse)
{
    // 1. What races do you run here?
    for (RaceIter it1(track); it1; ++it1) {
        cout << it1().number() << endl;
    }

    // 2. What time does a given race start?
    cout << race.postTime() << endl;

    // 3. What horses are running in a given race?
    for (HorseIter it3(race); it3; ++it3) {
        cout << it3().name() << endl;
    }

    // 4. What is the number of this horse?
    cout << horse.number() << endl;
}
```

图 5-45 对一个赛马跑道会问的一些常见问题

顶层 track 组件的最初片段在图 5-46 中给出。在这个体系结构中，一个 Track 拥有 Race 对象的一个集合，并提供一个 RaceIter 来对这个 track 上今天的赛跑比赛进行迭代。Track 接受赌注（bets）并且发（指针给）给 Wager 对象，Wager 可以在赛跑比赛结束后被赎回。

```
// track.h
#ifndef INCLUDED_TRACK
#define INCLUDED_TRACK

class Horse;
class Race;
class RaceIter;
class Track;

class Wager {
    const Horse& d_horse;
    double d_amount;
    // ...
    Wager(const Horse& horse, double amount);  // For track's use only
    Wager(const Wager&);                       // -- i.e., not for use
    Wager& operator=(const Wager&);            // by the public.
    friend Track;
```

```
  public:
    const char *horseName() const;
    int raceNumber() const;
    Track& track() const;
    double amount() const;
};

class Track {
    Race *n_races_p;
    // ...
    friend RaceIter;

  public:
    // ...
    const Race *lookupRace(int raceNumber) const;
    const Horse *lookupHorse(const char *horseName) const;
    Wager *bet(const Horse& horse, double wagerAmount);
    double redeem(Wager *bet) const;
};

class RaceIter {
    // ...
  public:
    RaceIter(const Track& track);
    void operator++();
    operator const void *() const;
    const Race& operator()() const;
};

#endif
```

图 5-46　顶层组件 track 的头

每个 Race 对象保持着该场比赛的号码、起始时间以及正在比赛的马的集合。组件 race 也提供一个 HorseIter 来迭代在一场指定 Race 中赛跑的马。假如有一个 Race 对象，就有可能确定比赛将在哪一个跑道上进行。组件 race 的一个粗略版本显示在图 5-47 中。

```
// race.h
#ifndef INCLUDED_RACE
#define INCLUDED_RACE

class HorseIter;

class Race {
    // ...
    friend HorseIter;

public:
  Race(const Track& track, int raceNumber, double postTime);
  // ...
```

```
    int number() const;
    double postTime() const;
    const Track *track() const;
};

class HorseIter {
    // ...
  public:
    HorseIter(const Race& race);
    void operator++();
    operator const void *() const;
    const Horse& operator()() const;
};

#endif
```

图 5-47 中间层组件 race 的头

Horse 定义在赛马跑道子系统的物理层次结构的最底层。一个 Horse 保持它的名字和号码，可用来确定它被安排在哪一场比赛中跑。处在我们的叶子层的 horse 组件的开始片段如图 5-48 所示。

```
#ifndef INCLUDED_HORSE
#define INCLUDED_HORSE

class Race;

class Horse {
    const Race& d_race;
    char *d_name_p;
    int *d_number;
    // ...
  public:
    Horse(const Race& race, const char *HorseName, int horseNumber);
    // ...
    const char *name() const;
    int number() const;
    const Race *race() const;
};

#endif
```

图 5-48 叶子层组件 horse 的头

在这个最初的实现中，一个 Wager 只用了两个数据成员来实现，如下所示：

```
class Wager {
    const Horse& d_horse;
    double d_amount;
```

```
    // ...
  public:
    // ...
};
```

若拥有一个指向一个 Horse 的指针，在足够高层次的客户就有可能使用被惟一识别的 Horse 来获得一个指针，指向那匹马将要参加的 Race（无论在世界的何处）。然后可以遍历该 Race 指针，并且作为结果的 race 对象通常会获得一个指针，指向那个将要举行比赛的跑道。

上面所描述的赛马跑道系统的功能意味着存在一种循环内部数据结构：每个 Track 知道它所举行的比赛，每个 Horse 知道它要参加哪一场比赛，每个 Race 既知道它将在哪个跑道举行也知道将要参加本次比赛的马匹。但是，这种数据结构可以通过使用不透明指针以非循环物理依赖的形式来实现，如图 5-49 中的组件/类图所示。

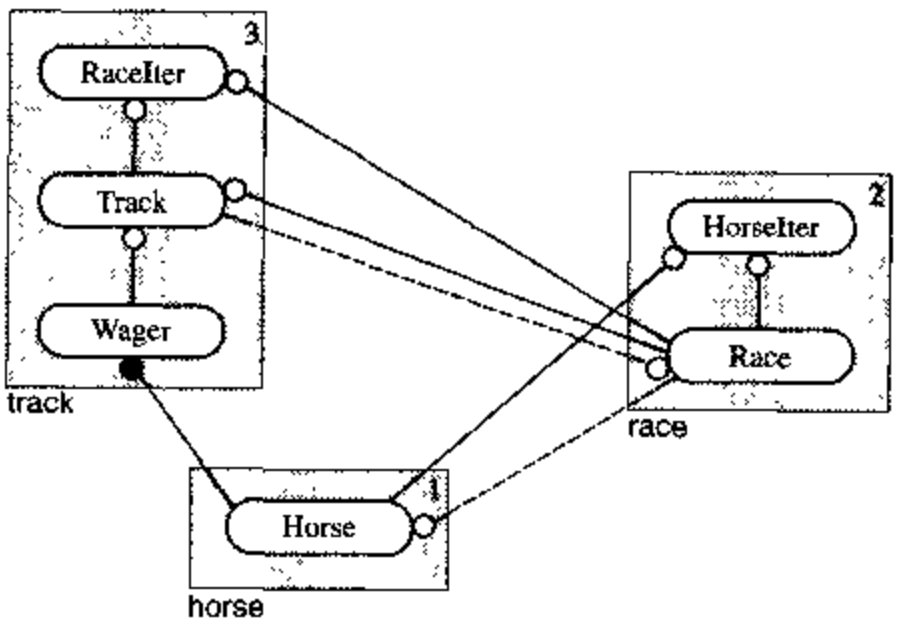


图 5-49 赛马跑道子系统的组件/类图

最初为赛马跑道子系统设计的体系结构没有循环物理依赖，但在名称上却是循环依赖的。虽然有两个组件知道一个或多个定义在其他组件中的对象的名称不一定是坏事，但需作一些权衡，我们对此略作讨论。

假设我们识别某个系统的对象不是通过它们的绝对地址，而是根据对象的索引顺序表，则这些对象只在父对象的上下文中才有意义。

于是 Track 将拥有一个 Race 对象的序列（数组），并且每个 Race 都将有一个关联整数“index”。Race 索引只在一个 Track 对象的上下文中有意义。因为 Race 索引可以相应于公共可访问的 Race 编号，倘若我们为 Track 提供一个访问函数来报告今天举行的比赛的总数，对 RaceIter 的需求就减少了。

通过同样的参数，一个 Race 中的每个 Horse 都被自然地赋值一个号码。假如一个 Race 有一个马的序列，我们可以通过提供马相对于那场比赛的索引来识别一个 Race 内的 Horse。

因此我们也可以为 Race 免除 HorseIter。

当要赎回赌注时，Track 定义了一个比整个地址空间小得多的上下文（可通过指针访问）。在最初的实现中，我们使用了不透明后退指针（back pointers），以 Horse 开始，以一种自下而上的方式移动到达 Race，最后到 Track。在这个推荐的实现中，Track 的受限制的上下文被用来识别 Race 和 Horse，借助于一对整数下标来实现，如图 5-50 所示。

```
class Wager {
    const Track& d_track;
    double d_amount;
    short int d_raceIndex;
    short int d_horseIndex;
    // ...
  public:
    Wager(const Track& track,
          int horseNumber,
          int raceNumber,
          double amount);
    const Track& track() const;
    double amount() const;
    int horseNumber() const;
    int raceNumber() const;
    // ...
};
class Track {
    Race *d_races_p;
    int d_numRaces;
    // ...
  public:
    Wager *bet(int race, int horse, double amount);
    double redeem(Wager *bet) const;
    const Race *lookupRace(int raceNumber) const;
    const Horse *lookupHorse(const char *horseName) const;
    const Horse *lookupHorse(const Race& race, int horseNumber) const;
    int numRaces() const;
    // ...
};
```

图 5-50 对组件 track 的修改

观察一下，因为这个非常受限制的上下文，我们可以安全地使用 16 位整数而不是 32 位整数。如果赌注的数量在任何一个时候变得非常巨大，这个事实将是很有意义的。例如，在我的 32 位机器上，一个双精度型是 8 字节长且是自然排列的[①]，当我们把索引变成 short 整数时，wager 对象的大小从 24 字节降到 16 字节——节约了 33%！

① 自然排列在 10.1.1 节中讨论。

图 5-51 展示了赛马跑道子系统的修改过的体系结构。新的系统明显地更简单了。这个系统没有组件间的循环依赖——即使是名称上的——而且明显有更少的类。主要的变化只是识别 Horse 的方式。

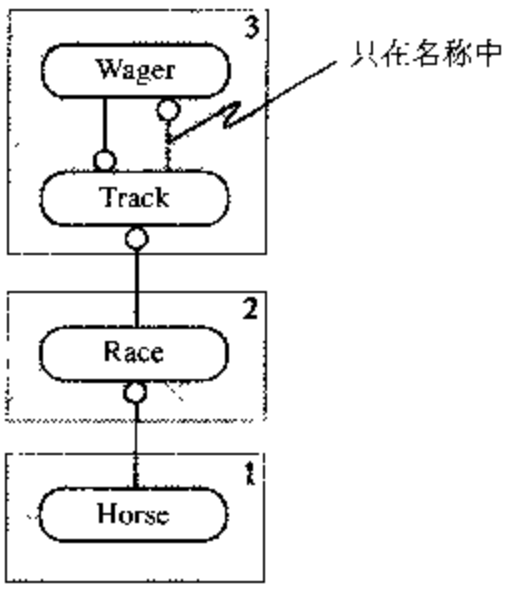


图 5-51 赛马跑道子系统的修改过的组件/类图

哑数据可能比用于识别其他对象的不透明指针更方便并且偶尔会更简洁。如果新的 Wager 对象是通过不透明指针而不是 short 整数索引来识别 Race 和 Horse，那么在我的机器上 Wager 的大小将又是 24 字节而不是 16 字节。

另一个好处是存储成哑数据的值不是机器地址，因而包含了可以被显式测试的有意义的值。在这个赛马应用程序中，索引的方法尤其吸引人，因为索引（是公共可访问的）确实在用户领域有着合理的效用。在下注窗口听到一个经常出入赛马场的跑道赞助人要求一个赌金代理“给我 2 美元赌 4 号在第 9 场（赢）!”并不少见。

这种索引方法的一个缺点是，与不透明指针相比，它确实牺牲了相当程度的类型安全，因为 Race 和 Horse 索引只是整数。另一个缺点是这种实现强迫 Race 和 Horse 集合被编入索引而不是保持任意的集合。最后对封装造成的侵蚀如果暴露给一般的公众，可能很容易对可维护性产生消极影响。

在我们的赛马实例以外的情形中，用来以这种方式识别了对象的哑数据索引也许对子系统的客户是毫无意义的。因此，哑数据的使用是一种典型的封装在一个子系统内的优化实现技术，它不会在一个系统的更高层次暴露。

原 则

哑数据可以用来打破 **in-name-only** 依赖，促进易测试性和减少实现的大小。但是，不透明指针可以同时保持类型安全和封装；而哑数据通常是不能的。

作为一个类似但更严肃的例子，我们来考虑这样一个任务：为一个电路组件内部的连通

件建立模型，该电路由电气组件的一个异类集合组成[1]。一个门级电路，例如在 4.7 节介绍层次化时引入的图 4-10 中的那一个，可以描述为一幅由节点[称为 **gate**（门）]和边[称为 **wire**（电线）]组成的图。每个 gate 都有一个电气上不同的连接点[称为 **terminal**（接头）]的集合。在概念上，表现一个电路就是要维护门的一个异类集合和双向电线的一个同类集合。每一根电线都被接在电路内的两个不同的接头上，建立连通性。

在传统的实现中，一个电路可能包含接头的一个集合，用于定义电路的主要输入输出。电路中的每一个门也将包含接头的一个集合。在这个模型中电线不是明确的对象。但是，每个接头都包含一个指向（其他）接头的指针的集合。指向另一个接头隐含建立了一个到该接头的连接。在图 5-52 所示的例子中，门 g0 的接头 z 被连接到门 g1 的接头 x 上；因此，g0 的接头 z 将拥有一个指向 g1 的接头 x 的指针。对称地，g1 的接头 x 也连接到 g0 的接头 z 上，这样 g1 的 x 也将拥有一个指向 g0 的 z 的指针。

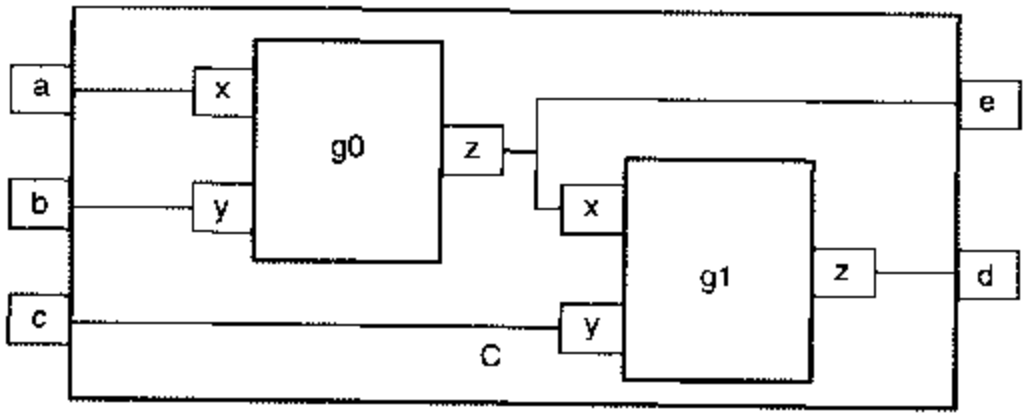


图 5-52 按两个门（g0 和 g1）来实现的电路 C

为了遍历图，一个 Terminal 必须保持一个指向它的父 Gate 或 Circuit 的不透明指针。注意，Circuit 可以看作是一个特殊种类的 Gate，它包含了其他门的实例[2]。循环物理组件依赖可以通过使用不透明指针来避免，如同在图 5-53 的部分组件/类图中显示的那样（集合迭代器被省略）。

这里又有一个机会允许我们通过“在上下文中”定义一个连接来打破即使是名称上的循环依赖。如果一个 Circuit 包含 gate 的一个索引的集合（一个数组），并且每个 Gate 都同样地包含一个接头的数组，那么我们可以在一个 Circuit 的上下文中将一个连接点标识为一对简单的整数下标。

---

① 这个例子描述了哑数据在一个很不一样的上下文中的应用。但是基本技术和用在赛马跑道例子中的技术是相同的。

② 这个例子解释了递归复合（recursive composition）——一种称为复合（*Composite*）*的设计模式*（**gamma**，第 4 章，163~173 页）——的另一个实例。这种模式在前面介绍过（在 4.7 节的图 4-13 中根据 Node、File 和 Directory 阐释过）。这种复合设计模式已被有效地用于实现分层次的电路描述。

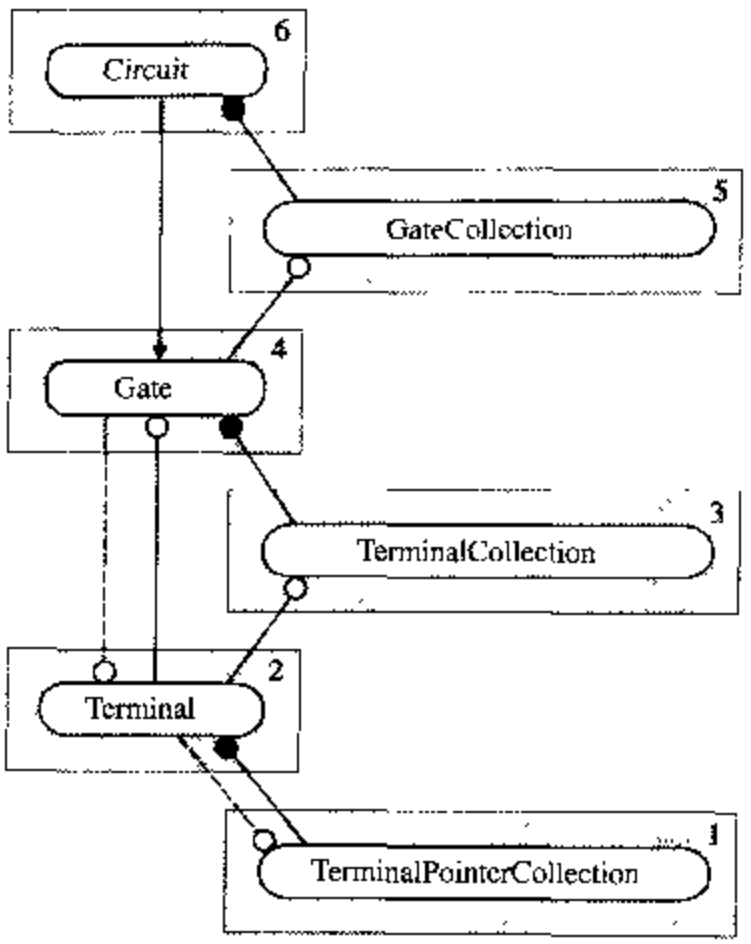


图 5-53 Circuit 实现的部分组件/类图

再考虑图 5-52 中的例子。假设 circuit 的实现由一个两个门（g0 和 g1，分别带着下标 0 和 1）的数组组成。g0 和 g1 的接头都是 x、y 和 z，并且碰巧分别有下标 0、1 和 2。我们现在可以把连接点“门 g1 的接头 x”描述为一对整数下标（1，0）。同样地，我们也可以把连接点 g0 的 z 描述为坐标对（0，2）。

按照惯例，我们可以通过为门的索引使用一个合法范围之外的索引（例如–1）来标识封闭的电路。如果电路的接头 a 有下标 0，它的连接坐标可以表示为下标对（–1，0）。这个电路的连接的完整列表以整数坐标描述，如图 5-54 所示。

```
C.a = (-1, 0) <-------> ( 0, 0) = g0.x
C.b = (-1, 1) <-------> ( 0, 1) = g0.y
C.c = (-1, 2) <-------> ( 1, 1) = g1.y
g0.z = ( 0, 2) <-------> ( 1, 0) = g1.x
g0.z = ( 0, 2) <-------> (-1, 4) =  C.e
g1.z = ( 1, 2) <-------> (-1, 3) =  C.d
```

图 5-54 用整数下标表示连通性

一对整数不存在对任何东西的物理依赖。因此我们可以像下面这样在一个叶子组件中定义一个 Connection 类：

```
class Connectior {
    int d_gateIndex;
```

```
    int d_terminalIndex;

  public:
    Connection(int gateIndex, int instanceIndex);
    int gateIndex() const;
    int terminalIndex() const;
};
```

图 5-55 描绘了一个完全可层次化的组件层次结构，其中的 Connection、ConnectionCollection、Terminal、TerminalArray、Gate、GateArray 以及最后的 Circuit 都可以被顺序地测试和校验。

Circuit 的图型特性从子组件看并不明显。一直要到定义了 GateArray 类的组件的那一层，电路的连通性才建立，因为只有在那一层才存在足够的上下文来理解隐含的图。Circuit 的用户没有必要暴露给较低层次的 Gate 和 Terminal 类，并且他们可以通过被在内部转换成索引的名称来标识门和接头以加紧“设计”电路。

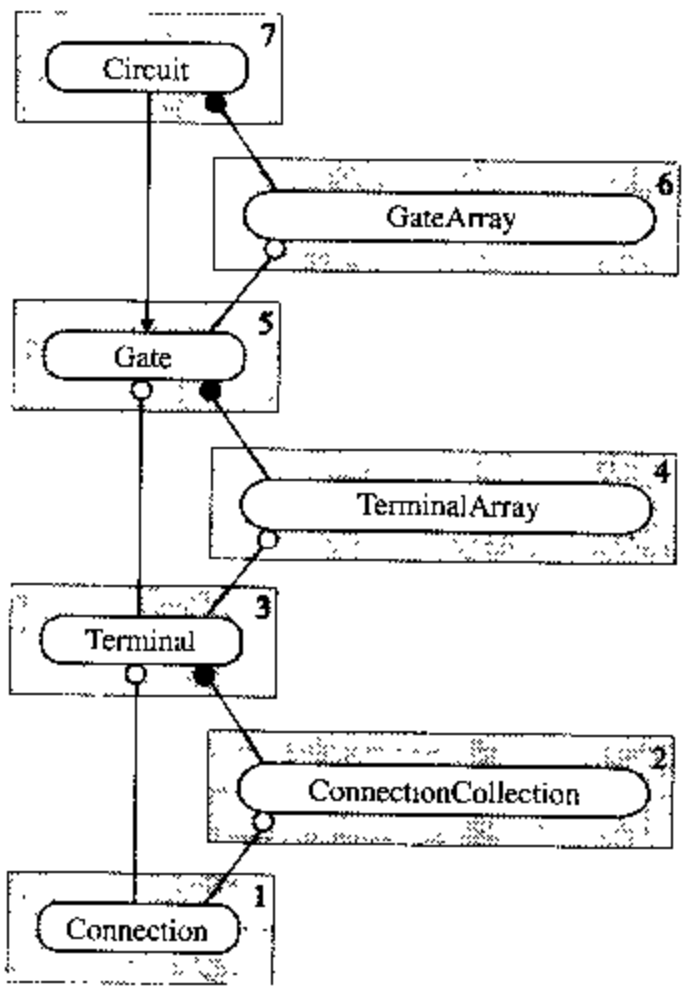


图 5-55 Circuit 实现的新的组件/类图

结论：哑数据是不透明指针的一种泛化，它有助于子系统的实现，在这些子系统中低层次的对象必须隐含地引用其他低层次对象。这种技术在这样的地方尤其有用：某些引用在子系统的较低层次不必解释，而只在某个（通常）较高层次的对象的上下文中解释。在这种受约束的上下文中，尽管会损害类型安全和封装，但实现可以更简洁。哑数据的使用是典型的

低层次实现细节，通常不会暴露在较高层次子系统的接口中。

5.6 冗余

任何种类的重用都隐含着某种形式的耦合。在有些情况下耦合还可能很严重。在本书中，冗余指的是为了避免由重用导致的不需要的物理依赖而故意重复代码或数据的技术。

原 则

与一些重用形式相关的额外耦合可能会超过从该重用获得的利益。

当功能存在于一个独立的物理单位中并且要重用的功能数量相对较小，而会导致的耦合的数量却是不合比例地大，以致于超过了重用的好处时，冗余是必要的。在重用的数量比较多的情况下，通常适合将共有代码降级到一个较低的层次，在那里可以共享它。

甚至在单个的子系统内也有一个限度，在此限度之下外部功能的重用可能不是有益的。考虑两个独立的大组件，有可能其中一个组件实现了一小块功能（例如 min、max 等），而另一个组件可能重用它。把这一小块实现降级到一个独立组件中会不合道理地增加该子系统的物理复杂性，仅仅为了这样一点点重用而导致一个组件依赖另一个组件也会不合道理地增加子系统的 CCD。若允许一个组件去支配另一个组件，那么会减少添加其他依赖关系的灵活性（即将来对组件的增强导致的添加其他依赖关系的灵活性）。有时候对重用来说一个可行的替代方案就是重复代码和避免耦合。

作为一个常见的、实际的例子，考虑图 5-56 中描绘的情形。图中某个低层次的对象 Cell 有（has）一个 name，这个名称在构造该对象时指定，在该对象的整个生存期都不能改变，并且随着该对象的删除而删除。在该对象的公共接口中的一个访问函数应要求提供了这个名称（作为一个 const char*）。除了这个名称，在这个对象中没有任何其他对 String 的使用。

在这里对 String 的使用是 Cell 类的一个封装的实现细节。使用 String 的好处在于没有必要（直接）在 Cell 的构造函数实现中使用 new 运算符，因而不必担心要为尾部的空指针分配额外的字节，或者把引入的 String 拷贝到新分配的缓冲区。或许最大的好处是不必费心在 Cell 的析构函数中删除这个 String。

对于有经验的 C 程序员，上述情况都不应该造成任何要引起注意的维护问题。依赖 String 的不利之处在于，它是必须跟随在 cell 组件周围的额外“行李”。如果 str 不是相同子系统的一部分或者它依赖于其他的组件，那么使用 String（而不仅仅是一个 char*）可能导致必须拖着其他组件甚至库到处跑，从而进一步增加使用 Cell 的负担。

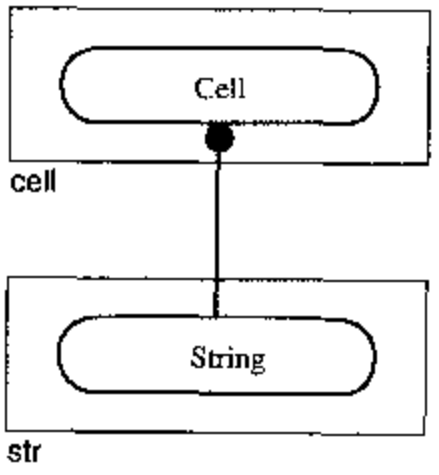
正如图 5-56（a）中定义的，Cell **有（has）**一个 String，因而实质依赖组件 str。Cell 的所有客户程序不仅要承受对组件 str 的一个连接时依赖，而且要承受对 str 的一个编译时依赖。如果 Cell 是像在图 5-56（b）中那样定义的话，这个问题就可以避免。（有关不必要的编译时

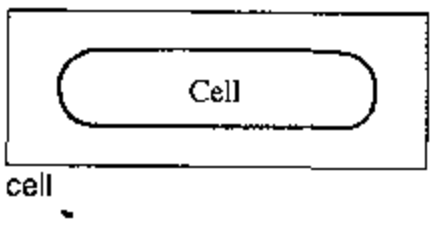
依赖的问题是第 6 章的主题。）

在像图 5-56 中所示的这种情形下，避免对 str 组件的耦合可能会超过重用的益处。如果 cell 组件对 String 的功能进行了任何有意义的使用，例如串联（concatenation），或者 String 在 Cell 的定义中出现多次，情况就不会这样了。

原　则

提供少量的冗余数据可以使对一个对象的使用只是在名称上，从而消除连接到那个对象类型的定义的开销。






```
// cell.h
#ifndef INCLUDED_CELL
#define INCLUDED_CELL

#ifndef INCLUDED_STR
#include "str.h"
#endif

class Cell {
    String d_name;
    // ...
  public:
    Cell(const char *name);
    ~Cell();
    // ...
    const char *name() const;
    // ...
};

#endif
```

（a）使用 String 类

```
// cell.h
#ifndef INCLUDED_CELL
#define INCLUDED_CELL

class Cell {
    const char *d_name_p;
    // ...
  public:
    Cell(const char *name);
    ~Cell();
    // ...
    const char *name() const;
    // ...
};

#endif
```

（b）重新实现 String 的功能

图 5-56　两种方法来实现有（has）一个名称的对象

可以不同的方式来有效地利用冗余，与其他技术结合在一起来减少物理依赖。尤其是，

选择只在名称上使用对象不仅可以有效地打破一个子系统内部的循环依赖，而且可以有效地减少对其他子系统的物理依赖。但是有时候为了让某些对象不透明，有必要提供少量的冗余信息。

考虑图 5-57 中所描绘的场景。我们试图在一个大型的 shape 子系统（假定出 1000 个组件组成）的顶层实现一个 shape 分析器。

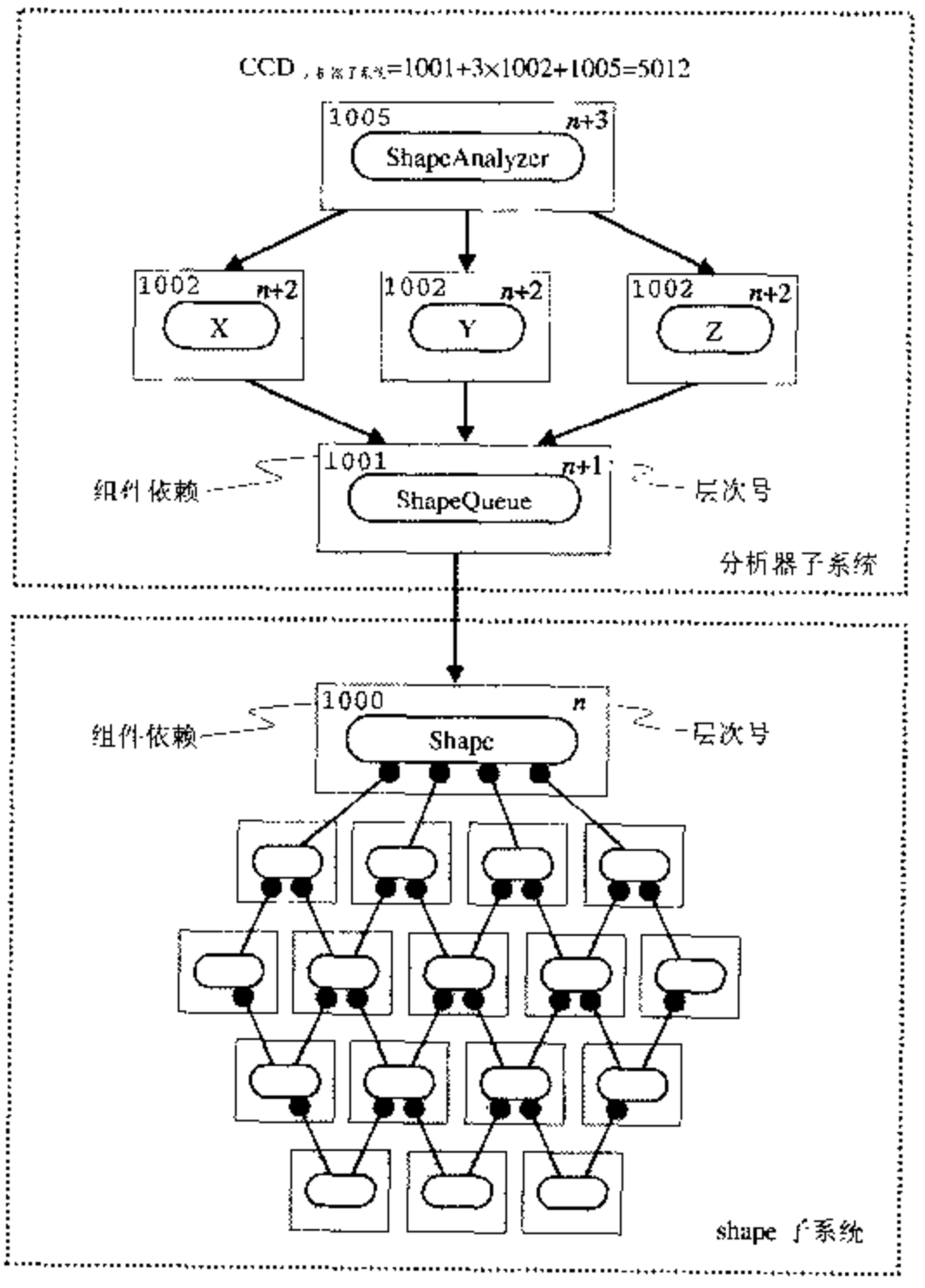


图 5-57 ShapeAnalyzer 被迫使用高度耦合的 Shape 子系统

幸运的是我们只需要使用这个子系统的一小部分，具体说来，就是 Shape 组件。但是这个组件完全依赖于其子系统的其他部分，强加给 shape 一个不合比例的巨大连接开销（如果按它的组件依赖来计算是 1000 单位）。仅仅分析器子系统的五个组件的 CCD（就是说，除了维护 shape 子系统的局部连接开销）就达到了 5012。

常常会有复杂的容器对象（例如一个优先队列）包含（hold）其他的对象，但不需要以任何实质的方式依赖被包含的对象。ShapeQueue 的工作是维持一个按面积排序的 Shape 对象的堆。Shape 类提供一个公共成员函数来返回它的面积。把 ShapeQueue 设计成直接使用 Shape 的 area()成员，将把开发和维护 ShapeQueue（以及它的所有客户程序）的开销捆绑到由 Shape 强加的非常巨大的 CCD 上。

图 5-58 描述了另一个可供选择的体系结构，完全是为了减少维护和测试开销。它不再让 ShapeQueue 直接从一个 Shape 获取面积数据，而是用一个 ShapeManager 析取这个值，并且将这个值以及每个不透明的 Shape 指针一起输入到 ShapeQueue 中。ShapeAnalyzer 的其他实现已经被重新分解，从而使对类 Shape 的所有实质的使用现在都只出现在 shapemanager 组件中，一些额外的冗余数据（即面积）存储在每个 ShapeQueue 条目中，供组件 x、y 和 z 使用。

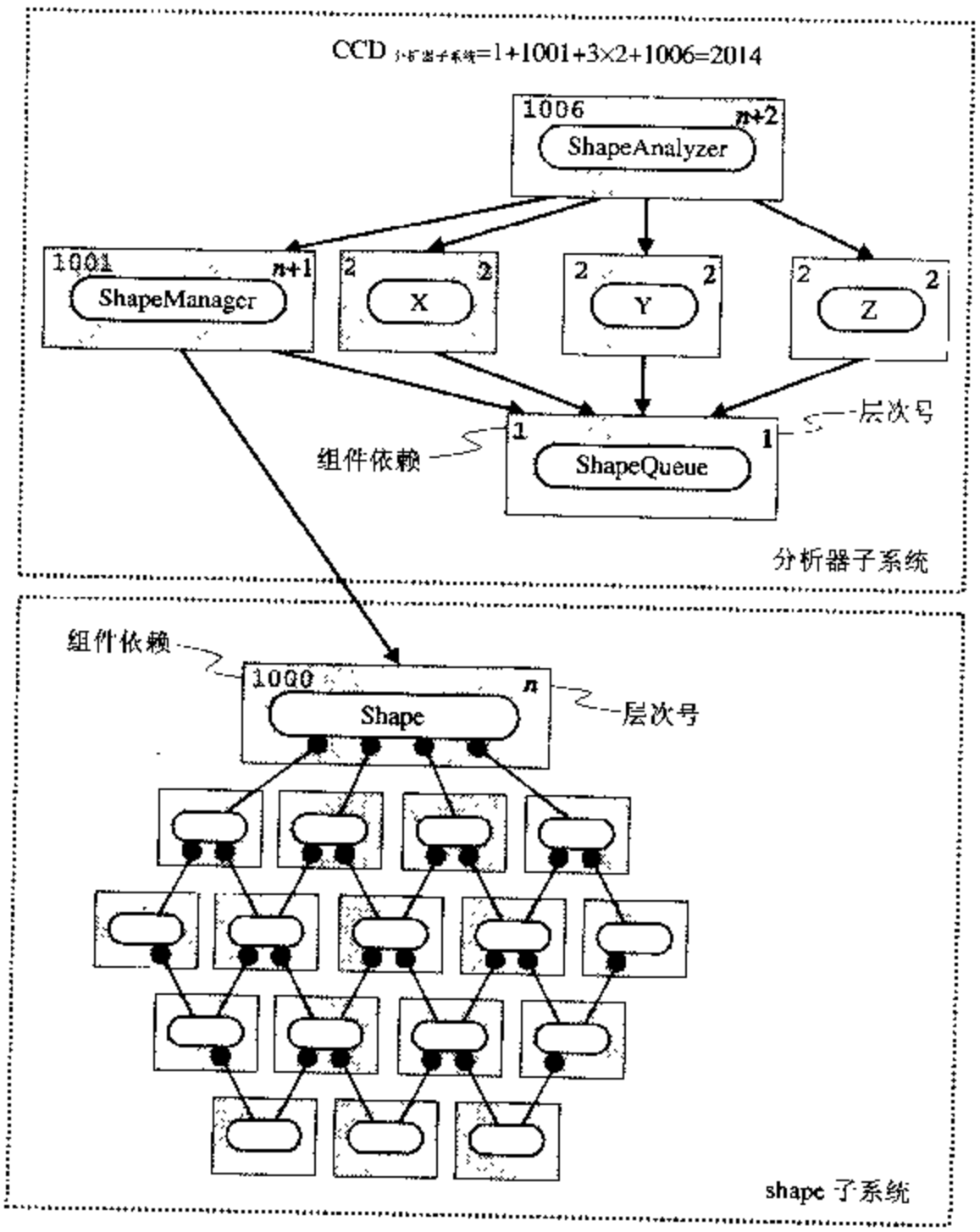


图 5-58 通过使用冗余数据和不透明指针来减少 CCD

原　则

将子系统打包，使其连接到其他子系统的开销最小化，这是一个设计目标。

与分析器子系统有关的、将近 60%的维护开销的减少并非不寻常，与连接到一个大型的高度相互依赖的子系统有关的、相对较大的开销也并非不寻常。一个精心设计的子系统通常会包含相当大比例的不依赖任何其他子系统的组件，并且很少有组件会依赖巨大的、紧密耦合的子系统（就像包含 Shape 的子系统）。

简而言之，重用很少没有开销，并且它的益处必须抵得上由增加的耦合引起的开销。非常常见的是，开销以一种增加的物理依赖的形式出现。用于减少物理耦合的技术，例如不透明指针，有时候会要求提供少量的冗余信息，以便成功应用。在这样的情况下，需要根据现存耦合的数量来决定可以容忍的冗余数量。

5.7　回调

一个**回调（callback）**是一个函数，由一个客户提供给一个子系统，它允许该子系统在该客户程序的上下文中执行一个特定的操作。

作为一个简单的例子，考虑 C 库函数 qsort[①]，它是 Quicksort 算法的一个实现：

```
#include <stdlib.h>

void qsort(const void *base,
           size_t numElements,
           size_t sizeofElement,
           int (*compare)(const void *elem1, const void *elem2));
```

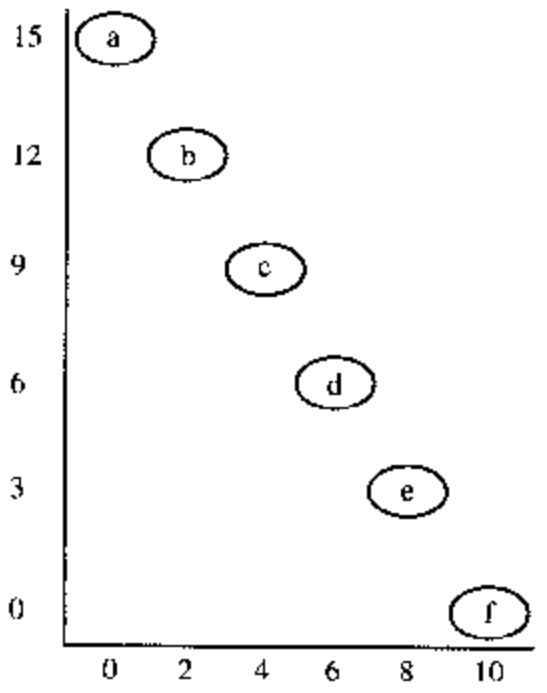
qsort 的第一个参数 base 指出了一个同类对象数组的开始位置，这些对象的类型对 qsort 例行程序来说是未知的。第二个参数 numElements 指出了 base 数组中对象的数量。第三个参数 sizeofElement 指出了每个元素的统一大小（就像由 sizeof 运算符所定义的那样）。第四个也是最后一个参数 compare，是一个指向一个 callback 函数的指针。

qsort 函数假定回调函数 compare 将会正确地判定，它的两个类属指针参数隐含的第一个对象 elem1，应该被认为是小于、等于还是大于第二个参数 elem2，并分别返回一个负值、0 或者正值。

为了说明回调的使用（无危险的），考虑这样一个简单问题：为一个笛卡尔点的集合排序（基于它们与一个二维坐标系统的原点的相对距离）。图 5-59（a）描述了这个问题的一个实

① **plauger**，第 13 章，357~358 页。

例，包含6个点，标记为a~f。图5-59（b）给出了一个Point的定义。




```
// point.h
#ifndef INCLUDED_POINT
#define INCLUDED_POINT

class Point {
    int d_x;
    int d_y;

  public:
    Point(int x, int y) : d_x(x), d_y(y) {}
    Point(const Point& p) :
      d_x(p.d_x), d_y(p.d_y) {}
    ~Point() {};
    Point& operator=(const Point& p) {
      d_x = p.d_x; d_y = p.d_y; return *this; }
    void setX(int x) { d_x = x; }
    void setY(int y) { d_y = y; }
    int x() const { return d_x; }
    int y() const { return d_y; }
};

#endif
```

（a）点集　　（b）简单的point组件的头

图5-59　使用函数qsort为Point对象的一个集合排序

函数qsort通过盲目地将指定元素大小的存储区域与另一个存储区域交换（只是基于从回调函数返回的值）来重新排列条目。位拷贝是使用一个像C库函数memcpy这样的函数来执行的。

通常，用memcpy把对象拷贝到新的位置是危险的（见10.4.2节），因为一个对象可能包含一个指向它自己或其他对象的指针或引用，该对象要负责删除它。另一方面，用memcpy拷贝或移动指向对象的指针又总是安全的。假设我们创建了一个有六个指向Point对象的指针的数组，并且在其中存储了六个Point对象（代表图5-59（a）中的点）的地址。

```
static Point a(0,15), b(2,12), c(4,9), d(6,6), e(8,3), f(10,0);
static Point *array[6] = { &a, &b, &c, &d, &e, &f };
```

为了使用qsort，我们需要给qsort提供一种方法来比较两个不透明条目，这样它才能确定它们的相对顺序。就是说，如果给定了一对指向点的指针的地址（const void *类型的），我们需要一种判定方法来判定，原点到由第一个存储地址指示的Point的距离是小于、等于还是大于原点到由第二个存储地址指示的Point的距离。图5-60给出了这个回调函数的一个适合

我们的直接需要但还不完美的实现[1]。

```
static int pointCompare (const void *addrPoint1Ptr,
                         const void *addrPoint2Ptr)
{
    // poor implementation
    const Point &p1 = **(const Point **) addrPoint1Ptr;
    const Point &p2 = **(const Point **) addrPoint2Ptr;
    int d1sq = p1.x() * p1.x() + p1.y() * p1.y(); // bad idea (overflow?)
    int d2sq = p2.x() * p2.x() + p2.y() * p2.y(); // bad idea (overflow?)
    return d2sq - d1sq;
}
```

图 5-60　比较两个 Point 对象的回调函数的实现

利用指示开始位置、条目数量、每个条目大小的数据，以及一个回调函数（用于判定两个条目在上下文中的位置顺序）来编程，我们可以重用 Quicksort 算法的这种模块化实现来解决我们的问题，如图 5-61 所示。

```
// point.t.c
#include "point.h"
#include <stdlib.h> // qsort()
#include <iostream.h>

static int pointCompare (const void *addrPoint1Ptr,
                         const void *addrPoint2Ptr)
{
    // better (more practical) implementation
    const Point &p1 = **(const Point **) addrPoint1Ptr;
    const Point &p2 = **(const Point **) addrPoint2Ptr;
    double d1sq = p1.x() * (double) p1.x() + p1.y() * (double) p1.y();
    double d2sq = p2.x() * (double) p2.x() + p2.y() * (double) p2.y();
    return d1sq < d2sq ? -1 : d1sq > d2sq; // Warning: may fail on
                                           // points far from origin.
}

static ostream& operator<<(ostream& o, const Point& p)
{
    return o << '(' << p.x() << ',' << p.y() << ')';
}

static ostream& print(ostream& o, const Point *const *array, int size)
```

[1] 在实践中，一个更好的实现应该是为中间计算结果使用一个 double，以避免溢出。这种解决方案是依赖于实现的，并且可能对被放在离原点几乎同样（大）距离的两个点无效。一个健壮但运行时效率更低的解决方案应该是使用一个用户自定义类型（例如 DoubleInt），它被允许拥有至少两倍于 int 的比特。

```
{
    o << '{';
    for (int i = 0; i < size; ++i) {
        o << ' ' << *array[i];
    }
    return o << " }";
}

const int SIZE = 6;

static Point a(0,15), b(2,12), c(4,9), d(6,6), e(8,3), f(10,0);

static Point *array[SIZE] = { &a, &b, &c, &d, &e, &f };

main()
{
    print(cout, array, SIZE) << endl;
    cout << "Now sort by distance from origin:" << endl;
    qsort(array, SIZE, sizeof *array, pointCompare);
    print(cout, array, SIZE) << endl;
}
```

图 5-61 使用一个回调来对 qsort 的比较行为进行编程

读者要认识到，远在 Point 类或这个例子编写之前，qsort 已被开发、测试和重用了很多很多次。大多数由 Quicksort 算法所做的工作都是可重用的。只有一个行为 compare，每个用法都不尽相同。提供一个回调是因为它使我们能够分解和重用这个功能。

原 则

不加选择地使用回调可能导致难以理解、调试和维护的设计。

在 qsort 的接口中缺少类型安全这一点是非常明显的。但是因为 qsort 是一个有着单一可编程行为的无状态算法，所以对于是否需要采用类属排序对象是有争议的。但是，这里有一个隐含的数据结构。如果我们有理由维护多种已排序的 Point 集合（它们由不同的比较例行程序进行排序），那么创建一个有着相应迭代器的抽象 OrderedPointCollection 基类（见图 5-62）通常被证明是有用的。这样做也可以在编译时捕获大多数类型错误。

```
class OrderedPointCollection {
    // ...
  public:
    // CREATORS
    OrderedPointCollection();
    virtual ~OrderedPointCollection();

    // MANIPULATORS
    void add(Point *point);
```

```
    private:
      // ACCESSORS
      virtual int compare(const Point& point1, const Point& point2) = 0;
};
```

图 5-62 一个任意排序点集合的抽象基类

通过派生一个简单的 struct 来指定比较函数：

```
struct MyPoints : OrderedPointCollection {
    compare(const Point& point, const Point& point);
    ~MyPoints(); // empty -- (see Section 9.3.3)
};
```

现在我们可以像下面这样在类型安全的意义上覆盖这个比较函数：

```
int MyPoints::compare(const Point& p1, const Point& p2)
{
    // better (more robust) implementation
    DoubleInt p1x = p1.x();                          // DoubleInt is a type
    DoubleInt p1y = p1.y();                          // that is at least
    DoubleInt p2x = p2.x();                          // twice as big as int.
    DoubleInt p2y = p2.y();
    DoubleInt d1sq = p1x * p1x + p1y * p1y;          // robust but slow
    DoubleInt d2sq = p2x * p2x + p2y * p2y;          // robust but slow
    return d1sq < d2sq ? -1 : d1sq > d2sq;           // robust but slow
}
```

系统的层次化显示在图 5-63 中。注意类 OrderedPointCollection 只在名称上依赖 Point，但是 MyPoints::compare 却实质依赖 Point。虚函数正在扮演一个“回调”函数，因为比较操作必须在 Point 的真正定义的上下文中执行。与回调函数获取两个类属指针不同，虚函数期望的是指向 Point 对象的 const 引用。这种 OrderedPointCollection 对 Point 的 in-name-only 依赖提供了一种受欢迎的类型安全，在使该组件更容易使用的同时改进了可维护性。

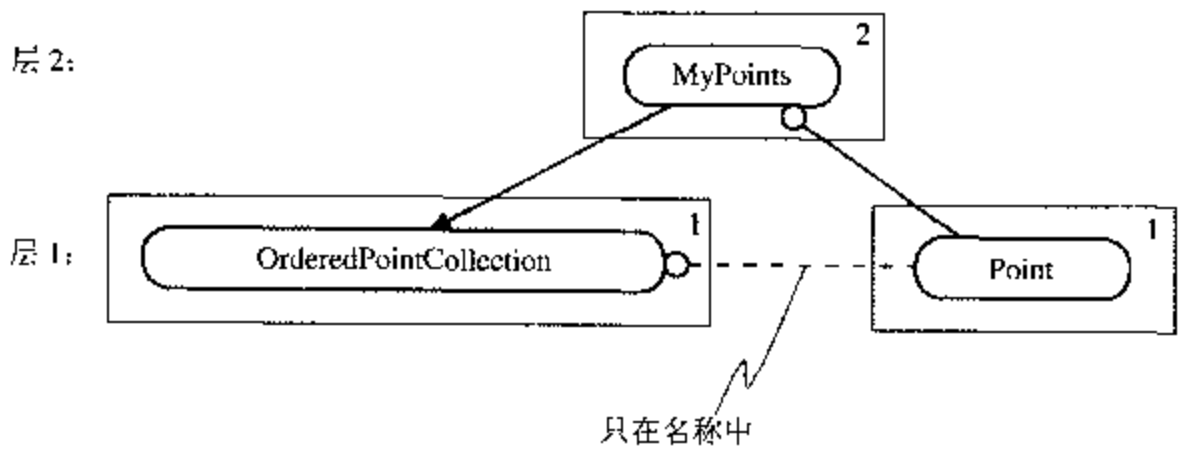


图 5-63 把一个虚函数用作一个回调来使分解能够进行

回调是强有力的消除耦合的工具，但是应该只在必要的时候使用它们。由一对互相调用

对方的成员函数的类引起的相互依赖是拙劣设计的一个症状。回调有时候可以用来打破循环，但通常这个问题最好通过对功能重新打包来处理。

再来考虑一下图 5-27 所示的最初的、分解拙劣的运行时数据库体系结构。如果每个解析器的 read 函数都实现一个无状态算法，我们可以令人信服地把解析函数作为一个回调传递给 RuntimeDB：

```
RuntimeDB::Status RuntimeDB::read(
                    RuntimeDB::Status(*parseFunc)(const char *),
                    const char *filename);
```

但是，它导致的困惑可能是不合道理的。和前一个例子中的 OrderedPointCollection 没有实质依赖 Point 不同，每一个具体的解析器都必须知道数据库的所有情况才能加载它。如果解析涉及状态和/或一个多功能接口，标准的面向对象方法将会创建一个抽象的解析器基类，并为特定格式的使用派生具体的解析器，如图 5-64 所示。

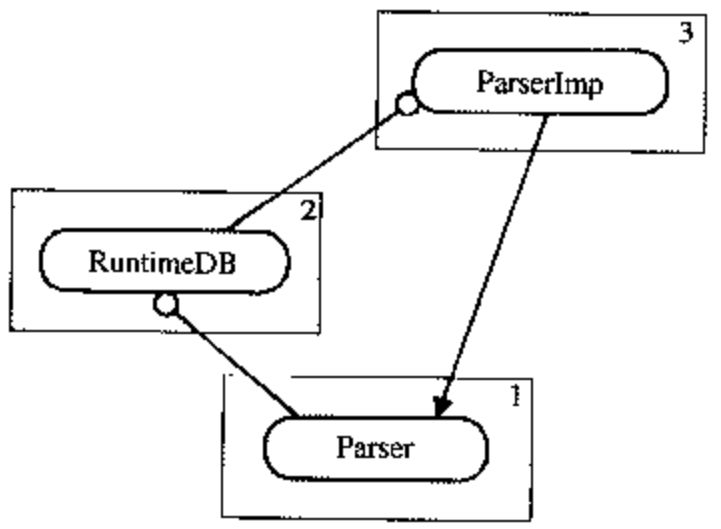


图 5-64 将接口降级来实现一个回调

这个可供选择的修改过的体系结构比最初在 5.3 节中介绍的解析器设计要好，因为在单独的解析器之间既没有物理耦合，也没有任何处理器对任何解析器实现的依赖。但是，这个体系结构并不是最优的，因为它强迫运行时数据库知道对所有解析器都通用的接口：

```
#include "parser.h"

RuntimeDB::Status RuntimeDB::read(Parser *parser, const char *filename)
{
    parser->read(this, filename);     // virtual function call
}
```

运行时数据库可能被其他不需要解析器的系统重用。将运行时数据库和一个特定的解析器接口连接在一起不必要地限制了子系统，使它更不通用、更不可理解和更不适合重用。如果在解析过程中所需要的信息是经常更新的，那么这种不必要的耦合也可能会对运行时数据库的可维护性产生不利影响。

这个系统最好的设计是在 5.3 节的图 5-29 中介绍的修正的体系结构，它把数据库放在系统层次结构的底部，对解析器完全没有依赖。该体系结构允许数据库组在完全隔离的情况下开发和测试它的子系统，而不是夹在开发解析器的数据库组提供的接口与实现之间。这个例子的寓意是，有时候要避免回调的不必要使用。

回调也可以静态安装（即，在任何实例之外）。例如，new 处理程序[1]是一个有着合理初始行为的静态回调函数。客户可以用他们自己的函数来替换默认的函数，以便允许他们在一个更高层次的上下文中整理他们的应用程序。

原　则

对回调的需求可能是不良的整体体系结构的一个症状。

但是有时候静态回调可以用来很好地消除对大型子系统的依赖。考虑图 5-65 中所示的子系统，我们需要实现一列不同种类的行星——也就是通过继承与基类 Planet 相关的一列对象。因为这个列表是多态的，所以每个连接都不能 **have** 一个 Planet，而必须代之以 **hold** 一个 Planet（的地址）。实际的 Planet 对象由客户动态分配并移交给列表，从那时起由该列表负责 Planet 的存储。当 PlanetList 被析构时，它所包含的所有 Planet 对象也将被析构。

正如您可能完全想象得到的，Planet 是一个非常大而且复杂的基类对象，有着许多依赖和一个相对较高的连接时开销。我们想要避免 Planetlist 对 Planet 的一个物理连接时依赖，尤其是在这种（无疑是不寻常的）情况下：SolarSystem 另外又只在名称上依赖 Planet。

我们可能试着只用指向 Planet 对象的不透明指针来实现 PlanetList。这样做的问题在于我们的 PlanetList 不是已看到了 Planet 类的定义，因而不会知道如何析构它。我们可以修改 PlanetList 的规范，使它不能自己析构 planet，并且把该功能升级到一个更高的层次（例如：SolarSystem），正如在图 5-66 中所建议的。

但是在我们这个例子中，甚至 SolarSystem 也只是在名称上使用了 Planet。因为 PlanetList 类型的使用是 SolarSystem 的一个封装的实现细节，所以，如何把这个功能升级到任何更高的层次并不是显而易见的事情。

如果有对整个子系统的完全控制，一个好的解决方法可能是把 Planet 的接口降级，如图 5-67 所示。现在 Planet 只是一个接口，所有的物理耦合都提升到了一个更高的层次，不再影响 SolarSystem。现在，测试 PlanetList 要求在 PlanetList 驱动程序中为 Planet 派生一个微小的“桩”实现。

不幸的是，我们不能控制宇宙，因而必须忍受一个被不良分解的 Planet。我们仍然可以打破物理依赖，但需要使用一个冗余的回调函数。假设我们把一个以下类型的静态成员加入 PlanetList 类中：

---

① **stroustrup**，9.4.3 节，312~314 页。

```
typedef void DestroyPlanetFunc(Planet *);
```

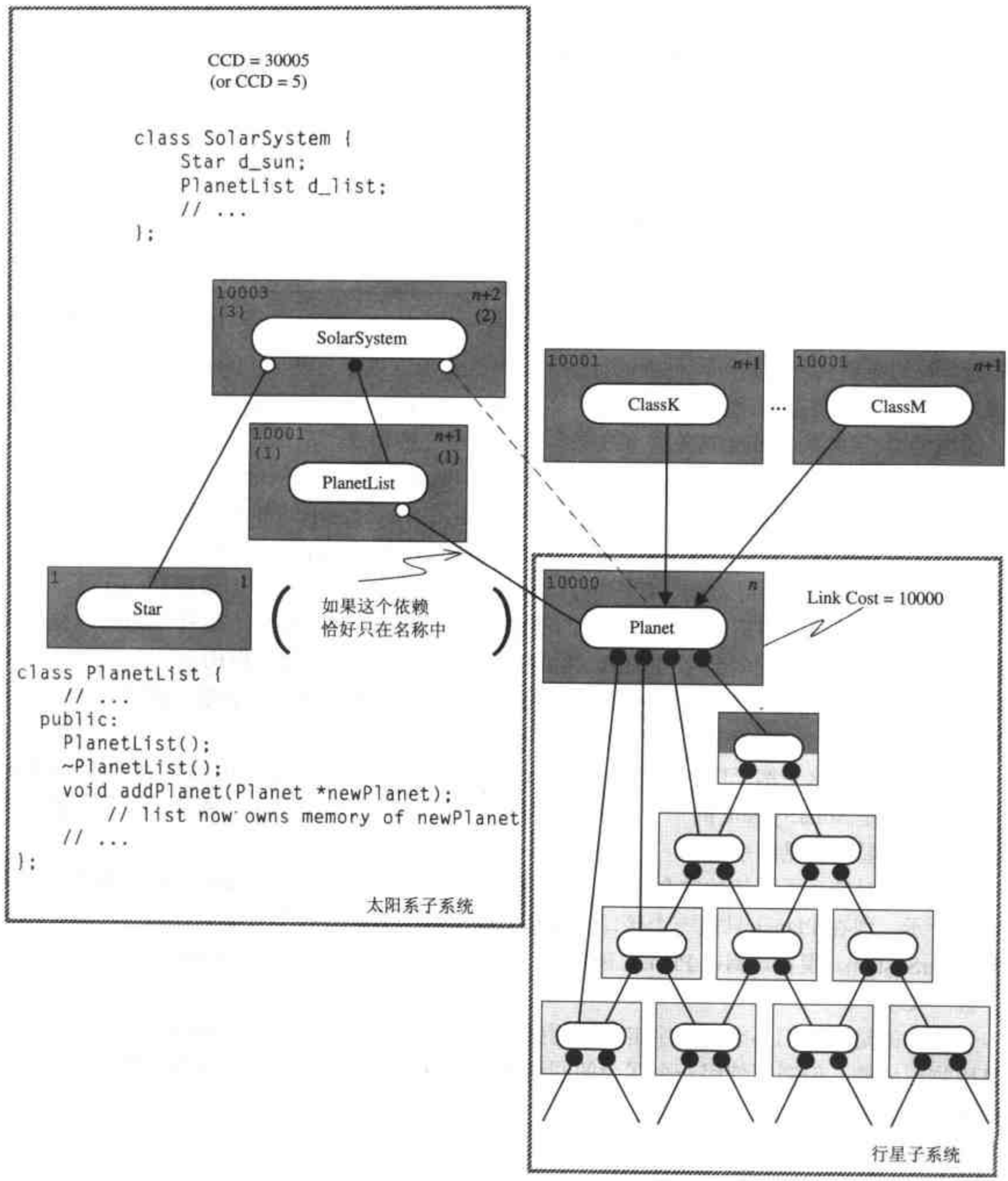


图 5-65 一个“大”问题

```
class SolarSystem {
    Star *d_sun_p;
    PlanetList d_list;
    static void destroyPlanets(PlanetList *list);
    // ...
  public:
    // ...
    ~SolarSystem() { destroyPlanets(&d_List); }
    // ...
};
```

图 5-66　把 planet 析构函数升级到一个更高的层次

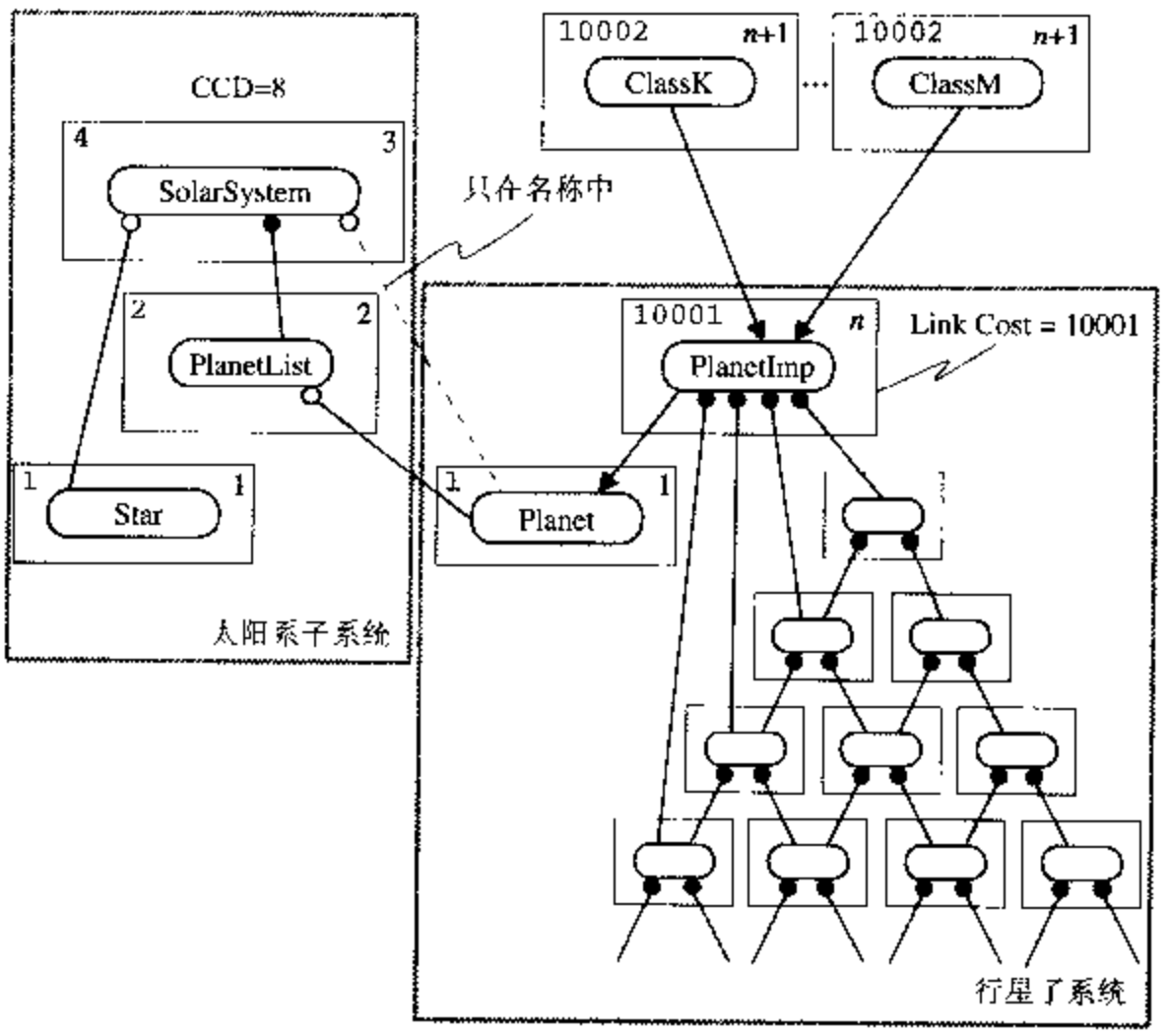


图 5-67　把 Planet 的接口降级到一个更低的层次

现在 PlanetList 类有了一个静态数据成员，即一个指向一个回调函数的指针，该回调函数潜在地提供析构 Planet 类的一个实例所需的必要的上下文。在第一次使用一个 PlanetList 之前，一个客户（他知道 Planet）应该通过传递一个适当定义函数的地址调用静态方法 PlanetList::setDestroyPlanetFunc 来“起动”这个类，如图 5-68 所示。当 PlanetList 被析构时，它可以调用它拥有的每个 planet 上的 destroyPlanet 函数。

图 5-69 给出了 planetlist 组件相关部分的粗略草图。类 PlanetList 提供了一个机制，允许

高层次的客户安装回调函数来析构一个 Planet。当析构一个 PlanetList 时，析构者检查是否已安装了一个析构函数，如果是，则依次把它应用到列表中的每个 Planet 上。如果在析构 PlanetList 时还没有安装回调函数，则被包含的 Planet 对象不被析构，并且与每个 Planet 相关的动态内存被“泄漏”。（内存泄漏在 10.3.5 中讨论。）

```
// client.c
#include "client.h"
#include "planet.h"
#include "planetlist.h"
// ...
static void destroyPlanet(Planet *p) { delete p; }
// ...
Client::init()
{
    PlanetList::setDestroyPlanetFunc(&::destroyPlanet);
    // ...
};
// ...
```

图 5-68 安装一个冗余的回调

以这种方式使用回调并不是最好的。使用 PlanetList 要求客户知道本应该不必为之费心的低层次细节。不推荐将这种方法应用于公共接口，因为人们在使用容器类之前很容易忘记对它进行初始化。使问题更糟糕一点的是，PlanetList 不是在 SolarSystem 的公共接口上。这就有必要让 SolarSystem 提供一个像下面这样的静态成员：

```
class SolarSystem {
    // ...
  public:
    static void init(void (*)(Planet *));
    // ...
};
```

然后它必须把初始化调用转递给 PlanetList 类。

总结：回调是一个函数，由一个客户提供，用以允许一个（通常是）较低层次的组件利用一个行为，这个行为需要一个（通常是）较高层次的上下文。虚函数可以用来实现一个类型安全回调机制。回调是打破协同操作类之间的依赖的强有力工具。回调对于图形学和基于事件程序设计是极其重要的。

如果不适当地使用，回调可能会模糊低层次对象的职责并导致不必要的概念上的耦合。通常，回调（就像递归一样）可能比传统的函数调用更难以理解、维护和调试。它们的（伪）异步特性需要开发人员给予一种不同类型的关注。作为一个规则，回调应该被当作是最后求助的避难所。

```
// planetlist.h
// ...
class PlanetListIter;
class PlanetList {
    // ...
    friend PlanetListIter;
  public:
    typedef void DestroyPlanetFunc(Planet *);
  private:
    static DestroyPlanetFunc *d_destroyPlanetFunc_p;
  public:
    static void setDestroyPlanetFunc(DestroyPlanetFunc *func);
    // ...
    ~PlanetList ();
    // ...
};
// ...
class PlanetListIter {
    // ...
  public:
    PlanetListIter(const PlanetList &list);
    ~PlanetListIter();
    void operator++();
    operator const void *() const;
    const Planet& operator()() const;
};
// ...
```

```
// planetlist.c
#include "planetlist.h"

PlanetList::DestroyPlanetFunc *PlanetList::d_destroyPlanetFunc_p = 0;

void PlanetList::setDestroyPlanetFunc(DestroyPlanetFunc *func)
{
    d_destroyPlanetFunction_p = func;
}

void PlanetList::~PlanetList()
{
    if (d_destroyPlanetList_p) {
        for (PlanetListIter it(*this); it; ++it) {
            (*d_destroyPlanetList_p)(it());
        }
    }
    else {
        // memory leak!
    }
}
```

图 5-69　使用回调使独立测试能够进行

5.8 管理类

在使复杂度和工作最小化的名义下，很容易使类变得过于简单。试图只用一个单一的类来实现一个整数列表是这种常见错误的一个很好的例子。有人可能会建议，一个列表可以只是一个指向一个 Link 的指针[图 5-70（a）]，或者连接操作可以和与 List 本身相关的方法合并[图 5-70（b）]。

方法（a）[如图 5-70（a）所示]的问题在于抽象级别太低，不能使一个应用程序有效地使用。方法（b）[如图 5-70（b）所示]未能封装 List 的私有实现细节。一个列表抽象的客户通常不愿意为管理单独连接内存的低层次细节而费心，或者为确保执行一个列表实现的低层次策略而费心。

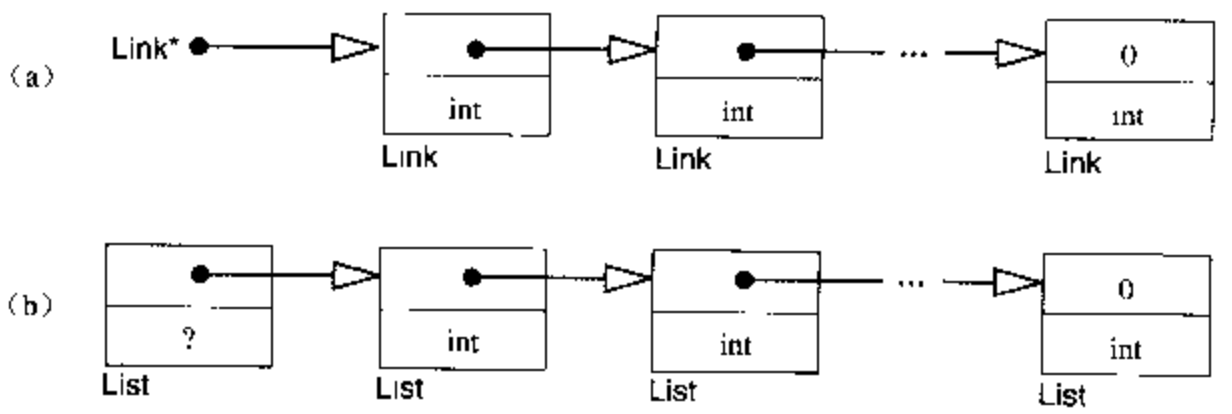


图 5-70 如何不去实现一个 list 组件

甚至在两个类的列表体系结构中，从属类的角色也可能被滥用。通常，一个列表对象自己会直接析构它的每一个连接，但是，如图 5-71 所示，这个 List 的析构函数只删除顶部 Link，每个 Link 依次递归地删除它的 d_next_p 指针。这种“一流的”方法（除了更慢并要冒溢出长列表的程序堆栈的危险之外）使得哪个对象拥有哪个对象更不清楚，主要是因为相同类型的实例有权彼此析构。为了让 List 类在被析构时更干净地消除，一个更好、更层次结构化的方法是遍历 Link 对象的列表依次删除每一个 Link，如图 5-72 所示。

```
class List {
    Link *d_head_p;

  public:
    // ...
    ~List() { delete d_head_p; }
                // bad idea
    // ...
```

```
class Link {
    Link *d_next_p;
    int d_data;
  public:
    // ...
    ~Link() { delete d_next_p; }
                // bad idea
    // ...
```

图 5-71 有 Link 的 List，它递归地删除下一个 Link

```
List::~List()
{
    while (d_head_p) {
        Link *p = d_head_p;
        d_head_p = d_head_p->next();
        delete p;
    }
}
```

图 5-72　有析构器的 List，它迭代地删除每个 Link

原　则

建立较低层次对象的分等级的所有权，可以使一个系统更容易理解和更可维护。

以一个公司为例，正规雇员不会彼此雇用和解雇，这种职权是给管理者保留的。本质的问题在于，用于实现一个抽象的类与用于执行策略、管理内存、调整实现类的管理类之间没有区别。注意，管理类知道它的下级类，但反之则不然。

类的实例之间时常出现的循环相互连接似乎在暗示，这种循环属性应该反映在一个系统的物理设计中。对于本质上紧密耦合、其定义又很容易适应于单个组件的小型循环依赖对象的网络来说，也许没有理由要消除这样的循环。也就是说，如果从易用性和重用的角度来看在单个物理单位中存在两个或更多的内聚逻辑单位有意义，并且这种组合实现的功能复杂性不会对有效测试造成障碍，那么也许就没有问题需要解决。另一方面，耦合也可能是由于不知道如何避免相互依赖或者甚至是没有首先考虑这个问题而导致的结果。

作为另一个可证明管理类概念有用的例子，考虑一幅简单的由节点和边组成的图。一幅图是最基本的异种类网络之一；而节点可能像局域网中的一台工作站或太阳系中的一个行星一样复杂。换句话说，节点和边的独立于图的那部分的大小和复杂度，和它的与网络相关的部分相比可能会非常大。正是在这种情况下有很大必要消除节点和边之间的相互影响。

让我们从图 5-10 给出的情形开始。假定 Node 和 Edge 是复杂的，并且应该属于截然不同的物理组件，那么通过在这个前提下尝试开发一个简单图，我们可以说明有关构建异种对象的可层次化的互连网络的原理。

一个众所周知的避免循环物理依赖的有效技术就是让所有指向较高层次组件的指针和引用都是只在名称上的。或许我们可以策划一个可层次化的子系统，其中 edge 支配 node。我们的策略将是让 Node “hold” 不透明 Edge 指针的一个集合，如图 5-73 所示。采用这种方法意味着所有涉及边的实质问题都不能在 node 层回答。

在试图独立测试组件 node 的过程中，我们马上意识到了一些问题（参考图 5-74）。首先，客户一定不能直接增加一个指向 Node 的 Edge 指针，否则 Node 对象会知道这个新增加的 Edge，但是 Edge 对象却仍然不知道它有一个到 Node 的新连接，这使系统处于一种矛盾的状态。所以，只有 Edge 对象可以增加一个指向 Node 的 Edge 指针，但是由于 Node 和 Edge 定

义在不同的组件中，这项策略没有办法执行（见 3.6.2 节）。

其次，测试 Node 需要创建一个哑 Edge 类，以便能够访问私有的 addEdge 函数——就是说，我们不能够只从 Node 准备建立的公共接口来测试 Node。

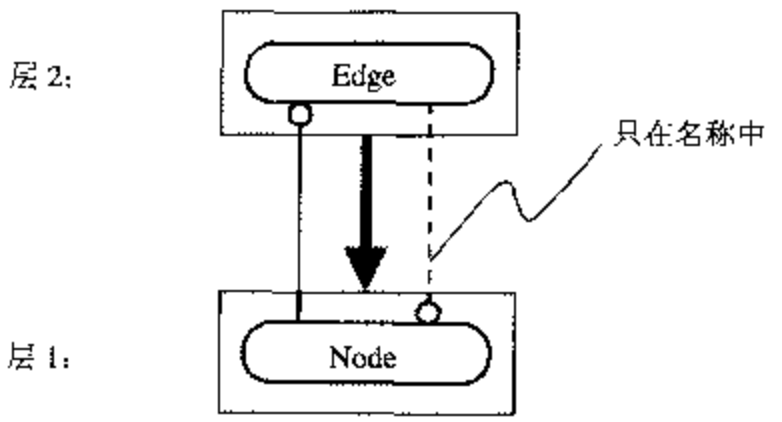


图 5-73 Node 只在名称上使用 Edge

```
class Node {
    //...
    friend Edge;                        // long-distance friend
    void addEdge(Edge *edge);           // private, set only by edge
    Node(const Node&);
    Node& operator=(const Node&);

  public:
    Node(const char *name);             // Who owns the memory for nodes?
    ~Node();                            // Who is allowed to destroy them?
    const char *name() const;
    int numEdges() const;
    Edge& edge(int index) const;        // Reference hampers testing slightly
};                                      // since Edge is used in name only.
```

图 5-74 与最初图的设计相关的问题

第三，Node 的 edge 函数是设计正确的（从最终用户的角度来看），它返回引用，不返回指针。一个引用（甚至是不透明的一个）和一个指针不同，它必须标识一个有效对象的地址，因而不能是空的或指向一个非法地址。这样，如果我们请求一个新创建 Node（它没有边）的 Edge，我们就会陷入麻烦。在 node 组件的层次上增量式地测试 Node 的公共 edge 函数，不仅需要创建一个哑 Edge 类来访问 Node 的私有 addEdge 函数，而且还需要增加这个假 Edge 类的真实的实例，这样它们的（合法）地址才能在以后和 Node::edge(int)返回的左值相比较。

最后，谁拥有 Node 实例的内存或谁被允许创建和析构它们，都是不明了的。例如，如果我们试图在还未删除一个 Node 的所有的边之前析构这个 Node，会发生什么？答案是没有什么不寻常的事发生——至少不会马上发生。因为 Node 不知道 Edge，它不知道怎样去析构一个 Edge。用一个 Edge 去访问一个已删除的 Node 当然会导致不可预测的行为。我们可能将一个回调函数传递给 Node，它知道如何删除一个 Edge，但是这样的话我们必须确保 Edge 对象

只在堆中创建。

到这个时候我们的设计已经失去了势头。正如在实践中经常发生的那样，我们必须返回去考虑这个我们试图实现的抽象，也就是一幅图。就像 rectangle 和 window、node 和 edge 那样本质上都不支配对方。这里有一个涉及所有权的相互依赖，我们需要将它升级到系统的更高层次上。

图 5-75 显示了新设计的基本体系结构，它将被作为一个合理的出发点。类 Graph 负责管理与 Edge 和 Node 的实例有关的内存。节点和边将通过 Graph 的接口加入到图中，而不再独立地创建它们。当从图中删除一个 Node 时，Graph 本身将确保所有连接在那个 Node 上的 Edge 对象首先被删除。这个基本设计仍然要受到 Node 和 Edge 都必须将 Graph 声明为一个友元的问题的困扰，否则不守规矩的客户可能会使图子系统内部出现不一致，例如，增加一个 Edge 到一个 Edge 和 Graph 都不知道的 Node 上。因为我们希望 Node 和 Edge 定义在不同的组件中，所以我们（对这个设计）还是不满意。

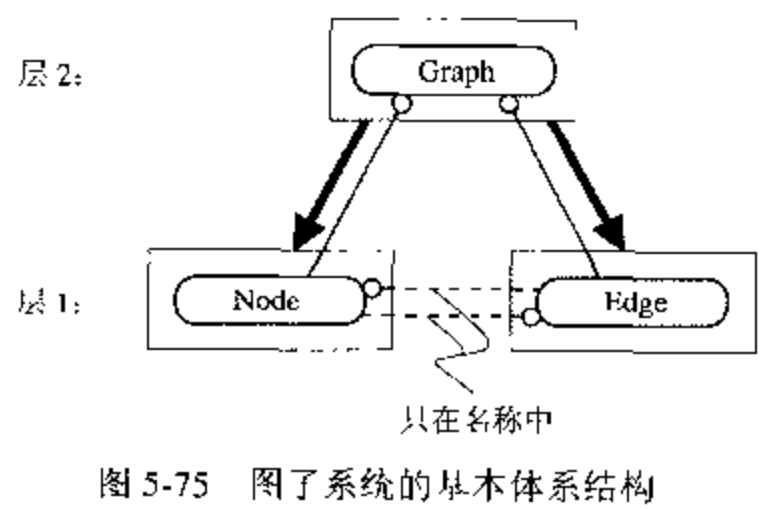


图 5-75 图子系统的基本体系结构

对一个简单图来说，可能完全有理由把所有三个类都放在一个单一的组件内。但是因为在这里我们的目标是要以此例阐明如何实现复杂得多的网络，所以我们将不采用那种方法。（至少）有两种其他的方法可用来处理这个问题：

（1）从耦合系统中分解出尽可能多的代码放到独立组件中，并且将其余的相互依赖的类放在一个单个的组件中。

（2）若在某层上可通过对整个子系统进行封装来消除对低层友元关系的需求，那么就升级该层。

这两项技术分别在下面两节中详细讨论。

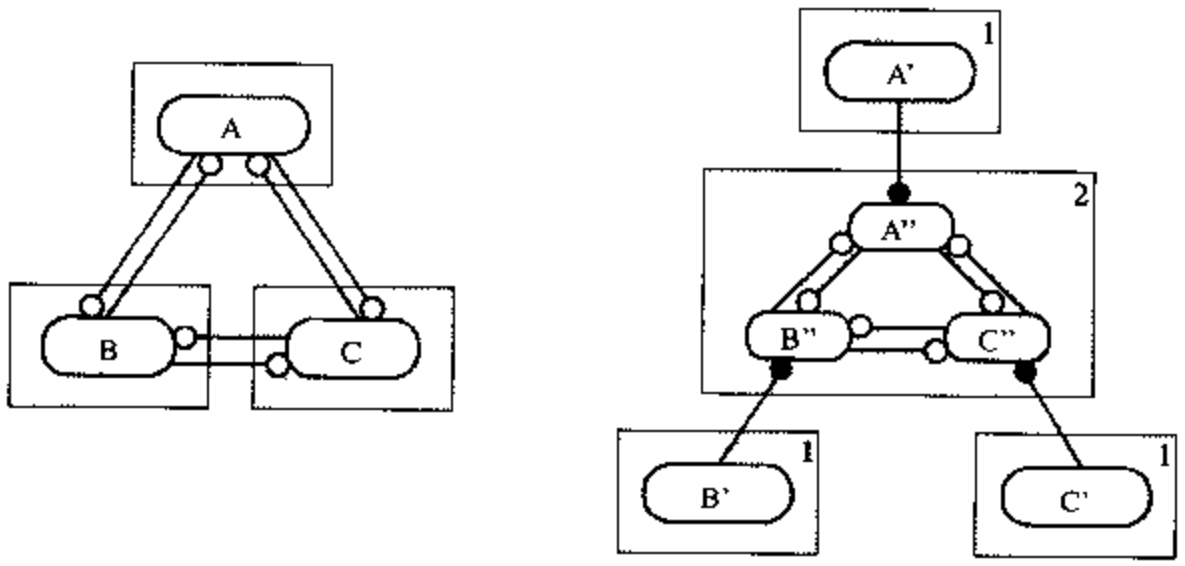
总结：建立协同操作对象的清晰的所有权对好的设计来说是基本的。如果两个或更多的对象共享共有的所有权，那个功能应该升级到一个管理类。

5.9 分解

分解（factoring）的意思是提取小块的内聚功能，并把它们移到一个较低的层次，以便它

们可以被独立地测试和重用。分解是减轻由循环依赖类强加的负担的一种非常普通而高效的技术。分解和降级类似，只是分解的行为不必然消除任何循环；取而代之的是它只减少参与到循环中的功能的数量。通过分解可以将循环依赖升级到一个更高层次（在那儿它们的不利影响较不显著）。

为了说明分解的用途，假设给定了一个设计，由三个本质上相互依赖的类 A、B 和 C 组成，如图 5-76（a）所示。进一步假设原有的逻辑接口是铁板一块，不可以修改。但更有可能的是，不是所有在这三个类中实现的功能都不可分离地与其余部分耦合。我们可以用分解技术来提取任意的独立可测试的实现复杂事务，从而减轻维护代码的真正循环依赖部分的负担。正如在图 5-76（b）中所阐明的那样，如果我们成功地把相当数量的实现分解进了独立的组件中，那么剩余的相互依赖的代码可能足够小，以至于可以将其放进一个单个的组件内。



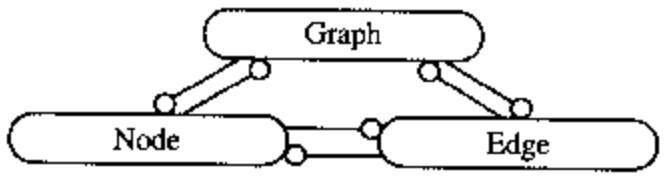
（a）初始的、不可层次化的设计　　（b）分解过的、可层次化的设计

图 5-76　分解出独立可测试的实现细节

原　则

将独立可测试实现细节分解出来并降级，能够减少维护一个循环依赖类集合的开销。

幸好，graph 示例没有上述假设的情况那么极端。我们的逻辑设计中有一些灵活性，隐含的物理依赖也没有我们假设的那么严重。眼下，让我们继续假设最坏的情况——即，我们最初的 graph 子系统是一个由三个本质上相互依赖的类组成的设计：



首先，使用分解将 Node 中的拥有与 graph 相关的数据的部分和拥有独立于 graph 的数据的部分分离开来。继承对这种分解比较理想。我们可以对 Edge 做同样的事情。基本思想如图

5-77 所示。

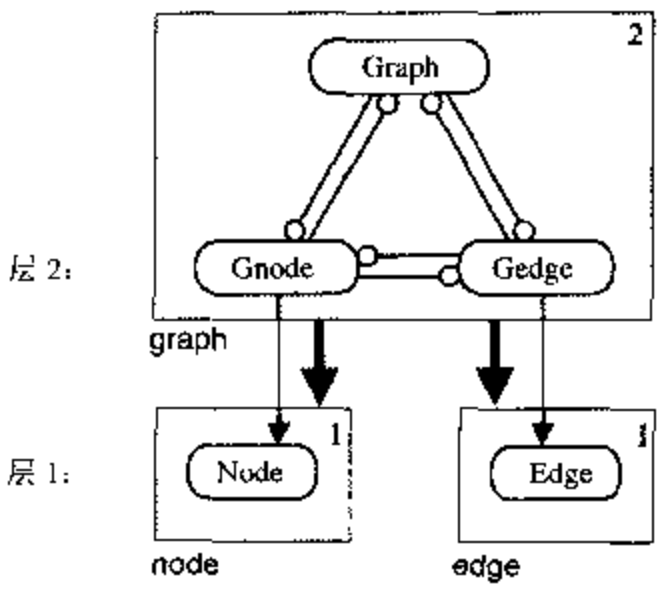


图 5-77 分解出网络独立数据

在这个新设计中，所有紧密耦合的、与 graph 相关的功能都驻留在一个单个的组件内，用三个类 Graph、Gnode 和 Gedge 来实现。包含在 Node 和 Edge 中的独立于 graph 的数据现在都被推到下一个更低层次，并且可以被不关心与 graph 相关的功能的其他应用程序共享。

图 5-78 描述了 node 和 edge 的分解的、独立于网络的部分。在这个微小的例子中，一个 Node 只有一个名称，一个 Edge 也只是一个 double。但是暂时假设图中的节点是现实中的城市，边是道路。一个城市的网络组件，在 Gnode 中是隐含的，不必对一个 Node 本身执行许多复杂操作。Gnode 只是一种特殊的 Node，它参与 Graph 操作。一旦从 Graph 获得了 Gnode 的一个实例，该实例可以用在需要 Node 的任何地方，如图 5-79 所示。

```
// node.h
#ifndef INCLUDED_NODE
#define INCLUDED_NODE

class Node {
    char *d_name_p;

  public:
    Node(const char *name);
    Node(const Node&);
    ~Node();
    Node& operator=(const Node&);
    const char *name() const;
};

#endif
```

（a）独立 node 组件

```
// edge.h
#ifndef INCLUDED_EDGE
#define INCLUDED_EDGE

class Edge {
    double d_weight;

  public:
    Edge(double weight);
    Edge(const Edge&);
    ~Edge();
    Edge& operator=(const Edge&);
    double weight() const;
};

#endif
```

（b）独立 edge 组件

图 5-78 分解的、独立于网络的组件 node 和 edge

```
class Node;
class ostream;

class Census {
    static int countPeople (const Node& node)
        // ...
};

#include "graph.h"
int g(const Gnode& gnode)
{
    return Census::countPeople(gnode); // uses only the Node portion
}
```

图 5-79 重用 Node 的独立于网络的部分

分解节点和边的另一个好处涉及一个称为**值语义（value semantics）**的概念。说一个类型有值语义的意思就是，一个拷贝构造函数和（通常是）一个赋值运算符对该类型天生就是（即，语义上就是）合法操作[1]。

例如，考虑一个公寓综合建筑，它包含固定数量的土地，上面建造了单身家庭住宅。土地分成 25 块，以 5×5 的网格排列。地块的行标记为 A~E，列标记为 1~5，如图 5-80 所示。

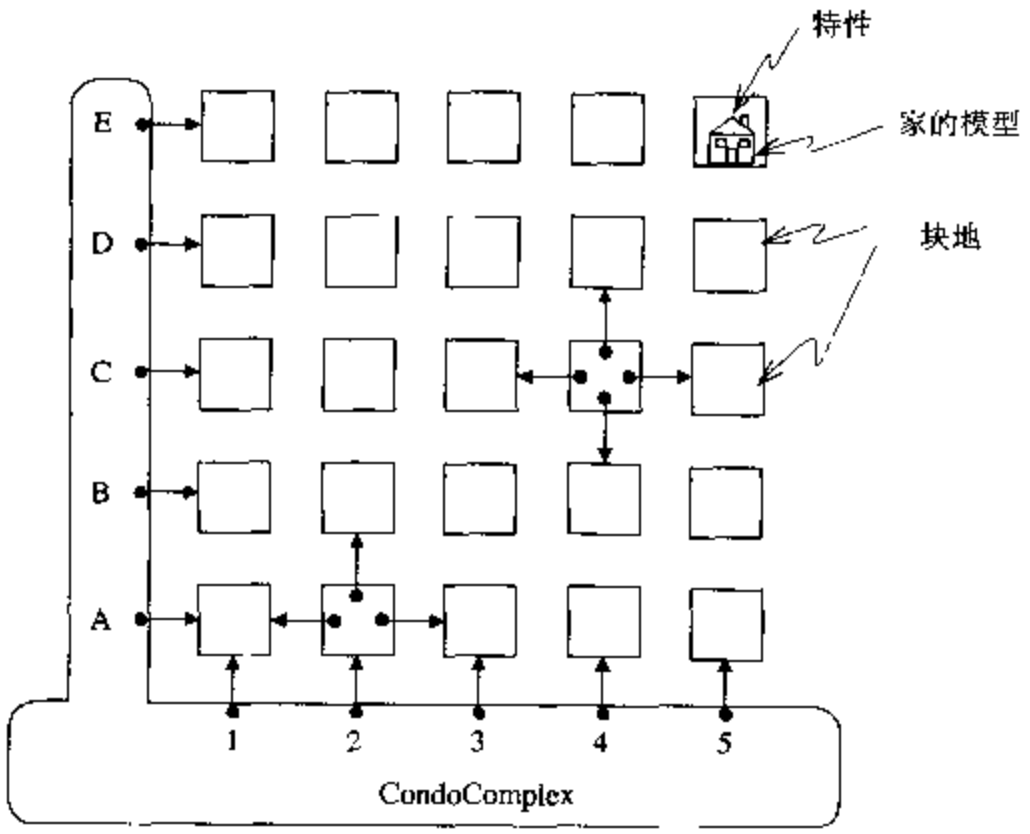


图 5-80 CondoComplex 例子中的 25 个地块的数组

① 有时候我们选择不去实现一个拷贝构造函数（例如，为一个迭代器）（甚至在这个操作可能有意义时）；但是，这个抽象本身有值语义。

每个 Lot 是一个单独的对象，它维护它自己的临近地块的列表，并由 CondoComplex 对象来管理。例如，Lot A2 拥有指向 Lot 对象 A1、B2 和 A3 的指针。一个 House 有值语义，因为拷贝构造对 House 有意义。换句话说，从 Lot 拷贝一个 House 给 Lot 是有意义的——也就是说，所有的房子可能看起来是完全一样的。

现在假设一个 Property 由 House 和它所坐落的 Lot 构成，并且 CondoComplex 对象管理着 Property 对象（而不再是 Lot 对象）的一个数组。一个 Property 也有值语义吗？答案是“没有”，因为我们不能拷贝一个房地产给另一个房地产。

如果我们试图把带有 Lot A2 的 Property 赋值给带有 Lot C4 的 Property，我们会破坏与 Lot C4 有关的邻近列表，并使更大的 CondoComplex 对象无效。所以我们不能像对一个 House 那样任意独立地拷贝一个 Property。因而 Property **没有**值语义。

虽然一个节点（由 Gnode 定义的）的网络部分没有值语义，但是由 Node 定义的那部分却可能有。从 C++的角度来看，这意味着有必要屏蔽（即声明为私有）Gnode 和 Gedge 的拷贝构造函数和赋值运算符，但 Node 和 Edge 各自都能够定义有意义的拷贝构造函数和赋值运算符，如图 5-81 所示。（graph 的完整接口如图 5-86 所示。）

```
void f(const Graph& g)
{
    const Gnode& a = g.node("Zurich");// fine - lvalue returned
    Gnode b = g.node("London");       // error - no value semantics
    Node& c = g.node("Paris");        // fine - modifiable lvalue returned
    Node d = g.node("Tokyo");         // fine - value semantics
    a = b;                            // error - no value semantics
    c = d;                            // fine - make Tokyo look like Paris
}
```

图 5-81 举例说明一幅图中的值语义

原 则

在循环物理依赖不可避免的地方，将其升级到尽可能高的层次可减少 CCD，甚至可以使循环能够由一个单个的、大小便于管理的组件代替。

我们的第二个分解机会来自于观察：为了正确地管理 Node 和 Edge，Graph 必须了解它分配的 Gnode 和 Gedge，这样当它被析构时，所有与该 graph 的节点和边相关的内存才可以被释放。而且，每个 Gnode 也必须了解它邻接的 Gedge 对象（只在名称上）。通过创建一个不透明指针的集合，我们可以从 graph 组件类中分解出所有这些功能。

袋（bag）是一种容器，不同于列表（list），它没有在它的元素上强加一个顺序，也不同于集合（set），它不要求元素是惟一的。因为袋的语义没有太多的规定，它的实现也就相当灵活。Graph 将维护一个 Node 指针的袋和一个 Edge 指针的袋。无论我们是否有一个有效的模板实现，我们都应该通过创建一个（类属）指针的袋来进一步分解这个问题。

图 5-82 显示了我们的分解过的一个类属指针袋的实现和专门的组件，这些组件利用这个类属容器来实现一个特定类型指针的袋。我们可以使用分层或私有继承来获得所需的特殊化和恢复单独的不透明指针的类型安全。模板是理想的，但有些实现可能在连接时间方面开销会非常大（正如在 10.4.1 节所讨论的）。为了纯粹实用，我们可能会被迫显式地表示专门的类型。无论哪一种实现，所有函数参数都通过 inline 函数转递给类属的 PtrBag 类，以避免由于传统函数调用而招致的任何额外开销。

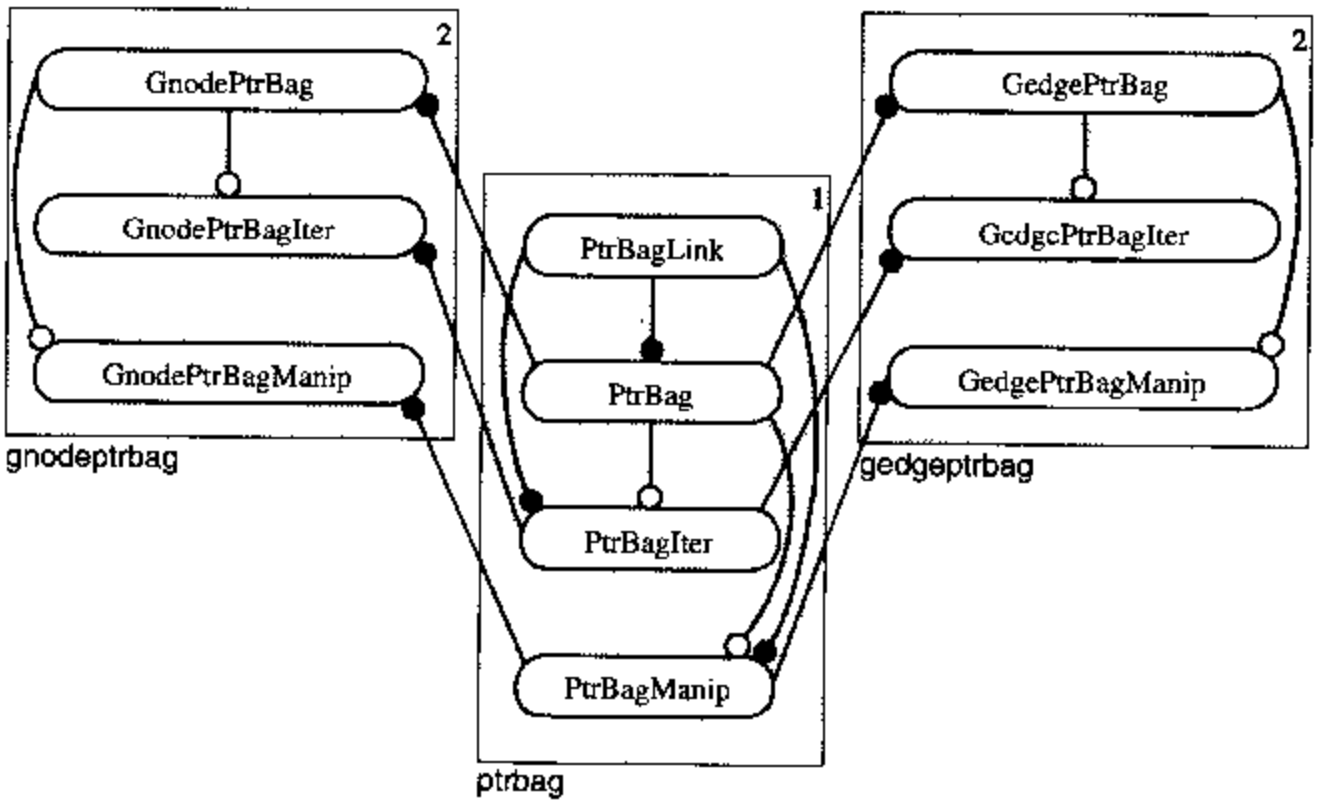


图 5-82 类属的 PtrBag 容器和特化

图 5-83 显示了由四个类组成的 ptrbag 组件的头文件。PtrBagLink 是一个低层次的实现类，它的使用是 ptrbag 组件中其他三个类的一个封装的实现细节。我们可以不再把 PtrBagLink 放在一个单独的组件里，而是把它完整地定义在 ptrbag.c 文件中，或者将它嵌套在 PtrBag 类中。（这些做法以及其他类似的可选择设计方案的优缺点将在 8.4 节比较和讨论。）

```
// ptrbag.h
#ifndef INCLUDED_PTRBAG
#define INCLUDED_PTRBAG

class PtrBagIter;
class PtrBagManip;

class PtrBagLink {
    void *d_pointer_p;
    PtrBagLink *d_next_p;

  private:
    PtrBagLink(const PtrBagLink&);
```

```
    PtrBagLink& operator=(const PtrBagLink&);

  public:
    PtrBagLink(void *pointer, PtrBagLink *next);
    ~PtrBagLink();
    PtrBagLink *&nextRef(); // used by manipulator
    PtrBagLink *next() const;
    void *pointer() const;
};

class PtrBag {
    PtrBagLink *d_root_p;
    friend PtrBagIter;
    friend PtrBagManip;

  private:
    PtrBag(const PtrBag&);
    PtrBag& operator=(const PtrBag&);

  public:
    PtrBag();
    ~PtrBag();
    void add(void *pointer);
    void removeAll(const void *pointer);
};

class PtrBagIter {
    PtrBagLink *d_link_p;

  private:
    PtrBagIter(const PtrBagIter&);
    PtrBagIter& operator=(const PtrBagIter&);

  public:
    PtrBagIter(const PtrBag& bag);
    ~PtrBagIter();
    void operator++();
    void *operator()() const;
    operator const void *() const;
};

class PtrBagManip {
    PtrBagLink **d_addrLink_p;

  private:
    PtrBagManip(const PtrBagManip&);
    PtrBagManip& operator=(const PtrBagManip&);

  public:
    PtrBagManip(PtrBag* bag);
    ~PtrBagManip();
```

```
    void advance();
    void remove();
    void *operator()() const;
    operator const void *() const;
};

// inline function definitions omitted

#endif
```

图 5-83 类属 ptrbag 组件的头文件

PtrBag 是一个用来保存类属指针的容器。我们给这个应用程序提供了一个冗余但方便的成员函数，用以从 PtrBag 中删除有指定值的所有指针。PtrBagIter 是一个指针袋的逻辑抽象的一部分，允许客户程序在袋上进行迭代，并以某种未指定的顺序返回它的内容。PtrBagManip 类似于 PtrBagIter，只是它允许其客户通过有选择地删除条目来修改袋——通过要求客户提供被操作容器的地址加入的一种功能。

在 ptrbag.h 中声明的大部分函数都可能内联实现，这只是因为它们内联产生的代码长度比一个函数调用产生的要小。少数函数是外联实现的，如图 5-84 所示。PtrBag 的析构器以及 removeAll 函数都有循环，使得它们成为内联的糟糕候选者。因为函数 add 访问全局自由存储，所以为了速度将其内联是没有用的。由足够代码组成的 remove 函数，调用一个函数将可能比在适当的地方代替源代码产生更少的对象代码。当 remove 函数调用增加了一些执行开销时，删除边就不太可能是一个频繁执行的功能。通过性能分析我们发现，remove 函数是这四个当中惟一一个有希望通过声明为内联来改进的函数。

```
// ptrbag.c
#include "ptrbag.h"

PtrBag::~PtrBag()
{
    PtrBagManip man(this);
    while (man) {
        man.remove();
    }
}

void PtrBag::add(void *pointer)
{
    d_root_p = new PtrBagLink(pointer, d_root_p);
}

void PtrBag::removeAll(const void *pointer)
{
    PtrBagManip man(this);
```

```
        while (man) {
            man() == pointer ? man.remove() : man.advance();
        }
    }

    void PtrBagManip::remove()
    {
        PtrBagLink *tmp = *d_addrLink_p;
        *(PtrBagLink **)d_addrLink_p = (*d_addrLink_p)->next();
        delete tmp;
    }
```

图 5-84　类属 ptrbag 组件的实现文件

这个新子系统的组件依赖图如图 5-85 所示。可以看到，所有从类的循环组提取出来的功能都隐藏在 graph 组件中。这个功能现在可以独立于该循环被测试和重用。甚至在 graph 组件内部也能以两种不同的方式重用 gedgeptrbag 中的功能：一种是在 Graph 类中跟踪所有的边，还有一种是在 Gnode 类中跟踪连接的边。至此，我们已经把循环依赖代码数量缩减到一个复杂度适合于一个单个组件 graph 的便于管理的层次。被 Node 或 Edge 标识的独立于 graph 的功能的复杂事务现在已被分离成了独立的组件，它们**在隔离的情况下**是可测试的。

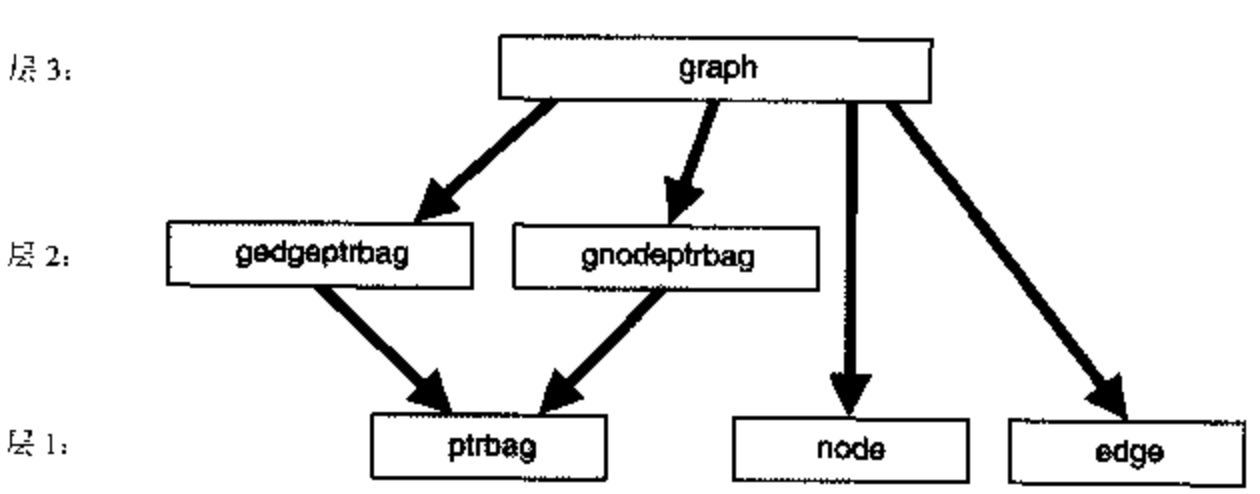


图 5-85　一个分解过的 graph 体系结构的组件依赖

图 5-86 给出了 graph 组件的完整的头文件。这个实现是高效、灵活和相当好维护的。但是这个组件并不那么易懂，因为一些接口（随同实现一起）已经被分解出来放在较低层次的可重用的组件中了。

```
// graph.h
#ifndef INCLUDED_GRAPH
#define INCLUDED_GRAPH

#ifndef INCLUDED_NODE
#include "node.h"
#endif

#ifndef INCLUDED_EDGE
```

```
#include "edge.h"
#endif

#ifndef INCLUDED_GNODEPTRBAG
#include "gnodeptrbag.h"
#endif

#ifndef INCLUDED_GEDGEPTRBAG
#include "gedgeptrbag.h"
#endif

class Graph;

class Gnode : public Node {
    GedgePtrBag d_edges;
    friend Graph;
    Gnode(const Gnode&);                          // not implemented
    Gnode& operator=(const Gnode&);               // not implemented

  private:
    Gnode(const char *name);
    ~Gnode();
    void add(Gedge *edgePtr);
    void remove(Gedge *edgePtr);

  public:
    const GedgePtrBag& edges() const;
};

class Gedge : public Edge {
    Gnode *d_from p;
    Gnode *d_to_p;
    friend Graph;
    Gedge(const Gedge&);                          // not implemented
    Gedge& operator=(const Gedge&);               // not implemented

  private:
    Gedge(Gnode *from, Gnode *to, double weight);
    ~Gedge();

  public:
    Gnode *from() const;
    Gnode *to() const;
};

class Graph {
    GnodePtrBag d_nodes;
    GedgePtrBag d_edges;
    Graph(const Graph&);
    Graph& operator=(const Graph&);

  public:
    Graph();
    ~Graph();

    Gnode *addNode(const char *nodeName);
```

```
    Gnode *findNode(const char *nodeName);
    void removeNode(Gnode *node);

    Gedge *addEdge(Gnode *from, Gnode *to, double weight);
    Gedge *findEdge(Gnode *from, Gnode *to);
    void removeEdge(Gedge *edge);

    const GnodePtrBag& nodes() const;
    const GedgePtrBag& edges() const;
};

#endif
```

图 5-86 定义了类 Gnode、Gedge 和 Graph 的 graph 组件的头文件

例如，假设你想在与 graph 中一特定节点相连的边上进行迭代，需要从 Gnode 获得 Gedge 指针的袋，然后用那个袋来构造 EdgePtrBagIter 的一个实例：

```
int sumOfEdgeWeights(const Gnode& gnode)
{
    int sum = 0;
    for (GedgePtrBagIter it(gnode.edges()); it; ++it) {
        sum += it()->weight();
    }
    return sum;
}
```

很方便地，同样的方法可用于从 graph 本身获取所有的边和节点，正如图 5-87 给出的 Graph 的输出运算符的实现所说明的那样。

```
ostream& operator<<(ostream& o, const Graph& graph)
{
    cout << "Graph: " << endl;
    GnodePtrBagIter nit(graph.nodes());
    if (nit) {
        o << "  Nodes:";
    }
    for (; nit; ++nit) {
        o << "  " << nit()->name();
    }
    const char *p = "  Edges:  ";
    const char *q = "          ";
    for (GedgePtrBagIter eit(graph.edges()); eit; ++eit) {
        o << endl << p << eit()->from()->name()
          << " ---(" << eit()->weight() << ")--> "
          << eit()->to()->name();
        p = q;
    }
    o << endl << "End Graph" << endl;
    return o;
}
```

图 5-87 Graph 中使用了低层次迭代器的一个运算符<<

图 5-88 给出了实现图 5-86 中的 graph 组件的一个测试驱动程序以及它的输出。注意，由 addNode 和 findNode 返回的 Gnode 指针都直接指向 Graph 内相应的 Gnode。Gnode 中的惟一公共可访问函数 edges()，提供了指向其 Gedge 指针袋的一个 const 引用，于是客户可以直接用它来遍历 graph。

Gedge 中惟一可利用的公共功能提供了对 Gedge 所连接的两个 Gnode 对象的访问。

```
// graph.t.c
#include "graph.h"
#include "gnodeptrbag.h"
#include "gedgeptrbag.h"
#include <iostream.h>

ostream& operator<<(ostream& o, const Graph& graph);

main()
{
    Graph g;

    {
        Gnode *n1 = g.addNode("Mindy");
        Gnode *n2 = g.addNode("Susan");
        Gnode *n3 = g.addNode("Rick");

        g.addEdge(n2, n1, 4);
        g.addEdge(n1, n3, 5);
        g.addEdge(n3, n2, 1);

        g.addNode("Franklin");
        g.addNode("Cathy");
    }

    g.addEdge(g.findNode("Susan"), g.findNode("Franklin"), 6);
    g.addEdge(g.findNode("Rick"), g.findNode("Franklin"), 2);
    g.addEdge(g.findNode("Rick"), g.findNode("Cathy"), 3);

    cout << g;
}

// Output:
        john@john: a.out
        Graph:
          Nodes:  Cathy  Franklin  Rick  Susan  Mindy
          Edges:  Rick ---(3)--> Cathy
                  Rick ---(2)--> Franklin
                  Susan ---(6)--> Franklin
                  Rick ---(1)--> Susan
                  Mindy ---(5)--> Rick
```

```
                Susan ---(4)--> Mindy
End Graph
john@john:
```

图 5-88　举例说明 graph 组件用法的简单测试驱动程序

原　则

授权友元关系不会产生依赖，但是为了保持封装可能会引起物理耦合。

在 Graph 的这个实现中，通过（局部）友元关系对 Gnode 和 Gedge 的私有访问对保持封装是非常重要的。这个设计通过在物理上联合系统中那些需要通过私有访问共享通用的实现细节的部分，排除了与远距离友元关系有关的问题。换句话说，通过把 Graph、Gnode 和 Gedge 结合在一个单个的组件内，所需的友元关系不再是远距离的了。

如图 5-89 所示，出现了 Gnode 和 Gedge 只在名称上彼此依赖，而没有对 Graph 的向后依赖的情况。虽然这二个组件没有循环相互依赖，但仍然有分解的必要。这个子系统的客户程序需要直接和 Gnode 和 Gedge 交互。使 Gnode 或 Gedge 的整个接口成为公共的，将会使客户接触 graph 组件的实现细节。更糟糕的是，这样做将允许客户违反由 Graph 管理类强制的重要策略。

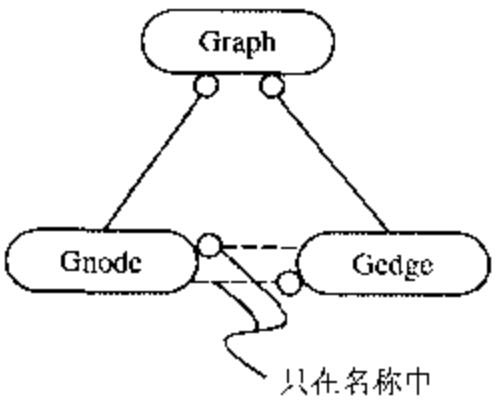


图 5-89　Graph、Gnode 和 Gedge 之间的实际关系

例如，使 Gedge 构造函数成为公共的将允许客户绕过 Graph 对象，在程序堆栈中创建 Gedge 的实例。没有什么可以阻止一个任性的客户将一个在程序堆栈中创建的 Gedge 加入到一个属于另一个合法 Graph 的合法 Gnode 中。

为了避免这些问题，有必要允许 Graph 类有权访问定义在 Gnode 和 Gedge 中的私有功能。然后为了避免远距离友元关系，我们被迫把这些密切依赖的类放在同一个组件中。虽然授权友元关系不会导致直接的物理依赖，但模块性和封装决定了如图 5-90 所示的有效的物理耦合。

物理耦合只因友元关系所致（正如在 3.6 节所讨论的）以及不强烈的物理依赖为另一技术提供了机会的事实，我们将在下一节中研究。为了保持完整性，图 5-91 中提示了 graph 组件的实现文件。

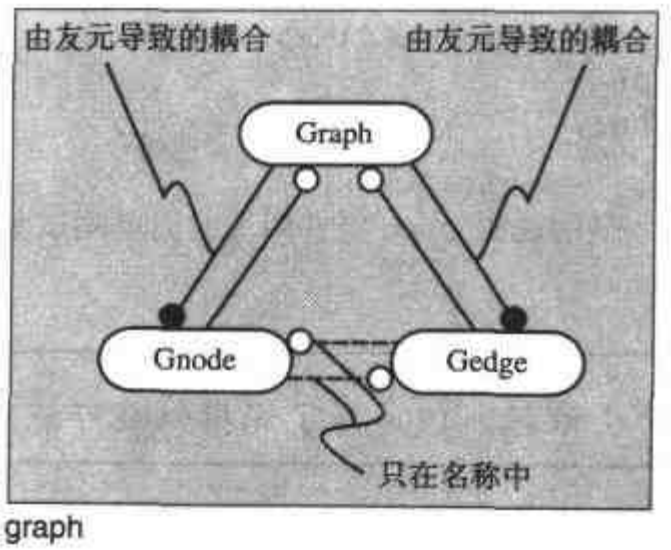


图 5-90 隐含的物理耦合避免远距离友元关系

```
// graph.c
#include "graph3.h"
#include <string.h>

                // -*-*-*-*- class Gnode -*-*-*-*-

Gnode::Gnode(const char *name) : Node(name) {}

Gnode::~Gnode() {}

void Gnode::add(Gedge *edgePtr) { d_edges.add(edgePtr); }

void Gnode::remove(Gedge *edgePtr) { d_edges.removeAll(edgePtr); }

const GedgePtrBag& Gnode::edges() const { return d_edges; }

                // -*-*-*-*- class Gedge -*-*-*-*-

Gedge::Gedge(Gnode *from, Gnode *to, double weight)
: Edge(weight)
, d_from_p(from)
, d_to_p(to) {}

Gedge::~Gedge() {}

Gnode *Gedge::from() const { return d_from_p; }

Gnode *Gedge::to() const { return d_to_p; }

                // -*-*-*-*- class Graph -*-*-*-*-

Graph::Graph() {}

Graph::~Graph()
{
    for (GedgePtrBagIter eit(d_edges); eit; ++eit) {
        delete eit();
    }
    for (GnodePtrBagIter nit(d_nodes); nit; ++nit) {
```

```
            delete nit();
        }
    }

    Gnode *Graph::addNode(const char *nodeName)
    {
        Gnode *p = new Gnode(nodeName);
        d_nodes.add(p);
        return p;
    }

    Gnode *Graph::findNode(const char *nodeName)
    {
        for (GnodePtrBagIter it(d_nodes); it; ++it) {
            if (0 == strcmp(it()->name(), nodeName)) {
                return it();
            }
        }
        return 0;
    }

    void Graph::removeNode(Gnode *node)
    {
        GnodePtrBagManip nodeMan(&d_nodes);
        while (nodeMan) {
            if (nodeMan() == node) {
                for (GedgePtrBagIter it(nodeMan()->edges()); it; ++it) {
                    d_edges.removeAll(it());
                }
                nodeMan.remove();
            }
            else {
                nodeMan.advance();
            }
        }
    }

    Gedge *Graph::addEdge(Gnode *from, Gnode *to, double weight)
    {
        Gedge *p = new Gedge(from, to, weight);
        d_edges.add(p);
        from->add(p);
        to->add(p);
        return p;
    }

    Gedge *Graph::findEdge(Gnode *from, Gnode *to)
    {
        for (GedgePtrBagIter it(d_edges); it; ++it) {
            if (it()->from() == from && it()->to() == to) {
                return it();
            }
        }
```

```
    return 0;
}

void Graph::removeEdge(Gedge *edge)
{
    GedgePtrBagManip edgeMan(&d_edges);
    while (edgeMan) {
        if (edgeMan() == edge) {
            edge->to()->remove(edge);
            edge->from()->remove(edge);
            edgeMan.remove();
        }
        else {
            edgeMan.advance();
        }
    }
}

const GnodePtrBag& Graph::nodes() const { return d_nodes; }

const GedgePtrBag& Graph::edges() const { return d_edges; }
```

图 5-91 graph 组件的 graph.c 实现文件

总结：分解是一种通用技术，可用来减少有着固有循环依赖的设计的维护开销。通过将一些实现复杂事务重新安装到较低层次组件中，可以独立于余下的循环相互依赖代码对该功能进行测试（并可能重用）。一般的分解会得到更灵活的体系结构，又不会牺牲运行时效率。然而，在分解一个子系统的接口时，客户可能被要求使用了系统层次结构中较低层次的组件接口。

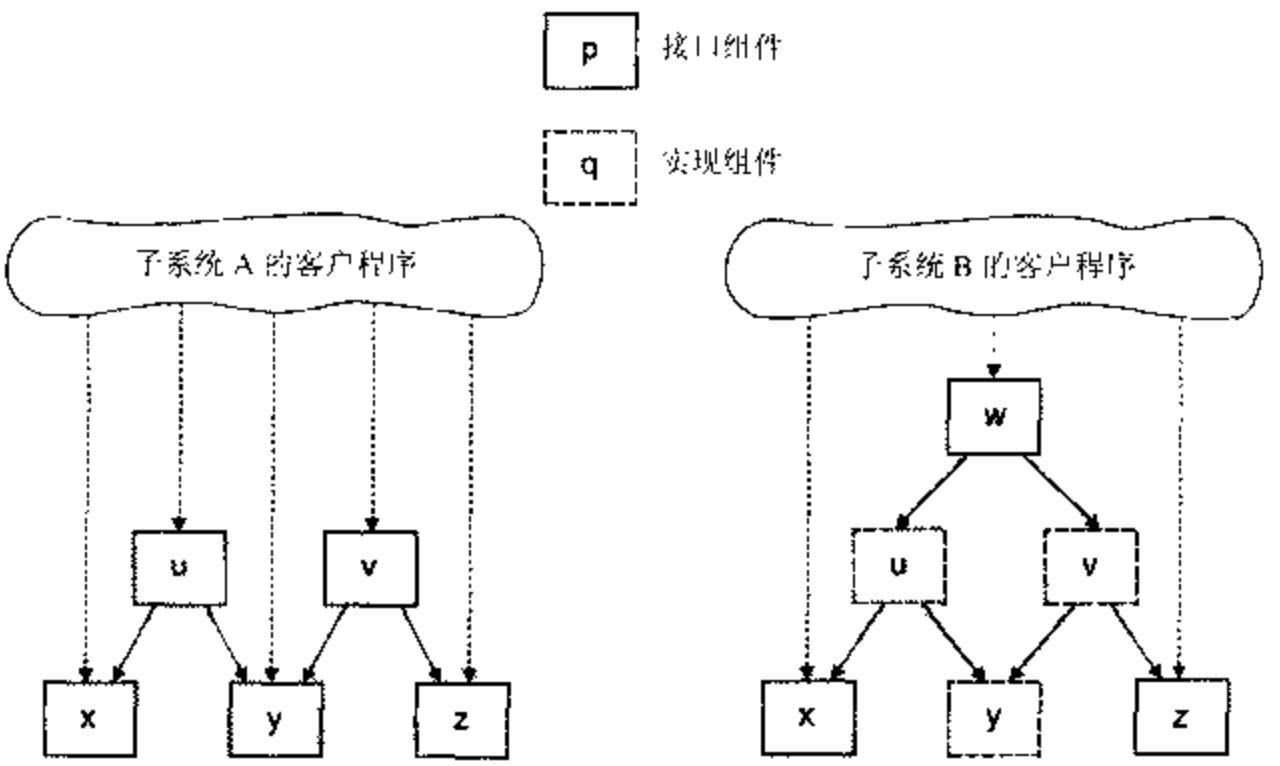
5.10 升级封装

作为一个 C++程序员，你无疑应该了解**封装**的概念。如果一个接口使其实现细节对客户是不可编程访问的，这个接口就是封装的。一个常见的误解是，每个单独的类或组件都有必要封装所有的实现细节，向整个世界展示一个健壮的接口。这样做将使大型、复杂的子系统变得更大、更慢和更复杂，这是不能容忍的。我们可以改为，将大量有用的低层次类藏在一个单个组件的接口后面（如同第 4 章中有 p2p_router 组件的例子那样）。我们会经常将这样的组件称为**包装器（wrapper）**[①]。

图 5-92（a）举例说明了一个子系统，其中的每个单独的组件都展示了公共接口，适于客户在利用那个子系统的上下文中直接使用。这个子系统的封装是按一种“每个组件”的原则进行的。图 5-92（b）也显示了一个子系统，定义在该子系统内的一些组件并不像由包装器组

① 包装器是一种称为 Facade（外表）设计模式（**gamma**，第 4 章，185~193 页）的基于组件的实现。

件 w 定义的组件那样暴露在总的子系统接口中。也就是说，定义在组件 u、v 或 y 中的类型没有哪个是 w 的公共或保护接口的一部分。虽然组件 u、v 和 y 可以单独地被任何人使用，但是没有任何编程的方法可以（甚至只是）检测这些组件是否被用来实现 w。因此，当和该子系统的实例进行交互时，也没有任何编程方法可以用来对定义在这些组件中的对象进行任何利用。子系统 B 的封装由处在该子系统最高层的包装器组件执行。



（a）“每个组件”封装　　（b）子系统级封装

图 5-92　封装的范围

原　则

什么是和什么不是实现细节取决于物理层次结构内部的抽象级别。

根据类推原则，一个 SparkPlug 是 Car 的实现细节，但是 SparkPlug 被用在 Engine（它被 Car 封装）的接口中。在 Car 的子系统内部的抽象级别上，SparkPlug 是构成 Car 的实现的组件层次结构的公共接口的一部分。在 Car 的客户程序的抽象级别上，SparkPlug 是隐藏的。

当我们在子系统的实现中使用一个低层次库组件（如 qsort）时，我们不会两次考虑怎样才能使组件在客户手里合适。我们是否会恰巧使用 qsort 仍然是我们子系统的一个封装的实现细节。我们不能阻止客户在他们自己那里使用 qsort；但是，无论我们自己是否使用了它，我们都能很容易隐藏这一细节。同样的原理可应用于定义在我们的子系统内的组件。

假设图 5-92（b）中的组件 y 定义了 5.7 节的 OrderedPointCollection。我们子系统的客户可能完全不需要有序的点集合，但这个组件被子系统内的其他组件用来实现更高层次的功能。在一个子系统的较低层次中，组件将相应地交换较低层次信息。这种信息，虽然对最终用户

来说是一个实现细节，对低层次组件的接口来说却是定义良好、可预测和适当的。

我们可以试着通过使 OrderedPointCollection 的所有接口功能私有，并授权特定的、较高层次的组件（例如 u 和 v）友元状态来隐藏它——但是为什么要让问题变复杂呢？客户可以利用 OrderedPointCollection 的定义，这样做并无害处，只要这个类型没有用在那些定义了系统总接口的组件的接口上[①]。

显示在图 5-92（a）中的子系统，在结构上和前一节介绍的 graph 子系统的分解后的实现是相似的。在那个体系结构中（图 5-85），在使用子系统的正常过程中客户被要求使用较低层次的组件（例如 ptrbag）。

graph 的一个可替代实现将提供一个包装器组件，graph 子系统的所有客户为了使用该子系统都必须通过它来交互。这个包装器[像图 5-92（b）中的 w 组件一样]不仅要管理 graph 子系统中的其他组件，还要封装先前在分解实现中暴露给用户的几个实现级对象。

回想一下，在 graph 子系统的分解实现中，Gnode 和 Gedge 都由 Graph 管理，意味着 Graph 独自就可以创建和析构 Gnode 和 Gedge 对象。在那个实现中，Gnode 和 Gedge 都不是子系统的封装细节；这些类型的实例，构成了 graph 的实现，它们很容易通过 Graph 类本身的接口被访问。为了阻止客户侵占管理类的权限，接口的许多部分对 Gnode 和 Gedge 都声明为 private（私有的），并且 Graph 单独被授予友元状态。仅仅是为了避免可能因远距离友元关系导致的封装裂口，我们被迫将 Graph、Gnode 和 Gedge 放在一个单个的组件内。

原　则

将封装所在的层次升级，能够消除对一个子系统内协同操作的组件授予私有访问权的需求。

总是强求在一个单个的组件内进行有特权的通信，可能会使组件大得可笑，并会失去分层次设计的优势。如果我们暂停考虑封装方面的问题，并且让 Gnode 和 Gedge 的所有功能都是公共的，我们可能会把这二个类的每一个都移到单独的组件中。因为 Gnode 和 Gedge 互相只在名称上使用，它们自动变得可独立于彼此测试。例如，现在对显示在图 5-93 中的新组件 Gnode 的全部功能进行直接测试是很容易、很方便的。

在分解解决方案中，只有 Graph 有对 Gnode 的私有访问权，并且对两个类被定义在同一个组件中。这种方法消除了客户不正当直接使用 Gnode 的可能，但是它也妨碍了 Gnode 的直接测试。

在这个新的方法下，不再是被迫通过 Graph 的接口来间接测试 Gnode 的低层次功能了（例如，增加和删除 Gedge 指针），现在测试工程师有可能直接检验这个当前公共的性能。但是，普通客户现在也将拥有对这个低层次功能的直接访问权。

① 如果一个实现类提供的函数改变了该类或.c 文件内的静态变量，这个原则可能不成立。

```
// gnode.h
#ifndef INCLUDED_GNODE
#define INCLUDED_GNODE

#ifndef INCLUDED_NODE
#include "node.h"
#endif

#ifndef INCLUDED_GEDGEPTRBAG
#include "gedgeptrbag.h"
#endif

class Gnode : public Node {
    GedgePtrBag d_edges;
    Gnode(const Gnode&);                    // not implemented
    Gnode& operator=(const Gnode&);         // not implemented

  public:
    Gnode(const char *name);
    ~Gnode();
    void add(Gedge *edgePtr);
    void remove(Gedge *edgePtr);
    const GedgePtrBag& edges() const;
};

#endif
```

图 5-93　新的定义 Gnode 类的独立 gnode 组件

最初，我们授予 Graph 对 Gnode 和 Gedge 的私有访问权以保持封装。可是封装却处在危险之中，因为 Graph 的客户被授权直接访问 Gnode 和 Gedge 对象，这两个对象本身大部分是 Graph 的实现细节。如果我们中止在 Graph 的接口上暴露 Gnode 和 Gedge，就可以完全避免这个问题。

原　则

私有的头文件不是适当的封装替代品，因为它们禁止并排（side-by-side）重用。

不公开头文件并不是解决办法——那是骗人的。不给客户授权访问一个或多个头文件将使某些类型的使用不透明，但是这些类型在名称上仍然是可编程访问的，因而不是封装的细节。例如，从系统的一部分获取的一个不透明指针，可能会出乎意料地被客户程序以一种会导致系统内部出现不一致的方式重新引进该系统的另一部分。

如图 5-94 所示，类 N 管理着类型为 W 和 E 的对象的一个集合。这两个子对象每一个都在它的接口中使用了一个 S 对象。S 对象本身是一个实现细节，因此客户在封装它的使用的过程中会被拒绝访问它的头文件。

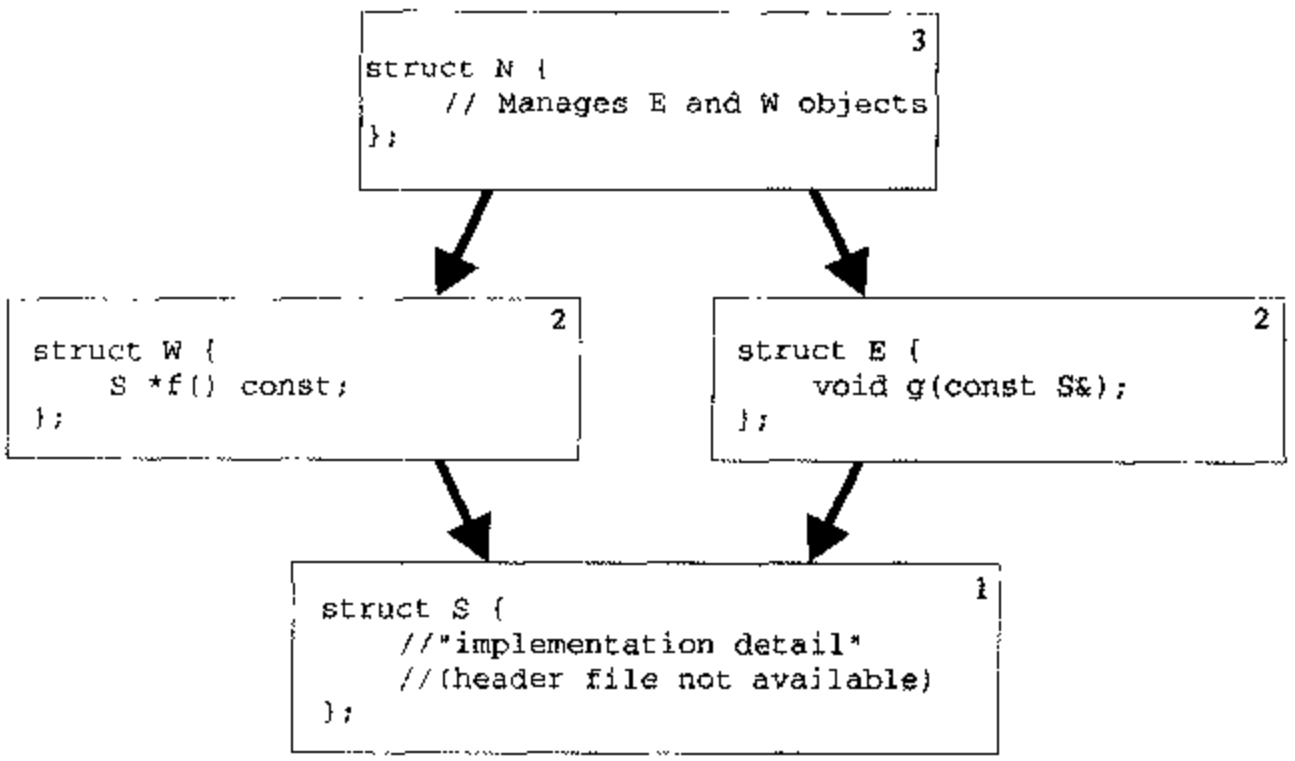


图 5-94 通过隐藏头文件的无效封装

请注意，一个客户从类 W 的实例中提取一个不透明 S 指针、并用它来直接影响类 E 的实例是多么的容易：

```
void myFunc(E *e, const W& w)
{
    e->g(*w.f());
}
```

把这种方法和适当将其实现细节隐藏在一个封装接口之后的设计（即，没有实现类型暴露在那个子系统的包装器组件的逻辑接口上的设计）相比较。即使可以访问所有的头文件，也仍然无法通过编程来访问隐藏在包装器的真正封装接口后的低层次实现对象。

适当封装的好处很多，一个明显的例子是重用。试图通过抑制一个头文件来封装一个实现类型，会极大地妨碍那个组件的公共重用。如果封装做得适当，客户可以并排地访问低层次类型和在内部使用它们的子系统，不用害怕会暴露子系统的私有细节。

可理解性和可维护性是适当封装的另外两个好处。区别哪个是和哪个不是“私有”头文件是最困难的。由于这些原因，我们极力推荐每个组件提供一个单个的头文件，并在其中清楚而完整地定义它的（单独的）接口。当目标不是封装的而是绝缘的时候，抑制头文件也许是适当的，这一点值得注意。

> **定义**：在层次系统中，**封装一个类型**（该类型定义在头文件内的文件作用域中）意味着隐藏了它的使用而不是隐藏了类型本身。

现在让我们返回到 graph 实例。对测试工程师和/或客户隐藏子系统的低层次实现类型并不能成功地对这个新 graph 体系结构进行层次化。其他人对他们自己的这些类型的实例做什

么并不会有什么不同。更确切地说，这个体系结构的成功层次化要通过“确保没有编程的方法可用来访问作为子系统实例一部分的任何实现类型的任何实例”的方法来获得。

为了在子系统级实现封装，我们需要引进一个包装器组件。图 5-95 给出了 graph 子系统新体系结构的一幅详细草图。旧的 Graph 类已经改名为 GraphImp，但其他方面未受本质影响。Gnode 和 GraphImp 都继续使用 ptrbag（在本图中显示为一个单个组件）中的分解实现和特化。新的 graph（包装器）组件现在定义了五个类供子系统的客户使用。

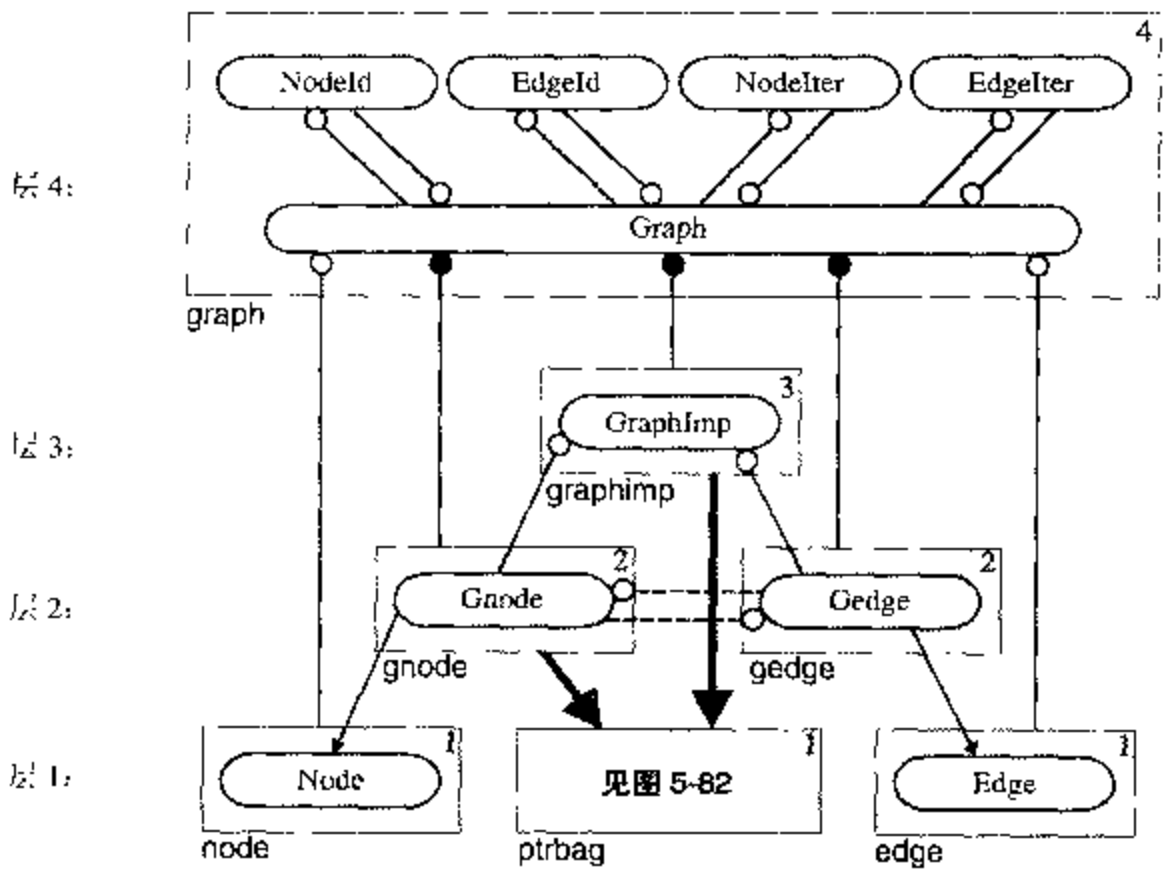


图 5-95 用一个包装器组件升级封装

原 则

包装器组件可以用来封装一个子系统内的实现类型的使用，但允许其他类型通过它的接口。

对于只包含网络独立数据的 Node 和 Edge 类，可以从新的 graph 组件的接口进行编程访问。但是，从图子系统的用户的角度来看，类型 Gnode、Gedge、GraphImp 以及所有定义在 ptrbag 中的类型现在都是实现细节，它们完全被新的包装器组件封装了。

为了评估这个解决方案，考虑一个有权访问 gnode.h 的客户程序，它仍然不能影响通过 graph 组件的接口已经创建的任何 Gnode。当然，用户仍然可以自由地创建和操纵他（或她）的**独立的** Gnode 实例（即，为了测试目的）。

图 5-96 显示了新图子系统的包装器组件的头文件。四个附加的支持类（*NodeId*、*EdgeId*、NodeIter 和 EdgeIter）建立了封装，并且要么提供要么需要对 Graph 的私有访问权。因此，所

有这些类都必须和 Graph 驻留在同一组件中，以避免远距离友元关系。

因为这个包装器只是为封装目的而设计的，它是一个非常薄的包装器，全部的功能只是转递对较低层次实现组件的请求。为了避免额外的函数调用开销，所有的包装器函数都定义为内联，而让 graph.c 文件空着。

```
// graph.h
#ifndef INCLUDED_GRAPH
#define INCLUDED_GRAPH

#ifndef INCLUDED_GRAPHIMP
#include "graphimp.h"
#endif

#ifndef INCLUDED_GNODE
#include "gnode.h"
#endif

#ifndef INCLUDED_GEDGE
#include "gedge.h"
#endif

class EdgeId;                   // forward declaration
class Graph;                    // forward declaration
class NodeIter;                 // forward declaration
class EdgeIter;                 // forward declaration

class NodeId {
    Gnode *d_node_p;
    friend EdgeId;
    friend Graph;
    friend NodeIter;
    friend EdgeIter;

private:
  NodeId(Gnode *node) : d_node_p(node) {}
  Gnode *gnode() const { return d_node_p; }
public:
  NodeId() : d_node_p(0) {}
  NodeId(const NodeId& nid) : d_node_p(nid.d_node_p) {}
  ~NodeId() {}
  NodeId& operator=(const NodeId& nid) { d_node_p = nid.d_node_p; return *this; }
  operator Node *() const { return d_node_p; }
  Node *operator->() const { return *this; }
};

class EdgeId {
    Gedge *d_edge_p;
    friend Graph;
    friend EdgeIter;
```

```
  private:
    EdgeId(Gedge *edge) : d_edge_p(edge) {}
    Gedge *gedge() const { return d_edge_p; }

  public:
    EdgeId() : d_edge_p(0) {}
    EdgeId(const EdgeId& eid) : d_edge_p(eid.d_edge_p) {}
    ~EdgeId() {}
    EdgeId& operator=(const EdgeId& eid) { d_edge_p = eid.d_edge_p; return *this; }
    NodeId from() const { return NodeId(d_edge_p->from()); }
    NodeId to() const { return NodeId(d_edge_p->to()); }
    operator Edge *() const { return d_edge_p; }
    Edge *operator->() const { return *this; }
};

class Graph {
    GraphImp d_imp;
    friend NodeIter;
    friend EdgeIter;

  private:
    Graph(const Graph&);                          // not implemented
    Graph& operator=(const Graph&);               // not implemented

public:
  Graph() {}
  ~Graph() {}
  NodeId addNode(const char *nodeName)
  {
      return NodeId(d_imp.addNode(nodeName));
  }
  NodeId findNode(const char *nodeName)
  {
      return NodeId(d_imp.findNode(nodeName));
  }

  void removeNode(const NodeId& nid)
  {
      d_imp.removeNode(nid.gnode());
  }
  EdgeId addEdge(const NodeId& from, const NodeId& to, double weight)
  {
      return EdgeId(d_imp.addEdge(from.gnode(), to.gnode(), weight));
  }
  EdgeId findEdge(const NodeId& from, const NodeId& to)
  {
      return EdgeId(d_imp.findEdge(from.gnode(), to.gnode()));
  }
  void removeEdge(const EdgeId& eid)
  {
```

```
        d_imp.removeEdge(eid.gedge());
    }
};

class NodeIter {
    GnodePtrBagIter d_iter;

  private:
    NodeIter(const NodeIter&);                    // not implemented
    NodeIter& operator=(const NodeIter&);         // not implemented

  public:
    NodeIter(const Graph& graph) : d_iter(graph.d_imp.nodes()) {}
    void operator++() { ++d_iter; }
    operator const void *() const { return d_iter; }
    NodeId operator()() const { return NodeId(d_iter()); }
};

class EdgeIter {
    GedgePtrBagIter d_iter;

  private
    EdgeIter(const EdgeIter&);                    // not implemented
    EdgeIter& operator=(const EdgeIter&);         // not implemented

  public:
    EdgeIter(const Graph& graph) : d_iter(graph.d_imp.edges()) {}
    EdgeIter(const NodeId& nid) : d_iter(nid.gnode()->edges()) {}
    void operator++() { ++d_iter; }
    operator const void *() const { return d_iter; }
    EdgeId operator()() const { return EdgeId(d_iter()); }
};

#endif
```

图 5-96 graph 子系统的封装包装器组件

注意，在这个接口中对任何 Gnode 和 Gedge 都没有直接访问权。增加或查找一个节点会返回一个类型 NodeId 的替代对象，它拥有一个指向一个 Gnode 的指针，但是在任何情况下，NodeId 都不会让客户有访问超出它所拥有的 Gnode 的 Node 部分的权限。

类似地，Gedge 也不再是暴露的。指向 Gedge 的指针现在被类型 EdgeId 的实例所代替。在处理 Graph 时，你可以像使用 Gedge 指针那样使用 EdgeId。在传送 Edge 信息时，你可以把一个 EdgeId 当作一个 Edge 指针来用——仅此而已。

将旧的测试驱动程序修改成适应新的包装器接口，只需要做一点小改动。具体说来，就是用 NodeId 类型代替（Gnode *）类型，删除一些不必要的#include 指令。输出当然也是一样的。修改后的测试驱动程序如图 5-97 所示。

```
// graph.t.c
#include "graph.h"
#include <iostream.h>

ostream& operator<<(ostream& o, const Graph& graph);

main()
{
    Graph g;

    {
        NodeId n1 = g.addNode("Mindy");
        NodeId n2 = g.addNode("Susan");
        NodeId n3 = g.addNode("Rick");

        g.addEdge(n2, n1, 4);
        g.addEdge(n1, n3, 5);
        g.addEdge(n3, n2, 1);

        g.addNode("Franklin");
        g.addNode("Cathy");
    }
        g.addEdge(g.findNode("Susan"), g.findNode("Franklin"), 6);
        g.addEdge(g.findNode("Rick"), g.findNode("Franklin"), 2);
        g.addEdge(g.findNode("Rick"), g.findNode("Cathy"), 3);

        cout << g;
}
```

图 5-97 举例说明新的 graph 包装器组件用法的驱动程序

包装器接口的使用在某些方面比一个分解实现还要简单，因为即使不是全部，大部分可用功能都展示在一个单一的、整体的头文件中。例如，要在一个图或节点中的边之上进行迭代，我们只需要看 graph 的头文件本身：

```
int sumOfEdgeWeights(const NodeId& nid)
{
    int sum = 0;
    for (EdgeIter it(nid); it; ++it) {
        sum += it()->weight();
    }
}
```

包装（wrapping）也有缺点，它使接口更不灵活，使遍历它的通信更慢。一个被包装的子系统开始开发时也可能开销更大。但是，包装可能是使包含有许多高度互相依赖组件的子系统完成层次化和封装的惟一真正有效的途径。

对于5.1.3节图5-10中的两个组件的简单例子我们已经讨论了很多，但是，图5-95中的七个组件为创作一个复杂但容易使用和高度可靠的子系统铺设了坚固的、层次化的基础。我们在6.4.3节中会继续包装器的话题，会讨论如何把客户程序与包装器组件下面的实现类型的编译时依赖绝缘。

总结：试图按“每个组件”的原则来封装一个子系统的实现，可能会妨碍低层次的通信和/或破坏一个其他的可行设计。比限制单独类中的客户可访问功能更好的是，我们可以限制在总的子系统接口中暴露给用户类的子集。通过使用一个包装器组件，我们可以将封装的层次升级到子系统的最高层，这样做可以消除对低层次友元关系的需要，从而也消除了将紧密耦合的类合并成一个单个的超大型组件的必要。

5.11 小结

通过考虑逻辑设计的物理隐含和前摄地将我们的系统设计为一个可层次化的组件集合，我们创建了可独立于设计的其他部分来理解、测试和重用的模块化抽象的一个层次结构。

实现层次化的技术包括：

- 升级　　将相互依赖功能在物理层次结构中提高。
- 降级　　将共有功能在物理层次结构中降低。
- 不透明指针　　让一个对象只在名称上使用另一个对象。
- 哑数据　　使用表示对同层对象的一个依赖的数据，但只在单独的、较高层对象的上下文中。
- 冗余　　通过重复少量的代码或数据避免耦合来故意避免重用。
- 回调　　使用客户提供的函数，这些函数可以使较低层次的子系统能够在更全局的上下文中执行特定任务。
- 管理类　　建立一个拥有和协调较低层次对象的类。
- 分解　　将独立可测试子行为从涉及过度物理依赖的复杂组件的实现中移出来。
- 升级封装　　将实现细节对客户隐藏的地点移到物理层次结构的更高层。

使用这些技术来创建可层次化的设计，趋向于减小大型的、有时甚至是势不可挡的逻辑设计空间，并且有助于将开发人员引导到更主流、更可维护的体系结构的方向上去。幸运的是，在好的逻辑设计和好的物理设计之间有一种偶然发现的协同作用。假以时日，这两个设计目标将会互相加强。

6 绝缘

好的物理设计的另一个重要方面是避免不必要的编译时依赖。过度的编译时耦合会严重地削弱我们维护一个系统的能力。一般来说，对驻留在一个组件的物理接口中的、通过编程不能访问的实现细节进行修改将强迫所有的客户程序重新编译。甚至对于一定规模的大型项目，重新编译整个系统的开销也将抑制对低层次组件的物理接口的修改，甚至限制我们对它们的实现封装细节作局部的修改。

本章介绍的物理过程在本书中称为**绝缘**。该过程与通常称为“封装”的过程类似。绝缘是指避免或消除不必要的编译时耦合的过程。

首先我们确立了这样一个目标：将绝缘作为整个体系结构设计的一部分来研究，并提供了理论上和经验上的论证。接着，我们将讨论许多特定的、可能引起编译时依赖而没有试图去减轻这种依赖的 C++结构，在 6.3 节，我们将讨论几种用于对暴露的实现细节进行绝缘的技术（通过以下机制）：

- 私有基类
- 嵌入成员数据
- 私有成员函数
- 保护成员函数
- 枚举类型
- 编译器产生的函数
- 包含指令
- 私有成员数据
- 默认参数

在 6.4 节，我们将讨论对实现的所有细节进行绝缘的大规模技术：

- 协议类

- 完全绝缘的具体类
- 绝缘包装器组件器

绝缘非常大型的子系统为开发者提出了一个独特的问题。在 6.5 节，我们将探讨为一个超大型 C++系统实现一个遵从 ANSI C 的过程接口。

最后，在 6.6 节，我们探讨需要绝缘的条件，介绍与绝缘相关的运行时开销，以及不适合绝缘的特定条件。我们还将举例说明应用绝缘的过程，并且定量分析与各种程度绝缘相关的运行时开销。

6.1 从封装到绝缘

绝缘是一个物理设计问题，它的逻辑相似物一般称为封装。在 2.2 节中，我们基于类讨论了封装。在 3.6 节中，我们基于组件讨论了封装。然后在 5.10 节中，我们基于在一个层次结构子系统头文件的文件作用域中定义的类讨论了封装。在每种情况下的重要方面如下：

（1）某个细节是某个实体的一部分。

（2）从该实体的接口不能编程访问到该细节。

考虑图 6-1 中显示的组件 stack 的头文件。Stack 类的逻辑接口完全封装了它的实现。从程序的角度看，没有办法将该实现与图 6-2 中显示的有相同接口的链表实现区别开来。

```
// stack.h
#ifrdef INCLUDFD_STACK
#dcfine !NCLUDED_STACK

class Stack {
    int *d_stack_p;
    int d_size;
    int d_length;

  public:
    Stack();
    Stack(const Stack &stack);
    ~Stack();
    Stack& operator=(const Stack &stack);
    void push(int value);
    int pop();
    int top() const;
    int isEmpty() const;
};

#endif
```

图 6-1 完全封装的基于数组的 Stack 实现

```
// stack.h
#ifndef INCLUDED_STACK
#define INCLUDED_STACK

class StackLink;

class Stack {
    StackLink *d_stack_p;

  public:
    Stack();
    Stack(const Stack &stack);
    ~Stack();
    Stack& operator=(const Stack &stack);
    void push(int value);
    int pop();
    int top() const;
    int isEmpty() const;
};

#endif
```

图 6-2　完全封装的基于链表的 Stack 实现

即使两个 Stack 类都完全封装了它们的实现，任何有经验的 C++程序员看到这些头文件都可以立即决定这些组件的一般实现策略。这些 stack 组件的每个头文件都说明了甚至使用封装接口来隐藏专有实现也是存在困难的。内联函数由于将算法的细节暴露给客户，还会使问题更严重。

但是对大型项目来说，保持组件实现的专有性不是最重要的问题。客户程序有权要求一个组件的逻辑接口保持不变，在理想情况下，对一个组件的逻辑实现的修改不应该影响客户程序。但是，在现实中，C++编译器依赖于一个头文件中的所有信息，包括私有的数据。如果一个人通过检查头文件来决定一个组件的实现策略，那么如果那个组件的实现策略发生了改变，该组件的客户程序很可能被迫重新编译。

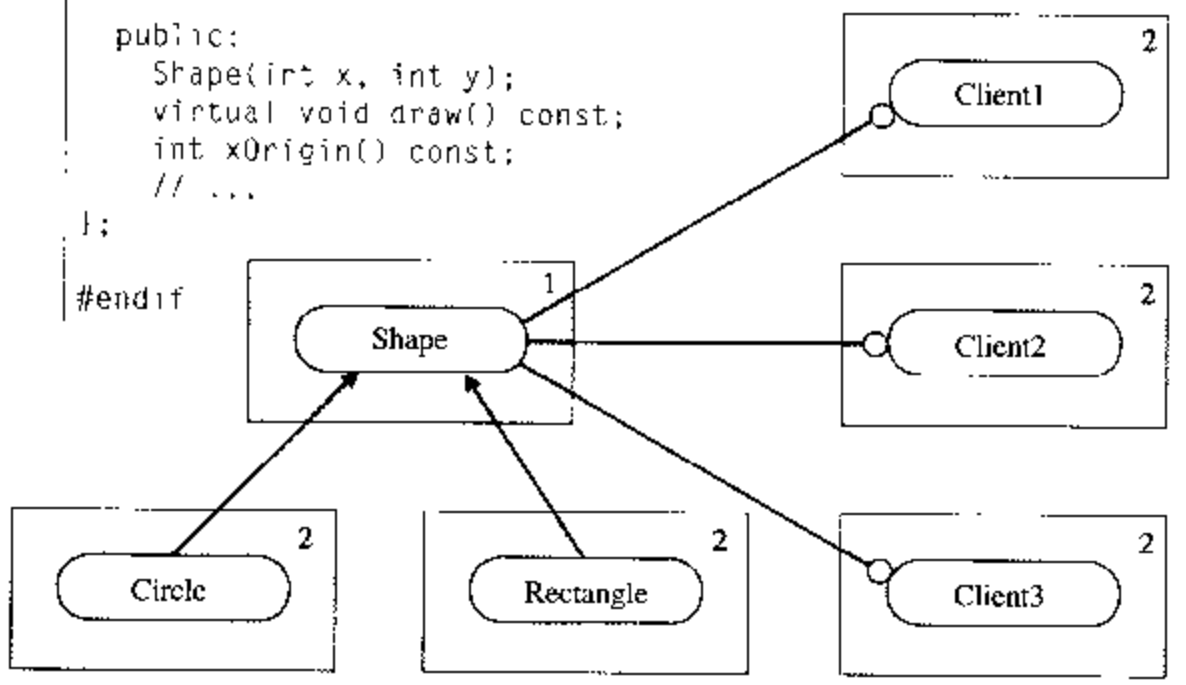
即使在只有一个组件的实现作出了改变的情况下，迫使客户程序重新编译也不是一个组件的受欢迎的物理特性。越多的组件依赖那个组件，就会有越多这种不受欢迎的编译时耦合出现。不能将客户程序与逻辑实现的变化“绝缘”，会对开发大型项目的开销产生严重的影响。

想象一个有 N 个组件的系统，系统中每个组件在编译时都依赖其他所有的组件。也就是说，编译一个组件意味着要从所有 N 个组件的头文件中包含定义并对定义进行语法分析。对这样的系统中的任何一个头文件进行一次修改的编译时开销都是惊人的。任何一个编译单元的编译开销都依赖于整个系统的大小，而不是与组件本身的大小成正比。随着整个系统规模的扩大，任一组件的编译开销会以一种不成比例的高速度增加。随着越来越多的头文件被读入每个编译单元，编译器数据结构的负荷越来越重。也就是说，若包含到一个编译单元中的行数翻倍，则用来对它进行语法分析的时间要比原来的两倍更多（如 6.1.1 节介绍的那样）。

即使对于相对小的系统（例如，共有50000行），这种类型的耦合的负担也是最繁重的。而对于中等规模和大型的项目来说，它是不可忍受的。例如，一个只需花几秒来编译的.c文件，现在要花几分钟来编译，单个的、没有绝缘的修改的编译时总开销现在不能用CPU秒而要用CPU小时来度量。

图6-3中显示的系统由一个基类Shape、一些从Shape派生而来的特定shape，以及一些只依赖基类Shape的客户程序组成。这个系统没有循环物理依赖，因此它是可层次化的。

```
// shape.h
#ifndef INCLUDED_SHAPE
#define INCLUDED_SHAPE

Shape {
    int d_x;  // could change to short
    int d_y;  // could change to short

  public:
    Shape(int x, int y);
    virtual void draw() const;
    int xOrigin() const;
    // ...
};

#endif
```




```
// circle.h
#ifndef INCLUDED_CIRCLE
#define INCLUDED_CIRCLE

#ifndef INCLUDED_SHAPE
#include "shape.h"
#endif

class Circle : public Shape {
    int d_radius;

  public:
    Circle(int x, int y, int r);
    void draw() const;
    // ...
};

#endif
```

```
// client3.c
#include "client3.h"
#include "shape.h"

// ...

void Client3::draw() const
{
    // draw each shape
    Shape *p;
    while (p = nextShape()) {
        p->draw();
    }
}

// ...
```

图6-3 一个编译时耦合系统的描述

最初，类 Shape 的作者决定使用整数来表示原点的坐标。后来作者认识到由 short int 提供的整数范围是足够的，并且 Shape 实例的大小可能会因此显著地减小。用于存储坐标的私有数据成员的基本类型，很明显是类 Shape 的实现细节。接口不会改变，并且它会继续接受和返回在有效的范围内的正常整数值（见 9.2 节）。实际上，该细节完全被 Shape 的接口封装起来了。但还是有问题。

假设 Shape 的作者将私有坐标数据类型从 int 改为 short int。图 6-3 中的哪个组件将被迫重新编译？不幸的是，正确的答案是“全部”。Circle 和 Rectangle 都是从 Shape 继承而来的，并且密切地依赖于 Shape 的内部物理布局。当 Shape 的任何一个数据类型改变后，Circle 和 Rectangle 的内部布局也必须相应地发生变化。

Shape 的客户的境况也不好。首先，shape 对象物理布局中的虚拟表指针的位置几乎肯定会受到从 int 到 short int 的影响。除非依赖的代码重新编译，否则它不会工作。更一般地说，不管什么时候修改一个头文件，所有包含该头文件的客户程序都必须重新编译。因此，实现的任何部分无论何时驻留在一个组件的头文件中，组件都不能将客户程序与其逻辑实现的部分“绝缘”。

定义：一个被包含的实现细节（类型、数据或函数）如果被修改、添加或删除时不会迫使客户程序重新编译，则称这样的实现细节被绝缘了。

可以形象地将术语**封装**比喻成围绕一个类的实现的极薄的透明膜，用于防止只通过编程来访问类的实现。而术语**绝缘**却意味着是一个无限厚的不透明的障碍，它排除了与组件的实现进行直接交互作用的任何可能性。

当一个大型系统的各种层次的内部版本之间发生错误时，修补绝缘组件（即，使客户程序与它们的实现绝缘的组件）要比修补没有绝缘的组件容易得多。就没有改变的接口而言，修改后的实现可以放在适当的位置，而不需要重新编译其他组件或者担心头文件过时。（我们将在 7.6.2 节再次介绍这个重要的论题。）

绝缘价值的一个最后的实证是它允许我们透明地替换动态装载的库。动态装载的库不是连接到一个单个的可执行代码中，而是实时地链入到一个运行的程序中。假设你是一个 C++ 应用程序库的销售商，如果你供应一个完全绝缘的库实现，那么你在增强性能和修复故障时完全不用打扰你的客户。给客户发送一个更新版本不会迫使用户程序重新编译或者连接。客户所要做的只是重新配置环境以指向新的动态装载的库，然后离开，让它们工作。

在下一小节中，我们将定量讨论编译时耦合的开销。然后我们再介绍某些特定的方法，在 C++中采用这种方法的实现细节可能变成不绝缘的。最后讨论可以改进绝缘程度的转换。

编译时耦合的开销

为了说明问题的严重性，我们设计了一个简单的实验[①]。我们机械地产生了可变数目的简单头文件，每个头文件100行。所有的头文件随后都包含在另外的空的.c文件中，所生成文件的提纲如图6-4所示。

```
// file.c                  // header0.h              // header1.h          // ...
#include "header0.h"       class Class_0_0 {       class Class_1_0 {
#include "header1.h"           // ...                    // ...
#include "header2.h"       };                      };
#include "header3.h"       class Class_0_1 {       class Class_1_1 {
#include "header4.h"           // ...                    // ...
#include "header5.h"       };                      }
// ...                     // ...                 // ...
```

图6-4 测量编译时开销的实验

然后，我们测量编译这个.c文件所需要的CPU时间。使用1000行的头文件代替100行的头文件重复该试验。图6-5提供的是在有32兆内存的SUN SPARC 20工作站上使用CFRONT 3.0编译器运行这个简单实验的结果。

第一栏表示的是系统的相对大小，其中N表示相同大小的组件的数目。第二栏和第三栏分别表示的是所测量的有100行和1000行的头文件的编译时开销。

系统大小：N（头文件数）	解析头文件所需的CPU秒	
	100行的头文件	1000行的头文件
1	0.1	0.4
2	0.1	1.0
4	0.2	3.4
8	0.4	11.0
16	0.8	32.2
32	2.4	137.7
64	8.2	497.5
128	26.5	超过一天
256	98.1	
512	397.6	
1024	超过一人	

图6-5 编译时耦合的实验开销

① 本小节提供实验数据以确证在本节中主张的观点，可以省略，不会损害连续性。

如果头文件包含的总行数是3000行左右（30个小型组件或3个大型组件），那么包含语句的行数将翻倍，编译时开销将大约翻三倍。对于这种规模的项目，使用CFRONT3.0重新编译一个.c文件的开销大致与$N^{1.6}$成正比，而对于更大的系统会变得更糟。一个原来只需几秒钟编译的编译单元，现在可能要花几分钟。

似乎这还不算糟。因为每个组件编译时依赖其他每个组件，对任何一个组件的一个非绝缘的改变意味着所有其他的组件都必须重新编译。在一个大型的编译时耦合的系统中，一个非绝缘的改变的开销不是与N^2成正比，而是更像与N^3成正比！

如果当编译任何单个的编译单元时，被包含的头文件信息的总量导致编译器超过了可用的物理内存，虚拟内存交换将完全占去编译开销的绝大部分，正如图6-5的第二列的最后一条记录和第三列的最后四条记录所示。也就是说，对于给定的编译器和系统配置，任何给定编译单元的绝对大小有相当硬性的限制。对于这个特定的配置，60000行是实际可行的；100000行则不行。

6.2 C++结构和编译时耦合

有时组件的逻辑和物理分解彼此是自然一致的。考虑一个类的非内联成员函数，其逻辑接口（声明）驻留在物理接口（.h文件）中，而其逻辑实现（函数体）驻留在物理实现（.c文件）中。在这种情况下，声明只是描述接口而没有暴露比需要或希望的更多的信息。

C++并不要求所有关于逻辑实现的细节都存在于.c文件中。由于性能方面的原因，C++允许这种紧密的编译时耦合。对于一个小的轻量级的组件（实现一个堆栈或列表），通过对其实现完全绝缘来避免编译时耦合在实践中对性能有很大影响。这种轻量级的组件一般很快可以达到一个稳定的状态，并且之后很少改动。

就提供高层次功能的组件（如，解析器或仿真器）而言，每个接口函数调用的有效工作的总量经常非常大。在这些情况下，对实现进行绝缘的运行时开销通常既不能测量也不是成比例的。

在C++中，很容易不经意地把实现细节引入一个组件的物理接口中。无论何时将一个组件的实现的一部分放到它的头文件中，我们都无法再将客户程序与这部分实现绝缘。通过使用下列结构，逻辑实现可以成为物理接口的一部分：

- 继承
- 分层
- 内联函数
- 私有成员
- 保护成员
- 编译器生成的函数
- 包含指令

- 默认参数
- 枚举类型

在下面的各个小节中，我们将分别探讨与编译时耦合有关的上述结构中每个结构的隐含意义。我们的目的只是揭示问题的特定性质，但不提供解决方案。我们将在 6.3 节开始系统地研究解决所有这些情况的绝缘技术。

6.2.1 继承（IsA）和编译时耦合

一个类无论何时派生自另一个类，即使是私有派生，也可能没有办法把客户程序与这个事实绝缘。即使私有继承被认为是派生类封装了的实现细节，派生对象的物理设计也会迫使包含派生类定义的每个客户程序都要见到基类的定义，因此对一个派生类而言，把含有基类的头文件显式地包含在该类的头文件中是合适的。无论何时修改基类的头文件（即使仅仅添加注释），在将该基类的派生类的客户程序连接为任何新的可执行程序之前，UNIX 工具（如 make）会迫使重新编译所有客户程序。

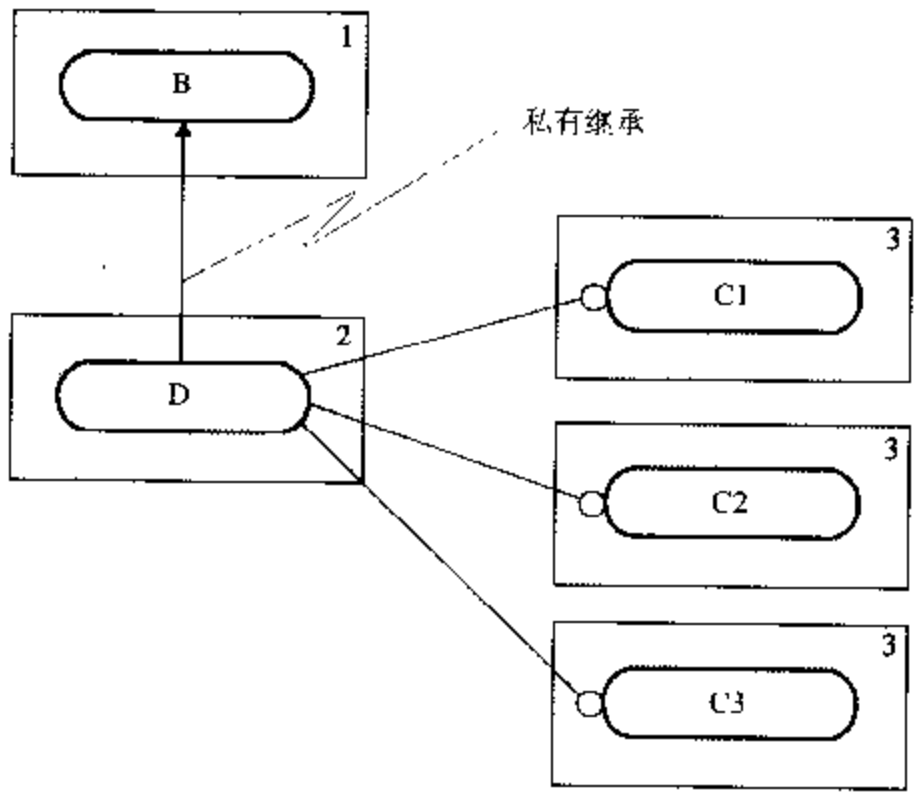


图 6-6 没有与客户程序绝缘的继承

图 6-6 说明，如果对 B 的物理接口作任何改变，那么不仅 D，而且 D 的所有客户程序（即，C1、C2 和 C3）都将被迫重新编译。

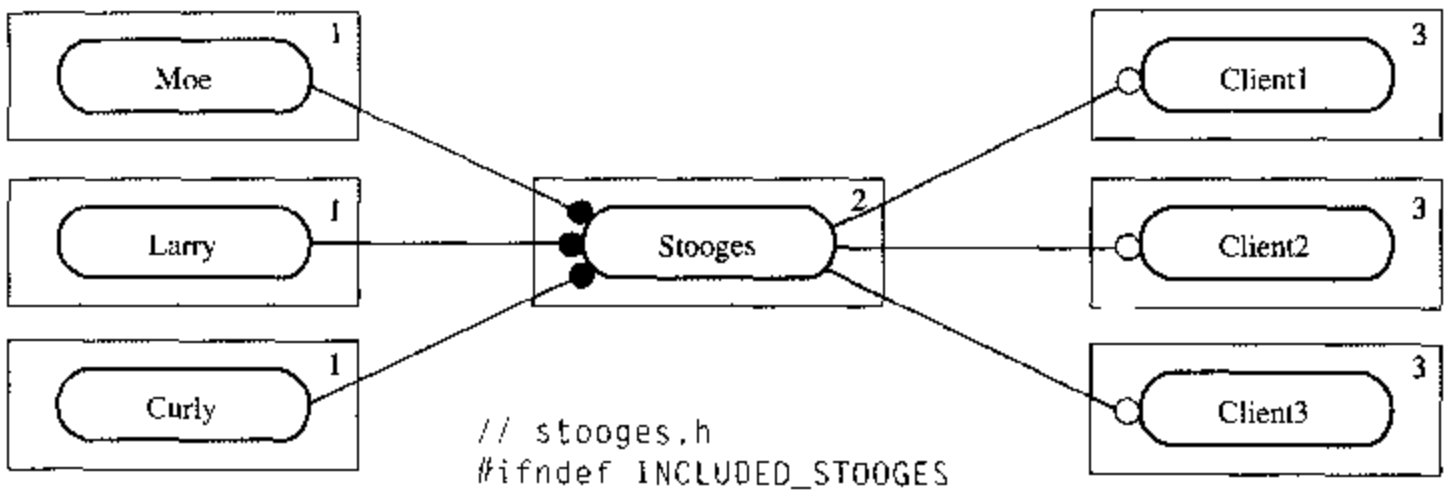
6.2.2 分层（HasA/HoldsA）和编译时耦合

当一个类在其定义中嵌入了另一个用户自定义类型的一个实例时（HasA），这个类的物理布局就会变得紧密依赖于该类型的布局。其结果是，如果没有见到嵌入到某个对象的每个分层子对象的定义，则一个客户程序不可能包含该对象的类定义。因此，若某些头文件包含了物理上嵌入到那个类的每个分层对象的定义，则一个复合对象显式地包含这些头文件是合适的。

与此相对的是，当一个类只拥有一个对象的地址时（HoldsA），这个类不必依赖这个被拥有对象的物理布局。如果是这样的话，包含这个类的文件头不去包含被拥有对象的头文件而只是声明其类型是合适的。

图 6-7 说明了类 Stooges（只在其实现中）使用类 Moe、Larry 和 Curly 的情形，与类 Larry

和 Curly 不同，Moe 被嵌入在每个 Stooges 对象中，因此不与 Stooges 的客户程序绝缘。对 Moe 的头文件的任何修改都将迫使 Stooge 的所有客户程序重新编译。




```
// stooges.h
#ifndef INCLUDED_STOOGES
#define INCLUDED_STOOGES

#ifndef INCLUDED_MOE
#include "moe.h"
#endif

class Larry;
class Curly;

class Stooges {
    Moe d_moe;
    Larry *d_larry;
    Curly& d_curly;

  public:
    Stooges();
    // ...
};

#endif
```

图 6-7　嵌入并分层的对象不能与客户程序绝缘

每个 Stooges 对象也都拥有一个指向 Larry 实例的指针和一个指向 Curly 实例的引用。为建立类 Stooges 的一个实例，类 stooges 的客户程序没有必要知道有关 Larry 或 Curly 的物理布局。因此修改 Larry 或 Curly 的头文件而不重编译任何类 Stooges 的客户程序是可能的。Larry 和 Curly 的物理布局和功能都是与 Stooges 的细节绝缘的。

6.2.3　内联函数和编译时耦合

如果某函数要在当前组件之外替换为内联的，那么声明为 inline 的这个函数必须定义在头文件中。该需求迫使将内联函数体放在组件的物理接口上。由于除了通过其自身的逻辑接口来调用内联函数之外不能通过编程来访问它，所以说一个内联函数是封装的。但是，对象

的此逻辑实现部分没有与客户程序绝缘，因此有以下后果：

（1）能使用该组件的任何程序员都能看到内联实现。

（2）改变一个内联函数的实现会迫使定义内联函数的组件的所有客户程序重新编译。

（3）将一个函数改为内联函数或从内联函数改回来，也会迫使定义该函数的组件的所有客户程序重新编译。

（4）从一个内联函数通过值返回的一个对象被实质地（in size）用在头文件中（见 5.4 节），因而决不会与客户程序绝缘（尽管从非内联函数通过值返回的一个对象可能会）。对一个在内联函数的函数体中被实质使用的对象也是这样。所以，当一个用户自定义对象传入[1]，被使用在一个内联函数中或通过值从一个内联函数返回时，显式地将定义了被使用对象的文件头包含在定义了该内联函数的头文件中是合适的。

图 6-8 说明了内联可以使类 Fred 的其他绝缘的实现细节非绝缘化的方式。例如，Fred 拥有指向类型为 Wilma、Betty、Barney 和 MrSlate 的对象的指针。因此 Fred 的对象布局不会依赖于这些类型的任何对象布局。因为成员函数 getWilma 通过值返回一个 Wilma 类型的对象并且被声明为 inline，所以类 Fred 的所有客户程序已经看到类 Wilma 的定义是必要的。因为成员函数 getBetty 没有声明为 inline，因此，不需要调用 getBetty 并且不另外实质依赖类型 Betty 的 Fred 的客户程序也不必包含类 Betty 的头文件。换句话说，没有使用类型 Betty 的客户程序不会在编译时被迫依赖 Betty。

类 Barney 的一个实例从成员函数 getBarney 通过引用返回，因此，除非一个客户程序实质依赖于类 Barney，否则客户程序没有必要为 Barney 包含类定义。客户程序又一次没有被迫依赖它所不需要的组件。

最后，成员函数 getSalary 在其实现中实质使用了封装的 MrSlate 对象。因为 getSalary 声明为 inline，所以 Fred 的所有客户程序都被要求已经看到了类 MrSlate 的定义，无论它们是否调用了 getSalary。当然，如果这些内联函数中的任何一个改变了，那么 Fred 的所有客户程序都将不得不重新编译。

6.2.4 私有成员和编译时耦合

一个类的每个私有数据成员——尽管封装好了——也没有与该类的客户程序绝缘。我们已经介绍了几个修改实现后要求改变私有数据成员，并反过来要求客户程序重新编译的例子。例如，将一个 Stack 类的封装实现从基于链表的栈改变为基于数组的栈，将迫使 Stack 的所有客户程序重新编译。正如读者已经看到的，即使很小的代码调整，如将 int 改为 short int，也足以迫使所有的客户程序重新编译。

读者经常被提醒，私有成员函数是一个类的封装细节，但是它们并不是已绝缘的实现细节——甚至在它们没有声明为 inline 的时候。仅改变一个类的私有成员函数的基调也足以迫使

① 几乎永远不会通过值来将一个用户自定义的类型传递到一个函数中（见 9.1.11 节）。

定义该类的组件的所有客户程序重新编译。

```
// fred.h
#ifndef INCLUDED_FRED
#define INCLUDED_FRED

#ifndef INCLUDED_WILMA
#include "wilma.h"
#endif

#ifndef INCLUDED_MRSLATE
#include "mrslate.h"
#endif

class Barney;
class Betty;

class Fred {
    Wilma *d_wilma_p;
    Barney *d_barney_p;
    Betty *d_betty_p;
    MrSlate *d_mrSlate_p;

  public:
    Fred();
    Wilma getWilma() const { return *d_wilma_p; }
    Betty getBetty() const;  // non-inline function
    const Barney& getBarney() const { return *d_barney_p; }
    double getSalary() { return d_mrSlate_p->askForRaise(); }
};

#endif
```

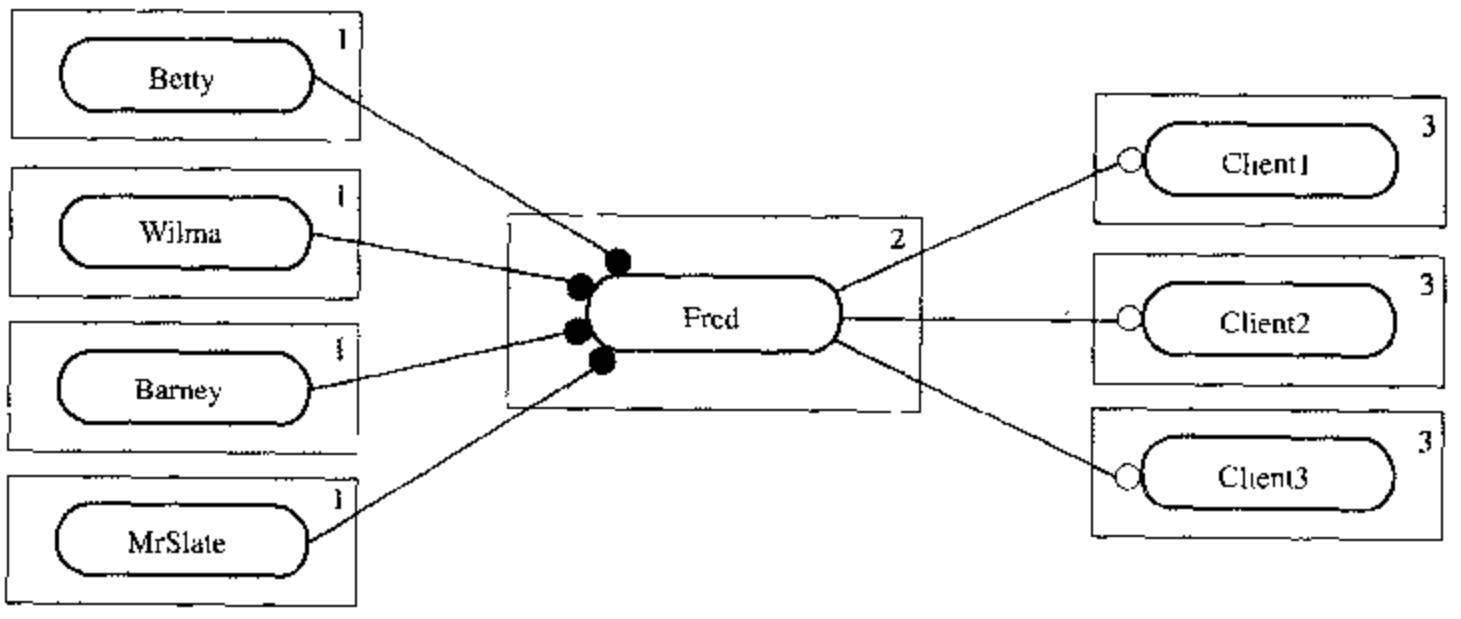


图 6-8　内联函数减少绝缘的方式

图 6-9 说明了私有成员的问题。d_length 成员是一个细节，添加它大概是因为作者认为追

踪长度比实时计算长度的效率更高。如果已证明该假设是不正确的，那么删除 d_length 将引起该组件的所有客户程序重新编译。类似地，copy 函数被实现来分解拷贝操作，以便既可以在拷贝构造函数中使用，也可以在赋值运算符中使用。如果我们现在决定将这个私有助手函数的基调从 copy(const String&)改变为 copy(const char *)，以便使其也能用来实现默认构造函数，那么所有的客户程序又将被迫重新编译。

```
// str.h
#ifndef INCLUDED_STR
#define INCLUDED_STR

class String {
    char *d_string_p;
    int d_length;
    void copy(const String& string);

  public:
    String(const char *str);
    String(const String& string);
    ~String(const char *str);
    String& operator=(const String& string);
    // ...
};

#endif
```

图 6-9 私有成员不是绝缘的

6.2.5 保护成员和编译时耦合

当考虑保护成员时，基类的作者现在必须面对两类不同的观众：派生类的作者和一般用户。保护函数在特别为派生类而编写的接口中，但是准备让一般用户作为实现细节来对待。注意，保护成员数据很少是合适的，尤其是在把绝缘作为一个设计目标的广泛使用的接口中。

表面上，保护接口为预期的派生类作者提供了一个方便的地方，可以查看并决定什么是需要的。但是，就像私有成员一样，保护接口是在类定义中声明的，因而就一般用户而言，它不是一个绝缘的实现细节。以任何方式修改一个基类的保护接口，将迫使以下三个方面重新编译：（1）基类的全部客户程序；（2）所有的派生类；（3）派生类的所有客户程序。

6.2.6 编译器产生的成员函数和编译时耦合

一些基本的成员函数是由编译器自动产生的，除非这些函数在一个类中被显式地声明[①]。特别地，除非指定了一个拷贝构造函数，否则编译器将产生一个具有像函数成员那样的拷贝

① 见 **meyers**，Item 45，172 页。

语义的拷贝构造函数。即，生成的构造函数会按照各自的初始语义来拷贝每个成员对象和基类对象[1]。

如果需要，一个隐含的赋值运算符也将以类似的方式产生，并根据它自己的赋值语义拷贝每个成员对象和基类部分。一个析构函数也将在需要的时候产生，以调用各层的和基类对象的析构函数。

在许多情况下，编译器产生的构造函数、赋值运算符以及析构函数所做的正是所需要的工作。但是，如果一个类的作者确定的需求与编译器产生的定义不同，那么有必要将合适的成员声明引入到类定义中。这样的声明引入不能认为是绝缘的，并且类的任何客户程序都将被迫重新编译。

如图 6-10 所示，类 ComplexSymbol 默认地实现了拷贝构造、赋值和析构。如果我们决定消除 ComplexSymbol 对 String 的实现依赖而使用 char *，那么有必要为拷贝构造函数、赋值运算符以及析构函数引入一个声明。在这个例子中，只在私有数据中的修改会迫使我们的客户程序重新编译；但是，即使我们解决了那个问题（并且我们能够解决），在组件的头文件中引入新的声明也是一种不能与客户程序绝缘的变化。

```
class ComplexSymbol : public Complex {
    String d_name;

  public:
    // CREATORS
    ComplexSymbol(const String& name, double re, double im = 0.0);
    // Default copy ctor and dtor are fine.

    // MANIPULATORS
    // Default assignment operator is fine.
    // ...

    // ACCESSORS
    // ...
};
```

图 6-10　依赖于编译器产生的函数

6.2.7　包含命令和编译时耦合

在本章的开始部分（见 6.1.1 节）证实编译时耦合的高开销的实验中，我们甚至没有考虑每个头文件可能直接包含每一个其他的头文件的可能性[2]。我们假设每个.c 文件显式地包含系统中的每一个头文件，因为需要这样做。在实践中，这种情况不会发生。

① 见 ellis，12.8 节，295 页。

② 在一些环境中，我们可能会遇到在某一时刻同时打开源文件的数目的限制。

更有可能发生的是每个头文件将包含一个或多个头文件，这些文件又依次包含一个或多个别的头文件，直到最后实际上在系统中的每个头文件都被包含了。这时，冗余包含卫哨（2.5节）通过消除二次性（我们从C++预编译器所花费的时间中观察到的），来减少编译开销。

考虑图6-11中的例子。一个Bank类在其接口中使用一个BankCard类和一个货币类的变体。Bank类并没有继承任何其他类。让我们假设Bank类没有实质使用任何BankCard类或任何货币类的内联函数。并进一步假设类Bank在其自己的定义中没有嵌入任何用户自定义类的实例（HasA）。

```
// bank.h
#ifndef INCLUDED_BANK
#define INCLUDED_BANK

#ifndef INCLUDED_BANKCARD
#include "bankcard.h"
#endif

#ifndef INCLUDED_GERMANMARKS
#include "germanmarks.h"
#endif

#ifndef INCLUDED_JAPENESEYEN
#include "japeneseyen.h"
#endif

#ifndef INCLUDED_UNITEDSTATESDOLLARS
#include "unitedstatesdollars.h"
#endif

#ifndef INCLUDED_ENGLISHPOUNDS
#include "englishpounds.h"
#endif

// ...
// ...
// ...

#ifndef INCLUDED_LAKOSIANFOOBARS
#include "lakosianfoobars.h"
#endif

class Bank {
    // ...
    Bank(const Bank&);                    // We don't want to copy
    Bank& operator=(const Bark&);         // or assign banks.

  public:
    // CREATORS
    Bank();
    ~Bank();
```

```
    // MANIPULATORS
    GermanMarks           getMarks(BankCard    *cashMachineCard, double amount);
    JapaneseYen           getYen(BankCard      *cashMachineCard, double amount);
    UnitedStateDollars    getDollars(BankCard  *cashMachineCard, double amount);
    EnglishPounds         getPounds(BankCard   *cashMachineCard, double amount);
    // ...
    // ...
    // ...
    LakosianFooBars       getFooBars(BankCard  *cashMachineCard, double amount);
};

#endif
```

图 6-11　在其接口中使用多种类型的类

现在考虑该 Bank 类的一个实例的一个美国客户。这个人一般喜欢带上他（或她）的银行卡去银行取出一定数量的美金。图 6-12 显示了一个 Person 的取钱成员函数的简单例子。

```
// person.c
#include "person.h"
#include "bank.h"

// ...

void Person::withdraw(double amount)
{
    d_wallet_p->putIn(d_bank_p->getDollars(d_cashMachineCard_p, amount));
}
```

图 6-12　Person 的取钱成员函数的简化实现

描述一个想象的 Lakos 岛共和国：其国家货币单位 FooBar 非常不稳定，并且很容易擅自变化。今天，该国再次宣称有意对 FooBar 的实现作出非绝缘的改变。世界财经界需要知道谁将被迫重新编译。

不仅 LakosianFooBars 的所有实际的客户程序必须重新编译，而且 Bank 的所有其他客户程序也将重新编译。即，如果我们将钱存入该银行，无论我们是否关心过或者听说过 LakosinanFooBars，对 lakosianfoobar.h 的任何改变都将使得软件配置管理工具（例如 make）为我们自动重新编译。

雪上加霜的是，没有真正的需求要我们在编译时依赖这种货币！我们的代码没有任何部分在编译时依赖这种货币。那么，为什么银行的作者决定在 bank.h 中而不是在 bank.c 中包含所有这些头文件呢？我们得到的答案可能是“为了客户的方便”。

Bank 组件的作者认为，就是为了防备我们可能会用到某种类的定义，所以将它包含进来。这种方法有相对较小的优点，只要我们包含 bank.h 文件，我们决不需要为 UnitedStatesDollars 或 BankCard 包含头文件。但是，这种方法也有相对较大的缺点，即我们将永远受潜在的大量的头文件的支配，这些头文件我们既不能控制也不必关心。

6.2.8 默认参数和编译时耦合

一个单个的算法经常依赖于数个参数——一些参数带有合理的默认值。将这些默认值放置到定义函数接口的头文件中可以简化自我建档，这只是因为它们将更多的信息放进了头文件。

```
class Circle {
    // ...
  public:
    Circle(double x = 0, double y = 0, double radius = 1);
    // ...
};
```

但是，这样的默认值会随着接口一起编译，而对这些值的任何修改将强迫客户程序重新编译。

6.2.9 枚举类型和编译时耦合

枚举类型、CPP 宏、typedef 和（默认的）非成员 const 数据都没有外部的连接（见 2.3.3 和 2.3.4）。同样地，如果它们被其他组件使用（或者如果它们出现于准备在组件外部使用的任何内联函数的函数体中），那么这些结构必须出现在组件的头文件中。

图 6-13 说明了在小型工程中将系统内的定义聚合到单个组件这种普遍的做法。当更多的组件添加到系统中时，这些组件一般会包含这个共同的定义文件。无论何时需要一个新的定义或返回一个状态，都添加到 sysdefs.h 文件中。添加的组件越多，添加到一个共同的定义的机会就越大。无论何时将一个共同的定义添加到 sysdefs.h 中，系统中的几乎所有组件都将被迫重新编译。

最终，系统到达某一点，此时对全局定义作一次添加的代价太昂贵了。因此不再在这个文件中放置一个有用的定义，而是将它们保持为局部的或私有的；不再把特定的新的返回状态添加到枚举类型中，而是反复使用以前已存在的代码（例如 UNSPECIFIED_ERROR），即使它们是模糊的甚至是不适当的。

这里的问题是，枚举类型和 typedef 不是实现细节，而是一个组件的公共接口的普通部分。这个组件的接口不是一个抽象的组织良好的、内聚的表达，而是一些细节的折衷的大杂烩。枚举类型的这种太普遍的使用，不能很好地适应项目规模的增大。

在这个系统中会产生编译时耦合，因为接口不是从物理层次结构的低层而是从要实现的高层驱动的。这种向上的依赖将一种隐含的编译时耦合强加到所有的客户程序身上，即使这些客户程序处于互不相干的系统部分中。这个例子说明了一种更一般的问题，涉及一个组件的所有权的共享问题。

在 6.3 节中，我们将讨论解决此问题以及与绝缘相关的其他问题的特定技术。

```
// sysdefs.h
#ifndef INCLUDED_SYSDEFS
#define INCLUDED_SYSDEFS

#ifndef INCLUDED_MATH
#include <math.h>                       // bad idea: should be insulated
#define INCLUDED_MATH
#endif

const double PI_BY_4 = M_PI/4;          // bad idea: should be class member
const double PI_BY_8 = M_PI/8;          // bad idea: should be class member

struct SysDefs {
    typedef int (*Pfdi)(double);
    typedef double (*Pfid)(int);

    enum ReturnStatus {
        SUCCESS = 0,
        WARNING,
        IOERROR,
        FILE_NOT_FOUND,
        // ...
        OUT_OF_RANGE,
        // ...
        OUT_OF_MEMORY,
        // ...
        // ...
        INVALID_GEOMETRY,
        // ...
        // ...
        // ...
        UNSPECIFIED_ERROR
    };
};

#endif
```

图 6-13　包含共同定义的组件

6.3　部分绝缘技术

不是每个组件都应试图将其客户程序与每个实现细节绝缘。但是，在其他条件都相同的情况下，将客户程序与实现细节绝缘比不这么做要好——即使只是为了减少物理接口中的混乱。

幸好绝缘不必是一个“要么全有要么全无”的命题。绝缘一个实现细节可能是所希望的——甚至在别的实现细节仍然是不绝缘的时候。一个组件的实现越绝缘，实现的改变迫使组件的客户程序重新编译的可能性就越小。

有时对一个实现细节进行绝缘处理与不对它进行绝缘处理一样容易。就像悬挂得很低的水果，我们花费很少的努力就能取得显著的收获。另一些时候，绝缘可能需要相当大量的和深思熟虑的工作。值得花费在绝像某个给定的实现细节上的工作量，取决于该细节的变化可

能影响客户程序的程度。

下面的小节提供了一个特定技术的集合。这些技术可以有选择地减少暴露在一个组件的物理接口上的实现细节的数量。

6.3.1 消除私有继承

与公共继承（和保护继承）不同，私有继承是一个实现细节。与分层相比，私有继承的“优点”之一是，可以更方便地表示，通过访问声明（access-declarations）[1]或使用声明（using-declarations）[2]有选择地向派生类的客户程序暴露一些（但不是全部）私有基类中的函数。

图 6-14 说明了一个类如何私有地继承另一个类，然后使用一个访问声明在它自己的接口上有选择地发布具有给定名字的所有成员。

```
// base.h
#ifndef INCLUDED_BASE
#define INCLUDED_BASE

class Base {
    // ...
  public:
    Base();
    ~Base();
    void f1(int);
    void f2(double);
    int f1() const;
    double f2() const;
};

#endif
```

(a) 私有基类头文件

```
// myclass.h
#ifndef INCLUDED_MYCLASS
#define INCLUDED_MYCLASS

#ifndef INCLUDED_BASE
#include "base.h"
#endif

class MyClass : private Base {
  public:
    MyClass() {}
    Base::f1; // access declaration
};

#endif
```

(b) 派生类头文件

图 6-14 私有继承和访问声明

有几个原因使得访问声明的用处不确定。它在公共接口中暴露一些函数，但是一个客户程序为了知道这些函数是什么，必须查看私有派生（实现）类的头文件，以便了解适当的参数和返回值。另一个问题是，使用访问声明的类不能将其客户程序与其私有基类绝缘。客户程序会受到私有的（未公开的）函数中的变化的影响（这些函数甚至可能没有在派生类的实现中使用）。

使用私有继承而不是分层的一个原因可能是想利用基类的虚拟表。通过覆盖声明在私有

---

① **ellis**，11.3 节，244 页。

② **stroustrup94**，17.5.2 节，419 页。

基类中的虚拟函数的行为，我们也许能够对那些在基类层次上依赖于被覆盖行为的其他行为进行“定制”或“编程”。另外，也有可能为了派生目的而创建一个哑元类（dummy class），然后使用该哑元类继续进行分层。如果绝缘不是问题，那么私有继承可能是合适的。但是，如果该类将成为一个更一般的公共接口的一部分，那么从继承转换到分层是恰当的。

图 6-15 说明了如何获得与图 6-14（b）一样的逻辑接口，而不会将实现类的细节暴露给客户。新的实现没有从 Base 类中私有派生，而是拥有一个向外的指向类 Base 的不透明指针。无论何时建立 MyClass 的一个实例，被声明为非内联的合适的构造函数都会动态地分配一个新的 Base 实例，并且将其地址赋给 MyClass 的成员 d_base_p。当这个 MyClass 的实例被删除后，非内联析构函数将删除这个实例。赋值运算符也需要适当地管理这个基类对象指针。

```
// myclass.h
#ifndef INCLUDED_MYCLASS
#define INCLUDED_MYCLASS

class Base;

class MyClass {
    Base *d_base_p;

  public:
    MyClass();
    MyClass(const MyClass& c);
    ~MyClass();

    MyClass& operator=(const MyClass& c);
    void f1(int i);

    int f1() const;
};

#endif
```

（a）对头文件进行绝缘

```
// myclass.c
#include "myclass.h"
#include "base.h"

MyClass::MyClass() : d_base_p(new Base

MyClass::MyClass(const MyClass& c)
: d_base_p(new Base(*c.d_base_p)) {}

MyClass::~MyClass() { delete d_base_p; }

MyClass& MyClass::operator=(const MyClass& c)
{
    if (this != &c) {
        delete d_base_p;
        d_base_p = new MyClass(*c.d_base_p);
    }
    return *this;
}

void f1(int i) { d_base_p->f1(i); }

int f1() const { return d_base_p->f1(); }
```

（b）对实现文件进行绝缘

图 6-15 使用分层而不是私有继承

MyClass 的新成员函数没有使用访问声明来有选择地发布一个私有基类的成员，而是定义为（非内联地）转发调用请求给定义在类 Base 中的相应的函数。注意，如果客户程序与 Base 的定义绝缘，那么所有实质依赖于 Base 的 MyClass 的成员函数都必须声明为非内联函数。

以这种方式，现在类 MyClass 将其客户程序与对类 Base 的所有组织上的改变绝缘了。如果类 Base 是抽象类，那么 d_base_p 将指向由 Base 派生的具体的哑元类，它可能完全在文件 myclass.c 中实现。注意所有这些绝缘都不是没有代价的（例如，额外的函数调用以及动态分配），正如在 6.6.1 节中所详细讨论的。

6.3.2 消除嵌入数据成员

即使性能需求阻止我们完全绝缘一个类，我们仍然可以通过将这个实现类的所有嵌入实例转换为指向那个类的指针（或引用），然后在类的构造函数、析构函数和赋值运算符中显式地管理这些指针，使客户程序与单个实现类绝缘。

图 6-16 显示了我们通过将一个 HasA 关系[图 6-16（a）所示]转换为一个 HoldsA 关系[图 6-16（b）所示]，实现了有选择地将客户程序与实现绝缘。这样做时，我们必须将所有那些以前操作 MyClass 数据成员的 YourClass 类型的内联函数重新声明为非内联函数。HoldsA 的负面影响是管理分层的实例所需的额外工作，并且还有与间接寻址、动态分配和非内联函数相关的附加的性能开销。注意我们是如何通过内联函数来继续访问性能攸关的成员数据（如 d_count）的。

```
// myclass.h
#ifndef INCLUDED_MYCLASS
#define INCLUDED_MYCLASS

#ifndef INCLUDED_YOURCLASS
#include "yourclass.h"
#endif

class MyClass {
    int d_count;
    YourClass d_yours;

  public:
    // ...
    int yourValue() const
    {
        return d_yours.value();
    }

    int count() const
    {
        return d_count;
    }
};

#endif
```

（a）将 YourClass 与 MyClass 的客户程序绝缘之前

```
// myclass.h
#ifndef INCLUDED_MYCLASS
#define INCLUDED_MYCLASS

class YourClass;

class MyClass {
    int d_count;
    YourClass *d_yours_p;

  public:
    // ...
    int yourValue() const;

    int count() const
    {
        return d_count;
    }
};

#endif
```

（b）将 YourClass 与 MyClass 的客户程序绝缘之后

图 6-16 将 HasA 转化为 HoldA 以改进绝缘性

6.3.3 消除私有成员函数

私有成员函数尽管封装了一个类的逻辑实现细节，但仍然是一个组件的物理接口的一部

分。非内联私有成员函数有外部连接；这使声明为该类的友元并且在其他编译单元中定义的函数和类能够调用它们。但是，正如 3.6 节所讨论的那样，视任何在一个组件之外定义的函数或类为友元，将使得不受约束的客户利用我们的类的私有细节。避免远距离的友元关系意味着只有定义在一个组件内的函数和类才能访问私有成员。幸好 C++（甚至 C）支持一个更严格的在组件范围内的访问控制形式。这种访问控制形式不再让函数成为类的私有成员，而是使它成为一个在一个组件的.c 文件作用域内声明的静态自由函数[1]。

有时函数成为私有成员不是因为它们需要私有访问，而是因为头文件的私有部分是一个存储这些被分解的助手函数的好地方。即，一些私有助手函数只使用类的公共接口就可以完成它们的所有工作。在这些情况下，从私有成员到静态自由函数的变换是很容易的，并且分两步就能很快完成。

第一步是通过添加一个合适的可写指针或只读引用参数到函数中，将每个私有成员函数转换为一个私有的静态成员。考虑类 MyClass[如图 6-17（a）所定义的]，类 MyClass 包含两个私有成员函数——f 和 g。成员 f 是一个 non-const 操纵函数，而成员 g 是一个 const 访问函数。操纵函数 f 潜在地改变对象，因此，为了与我们的策略一致（见 9.1.1 节），我们将通过 non-const 指针以及其他参数将实例传递给该函数。访问函数 g 是无害的（innocuous），我们将通过 const 引用以及 g 的其他参数来传递实例，如图 6-17（b）所示。

```
// myclass.h
#ifndef INCLUDED_MYCLASS
#define INCLUDED_MYCLASS

class MyClass {
    // ...
  private:
    void f(...);
    int g(...) const;

  public:
    // ...
};

#endif
```

（a）带有私有成员函数的原始类

```
// myclass.h
#ifndef INCLUDED_MYCLASS
#define INCLUDED_MYCLASS

class MyClass {
    // ...
  private:
    static void f(MyClass *myClass, ...);
    static int g(const MyClass& myClass, ...);

  public:
    // ...
};

#endif
```

（b）修改后只有私有静态成员函数的类

图 6-17　使私有成员函数成为静态成员

第二步是从头文件中整个删除这些函数声明，在.c 文件的函数定义中删除这些成员的表示法[如图 6-18（a）所示]，并且最后在这些定义中的每一项前加上关键字 static，如图 6-18（b）所示。注意第二步应该不需要对定义在.c 文件中的其他成员函数的实现进行任何改变。

① 一旦这种相对的新语言特性变得能够更广泛地使用，正如在 **stroustrup94**（17.5.3 节，419~420 页）中讨论的那样，通过使用未命名的名称空间，我们能够更优雅地取得同样的效果。

```
// myclass.c
#include "myclass.h"
void MyClass::f(MyClass *myClass, ...) { /* ... */ }
int MyClass::g(const MyClass& myClass, ...) { /* ... */ }
// ...
```

(a) 带有私有静态成员函数的原始类

```
// myclass.c
#include "myclass.h"
static void f(MyClass *myClass, ...) { /* ... */ }
static int g(const MyClass& myClass, ...) { /* ... */ }
// ...
```

(b) 修改后的有静态自由函数的类

图 6-18 将静态成员函数转换为自由函数

不幸的是，私有成员函数经常直接操作其他私有实现细节，这可能使这些函数更难以成为自由函数。考虑图 6-19 中定义的 list 组件，类 List 包含三个私有成员函数——copy、clean 和 end，这些函数被重复使用来帮助实现类 List 的公共功能。

```
// list.h
#ifndef INCLUDED_LIST
#define INCLUDED_LIST

class List;
class ListIter;
class ostream;

class Link {
    int d_data;
    Link *d_next_p;
    friend List;
    friend ListIter;

    Link(const Link& link);                    // not implemented
    Link& operator=(const Link& link);         // not implemented

    // CREATORS
    Link(int data, Link *next = 0);
};

class List {
    Link *d_head_p;
    friend ListIter;

  private:
    static Link *copy(const Link *link, Link *end = 0);
        // allocate and return new copy of given list of links

    void clean();
        // destroy and deallocate entire list of links

    Link *& end();
        // return a reference to the end of the list
```

```
    public:
      // CREATORS
      list();
      List(const List& list);
      ~List();

      // MANIPULATORS
      List& operator=(const List& list);
      void append(int i);
      void append(const List& list);
      void prepend(int i);
      void prepend(const List& list);
};

ostream& operator<<(ostream& o, const List& list);

class ListIter {
    // ...
};

#endif
```

图 6-19（a） 带有私有成员函数的 List 类的 list.h 文件

```
// list.c
#include "list.h"
#include <iostream.h>

                                // **********
                                // class Link
                                // **********

// CREATORS
Link::Link(int data, Link *next) : d_data(data), d_next_p(next) {}

                                // **********
                                // class List
                                // **********

// PRIVATE MEMBERS
Link *List::copy(const Link *link, Link *end)
{
    Link* linkPtr = end;
    for (Link **addrLinkPtr = &linkPtr; link; link = link->d_next_p) {
        *addrLinkPtr = new Link(link->d_data, *addrLinkPtr);
        addrLinkPtr = &(*addrLinkPtr)->d_next_p;
    }
    return linkPtr;
}

void List::clean()
{
    while (d_head_p) {
```

```
            Link *tmp = d_head_p;
            d_head_p = d_head_p->d_next_p;
            delete tmp;
        }
    }

    Link *& List::end()
    {
        Link **addrLinkPtr = &d_head_p;
        while (*addrLinkPtr) {
            addrLinkPtr = &(*addrLinkPtr)->d_next_p;
        }
        return *addrLinkPtr;
    }

    // CREATORS
    List::List() : d_head_p(0) {}
    List::List(const List& list) : d_head_p(copy(list.d_head_p)) {}
    List::~List() { clean(); }
    // MANIPULATORS
    List& List::operator=(const List& list)
    {
        if (this != &list) {
            clean();
            d_head_p = copy(list.d_head_p);
        }
        return *this;
    }

    void List::append(int i) { end() = new Link(i); }

    void list::append(const List& l) { end() = copy(l.d_head_p);

    void List::prepend(int i) { d_head_p = new Link(i, d_head_p);

    void List::prepend(const List& l) { d_head_p = copy(l.d_head_p, d_head_p); }

    // FREE FUNCTION
    ostream& operator<<(ostream& o, const List& list)
    {
        o << '[';
        for (ListIter it(list); it; ++it) {
            o << ' ' << it();
        }
        return o << " ]";
    }
                                    // **************
                                    // class ListIter
                                    // **************

    // ...
```

图 6-19（b） 带有私有成员函数的 List 类的 list.c 文件

copy 函数已经是一个静态成员，但是它需要访问辅助（“隶属”）类 Link。clean()和 end()都依赖于对标识列表头的私有数据成员 d_head_p 的访问，并且没有公共函数能够用来获得对它的访问。使这三个函数成为 List 的非成员，将剥夺它们访问 List 和 Link 的实现的特权。尽管这些函数将不再能够访问任何类的私有细节，但是这些函数的调用者仍是有完全访问权的成员，它们可自由提供这些信息。

正如图 6-20 所示的那样，我们可以修改 clean 和 end 的助手成员函数，使它们像 copy 函数一样被声明为静态的，并且使用它们要访问的私有信息作为参数。当这两个函数的客户程序进行调用时，它们现在必须提供更多一些的信息。但是这些函数将不再需要依赖对 List 类进行私有访问来完成它们的工作。剩下的惟一问题是，为了完成任务，这些函数仍然依赖对封装的 Link 类的私有功能的访问。

```
// list.h

// ...

class List {
    // ...
  private:
    static void clean(Link *link);
    static Link *& end(Link **addrLinkPtr);
    // ...
};

// ...
```

```
// list.c

// ...

void List::clear(Link *link)
{
    while (link) {
        Link *tmp = link;
        link = link->d_next_p;
        delete tmp;
    }
}

Link *& List::end(Link **addrLinkPtr)
{
    while (*addrLinkPtr) {
        addrLinkPtr = &(*addrLinkPtr)->d_next_p;
    }
    return *addrLinkPtr;
}

// ...
```

图 6-20　传递私有信息到静态自由函数中

一个解决方案是使 Link 类中被需要的功能可以被公共访问。由于 Link 的使用是 List 的一种封装的实现细节，所以允许客户（或者测试工程师）访问 Link 类的独立实例不会引起什么危害。但是，从绝缘的角度来看，更好的解决方案是将 Link 类的不重要的定义移到.c 文件中，并使它整个成为公共可访问的。这种解决方案不仅增加了 list 组件实现的绝缘程度，而且消除了许多在头文件中的不必要的混乱。List 的改进版本如图 6-21（a）和 6-21（b）所示。

```
// list.h
#ifndef INCLUDED_LIST
#define INCLUDED_LIST

class Link;
class List;
class ListIter;
class ostream;

class List {
    Link *d_head_p;
    friend ListIter;

  public:
    // CREATORS
    List();
    List(const List& list);
    ~List();

    // MANIPULATORS
    List& operator=(const List& list);
    void append(int i);
    void append(const List& list);
    void prepend(int i);
    void prepend(const List& list);
};

ostream& operator<<(ostream& o, const List& list);

class ListIter {
    // ...
};

#endif
```

图 6-21（a） 带有静态自由函数的 list 组件的 list.h 文件

```
// list.c
#include "list.h"
#include <iostream.h>

                                // **********
                                // class Link
                                // **********
```

```
struct Link {
    int d_data;
    Link *d_next_p;

    Link(const Link& link);                    // not implemented
    Link& operator=(const Link& link);         // not implemented

    // CREATORS
    Link(int data, Link *next = 0) : d_data(data), d_next_p(next) {}
};

                                    // **********
                                    // class List
                                    // **********

// STATIC FREE FUNCTIONS
static Link *copy(const Link *link, Link *end = 0)
{
    Link* linkPtr = end;
    for (Link **addrLinkPtr = &linkPtr; link; link = link->d_next_p) {
        *addrLinkPtr = new Link(link->d_data, *addrLinkPtr);
        addrLinkPtr = &(*addrLinkPtr)->d_next_p;
    }
    return linkPtr;
}

static void clean(Link *link)
{
    while (link) {
        Link *tmp = link;
        link = link->d_next_p;
        delete tmp;
    }
}

static Link *& end(Link **addrLinkPtr)
{
    while (*addrLinkPtr) {
        addrLinkPtr = &(*addrLinkPtr)->d_next_p;
    }
    return *addrLinkPtr;
}

// CREATORS
List::List() : d_head_p(0) {}
List::List(const List& list) : d_head_p(copy(list.d_head_p)) {}
list::~list() { clean(d_head_p); }

// MANIPULATORS
List& List::operator=(const List& list)
{
    if (this != &list) {
```

```
        clean(d_head_p);
        d_head_p = copy(list.d_head_p);
    }
    return *this;
}

void List::append(int i) { end(&d_head_p) = new Link(i); }

void List::append(const List& l) { end(&d_head_p) = copy(l.d_head_p); }

void List::prepend(int i) { d_head_p = new Link(i, d_head_p); }

void List::prepend(const List& l) { d_head_p = copy(l.d_head_p, d_head_p); }


// FREE FUNCTION
ostream& operator<<(ostream& o, const List& list)
{
    o << '[';
    for (ListIter it(list); it; ++it) {
        o << ' ' << it();
    }
    return o << " ]";
}
                                // ***************
                                // class ListIter
                                // ***************
// ...
```

图 6-21（b） 带有静态自由函数的 list 组件的 list.c 文件

有时私有成员函数可以转变为静态自由函数，这些自由函数不依赖于定义在当前组件中的类型。如果这些函数并不是微不足道的，那么尝试直接地检验它们是有益的。不要创建一个单个的带有不可访问的（但并不是微不足道的）静态自由函数的组件，而是考虑构造两个组件——一个组件带有公共静态成员，它们可用于实现另一个组件。

图 6-22 说明了从 myclass.c 文件中移走文件作用域内的独立静态函数、并使它们成为在一个独立的工具组件中可以公共访问的静态成员的结果。当函数是可重用的或并非微不足道的时候，这种技术是有意义的。而当仅仅这些函数的 CCD 就比原始组件的 CCD 要小得多时，这种技术就特别有用。

尽管静态函数在编译时耦合方面比私有函数更优越，但是性能可能会成为一个问题，尤其是当在文件作用域内有许多必须传入或传出静态函数的私有状态信息时。在这些情况下，其他更一般形式的绝缘（在 6.4 节中讨论）更可取。

6.3.4 消除保护成员

保护成员的好处是什么呢？即，什么时候对类成员进行保护性访问是合适的？一个简单的回答是，当我们要区分两种不同的观众（派生类作者和一般用户）时，保护成员是合适的。

当保护接口用于封装私有细节（见 2.2 节）时，它与公共接口同样重要。但是人们更注意公共接口，而较少注意保护接口。我们必须认识到，即使单个实例化对象的保护接口对于公众是不可访问的，任何人也都可以派生一个依赖于这些保护细节的类。

```
// myclass.c
#include "myclass.h"
#include "myclassimputil.h"

void MyClass::func(int x)
{
    int z = MyClassImpUtil::g(x);
    // ...
    double w = MyClassImpUtil::f(z,x);
    // ...
}
```

（a）原始组件的.c 文件

```
// myclassimputil.h
#ifndef INCLUDED_MYCLASSIMPUTIL
#define INCLUDED_MYCLASSIMPUTIL

struct MyClassImpUtil {
    static int g(int y);
    static double f(int a, int b);
    // ...
};

#endif
```

（b）新组件的.h 文件

图 6-22　将静态自由函数移到另一个组件中

下一个问题是“什么时候要在一个单个的类中区分两种不同的观众？”通常的回答是：“当某人试图对一个单个的类做很多事情的时候”。

原　则

以一个基类的保护成员函数的形式为派生类的作者提供支持，会将派生类的未绝缘的实现细节暴露给基类的公共客户。

考虑如图 6-23 所示的抽象基类 Shape 的头文件。大概每个派生的 shape 对象都有一个原点和面积，并且知道在一个给定的 Screen 上绘出自身。Screen 对象提供了画线和弧需要的所有功能；但是，编写代码来完成该功能的工作既乏味又容易出错。鉴于此，Shape 基类的作者必须提供一套保护成员函数来帮助派生类作者实现他（或她）自己的特殊 draw 函数。

```
// shape.h
#ifndef INCLUDED_SHAPE
#define INCLUDED_SHAPE

#ifndef INCLUDED_POINT
#include "point.h"
#endif

class Screen;

class Shape {
  public:
    // TYPES
    enum Status { IO_ERROR = -1, SUCCESS = 0 };
```

```
private:
  // DATA
  Point d_origin;
  Status d_drawStatus;

protected:
  // DERIVED CLASS SUPPORT
  static double distance(const Point& start, const Point& end);
  void resetDrawStatus();
  Status getDrawStatus() const;
  void drawLine(Screen *screen, const Point& start, const Point& end);
  void drawArc(Screen *screen, const Point& center, double radius,
                               double startAngle, double endAngle);
private:
  Shape& operator=(const Shape&);          // not implemented
  Shape(const Shape&);                     // not implemented

public:
  // CREATORS
  Shape(const Point& origin);
  virtual ~Shape();

  // MANIPULATORS
  void setOrigin(const Point& origin);

  // ACCESSORS
  const Point& origin() const;
  virtual double area() const = 0;
  virtual Status draw(Screen *screen) = 0;
};

#endif
```

图 6-23 含有对派生类作者的保护性支持的 Shape 类

图 6-24 显示的是一个派生的 Rectangle 类和它使用由基类提供的保护助手函数完成的 draw 函数的实现。Rectangle 只是通过它的左下角和右上角来定义，这样就隐含地迫使 Rectangle 的边必须是水平的或垂直的。派生类的作者也已经定义了左下角，以便与 shape 的原点一致。

为了画一个 Rectangle，我们需要画四条边。如果发生了错误，我们需要从 Rectangle::draw 函数返回 IO_ERROR 函数。我们的第一步是清除 draw 状态。然后确定合适的坐标并且调用必要的保护助手函数。如果在这个过程中发生了错误，这些助手函数将在内部将 draw 的状态设置为 IO_ERROR。当我们完成了所需做的工作时，只是返回 draw 的状态。

这是对于基类作者和派生类作者同样方便的一种完成任务的方式，但是，这样做会对一般客户造成损失，因为编译时耦合会使他们陷入他们不需要也不想要的许多实现细节中。这个情景如图 6-25 所示的组件/类图。

```
// rectangle.h
#ifndef INCLUDED_RECTANGLE
#define INCLUDED_RECTANGLE

#ifndef INCLUDED_SHAPE
#include "shape.h"
#endif

class Rectangle : public Shape {
    Point d_upperRightCorner;

  public:
    // CREATORS
    Rectangle(const Point& lowerLeft, const Point& upperRight);
    Rectangle(const Rectangle& rect);
    ~Rectangle();

    // MANIPULATORS
    Rectangle& operator=(const Rectangle& rect);
    void setUpperRightCorner(const Point& upperRight);

    // ACCESSORS
    const Point& upperRightCorner() const;
    double area() const;
    Shape::Status draw(Screen *screen);
};

#endif
```

```
// rectangle.c
#include "rectangle.h"

// ...

Shape::Status Rectangle::draw(Screen *screen)
{
    resetDrawStatus();
    int x1 = origin().x();
    int y1 = origin().y();
    int x2 = upperRightCorner().x();
    int y2 = upperRightCorner().y();
    drawLine(screen, Point(x1, y1), Point(x1, y2));
    drawLine(screen, Point(x1, y2), Point(x2, y2));
    drawLine(screen, Point(x2, y2), Point(x2, y1));
    drawLine(screen, Point(x2, y1), Point(x1, y1));
    return getDrawStatus();
}
```

图 6-24　派生的 Rectangle 图形及其 Draw 成员函数的实现

在这种情况下，没有什么正当的理由去用只有派生类作者才关心的细节来污染 Shape 类的公共接口，假设不是让每个派生类依赖于基类提供的服务，而是让每个派生类使用一个独立的组件（如果需要）以方便画图。这样，与保护的成员函数相关的不必要的耦合就会消除。

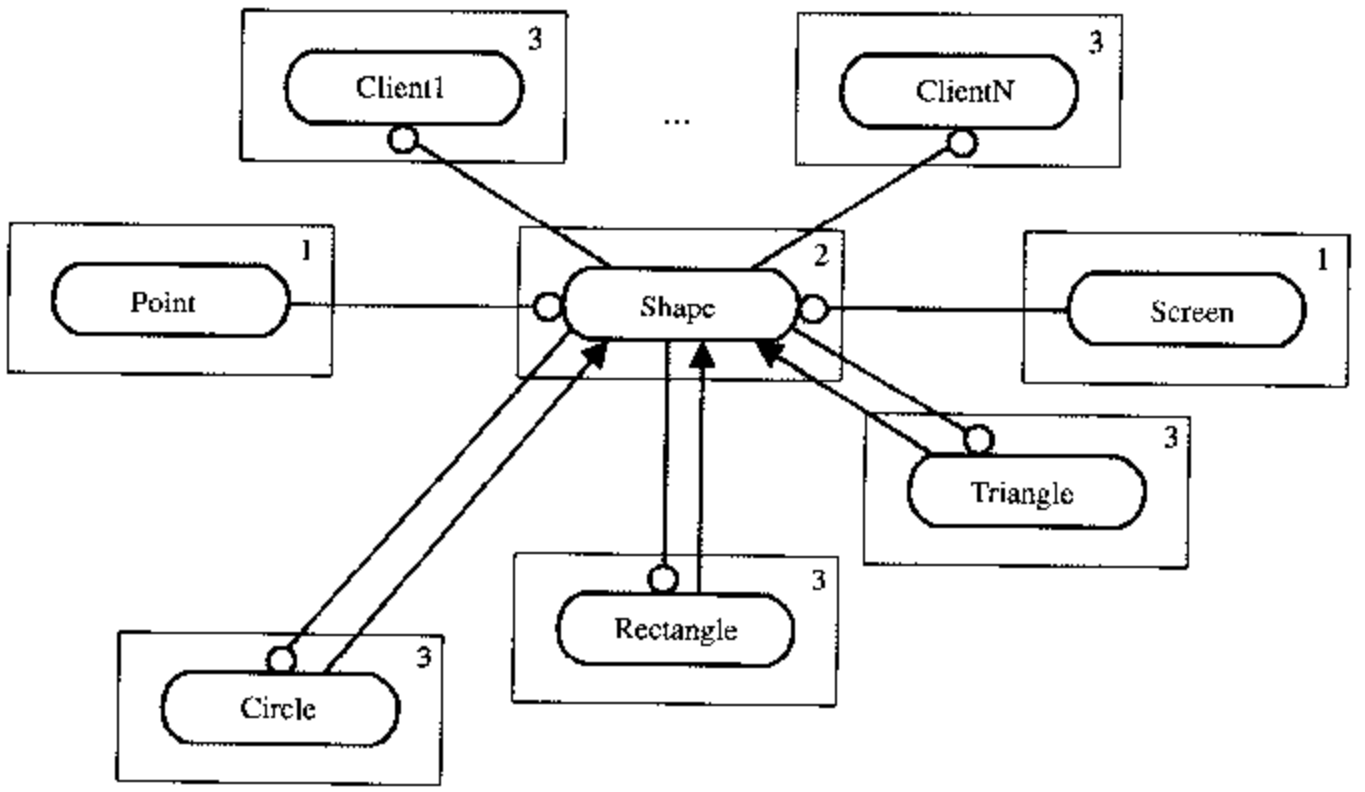


图 6-25 原始 Shape 系统的组件/类图

正如图 6-26 所示，新系统现在被分解了，以便派生类作者使用一般公众不能看见的独立的 scribe 组件。

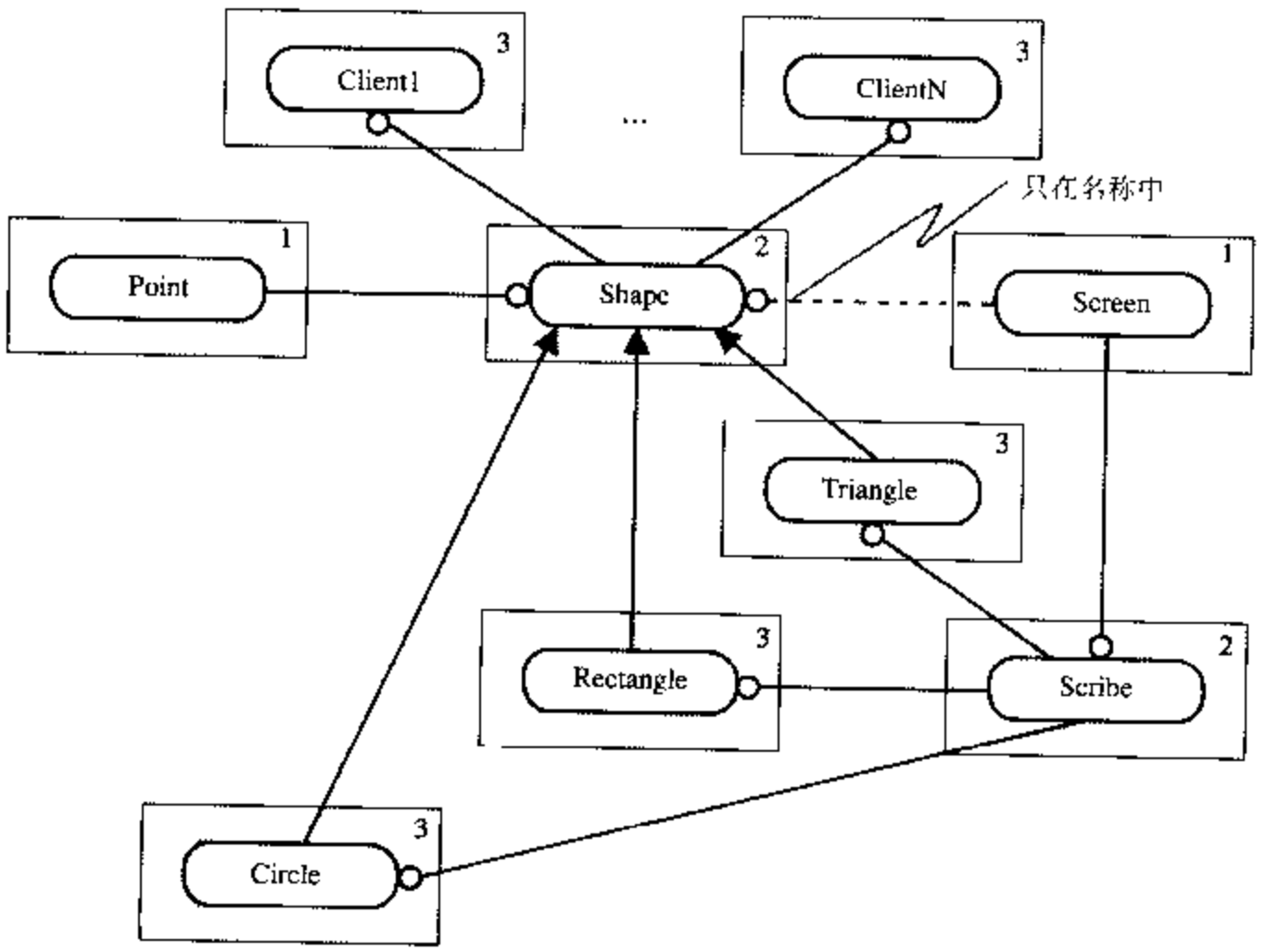


图 6-26 新 Shape 系统的组件/类图

scribe 组件的头文件如图 6-27 所示。由于在这个新组件中提供的功能不再嵌入 Shape，我们决定使它们完全脱钩。画图功能不再以任何方式依赖 Shape，现在这个功能可以很容易地被对象重用，而不是被那些需要在 Screen 上自我绘制的 Shape 的派生类重用。

```
// scribe.h
#ifndef INCLUDED_SCRIBE
#define INCLUDED_SCRIBE

class Screen;
class Point;

class Scribe {
    int d_hadError;

private:
  Scribe& operator=(const Scribe&);          // not implemented
  Scribe(const Scribe&);                     // not implemented

public:
  // STATICS
  static double distance(const Point& start, const Point& end);

  // CREATORS
  Scribe();
  ~Scribe();

  // MANIPULATORS
  void drawLine(Screen *screen, const Point& start, const Point& end);

  void drawArc(Screen *screen, const Point& center, double radius,
                               double startAngle, double endAngle);
  // ACCESSORS
  int hadError() const;
};

#endif
```

图 6-27　新的可重用 scribe 组件方便画图

派生类作者会发现使用 Scribe 类的公共成员不是很困难，但是使用基类的保护成员很困难。由于提供 scribe 组件只是作为一种辅助设施，因此那些认为其功能没有用的客户程序就既不必包含其头文件，也不必在编译时依赖它。Rectangle 重新实现的 draw 函数如图 6-28 所示。基类 Shape 的头文件的新版本如图 6-29 所示。

```
// rectangle.c
#include "rectangle.h"
#include "scribe.h"

Shape::Status Rectangle::draw(Screen *screen)
{
```

```
        Scribe u;
        int x1 = origin().x();
        int y1 = origin().y();
        int x2 = upperRightCorner().x();
        int y2 = upperRightCorner().y();
        u.drawLine(screen, Point(x1, y1), Point(x1,y2));
        u.drawLine(screen, Point(x1, y2), Point(x2,y2));
        u.drawLine(screen, Point(x2, y2), Point(x2,y1));
        u.drawLine(screen, Point(x2, y1), Point(x1,y1));
        return u.hadError() ? IO_ERROR : SUCCESS;
    }
```

图 6-28 Rectangle::Draw 的新实现

```
// shape.h
#ifndef INCLUDED_SHAPE
#define INCLUDED_SHAPE

#ifndef INCLUDED_POINT
#include "point.h"
#endif

class Screen;

class Shape {
    Point d_origin;

  private:
    Shape& operator=(const Shape&);        // not implemented
    Shape(const Shape&);                   // not implemented

  public:
    // TYPES
    enum Status { IO_ERROR = -1, SUCCESS = 0 };

    // CREATORS
    Shape(const Point& origin);
    virtual ~Shape();

    // MANIPULATORS
    void setOrigin(const Point& origin);

    // ACCESSORS
    const Point& origin() const;
    virtual double area() const = 0;
    virtual Status draw(Screen *screen) = 0;
};

#endif
```

图 6-29 消除了保护成员函数的 Shape 类

消除一个类的所有保护成员有时是不可行的。例如当一个派生类需要访问由一个基类提

供的保护性服务以便能够覆盖虚函数时，就是这种情况。

定义某些共享功能的抽象基类有时被称为**部分实现**。这种分解的实现类型允许派生类作者共享一个公共的实现，但是保护的功能再次给基类的一般用户添加了负担，因为它把未绝缘的实现细节暴露给了一般用户。

例如，图 6-30 显示了一个简单的基类，该类既被用于提供一个公共的接口又为 car 类分解公共实现。所有的 car 对象都有一个 location 属性，但是公众不能直接修改 location 属性。取而代之的是，客户必须调用公用成员函数 drive，通过 drive 以各种方式改变 location 属性。改变的方式取决于实际的（派生的）car 类的实现。

```
// car.h
#ifndef INCLUDED_CAR
#define INCLUDED_CAR

class Car {
    int d_xLocation;
    int d_yLocation;

  private:
    Car(const Car&);                        // not implemented
    Car& operator=(const Car&);             // not implemented

protected:
    Car(int x, int y);
    int setXLocation(int x);
    int setYLocation(int y);
        // Only derived classes can set the location of a car directly.
    void move(int deltaX, int deltaY);
    static double distance1(double acceleration, double time);
    static double distance2(double acceleration, double velocity);
    double howFar(int newXLocation, int newYLocation) const;

public:
    // CREATORS
    virtual ~Car();

    // MANIPULATORS
    virtual void drive(/* ... */) = 0;
        // Public clients alter the location of the
        // car by calling the public function drive.

    // ACCESSORS
    int xLocation() const;
    int yLocation() const;
};

#endif
```

图 6-30　包含保护成员函数的 Car 基类

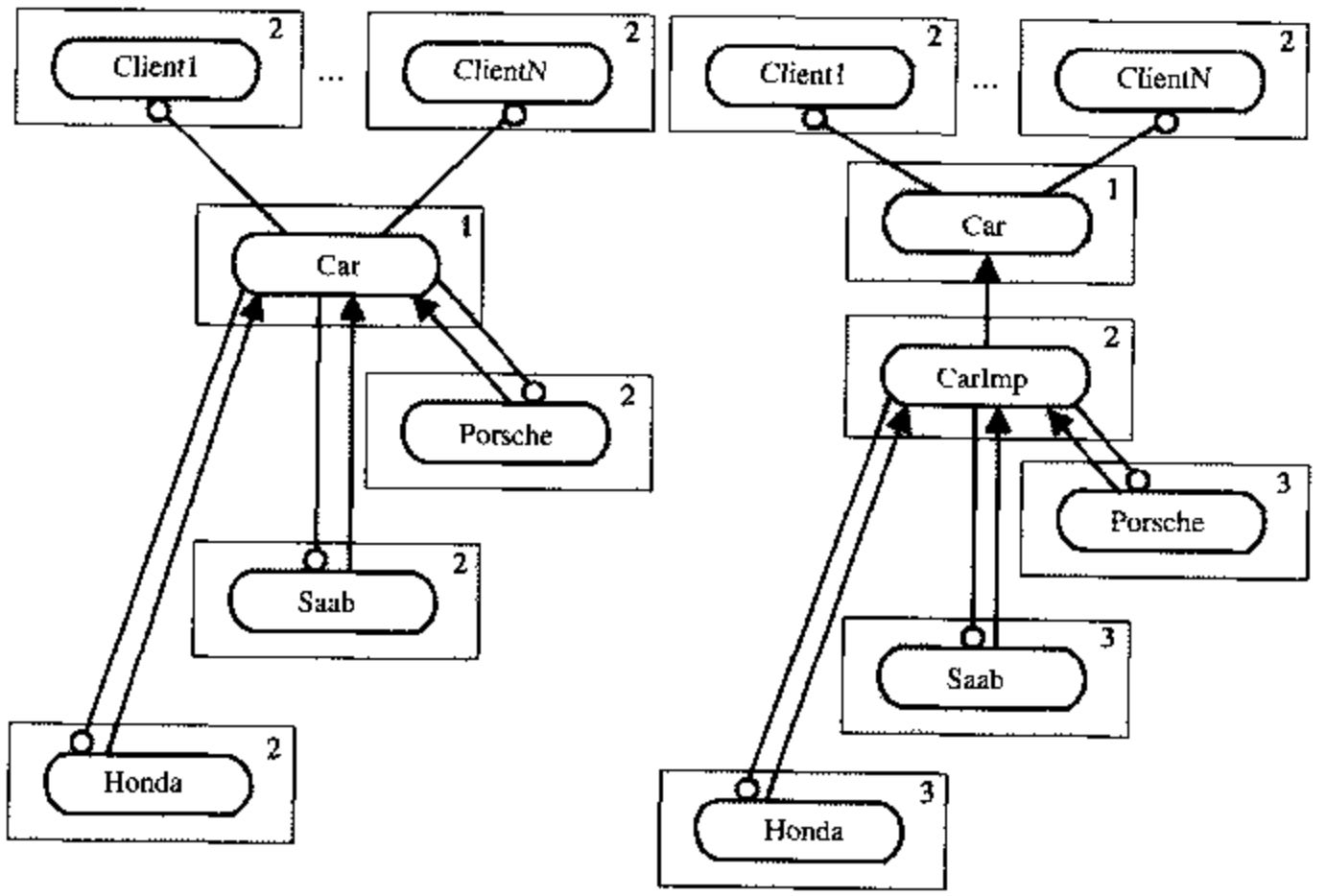
在该基类的保护性接口上提供了几个助手函数，以便帮助派生类作者实现他们自己特定类的 drive 函数。例如，函数 move 获取相对距离并且设置 Car 的新的绝对位置。静态函数 distance1 和 distance2 独立于实例数据，并为物理距离计算提供支持。howFar 访问函数比较当前位置和一个指定的新位置，并返回两点之间的直线距离。

但是，和 Shape 基类不同，Car 的接口定义了一个依赖于 Car 的实际派生类型的纯虚函数 drive，该函数必须反过来使用由其部分实现提供的保护函数来设置 Car 的位置的值。

基类 Car 的设计将接口与至少一部分实现耦合在一起。现在如果一个小汽车制造商要为小汽车开发一种全新的设计，那么它将被迫承担定义在基类中的部分实现的开销，而不管该部分实现是否被使用了。

在 Car 的例子中，一些功能（例如，静态函数和 howFar 函数）当然可以移到一个独立的工具类中，如 Shape 类所做的那样。但是从该基类中分离出部分实现需要更大量的工作。

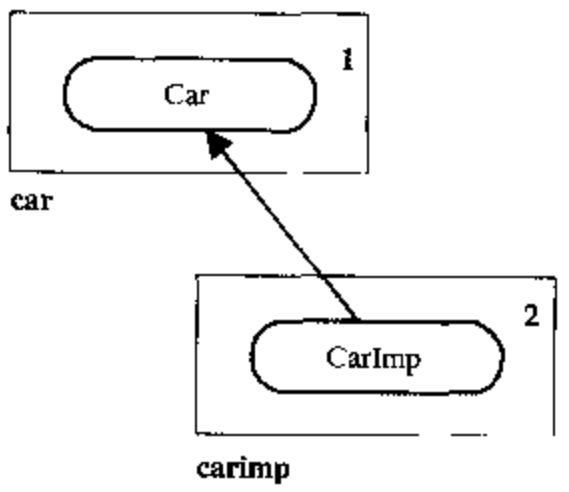
图 6-31（a）说明的是原始的、未绝缘系统的组件/类图。通过将 Car 的纯接口和部分实现分解为两个独立的类（分别为 Car 和 CarImp），我们能够将它们从物理上分开。通过将纯的接口放入一个独立的组件中，我们可以为 Car 的公共客户提供一种绝缘的接口，如图 6-31（b）所示。注意，由于 CarImp 是由 Car 派生而来的，选择共享公共实现的进一步的派生类可以继续这样做。幸好，CarImp 的物理组织变化不能影响 Car 的客户。析取出来的 Car 类的协议如图 6-32 所示。



（a）原始的 Car 层次结构　　（b）绝缘的 Car 层次结构

图 6-31　为 Car 类析取出一个协议

```
// car.h
#ifndef INCLUDED_CAR
#define INCLUDED_CAR

class Car {
  public:
    // CREATORS
    virtual ~Car();

    // MANIPULATORS
    virtual void drive(/* ... */) = 0;
        // Public clients alter the location of the
        // car by calling the public function drive.

    // ACCESSORS
    virtual int xLocation() const = 0;
    virtual int yLocation() const = 0;
};

#endif
```




```
// carimp.h
#ifndef INCLUDED_CARIMP
#define INCLUDED_CARIMP

class CarImp : public Car {
    int d_xLocation;
    int d_yLocation;
    // ...
  public:
    // ...
    // ACCESSORS
    int xLocation() const;
    int yLocation() const;
};

#endif
```

图 6-32　Car 类的协议和部分实现

为了使一般用户与所有的实现细节绝缘，我们所要做的是析取一个纯的接口（本书称为协议）。对于同时获得层次化和绝缘性来说，析取协议是一种非常普遍而强有力的技术。协议类以及如何析取它们是 6.4.1 节的主题。

6.3.5　消除私有成员数据

正如读者可以回忆起来的，在上一节中我们通过引入一个独立的工具支持派生类中的 draw 函数的实现，使我们能够从基类 Shape 中消除所有的保护成员。但是基类 Shape 仍然包含私有数据。

消除私有**静态**成员数据相对容易。图 6-33（a）显示了一个私有静态的整数数据成员 s_count，用于跟踪记录 MyClass 类的活动实例的数量。只要内联成员函数（或远距离友元）

不需要直接的访问，通常有可能将静态成员数据移到在组件.c 文件的文件作用域中定义的一个静态变量中[①]。消除非静态成员数据则相当棘手。

```
// myclass.h
#ifndef INCLUDED_MYCLASS
#define INCLUDED_MYCLASS

class MyClass {
    static int s_count;
    // ...
  public:
    // ...
};

#endif
```

```
// myclass.c
#include "myclass.h"
int MyClass::s_count;
// ...
```

（a）带有私有静态成员数据的原始类

```
// myclass.h
#ifndef INCLUDED_MYCLASS
#define INCLUDED_MYCLASS

class MyClass

    // ...
  public:
    // ...
};

#endif
```

```
// myclass.c
#include "myclass.h"
static int s_count;
// ...
```

（b）带有静态文件作用域数据的修改后的类

图 6-33 消除私有的静态成员数据

正如我们在 6.3.4 节所讨论的，改变这种封装的私有数据将迫使基类 Shape 的所有公共的客户程序重新编译。正如在上一节中对类 Car 所做的那样，我们可以将 Shape 分解成两个类，一个包含纯接口，另一个包含部分实现（包括原始数据的定义）。

分解后的 Shape 类的层次结构的组件/类图如图 6-34 所示。这种结构有两个明显的优点：

（1）Shape 类的客户程序与由 Shape 派生的实际对象的所有实现细节是绝缘的。

（2）有可能派生一个全新的 Shape 类子类型，而不会引起任何与定义在 ShapeImp 类中的部分实现相关的开销。

类 Shape 不再嵌入一个 Point 实例，因此 Shape 类的客户程序不会再因为使用 Shape 类而被迫包含 Point 定义。通过派生于 ShapeImp 而不是派生于 Shape，派生类可以继续共享 Shape 的部分实现。与往常一样，绝对没有与扩展继承层次结构的深度有关的额外运行时开销。惟一的额外开销是静态绑定的成员函数 origin 和 setOrigin，它们现在必须通过虚拟调用机制（见 6.6.1 节）来调用。

我们可以决定试用 Shape 的一个候选部分实现，即 MyShapeImp，该类利用了一对 short int 数据成员（而不是利用 Point）来保存原点的内部表示。原来的结构不支持这种程度的重新实现。即使 origin 和 setOrigin 成员函数声明为虚拟的，原来的结构也会迫使每个实例携带一个额外的 Point 类型的数据成员。

---

① 在极少数情况下，允许组件有多于一个.c 文件，可以使可重用库的开发者能够基于使用模式来对成员函数定义进行分区，以便减少一般客户程序的运行时大小。允许函数通过定义在.c 文件中的静态变量进行通讯，减少了将一个类的单个成员函数分割成独立的编译单元（.c 文件）的灵活性。

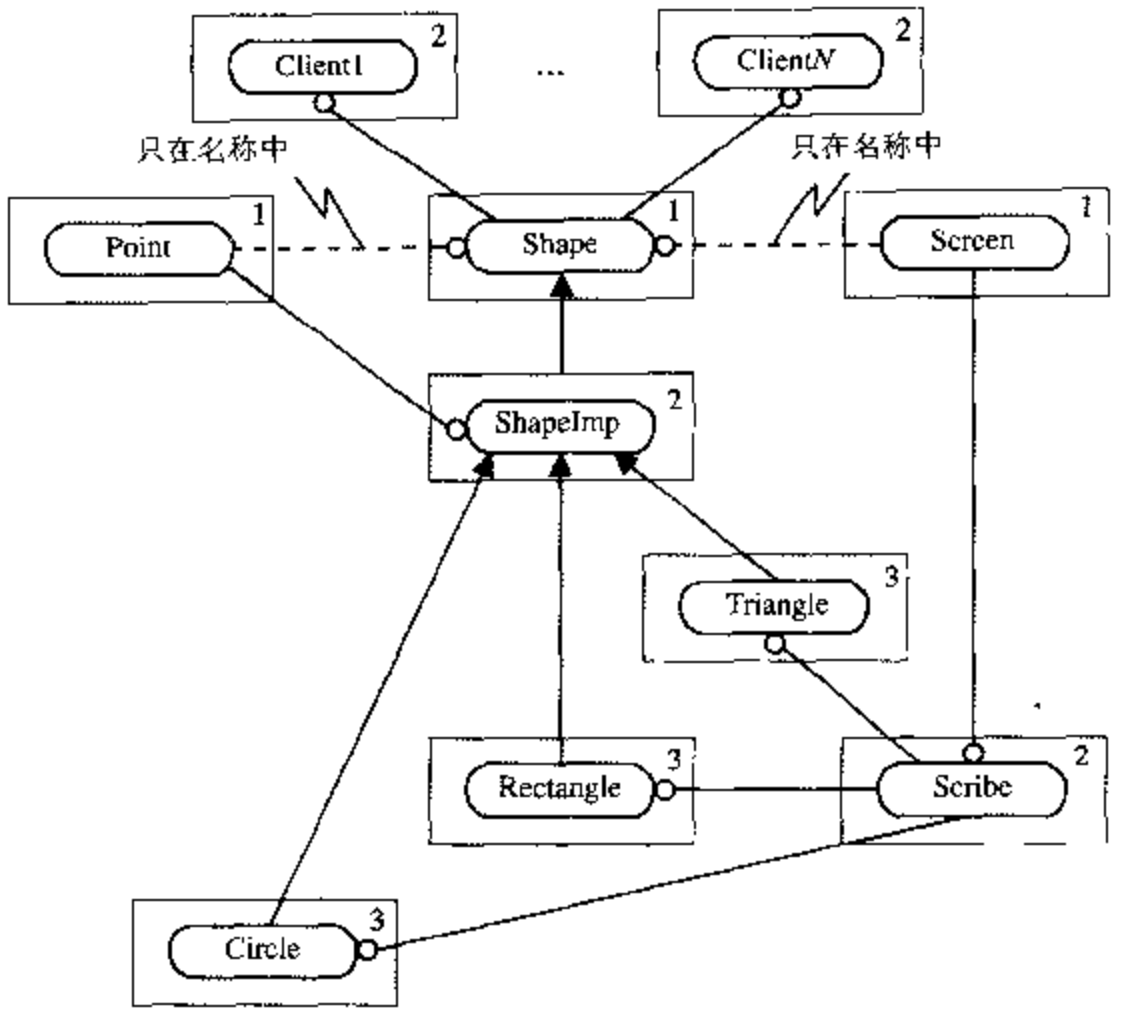


图 6-34 分解的 Shape 子系统的组件/类图

对于新的分解结构，实现的选择没有限制。我们现在可以提供 Shape 类的一个候选的有效的部分实现，它直接派生于类 Shape。从 ShapeImp、MyShapeImp 派生的特定而具体的 shape，或者直接从 Shape 本身派生的 shape 可以共存于同一个运行系统中，而不会影响其他的 shape 或客户程序。

一个任意 shape 的协议类如图 6-35 所示。即使 Shape 类现在将其公共客户程序与所有的实现细节绝缘，我们仍然选择在一个独立的组件中保持对 drawing 的支持，有以下两个原因：

（1）正如前面所提到的，支持 drawing 作为一个独立于 Shape 层次结构的屏幕工具，使得它在 rendering 对象中能够重用，而在那些派生于 Shape（或 ShapeImp）的对象中则不能。将这些支持函数嵌入到 ShapeImp 中，将使特定的部分实现与具有更普遍用途的功能结合起来。如果 drawing 支持定义在 ShapeImp 类中，那么 MyShapeImp 就不可能（例如）独立地使用对 drawing 的这个支持。

（2）scribe 组件提供了一种可选的服务，并且它不是部分实现的一个本质属性。对这种功能没有需求的派生类作者不但应该与该组件绝缘，而且在测试期间甚至不必与之连接。

6.3.6 消除编译时产生的函数

改变任何编译器产生的函数的定义意味着修改类定义来添加相应的声明，任何这样的修改都将迫使该类的所有客户程序重新编译。而允许编译器产生一个拷贝构造函数、赋值运算

符以及一个析构函数（如果需要）可能会很方便，一个真正绝缘的类必须显式地定义这些成员。通常，这些显式定义的函数将复制它们的默认行为。特别地，析构函数经常被定义为有一个空的实现。这是灵活性的代价。（声明这些特殊函数的其他原因，见 9.3.2 节和 9.3.3 节。）

```
// shape.h
#ifndef INCLUDED_SHAPE
#define INCLUDED_SHAPE

class Point;
class Screen;

class Shape {
  public:
    // TYPES
    enum Status { IO_FRROR = -1, SUCCESS = 0 };

    // CREATORS
    virtual ~Shape();

    // MANIPULATORS
    virtual void setOrigin(const Point& origin) = 0;

    // ACCESSORS
    virtual const Point& origin() const = 0;
    virtual double area() const = 0;
    virtual Status draw(Screen *screen) = 0;
};

#endif
```

图 6-35 一个 Shape 的协议

6.3.7 消除 Include 指令

不必要的include指令可能会在本不存在编译时耦合的地方引起编译时耦合。一个#include指令出现在一个组件的头文件中，通常有三种情况：

（1）IsA：在这个组件中的一个类派生于在被包含的文件中定义的一个类。

（2）HasA：该组件中的一个类嵌入了一个定义在被包含文件中的类的一个实例。

（3）Inline（内联）：在该组件的头文件中声明为内联的函数实质使用了一个定义在被包含文件中的类。

偶尔，一个包含局部连接结构（例如，类作用域中的 enum 或 typedef）的头文件，可能是在另一个头文件中包含一个头文件的另一个似乎合理的藉口。但是，在其他情况下放置一个#include 指令到一个头文件中很少是有道理的。

正如我们在前面讨论 Bank 例子时介绍到的（6.2.7 节），bank 组件作者的包含每种外币的决策对于 Bank 类的客户程序没有什么好处。这些货币（以名称）出现在接口上的事实决不会

隐含这样的意思：Bank 的客户程序需要知道它们的定义才能很好地使用 Bank 类。Person 类对这些外币的人工编译时依赖只是嵌套#include 指令的结果。

转换是简单的：将头文件中所有不必要的包含指令移动到.c 文件中，并且用适当的（“提前的”）类声明替换它们。该类声明告诉客户程序的 C++编译器，货币代表某个用户自定义的对象类型，但不提供任何有关其内部布局的细节。Bank 的客户程序现在与它们不使用的类型的变化绝缘了。Bank 组件的容易运行的绝缘版本如图 6-36 所示。

```
// bank.h
#ifndef INCLUDED_BANK
#define INCLUDED_BANK

class BankCard;                 // class declaration instead of #include
class GermanMarks;              // class declaration instead of #include
class JapaneseYen;              // class declaration instead of #include
class UnitedStatesDollars;      // class declaration instead of #include
class EnglishPounds;            // class declaration instead of #include

// ...
// ...
// ...

class LakosianFooBars;

class Bank {
    // ...
    Bank(const Bank&);                          // We don't want to copy
    Bank& operator=(const Bank&);               // or assign banks.

  public:
    // CREATORS
    Bank();
    ~Bank();

    // MANIPULATORS
    GermanMarks          getMarks(BankCard *cashMachineCard, double amount);
    JapaneseYen            getYen(BankCard *cashMachineCard, double amount);
    UnitedStateDollars getDollars(BankCard *cashMachineCard, double amount);
    EnglishPounds       getPounds(BankCard *cashMachineCard, double amount);
    // ...
    // ...
    // ...
    LakosianFooBars    getFooBars(BankCard *cashMachineCard, double amount);
};

#endif
```

图 6-36　在接口中使用多种类型来绝缘类

一般来说，在任何行得通的地方，消除一个内联函数或改变一个数据成员，以便使一个头文件中的#include 指令成为不必要的，都会获得以减少的编译时耦合的方式表现出来的确实

的利益。如果这个不必要的#include 指令被消除了，以前依赖这个头文件来包含另一个头文件的客户程序现在将不得不修改成直接包含那个头文件。

6.3.8 消除默认参数

很容易从一个接口消除默认参数并用等价的单个函数来替代它们[①]：

```
class Circle {
    // ...
  public:
    Circle(double x = 0, double y = 0, double radius = 1);
    // ...
};
```

我们可以将上面的接口修改为更绝缘的版本，如下所示：

```
class Circle {
    // ...
  public:
    Circle();
    Circle(double x)                    // do we really want this?
    Circle(double x, double y);
    Circle(double x, double y, double radius);
    // ...
};
```

上面的例子说明，在消除默认参数时，不是每个参数都要有相应的函数来替代，我们可以去掉其中的一个或多个。上面反映了我们可以决定不提供同样的功能，并消除一个或更多用默认参数自动创建给我们的选择。

有时我们可以消除编译时耦合，并通过解释一个无效的选择值（例如，一个空指针、一个 0size 或一个负的索引）在函数体中保留默认参数的因数（factoring）。回忆一下，在 p2p_Router 的接口中（图 4-2）有一个函数 findPath 使用了第一个“可选参数”，该参数是存储结果的地址：

```
class p2p_Router {
    // ...
  public:
    // ...
    int findPath(geom_Polygon *returnValue, const geom_Point& start,
                 const geom_Point& end, int width) const;
};
```

通过重新安排参数的顺序，我们可以使这个参数完全是可选的，而不用在接口中对任何没有绝缘的值进行硬编码。

```
class p2p_Router {
    // ...
  public:
```

① **ellis**，8.2.6 节，142 页。

```
    // ...
    int findPath(const geom_Point& start, const geom_Point& end,
                 int width, geom_Polygon *returnValue = 0) const;
};
```

默认参数将在 9.1.10 节进一步讨论。

6.3.9 消除枚举类型

接口中的枚举类型很自然也会引起编译时耦合。枚举类型、typedef 以及其他在接口上有内部连接的结构的合理使用，对于获得良好的绝缘是非常重要的。

考虑图 6-37 中显示的三种不同的枚举类型。第一种是类的一个私有实现细节，第二种是一个公共的可访问常量值，而第三种是一个命名的返回状态值的枚举列表。

```
// whatever.h
#ifndef INCLUDED_WHATEVER
#define INCLUDED_WHATEVER

class WhatEver {
    enum { DEFAULT_TABLE_SIZE = 100 };                  // 1

  public:
    enum { DEFAULT_BUFFER_SIZE = 200; };                // 2

    enum Status { A, B, C, D, E, F, G, H, I, J };       // 3

    Status doIt();
};

#endif
```

图 6-37 一个包含三种不同类型的枚举类型的类

图 6-37 中的第一个枚举类型放得不合适（除非我们在头文件中需要一个编译时常量——例如，为了实现一个固定数组的界）。这种枚举类型要么应该移到.c 文件的文件作用域中，或者如果需要，变成该类的一个私有静态 const 成员。把这个数表达为一个静态的类数据成员，不仅使内联函数和定义在这个变换单元之外的带友元状态的函数可以通过编程访问其值的可编程通道，而没有在头文件中暴露一个“魔幻数字”。

第二个枚举类型至少应该变成一个私有的静态 const 类成员，并且将一个公共的静态（也许内联）访问成员函数定义为返回该值。与大多数绝缘技术一样（见 6.6.1 节），我们为了减少编译时耦合而损失了运行时性能。在这种情况下，一个优化的编译器可以利用所知的编译时常量，例如在文件作用域中的声明为 const 的基本数据、枚举值以及直接量。通过在指令流中直接存储实际的值（而不是地址），可以避免一个额外的间接层次。但是，按照定义，这些编译时常量不能与客户程序绝缘。因此，对它们的任何改变将不可避免地迫使客户程序重新编译。如果通过这个接口访问各层组件时的性能太差，那么这个组件所处的层次可能太低了，

不能考虑作为绝缘的一个好的候选者。

原　则

给较高层次的客户程序授予修改较低层次共享资源的接口的权利，会隐含地耦合所有的客户程序。

第二个枚举类型很明显是接口的一部分。也许不是所有这些状态值都是通过这个组件中的函数返回的，而是选择该组件来为其他组件保存状态值。但是，为了减少编译时依赖，一个更好的方法是，将这些状态值分散到合适的组件中并且不要试图去重用它们。若允许这个枚举类型独立于该物理层次结构中的较高层次部分，那么分散这些枚举状态值可极大地减少耦合。局部地定义返回值会有另外的意义，不会将细微不同的意义强加到已经存在的状态值中。每个状态值的含义对于当前对象来说都是局部的，并且完全适合其目标要求。重用状态值只是重用的益处大于随后因耦合而产生的坏处的另一种情况。一种替代图 6-37 中定义的可能方案如图 6-38 所示。

```
// whatever.h
#ifndef INCLUDED_WHATEVER
#define INCLUDED_WHATEVER

class WhatEver {
    static const int s_defaultBufferSize;          // 2
  public:
    static int getDefaultBufferSize();             // 2
    enum Status { A, B, C };                       // 3
    Status doIt();
};

inline int getDefaultBufferSize()                  // 2
{
    return s_defaultBufferSize;
}

#endif
```

（a）whatever.h 头文件

```
// whatever.c
#include "whatever.h"

enum { DEFAULT_TABLE_SIZE = 100 };                 // 1

const int WhatEver::s_defaultBufferSize = 200;     // 2

WhatEver::Status WhatEver::doIt() { /* ... */ };
```

（b）whatever.c 实现文件

图 6-38　图 6-37 的三种枚举类型的可选定义

通过改为传送整数或字符串，有可能避开接口上的枚举类型的编译时耦合。这种方法确实消除了编译时耦合。但是，若特意在一个函数的接口上用一个枚举类型作为参数，这可能是一种有用的耦合形式，它有助于确保程序的一致性；绝缘所要消灭的并不是这种耦合。

考虑一个函数，它以一个字符串的形式返回一个“坏的”状态值。客户程序必须知道字符串的准确形式。因为这个值是绝缘的，对于客户程序来说，甚至首次确定这个字符串可能也是一个挑战。现在，假设一个返回的字符串恰好从 ioError 改为 IO_ERROR。没有任何一种编译器能帮助客户程序追踪到所有需要改变所调用的例行程序中的比较值的地方。即使忽略改变的可能性，不可避免的拼写错误也肯定探测不到。

一般来说，绝缘的目标是保护客户程序不受某种编译时依赖的影响，这种编译时依赖是因为知道了不必要的、封装的实现细节而引起的；绝缘并不意味着不能通过编程访问接口来访问客户程序，也不是为了保证类型安全。

6.4 整体的绝缘技术

在一个认真规划、精心构造的系统中，我们可以预先知道哪些接口是公共的和哪些接口不是公共的。这种知识可以帮助我们决定哪些接口应该绝缘和哪些接口不应该绝缘。从一开始就把一个接口设计为绝缘的，总是要比过后再去绝缘它更容易，代价也更低。

在实践中，开发者可能没有考虑到他们的设计决策可能出现的所有情况。有时有必要把一个设计得特别差的类与系统的其余部分进行绝缘，但是应用单个的绝缘技术可能会是枯燥的，并且要花费不必要的代价。

幸好有整体的技术可以使一个类、组件、甚至整个子系统的实现与其接口保持距离，且不会扰乱其正在运作的实现。在这些技术后面的物理动机在一些其他的有关 C++的书[①]上可以找到。这些技术经常是由一个完整的逻辑视图[②]激发的。许多技术会引入一个或多个新的组件作为实现的绝缘接口。使用这些技术，我们有时可以改进一个草率的接口的质量，使其能达到它首次出现时就应该达到的标准。

6.4.1 协议类

在理想的情况下，一个绝缘良好的接口绝对不会定义实现细节。它只是定义一个接口，通过这个接口，客户程序可以访问和操纵具体的派生类的实例[③]。

---

① **meyers**，Item 34，111~116 页；**murray**，3.3 节，72~74 页。

② **gamma**，Abstract Factory，第 3 章，87~96 页；Facade，第 4 章，185~194 页。

③ 这种需求有时会不很严格地允许在语言之外支持运行时类型信息（RTTI）。这将在附录 A 中讨论。

> **定 义**：满足下列条件的**抽象类**是一个**协议类**：
>
> （1）它既不包含也不继承那些包含成员数据、非虚拟函数或任何种类的私有（或保护的）成员的类；
>
> （2）它有一个非内联虚析构函数（定义了一个空实现）；
>
> （3）所有成员函数（除了包含被继承函数的析构函数）都声明为纯虚的，并任其处于未定义的状态。

协议类是一个**抽象类**，它没有用户专用的构造函数，没有数据，只有公共成员。除了该协议所继承的其他协议（见附录 A）的那些头文件之外，组件本身并不包含任何其他头文件。所有的成员函数（除了析构函数之外）都声明为纯虚函数。许多编译器需要至少一个非内联函数的实现，以便知道把虚函数表（见 9.3.3 节）放到哪个编译单元。由于析构函数是没有声明为纯虚函数的惟一的成员函数，所以它是在一个协议类中实现非内联的惟一可行的候选函数。

原 则

一个协议类几乎是一个完美的绝缘器。

图 6-39 说明了一个简单的文件抽象的一个协议。这个抽象的.c 文件几乎是空的，只包含以下三行：

```
// file.c
#include "file.h"
File::~File() {}                    // defined empty and out-of-line
```

注意，将地址编码为一个整数而不是一个枚举类型的值，使我们可以在不要求已存在的客户程序重新编译的情况下添加新的整数值。同样的道理，我们也可能在不能够在编译时探测一致性的情况下消除或改变这些值。若消除编译时耦合是以编译时的类型检查为代价的，那就不是我们所希望的。

```
// file.h
#ifndef INCLUDED_FILE
#define INCLUDED_FILE

class File {
  public:
    // TYPES
    enum From { START, CURRENT, END };

    // CREATORS
    virtual ~File();                    // not pure virtual!

    // MANIPULATORS
    virtual void seek(int distance, From location) = 0;
```

```
        virtual int read(char *buffer, int numBytes) = 0;
        virtual int write(const char *buffer, int numBytes) = 0;

        // ACCESSORS
        virtual int tell(From location) = 0;
};

#endif
```

图 6-39 一个文件的协议

我们选择在接口上显式地定义有效的地址值的集合。因此在类 File 中把它们枚举出来是合适的。这种枚举决不是一种实现细节；严格地说它是类 File 的逻辑接口的一部分。即，添加或改变枚举值就像添加或改变虚函数的集合——所有派生类和所有客户程序都将被迫重新编译。

类 File 是抽象的：它定义了一个完全的接口但没有定义实现。例如，程序员不能在程序堆栈上创建一个 File 类型的对象作为一个自动变量。在某个地方，某个程序员必须从 File 派生一个具体的实现类并对它进行实例化。可能要使用一个管理组件（例如图 6-40 所示的这个）来追踪文件。

```
// filemgr.h
#ifndef INCLUDED_FILEMGR
#define INCLUDED_FILEMGR

struct FileMgr {
    static File *open(const char *filename);
};

#endif
```

图 6-40 一个文件管理组件的头文件

在一个系统中的一个或多个客户程序可能会调用 FileMgr 类，以便创建 FileImp 的一个实例——一个派生于 File 协议类的具体实现类。该类一旦产生，一个指向这个实现对象的指针就可以作为一个指向 File 类型的对象的指针在系统中传递，它对其实现没有任何编译时依赖。

图 6-41 说明了一个完全与其实现绝缘的、使用 File 类型的系统。SubSys1 类是系统中负责实例化 File 类型的新对象的部分，因而它在连接时而不是编译时依赖于类 FileImp。SubSys2 和 SubSys3 只是使用 File 协议。这些组件在编译时和连接时都不依赖于 FileMgr 甚至 FileImp。这样，组件 subsys2 和 subsys3 都可以独立于 FileMgr 进行测试。如果在一个测试驱动程序中提供给 File 协议一个合适的桩实现类，那么这些组件甚至可以独立于 FileImp 进行测试。

原 则

一个协议类可以用来消除编译时依赖和连接时依赖。

正如我们在图 5-32 中所看到的带有库子系统的例子，析取一个协议可用来打破循环连接时依赖。通过将 Report 的接口与其实现在物理上分离，我们允许 StatUtil 依赖更低层的 Report 协议，而只有更高层的 ReportImp 部分实现向后依赖于 StatUtil。这里新颖而重要的是，较高层次实现组件的变化——甚至它的头文件中的变化——绝对不会对在该协议的同一层或更低层的客户程序有编译时影响。

有时我们会遇到一个将它的一些函数声明为 virtual 的可实例化的基类。这种类经常包含私有数据。有时它也会包含私有的或保护的函数。有些成员函数可能会声明为内联的。这种类可能包含准备给派生类使用的静态函数和枚举类型。它还可以包含准备给派生类使用的保护的（甚至私有的）虚函数。这种类甚至可以派生于或嵌入那些通过类的公共接口不能编程访问的其他类的实例。简言之，在这个类中可以有比公共客户程序需要知道的多得多的内容。

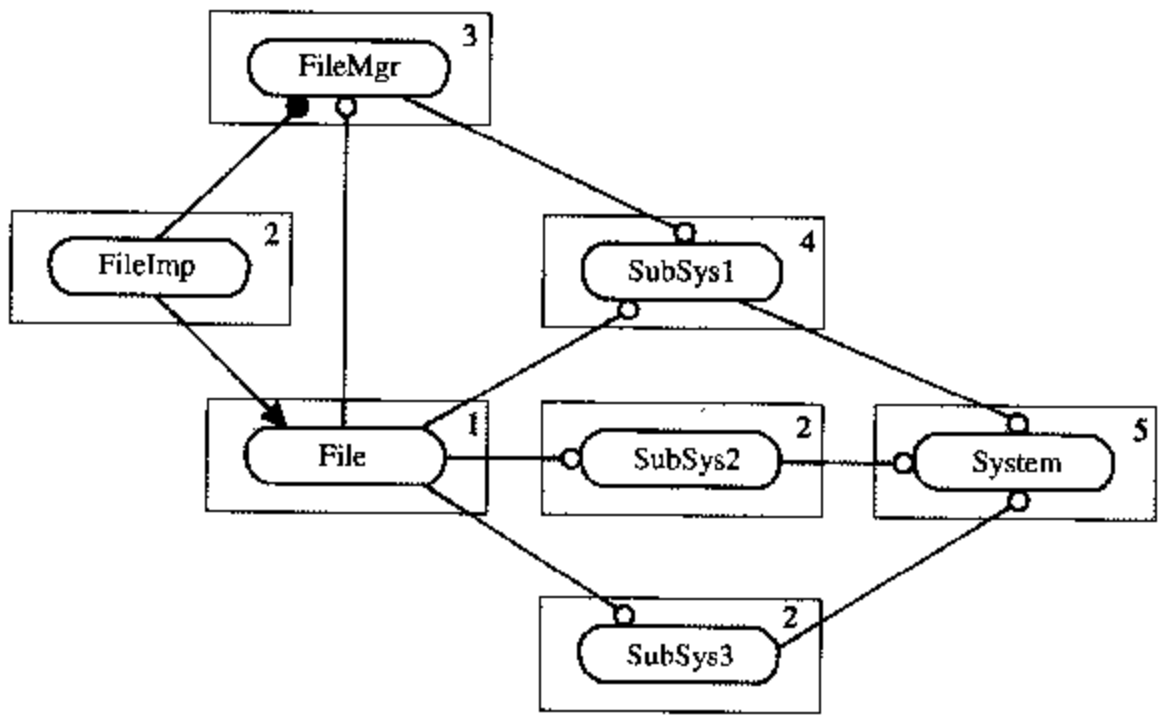


图 6-41　使用一个 File 协议的系统

考虑一个名为 Elem 的可实例化的基类，它符合上段的描述，其用法如图 6-42 所示。Elem 的公共接口在整个系统中被客户程序广泛地用于操纵 Elem 类型的对象（或从 Elem 派生的对象）。系统的体系结构已经仔细地隔离了 Elem 对象的创建，该对象只有单个客户 Client1。

不幸的是，基类 Elem 中天生缺乏绝缘机制，对类 Elem 的所有客户程序暴露了许多上面描述的不必要的封装实现细节。很明显，Elem 基类的设计远不是完美的，并且最好重新设计。重新设计（像第一次设计一样）需要审慎的思考和努力。现在，通过从类 Elem 中析取一个协议，我们可以把一般的公共用户与不必要的细节绝缘。

如图 6-43 所示，在较低层次上建立一个协议类，然后将静态功能和构造函数功能升级到更高层次上的一个工具类中。协议仅包含访问和操纵从 Elem 派生而来的类型的实例所需要的信息。工具类支持所有的静态方法，包括对创建 Elem 的派生类型的具体实例的绝缘。

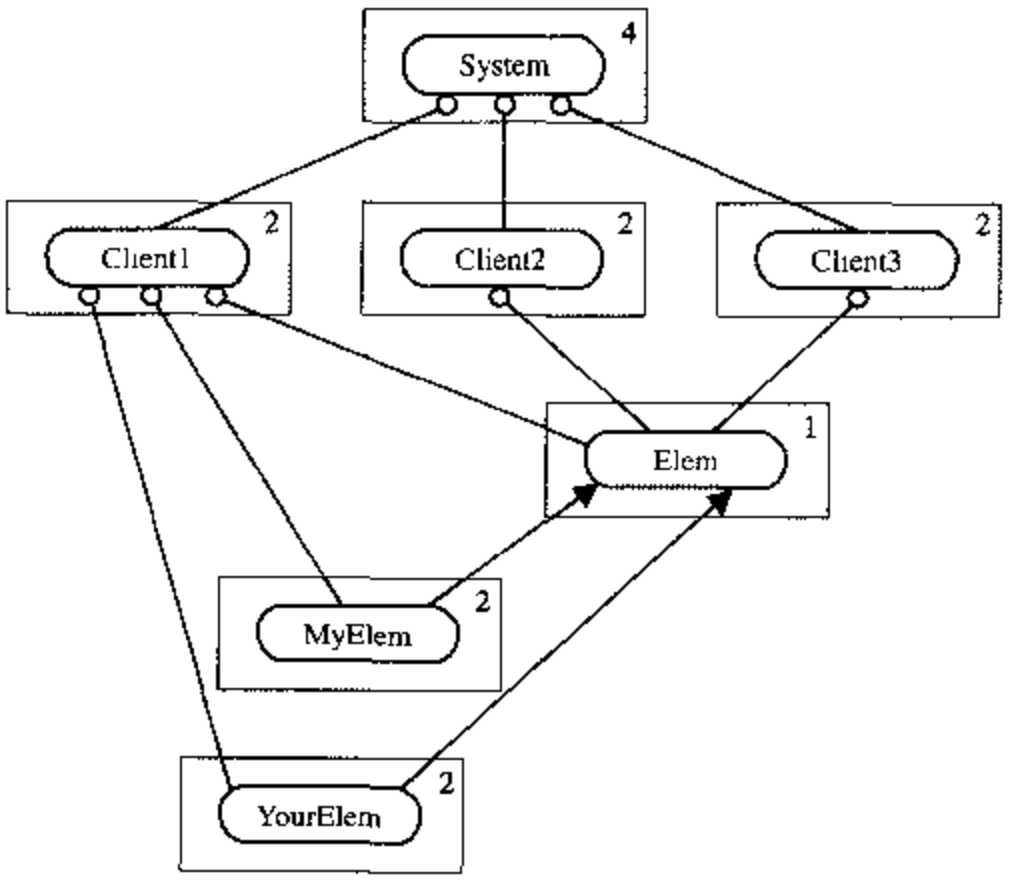
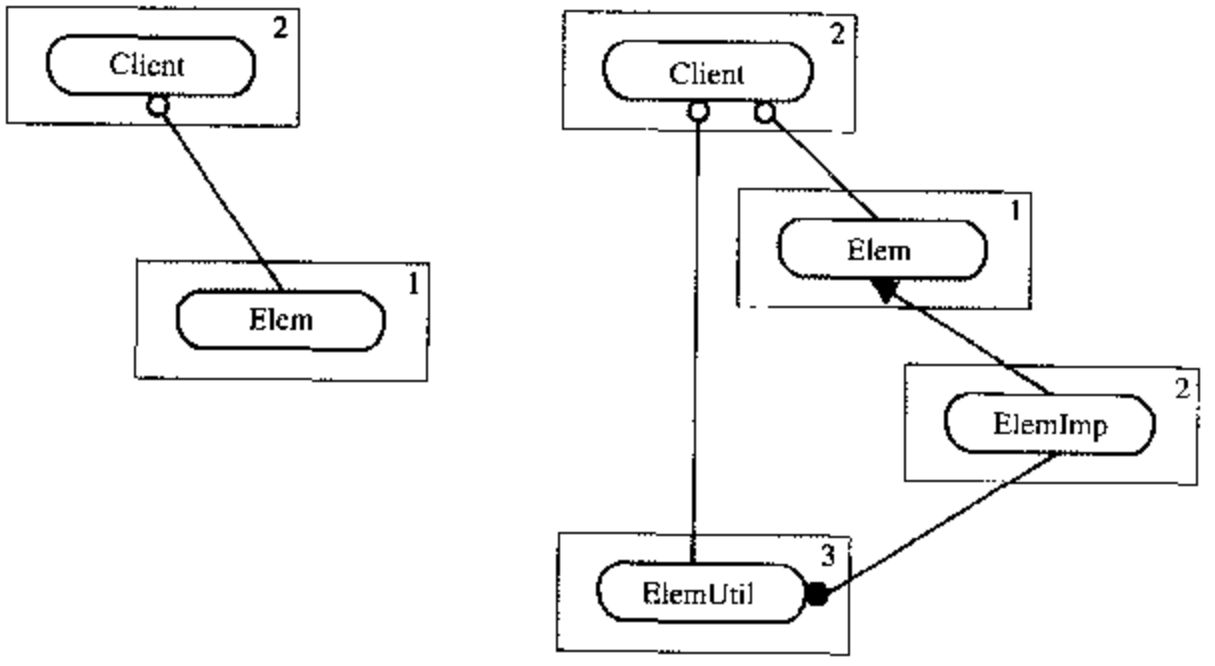


图 6-42　使用一个非绝缘 Elem 基类



（a）单个非绝缘的组件　　（b）多个绝缘组件

图 6-43　为基类 Elem 析取一个协议

考虑类 Elem 的原始头文件，如图 6-44 所示。

```
// elem.h
#ifndef INCLUDED_ELEM
#define INCLUDED_ELEM
#ifndef INCLUDED_FOO
#include "foo.h"
```

```
#endif

#ifndef INCLUDED_BAR
#include "bar.h"
#endif

class Elem {
    Foo d_fooPart;
    Bar d_barPart;

  private:
    // ...
  protected:
    // ...
  public:
    enum Status { GOOD = 0, BAD, UGLY };
    Elem();
    Elem(const Foo& fooPart);
    Elem(const Bar& barPart);
    Elem(const Foo& fooPart, const Bar& barPart);
    Elem(const Elem& elem);
    virtual ~Elem();
    Elem& operator=(const Elem& elem);
    static double f1() { /* ... */ };
    static void f2(double d);
    Foo f3() const { /* ... */ };
    void f4(const Foo& foo);
    virtual const char *f5() const;
    virtual void f6(const char *name);
    virtual Status f7();
};

#endif
```

图 6-44 一个高度非绝缘的 Elem 类的原始头文件

我们可以像这样从类 Elem 中析取协议：

（1）拷贝已存在的组件 elem（包括基类 Elem）到一个新的名字 elemimp，并把被包含的类的名字改为 ElemImp。任何以前直接从 Elem 类继承的类，现在应该改为直接从 ElemImp 继承。这种修改要求调整所有派生类的类定义的继承部分，并要求调整包含一个或多个这种类的每个组件的#include 指令。派生类构造函数的初始化列表也可能需要作某些调整。注意，Elem 的类型参数和所有已存在的 ElemImp 的非构造函数成员的返回值都应该保持 Elem 类型（即，不应该改为 ElemImp 类型）。

（2）删除原始类 Elem 的几乎所有东西（除了公共接口）。如果在类作用域中指定的枚举类型或 typedef 类型被用于 Elem 的一个或多个非静态的、公共的函数的接口中，那么它们应该保留。

（3）从该类中删除构造函数和所有其他静态成员函数，但必须确保留下一个声明为非内

联且定义为空的虚拟析构函数。

（4）使所有在类 Elem 中保留下来的成员函数成为纯虚函数，并删除它们的定义。

（5）从 elem 组件中删除所有#include 指令。当一个用户自定义类型被用在一个纯虚函数的接口上时，提供一个“提前的”类声明。现在一个新的绝缘的 Elem 类如图 6-45 所示。

```
// elem.h
#ifndef INCLUDED_ELEM
#define INCLUDED_ELEM

class Foo;

class Elem {
  public:
    enum Status { GOOD = 0, BAD, UGLY };
    virtual ~Elem(); // defined out-of-line and empty
    virtual Elem& operator=(const Elem& elem) = 0;
    virtual Foo f3() const = 0;
    virtual void f4(const Foo& foo) = 0;
    virtual const char *f5() const = 0;
    virtual void f6(const char *name) = 0;
    virtual Status f7() = 0;
};

#endif
```

图 6-45　新的绝缘的协议组件 elem

（6）把类 ElemImp 修改为直接公共继承自类 Elem。基类的头文件 elem.h 现在应该直接包含在部分实现的头文件 elemimp.h 中。每个公共的非静态成员函数现在都声明为虚拟的，并且尽可能（尽管不是必须）声明为非内联的。注意虚拟赋值运算符

```
virtual Elem& operator=(const Elem& elem)
```

的实现（现在继承自类 Elem）和新的非虚拟运算符

```
ElemImp& operator=(const ElemImp& elemImp)
```

（为这个具体的实现类而显式地定义的）。

（7）从 ElemImp 中删除所有的冗余接口信息，例如在新的协议类 Elem 的接口中已经指定的枚举类型和 typedef 类型。

新的 ElemImp 类现在应该如图 6-46 所示。使用/*virtual*/表明关键字 virtual 是可选的。定义在原始类 Elem 中的非内联静态函数是其接口的一部分，可能已保留在基类中。但是，如果我们这样做，会面临下列令人不愉快的选择方案：

- 如果我们在基类 Elem 中定义这个函数来转发对派生类 ElemImp 的调用，则破坏了层次化的原则（见 4.7 节）。

- 如果我们在 elemimp 组件中实现了实际的 Elem 函数的定义，则违背了要求组件实现它们的输出功能的主要设计规则（见 3.2 节）。
- 如果我们直接在 elem.c 文件中实现函数，则会在物理上使我们的协议接口与一个特定的实现耦合，从而违反了在本节前面给出的协议的定义。

如果我们保留原始的接口则会更好。但是，上述选择方案没有一个是特别令人满意的。

```
// elemimp.h
#ifndef INCLUDED_ELEMIMP
#define INCLUDED_ELEMIMP

#ifndef INCLUDED_ELEM
#include "elem.h"
#endif

#ifndef INCLUDED_FOO
#include "foo.h"
#endif

#ifndef INCLUDED_BAR
#include "bar.h"
#endif

class ElemImp : public Elem {
  Foo d_fooPart;
  Bar d_barPart;

private:
  // ...
protected:
  // ...
public:
  ElemImp();
  ElemImp(const Foo& fooPart);
  ElemImp(const Bar& barPart);
  ElemImp(const Foo& fooPart, const Bar& barPart);
  ElemImp(const ElemImp& elemImp);
  /* virtual */ ~ElemImp();
  /* virtual */ Elem& operator=(const Elem& elem);
  ElemImp& operator=(const ElemImp& elemImp);
  static double f1() { /* ... */ }
  static void f2(double d);
  /* virtual */ Foo f3() const;
  /* virtual */ void f4(const Foo& foo);
  /* virtual */ const char *f5() const;
  /* virtual */ void f6(const char *name);
  /* virtual */ Status f7();
};

#endif
```

图 6-46 新的实现组件 elemimp

（8）为了保持层次化并确保完全的绝缘，从类 ElemImp 创建另一个组件 elemutil，它包含结构 ElemUtil。必须确保在 elemutil.c 中包含 elemimp.h。将所有定义在 Elem 中的静态成员函数移到 ElemImp 中。现在将以前定义在 Elem 中的所有的公共静态函数拷贝到 ElemUtil 中，并且重新实现它们（非内联），来转发客户程序对现在定义在类 ElemImp 中的相应函数的所有请求。

（9）由于 ElemImp 不是抽象的（即，由于它不包含任何纯虚函数），因此我们希望为客户程序提供一个绝缘机制，使它们不必实际包含非绝缘的类定义就能够实例化 ElemImp。（一个独立的组件也需要对从类 ElemImp 派生而来的每个对象的创建进行绝缘。）为每一个在 ElemImp 中定义的构造函数，在类 ElemUtil 中定义一个新的静态成员函数，取名为 createElem，采用与构造函数完全相同的参数基调并且返回一个指针，该指针指向动态分配的、由类 ElemImp 完全构造的实例，就像一个指向 non-const Elem 的指针一样。

新的绝缘的 ElemUtil 类现在应该如图 6-47 所示。

```
// elemutil.h
#ifndef INCLUDED_ELEMUTIL
#define INCLUDED_ELEMUTIL

class Elem;
class Foo;
class Bar;

struct ElemUtil {
    Elem *createElem();
    Elem *createElem(const Foo& fooPart);
    Elem *createElem(const Bar& barPart);
    Elem *createElem(const Foo& fooPart, const Bar& barPart);
    Elem *createElem(const Elem& elem);
    static double f1();
    static void f2(double d);
};

#endif
```

图 6-47　新的绝缘的工具组件 elemutil

修改后的系统如图 6-48 所示。新的 Elem 协议的公共客户程序现在摆脱了以前与 Elem 相关的编译时耦合。所有这种紧密耦合在元素子系统中都被完全隔离了。

通过为一个独立的工具组件提供创建函数来实例化每个派生类的方法，我们可以继续将所有的公共客户程序与在类 Elem 的层次结构中的所有复杂的实现细节绝缘。这种绝缘甚至可以应用到那些努力创建 Elem 的派生类新实例的客户程序中（例如 Client1）。但是，派生类仍然受基类 ElemImp 中的任何非绝缘改变的控制。注意，提供“额外”功能（即，那些通过 Elem 协议无法访问的功能）的派生类的客户程序不幸被迫在编译时依赖于派生类（因而也依赖于

elemimp)。

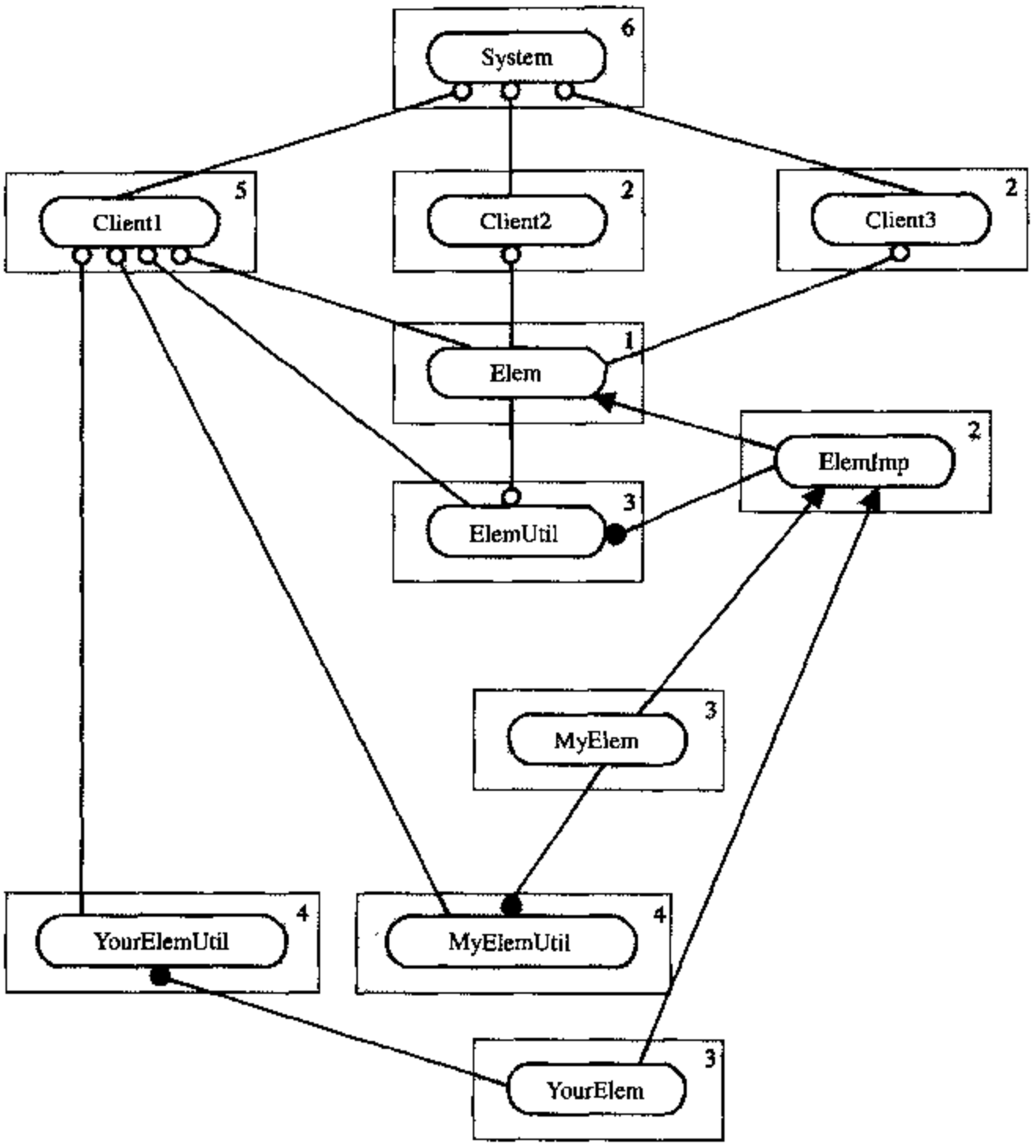


图 6-48 使用新的绝缘基类 Elem

6.4.2 完全绝缘的具体类

一个具体类不仅仅是一个接口——它定义了一个有用的对象，该对象可以实例化为程序堆栈上的一个自动变量。协议类（在 6.4.1 节中讨论的）与纯面向对象设计是一致的；但是，工程决不是纯的。有时我们会为一个协议类构造对象实例（就像其他具体类一样），这样做有时有好处。

考虑图 6-49 所示的类 Example。这个类包含用户自定义的类型 A、B 和 C，作为嵌入的数据成员。所有的成员函数隐式地声明为内联的，并且.c 文件实际为空。该类的实现很显然没

有与客户程序绝缘。假设我们现在认为该类将会被广泛使用并且其实现经常会变化。在这个例子组件中，为了客户程序能与实现细节的变化绝缘，我们能做什么呢？

```
// example.h
#ifndef INCLUDED_EXAMPLE
#define INCLUDED_EXAMPLE

#ifndef INCLUDED_A
#include "a.h"
#endif

#ifndef INCLUDED_B
#include "b.h"
#endif

#ifndef INCLUDED_C
#include "c.h"
#endif

class Example {
    A d_a;
    B d_b;
    C d_c;
    double value2() const { return d_a.value() + d_b.value(); }

  public:
    Example() {}
    Example(const Example& e) : d_a(e.d_a), d_b(e.d_b), d_c(e.d_c) {}
    ~Example() {}

    Example& operator=(const Example& e)
    {
        d_a = e.d_a;
        d_b = e.d_b;
        d_c = e.d_c;
        return *this;
    }

    double value() const
    {
        return value2() + d_c.value();
    }
};

#endif
```

```
// example.c
#include "example.h"
```

图 6-49　包含一个非绝缘的具体类的组件

第一步是用一个表面上不透明的指针来存储数据，以替换所有嵌入数据。通过消除嵌入

的实例，我们可以使我们的客户程序不必看到类 A、B 和 C 的定义。因此我们可以从 example.h 中消除显式的#include 指令，并且用类定义来替换它们。这样做经常要求把以前定义为内联的函数改为非内联的，这与我们对绝缘的希望是完全一致的。

图 6-50 显示了这种变换对于 example 组件来说看起来会怎样。如图所示，.h 文件更小了，而.c 文件不再是空的。Example 组件的客户程序现在与所有的对组件 a、b 和 c 的实现——甚至是接口——的修改绝缘。

但是，我们的客户程序并没有完全与 example 组件本身的实现变化绝缘。特别地，example 的客户程序没有与包含在 Example 类定义中的外表不透明的指针的实际数量绝缘。在类 Example 的私有数据中添加一个基本类型的实例也会迫使其所有的客户程序重新编译。修改任何私有成员函数的基调或返回类型也会产生同样的效果。

原　则

只保留一个不透明指针（指向包含一个类的所有私有成员的结构），会使一个具体的类能够将其客户程序与其实现绝缘。

我们如何完全绝缘类 Example 的实现，并使它仍然保持为一个具体的类？答案集中在除去单个的私有数据成员并且将它们替换为一个指向类表示的不透明指针上[①]。

定　义：一个具体类如果满足下列条件，就是完全绝缘的：

（1）只包含一个数据成员，它表面上是不透明的指针，指向一个定义具体类的实现的 non-const struct（定义在.c 文件中）；

（2）不包含任何其他种类的私有的或保护的成员；

（3）不继承任何其他类；

（4）不声明任何虚拟的或内联的函数。

图 6-51 显示了把一个没有将客户程序与其实现绝缘的类转换为一个完全绝缘的类的结果。所有的公共内联函数都被消除了。所有的私有成员数据和函数现在都成了一个辅助 struct 的一部分，该 struct 在组件的.c 文件中完全定义。注意，在这个例子中，辅助 struct 的默认的成员拷贝语义恰好是对的，因而没有被显式地实现。

原　则

所有完全绝缘的类的物理结构在外表上都是一样的。

① **murray**，3.3 节，72~74 页。

```
// example.h
#ifndef INCLUDED_EXAMPLE
#define INCLUDED_EXAMPLE

class A;
class B;
class C;

class Example {
    A *d_a_p;
    B *d_b_p;
    C *d_c_p;
    double value2() const;

  public:
    Example();
    Example(const Example& example);
    ~Example();

    Example& operator=(const Example&);

    double value() const;
};

#endif
```

```
// example.c
#include "example.h"
#include "a.h"
#include "b.h"
#include "c.h"

Example::Example()
: d_a_p(new A)
, d_b_p(new B)
, d_c_p(new C)
{}

Example::Example(const Example& example)
: d_a_p(new A(*example.d_a_p))
, d_b_p(new B(*example.d_b_p))
, d_c_p(new C(*example.d_c_p))
{}

Example::~Example()
{
    delete d_a_p;
    delete d_b_p;
    delete d_c_p;
}

Example& Example::operator=(const Example& e)
{
    if (&example != this) {
        delete d_a_p;
        delete d_b_p;
        delete d_c_p;
        d_a_p = new A(*e.d_a_p);
        d_b_p = new B(*e.d_b_p);
        d_c_p = new C(*e.d_c_p);
    };
    return *this;
}

double Example::value2() const
{
    return d_a_p->value() + d_b_p->value();
}

double Example::value() const
{
    return value2() + d_c_p->value();
}
```

图 6-50　包含一个部分绝缘的具体类的组件

```
// example.h
#ifndef INCLUDED_EXAMPLE
#define INCLUDED_EXAMPLE

class Example_i;      // fully insulated implementation
class Example {
    Example_i *d_this;

  public:
    Example();
    Example(const Example& example);
    ~Example();
    Example& operator=(const Example& example);
    double value() const;
}

#endif
```

```
// example.c
#include "example.h"
#include "a.h"
#include "b.h"
#include "c.h"

struct Example_i {
    A d_a;
    B d_b;
    C d_c;
    double value2() const { return d_a.value() + d_b.value(); }
}

Example::Example() : d_this(new Example_i) {}

Example::Example(const Example& example)
: d_this(new Example_i(*example.d_this)) {}

Example::~Example() { delete d_this; }

Example& Example::operator=(const Example& example)
{
    *d_this = *example.d_this;
    return *this;
}

double Example::value() const
{
    return d_this->value2() + d_this->d_c.value();
}
```

图 6-51 包含一个完全绝缘的具体类的组件

一个完全绝缘的类的重要特性是，改变其表示法不会影响客户程序认识一个实例的物理布局，因为它的实现（对象布局）永远只是一个单个的不透明指针。若忽略目的或功能，一个完全绝缘的类的实例看起来与其他完全绝缘的类的每个实例是一样的。正是这种物理上一

致性使我们不必以任何方式改变头文件就可以任意地重新实现类的接口。

允许继承或虚函数存在将影响对象的布局，因为会引入附加的数据或附加的虚函数表指针。注意，即使继承一个空的 struct 也会影响派生对象的大小。因此，从一个基类中继承的、且在其他方面完全绝缘的类的实例，物理上一定不同于一个没有继承基类的完全绝缘的类的实例。换句话说，从一个基类继承会增加一个完全绝缘的类的大小，其大小会超过单个指针的大小，这在物理上可以将其实例和其他完全绝缘的类的实例区分开来。

原　则

所有完全绝缘的实现都可以在不影响任何头文件的情况下进行修改。

完全绝缘的另一个重要特性是，类可以完全控制和访问定义内部表示的 struct。让内部数据成员直接指向定义在一个独立的组件中的类的实例，可能损害我们对自己的实现进行独立的、绝缘的修改的能力。为了在不影响客户程序的情况下添加一个私有成员，我们将被迫改变一个可独立访问和独立测试的对象的接口。

当为一个完全绝缘的具体类编写成员函数的实现时，不要用隐式的表示法：

```
d_c       的意思是 this->d_c
value2()的意思是 this->value2()
```

我们现在必须改为使用 d_this 指针显式地表示如下：

```
d_c       变为  d_this->d_c;
value2() 变为  d_this->value2();
```

数据结构类型的名称（例如，Example_i）和（特别是）实例变量的名称（例如，d_this）大多数都是风格的问题，并不需要在所有的情况下都一样。但是，因为 Example_i struct（“隐藏”在.c 文件中）可能包含函数或带外部连接的静态数据成员，所以有可能与在该组件的外部定义的、同名的类的成员发生意想不到的连接时冲突。正因为这个原因，用于定义完全绝缘的实现的 struct 的命名惯例，应该与用于普通类的命名惯例区分开来。附加公共可访问类名的前缀（后面紧跟一个下划线），可以确保一个组件的局部实现类不会与定义在这个组件的外部的类冲突。当开发一个大型项目时，我们会发现这种一致性惯例有助于识别完全绝缘的类表示法。

6.4.3　绝缘的包装器

在 5.10 节中介绍的包装器是作为一般的封装技术提出来的，它不仅适用于单个组件而且适用于整个子系统。在那一节引入包装器组件不是为了将那些对子系统的用户来说似乎是一

个实现细节的东西封装在每个组件内，而是去封装这些实现组件的使用。

因为一个子系统的客户程序没有被授权可对定义在较低层次上的实现组件中的对象进行编程访问，所以我们能够迫使这些客户程序只通过包装器接口与该子系统进行交互作用。

这里我们建议使包装器不仅可以进行封装，而且可以进行绝缘。因此，我们要努力消除不必要的混乱以及消除与包含不相关信息（甚至是他人专有的）的接口有关的编译时耦合。

6.4.3.1 单个组件包装器

一种产生绝缘包装器组件的方法是，把 6.4.2 节介绍的整体绝缘技术应用到一个封装的包装器中定义的单个对象上。我们可以在不影响任何用来实现该包装器的较低层次的对象的情况下做到这一点。

图 6-52 显示的是从图 5-95 得到的 graph 包装器组件的依赖关系。正如我们可以回忆起来的，graph 的客户程序不允许访问定义在实现组件中的对象：graphimp、gnode、gedge 和 ptrbag。但是，graph 的客户程序并没有与这些头文件的变化绝缘。

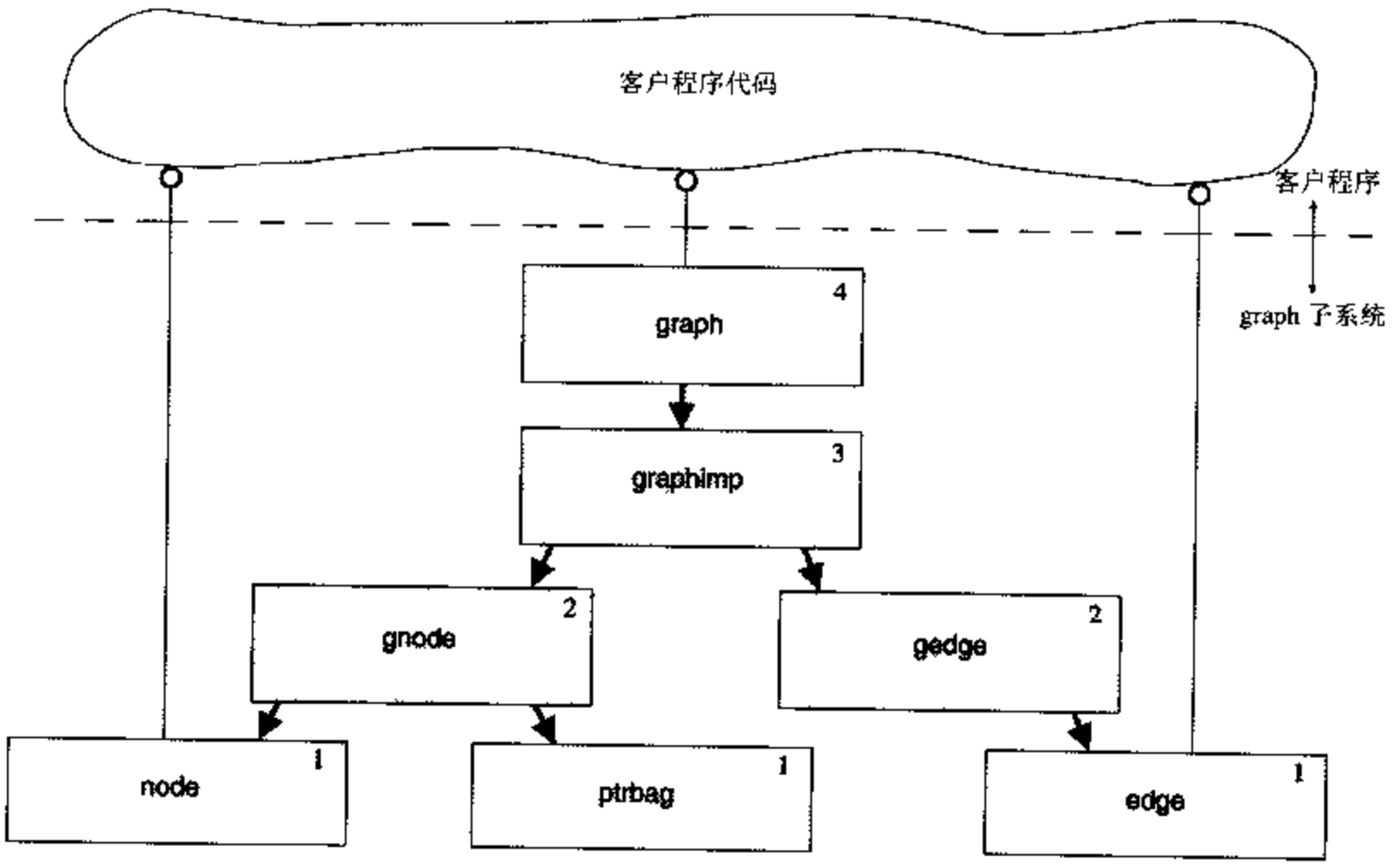


图 6-52 由图 5-95 得到的 Graph 包装器的组件依赖关系

让我们来考虑绝缘图 5-95 的 graph 包装器组件。使用 6.4.2 节介绍的整体绝缘技术对 graph 进行强制转换，会产生如图 6-53 所示的头文件。该接口确实取得了整体绝缘的效果，但是由

于需要额外的动态内存分配，因而运行时的开销太大了。

```
// graph.h
#ifndef INCLUDED_GRAPH
#define INCLUDED_GRAPH

class Node;                    // used in the interface of graph
class Edge;                    // used in the interface of graph

class NodeId_i;                // should be changed to:  class Gnode;
class EdgeId_i;                // should be changed to:  class Gedge;
class Graph_i;                 // fully insulated implementation
class NodeIter_i;              // fully insulated implementation
class EdgeIter_i;              // fully insulated implementation

class NodeId {
    NodeId_i *d_this;          // should be changed to:  Gnode *d_node_p;
    friend EdgeId;
    friend Graph;
    friend NodeIter;
    friend EdgeIter;

  public:
    NodeId();
    NodeId(const NodeId& nid);
    ~NodeId();
    NodeId& operator=(const NodeId& nid);
    operator Node *() const;
    Node *operator->() const;
};

class EdgeId {
    EdgeId_i *d_this;          // should be changed to:  Gedge *d_edge_p;
    friend Graph;
    friend EdgeIter;

  public:
    EdgeId();
    EdgeId(const EdgeId& eid);
    ~EdgeId();
    EdgeId& operator=(const EdgeId& eid);
    NodeId from() const;
    NodeId to() const;
    operator Edge *() const;
    Edge *operator->() const;
};

class Graph {
    Graph_i *d_this;
    friend NodeIter;
    friend EdgeIter;

  private:
    Graph(const Graph&);                    // not implemented
```

```
    Graph& operator=(const Graph&);       // not implemented

  public:
    Graph();
    ~Graph();
    NodeId addNode(const char *nodeName);
    NodeId findNode(const char *nodeName);
    void removeNode(const NodeId& nid);
    EdgeId addEdge(const NodeId& from, const NodeId& to, double weight);
    EdgeId findEdge(const NodeId& from, const NodeId& to);
    void removeEdge(const EdgeId& eid);
};

class NodeIter {
    NodeIter_i *d_this;

  private:
    NodeIter(const NodeIter&);
    NodeIter& operator=(const NodeIter&);

  public:
    NodeIter(const Graph& graph);
    ~NodeIter();
    void operator++();
    operator const void *() const;
    NodeId operator()() const;
};

class EdgeIter {
    EdgeIter_i *d_this;

  private:
    EdgeIter(const EdgeIter&);
    EdgeIter& operator=(const EdgeIter&);

  public:
    EdgeIter(const Graph& graph);
    EdgeIter(const NodeId& nid);
    ~EdgeIter();
    void operator++();
    operator const void *() const;
    EdgeId operator()() const;
};

#endif
```

图 6-53 完全绝缘的 graph 包装器组件的头文件 graph.h

如图 6-54 所示，无论何时通过值传递返回一个 NodeId，类 NodeId 的完全绝缘的版本现在都需要动态内存分配。

```
NodeId Graph::findNode(const char *nodeName)
{
```

```
    NodeId id;                              // causes dynamic allocation
    id.d_this->d_node_p = d_this->d_imp.findNode(nodeName);
    return id;
}
```

```
// (from graph.h)

class NodeId_i;  // fully insulated
class NodeId {
    NodeId_i *d_this;
    friend EdgeId;
    friend Graph;
    friend NodeIter;
    friend EdgeIter;

  public:
    NodeId();
    NodeId(const NodeId& nid);
    ~NodeId();
    NodeId& operator=(const NodeId&);
    operator Node *() const;
    Node *operator->() const;
};
```

```
// (from graph.c)

struct NodeId_i {
    Gnode *d_node_p;
};

NodeId::NodeId()
{
    d_this = new NodeId_i;
    d_this->d_node_p = 0;
}

NodeId::NodeId(const NodeId& nid)
{
    d_this = new NodeId_i;
    d_this->d_node_p = nid.d_this->d_node_p;
}

NodeId::~NodeId()
{
    delete d_this;
}

NodeId& NodeId::operator=(const NodeId& nid)
{
    d_this->d_node_p = nid.d_this->d_node_p;
    return *this;
}

NodeId::operator Node *() const
{
    return d_this->d_node_p;
}

Node *NodeId::operator->() const
{
    return *this;
}
```

图 6-54　图 5-95 中的 NodeId 的完全绝缘的重新实现

不再坚持对所有的包装器类进行完全的绝缘，而是部分地绝缘 NodeId 和 EdgeId 类，可以以相当低的运行时代价获得绝缘的大多数好处。通过在包装器头文件中只暴露这些实现类的名称，我们放弃了对包装器类添加独立成员的灵活性；但是，我们保留了以任何我们认为合适的方式对 Gnode 和 Gedge 的组织进行修改的权利。

图 6-55 说明了我们可以怎样为轻量级的类调整整体的绝缘以提高性能。通过值返回 NodeId 的函数现在部分地绝缘了，并且无需负担动态分配内存的代价——这种代价我们将在 6.6.1 节进行量化：

```
NodeId Graph::findNode(const char *nodeName)
{
    NodeId id;  // no dynamic allocation here
    id.d_node_p = d_this->d_imp.findNode(nodeName);
    return id;
}
```

```
// (from graph.h)

class Gnode;  // partially insulated

class NodeId {
    Gnode *d_node_p;
    friend EdgeId;
    friend Graph;
    friend NodeIter;
    friend EdgeIter;

  public:
    NodeId();
    NodeId(const NodeId& nid);
    ~NodeId();
    NodeId& operator=(const NodeId&);
    operator Node *() const;
    Node *operator->() const;
};
```

```
// (from graph.c)

NodeId::NodeId() : d_node_p(0) {}

NodeId::NodeId(const NodeId& nid)
: d_node_p(nid.d_node_p) {}

NodeId::~NodeId() {}

NodeId& NodeId::operator=(const NodeId& nid)
{
    d_node_p = nid.d_node_p;
    return *this;
}

NodeId::operator Node *() const
{
    return d_node_p;
}

Node *NodeId::operator->() const
{
    return *this;
}
```

图 6-55 图 5-95 中的 NodeId 的部分绝缘的重新实现

尽管由于部分绝缘 NodeId 和 EdgeId 运行时性能有了提高，但是对于其余三个类——

Graph、NodeIter 和 EdgeIter，则是一件完全不同的事情。在每一种情况下，把包装器的客户程序与实现对象绝缘，无论如何都需要动态内存分配。分配一个包含实现对象的 struct，就运行时开销而言不会比分配实现对象本身花费更多，也不会有与额外的间接相关的运行时开销。我们必须注意保证只有一个指针——添加一个理论上的 0 偏移量会被标准的编译时优化过程消除。在性能方面，完全绝缘这些类所花费的开销不会比部分绝缘这些类的开销大，因此我们也可以这么做。

注意，Graph、NodeIter 和 EdgeIter 都使拷贝构造函数和赋值函数失效了。因为正常使用这些对象要求建立和消除它们的频率比 NoteId 和 EdgeId 要低得多，它们自然是绝缘的更好的候选项。完全绝缘的 Graph、NodeIter 和 EdgeIter 的实现，以及对应于图 6-53 的头文件中所建议修改的 NodeId 和 EdgeId 的部分绝缘的实现如图 6-56 所示，仅供参考。

```
// graph.c
#include "graph.h"
#include "graphimp.h"
#include "gnode.h"
#include "gedge.h"

NodeId::NodeId() : d_node_p(0) {}

NodeId::NodeId(const NodeId& nid) : d_node_p(nid.d_node_p) {}

NodeId::~NodeId() {}

NodeId& NodeId::operator=(const NodeId& nid)
{
    d_node_p = nid.d_node_p;
    return *this;
}

NodeId::operator Node *() const { return d_node_p; }

Node *NodeId::operator->() const { return *this; }

EdgeId::EdgeId() : d_edge_p(0) {}

EdgeId::EdgeId(const EdgeId& eid) : d_edge_p(eid.d_edge_p) {}

EdgeId::~EdgeId() {}

EdgeId& EdgeId::operator=(const EdgeId& eid)
{
    d_edge_p = eid.d_edge_p;
    return *this;
}

NodeId EdgeId::from() const
{
    NodeId id;
```

```
    id.d_node_p = d_edge_p->from();
    return id;
}

NodeId EdgeId::to() const
{
    NodeId id;
    id.d_node_p = d_edge_p->to();
    return id;
}

EdgeId::operator Edge *() const { return d_edge_p; }

Edge *EdgeId::operator->() const { return *this; }


struct Graph_i {
    GraphImp d_imp;
};

Graph::Graph() : d_this(new Graph_i) {}

Graph::~Graph() { delete d_this; }

NodeId Graph::addNode(const char *nodeName)
{
    NodeId id;
    id.d_node_p = d_this->d_imp.addNode(nodeName);
    return id;
}

NodeId Graph::findNode(const char *nodeName)
{
    NodeId id;
    id.d_node_p = d_this->d_imp.findNode(nodeName);
    return id;
}

void Graph::removeNode(const NodeId& nid)
{
    d_this->d_imp.removeNode(nid.d_node_p);
}

EdgeId Graph::addEdge(const NodeId& from, const NodeId& to, double weight)
{
    EdgeId id;
    id.d_edge_p = d_this->d_imp.addEdge(from.d_node_p, to.d_node_p, weight);
    return id;
}

EdgeId Graph::findEdge(const NodeId& from, const NodeId& to)
{
    EdgeId id;
    id.d_edge_p = d_this->d_imp.findEdge(from.d_node_p, to.d_node_p);
```

```
    return id;
}

void Graph::removeEdge(const EdgeId& eid)
{
    d_this->d_imp.removeEdge(eid.d_edge_p);
}

struct NodeIter_i {
    GnodePtrBagIter d_iter;
    NodeIter_i(const GnodePtrBag& nodes) : d_iter(nodes) {}
};

NodeIter::NodeIter(const Graph& graph)
: d_this(new NodeIter_i(graph.d_this->d_imp.nodes())) {}

NodeIter::~NodeIter() { delete d_this; }

void NodeIter::operator++() { ++d_this->d_iter; }

NodeIter::operator const void *() const { return d_this->d_iter; }

NodeId NodeIter::operator()() const
{
    NodeId id;
    id.d_node_p = d_this->d_iter();
    return id;
}

struct EdgeIter_i {
    GedgePtrBagIter d_iter;
    EdgeIter_i(const GedgePtrBag& edges) : d_iter(edges) {}
};

EdgeIter::EdgeIter(const Graph& graph)
: d_this(new EdgeIter_i(graph.d_this->d_imp.edges())) {}

EdgeIter::EdgeIter(const NodeId& nid)
: d_this(new EdgeIter_i(nid.d_node_p->edges())) {}


EdgeIter::~EdgeIter() { delete d_this; }

void EdgeIter::operator++() { ++d_this->d_iter; }

EdgeIter::operator const void *() const { return d_this->d_iter; }

EdgeId EdgeIter::operator()() const
{
    EdgeId id;
    id.d_edge_p = d_this->d_iter();
    return id;
}
```

图 6-56　graph 几乎完全绝缘的重新实现（graph.c）

如果设计合理，单个的包装器组件可以有效地把客户程序与许多较低层次上的实现组件的组织细节绝缘。

6.4.3.2 多组件包装器

分别包装组件是可能的，但是只有在不需要客户程序与基础组件直接交互时才可以。作为一个有指导意义（但不大可能的）的例子，考虑为一个非绝缘的基于列表的 stack 组件建立完全绝缘的包装器组件 pubstack。

正如图 6-57 所示，原来的 stack 组件在它的头文件中暴露了三个类和两个运算符。其中一个类 StackLink 是其他两个类（Stack 和 StackIter）的一个封装的实现细节。包装器组件 pubstack 暴露了两个类、两个自由运算符，但没有暴露任何基础实现细节。不管 Stack 和 StackIter 是如何实现的，包装器类的客户程序与所有的实现细节都绝缘了。

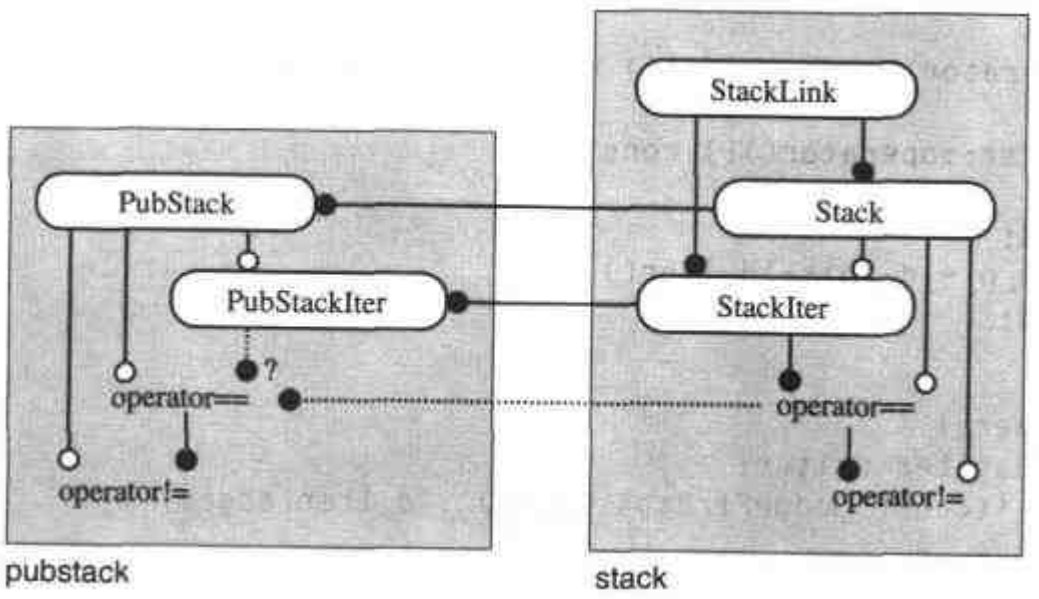


图 6-57 stack 及其包装器的完整的组件/类图

图 6-58 显示了 stack 组件的一个完全绝缘的包装器头文件。在这两个包装器类中，每个类都只有一个指向其内部定义的实现结构的私有不透明指针。在包装器的物理接口上，没有任何其他种类的私有或保护的成员。所有的函数都定义为非内联的。从其他（作为参数传递的）包装器对象中提取基础的被包装对象所必需的友元关系，是在这个包装器组件的物理接口中仅有的实现细节。

```
// pubstack.h
#ifndef INCLUDED_PUBSTACK
#define INCLUDED_PUBSTACK

class PubStackIter;

class PubStack_i;
class PubStack {
    PubStack_i *d_this;
```

```
    friend PubStackIter;
    // May want to grant access to improve performance and/or reuse:
    //friend int operator==(const PubStack&, const PubStack&);

  public:
    PubStack();
    PubStack(const PubStack& stack);
    ~PubStack();
    PubStack& operator=(const PubStack& stack);
    void push(int value);
    int pop();
    int top() const;
    int isEmpty() const;
};

int operator==(const PubStack& left, const PubStack& right);
int operator!=(const PubStack& left, const PubStack& right);

class PubStackIter_i;
class PubStackIter {
    PubStackIter_i *d_this;
    PubStackIter(const PubStackIter&);
    PubStackIter& operator=(const PubStackIter&);

  public:
    PubStackIter(const PubStack& stack);
    ~PubStackIter();
    void operator++();
    operator const void *() const;
    int operator()() const;
};

#endif
```

图 6-58　完全绝缘的 stack 包装器接口（pubstack.h）

图 6-59 显示了 pubstack 组件是如何实现的。事实上，由 PubStack 提供的所有功能就是非内联地将调用传递给绝缘的实现对象 Stack 的相应函数。PubStack 的每个构造函数只分配一个它的辅助结构 PubStack_i 的实例。PubStack 的析构函数会删除这个动态分配的实例，并且所有的成员函数只把它们的输入传递给 Stack 对象的相应的成员函数，该 Stack 对象嵌入在 PubStack_i 的受管实例中。

```
// pubstack.c
#include "pubstack.h"
#include "stack.h"

struct PubStack_i {
    Stack d_stack;
};

PubStack::PubStack()
```

```
: d_this(new PubStack_i) {}

PubStack::PubStack(const PubStack& s)
: d_this(new PubStack_i(*s.d_this)) {}

PubStack::~PubStack() { delete d_this; }

PubStack& PubStack::operator=(const PubStack& s)
{
    *d_this = *s.d_this;
    return *this;
}

void PubStack::push(int v) { d_this->d_stack.push(v); }

int PubStack::pop() { return d_this->d_stack.pop(); }

int PubStack::top() const { return d_this->d_stack.top(); }

int PubStack::isEmpty() const { return d_this->d_stack.isEmpty(); }

int operator==(const PubStack& left, const PubStack& right)
{
    PubStackIter lit(left);
    PubStackIter rit(right);
    for (; lit && rit; ++lit, ++rit) {
        if (lit() != rit()) {
            return 0;
        }
    }
    // at least one of lit and rit is now 0
    return lit == rit;
}

int operator!=(const PubStack& left, const PubStack& right)
{
    return !(left == right);
}

struct PubStackIter_i {
    StackIter d_stackIter;
    PubStackIter_i(Stack &stack) : d_stackIter(stack) {}
};
    PubStackIter::PubStackIter(const PubStack& stack)
    : d_this(new PubStackIter_i(stack.d_this->d_stack)) {}

    PubStackIter::~PubStackIter() { delete d_this; }

    PubStackIter::operator const void *() const
    {
        return d_this->d_stackIter.operator const void*();
    }
```

```
int PubStackIter::operator()() const
{
    return d_this->d_stackIter.operator()();
}
```

图 6-59 完全绝缘的 stack 包装器的实现（pubstack.c）

在这个例子中，为了实现其功能，自由的 operator==并不是绝对需要访问基础子对象的私有实现。operator==可以通过迭代器的公共版本局部地实现其功能，该迭代器确实对基础实现拥有私有访问权。如果认为这种开销过大，那么就很容易将包装器函数

```
int operator==(const PubStack& left, const PubStack& right)
```

声明为类 PupStack 的一个友元。这样做会授予该自由运算符对 PubStack 的基础 Stack 对象的私有访问权，使它能够直接调用相应的较低层次上的 operator==：

```
int operator==(const PubStack& left, const PubStack& right)
{
    return *left.d_imp_p == *right.d_imp_p;
}
```

因为预期到优化的可能，所以我们可以选择把在接口上使用 PubStack 的所有的类和自由运算符都声明为 PubStack 的友元，从而授予它们直接访问其基础表示对象 Stack 的权利。对于适合放在单个组件中的完整包装器层（例如，p2p_router、pubstack、graph），这种方法是十分可行的。对于单个的组件来说，任何隐含的友元关系都是局部的，因而既不会强加额外的耦合，也不会威胁到封装。

原　则

没有什么办法可以从一个组件的外部通过编程来确定一个组件是否为一个包装器。

因为一个包装器完全封装了它的基础实现，所以包装单个组件一般来说不大可行。如果我们试图用单个包装器组件去绝缘一个大型的子系统（用一种设法为基础实现建立镜像的方法），那么对于远距离友元关系的需求将很快变得很明显。

图 6-60 说明了包装那些必须直接交互作用的组件的问题。ElemSet 是一个管理 Elem 类型的对象集合的对象。ElemSet 有一个成员函数 void add(const Elem&)，该函数接受一个元素并将元素值的拷贝添加到集合中。PubElemSet 有一个相似的成员函数 void add(const PubElem&)，该函数接受一个 PubElem 并将其值的一个拷贝添加到集合中。你建议怎样实现 pubElemSet::add？惟一显而易见的实现

```
void pubElemSet::add(const PubElem& elem)
{
    d_this->d_elemSet.add(elem->d_this.d_elem);
}
```

会迫使较高层次上的 PubElemSet 成为 PubElem 的一个远距离友元。这是一个封装裂口（见 3.6 节）。

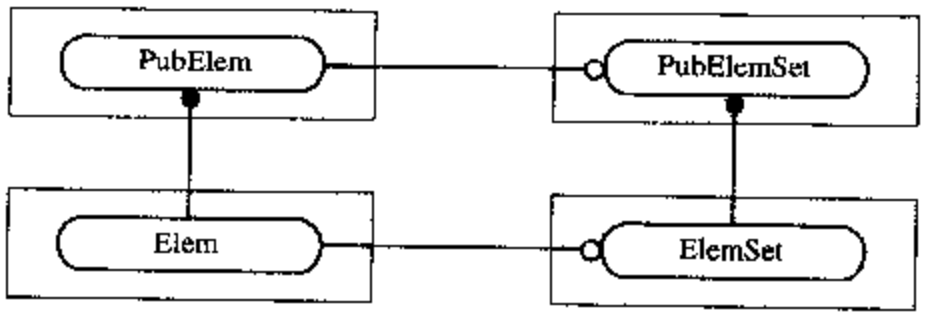


图 6-60 包装单个组件的困难

暂时忽略远距离友元关系的固有问题，所要求的友元关系的纯粹数量也很快会证明这个策略是不可行的。每个被用作为一个包装器类的成员函数（或自由运算符）的参数的包装器类型都必须将类或运算符声明为一个友元，以便使它能够访问正被传递的基础表示对象。如图 6-61 所示，两个包装器 PubA 和 PubB 现在正被用于 PubX.PubC 的公共接口中，它们以前不在 PubX 中使用，现在则在一个将要添加到 PubX 中的成员函数的基调中。正如图中所示，添加成员函数 void h(const PubC& c)到一个高层类 PubX 中，可能迫使添加一个友元声明到一个更低层次的类定义 PubC 中。这种修改反过来又会迫使那个类和它的所有客户程序重新编译！

原 则

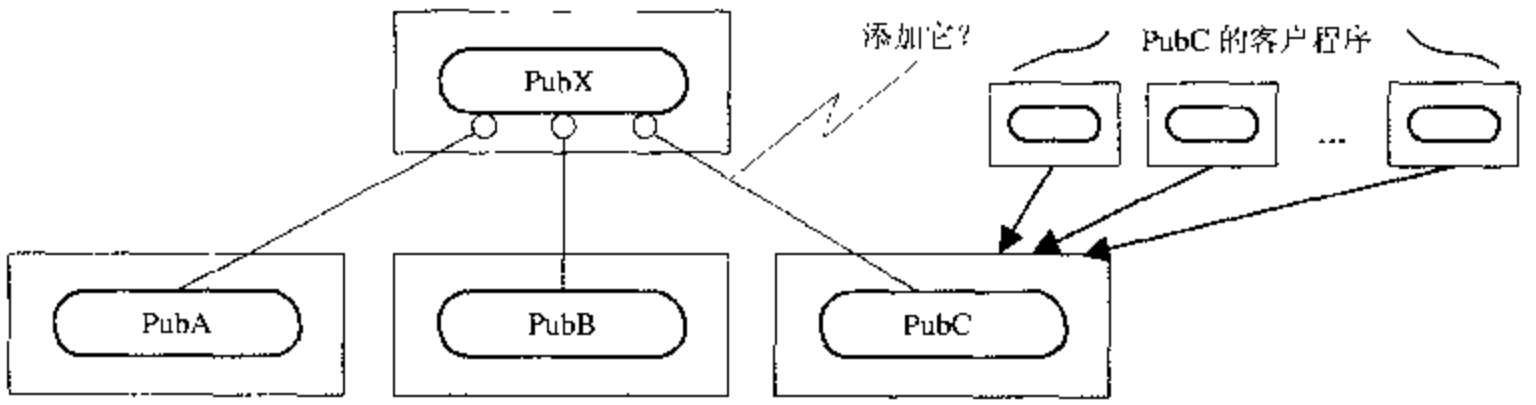
一个定义在 A 包装器组件中的类型，无论什么时候被传递进一个定义在 B 包装器组件中的类型中，B 组件都不能访问基础的被包装的实现对象，只能访问包装器的公共功能。

尽管如此，如果仔细设计的话，创建很有用的多组件绝缘包装器还是有可能的。创建这样一个包装器层的秘密是，要认识到在单个组件中只有类和运算符可以通过友元关系合法地利用在那个组件的接口之下的东西。

考虑图 6-62 所示的组件/类图。低层次上的子系统实现不仅被封装了——而且通过相对较少数量的包装器组件，也与子系统的其余客户程序绝缘了。在这个体系结构中，每个包装器组件都尊重每个其他组件（包装器或其他）的实现的私有内容，并限制对它们的公共接口的访问。做其他任何事情都会破坏我们在这个体系结构中所追求的封装。

定义在单个包装器组件中的包装器对象，在需要的时候可以自由地使用友元关系来看到局部的接口之下的基础表示，并直接操纵它们。例如，在图 6-22 中假设（正如 ElemSet 与 Elem 那样）类 E 在其接口上使用类 B，并且我们要把 E 和 B 的公共版本暴露给客户程序。类 *PubE* 将需要私有访问权以获得被封装在 PubB 中的 B 的实例。我们被迫将 PubE 声明为 PubB 的一

个友元，并有必要将 PubB 和 PubE 放在同一个包装器组件中以保护封装[1]。




```
class PubA;
class PubB;
//class PubC;
class PubX {
    X *d_imp_p;

  public:
    void f(const PubA& a);
    void g(const PubB& b);
    //void h(const PubC& c);
};
```

如果我们添加了上两行，则我们被迫加上下两行

```
class PubX;
class PubA {
    A *d_imp_p;
    friend PubX;

  public:
    // ...
};
```

```
class PubX;
class PubB {
    B *d_imp_p;
    friend PubX;

  public:
    // ...
};
```

```
//class PubX;
class PubC {
    C *d_imp_p;
    //friend PubX;

  public:
    // ...
};
```

图 6-61　由包装器之间的使用（uses）关系引起的双向耦合

避免远距离友元的设计目标使得包装器层的组件通常要比在基础的、低层次实现中的组件大得多，并且一般明显要比那些组件定义更多的对象。特别地，在图 6-62 中的包含 PubG 包装器的组件，就像 6.4.3 节中的 graph 包装器组件一样，提供了附加的迭代器类，以提供给客户程序访问低层次功能的绝缘通道。

这种类型的封装和绝缘了系统的体系结构非常强大。当拥有 5.10 节所讨论的封装包装器时，通过减少对基础功能的直接的公共访问，我们能够实施策略。由于在较高层次上也可进

---

① 该技术不应该被解释为避免非包装器类之间的远距离友元关系的万能药。因为包装器类在本质上是简单的，将几个包装器类合并到一个组件中不一定会威胁有效的测试。例如，合并实现组件（每个组件的复杂性都是便于管理的）可能会破坏设计单独组件的层次结构的目标。

行绝缘，所以没有必要在低层次上单独对每个组件进行绝缘。在这些低层次的组件中，我们可以自由地利用紧密的编译时耦合来提高性能。例如，内联函数常用于访问数值数据，而对象经常嵌入到其他对象中以避免动态分配的开销。简言之，在子系统的更高层次上，编译时耦合的影响被处理为“上行流”。

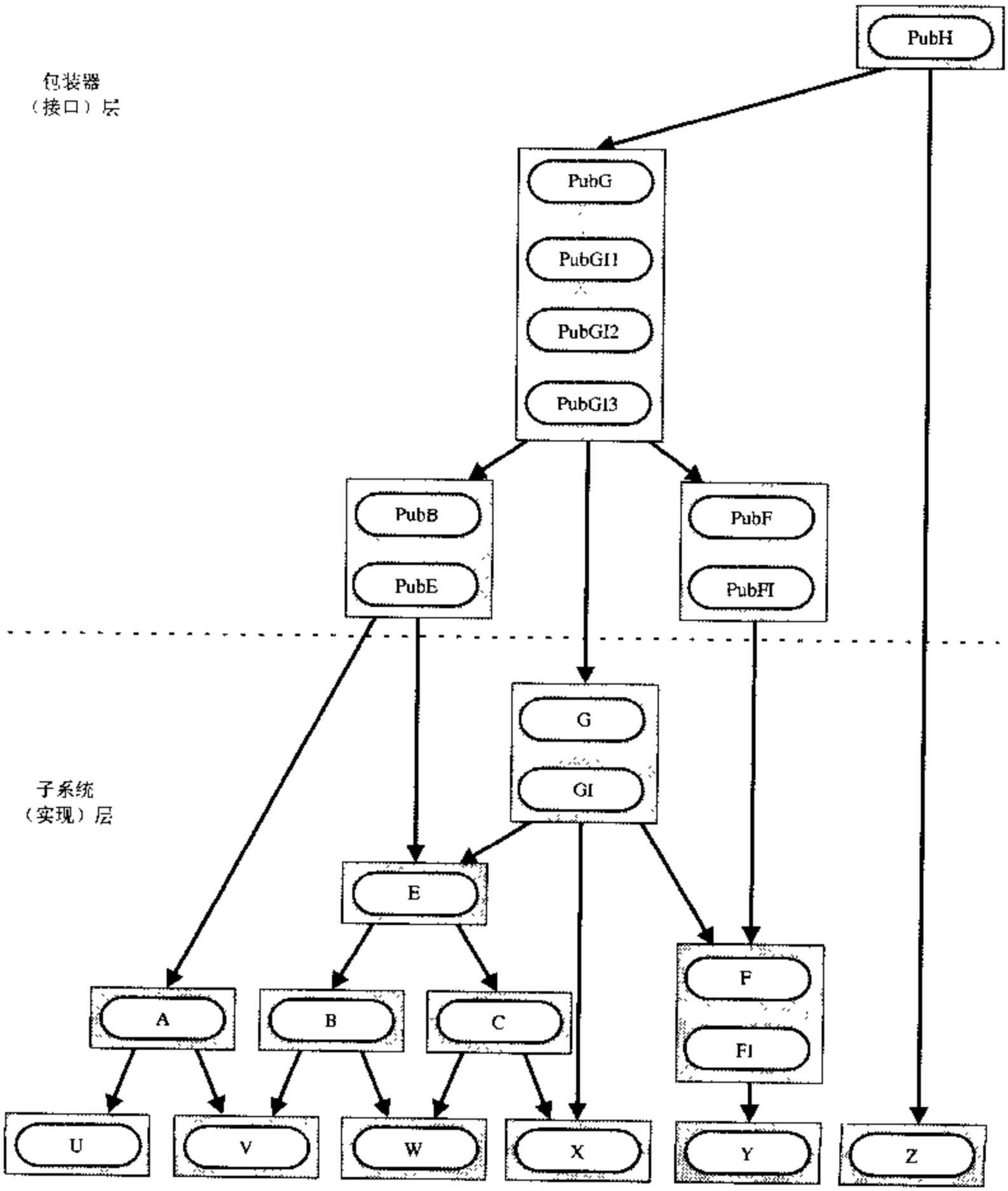


图 6-62 建立一个绝缘的多组件包装器

为了加强封装，包装器组件本身经常很大；但是，它们不必很复杂。一个包装器组件的一个重要功能是委派和调整复杂任务——而不是执行它们。在低层次组件中实现的复杂功能可以独立地进行测试和重用。测试包装器也就是检验被完全测试的基础组件是否已经正确地连接在一起。尽管当信息在包装器层和低层次子系统之间传递时会有很大的开销，但是选择在合适的层次上进行绝缘，可以确保对象之间的大部分通信出现在包装器层之下。如果小心设计，绝缘不一定会增加很多性能上的负担。

总的来说，一个绝缘的包装器也是封装的。包装器对象的逻辑接口上没有地方可使用定义在实现组件中的类型。在不破坏包装器的封装性的前提下，绝缘包装器的封装特性允许在别的子系统中重用一个被包装的子系统的基础实现组件。

如果包装器的绝缘是完全的，那么在包装器组件的物理接口上就无法命名定义在实现组件中的类型。一个部分绝缘的包装器可以拥有指向对象的指针，这些对象本身是定义在独立可访问组件中的“第一公民”。这些对象可以独立于包装器被重用，因而不容易进行修改。

相比之下，一个完全绝缘的包装器只保留一个指向简单 struct 的指针，该 struct 在其.c 文件中定义私有实现。因为没有独立的组件，所以就没有独立的方法与表示法直接地交互。和部分绝缘的包装器不同，完全绝缘的包装器不改变**任何**头文件就可能添加任意的私有数据。

在任一情况下，用于实现包装器的对象都可以通过它们自己的在子系统的较低层次上的封装（但通常是非绝缘的）接口自由地进行有效的交互。

因为在组件之间存在大量潜在的交互，所以包装一个子系统的每个组件经常是行不通的。但是，如果仔细规划，有可能为一个子系统建立一个多组件包装器。与所有其他组件一样，只有包装器组件的公共接口才能被其他组件访问。也就是说，只有定义在同一个包装器组件中的对象和运算符才能访问彼此的基础实现。

6.5 过程接口

我们发现大型商业面向对象系统（例如，数据库、框架等）经常有必要为顾客提供访问功能子集的编程接口，这些功能对于他们自己的核心系统开发者是可得到的。例如，一个数据库可以提供一个高层次的语言解释器（例如，SQL 或 Scheme），以便赋予顾客交互式地访问数据库中的信息的权利；另外，也经常会提供一个独立的接口，以允许最终用户用 C++（或标准 C）编写的程序直接操纵数据库。注意，“最终用户”指的是在我们公司或组织之外的想象中的客户。

对这样一个编程接口的需求一般包含以下几个方面：

- 接口必须提供必要的功能来操纵基础系统。
- 接口一定不能暴露私有的实现细节。
- 基础组织的变化必须与客户程序绝缘。
- 与该接口相关的开销一定不能过大。

如果接口是一个 C++接口，那么一个绝缘的包装器组件是理想的。但是，不是每个系统都可以包装。有些系统太大了，不能在单个组件中由一组包装器类合理地安置；也可能相互之间的联系太紧密了，不允许一个适当封装的实现使用一个多组件包装器。简言之，如果没有明确地设计成一个包装器，那么在不对体系结构作实质性修改的前提下，为一个现存的系统建立一个绝缘包装器层是不可行的。

如果我们的主要目标是把客户程序与在一个绝缘层（为一个非常大且复杂的系统所建立的）之下的所有的东西都绝缘的话，那么我们必须妥协。其中一种妥协是放弃一个包装器的真正的逻辑封装，而依赖带有未公布的头文件的外表不透明的指针来获得封装。这种类型的接口一般称为**过程接口**。

6.5.1 过程接口体系结构

我们提供的接口一般比那些开发者开始用于创建实现的接口要抽象得多。出于在第 4 章中讨论的同样的原因，通过只从过程接口测试来确保这样一个系统的可靠性非常困难。幸好，复杂性取决于系统的较低层次。我们作为过程接口的作者的工作是，确定在低层次上的实现组件中已经定义的类型和功能的一个合适的子集，这些实现组件允许最终用户完成他们所要的应用层任务。

图 6-63 是过程接口的组织方式的一个说明。所有公共可访问的接口函数都是彼此独立的，并且所有这些接口所处的层次比每个实现组件的层次都高。除了每个接口函数只依赖于基础实现之外，没有层次化的问题：过程接口函数不应该彼此依赖。

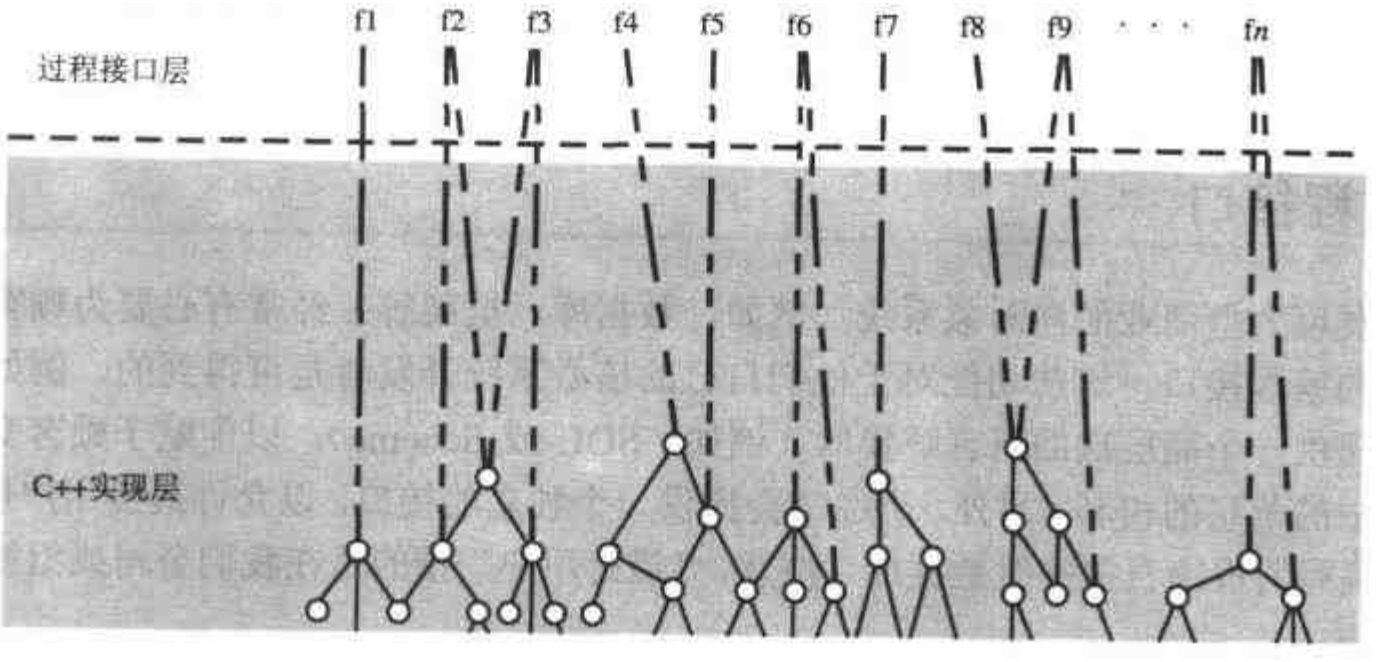


图 6-63 一个过程接口的示意性说明

如果我们正在实现一个封装/绝缘的包装器组件，那么我们要做的最后一件事情是在包装器接口中暴露实现类型。但是，最终用户对我们的实现组件的独立重用在这里很可能是不相关的。放弃顾客的独立重用会使我们牺牲包装提供的逻辑封装。

如果我们决定使用一个过程接口来进行绝缘，那么我们不会招致建立新包装器对象的开销，或者将我们自己限制在单个的组件中以避免远距离友元关系。我们可以在不公布定义的前提下，在过程接口中只暴露基础类型名称的一个适当的子集。

注意，这里的需求与 5.10 节中的不同。在那里我们的目标是封装组件的使用。而在这里，我们的目标是把客户程序与我们要它们使用的对象的定义绝缘。

使用与定义在基础实现中的相同的类型名称不会泄漏什么信息，却可以保护跨越接口的类型安全。该接口的最终用户也将受益于在他们自己的应用程序中的编译器强制类型安全。

除了减少附加类的开销和编译器强制类型安全之外，以名称来暴露基础类型可以在市场上获得很有吸引力的利益。一些顾客可能要利用系统的基础面向对象组织，而且愿意为这种特权付额外的费用。通过为这些顾客提供一些来自基础系统的关键（协议）基类头文件，有可能使他们能够派生他们自己的特殊类型用于系统内部，而不会暴露一个实现细节。

类似地，一些顾客可能需要比一个绝缘的过程接口所能提供的还要更好的性能。通过只公布最底层的具体对象（例如，Point、Box、Polygon）的头文件，愿意的顾客可以建立这些对象，将这些对象作为自动变量，并用内联函数直接访问它们。通过维护在过程接口上的类型名称的一致性，所有这种集成都可以是无缝的。注意，这对于封装的包装器是多么的不可能！

6.5.2 建立和消除不透明指针

为了讨论方便，我们假设要建立一个遵从 ANSI C 的接口。因此我们将被迫使用自由函数——这是对第二章讨论的一个主要设计原则的必要的违背。为了有助于在全局名称空间中避免名称冲突，每个自由函数的名称的开头都附加一个一致的注册前缀（正如 7.2 节所讨论的那样）。ANSI C 并不支持 C++引用，但支持 const 和 non-const 概念。所以所有的对象都将通过指针来传递，并且只有 non-const 对象才可以被修改或析构。

图 6-64 描述了建立和析构如图 3-2 中定义的一个 Stack 对象的过程接口。一旦建立，Stack 对象将一直存在，直到客户程序用类型安全函数 pi_destroyStack 显式地析构它。

```
/* CREATORS */
Stack *pi_createStack();
void pi_destroyStack(Stack *thisStack);
```

图 6-64 建立一个不透明的 Stack 对象的过程接口

ANSI C 并不支持函数名重载，这使得命名过程很成问题——特别是对于构造函数来说。由于用过程接口建立的对象不能是自动变量，所以它们的建立和删除比赋值的开销要大得多。因为这些原因，我们可以选择省略访问拷贝构造函数，而依赖默认的构造函数和重复使用赋值运算符来完成。

ANSI C 提供的类型安全在保护客户程序不受自我伤害时绕了个大弯。但是，因为这是一

个C接口而不是C++接口，所以也有很大的危险，它们可能意外地试图删除某个它们没有分配（以及不拥有的）的东西，或者试图删除某个是他们分配的但是分配了多次的东西。一个典型的公共内存分配错误的例子如图6-65所示。

```
void f()
{
    Stack *s1 = pi_createStack();
    Stack *s2 = pi_createStack();

    /* ... */

    pi_destroyStack(s1);
    pi_destroyStack(s1);              /* Oops! */
}
```

图6-65 在ANSI C中常见的内存崩溃

这种类型的顾客错误是最难排除的，并且这些错误对于客户服务的组织来说是一个非常昂贵的开销。幸好有一个探测大多数内存分配错误的有效方法。附录B中提供了一个在实际产品中探测和报告与内存分配相关的客户程序错误的内存分配器，该分配器已被证明是一种非常有效的方法。

6.5.3 Handles（句柄）

如果我们要建立一个过程接口供使用C++编程的顾客使用（因为反对ANSI C），那么可以采用比仅仅返回指针更好的处理动态分配对象的方法。这个流行的、管理由函数返回的动态分配对象的方法涉及一个通常称为handle的特殊的类。

> **定义**：在本书中，**handle**是一个类，它维持一个指向一个对象的指针，该对象可以通过handle类的公共接口编程访问。

本质上，一个handle是一个用于引用另一个对象的对象[①]。通常，一个handle除了拥有指向一个“被拥有”对象的指针之外，不包含其他信息，如图6-66所示。与包装器不一样，handle所指向的对象是可以通过handle的接口访问的。以这种方式使用的handle有时称为**smart pointer**[②]。在C++中，有许多handle模式的应用。图5-95中的NodeId包装器类扮演的是Gnode对象的Node部分的一个handle，它拥有一个指向该对象的指针。EdgeId也起同样的作用。

---

① **stroustrup**，13.9节，460页。

② **stroustrup**，7.9节，244页。

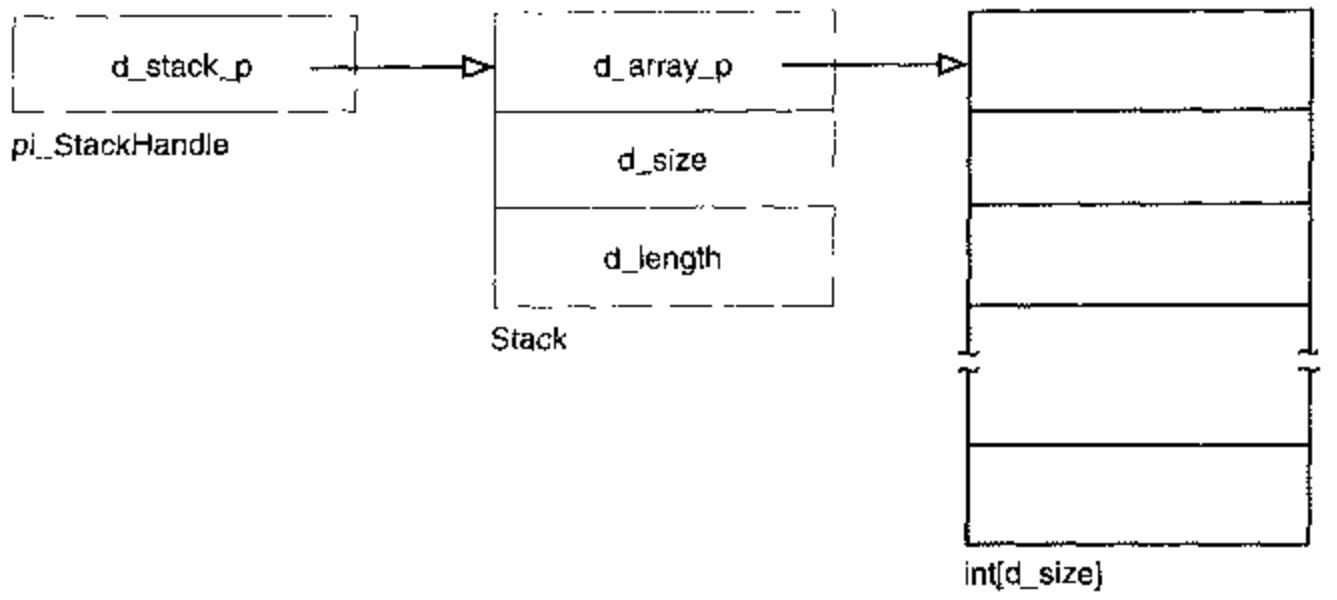


图 6-66 说明 pi_StackHandle 和 Stack 组织的对象图

在 C++中，一个常见的管理动态分配的对象的方法是，将其地址放入一个专门管理它的独立对象中。这样一个管理对象正好是一个特殊类型的 handle 对象——一个管理者 handle。一个 Stack 的管理者 handle 的头文件如图 6-67 所示。

```
// pi_stackhandle.h
#ifndef INCLUDED_PI_STACKHANDLE
#define INCLUDED_PI_STACKHANDLE

class Stack;

class pi_StackHandle {
    Stack *d_object_p;

  private:
    pi_StackHandle(const pi_StackHandle&);              // not implemented
    pi_StackHandle& operator=(const pi_StackHandle&);   // not implemented

  public:
    // CREATORS
    pi_StackHandle();
    ~StackHandle();

    // MANIPULATORS
    void loadObject(Stack *stack);    // Not intended for public use.

    // ACCESSORS
    operator Stack *() const;         // Conversion operator to allow use
};                                    // of this object as if this were
                                      // a writable pointer to a Stack.

#endif
```

图 6-67 类 Stack 的一个管理者 handle

我们建立一个包含 handle 的过程接口的整个方法如图 6-68 所示。这里，pi_Stack 就是一个只包含静态成员函数的 struct（定义一个名称空间）。通过外表不透明的指针，这些函数充当 C++过程接口来操纵 Stack 对象。

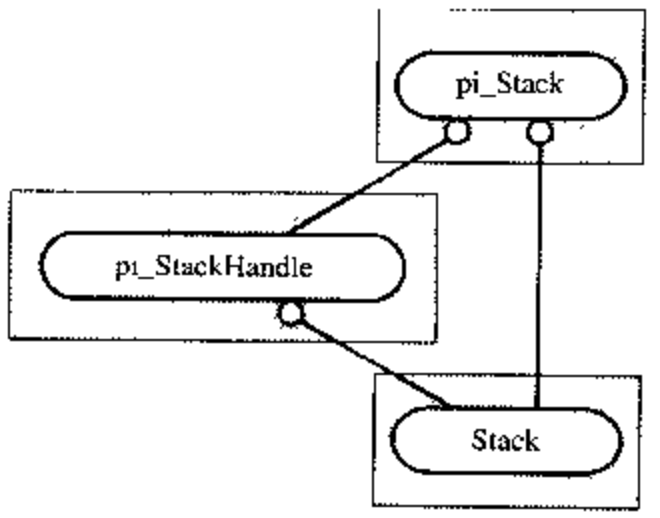


图 6-68 一个带有 handle 的过程接口的组件/类图

由于我们计划用 handle 来管理内存，所以将修改我们在 ANSI C 中使用的建立堆栈的函数 Stack *pi_createStack()。在一个基于 handle 的体系结构中，该函数的一个 C++等价体接受一个指向堆栈 handle 对象的可写指针作为参数。我们通过使该函数成为类 pi_Stack 的一个静态成员函数来避免（使用）ANSI C 版本的自由函数。整个 pi_Stack 组件的头文件如图 6-69 所示。

现在，不是返回一个直接指向一个动态分配的对象的指针，而是，一个 pi_StackHandle 对象首先由一个客户程序建立并作为一个自动变量，然后这个 handle 的地址被传入 create 函数中，在这个函数中，指针与一个动态分配的 Stack 对象一起装载，如图 6-70 所示。一旦装载了 handle，其转换运算符允许该 handle（见图 6-67）被当作是一个指向 Stack 的指针使用。

没有必要在 pi_Stack 接口上实现一个相应的析构函数。当 handle 超出作用域时，handle 的析构函数会反过来析构所包含的（动态分配的）Stack 对象，如图 6-71 所示。

由 C++类提供的作用域和 C++中提供的函数名重载功能简化了命名任务，添加句柄和作用域函数的名称的修饰并没有改变该接口的基本特性——它仍然是过程的。

特别地，努力使一个 handle 看起来像一个包装器是一个不好的建议。考虑一下，如果我们将 pop 函数实现为类 pi_StackHandle 的一个非静态成员函数（没有任何参数），以代替当前的接口，会发生什么呢？

```
class pi_StackHandle {
    // ...
  public:
    // ...
    int pop();
    // ...
};
```

明显的语义是，pop()成员函数应弹出并返回由该 pi_StackHandle 所管理的 Stack 对象的顶元素。但是，假设我们得到的是一个指向我们并不拥有的 non-const Stack 的指针，我们如何才能将它弹出呢？

```
// pi_stack.h
#ifndef INCLUDED_PI_STACK
#define INCLUDED_PI_STACK

class pi_StackHandle;
class Stack;

struct pi_Stack {
    // (Stack Creators)
    static void create(pi_StackHandle *handleToBeLoaded);

    // (Stack Manipulators)
    static Stack *assign(Stack *thisStack, const Stack *thatStack);
    static void push(Stack *thisStack, int value);
    static int pop(Stack *thisStack);

    // (Stack Accessors)
    static int top(const Stack *thisStack) const;
    static int isEmpty(const Stack *thisStack) const;
    static int isEqual(const Stack *left, const Stack *right) const;
};

#endif
```

```
// pi_stack.c
#include "pi_stack.h"
#include "pi_stackhandle.h"
#include "stack.h"

void pi_Stack::create(pi_StackHandle *h)
{
    h->loadObject(new Stack);
}

Stack *pi_Stack::assign(Stack *thisStack, const Stack* thatStack)
{
    *thisStack = *thatStack;
    return thisStack;
}

void pi_Stack::push(Stack *thisStack, int value)
{
    thisStack->push(value);
}

// ...
```

图 6-69　组件 pi_stack 的过程接口

```
void myFunc()
{
    pi_StackHandle h;                  // automatic variable
    pi_Stack::create(&h);              // load with dynamically allocated object
    for (int i = 0; i < 10; ++i) {
        pi_Stack::push(h, i);          // push 0, 1, ..., 9 on the managed stack
    }
    int x = pi_Stack::pop(h);          // pop 9 from managed stack into x
    // ...
}
```

图 6-70 一个管理者 Stack Handle 的使用模型

```
// pi_stackhandle.h
#ifndef INCLUDED_PI_STACKHANDLE
#define INCLUDED_PI_STACKHANDLE

class pi_StackHandle {
    Stack *d_stack_p;

  public:
    // ...
    ~pi_StackHandle();
    // ...
};

#endif
```

```
// pi_stackhandle.c
#include "pi_stackhandle.h"
#include "stack.h"

// ...

pi_StackHandle::~pi_StackHandle()
{
    delete d_stack_p;
}

// ...
```

图 6-71 管理者 handle 的析构函数析构它所“拥有”的对象

作为顾客，如果我们所能支配的是类 StackHandle 的成员函数 pop()，那么在我们能够操纵 Stack 对象之前，将被迫使用 loadObject()成员函数来将 Stack 对象指针放入一个 handle。但是，如果我们这样做，我们现在就有两个管理同一个 Stack 对象的内存的代理了！

过程接口中的 handle 的惟一用途是管理动态分配对象的内存。除了用一个新分配的 Stack 对象装载 pi_StackHandle 的 pi_Stack::create 函数之外，所有定义在 pi_Stack 过程接口中的功能，都应该直接引用基础 Stack 而不是 pi_StackHandle。通过遵循这个策略，顾客决不会被迫或受诱惑去滥用一个 handle 来获得访问基础对象功能的权利。

6.5.4 访问和操纵不透明对象

让我们回到对 ANSI C 的兼容性的假设。在一个过程接口中模拟“成员函数”的常见的分层次命名惯例如下：<prefix>_<Subject><Verb><Object>.

由于一致性问题，我们希望主题（subject）类型（例如，Stack）总是以首字母大写的形

式出现。如果忽略前缀，我们希望实际的函数名与 2.7 节所讨论的设计规则一致，即所有的函数名都以一个小写字母开头。为了从词法上把全局函数与全局类型区分开来，我们在实际的函数名前插入一个字母 f。例如：

```
int Stack::pop();                    ⟶ int pi_fStackPop(Stack *);
double Angle::getDegrees() const;    ⟶ double pi_fAngleGetDegrees(const Angle *);
void List::append(const Elem&);      ⟶ void pi_fListAppendElem(List *, const Elem *);
```

尽管这种命名风格完全适合于过程接口层，但是它没有必要直接翻译给实现层的基础对象和成员函数。例如，将转换函数（见图 5-15）

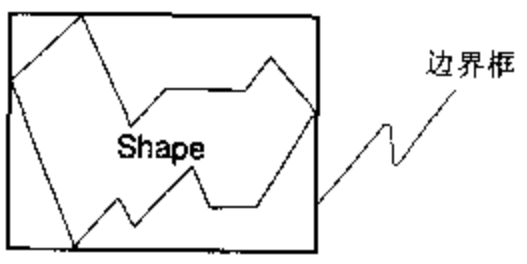
```
struct Convert {
    static Window toWindow (const Rectangle& r);
    static Rectangle toRectangle (const Window& w);
};
```

表示为：

```
void pi_fRectangleGetWindow(const Rectangle *thisRect, Window *returnValue)
{
    *returnValue = Convert::toWindow(thisRect);
}

void pi_fWindowGetRectangle(const Window *thisWind, Rectangle *returnValue)
{
    *returnValue = Convert::toRectangle(thisWind);
}
```

这对于过程的最终用户来说更直观，因而是一个合适的抽象。因为这些函数中的每一个都只依赖于物理层次结构的更低层次上的实现对象，所以没有层次化的问题。但是，如果实现为相应基础实现类的成员函数，那么这种方法将很快导致一个非层次化的结构。我们怀疑，这种试图将这种命名风格映射到 C++类和它们的成员函数上的幼稚的方法，是许多现存系统的循环物理依赖的形成原因。

我们现在将注意力转向图 6-72 所示的类 Shape。一个边界框是一个由水平和垂直边构成的最小的矩形（这些边限制点集合的范围）。其中的每个 Shape 都知道（通过值）如何返回一个包含 Shape 的 Box 类型的边界框。




```
class Box;

class Shape {
    // ...
  public:
    // ...
    virtual Box bBox() const;
    // ...
};
```

图 6-72　通过值返回一个用户自定义类型

由于指针是不透明的，所以在一个过程接口中对于用户自定义类型可能没有返回值。显而易见的选择是分配一个新的 Box 并返回一个指向它的指针，如图 6-73（a）所示。这种方法的一个问题是，通过值返回的对象一般是小对象，这些对象没有相关联的动态内存。每次访问动态分配的轻量级对象，如 Box 和 Point，都会产生相当大的不必要的开销。另一个问题是，返回不受管理的对象会引起内存混乱并增加泄漏的可能性。

```
Box *pi_fShapeGetBbox1(const Shape *thisShape)
{
    return new Box(thisShape->bBox());
}
```

（a）低效率，多危险

```
void pi_fShapeGetBbox2(const Shape *thisShape, Box *returnValue)
{
    *returnValue = thisShape >bBox();
}
```

（b）高效率，少危险

图 6-73　为通过值返回的对象提供一个过程接口

原　则

在一个过程接口中，使客户程序只显式地析构那些他们显式创建的对象，可以减少所有权关系上的混乱，并且可以提高性能。

通过坚持一个简单的原则，我们可以避免运行时的开销和所有权的混乱。这个简单原则是，只有被过程接口的客户程序显式分配的对象，才能由那个客户程序析构——所有其他的对象都由系统管理和拥有。图 6-73（b）中显示了一个更好的过程接口。

图 6-73（b）中的运行时效率的改进可能是显著的。图 6-74 显示了一个函数的两个实现，该函数返回一组图形的边界框的面积和。对于小的、轻量级的对象来说，例如 Point 和 Box，它们在一个函数中被反复多次获得，在循环的每次迭代中的动态分配[图 6-74（a）]和回收很容易占据函数调用的运行时开销的主要部分。相反，我们可以在循环之外做一次分配[图 6-74（b）]，然后反复多次重用分配的对象，这样可以显著提高运行时效率。

有时系统本身会动态地分配一个对象并将它返回给客户程序。在这种情况下，一个 handle 类通常由基础系统提供来管理那个对象的内存。例如，考虑一个 Shape 接口，该接口是所有 Shape 对象的一个协议类。现在假设有一个类 PointIter 也是各种顺序访问某个点集合的特定迭代器对象的一个协议。要求一个任意的 Shape 通过其协议来分配一个特定 shape 的迭代器（派生于 PointIter），并且通过装载一个用户提供的 PointIterHandle 的实例将它返回是有可能的，如图 6-75 所示[1]。

---

(1) 也可参见在 **gamma**（第 5 章，257~271 页）中的迭代器设计模式。

```
double sumArea1(const Shape *shape[],
                int size)
{
    double sum = 0;
    int i;
    for (i = 0; i < size; ++i) {
        Box *box = pi_fShapeGetBbox1(shape[i]);
        sum += pi_fBoxGetArea(box);
        pi_destroyBox(box);
    }
    return sum;
}
```

(a)

```
double sumArea2(const Shape *shape[],
                int size)
{
    double sum = 0;
    Box *box = pi_createBox();
    int i;
    for (i = 0; i < size; ++i) {
        pi_fShapeGetBbox2(shape[i], box);
        sum += pi_fBoxGetArea(box);
    }
    pi_destroyBox(box);
    return sum;
}
```

(b)

图 6-74 比较两种过程接口模型的使用/效率

```
class Point;

class PointIter {
  public:
    // CREATORS
    virtual ~PointIter();

    // MANIPULATORS
    virtual void reset() = 0;
    virtual void operator++() = 0;

    // ACCESSORS
    virtual operator const void *() const = 0;
    virtual const Point operator()() const = 0;
};

class PointIterHandle {
    PointIter *d_iter_p;
    PointIterHandle& operator=(PointIterHandle&);
    PointIterHandle(PointIterHandle&);

  public:
    // CREATORS
    PointIterHandle();
    PointIterHandle(PointIter *iterator);
    ~PointIterHandle();

    // MANIPULATORS
    void loadIter(PointIter *newDynamiclyAllocatedIterator);

    // ACCESSORS
    PointIter& operator()() const;
    operator PointIter&() const;
    PointIter *operator->() const;
    PointIter& operator*() const;
};
```

```
class Shape {
    // ...
  public:
    // ...
    // ACCESSORS
    virtual void getVertices(PointIterHandle *returnValue) = 0;
    // ...
};
```

图 6-75 在 C++中使用 Handle 来管理动态内存

在这个例子中，系统正动态地分配一个迭代器对象，并且将它放在一个用户提供的 handle 中。由于基础系统本身正在分配内存，所以客户程序没有被授权可以删除它。但是，客户程序被授权创建和删除 PointIterHandle 的一个实例。因此顾客创建一个 PointIterHandle 并将它传递给 Shape 的 getVertices 函数。然后系统用一个动态分配的指向 PointIter 的指针装载 handle。顾客使用 handle 中包含的、并且由 handle 管理的对象。当客户程序删除 handle 时，handle 的析构函数反过来会删除所包含的动态分配的迭代器。重用一个 handle 以获得另一个迭代器也会促使 handle 在装入新的迭代器之前，删除所有以前安装的迭代器。一个遵从 ANSI C 的图 6-75 所示功能的过程接口如图 6-76 所示。这样一个接口的使用如图 6-77 所示。

```
typedef struct Point Point;                          // ANSI C compatibility
typedef struct PointIter PointIter;                  // ANSI C compatibility
typedef struct PointIterHandle PointIterHandle;      // ANSI C compatibility
typedef struct Shape Shape;                          // ANSI C compatibility


                              /***** PointIter *****/
/* MANIPULATORS */
void pi_fPointIterReset(PointIter *thisIter);
void pi_fPointIterAdvance(PointIter *thisIter);

/* ACCESSORS */
int pi_fPointIterIsValid(const PointIter *thisIter);
void pi_fPointIterGetItem(const PointIter *thisIter, Point *returnValue);


                              /***** PointIterHandle *****/
/* CREATORS */
PointIterHandle *pi_createPointIterHandle();
void pi_destroyPointIterHandle(PointIterHandle *thisHandle);

/* MANIPULATORS */
/* void pi_fPointIterHandleLoadIter(PointIterHandle *thisHandle,
 *                                  PointIter *newDynamicPointIter);
 * Note: not necessary to expose this dangerous function
 */

/* ACCESSORS */
PointIter *pi_fPointIterHandleGetIter(const PointIterHandle *thisHandle);
```

```
/* Note: for a procedural interface, this one accessor is sufficient */


                                /***** Shape *****/
/* ACCESSORS */
void pi_fShapeGetVertices(Shape *thisShape, PointIterHandle *returnValue);
```

图 6-76　一个包含 Handle 的且遵从 ANSI C 的过程接口

```
void f(Shape *shape)
{
    PointIterHandle *handle = pi_createPointIterHandle();
    Point *pt = pi_createPoint();
    PointIter *it;

    pi_fShapeGetVertices(shape, handle);
    it = pi_fPointIterHandleGetIter(handle);

    for (; pi_fPointIterIsValid(it); pi_fPointIterAdvance(it)) {
        pi_fPointIterGetItem(it, pt);
        /* do stuff with current point */
    }

    pi_destroyPoint(pt);
    pi_destroyPointIterHandle(handle);
}
```

图 6-77　使用一个包含 Handle 的且遵从 ANSI C 的过程接口

作为一个为大系统编写过程接口的作者，你可能会发现一个直接返回一个动态分配对象并且没有将它放到一个客户提供的 handle 中的类接口。在这样的情况下，我们有必要寻找（或创建）一个合适类型的 handle，并且要求我们的客户程序传递一个指向那个 handle 的非 const 指针到我们的接口函数中。然后我们必须自己装入一个带有系统分配对象的 handle。这样做可以保持过程接口的原则，即客户程序只被授权删除它们显式分配的对象。

6.5.5　继承与不透明对象

因继承而相关的类型之间的转换，也是为必须进行的面向对象设计而编写过程接口的另一个方面。目前的问题涉及我们如何提出一个类型安全的、支持指针转换概念的过程接口，这种转换是继承所隐含的。

考虑图 6-78 所示的类图。类 B 从 A1 和 A2 公共派生而来，这意味着 A1 和 A2 的所有功能通过 B 的公共接口都是可访问的。但是，绝缘甚至使一个过程接口的 C++顾客都无

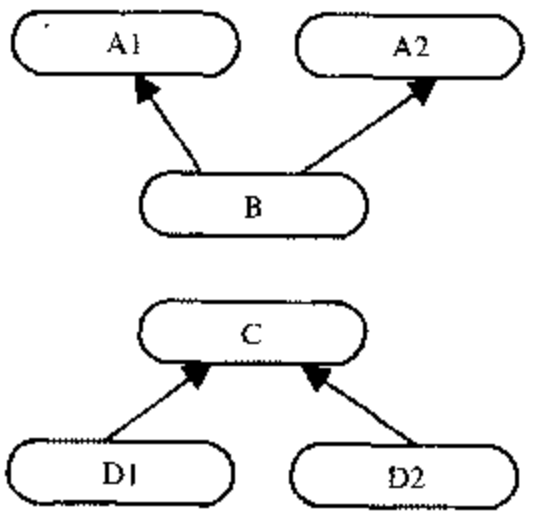


图 6-78　继承关系的例子

法知道类型 A1、A2 以及 B 是如何关联的。例如，如果我们有一个指针指向 B 类型的对象并且我们要调用一个定义在 A1 中的成员函数，那么我们的运气就不会太好；这显然是不可接受的。

我们首先想到的可能是在 B 中复制定义在 A1 和 A2 中的功能。这样做会产生大量的冗余函数，并且只解决了问题的一半。假设我们要在一个函数中使用一个 B 类型的对象，该函数使用了一个 A1 类型的对象。那么我们也应该为每个派生类型的组合复制每个函数吗？作者认为不必这么做。

当一个给定类型公共继承（直接或间接）另一种类型时，C++语言支持从第一种类型的指针到第二种类型的指针的隐含的（标准的）转换；有必要在过程接口中使这种转换显式地进行。

对应于图 6-78 的显式转换函数如图 6-79 所示。在这个例子中，4 个继承关系引导出了 8 个函数。注意有两种函数：一种是为 const 对象服务的，另一种是为非 const 对象服务的。尽管这似乎已经很棘手了，但它还会变得更糟糕。

```
A1 *pi_convertBA1(B*);
A2 *pi_convertBA2(B*);
C *pi_convertD1C(D1*);
C *pi_convertD2C(DC*);
const A1 *pi_convertConstBA1(const B*);
const A2 *pi_convertConstBA2(const B*);
const C *pi_convertConstD1C(const D1*);
const C *pi_convertConstD2C(const D2*);
```

图 6-79 （标准）转换函数的例子

现在考虑如果我们多引入一个从 C 到 B 的继承关系会发生什么，如图 6-80 所示。除了两个明显的额外转换程序

```
B *pi_convertCB(C*);
const B *pi_convertConstCB(const C*);
```

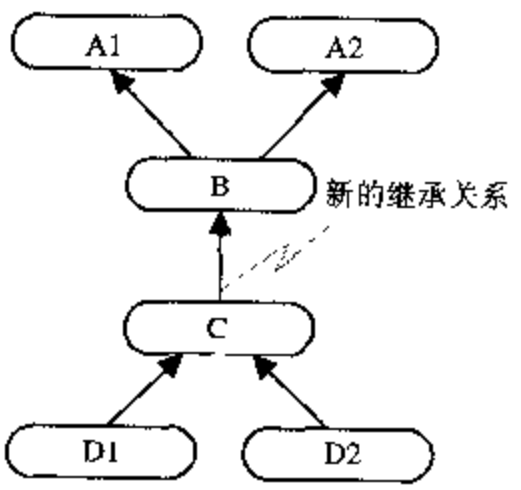


图 6-80 在继承层次结构中的转换传递

此外，IsA 关系的传递性也潜在地引入了下列 16 个转换：

```
B *pi_convertD1B(D1*);
A1 *pi_convertD1A1(D1*);
A2 *pi_convertD1A2(D1*);
B *pi_convertD2B(D2*);
A1 *pi_convertD2A1(D2*);
A2 *pi_convertD2A2(D2*);
A1 *pi_convertCA1(C*);
A2 *pi_convertCA2(C*);
const B *pi_convertConstD1B(const D1*);
const A1 *pi_convertConstD1A1(const D1*);
const A2 *pi_convertConstD1A2(const D1*);
const B *pi_convertConstD2B(const D2*);
const A1 *pi_convertConstD2A1(const D2*);
const A2 *pi_convertConstD2A2(const D2*);
const A1 *pi_convertConstCA1(const C*);
const A2 *pi_convertConstCA2(const C*);
```

尽管不是绝对需要有一个函数来提供从类型 D1 到类型 A2 的直接转换，但是过程接口的用户可能已经为不得不使用一个转换函数而感到生气，更不用说使用三个了。很明显，在需要提供的转换函数定义的数量和在运行时需要调用的转换函数的数量之间明显有一种折衷。

维护并归档所有手边这些函数不仅费用昂贵而且易于出错。幸好这些转换函数是微小并且规则的，而且使用类似于那些在附录 C 中用于确定层次号的技术可以很容易地准确产生这些转换函数。注意，不要试图去将所有这些转换函数归档，而是给用户显示如何基于以下两种类型的名称来推断出合适的名称，这样要好管理得多：

```
<Type2> *pi_convert<Type1><Type2>(<Type1>*);

const <Type2> *pi_convertConst<Type1><Type2>(const <Type1>*);
```

小结：过程接口完全不同于本章提出的其他绝缘技术，它满足一组完全不同的目标。像析取一个协议类和创建一个包装器这样的技术，既服务于封装也有助于将客户程序与子系统的低层组织绝缘。这些其他技术使我们能够并排（side-by-side）重用低层实现组件，而完全不可能破坏内部使用这些组件的子系统的封装性。

通过比较，一个过程接口的主要用途是将在一定层次之下的所有组织方面都与客户程序绝缘。通常不允许顾客重用基础对象——经常是因为所有权的原因。这种绝缘技术的主要优点是不需要进行预先的设计工作（这种设计工作伴随着要提供一个封装接口），也不强加额外对象的开销来获得封装。远距离友元关系的问题也消除了，因为基础类型在名称上是公共可用的，但是那些能够直接访问功能（为这些基础类型所定义的）的头文件通常是顾客不能访问的。

提供一个过程接口有一个明显的不足，从纯形式上看，客户程序失去了通过使用继承扩展系统功能的能力。但是，只要仔细设计，有可能提供一个过程接口并用一些选好的头文件来增大接口，以缓解这个问题。过程接口要求使用长而乏味的函数名称来完成通常由类作用域、运算符和标准转换内的成员函数来完成的工作。当接口要遵从 ANSI C 时，函数名称变得更加冗长乏味。

一个过程接口既不是面向对象的也不是格外优秀的，但它有一个很大的优点：过程接口总是能够用于使客户程序与一个大系统的组织绝缘——即使在设计的早期阶段并没有考虑这样的接口。

6.6 绝缘或不绝缘

组件的客户程序由于改变头文件而不断地重新编译，这是我们要极力避免的。在开发过程中，这样的“自发”重新编译既使人生气又代价昂贵。

如果我们改变了组件的逻辑接口，我们就很难使这些改变与客户程序绝缘——事实上我们强调在设计过程的早期就要使主要接口正确。成批地处理这些变化，并以软件版本的形式偶尔地公布这些变化（见 7.6 节），可以减少但不能消除这种开销。

正如我们在本章的前面几节中所讨论的的那样，我们可以采取一些方法来减少甚至消除由于组件的逻辑实现的变化而引起的客户程序的重新编译的开销。但是，绝缘本身不是没有开销。有时为一个组件创建一个绝缘的接口要花费更多的努力，并且在有些情况下绝缘会显著降低运行时性能。

在下面的小节中，我们将讨论绝缘的开销问题，讨论什么时候绝缘是（或不是）合适的，以及哪种绝缘技术适合于实践中常出现的一些特别情况。

6.6.1 绝缘的开销

绝缘一个类很显然会影响其运行时的性能。影响的程度取决于类本身、使用绝缘的方式以及使用绝缘的技术。

原 则

尽管计算机体系结构和编译器各不相同，但是下列的规则可以帮助系统结构设计师决定：在设计的早期，是否绝缘以及如何绝缘。

访问	单独访问的相对开销
借助内联函数通过值	1
借助内联函数通过指针	2
借助非内联、非虚拟函数	10
借助虚拟函数机制	20
创建	**单独分配的相对开销**
自动化	1.5
动态	100+

图 6-81 为不同形式的函数调用和对象实例化提供了一些数字。如图所示，直接访问数据和通过一个内联函数访问的开销在统计上是一样的。在一台 SUN SPARC-2 工作站上使用 CFRONT 3.0 C++编译器（不带优化），访问一个整型数据成员（直接访问或者借助一个内联函数），并且把它赋给另一个整型变量（如图 6-81a、c）花了大约 1/8 微秒。注意，如果访问必须通过一个指针的话，那么在一台 SPARC-2 上通过指针完成这个操作会比直接访问或借助内联函数多花费 60%，而在一台 SPARC-20 上通过指针完成这个操作会比直接访问或借助内联函数多出一倍的开销（b、d）[①]。

```
struct A {
    int d_d;
    inline int i() const;
    int f() const;
    virtual int v() const;
};
int A::i() const { return d_d; }
int A::f() const { return d_d; }
int A::v() const { return d_d; }

main ()
{
    A a, *p = &a;
    int j;
                                    //      TIME IN MICROSECONDS
                                    //      SPARC-2      SPARC-20

    j = a.d_d;                      // a.    0.124        0.040
    j = p->d_d;                     // b.    0.200        0.080
    j = a.i();                      // c.    0.125        0.040
    j = p->i();                     // d.    0.199        0.080
```

① 在更快的 SPARC-20 上，这个操作为存储访问时间所限制。

```
    j = a.f();                          // e.   0.575        0.301
    j = p->f();                         // f.   0.599        0.301
    j = p->v();                         // g.   1.076        0.543

    { A a }                             // h.   0.175        0.060
    { A *p = new A; delete p; }         // i.  11.757        5.478
}
```

图 6-81 访问和创建的一些相对开销

对于一个完全绝缘的类来说，可能没有内联函数，所以一个私有成员的每个访问都要求间接地通过一个指针。访问一个带有常规函数而不是内联函数的私有成员的开销大约增长了4倍（e）。注意一点，由于间接访问而增加的操作开销不超过5%（f）。这就是说，不将一个成员函数定义为内联所增加的访问开销远大于间接访问造成的附加开销。

对于一个协议类，可能没有非虚拟函数，并且强制进行对指针的间接引用。所有的函数调用必须通过虚拟函数调用机制。执行带有一个动态边界函数而不是静态边界函数的相同操作再次成倍增加了操作开销（g）[1]。虽然虚拟函数调用机制比直接调用机制要慢一些，但是对于小的访问函数来说，虚拟函数调用机制比起直接访问数据就慢得太多了，这是因为使用了一个内联函数。然而在通常情况下，如果可以使函数成为非内联的话，那也可以使函数成为虚函数。注意，随着函数的增长，与执行函数体相关的运行时开销将大大超过调用机制所使用的开销；跨越动态边界函数调用的内联对速度性能的改善将变得微不足道。

一个具体类的一个显著特性是它可以被实例化。当一个有完全绝缘的实现的对象被创建为一个自动变量（在程序堆栈上）时，其实现结构必须被单独分配。正如图6-81的底部所示，（在堆上）动态分配一个 struct 的开销比自动分配慢两个数量级（h、i）。在特定对象内存管理的工作量很大，付出这种开销是必需的。并且基于类的管理技术可能会导致不受欢迎的副作用（见10.3.4.2节）。更糟糕的是，动态分配的开销一般依赖于应用程序的大小，特别是依赖于特定的运行时动态内存管理系统中当前内存的使用情况/或内存碎片的情况。由于这些原因，对于极小的、轻量级的具体类的完全绝缘可能是不妥当的，特别是对于在程序堆栈上经常创建的对象或者从函数通过值返回的对象。

6.6.2 什么时候不进行绝缘

将客户程序与一个组件本身的封装的实现细节所作的变化绝缘是好的。但是，不是所有的组件接口都应该绝缘。

---

(1) 如果不影响虚拟函数的运行时性能，那么重新迭代一个继承层次结构的深度是值得的。每个类维护她自己的虚拟表，所以分派任意虚拟函数的开销独立于类层次中的继承数量。

原　则

对于一个组件的实现不进行绝缘的决策，可能是基于该组件不是很广泛使用的认识。

有些组件的确是不准备通用的。如果一个组件的用户是有限的，绝缘就不再是关键的了。在那种情况下，对非绝缘实现的改变的影响可能不会造成很大的威胁。事实上，对于一个定义了一些接口（包装器）组件的子系统来说（且接口或包装器组件本身将整个多组件子系统与一般用户绝缘），某个组件可能是专用的。第 4 章的 p2p_router 以及 6.4.3.1 节中的 graph 就是这样的例子。将组成这样一个子系统的每个组件进行绝缘而努力的结果可能是南辕北辙。

原　则

除非已经知道性能不是一个问题，否则避免用极小的访问函数（它们在整个系统中广泛使用）对低层次类的实现进行绝缘可能是明智的。

有两个不同的方法可用来减少由于实现的变化而引起的重新编译的频率：

（1）更好地对实现进行绝缘。

（2）减少对实现的改变。

定义：**轻量级（Light-weight）**是一个术语，其含义依赖于它被使用的上下文：

- 不依赖于（许多）其他的组件；
- 创建/析构都不昂贵；
- 不分配额外的动态内存；
- 能有效地利用内联函数来访问/操纵嵌入数据。

对于广泛使用的组件来说，尤其不希望由于封装细节的变化而引起客户程序重新编译。但是，正如我们所知的，绝缘也不是没有性能代价。小的、轻量级的类如 Point、Stack、List 以及其他在计算机科学中定义良好的具体的数据结构都不太适合绝缘，即使它们在整个系统中被广泛地使用。对这些类进行绝缘会增加一个全面的性能负担，这种负担在许多情况下很难证明是正当的。事实上，对于那些经常重复调用的类（例如带有极小访问函数的那些类）来说，使用间接的和非内联的函数可能会使它们慢一个数量级，正如 6.6.1 节所显示的那样。

当在一个给定的函数调用中所做工作的运行时开销相对于调用本身的开销大时，绝缘将不会引起严重的性能问题。因此，若不考虑任何假设的性能需求，如果一个类被广泛地使用并且其成员函数是大型的，那么那个类的实现就应该绝缘。另一方面，有极小访问函数的高度重用的公共组件[1]可能不应该绝缘，除非性能很明显不是一个问题。这些因素概括在图 6-82 中。

[1] 术语 **Public** 在这里意味着在整个系统中被广泛使用的一个低层组件或接口。

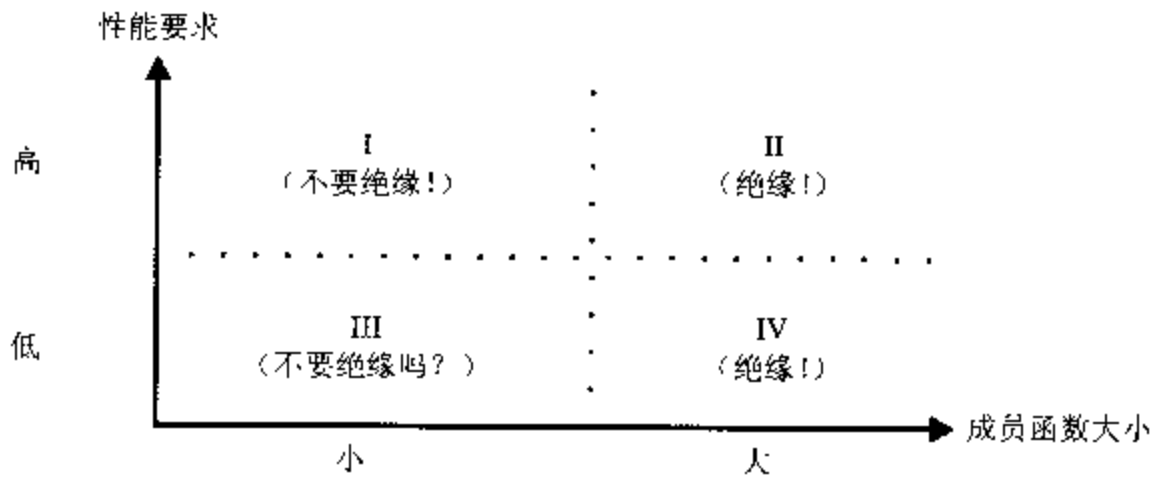


图 6-82 什么时候要绝缘一个被广泛使用的组件

幸好公共的、低层次的类一般在开发过程的早期进行开发、调试和完全的测试。此后，很少修改它们。这些不准备封装的、在全局使用的类几乎变成了系统中的基础类型。像 Point、String 以及 List 这样的类，既经常在内部使用又经常用作主要子系统之间的“交换媒体”。开发者认为这些高度重用的类型不大可能变化。

原 则

绝缘轻量级的、被广泛使用的并且通常通过值返回的对象，会显著地降低整体运行时性能。

对于一个在构造时没有分配额外动态内存的极小的对象来说，通过值返回该对象的一个完全绝缘的版本的额外开销可能会非常严重，会影响到使用它的接口的设计。

```
const Point pt(1,2);

Point getPointA()                                   // return by value
{
    return pt;
}

void getPointB(Point *returnValue)                  // return via parameter
{
    *returnValue = pt;
}
```

图 6-83 显示了部分和完全绝缘 Point 类所增加的、有关从一个非内联函数通过值返回一个 Point 的运行时开销。使用原始的非绝缘 Point 类的图 5-59 中的实现，调用 getPointA 函数来通过值返回一个 Point，在一台 SPARC2 上要花费 1.52 微秒。在保留嵌入到类定义中的数据成员同时，将所有的函数定义移出内联函数，会使得所需时间是前述的两倍多（3.39 微秒）。对该类进行完全绝缘（隐含着数据的动态分配），会使得函数调用花费 10 倍于非绝缘实现所消耗的时间。对于一个极其轻量级的类，如 Point，由于对其逻辑实现进行绝缘而引起的运行时性能的降低可能是无法接受的。

	通过值返回		通过参数返回	
Point 类的描述	SPARC 2	SPARC 20	SPARC 2	SPARC 20
原始 Point 类	1.52	0.75	1.16	0.52
无内联函数	3.39	1.67	1.23	0.71
完全绝缘的版本	15.82	6.96	1.49	0.73

（函数调用所用的微秒数）

图 6-83　返回一个完全绝缘的 Point 对象的开销

我们也许会试图通过传进一个以前构造好的 Point 对象并且赋值给它，代替通过值返回一个绝缘的 Point，来避免与一个临时 Point 对象的构造相关的动态分配的开销。在这种情况下，完全绝缘 Point 的开销，在任一体系结构中都不会超过 30%。由客户程序创建一个可重用的 Point 实例的开销，现在可以由多次重复的、对通过参数列表返回 point 的函数的调用来分摊。这种接口风格类似于在 6.5.4 节中为过程接口建议的那种类型风格，我们将在 9.1.8 节中进一步讨论

不进行绝缘的其他原因是人员短缺。可能没有要进行绝缘的强制性原因，并且可能认为对于获得绝缘所必需的在开发时间上的增量式增加不划算。创建一个包装器需要很好的计划和很多的工作量；时间限制和缺乏经验都可能使潜在可包装的子系统没有被很好地包装。

绝缘还可能因为认为：引入另一个组件而增加的物理复杂性与通过绝缘而获得的潜在利益不相当被省略。协议和包装器都涉及创建一个独立的组件作为接口。这个独立的物理实体会增加物理结构的整体复杂性。

最后，绝缘是对一个组件或子系统的实现的一个附加的、独立的约束。满足这种需求会导致某种更复杂的实现，这种实现可能对于某些人来说更难以理解，并且比没有绝缘的组件或子系统更难以维护。例如，完全绝缘一个类需要在.c 文件中创建一个独立的结构，并且要记住在构造和析构过程中分别动态分配和删除该结构。

增加了初始开发的费用，增加了组件的数量并且增加了复杂度，这些至少是抵制不必要的绝缘的站得住脚的理由。但是，从绝缘可以获得整体维护的好处。在缺少强制性理由的情况下，无论如何请记住，在开发过程的后期，绝缘的删除比绝缘的安装更经济。

原　则

对于大型的、广泛使用的对象来说，要尽早进行绝缘。如果有必要，以后可以选择性地删除该绝缘。

一旦一个系统完成了，并且性能分析证明从一些关键组件删除绝缘可显著提高整个系统的性能，则至少绝缘的好处已在大部分的开发工作中实现了。在一个大型项目结束时才凭经验决定哪个组件可以绝缘而不会在性能上有显著的损失，会牺牲绝缘所能提供的许多初始维

护的利益。

小结：绝缘一个接口可能导致动态分配内存、额外的间接调用、非内联或虚函数调用、动态分配对象的显式管理以及附加的编译单元。对一个接口进行绝缘，在性能、开发工作量或复杂性方面的开销有时会超过绝缘的好处。另一方面，一些组件既不是轻量级的也不是稳定的，并且在整个系统中经常使用。这些大的高层次的、不稳定且被广泛使用的对象比较适合绝缘。

6.6.3 如何进行绝缘

在实践中，将客户程序与类的逻辑实现绝缘的主要方法有两种：

（1）为要绝缘的类提取一个协议。

（2）其他方法：

- 使用部分实现技术。
- 将该类转换为一个完全绝缘的具体类。
- 为该类创建一个绝缘的包装器。
- 为该类创建一个过程接口。

由于一个协议定义一个纯的接口，协议的客户程序不仅不依赖于编译时的实现，而且与其他技术不同，它们在连接时也不需要依赖于任何特定的实现。

已经利用了虚函数的类可能适合通过提取一个协议类而进行“完美”绝缘。这些对象已经被作为基类对象看待，并且它们已经引起了在类的每个实例中承载一个指向虚函数表的指针的额外开销。带有虚函数的基类经常不准备进行实例化。如果一个类既没有声明任何纯虚函数，也没有将其所有构造函数都声明为 non-public，那么基类**不能**由公共的客户程序在程序堆栈上进行实例化。因此，用一个协议来绝缘这样的类，它们的使用不会受到实质上的影响。

对于作为模块的工具类（如图 5-21 中的 GeomUtil），使用其功能时没有必要创立该类的一个实例。在那种情况下，把所有的成员函数声明为静态的和非内联的，并且将所有的静态成员数据移到.c 文件中（在文件的作用域内），可以消除实例化以及虚拟调用机制的开销。

对于一个有着大多数并非微不足道的访问函数的“小型”类来说（例如一个适度的 String 类），进行整体绝缘是合适的。注意，在这里甚至简单函数的实现，例如相等（==）和赋值（=），都潜在地包含循环的、额外的动态分配，或者至少会有另一个对 strcmp 或 strcpy 的非内联函数调用。由于允许不同的实现策略（例如，引用计数和高速缓冲存储器长度）在实际使用的上下文中被简要描述和评估，对这个类的绝缘实际上便于性能调整，而不会使整个系统重新编译。在开发阶段的很后的时期内，（如果需要）总是可以将绝缘删除。

在接口上没有使用继承或虚函数的大型、高层的可实例化对象（例如，一个电路仿真器或解析器），通常可以进行“完全的绝缘”或“包装”，而对对象的大小或运行时开销影响很小。绝缘正是这样的对象所需要的，特别是当对象准备在本软件开发小组之外广泛传播和通

用的时候。

图 4-2 中显示的 p2p_Router 说明了一个完全绝缘的包装器的理想的例子。router 不是一个模块；router 的一个实例在使用它之前必须创建并“编程”。建立 p2p_Router 的一个实例所需要的工作并不是微不足道的，用来对它进行编程的 addObstruction 函数要做的工作也不少。但是，使用该 router 所花费的时间，完全由在 router 子系统的更低层次上对 findPath 函数的每次调用所做的工作来决定。因此，对 router 进行绝缘而增加的运行时间开销完全可以忽略不计。

作为最后一个例子，考虑我们如何对某个其他人的类 Solid 进行绝缘，Solid 的头文件如图 6-84 所示。Solid 准备作为各种 solid 的一个公共基类，但它本身不能实例化。通过观察该类的构造函数和赋值运算符都声明为 protected，可以确认这个意图。

```
// solid.h
#ifndef INCLUDED_SOLID
#define INCLUDED_SOLID
#ifndef INCLUDED_IOSTREAM
#include <iostream.h>
#define INCLUDED_IOSTREAM
#endif

class Solid {
    int d_color;
    double d_scale;
    double d_density;
    ostream *d_errorStream_p;

  protected:
    // STATIC MEMBERS
    static double distance(double x1, double y1, double x2, double y2);

    // CREATORS
    Solid(ostream *errorStream, double density, double scale = 1.0);
    Solid(const Solid& solid);

    // MANIPULATORS
    Solid& operator=(const Solid& solid);
    void setColor(int color) { d_color = color; }

    // ACCESSORS
    virtual double surfaceEquation(double x, double y, double z) = 0;
        // Point(x,y,z) is on the surface when function returns
        // approximately 0 (to within some small tolerance).
    ostream& error() { return *d_errorStream_p; }
    double mass() const { return density() * volume(); }

  public:
    // CREATORS
    virtual ~Solid();

    // MANIPULATORS
    virtual void setTemperature(int degrees) = 0;
        // Changing the temperature may affect color, depending on the
```

```
        // actual object.
    void setScale(int scale) { d_scale = scale; }

    // ACCESSORS
    virtual double temperature() const = 0;
    int scale() { return d_scale; }
    int color() const { return d_color; }
    double density() const { return d_density; }
    double volume() const;
    double centerOfMassInX() const;
    double centerOfMassInY() const;
    double centerOfMassInZ() const;
    // ...
};

#endif
```

图 6-84 带有公共和保护接口的基类 Solid

Solid 的 scale 属性决定对象的相对大小。Solid 的用户可以直接访问和修改 scale 属性。保护的纯虚函数 surfaceEquation 允许一个派生类对惟一的特性进行编程，该特性对于通过一个隐含的方程来描述其自己的接口（被 scale()参数化）是必需的。例如，一个球体的表面积可以描述为

```
double Sphere::surfaceEquation(double x, double y, double z)
{
    return x * x + y * y + z * z - scale() * scale();
}
```

基类中的非虚函数使用表面积方程来计算在其他对象中的 Solid 的体积和在各个空间方向上的质心。使 surfaceEquation 函数成为保护的，阻止了由公共程序对 surfaceEquation 的直接访问[①]。

由于是由派生对象来定义获得和设置温度、以及温度如何影响特定对象的颜色的特性，所以 Solid 的公共函数 temperature 和 setTemperature 声明为纯虚函数。（注意温度的内部表示已经与基类的客户程序绝缘。）

由于所有的对象都有 color 属性（编码为一个整数），基类中提供了一个私有的整数数据成员和一个公共的内联访问器。Solid 的一般客户程序不允许直接设置一个实例的 color 值。而是，要求它们调整温度，该温度反过来可以影响对象的 color 属性。因此，setColor 操纵函数是保护的，这样只有派生对象本身才能直接改变实例的 color 属性。

公共接口提供了几个访问函数，其中一些（例如 volume）在被调用时作实质性的数字的工作。保护的接口为派生类作者提供了几个助手函数，例如 setTemperature。这些助手函数在实现所需要的虚函数时被证明是有用的。

① 通过使该函数成为私有的，也能取得希望的效果，但那会使得派生类开发者的任务更不明显。

不是在保护接口上直接暴露 ostream 指针数据成员，而是由保护函数 error 来提供一个方便的流引用以报告错误（例如设置温度太高或太低）。质量在决定颜色（特别是为一个非常大的质密的 Solid，如黑洞）时可能也要起作用。计算质量的保护成员函数 mass 使用公共接口上提供的成员，它是为了方便派生类作者而提供的。最后，distance 是一个派生类作者经常使用的函数。distance 和 mass 助手函数不同，它不依赖于任何一个类的实例，因而被设置为类 Solid 的一个保护的静态成员。

很明显，基类 Solid 的最初作者不认为绝缘是一个重要的设计标准，Solid 中随意地使用内联函数就是这种判断的证据。幸好，我们有几种技术可以用来改善 Solid 实现的绝缘。这些绝缘改进的属于两种基本类别：整体的和部分的。

作为一个练习，让我们首先看一下我们可以对 Solid 类作哪种类型的增量式改进：

```
# fndef INCLUDED_IOSTREAM
#include <iostream.h>
#define INCLUDED_IOSTREAM
#endif
```

无需过多考虑，我们就可将上面的代码转换为 class ostream，以消除不必要的对 iostream 头文件的编译时依赖，并且在启动时不必在包含 solid.h 的每个编译单元中创建一个静态的伪对象（见 7.8.1.3 节）。

```
protected:
  static double distance(double x1, double y1, double x2, double y2);
```

由于 distance 是一个静态函数，所以它不依赖于 Solid 实例数据；它可以很容易地被移到一个独立的工具组件中。

```
protected:
  double mass() const { return density() * volume(); }
```

由 mass()所执行的计算只依赖于从 Solid 的公共接口可直接访问的属性。一个修改后的静态 mass 函数可以移入一个独立的工具组件中。

```
class ostream;          // already changed from #include <iostream.h>

  private:
    ostream *d_errorStream_p;

  protected:
    ostream& error() { return *d_errorStream_p; }
```

将一些派生 Solid 对象设置为任何温度并且避免出错是可能的。一些派生类作者可能会发现错误流函数 error()是一个很有用的方便措施。我们可以只是将 d_errorStream_p 数据成员从分解后的实现中删除（正如我们在图 6-26 所示的图形子系统中对 Scribe 所做的那样），并让派生类作者只在需要的时候实现一个错误流。

```
private:
  int d_color;
  double d_scale;
  double d_density;

protected:
  void setColor(int color) { d_color = color; }

public:
  int scale() { return d_scale; }
  int color() const { return d_color; }
  double density() const { return d_density; }
```

如果我们假设没有一个 Solid 的内联函数会被非常频繁地调用，以致于会产生性能问题，那么我们可以将所有的内联函数转换为非内联函数。这样做也使我们能够将一个分解的数据收集到定义在.c 文件内的一个 struct 中。一个更绝缘一些的 Solid 基类的版本如图 6-85 所示。

```
// solid.h
#ifndef INCLUDED_SOLID
#define INCLUDED_SOLID

class Solid_i;                  // insulated member data

class Solid {
    Solid_i *d_this;

  protected:
    // CREATORS
    Solid(double density, double scale = 1.0);
    Solid(const Solid& solid);

    // MANIPULATORS
    Solid& operator=(const Solid& solid);
    void setColor(int color)

    // ACCESSORS
    virtual double surfaceEquation(double x, double y, double z) = 0;
        // Point(x,y,z) is on the surface when function returns
        // approximately 0 (to within some small tolerance).

  public:
    // CREATORS
    virtual ~Solid();

    // MANIPULATORS
    virtual void setTemperature(int degrees) = 0;
        // Changing the temperature may affect color, depending on the
        // actual object.
    void setScale(int scale);
```

```
    // ACCESSORS
    virtual double temperature() const = 0;
    int scale();
    int color();
    double density() const;
    double volume() const;
    double centerOfMassInX() const;
    double centerOfMassInY() const;
    double centerOfMassInZ() const;
    // ...
};

#endif
```

图 6-85　更绝缘一些的抽象类 Solid

此时，我们不得不使用某种形式的整体绝缘技术将工作做得再更好一些。按现在的情况，我们不能对该类的实现进行完全的绝缘，因为它使用了虚函数。对这个类进行包装会阻碍一般用户按自己的意愿派生新的 Solid。在本章提供的绝缘技术中，提取一个协议是目前最好的选择方案。

正如图 6-30 所显示的 Car 类那样，我们不能简单地删除所有的保护函数，并把它们放到一个独立的工具类中，因为它们与实例本身有密切的交互。也就是说，在保护接口中的函数（例如，setColor）只是提供来实现定义在派生类中的虚函数（例如，setTemperature）的。同时，这些保护函数直接依赖于实例信息（例如，d_color），客户程序可以通过定义在该基类中的公共函数（如，color）访问这些实例信息。我们被迫提取一个协议来实现整体绝缘，主要是因为虚函数对固有实例数据的依赖。

图 6-86 显示了从原始的或部分绝缘的 Solid 中提取的一个协议的结果。注意一个问题，如何提取一个协议类才能**总是**使我们避免暴露基类的保护成员。

```
// solid.h
#ifndef INCLUDED_SOLID
#define INCLUDED_SOLID

class Solid {
  public:
    // CREATORS
    virtual ~Solid();

    // MANIPULATORS
    virtual void setTemperature(int degrees) = 0;
        // Changing the temperature may affect color, depending on the
        // actual object.
    virtual void setScale(int scale);

    // ACCESSORS
```

```
    virtual double temperature() const = 0;
    virtual int scale();
    virtual int color();
    virtual double density() const;
    virtual double volume() const;
    virtual double centerOfMassInX() const;
    virtual double centerOfMassInY() const;
    virtual double centerOfMassInZ() const;
    // ...
};

#endif
```

图 6-86 协议类 Solid

6.6.4 绝缘的程度

虽然第 4 章中提供的 p2p_Router 的实现是绝缘的，但是它不是“完全绝缘的”（就像 6.4.2 节所讨论的那样），因为包装器类拥有一个指向另一个类的指针，而组件不能惟一且完全地控制那个类。例如，我们不可能在不破坏独立测试的实现组件 p2p_RouterImp 的情况下，将一个绝缘的数据成员添加到 p2p_Router 类中。

原 则

有时整体绝缘的运行时开销不会比部分绝缘大。

在许多情况下，这种绝缘程度可能是足够好的。但是，如果 p2p_router 定义了一个非常公共的接口，那么我们可以做得更好。一个完全绝缘的 p2p_router 组件会（提前）声明自己的实现结构（例如，p2p_Router_i）。然后，在 p2p_router.c 文件中，struct p2p_Router_i 将由类型 p2p_RouterImp 的单个嵌入成员来定义：

```
// p2p_router.c
#include "p2p_router.h"
#include "p2p_routerimp.h"

struct p2p_Router_i {
    p2p_RouterImp d_imp;
};

// ...
```

现在将一个“私有的”数据成员添加到一个 p2p_Router_i 中，既不会影响 p2p_Router 的客户程序，也不会要求改变 p2p_RouterImp：

```
// p2p_router.c
#include "p2p_router.h"
#include "p2p_routerimp.h"
```

```
struct p2p_Router { {
    p2p_RouterImp d_imp;
    int d_moreData;       // added fully insulated detail
};
```

在这种情况下，要正确地完成任务只需要多做一点点工作，但却在完全没有影响运行时性能的情况下获得了整体绝缘。

原　则

有时获得最后 10%的绝缘要以增加 10 倍的运行时间为代价。

有时最后一点点绝缘的获得可能要付出很大的代价。回想一下，在 6.4.3 节中我们就选择不对所有的 graph 组件类进行完全的绝缘。如果那样做，我们在运行时性能方面要付出不成比例的高代价。为了说明这个原则，考虑我们在本章和前几章中已经介绍的四个相关的 graph 子系统的实现：

系统 1：分解的对象。这个子系统对应于 5.9 节中的分解的实现。在这个体系结构中，子系统是可层次化的，但是客户程序为了使用图形必须知道和使用更低层次上的组件。

系统 2：封装的包装器。这个子系统对应于 5.10 节中的封装的包装器实现。在这个体系结构中，客户程序可以从单个包装器组件做任何必要的工作。

系统 3：绝缘包装器。这个子系统对图 6-53 中提出的五个包装器类中的其中三个的实现进行了完全绝缘。剩下的两个类 NodeId 和 EdgeId，在它们的物理接口（但不是逻辑接口）上暴露了它们各自的实现类的名称 Gnode 和 Gedge。

系统 4：完全绝缘的包装器。这个子系统对图 6-53 中提出的所有五个包装器类的实现进行了完全绝缘。

系统 1 更侧重运行时性能而不是封装。对底层接口的修改会使客户程序重新编程。系统 2 在图形子系统上加了一层很薄的封装包装器。从逻辑上说，客户程序是独立的，但是，它们没有与低层次上组件的改变绝缘，可能被迫重新编译。系统 3 将系统 2 的封装包装器转变为一个几乎完全的绝缘包装器。对任何基础组件的改变都不能在编译时影响包装器的客户程序；但是，不可能在不影响 Gnode(Gedge)或客户程序的情况下，添加私有数据到 NodeId(EdgeId)中。系统 4 展示了一个完全绝缘的包装器：对这五个包装器类实现的任何改变都既不会影响客户程序也不会影响基础实现组件。

为了在一定的操作条件的范围内说明这些不同 graph 体系结构的运行时开销，我们编写了一个小型的测试程序来进行一系列的实验。在该程序中，graph 子系统被用于创建任意的 graph 结构，如图 6-87 所示。在这个图中，每条边的权值碰巧为 1，但是边的特定的权值不会影响实验。

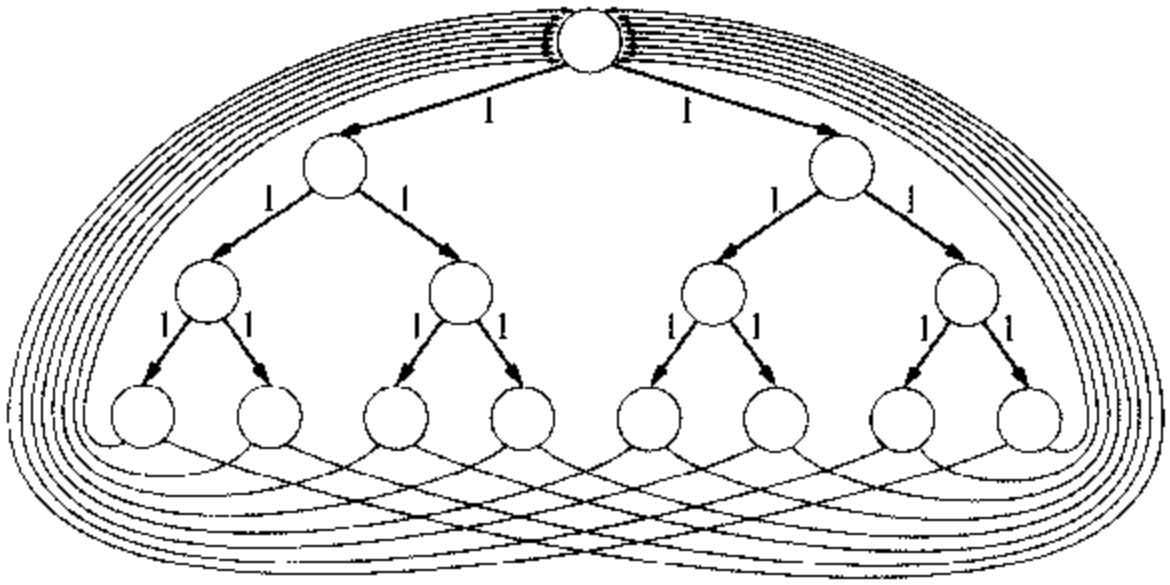


图 6-87 由 15 个节点（每个节点的度为 3）组成的任意图形

在建立了该图的一个实例后，程序调用了 NodeIter，在图中的所有 15 个节点上进行迭代，并累加了在每个节点上调用 sum 得到的值。递归函数 sum 从指定的节点到指定的深度对图进行遍历，沿着遍历的路径累加所遇到的边的权值。由于 sum 遍历的是二叉树，所以 sum 的运行时间与遍历的深度成指数比例。

实际测试程序的源代码在图 6-88 中提供。测试驱动程序的第一个命令行参数指定的是 sum 遍历图的深度。第二个命令行参数指定了重复（相同）试验的次数；这第二个参数用于获得一个平均迭代的准确时间。

```
// graph.t.c
#include "graph.h"
#include "node.h"
#include "edge.h"
#include <iostream.h>
#include <stdlib.h>

double sum(const NodeId& node, int depth)
{
    double result = 0;
    if (depth > 0) {
        for (EdgeIter it(node); it; ++it) {
            if (it().from() != node) {
                continue;
            }
            result += it()->weight();
            result += sum(it().to(), depth - 1);
        }
    }
    return result;
}

main (int argc, char *argv[])
{
    int depth = 1; int repeat = 1;
```

```
if (argc > 1) depth = atoi(argv[1]);
if (argc > 2) repeat = atoi(argv[2]);
cout << "GRAPH: depth = " << depth
          << "  repeat = " << repeat << endl;

double total;

for (int i = 0; i < repeat; ++i) {
    Graph g;
    NodeId n = g.addNode("n");

    NodeId n0 = g.addNode("n0");
    NodeId n1 = g.addNode("n1");
    g.addEdge(n, n0, 1);
    g.addEdge(n, n1, 1);
    NodeId n00 = g.addNode("n00");
    NodeId n01 = g.addNode("n01");
    NodeId n10 = g.addNode("n10");
    NodeId n11 = g.addNode("n11");
    g.addEdge(n0, n00, 1);
    g.addEdge(n0, n01, 1);
    g.addEdge(n1, n10, 1);
    g.addEdge(n1, n11, 1);

    NodeId n000 = g.addNode("n000");
    NodeId n001 = g.addNode("n001");
    NodeId n010 = g.addNode("n010");
    NodeId n011 = g.addNode("n011");
    NodeId n100 = g.addNode("n100");
    NodeId n101 = g.addNode("n101");
    NodeId n110 = g.addNode("n110");
    NodeId n111 = g.addNode("n111");
    g.addEdge(n00, n000, 1);
    g.addEdge(n00, n001, 1);
    g.addEdge(n01, n010, 1);
    g.addEdge(n01, n011, 1);
    g.addEdge(n10, n100, 1);
    g.addEdge(n10, n101, 1);
    g.addEdge(n11, n110, 1);
    g.addEdge(n11, n111, 1);
    g.addEdge(n000, n, 1);
    g.addEdge(n001, n, 1);
    g.addEdge(n010, n, 1);
    g.addEdge(n011, n, 1);
    g.addEdge(n100, n, 1);
    g.addEdge(n101, n, 1);
    g.addEdge(n110, n, 1);
    g.addEdge(n111, n, 1);
    g.addEdge(n000, n, 1);
    g.addEdge(n001, n, 1);
    g.addEdge(n010, n, 1);
    g.addEdge(n011, n, 1);
    g.addEdge(n100, n, 1);
    g.addEdge(n101, n, 1);
    g.addEdge(n110, n, 1);
```

```
        g.addEdge(n111, n, 1);

        total = 0;
        for (NodeIter it(g); it; ++it) {
            total += sum(it(), depth);
        }
    }

    cout << "total = " << total << endl;
}
```

图 6-88 测量 Graph 子系统的运行时效率的测试驱动程序

测试驱动程序运行的深度为 0~20，在上述四个系统中，各层次的重复次数为：1000（深度 0~5）、100（深度 6~10）、10（深度 11~15）、1（深度 16~20）[1]。这个很具说服力的实验的结果如图 6-89 所示。

	在 SUN SPARC 20 上的运行时间（CPU 秒）			
深度	分解的（原始的） 系统 1	封装的包装器 系统 2	大多数绝缘的包装器 系统 3	完全绝缘的包装器 系统 4
0	0.0018 （100%）	0.0018 （100%）	0.0020 （111%）	0.0033 （183%）
1	0.0019	0.0020	0.0026	0.0115
2	0.0022	0.0024	0.0050	0.0381
3	0.0026	0.0031	0.0086	0.0675
4	0.0033	0.0043	0.0144	0.1023
5	0.0046 （100%）	0.0063 （137%）	0.0248 （539%）	0.1438 （3126%）
6	0.009	0.014	0.063	0.334
7	0.016	0.025	0.120	0.609
8	0.027	0.044	0.213	1.054
9	0.048	0.078	0.380	1.836
10	0.121 （100%）	0.202 （167%）	0.998 （825%）	4.895 （4045%）
11	0.23	0.38	1.91	9.37
12	0.41	0.68	3.40	16.42
13	0.74	1.22	6.06	28.94
14	1.92	3.22	15.95	77.87
15	3.69 （100%）	6.15 （167%）	30.51 （827%）	148.44 （4023%）
16	6.6	10.9	54.4	262.2
17	11.8	19.4	97.0	462.4
18	30.7	51.5	255.1	1245.5
19	58.9	98.5	488.1	2374.4
20	105.8 （100%）	175.2 （166%）	870.6 （823%）	4194.8 （3965%）

图 6-89 各种 Graph 系统体系结构的运行时开销

① 对这个测试驱动程序作了少许修改，以适应系统 1 的略微不同的接口。

当遍历图的深度指定为 0 时，不会发生图的遍历。大多数时间花费在建立和拆卸图结构上。这种操作本质上相对昂贵；正如图 6-89 第一行所指出的，封装甚至绝缘的影响都很小，可以忽略不计。当完全绝缘时，运行时开销比不绝缘时的开销高出 83%。这是因为通过值从 NodeIter 返回一个完全绝缘的 NodeId 的开销有非常明显的增加。

当我们增加图形的深度时，遍历的开销开始影响整体性能。用于读取一个图中的信息的函数比那些用于建立图的函数要小得多，并且每次调用所做的工作也要少得多。但是，这些轻量级的函数在图的遍历过程中要调用很多很多次。

深度为 5 时，实验运行在系统 1 上所花费的时间是深度为 0 时所花费时间的 2.5 倍；但是，发生了许多倍于那个数字的额外函数调用。如果这些小函数的开销过于昂贵，那么运行时的性能就会受到影响。在同样的深度上，封装的系统 2 比未包装的系统 1 的运行时间增加了 37%。部分绝缘的系统 3 则使实验花费了 5 倍的时间。在系统 4 中，由于对 NodeId 和 EdgeId 进行整体绝缘而引起的动态分配则花费了 30 倍的时间！

深度为 10 的时候，实验在系统 1 上所花费的时间是深度为 0 时的 100 倍。现在花费在调用“小”函数上的时间占了运行时间的大部分。对于一个封装的包装器（系统 2），实验的运行时间要多 67%。对于一个绝缘的包装器（系统 3），实验花费了超过 8 倍的时间，而对于完全绝缘包装器（系统 4），实验花费了整整 40 倍的时间。

通过从这一点往下仔细分析图 6-89，我们可以看到我们已经到达了另外的渐近线：增加深度不会再进一步增加这些 graph 子系统（变量）的运行时性能的比率。

从这个实验中我们能得到什么启示呢？

（1）如果一个对象的函数已经在绝缘层之下做了大量的工作，那么对该对象的实现进行绝缘在运行时性能上不会有明显的影响（表明绝缘的层次是合适的）。

（2）如果一个子系统的函数已经在封装层之下做的工作不是微不足道的，那么对该子系统用包装器进行封装，对运行时性能的影响是可以忽略不计的（表明封装的层次是合适的）。

（3）为频繁调用的轻量级函数提供一个封装包装器，会显著地影响整体性能（也许意味着封装的层次应该升级）。

（4）为频繁调用的轻量级函数提供一个绝缘包装器，会对整体性能产生压倒性的影响（强迫绝缘的层次应该升级）。

（5）如果极小对象经常被通过值返回，那么为该极小对象提供整体绝缘包装器会对整体性能产生灾难性的影响（强迫减少绝缘的程度或者升级绝缘的层次）。

6.7 小结

在本章我们介绍了**绝缘**的概念，它是通常称为“封装”的逻辑概念在物理上的相似概念。如果一个组件的实现细节修改后，不会迫使该组件的客户程序重新编译，那么该组件的实现细节就是被绝缘了。

下面几种结构被认为可能会潜在地导致不希望的编译时耦合：

- **继承**和**分层**迫使客户程序看到继承的或嵌入对象的定义。
- **内联函数**和**私有成员**把对象的实现细节暴露给了客户程序。
- **保护成员**把保护的细节暴露给了公共的客户程序。
- **编译器产生的函数**迫使实现的变化影响声明的接口。
- **包含指令**人为地制造了编译时耦合。
- **默认参数**把默认值暴露给了客户程序。
- **枚举类型**会引起不必要的编译时耦合，因为不合适的放置或不适当的重用。

在其他条件都一样的情况下，将一个特定的实现细节与一个客户进行绝缘比不进行绝缘要好——即使其他细节仍然不是绝缘的。部分实现技术可用来降低编译时耦合的程度，并且不会招致整体绝缘可能隐含的开销：

- 通过将 WasA 关系转化为 HoldsA 关系来消除私有继承。
- 通过将 WasA 关系转化为 HoldsA 关系来消除嵌入的数据成员。
- 通过使文件作用域中的私有成员函数成为静态的，并将它们转到.c 文件中，来消除私有成员函数。
- 通过建立一个独立的工具组件和/或提取一个协议来消除保护成员函数。
- 通过提取一个协议和/或移动静态数据到.c 文件中，来消除文件作用域中的私有数据成员。
- 通过显式地定义编译器产生的函数来消除这些函数。
- 通过删除不必要的包含指令或提前用类声明替换它们来消除包含指令。
- 通过用无效的默认值替换有效的默认值或使用多函数声明来消除默认参数。
- 通过将枚举类型重新定位到.c 文件中，用 const 静态类成员数据替换它们，以及在使用枚举类型的类中重新分布枚举类型，来消除枚举类型。

对于广泛使用的接口来说，避免对基础实现细节的所有编译时依赖是非常值得做的工作。我们讨论了三种用于把客户程序与所有的实现细节绝缘的一般绝缘方法：

- 协议类：建立一个抽象的“协议类”是一种通用的绝缘技术，可用以分离一个抽象基类的接口和实现。不仅客户程序可以与编译时实现细节的变化绝缘，而且可以消除对一个特定实现的连接时依赖。
- 对具体类进行完全绝缘：一个“完全绝缘的”具体类拥有一个指向一个全部定义在.c 文件中的私有结构的不透明指针。该 struct 包含了以前在原始类的私有部分中的所有实现细节。
- 绝缘包装器组件：封装包装器组件的概念（来自第 5 章）可以扩展为完全绝缘包装器组件。包装器一般用于对几个其他的组件甚至整个子系统进行绝缘。与过程接口不一样，一个包装器层要求预先进行相当多的计划工作和非常严密的设计工作。特别地，要小心进行多组件包装器的设计，以避免对远距离友元关系的需求。

一个**过程接口**是函数的集合，这些函数都是现有函数集合的成员，并将功能集的一个子集暴露给最终用户。过程接口是整体绝缘的一种选择方案。与本章提出的其他三种整体绝缘技术不同，过程接口既不是逻辑封装，也不是整体绝缘。一个过程接口考虑了对非常大型系统（还没有用过程接口进行设计）起作用的独特的因素。

一般来说，如果一个组件在系统中被广泛地使用，那么其接口应该绝缘。但是，不是所有的接口都应该绝缘。例如，绝缘可能不是切实可行的，特别是对于轻量级的、可重用的组件来说。选择不对一个组件进行绝缘的常见原因包括如下几个：

- 暴露：客户程序的数量可能已知是很少的。
- 访问数据的时间：该类可能已嵌入了数据并且有效地利用了极小的内联函数来访问它。
- 创建对象的时间：一个极小的类（例如，Point）可能还没有分配动态内存。
- （初始）开发的开销：可能没有进行绝缘的强迫性理由；额外的开发工作可能是不合算的。
- 组件的数量：绝缘可能要求需要另一个组件（例如，用以保留一个协议或包装器），从而增加了维护的开销。
- 组件复杂性：一个绝缘的实现（例如，一个“完全绝缘的”定义在.c 文件中的 struct）可能比一个没有绝缘的实现更难以理解和维护。

7 包

一个大型项目可能跨越许多开发人员、若干管理层甚至多个地理位置。系统的物理结构不仅反映了应用程序的逻辑结构，也反映了实现它的开发队伍的组织结构。大型系统需要分层次的物理组织，这超出了个体组件的一个可层次化层次结构单独所能完成任务的范围。为了完成更复杂的功能，我们需要在更高的抽象层次上引进一个物理设计单位。本章研究支持超大型系统开发所需的物理结构。特别地，我们引入了一个物理设计的宏观单位，本书称之为“包”。

包将相关组件的集合聚集为一个逻辑上内聚的物理单位。每个包都有一个与之关联的**注册**前缀，它直接将文件和文件作用域逻辑结构标识为属于那个包。本章头两节将介绍包的语义和物理结构之后，在 7.3 节我们将层次化的概念应用在包级别上。在这里我们发现把许多在组件级上应用的技术应用到整个包时同样也起作用。但同时，也出现了必须单独处理的新问题。在 7.4 节，我们将探讨在包级别上的绝缘概念，以提高复杂了系统对客户的可用性。然后，在 7.5 节，我们通过将包进行层次化分组，扩展项目规模的范围。在 7.6 节，我们将讨论发布一个系统的稳定快照的过程，介绍发布一个大型系统的可能的目录结构。然后我们将研究一种用于在发布版本之间更新已发布软件的、称为**修补（patching）**的技术。接着在 7.7 节，我们将研究 main()在面向对象软件系统中的角色，以及拥有一个文件的“顶”的特殊特权和责任。最后，在 7.8 节，我们将研究程序执行生存期的最初几个时刻，正是在这个时刻，所有文件作用域内的静态数据被（潜在地）初始化。

在大型系统中，静态初始化会导致令人无法接受的漫长启动时间。我们将研究四种可供选择的初始化策略，并比较它们的相对优势和弱势。我们也将研究程序退出之前的清理需求，以便于进行内存回归测试。

7.1 从组件到包

在第 3 章，我们将组件介绍为物理设计的最小单位。一个普通的组件包含一个、两个甚至多个类，经常伴随着适当的自由运算符。通常一个组件由数百行 C++源代码和注释构成，经常有相当长度的.h 文件和.c 文件。偶尔，一个在低层次上定义的组件会有少于 100 行的代码，并且.c 文件是空的。有时大型子系统的包装器组件或者机器生成组件有数以千计的代码行。但是，根据经验，数百到一千行是组件保持可理解、测试和重用性的实际可行的大小。

正如读者在第 4 章的 p2p_router 例子中所看到的，可以只使用少量的组件来构造相当复杂的子系统。在那个例子里，在单个组件接口内声明的高层次功能的实现被分散到组件的一个层次结构中，这极大地提高了它的易测试性。很容易支持一个有数万行代码的系统，不必进一步分割。但是如果我们的系统比这还大呢？假设它们由数十万行代码组成。我们怎样来处理差不多有几百个组件的物理组织？与以往一样，我们仍然采用经过实践检验的可靠方法，即用抽象和层次结构来处理复杂性。

从最高层开始设计一个系统时，几乎总是有大的片断值得作为独立单位进行抽象讨论。考虑一种大型语言（如 C++）的解释程序的设计，如图 7-1 所示。在这个设计中所描述的每个子系统都可能太大太复杂了，不适合放在一个单个的组件内。这些较大的单位（在图 7-1 中以双层方框表示），每一个都可作为可层次化组件的一个集合来实现。

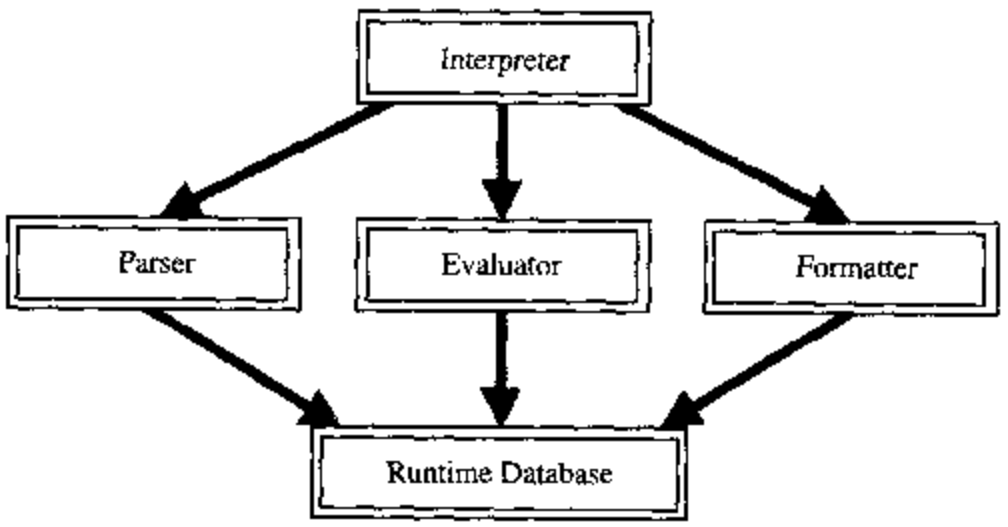


图 7-1 高层次解释程序体系结构

图 7-1 中这些较大单位之间的依赖表示为一个**封套（envelope）**，它包含构成每个子系统的组件之间的聚集依赖。例如，运行时数据库（runtime database）是一个独立的子系统：它没有对任何外部组件的依赖。每个解析器（parser）、求值程序（evaluator）以及格式程序（formatter）子系统都有一些组件依赖于运行时数据库中的一个或多个组件，但是这三个子系统中的任何一个都没有一个组件依赖于其他两个并行子系统中的任何组件。构成顶层解释程序（interpreter）的组件依赖于每一个并行子系统内部的组件（或许直接依赖于运行时数据库中的组件）。在将项目开发工作分配给多个个人、开发组或地理位置时，小心地将系统划分成

大单位，然后考虑这些单位之间的聚集依赖是非常关键的。

虽然图 7-1 中的设计不会被认为是一个大项目，但它很可能被分配给多个开发人员。有一个天然的划分允许若干开发人员在这个项目上并行工作。在运行时数据库设计完之后，三个并行的开发工作就可以在解析功能、求值功能和格式功能上起动了。一旦这些片断各就各位，顶层解释程序的实现和测试就可以开始了。

到目前为止，我们是将这些分离的子系统作为没有实际物理划分的概念单位来进行讨论的。如果整个项目预期只需要 2 万行代码并由一个开发人员来实现，也许就没有将整个体系结构划分成不同的物理单位的迫切需要。但是，如果这个设计（假定）有 8 万行代码，或者在任何给定的时间内会有不止一个开发人员在这个项目上工作，那么就会有更强烈的需求要把概念上的物理划分变成具体现实。

> **定义**：一个包（package）就是被组织成一个物理内聚单位的组件集合。

术语**包**通常是指一个非循环的、层次化的组件集合，这些组件共同有着一个内聚的语义目标。在物理上，一个包由一个头文件的集合和一个包含相应目标文件（.o）中的信息的单个库文件组成。一个包可能由低层次、可重用组件的一个松散耦合的集合构成，例如原来的 AT&T 的标准组件库[①]，以及现在的在 Hewlett-Packard[②]开发的新的标准模板库（Standard Template Library，STL）。一个**包**也可能由一个有特殊用途的、只打算给一个客户使用的子系统构成，例如第 4 章的 p2p_router 子系统。

图 7-2 描绘了一个文件系统内部的包的可能的组织。在这个组织中，所有的包都存在于目录结构的同一层次上，没有考虑它们的物理相互依赖。所有的头（被要求在一个给定的包之外）都放在一个单个的、被称为 include 的系统范围的目录中。对应于每个包的库文件则被放在一个单个的、被称为 lib 的系统范围目录中。

每个包目录都包含为组件（与那个包相关的）保存源代码的文件。正如图 7-2 所示，包 pk 在它的源目录中包含了 n 个组件：pk_c1、pk_c2、…、pk_cn。每个组件（例如 pk_ci）有一个相关的头文件（pk_ci.h）、一个实现文件（pk_ci.c）以及一个可用来试运行在组件中增量式实现的功能的独立测试驱动程序（pk_ci.t.c）。注意，若要有效，则这些分层次的测试驱动程序应该被看作是如同它们所测试的组件一样，系统源代码的重要部分。很容易通过它们的.t.c 后缀将这些驱动程序与实现文件区别开来。

除了源目录，在每个包目录下还有两个文件。依赖（dependencies）文件保存这个包被授权依赖的所有其他包的名称。也就是说，为了使用这个包，客户将不必包含或连接到任何定义在另一个包中的其他组件，除非在与这个包相关的依赖文件中指定了那个包。虽然包依赖很少改变，但偶尔也会发生。详细说明这些依赖是系统体系结构设计者的工作；检验它们则

---

① **stroustrup94**，8.3 节，184~185 页。

② STL 已被接受为 ANSI/ISO (Draft) C++ Standard 的一部分（见 **musser**）。

是能够并应该自动化的一个过程。

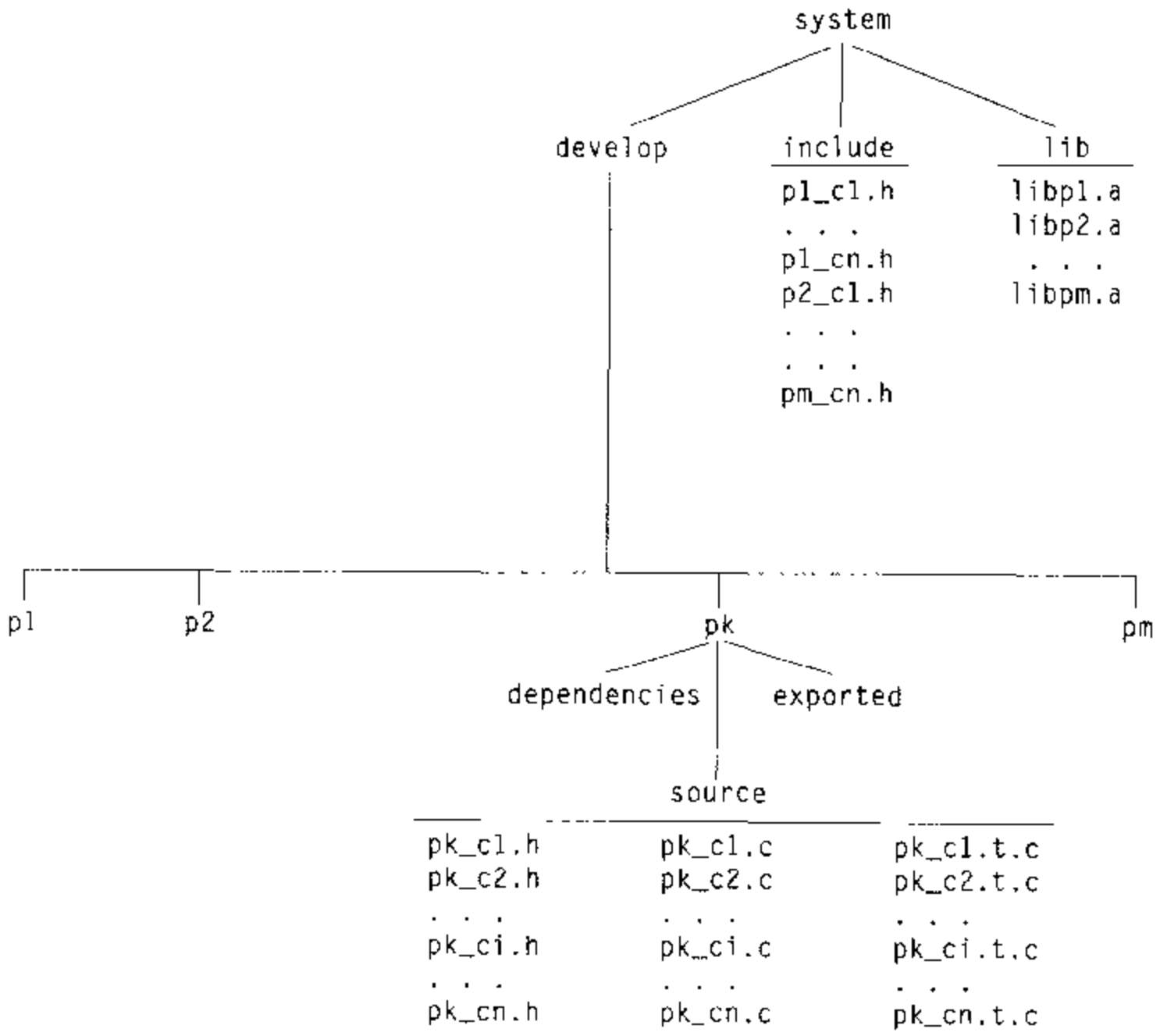


图 7-2 一个简单的包的开发组织

输出（exported）文件包含一列组件头，它们将被放在图 7-2 的系统范围的 include 目录中，供一般客户使用。因为不是所有定义在包里的头都打算供外部客户使用，所以，可以被输出的头的集合可能是定义在包中的组件的一个真子集。

将用于包之外的头放在单个的 include 系统目录中，可以更方便地指定在哪里寻找输出头文件。只输出其他包所需要的头的子集，减少了客户为使用该产品而必须艰难穿越的混乱。把这些头放在单个的目录中，也可以提高客户与访问多处包目录相关的编译时效率（见 7.6.1 节）。将库文件放在单个目录中只是为了更方便地使用它们。

到现在为止，我们只在组件级研究了层次化。回想 4.7 节，不依赖任何其他（局部）组件的组件被指定为 level 1。局部组件，我们指的是定义在我们的包中的组件；定义在其他包中的组件被指定为 level 0。

图 7-3 说明了我们一直用来处理我们的子系统（pkgb）对另一个子系统（pkga）的依赖

的方式。在分层次测试我们自己的包时，我们假设定义在我们的包之外的组件是已经测试过的，并且已知其内部是正确的。因而我们可以把任何一个这样的外部组件的层次号指定为 0（相对于我们的局部组件）。在我们自己包内的组件，若它们不依赖于本包中的任何其他的局部组件（例如，i 和 j），则被定义为具有层次号 1。局部依赖于 level 1（不是更高层次）中组件的组件（例如，k 和 l）是在 level 2 上。

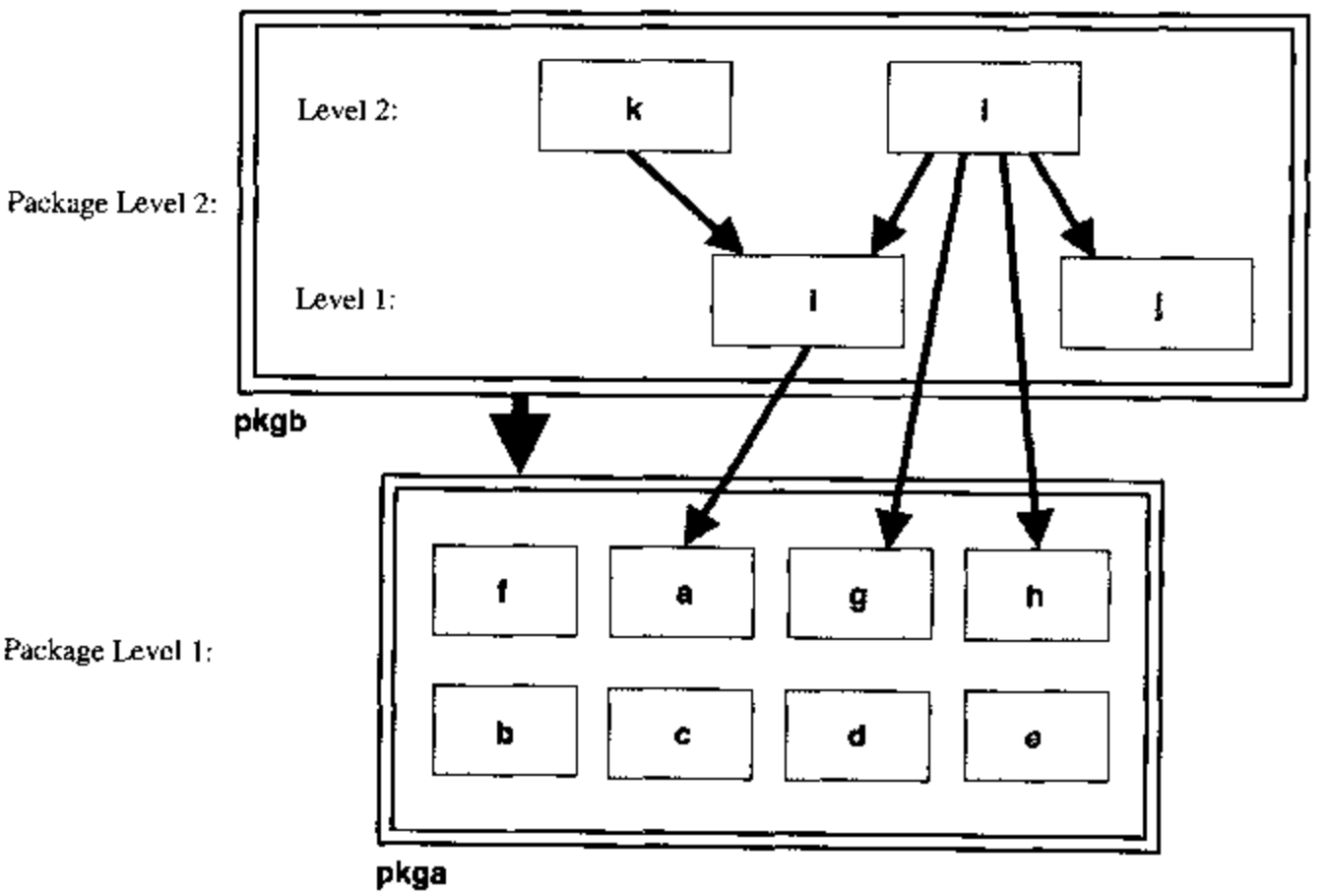


图 7-3 对其他包中的组件的依赖

> **定义**：如果包 x 中的一个或多个组件依赖（DependsOn）另一个包 y 中的一个或多个组件，则称包 x **依赖（DependsOn）**包 y。

正像定义在组件内的逻辑构造之间的关系隐含物理依赖一样（见 3.4 节），包之间的依赖由构成它们的组件之间的单个依赖所隐含。例如，在图 7-3 中，pkgb 中的组件 i 依赖（DependsOn）pkga 中的组件 a，pkgb 中的组件 l 依赖（DependsOn）pkga 中的组件 g 和 h。所以按照定义，pkgb 依赖（DependsOn）pkga。倘若 pkga 不反过来依赖 pkgb，那么我们可以将这些包作为一个整体来指定层次号，正如我们对一个包内的单个组件所做的那样。

包给开发人员和体系结构设计者同样地提供了强大的抽象机制。图 7-4（a）显示了一个有 20 个组件的集合，从 a 到 t，分组成四个包：pkga、pkgb、pkgc 和 pkgd。每个包定义一个高层次的体系结构单位，由协同操作组件的一个内聚的层次结构构成，它们为了共同的目的而联合。

与之相对，图 7-4（b）显示了同一个系统，它被表示为由单个组件构成的一个未打包的、

可层次化的集合。高层次体系结构的模块性和抽象消失了；我们已经失去了附着在这些在自顶向下设计过程中创建的高层次划分上的语义值。

如图 7-4（a）所示，图 7-4（b）中的跨越包边界的单个组件依赖已经被抽象掉了，取而代之以整个包的依赖。例如，显示在图 7-4（b）中的组件 p 对组件 d 的依赖，以及组件 q 对组件 e 和 f 的依赖，在图 7-4（a）中全部由 pkgc 对 pkga 的包依赖来表示。

注意，在图 7-4（a）中的每个包内的局部组件层次号仍然以 level 1 开始。这还是因为对其他包的依赖被看作是“初级输入（primary input）”（见 4.7 节），另外，为了分层次测试的目的，这些依赖被假设是正确的。正如我们常见的那样，这些包每一个都含有叶子组件（即像 t 这样的不依赖于该系统中其他任何组件的组件）。在一个未打包的系统中[图 7-4（b）]，这些叶子组件都将有一个绝对的组件层次号 1。因此许多组件都有下降到未打包图表的较低层次的倾向，这可能会使它们的目标变得模糊。通过将这些叶子组件连同它们的客户程序一起打包，我们可以提高系统的模块性。

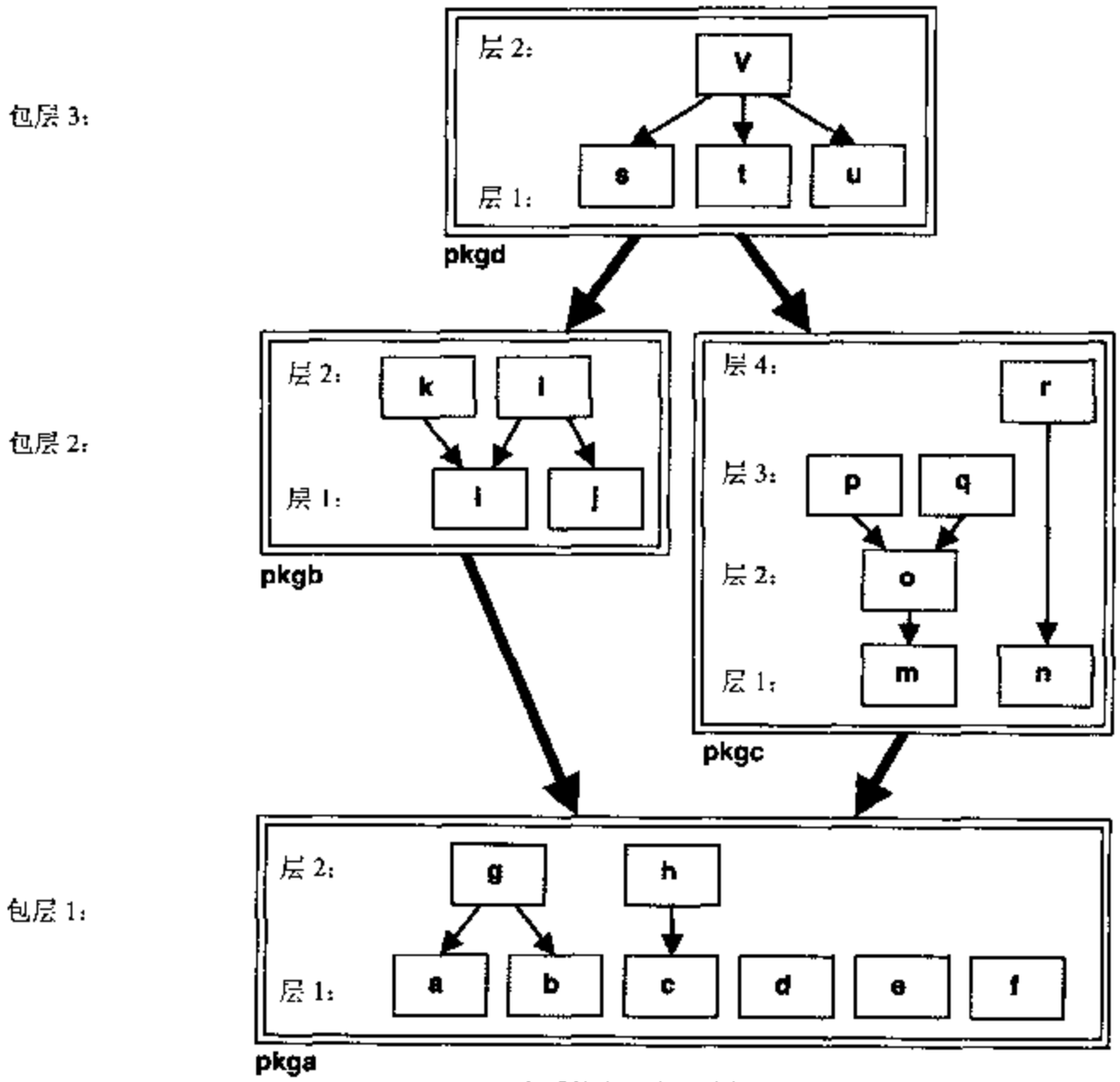
通常一个包会拥有许多组件。当一个典型的组件可能由 500 到 1000 行源代码组成时，一个典型的包可能包含大约 5000 到 50000 行源代码。将大型设计分解成易管理大小的内聚的包，极大地简化了开发过程。对开发者来说，了解一个包内的几打组件和它们的详细相互依赖[如图 7-4（a）]，比理解潜在存在于数百个未打包组件之间的任意依赖[如图 7-4（b）]要容易得多。

打包也使得系统体系结构设计者可以在比别的方式高得多的抽象级别上理解、讨论和开发一个大型系统的全面的体系结构。例如，体系结构设计者可以描绘一个包的职责，然后指定作为全部系统设计一部分的完整的包之间的可接受的依赖，而不必专注于单个的组件。真正的包依赖以后可以从源代码中提取，并与体系结构设计者的规范相比较。

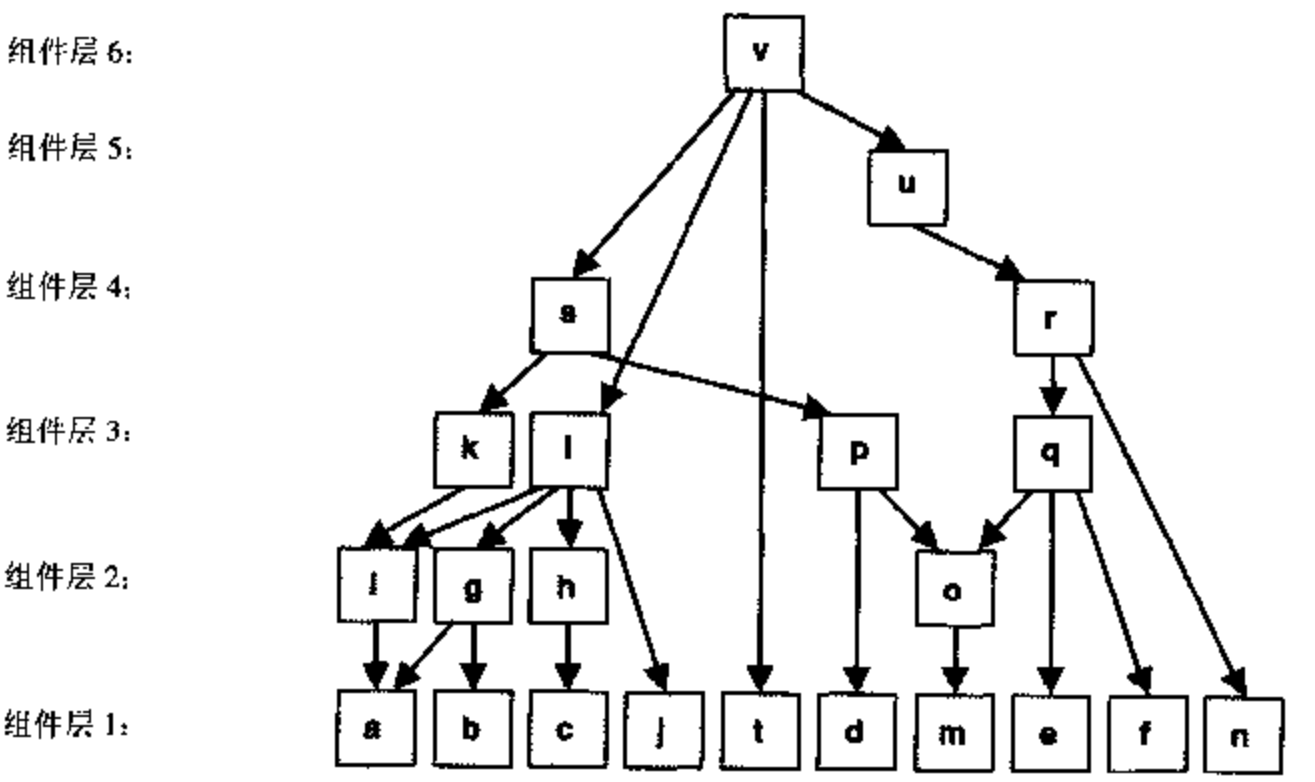
让所有的包处在目录结构的同一层使得它们很容易被开发人员访问。使用特殊用途工具（见附录 C），物理的包相互依赖关系可以从体系结构设计者的规范中提取（该规范存在于图 7-2 中的开发结构依赖关系文件内），并与规范相比较。注意，在测试一个包的新版本时，若要保证包级别上的层次化，则只有那些在依赖文件中命名的包才应该允许它们的输出头被包含，或者允许它们的库文件出现在连接命令中，这样才能保证包一级的层次化。

将组件划分成包不仅仅由一些规模或复杂性的任意阈值来决定。识别功能内聚的大小与包相当的单位是自顶向下设计的自然结果。由于带有单个组件内的类依赖，所以包内的组件依赖通常要比跨越包边界的依赖更多和更复杂。因为它们的更局部化的性质，包内组件之间的依赖关系的物理特性常常比它们的包之间的依赖关系的物理特性涉及更多的编译时耦合。事实上，一些定义在包内的组件，可能只是定义在同一包内的其他组件的绝缘的实现细节；这些实现组件的头可能不允许在包外访问。

打包也反映了开发组织。一般地，一个包由一个开发者拥有/创作。包内变化的影响可以被它的主人充分理解并一致有效地处理。跨越包边界的变化会影响其他开发人员，甚至可能影响整个系统。因此，将系统中高度耦合的部分作为单个包的一部分比较好。



（a）一个系统自顶向下分解成组件包



（b）同样的未打包的系统

图 7-4 一个系统的两个不同的视图

系统的组成部分作为一个单位被重用的可能程度也在组件的打包中扮演一定的角色。在图 7-1 的例子中，运行时数据库可能被一套工具使用，而三个并行的子系统则只被使用一次。即使运行时数据库与系统的这些其他部分相比很小，将这个低层次的子系统放在它自己的包里，以避免其可重用的功能被绑在任何其他较少使用的包上，也可能是有意义的。（在 5.3 节，图 5-24 和图 5-25，为了降级 enum E，介绍了一个类似的论点。）

总结：包是物理设计的一个聚集单位。就像一个组件一样，包是一个完成共同目的的相关功能的内聚单位。包既是体系结构设计者的抽象又是开发人员的分区。包的构成由若干因素决定，包括语义内聚度、物理依赖的性质、开发队伍的组织以及独立重用的潜能。

7.2 注册包前缀

正如在 2.3.5 节中所讨论的那样，在头文件的文件作用域中声明的逻辑实体只能是类、结构、联合和自由运算符。作这个限制是要减少名称冲突的机会。当只涉及一个开发人员时，通过遵循这个策略来避免名称冲突并不困难。名称空间（namespaces）（如在 7.2.2 节所讨论的）可以用来阻遏由完全独立开发工作的集成所产生的全局名称的无组织的增加。但是，当处理许多在大型的统一标准的系统上工作的开发人员（他们在多个地点工作）的问题时，就需要更结构化的方法了。

7.2.1 对前缀的需求

这里所采用的方法能确保惟一的全局类名称，它要求每个包都与一个惟一的、由 2~5 个字符组成的**注册**前缀相联系。当一个包第一次被创建时，其前缀在某个公司范围的权限（authority）或服务（service）上注册，这样其他包开发者就不会无意中重用它。头文件中的每一个在文件作用域中声明的结构都将被附加包前缀。实现这个组件的.c 和.h 文件也都要附加同样的前缀。通过对每个全局名称都附加这个注册的前缀，我们能够保证定义在不同包里的相似名称不至于冲突。

主要设计规则

为每个全局标识符都附加它的包前缀。

例如，一个几何图元包由许多独立可重用的组件构成。一个定义基本点的组件不被命名为 point，而是命名为 geom_point，在这里“geom”是与 geom 包相关的惟一的注册前缀。定义一个几何点的类也不叫做 Point，而是被称为 geom_Point[①]。

① 注意，为了区别类型名称和非类型名称（如在 2.7 节中所介绍的），我们已经选择了不把前缀当作标识符的一部分。一个同等的和同样有效的惯例是，改为将前缀的首字母大写（例如 Geom-point）。将 Point（而不是 Geom）的首字母大写只是强调了 geom-Point 是一个在 geom 包中的 Point **类型**。

每个定义在文件作用域中的标识符都必须加上一个注册的前缀，以确保避免跨越包边界的名称冲突。虽然在文件作用域中只允许有类、结构、联合和自由运算符，但在特殊情况下（例如，6.5.4 节的遵循 ANSI C 的接口）可能会对这条规则强加一个例外。如果因为某种原因要在一个头文件的文件作用域声明一个函数、变量、枚举类型或 typedef，那么我们仍然需要确保为它的每个文件作用域标识符附加适当的包前缀。这条独立的设计规则在图 7-5 中举例说明。

```
// geom_polygon.h                  // Filenames are always all lowercase.
#ifndef GEOM_POLYGON               // CPP macros are always all uppercase.
#define GEOM_POLYGON               // Hence, the prefix must be case insensitive.

enum geom_Color { geom_RED, geom_GREEN, geom_BLUE };
    // Proscribed global enumeration must still use package prefixes.

typedef short int geom_Int16;
    // Proscribed global typedefs must still use package prefixes.

class geom_Polygon {
    // Global class definitions are not a design rule violation.
};

int operator==(const geom_Polygon& left, const geom_Polygon& right);
    // Global operators are not a design rule violation.

geom_area();
    // Proscribed global functions must still use package prefixes.

double geom_scaleFactor;
    // Proscribed global variables must still use package prefixes.

#endif
```

图 7-5 甚至在文件作用域中被禁止的结构都需要前缀

在类作用域声明的标识符不必附加包前缀，因为封闭的类（已加了前缀）提供了防止冲突的天然屏障，并提供了相关功能的适当分组。类似地，有内部连接、完全在单个的.c 文件中声明和使用的标识符也不需要附加前缀。就是说，在一个.c 文件内指定的 typedef、枚举类型、静态变量或静态（或内联）自由函数的作用域被限制在一个编译单元中，因而不会与局部定义在另一个编译单元内的同样的短名称相冲突。静态类成员函数和非内联成员函数有外部连接，因此，在类名称附加包前缀是适当的，甚至当这个类本身是在一个.c 文件内完整地定义和使用时也是如此。否则我们就要冒这样的风险：这样一个隐藏的类将产生外部符号，该符号在连接时可能会与在某个.c 文件（该.c 文件属于某个其他包的组件）中隐藏的类的外

部符号产生冲突[1]。

主要设计规则

为每一个源文件名附加上它的包前缀。

由编译器产生的名称有时候需要调整为适合源文件本身的名称。在 CFRONT 中，文件名称被用作命名虚表和入口点（用于初始化和析构定义在文件作用域的用户自定义类型实例）的基础——它们都有外部连接。所以，如果要避免连接时冲突，系统中所有源文件都有一个惟一的名称是很重要的。包含 geom 包的所有.o 文件的库，其名称也应该以某种方式包含“geom”前缀（例如，在一个 Unix 系统中是 libgeom.a）。

对于许多系统来说，对文件名长度的严厉限制使得附加惟一的前缀很困难。如果限制是 8 位或更少字符，那么文件名可能变得相当隐晦。在某些系统上（例如 Unix），除了对可放在一个库档案文件中的.o 文件的名称长度有陈旧的约束外，文件名长度并不是一个问题。对应的.c 文件名可能需要限制在某一相对较小的长度内（在一些基于 Unix 的系统上是不超过 14 位字符）。在这个例子中，我们可以让.h 文件名相应较短来匹配.c 文件，或者提供某种外部交叉引用来允许较长的头文件名与较短的（缩写的）实现文件名相关联。在我的 Unix 系统上，我在开发过程中使用符号连接来获得这种映射。

7.2.2 名称空间

1993 年 7 月，ANSI/ISO 委员会采用了由 Bjarne Stroustrup 设计的名称空间结构来帮助解决同名的全局标识符之间的冲突[2]。例如：

```
namespace geom {
    class Point { /* ... */ };
    Point& operator==(const Point& left, const Point& right);
    class Polygon { /* ... */ };
    // ...
}
```

定义了一个名称空间 geom。在大括弧中声明的结构被放在它们自己的作用域中，所以不会和全局名称或在任何其他名称空间中声明的名称产生冲突。**using 指令（using directive）**起初提供来减轻转换负担，其目的是通过 **using 声明（using-declaration）**来使用显式的条件[3]。

---

① 注意，如果开发者能保证某隐藏类的连接的所有方面都是内部的，那么前缀对于该隐藏的类来说就不是严格必备的。用于命名有外部连接的类（它们对一个组件来说是私有的）的包前缀技术的一个通常有用的扩展，已在完全绝缘的类的上下文中介绍过了（6.4.2 节的末尾）。

② **stroustrup94**，17.1 节，400 页。

③ **stroustrup94**，17.4.2 节，408 页以及 17.4.5.3 节，414 页。

```
void mySpace::Class::f()
{
    geom::Point p(3,2);
    // ...
}
```

正如读者所看到的，名称空间和注册前缀都可以类似的方式避免一个公司内开发的类之间的名称冲突。但是，在其他情形下，它们不能彼此替代方法。

当处理由两个不同的供应商提供的C++应用程序库时，有几个潜在的问题。如附录B所描绘的那样，如果用来开发这些库的编译器是不兼容的，则你的运气不好。但是即使你可以让两个供应商提供兼容的库（体系结构、操作系统和编译器/连接器），却没有中央权限（central authority）可控制注册前缀；因此，全局定义的名称很可能冲突。名称空间结构的能力就在于此。

若将一个公司开发的所有库代码都放在单个的名称空间包装器内，则不可能确保只通过显式限制一定能解决前缀匹配及标识符匹配的问题（即使这种问题不太可能发生）。假设两个公司，SDL和SCI，都提供几何库软件。两个公司都决定创建一个称为geom的“惟一的”的包前缀。显然这些包内的一个或多个几何名称（例如，Point、Line、Polygon）存在着冲突的可能。

如果两个公司中有一个或两个有远见地将它们的代码放在单一的公司范围名称空间内，标识符名称冲突解决方案问题就会消失[①]。

图7-6展示了结合包前缀和名称空间来解决多个供应商之间的名称冲突的技术。即使SCI没有使用名称空间，我们仍然可以通过附加作用域标识符（::）指定真正的文件作用域来访问它们的geom_Point类。注意，SDL已经保护了自己，但是如果某个其他的供应商或它的一个客户并没有采用这些预防措施的话，SCI就会处在危险之中。

因为C++语言支持名称空间的任意嵌套[②]，所以我们可以选择用包名称空间代替包前缀来解决公司内部的包之间的名称冲突问题。例如：

```
void f
{
    SDL::geom_Point pt;                    // package prefix
    // ...
}
```

将被替换成

```
void f
{
    SDL::geom::Point pt;                   // package namespace
    // ...
}
```

① 如果有外部连接的编译器产生的符号被基于文件名而产生，我们可能还会有问题（如某些实现中的这种情况）。

② **stroustrup94**，17.4.5.4节，415~416页。

```
// sdl/geom_point.h
#ifndef INCLUDED_SDL_GEOM_POINT
#define INCLUDED_SDL_GEOM_POINT
namespace SDL {

class geom_Point {
    // ...
  public:
    geom_Point(int x, int y);
    geom_Point(const geom_Point& point);
    ~geom_Point();
    geom_Point& operator=(const geom_Point& point);
    void setX(int x);
    void setY(int y);
    int x() const;
    int y() const;
};

int operator==(const geom_Point& left, const geom_Point& right);
int operator!=(const geom_Point& left, const geom_Point& right);

}

#endif
```

```
// sci/geom_point.h
#ifndef INCLUDED_GEOM_POINT
#define INCLUDED_GEOM_POINT

class geom_Point { /* ... */ };

int operator==(const geom_Point& left,
               const geom_Point& right);
int operator!=(const geom_Point& left,
               const geom_Point& right);

#endif
```

```
// my_class.c
#include "my_class.h"
#include <sdl/geom_point.h>
#include <sci/geom_point.h>

void my_Class::f() {
    SDL::geom_Point p(1,2);
    ::geom_Point q(3,4);
    // ...
}
```

图 7-6　使用名称空间来解决供应商之间的名称冲突

但是，正如我们马上要介绍的，用包名称空间代替包前缀不是一个好建议。

在写这本书时（1996 年 5 月），C++语言的名称空间特性还不是普遍可用的。即使普遍可

用，它也不会影响对前缀的需求，因为前缀有着许多优势，并不仅仅可以避免名称冲突。一个包要满足内聚性，即将组件结合进包里。每个包都趋向于具有自己的特点。这种现象部分要归功于包的本质特征，也要归功于其作者独特的创造风格。通过标识一个组件或类属于一个特定的包，你立刻就提供了一个有助于了解包的更广泛用途的上下文[①]。在阅读依赖多个包中组件的应用程序代码时，包前缀将成为最吸引你眼球的东西。

原　则

前缀的主要用途是惟一地标识定义组件或类的物理包。

除了其语义的内聚性，包也是一个物理单位。包前缀的一个重要功能是标识在文件系统的什么地方可以找到一个给定的类或组件的定义。包前缀也可以使寻找一个特定包的“使用”更容易。包前缀还有许多其他的小用途。例如，如果你忘记连接一个特定的包，这个问题的性质立刻会变得很明显，如图 7-7 所示。

```
john@john: CC -g geom_iter.o  geom_util.o  geom_file.o  geom_print.o \
-o a.out -L/home/sys/lib -lxref  -lne  -llst  -lcrx
ld: Undefined symbol
    ___ct__10stdc_ErrorFCQ2_10stdc_Error8errorNumPCciT2
    stdc_AssocList::operator=(const stdc_AssocList&)
    stdc_AssocList::operator+=(const stdc_NameValue&)
    stdc_AssocList::setAssociation(const stdc_NameValue&)
    stdc_PIcontext::pop() const
    stdc_PIcontext::push() const
    operator==(const stdc_AssocList&,const stdc_AssocList&)
    stdc_AssocListIter::operator()() const
    stdc_Error::operator=(const stdc_Error&)
    stdc_PIcontext::~stdc_PIcontext()
    operator<<(ostream&,const stdc_Error&)
    stdc_AssocList::~stdc_AssocList()
    __vtbl__14stdc_AssocList
    stdc_Error::~stdc_Error()
Compilation failed
john@john:
```

图 7-7　因遗漏 stdc 包库导致的连接时错误

更重要的是，一个包，就像一个组件，代表着一个内聚的单位。在进行组件级设计时，包的逻辑设计和物理设计是紧密交错的。当讨论包（特别是它们的物理相互依赖关系）时，每个包同时具有的逻辑和物理性质是很重要的。

① 有关对全局名称空间进行分段的这个观点以及其他观点，见 **stroustrup94**，17.4.1 节，406 页；以及 17.4.5.5 节，416~417 页。

7.2.3 保持前缀完整性

前缀的用途是为组件或全局逻辑结构定义的物理位置提供层次结构的标识。对于有着内聚功能的设计良好的包来说，包前缀提供了语义及物理信息。只使用前缀来标识语义特性有违于它的主要目标：迫使具有类似前缀的内聚逻辑功能被打包在同一个物理库内。

原　则

理想状态是，包前缀不仅表示定义组件或类的物理库，还将暗示包内聚的逻辑特征和组织特征。

有时候可能有一种强烈的诱惑，要把逻辑相关的单位分散到多个物理库中，并给这些逻辑单位指定一个共同的包前缀。例如，一个给定的包（pub）可能提供了一组低层次、可重用的容器类型。定义在那里的每个组件和每个类型都会以前缀“pub_”开始。现在假设我们正在开发我们自己的应用程序包（xr2e），突然发现需要一个新的类型 Btree，这个类型碰巧和在 pub 包中发现的那些东西有着相似的特征（低层次、容器、可重用），我们应该怎么办呢？

我们或许会试图将这个组件命名为 pub_btree，并把它放在我们自己的库中，来反映它对 pub 包的逻辑关系。这种主张应予以限制。有着一个给定包前缀的所有组件驻留在一个单个的物理库中这一事实，对于理解和管理大型系统的组织来说太宝贵了，不能牺牲掉。

我们另外还有两种可行的选择——每个选择都有自己的长处：

（1）将组件命名为 xr2e_Btree 并放在我们自己的包中。

（2）将组件命名为 pub_Btree 并放在 pub 包中。

或许更容易的做法是，将类命名为 xr2e_Btree 并把它定义在一个组件中，这个组件是我们自己的包的一部分。局部地实现这个对象减少了它被重用的可能性——可能是好事也可能是坏事。通过将 Btree 定义在同一个包内，我们保住了所有权，因而不必费心去对它进行修改或增强（如果它能满足我们这样做的需要）。

重用的潜力并不总是一个显而易见的先验结果。我们可能认为不会有其他人需要一个 Btree 类型，所以只是为我们自己编写和保存它。如果其他人也这么想，并且 btree 组件结果确实是可重用的，我们最终可能会看到 Btree 的若干个冗余版本在我们的系统中冒出来。作为一条规则，如果我们在我们的系统中看到 btree 组件的三个或者更多的兼容版本，这个组件也许是非常好的重用候选项。此时，我们也许应该评估一下以下做法的效果：通过将 Btree 的单个的、统一的版本移到更公用的 pub 包（并将其前缀改为 pub_）来巩固我们的系统。

我们经常相信一个组件是可重用的，不料竟发现没有其他人需要它。将这样的重负放在高度可重用的包里，比耽搁了将潜在可重用组件放进 pub 包还要糟糕。让功能更公共些几乎总是比使它更不公共些要更容易做到。如果有疑问，最好推迟将一个组件加入一个更广泛使用的包里，直到能够证明确有正当理由才这么做。

如果我们确信在开始时一个组件绝对应该放在另一个包里，那么我们需要和负责维护那个包的开发人员商谈。如果你的建议是引人注目的，就像可能对 Btree 所作的建议那样，pub

的主人可能会同意为你编写 btree 组件，并把它放在 pub 包中供所有人使用。注意，现在你只是 pub 包的另一个用户，并且放弃了给 pub_btree 组件加入强制的特殊用户定制的权利。

进度限制也许会迫使你自己编写这个组件，并把它（连同它的增量式测试驱动程序）移交给 pub 包开发者。经过仔细检查之后，这个开发者将承担所有权，而你又会变得和其他任何没有特殊权利的客户一样。

在这里重要的权衡是，如果你冗余地创建了一个组件，那么你可以使它完全符合你的要求，你不必和其他的包开发者协商，也许你还能避免额外的包依赖。如果将这个组件移交给某个别的开发者，你就放弃了对它的责任和对它的功能的控制。如果这个组件本质上并不是可重用的，对你以及共享它的其他人的开销可能会超过任何好处。如果这个组件是很好的重用候选项，那么可能每个人都最希望它在一个单个的、语义上内聚的低层次包中定义和维护，在那里很容易发现和重用它。

虽然编译单元的概念在 C++语言中被适当定义了，但包的概念则完全是系统开发人员的工作，并且它的实现是依赖于特定的操作系统的。因为包不是语言的一部分，所以，创建这些在一个大型系统内的内聚的分区要由系统体系结构设计者和开发者来完成，几乎完全要靠他们自己。

像 browsers 这样的计算机辅助软件工程（CASE）工具，有助于揭示许多详细的特性以及类的大型集合之间的相互依赖。好的工具是设计过程的一个重要部分，但是它们不是将语义上内聚的功能经推敲划分成不同的物理单位的替代品。即便是最神奇的运行时环境，也很难及时传递由始终用它们的物理包前缀标记的、逻辑上内聚的全局结构所提供的语义信息。

编程人员刚开始可能很难接纳为所有全局标识符和文件的注册前缀的惯例。很快，大多数人不仅会适应它，而且在他们的日常开发工作中会开始依赖它。对于开发超大型项目而言，注册包前缀带来的好处绝对抵得上为此所付出的额外努力。

7.3 包层次化

打个比方，一个组件对于它的包就像一颗行星对于太阳系。每个组件描述了一个物理实体，而每个包描述了这些物理实体的一个内聚的聚集。在一个包内部的相邻组件之间的物理耦合，一般要比在不同包里的组件之间的耦合更严重。

7.3.1 使包层次化的重要性

正如你能回忆起来的，避免单个组件之间的循环依赖是一个重要的设计目标，因为这样有助于增量式地理解、测试和重用组件。

主要设计规则

避免包之间的循环依赖。

避免包之间的循环依赖由于以下原因也成为一条主要设计规则：

（1）开发。当连接整个系统或它的任意一部分时，必须指定包库调用的顺序以解析未定义符号。如果单个包内的组件之间的依赖封套（envelope）是非循环的，那么至少会有一个顺序能保证在连接过程中解析所有的符号。在 Unix 中，包之间的循环依赖意味着在连接命令中必须至少两次包含一个或多个库。这样做增加了连接一个子系统的必要时间，因为这样做会迫使一个或多个库被搜寻多次。更糟糕的是，对函数调用顺序的少量修改可能导致连接命令所需的库顺序改变，从而导致连接失败。于是，确定一种新的不会导致未定义符号的库连接顺序就成了并非微不足道的工作。

（2）推销。通常一个系统会有一个基本功能以及若干可自由选择的附加功能包，如图 7-8 所示。如果系统本身依赖于任何一个附加功能包，那么附加包就不是可自由选择的了，它必须和系统一起销售。如果这些附加包中有任何一个是多重依赖的，它们就不能作为真正独立的可选择的东西来推销和出售。

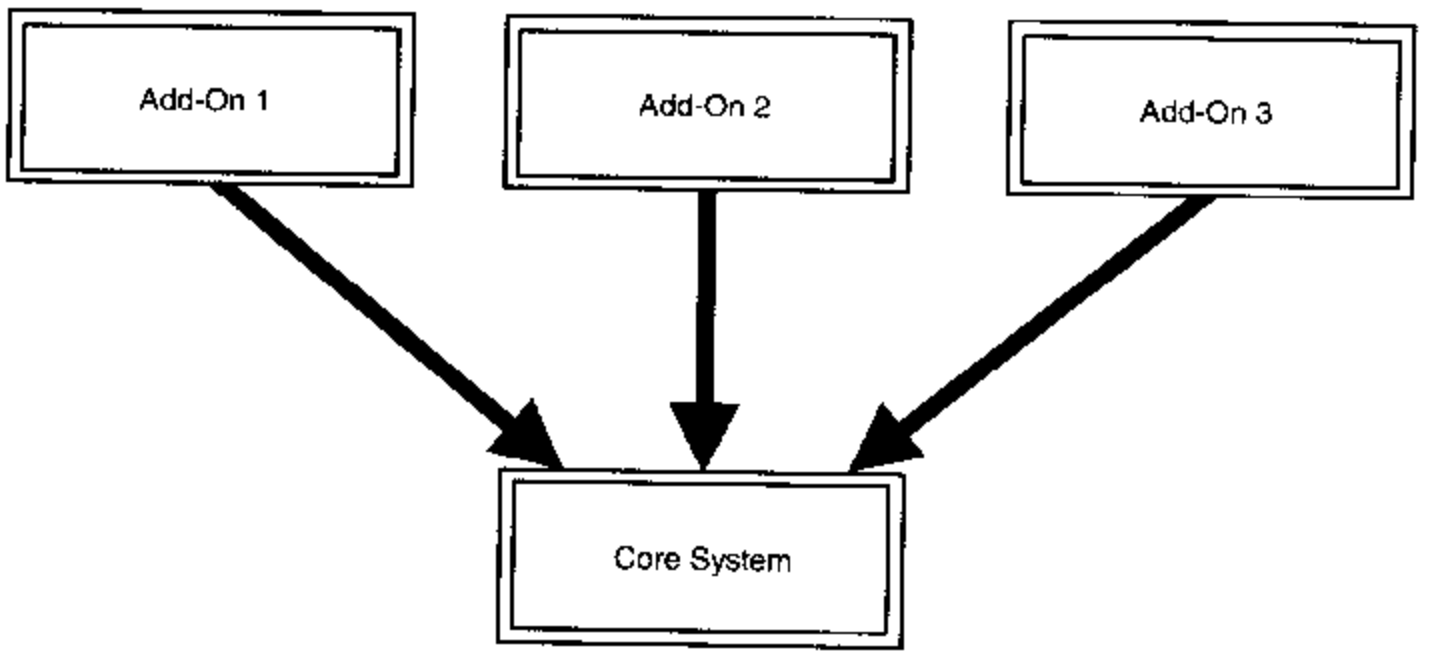


图 7-8　非循环包依赖提供灵活性

（3）易用性。即使推销不是一个问题，用户也不希望只为了使用基本系统的某个简单功能（或可能是独立的应用程序之一）而必须连接一个巨型库或若干大型库。使包相互依赖最小化，减少了必须被连接进一个应用程序的库的数量，反过来也有助于减少执行映像的最终大小（在主存和磁盘上都是如此）。

（4）产品。为了支持超大型系统的并行开发，有一个分阶段的发布过程是行之有效的（如在 7.6 节中所讨论的）。包的非循环层次结构被收集进更大型的体系结构单位**群（groups）**中，然后群层次化将这些**群**划分成**层（layers）**，层再被自下向上以层次化的顺序发布。允许包之间的循环依赖会妨碍形成群和进行分阶段发布的能力。

（5）可靠性。易测试性设计要求有一种方法可增量式地、分层次地测试大型系统。避免

系统宏观部分之间的循环依赖只是这种范例的一种自然结果。

虽然我们也许能足够平静地容忍在单个包内的一些组件之间由于疏忽、无知或特殊环境造成的循环依赖，但我们必须坚决地避免包之间的循环依赖。

7.3.2 包层次化技术

用来避免包之间循环依赖的技术和那些用来避免组件之间循环依赖的技术是相似的。基本目的是要保证，如果包 b 中的组件依赖于包 a 中的组件提供的服务，那么包 a 中的组件既不能直接也不能间接依赖于包 b 中的组件。

图 7-9 阐述了这样一种情形：两个包，r2d2 和 c3po，已经变得相互依赖了。这个问题和我们在图 5-3 中遇到的问题是完全类似的，在图 5-3 中，rectangle 和 window 中的逻辑结构导致了这两个组件之间的相互依赖。

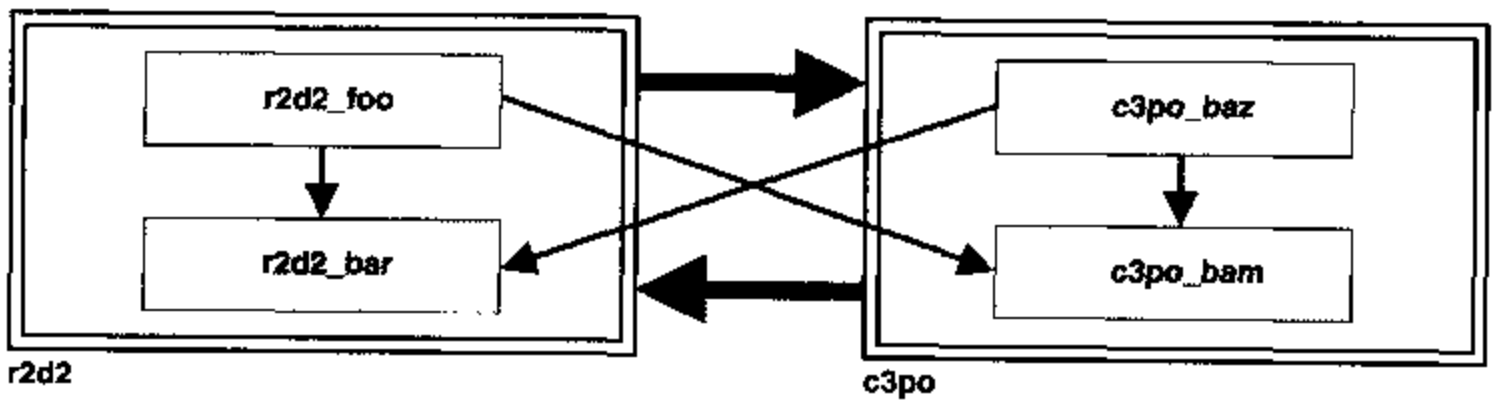


图 7-9 两个相互依赖的包

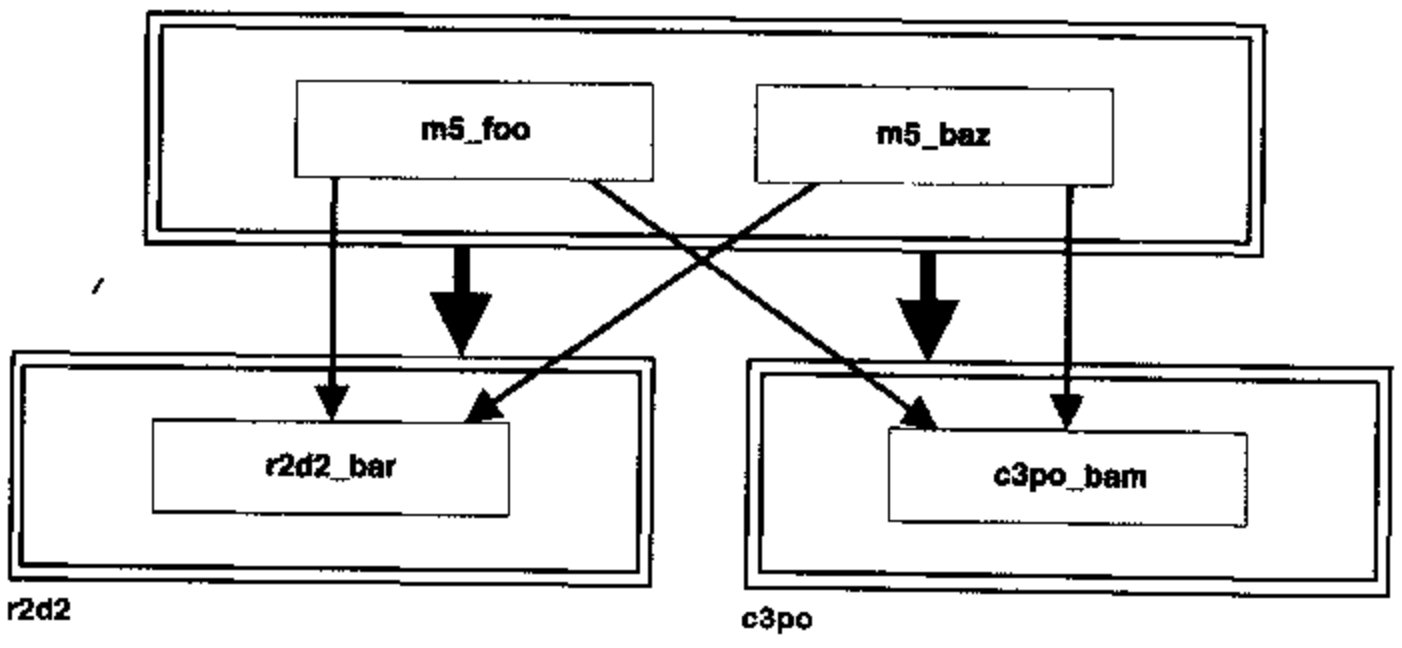


图 7-10 升级包之间的相互依赖

幸好，与在第 5.2 节给出的用于解开 rectangle 和 winow 组件依赖的方法相类似的补救措

施在这里也适用。例如，我们可以将这两个导致包层次上相互依赖的组件升级到更高的包层次，如图 7-10 所示。或者我们也可以决定使用图 5-36 所示的更通用的重打包技术来产生两个全新的包。

原　则

给被一个多包子系统的客户直接使用的组件子集指定一个单一包前缀并不是一定可能的。

包的用途是将紧密相关的组件集合联合成为可以被抽象引用和有效重用的模块化的物理实体。图 7-11 显示了其依赖构成了一个无向树的组件的层次结构。显然这些组件是可层次化的。但是，正如在 7.2.3 节所讨论的，有同样包前缀的所有组件都应该属于同一个物理库。所以，由这些前缀所隐含的包是不可层次化的，如图 7-12 所示。

当一个前缀被指定给一个概念展示包（conceptual presentation package）（就是包含可被一个多包子系统的客户直接使用的所有东西的包）时，由图 7-12 指出的问题可能会在实践中出现。如果这个展示包同时定义了协议类（本来就是非常低层次的）和包装器组件（本来就是非常高层次的），就不可能将单独的中间层次上实现包的组件隔开并维持一个可层次化的包层次结构。对这种常见问题的解决办法是，提供两个单独的包用于向客户展示。一个包驻留在包层次结构的底层，只包含定义协议类的组件；另一个包驻留在了系统的顶层，只定义包装器组件。

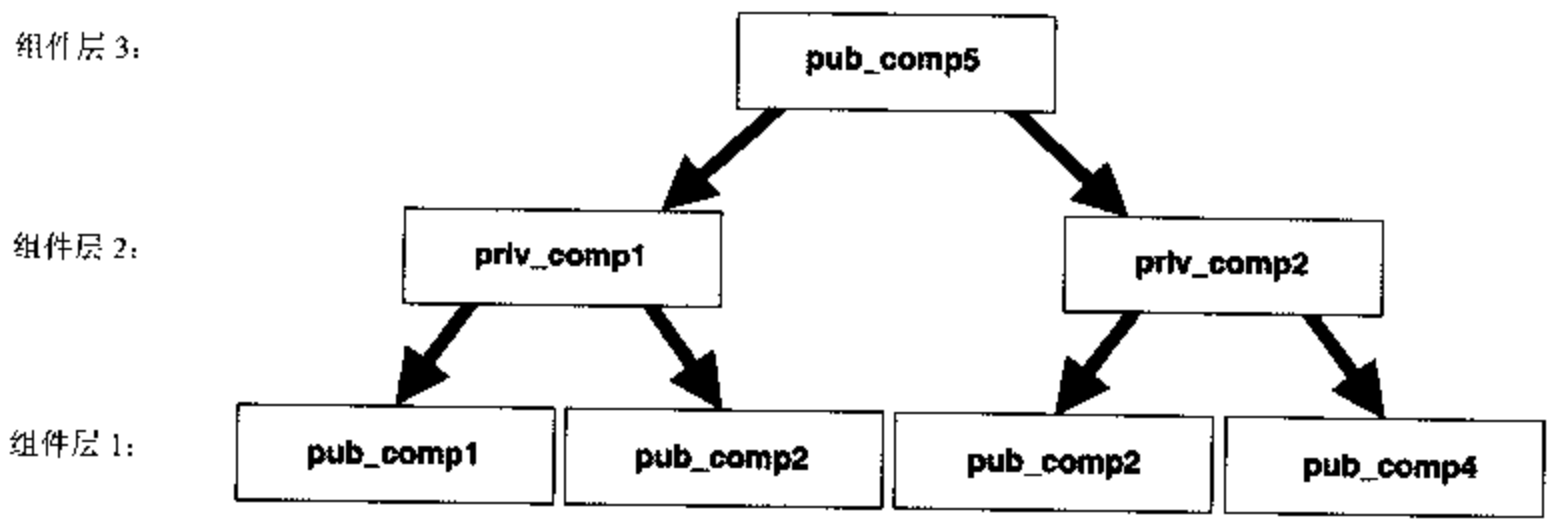


图 7-11　隐含的循环包依赖

7.3.3　划分一个系统

虽然确保包之间的可层次化性（levelizability）是必要的，但仅此还是不够。例如，图 7-13（a）阐释了一种自底向上的打包方法：我们只取出图 7-13（b）中的未打包的设计，并仔细地将它切成包，这些包对其他包的聚集依赖构成非循环图表。但是简单地将大量可层次化的

组件划分成一个另外的可层次化包的任意集合，这并没有顾及到设计的另一个重要方面——内聚性。要想有效，一个包应该由有着相关语义特性、紧密耦合的组件和逻辑实体组成，或由其他的应该打包在一起和在更高层次被抽象对待的组件和逻辑实体组成。

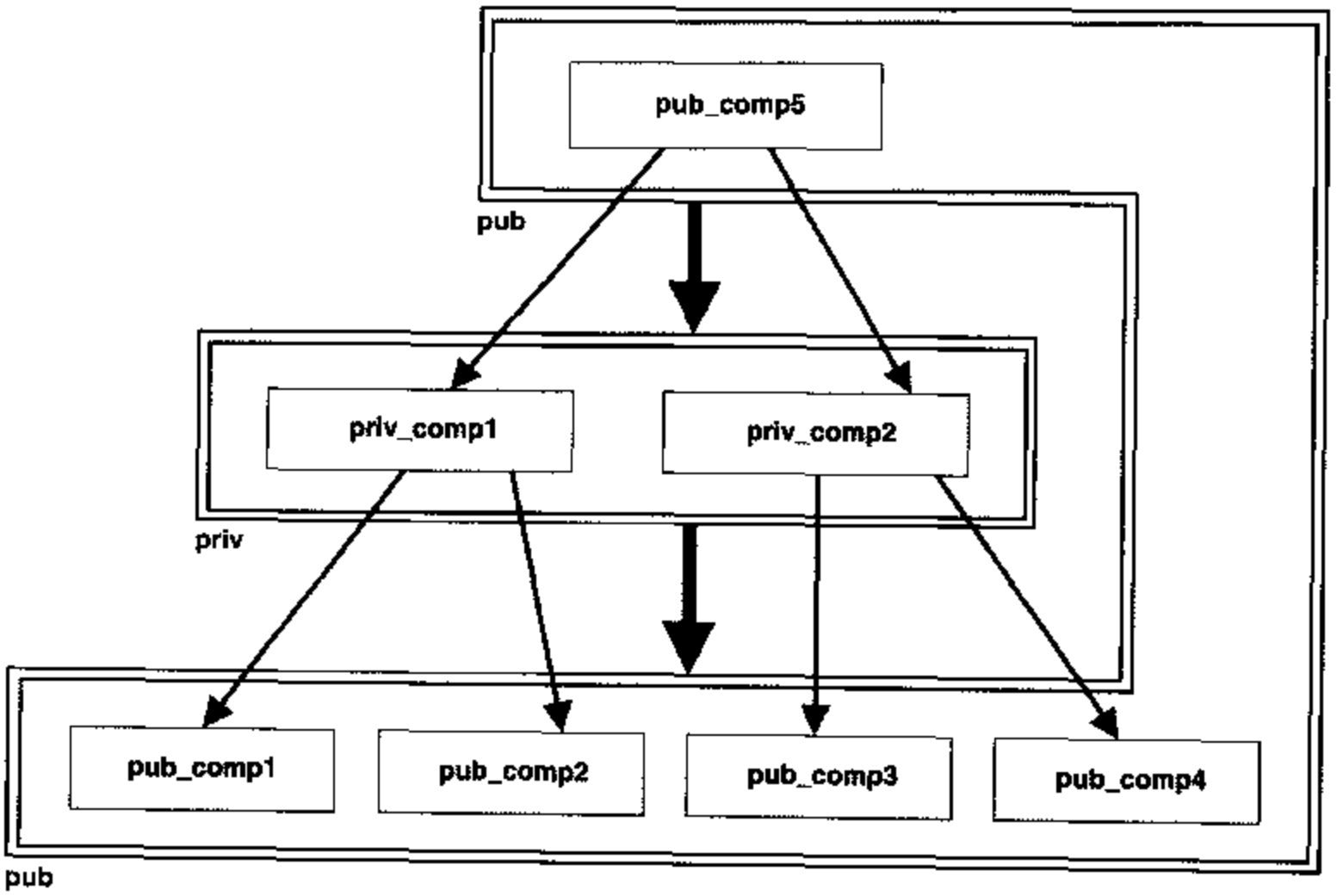


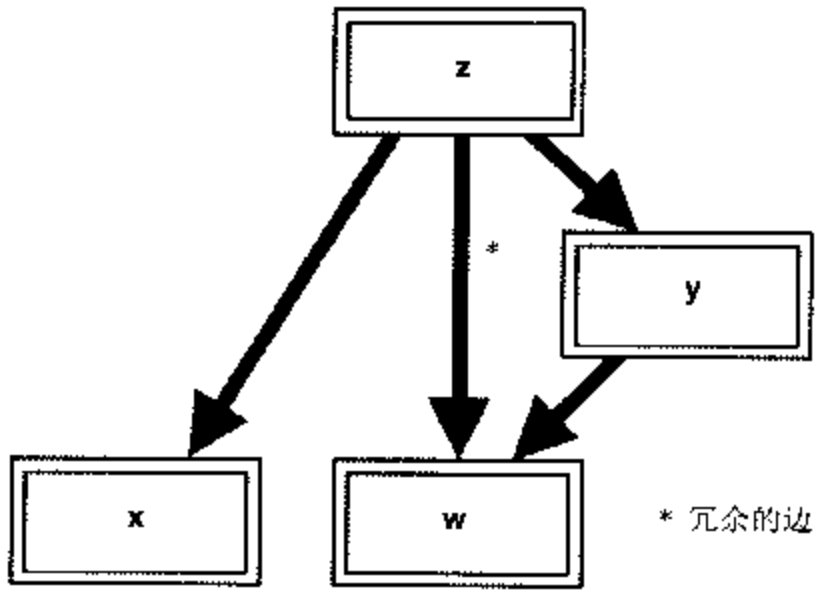
图 7-12 可层次化的组件层次结构：不可层次化的包层次结构

原 则

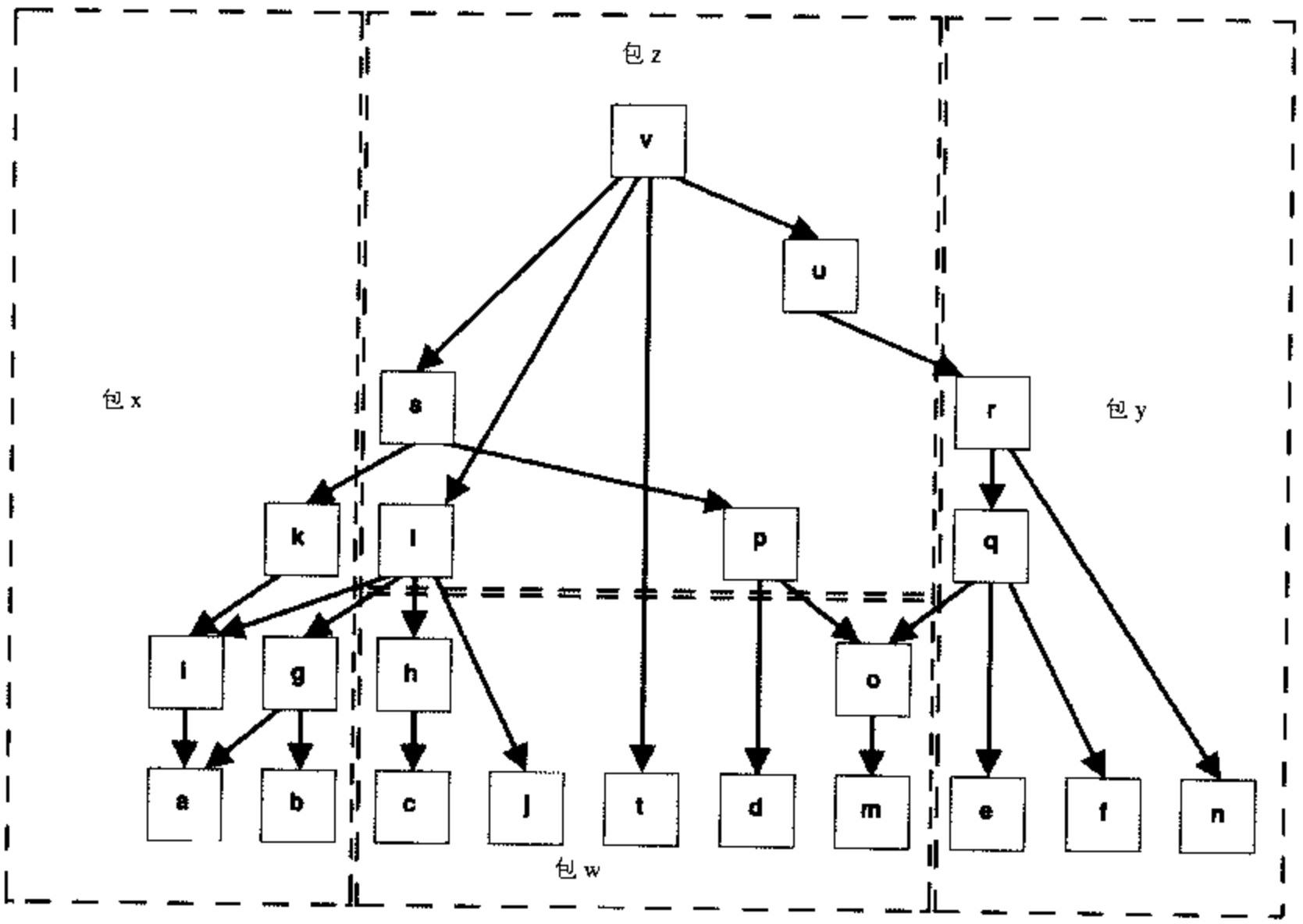
在把一个新组件加入到包中时，这个组件的逻辑特性和物理特性都应该考虑。

正如在 5.7 节关于子系统方面所作讨论，在将组件合并进包时，依赖也是一个应该考虑的因素。假设一个给定的包在特征上是轻量级的，不依赖于任何其他的包。进一步假设，加入一个单个的、逻辑上内聚的组件，会迫使这个包的客户与十个其他的包相连接。即使这个组件的逻辑内聚性对这个包来说是理想的，但额外依赖的影响也可能会压倒任何其他因素。在定义一个包的特性时，逻辑内聚性和组织内聚性都应该考虑。

在这个例子中，一个更好的解决方法是，为这个新组件创建一个单独的包，并附加类似的但也许不是完全相同的前缀，用以传达逻辑语义的类似特性，同时区别物理依赖隐含。通过将这个重量级的组件放在一个单独的包里，轻量级包的客户就不必承受由不必要的沉重依赖（对他们不需要的库的依赖）所带来的系统开销。



（a）抽象的包层次依赖图



（b）详细的包/组件依赖图

图 7-13　更没用的、物理上划分的系统（和图 7-4 相比）

7.3.4 多地点开发

因包间依赖而相关的开发团队的地理分布，将影响在开发人员之间分配包的所有权的方式。考虑在图 7-14 中描述的包的系统。假设我们公司有两个地理上分开的开发地点，N 和 S。我们应该如何在这两个不同地点间分配工作量呢？

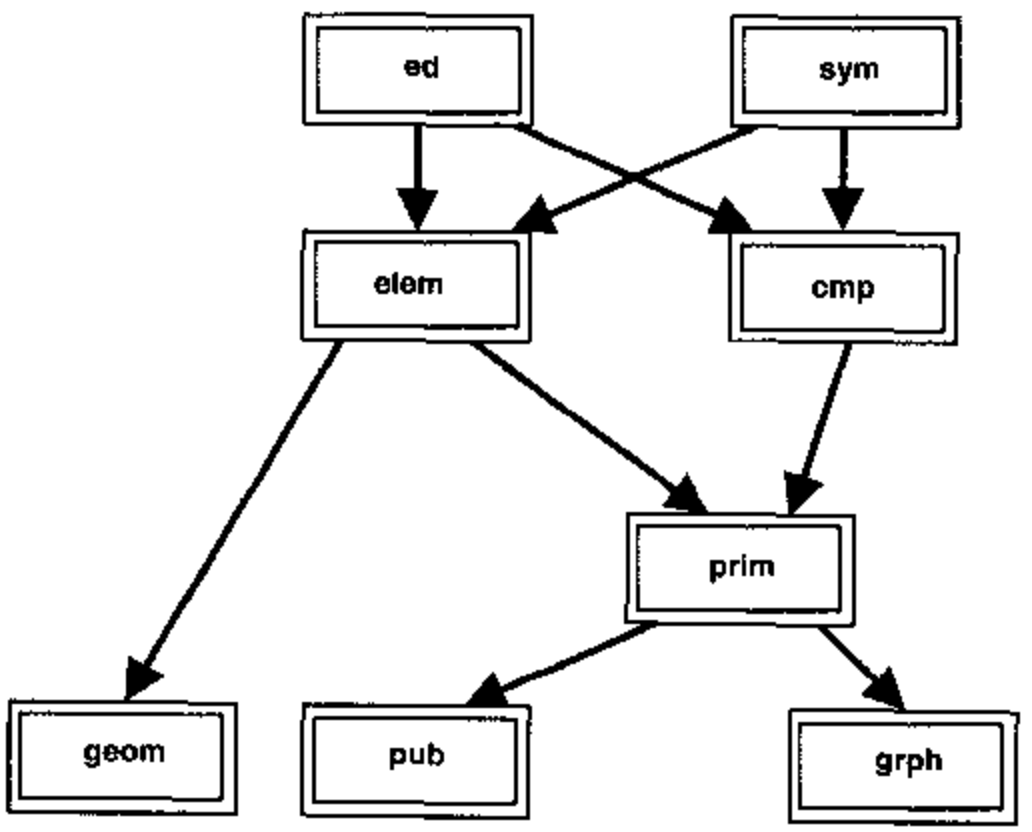
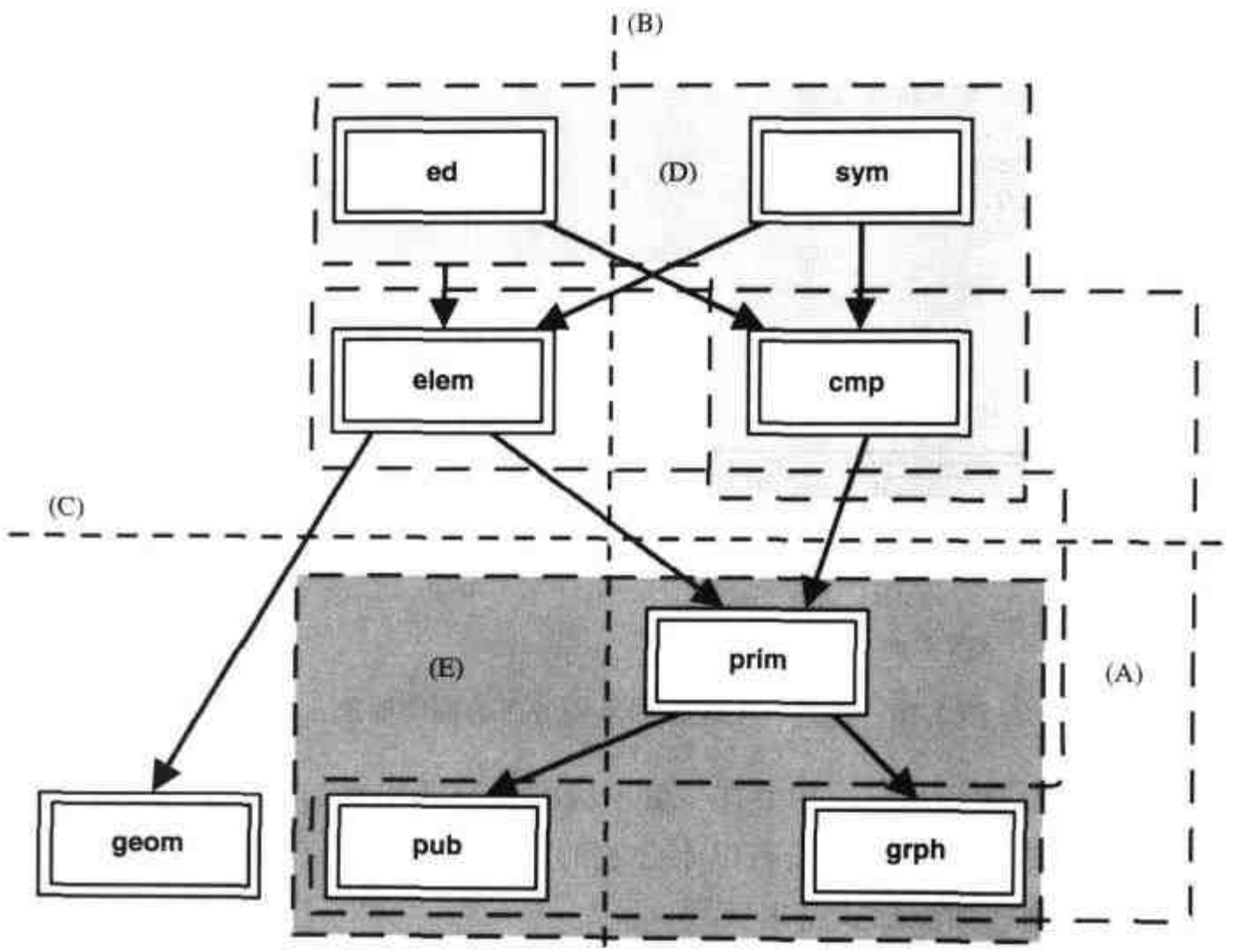


图 7-14 包的系统和它们的物理依赖

在逻辑上，为了减少因不同地点间的通讯造成的低效率，将不同地点的包之间的依赖最小化到任何可能的程度都是有意义的。考虑图 7-15 中建议的包开发分布。分布（A）是非常糟糕的，它有七个跨地点的直接包依赖。用一根垂直线分开图表的（B）展示了另一种不恰当的划分，这种划分存在五个直接的跨地点的依赖。用一根水平线分开图表的（C）可能是最佳的解决方案，只需承担三个远距离直接依赖的开销。如果在每个地点的包和/或可利用资源的复杂性不是均匀分布的话，（D）和（E）也可能是最佳解决方案。

标识包和描绘它们的相互依赖可以影响到较大型项目的成功与否。最小化包间依赖的开销，在整个设计过程中都应该是每个体系结构设计者最先要考虑的。最重要的是，如果系统的灵活性和可维护性能够得以保持的话，避免包之间循环依赖的昂贵开销是必要的。

总结：将一个系统划分成包的可层次化的集合，对于大型项目的成功是关键的。在第 5 章所讨论的获得组件的可层次化集合的大多数技术同样可以应用到包上。除了由远距离友元关系导致的耦合之外，使我们能够减少 CCD 的同样的推理也可以用来减少包间依赖的开销。无论何时我们能利用这些技术来减少包的相互依赖，我们就是在对整个系统的灵活性和可维护性进行重大的改进。



N	S
pub	geom
grph	prim
elem	ed
cmp	sym

(A) cost = 7

N	S
geom	grph
pub	prim
elem	cmp
ed	sym

(B) cost = 5

N	S
geom	elem
pub	cmp
grph	ed
prim	sym

(C) cost = 3

N	S
geom	cmp
pub	ed
grph	sym
prim	
elem	

(D) cost = 3

N	S
pub	geom
grph	elm
prim	cmp
	ed
	sym

(E) cost = 3

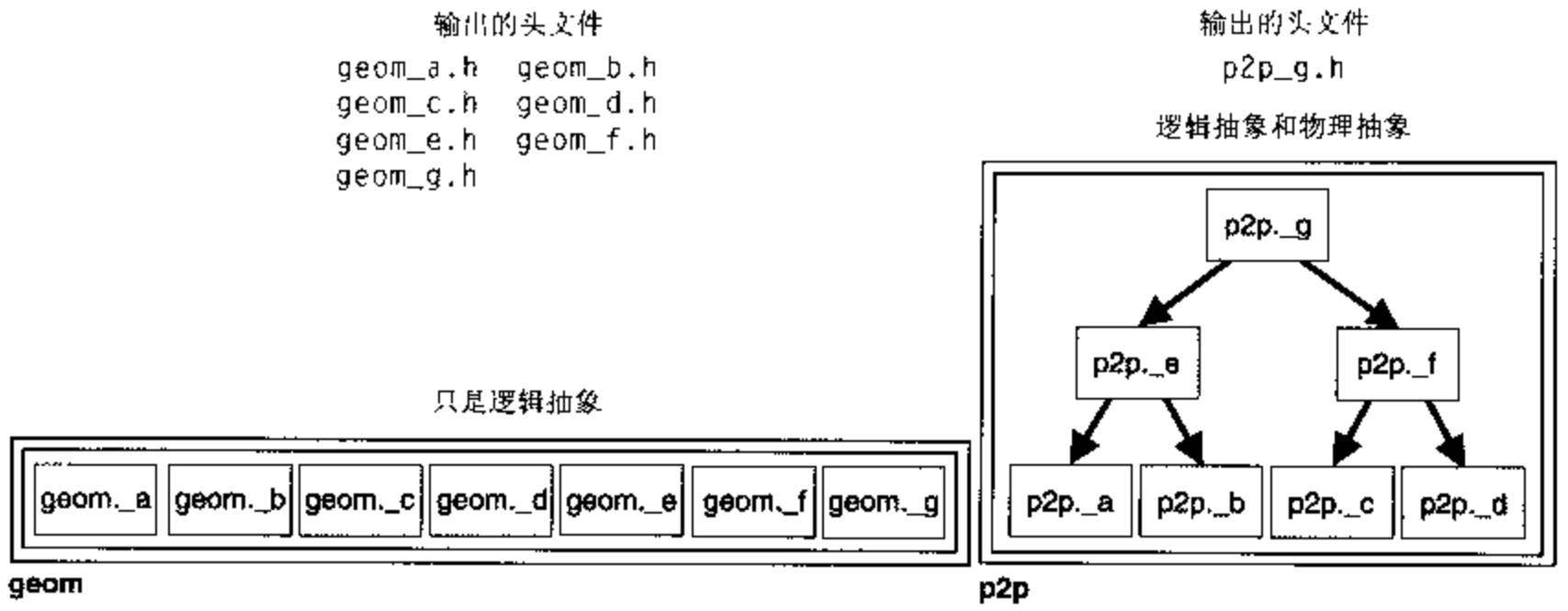
图 7-15　跨越地点的可能的包开发分配

7.4　包绝缘

包呈现了一种比组件更高层次的抽象。对于有着水平依赖结构的包来说，例如 geom（见 4.13 节），我们必须输出大部分独立的组件头文件，以使包的功能可以为客户所用[见图 7-16（a）]。即使将这些物理上独立的组件放在单个的包内不能隐藏任何额外的细节，我们仍然可以受益于抽象地称这些组件的聚集为 geom 的能力——这是一种不应该被低估的好处。

原　则

最小化输出头文件的数量和大小，可以提高可用性。



（a）水平的 geom 包　　（b）树形的 p2p 包

图 7-16　包是一个逻辑抽象和潜在的物理抽象

在树型包的例子中（例如 p2p，其中有少量绝缘的包装器组件），我们不仅可以获得概念抽象，而且可以获得物理抽象。由于没有以不必要的输出头文件的形式暴露过多的信息[如图 7-16（b）所示]，这个物理抽象形式得以实现。

当有一个好的组件接口时，我们在包的接口中暴露的细节越少，包开发者就越容易维护和调整它的实现。最小化暴露给客户的物理接口的大小也能够提高可用性。虽然一个水平包的接口区域天生较大，但这种需要和树形包的情况不同。

实现一个复杂的、应用特定的子系统的包，例如 p2p，一般会展示大量的功能。该子系统的实现可能会跨越许多组件。为了让客户使用这个包，组件的一个非空子集必须输出它们的头文件（即允许定义在这个包外的组件访问）。虽然包开发者和测试工程师总是可以访问所有的头，但包的一般客户不需要一定接触那些封装了使用的头。

对于定义在一个给定包中的一个特定组件，若以下问题的答案都是“是”，则意味着该组件的头必须输出：

（1）为了使用这个包整体提供的功能的任何一部分，这个包的客户都必须访问这个组件吗？

（2）这个包的任何其他输出的组件都不能使其客户与这个组件定义绝缘吗？

（3）其他包需要访问这个组件吗（例如在它们自己的实现中独立重用该组件的功能）？

考虑一下像 p2p 这样的包，它被分层实现，完全只通过包装器组件的一个小集合（在这个例子中是一个包装器组件）的接口来展示它的公共功能。这些包装器组件必须被输出到全局包含目录（见图 7-2），以便于外部客户使用这个包。但是，也许没有必要输出其余组件的头文件。

注意，我们不是在建议为了封装细节而在这里抑制头文件，而是在建议一种减少客户为了使用我们的包而必须费力通过的混乱的方法。我们是否输出实现组件的头文件取决于它们对于目标来说是否是需要（或有用）的，而不是为了创建属于这个包库的.o 文件。

如果一个包装器组件是封装的，但不是绝缘的（见 6.4.3 节），客户的编译器为了编译包装器接口也许有必要先看到一个或多个它的实现组件的定义。如果是这样，你就要被迫输出实现头，你的客户在编译时将依赖它们，并且你对它们进行修改的灵活性也将受到限制。

最后，在实现我们的包的过程中，我们也许偶然地创建了一个或多个这样的实现组件：其他开发者在实现自己的包时发现它们是有用的。在这种情形下，我们可能慷慨地为这些组件发布头文件。这样做使重用可以进行，但也会导致额外的包间耦合。这种耦合可能对我们维护自己的包的能力会有潜在的不利影响，并且可能会引进新的未经系统体系结构设计者允许的包级依赖。这样的额外包级依赖会进一步抑制整个系统的可层次化性（levelizability）。

如果一个组件头没有输出，那么我们的客户仍然与它是完全绝缘的。我们可能感觉很自由，对它可以想怎么改就怎么改。一旦一个头文件输出了，我们对它的接口进行的修改会潜在地影响许多其他想重用其功能的人。即使我们保持了这个功能，对一个输出组件的头文件进行的任何修改都将令人讨厌地强迫包含该头文件的客户重编译。这个例子还说明了另一种情形：重用并不一定是好事。

在实践中，有可能是一些低层次的（水平的）包输出了相当大数量的、逻辑上相关和可能被广泛使用的组件头。然后大部分其余的包会实现复杂的、在普通的低层次类型上操作的功能。理想状态下，这些较高层次的包会以绝缘包装器组件头的形式输出相对较小的高层次接口。

7.5 包群（package groups）

在超大型系统中（包含几十万行 C++代码），甚至一个不是在足够高抽象层次的包，在讨论整个系统的体系结构时也是有用的。在自顶向下设计过程中，体系结构设计者将确定系统的主要部分。每个主要子系统都将由一个开发组来实现；每个子系统将由包的一个内聚集合构成，该集合被称为**群（group）**。

> **定义：包群（package group）就是一个包的集合，它被组织成一个物理上内聚的单位。**

正如相关的组件被收集进包一样，相关的包被收集进群。一个单独的包最初由一个开发者拥有和维护，但一个包群通常由负责其实现的开发组的项目管理者（或首席工程师）拥有。

适用于单个包的组成和它们之间的相互依赖（例如逻辑内聚性和避免循环依赖）的原理同样可应用于作为整体的包群。像包一样，群应该带有定义良好的体系结构意义，用以控制

什么应该（和什么不应该）属于那个群。例如，如果一个群被命名为“核心功能”，那么我们应该阻止将不符合这个标签的包放进这个群中。

> **定义**：如果包群 g 中的一个或多个包 **DependsOn**（依赖）另一个包群 h 中的一个或多个包，则称包群 g **DependsOn** 包群 h。

考虑图 7-17 所示的大型系统。虽然这个系统在完成时将由大约 40 个包（50 万行）组成，它的功能却自然地划分为五个垂直排列的包群。每一个群都由若干个包组成。不仅这些包是独立可层次化的，而且全部群的如上述定义的依赖也是非循环的。也就是说，较高层次的群包含着的包依赖于较低层次的群中的包，但反之则不然。

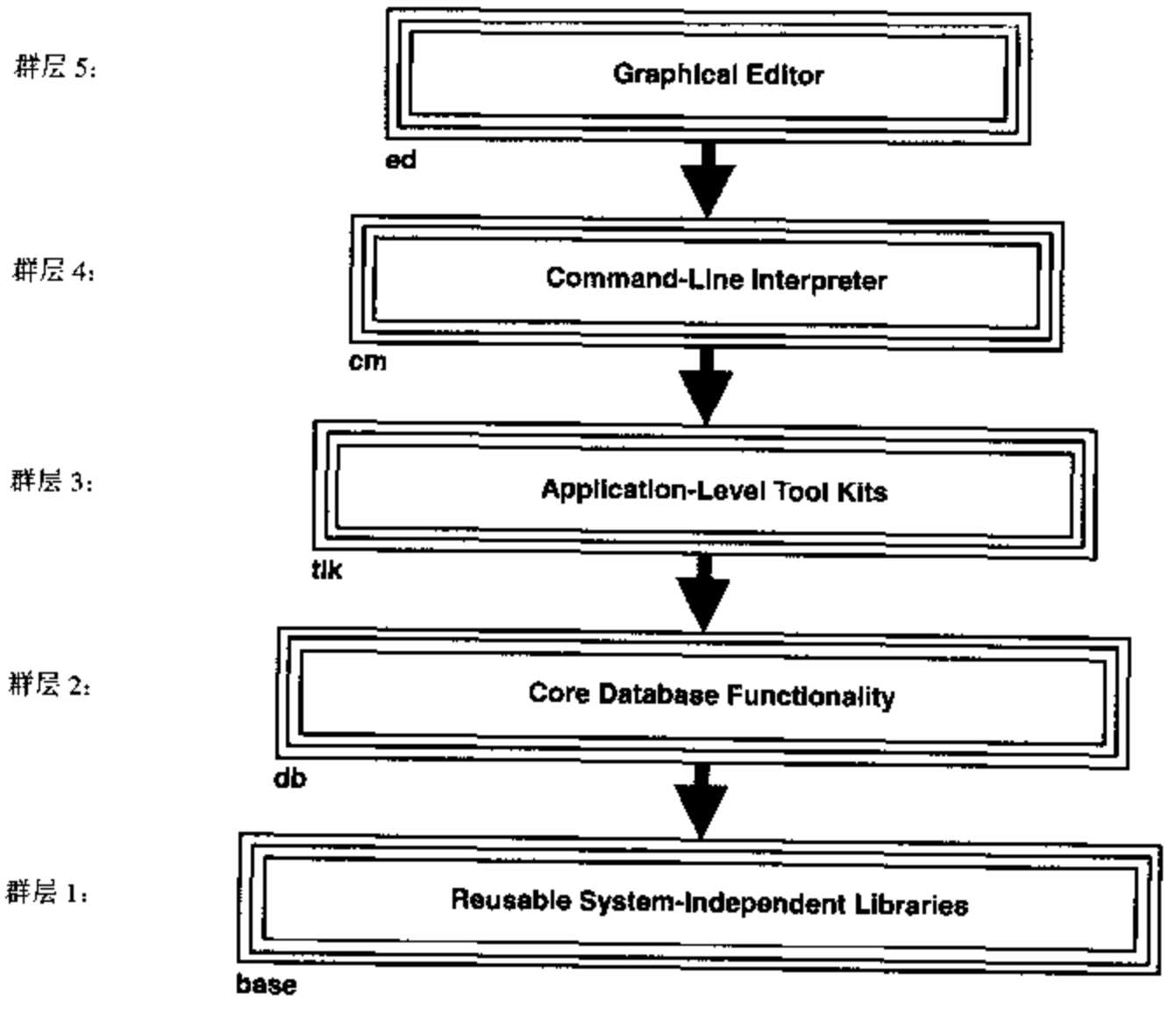


图 7-17 一个大型系统的体系结构

有很好的理由需要将独立的包库合并成单个的大型群库。其中的许多理由和要把组件的.o 文件合并成单个包库的原因相似。考虑图 7-18 所示的核心数据库群的内部包级组织。在这个体系结构中，有几个包被用在核心数据库功能的实现中。

在核心数据库群的最底层，dbt 包代表了一个用于整个群及其客户的类型和协议的水平集

合。第二层是一个有五个独立的实现包的集合。包 dbi 提供了包装器组件集合，向在较高层次的群的客户提供实现包的复合功能。

通常，在一个中间层实现包（例如，dba）中直接为一个子系统提供一个包装器组件是可能的，因而从那个包只需暴露一个单个组件头给群的其余部分。但是，正如在这个特殊实例中遇到的情形，有时候有必要将封装（和绝缘）升级到更高的层次——在这个例子中，是升级到群中的一个更高层次的包（例如，dbi）。

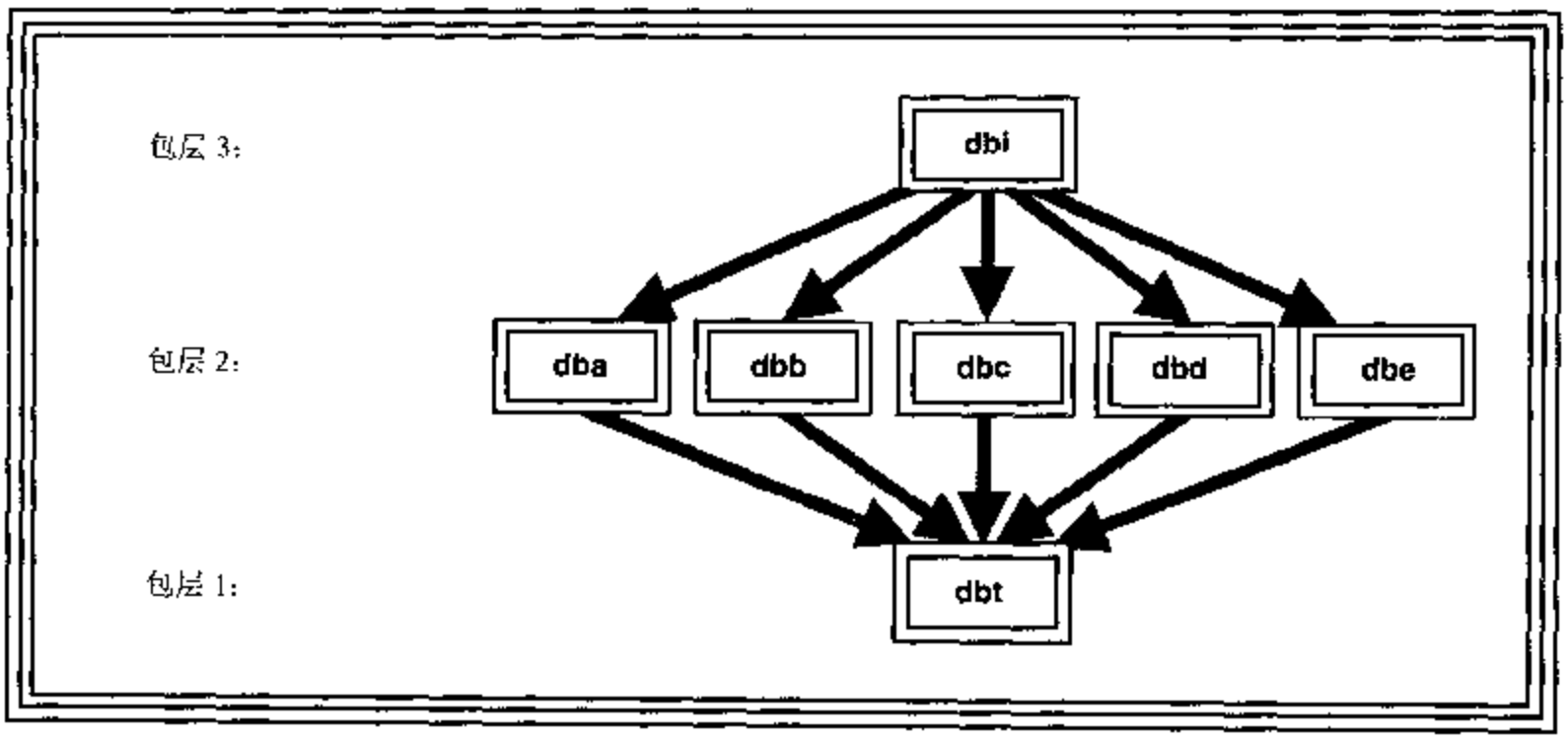


图 7-18 核心数据库群的包组织

除了定义在 dbt 中的低层次类型和协议，核心数据库群的整个功能都可以通过只在 dbi 中提供的包装器组件来访问。因为 dbi 是 dba、dbb、dbc、dbd 和 dbe 的包装器组件的一个封装和绝缘的包，所以没有强制性的理由要求给这个群的客户提供在这些实现包中定义的组件的头。一旦我们已经建立了 dbi 包库，将这些头输出到更高层次的群只会搅乱全局包含目录。请再次注意，暴露这些头不是一个封装的问题，而是一个绝缘和抽象问题。

建好数据库群之后，我们将只允许群的客户访问定义在 dbi 和 dbt 包中的头的子集。为了方便客户，我们将把所有单独的包库合并成单个的群库文件（带有相关的前缀 db）[①]，并且让该文件可公共访问。

对于核心数据库的客户来说，我们现在似乎已经将数据库实现为一个单个的包 db，它有两个相关的前缀 dbi 和 dbt。现在也许有一种诱惑想将 dbt 和 dbi 都改为更简单的 db；但这将是一个错误。在构成核心数据库群的包集合内，我们可能会看到差不多几十万行代码。某些

① 注意，这个名称（db）太需要被注册了（即可能注册过多），因而不能避免在其他群和包库名称之间的冲突。

人会认为大型系统应该这样。但是，如果我们改变了这些组件的前缀名称，那么我们就放弃了我们的系统的一个重要维护特性——前缀标识了在其中可以找到源代码的包，此外，我们还将失去防止这两个包之间的名称空间冲突的保护。

如果我们对这些问题的解决方案是将这两个包合并成一个单个的低层次包，我们就放弃了包层次化以及分层次地开发和测试系统的合理能力。我们又回到了图 7-12 中描述的问题。从一个纯粹实践的角度来看，我们必须牢记，在取悦客户（或我们自己）的审美观的努力中不要忘记可维护性。

原　则

在一个包群内将协议降级和将包装器升级，有助于避免输出（展示）包和非输出（实现）包之间的循环依赖。

回顾一下，注意协议是最底层包（dbt）的一部分，而不是展示包（dbi）的一部分。升级包装器和降级协议是一种通用和有效的技术，有助于避免一个群内的公共和私有包之间的循环依赖。

即使大部分客户不会关心内部划分，低层次包分区仍被用于许多有用的目的。例如，在开发过程中，不可避免地会出现程序错误。于是，和已经编译过的、包含可调试符号的单独包的版本进行连接可能是有用的。对于超大型系统，试图连接许多包并使用可调试版本来调试它们，会产生巨大的可执行程序，并使整个进程非常缓慢。单是保存一个可执行程序（在这个执行程序中，群中的每个组件都与调试选择一起进行了编译）所需的磁盘空间的数量就会造成显著的开发负担。用于在运行时检测低层次代码错误的高效商用工具[①]，可能会产生差不多三倍于正常大小的可执行程序，其运行速度要慢一个数量级。和这样大型的、特殊用途的群库连接只有两个选择——全部或决不——通常是不可行的。

幸好，对于群内哪个包很可能引起问题，工作在一个包群内部或直接工作在包群之上的大部分开发者可能会有好建议。这些开发者知道如何调整他们的连接命令来只链入适当的特殊用途包库，而不会对使其余包库的访问受到影响。提供一种能力，使我们能够从一个群中选择特别建立的单个包库，这样做有助于扩展能够用一套给定的工具在一个给定的硬件平台上开发的系统封套。

包聚集的大小和结构是没有限制的。在图 7-7 的例子中，这些包群采用了垂直排列形式。正如我们在下一节中将要介绍的，这种群的垂直排列略微简化了内部发布过程。在更大型（即超过一百万行源代码）的系统中，群可能形成一种树状或齿轮状（DAG-like）结构（见图 7-19）——也许反映了开发工作的工程管理结构。当然，在一个实际的设计中，群依赖可能不会像图中显示的那样规则。

① 一个特别有效的工具的例子是 Purify，由 Pure 软件公司设计。

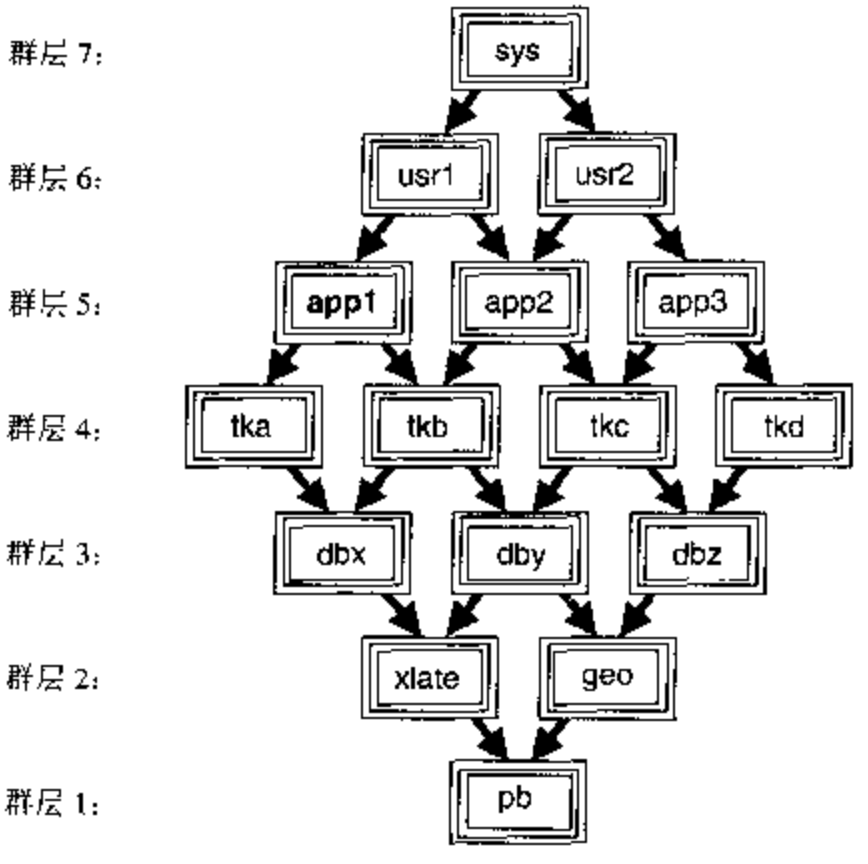


图 7-19 假设的有着齿轮状群依赖的超大型系统

简而言之，包群和组件包是类似的。群应该由逻辑上内聚或其他有理由组成单个内聚物理单位的包构成。和包一样，一个群定义的目的应该决定它的内容；不切题的东西不应该成为群的一部分。当然，包群之间的依赖应该形成一幅有向的非循环图。虽然一个包的大小适合于一个开发者拥有和实现，一个群却更有可能被一个项目经理拥有和由一个开发组实现。从客户的角度来看，群看起来就像一个单个的、有着紧密相关前缀集合的巨型包；但是，在所有情况下，每个群中的每个独立的包的完整性和惟一性都应该保持。在开发过程中，可能需要访问单个的特殊用途的包级库。

7.6 发布过程

作为并行工作在一个大型项目上的许多个开发者之一，可能很难确定为什么你的回归测试是失败的——是你刚刚对这个包所作的修改还是对某个较低级包所作的修改导致的？在一个可能出现自然变化的环境中开发软件，甚至会影响小型项目的生产率，而对于大多数更大型的系统来说可能根本就行不通。

内部发布是任何大型开发系统的一个必备部分。包群通常是发布的最小功能单位。每隔一段固定的预定时间间隔，一个包群（例如上一节中的核心数据库群 db）的代码就会被冻结[1]，并开始建立一个稳定的内部发布的过程。

① 自由升级到软件的这个版本的特权被暂停。

> **定义**：一个**层（layer）**对应于在系统的一个给定层次上的所有包群。

发布一个群的过程是以一种有秩序的、自下向上的方式来完成的，由群中的包的层次化来控制。群的最低层的包在隔离环境中建立和测试。一旦这些包通过了它们的独立的组件级的分层次回归测试，第2层的包就可以建立和测试了（只和第1层的包连接）。重建一个系统的过程和一个包内的单个组件的开发和测试的方式极为相似，但是规模更大。

包群的层次化在发布过程中有着特殊意义。在系统的每一层次上的所有的群被统称为一个**层（layer）**。对于有着垂直排列群的系统来说（见图7-17），每个层都仅由一个单个的群构成。对于有着更复杂群排列的更大型系统来说（见图7-19），一个给定的层可能由若干个群构成。为了保证整个系统的一致性，很重要的一点是，在一个给定的群g的代码被冻结以及g被发布之前，g所依赖的所有群要先被发布。例如，图7-19中的群dby在第3层，要等到xlate群也已经被发布了，dby群才能更新它对geo群的新版本的依赖。与此相对照，dbz群只依赖于geo群，因而开始更新过程并不需要等待xlate群的发布。

按照定义，在一个给定层次上的所有群都是彼此独立的。每一个群的发布过程都可以独立出现。虽然不是所有的更高层次的群都会依赖低层次的所有群，但在发布过程中追踪单个群的依赖可能是极有成效的。只要坚持让一个给定层次上的所有群都在更高层次的群的发布过程开始之前发布，就可以保证整个系统的一致性，同时我们也可以简化发布过程。

当这一层上的所有群的发布过程都已完成时，就可宣布新的包群的可用性了。工作在更高层的开发者继续使用较低层次上的前一个版本，直到它们到达了一个方便的终止点。在最后一次重新运行了他们自己的回归测试之后，这些开发者现在可以——在他们有空的时候——调整他们涉及较低层次软件的更新版本的环境。

此时开发者可能必须对他们自己的代码进行修改，以适应自上一版本以来对较低层次包群所作的任何接口修改——一个有时被称为**移植（porting）**的过程[1]。显然，如果计划得好，这样的修改可以最小化。在进行了一些小调整之后，开发者应该能够重新运行他们的回归测试程序，来验证那些对较低层次软件所作的修改并没有改变所需功能的特性。这些开发者现在可以使用新的稳定的软件版本重新开始开发。在某一时刻，这些客户将依次冻结他们的代码并经历一个类似的发布过程。

注意如何避免强迫直接前一层的客户在一个新版本发布时立即作出响应。经验表明，在连续的层的发布之间提供一些空隙，是管理大型系统的内部发布的一种有效的方式。

7.6.1 版本（release）结构

图7-20显示了一种为图7-18中所展示的系统组织开发层次结构的方法。这种开发目录结

---

[1] 术语porting（移植）指将一个软件系统移到一个新的平台。这个新平台可能采用新的硬件形式、一个新的操作系统或系统本身的较低层次的一个新版本。

构支持多个版本，并支持包之间共享头文件的概念（这些包没有被输出到群外）。在目录结构的根部有五个群目录，对应于图 7-17 所示系统的五个群；每个群都有一个子目录结构，类似于这里显示的核心数据库群 db 的子目录结构。在 db 目录之下是保存这个群过去的若干平行版本结构的子目录；图 7-20 所示的 db 群的这个版本是版本 1.6.3。

在群的版本目录下面是四个目录和一个文件。目录分别为 dependencies、source、include 和 lib，文件为 exported。目录 dependencies（依赖）指出这个群依赖的其他群的名称和发布版本。在一个基于 Unix 的系统中，每一个依赖都可能表示为一个符号连接，它向后引用较低层次群的特定版本，以构造这个群。提供这些引用允许客户的 include 和 link 目录在它们的一个指针从一个群的老版本更新到新版本时保持相关。

source（源）子目录的组织方式和图 7-2 中所示的简单得多的包开发结构的组织方式是一样的。如图 7-20 所示，群中每个包的所有源都存在于一个对应于它的包前缀的目录之下，包前缀使开发者很容易找到定义在群中的包的位置。但是，要找到定义在群之外的包的位置现在变得更困难了。这个问题可以这样来解决：让一个群中的包扩展一个共有的群前缀（例如 dba 或 dbb），或者更不理想些，在全局包注册中标识群的位置。“给前缀加前缀”存在着另一个问题——那就是，怎样让任何人都知道，对于某个新的不在群 db 中的包，dbq 并不是一个合法的前缀？

配置控制必须是开发过程的一个组成部分。像 SCCS 和 RCS 这样的系统需要集成进开发环境中。甚至更强大的系统也可以从市场上买到，但是对这些工具的使用的详细讨论已经超出了本书的范围。

为了支持这个群的输出与局部头的概念，include（包含）目录现在更复杂了。在 include 之下的 local 子目录类似于图 7-2 中的全局包含域（global include area），但是只能从 db 群内部访问。这个局部目录包含支持该群内包间通讯所必需的头文件。直接定义在群版本之下的 exported 文件的内容，标识了那些头对于群之外的客户是可访问的独立组件或完整的包。在发布过程中，这些头被直接拷贝进该群的 included 目录[①]。

最后，为了支持单个群库的概念，lib 目录现在也更复杂了。lib 之下的子目录 local 也类似于图 7-2 中的全局 lib 目录，因为这个子目录保存了单个包库文件的所有不同版本。lib 没有包含对应于每个包的库文件，而是包含了一个代表了它们的联合的单个库文件。只提供一个库文件可以使普通客户使用这个群时更加方便。

如图 7-20 所示，每个单独的包库可能有不止一个版本。后缀_g 用来指明某个库有调试符号。为了诸如性能监控或运行时内存边界校验等目的，许多其他的特殊库形式也可能存在。如果这个群很大，为整个群使用甚至建立特殊用途的库可能是不可行的。但开发者一般会改为标识他们想更仔细分析的群中的单个包。

① 符号连接或等价物可以代替真正的拷贝。

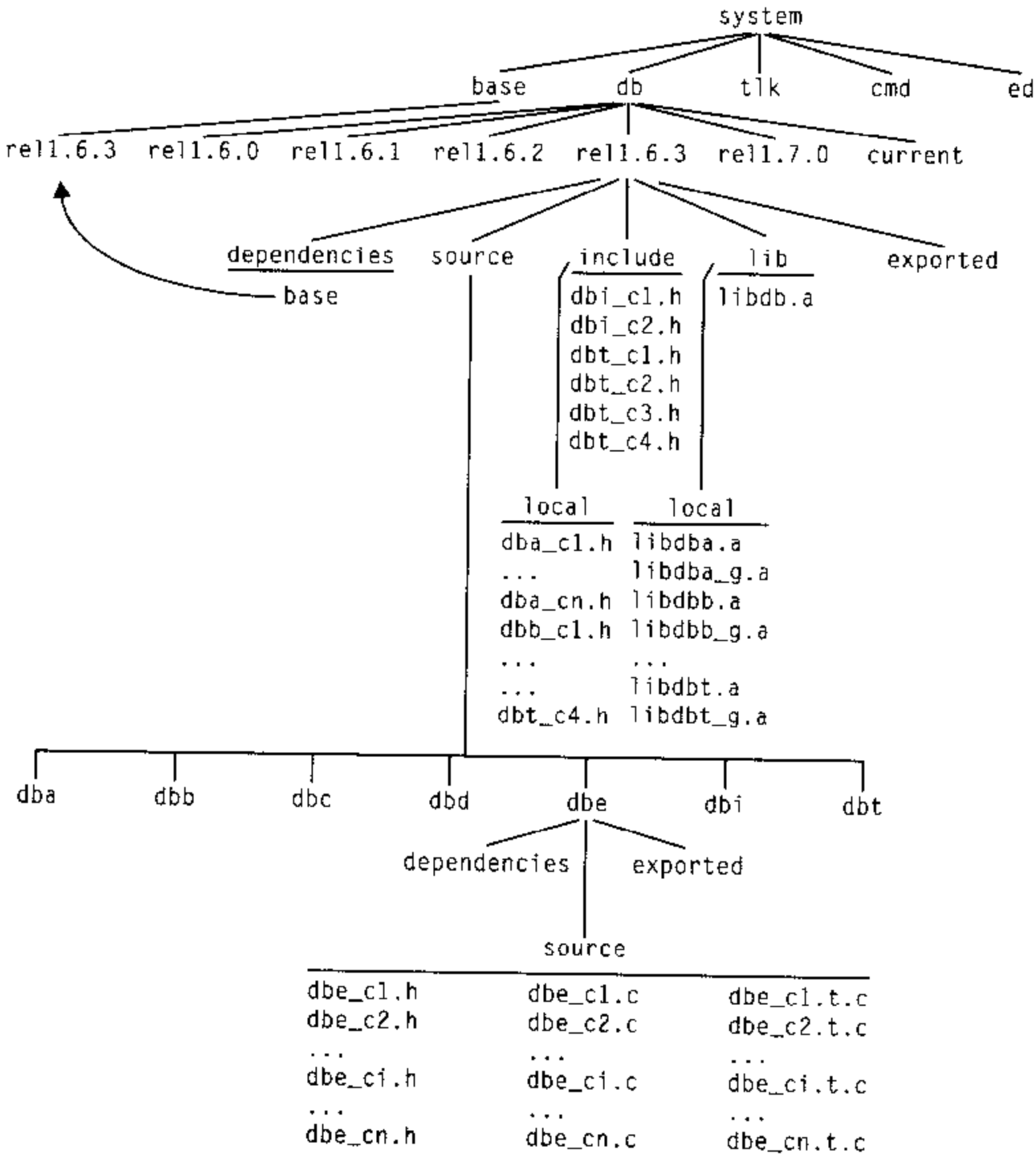


图 7-20 包群的一个开发目录层次结构

用这种开发目录结构来发布一个群是直截了当的。这个群的整个目录和文件结构（除了包含在 include 和 lib 目录下的文件）以及版本在一个新的版本下（例如，system/db/rel1.7.1）被复制。这个新版本的依赖（例如，system/db/rel1.7.1/dependencies/base）被调整指向较低层次软件的新版本（例如，system/base/rel1.7.1）。然后该群中的每个包都以分层的顺序被拷贝到新版本、重建和测试。

当每个包被建立时，为了给群中的其他包使用而从包中输出的头文件被放在局部的include 目录（例如，system/db/rel1.7.1/include/local/dba_c3.h）。同时，单个包库的每个版本都被放进该群的局部 lib 目录（例如，system/db/rel1.7.1/lib/local/libdba.a）。一旦该群的所有局部包都已经建立了，包库就被合并成一个单个的库，并被放进 lib 目录（例如，system/db/rel1.7.1/lib/libdb.a）。只有该群的客户为了使用这个群而需要的那些头才会输出到 include 目录（例如system/db/rel1.7.1/include/dbi_c1.h）。

目录 current 不发布，但是会保留给正在进行的开发。虽然对已发布版本的修改很少，并且会小心控制（见 7.6.2 节），但对当前（开发）版本的修改可能会经常出现。

通过在层次结构中增加一个节点（就在依赖机器的文件之前），我们可以扩展这个目录结构以支持多种平台。例如：

```
system/db/rel1.7.1/lib/libdb.a
```

替换成

```
system/db/rel1.7.1/lib/sun4os4/libdb.a
```

或

```
system/db/rel1.7.1/lib/hppaux9/libdb.a
```

以反映所需的机器体系结构和操作系统的组合。

原　则

最小化修改之后的源代码的重编译时间，可以显著地减少开发开销。

编译开销部分地取决于一个 include 目录中头文件一个函数的数量，但是它更依赖于编译器为了定位所有需要的头文件而必须搜寻的目录数量。在大多数系统上，当所有的头文件都只驻留在少数几个目录中时，编译组件的速度要明显快于在头文件分布在许多单个（包级）include 目录的情况下做这项工作的速度。

为了使过多数量的单个 include 目录的开销具体化，我设计了一个实验来比较用单个的包 include 目录、群 include 目录、层 include 目录以及单个的全局目录来编译一个组件的系统开销。我作了若干数量级假设：

- 每个组件 10 个#include 指令
- 每个包 10 个组件
- 每个群 10 个包
- 每个层 10 个群
- 每个系统 10 个层

这个实验对包含 1 个、10 个、100 个、1000 个和 10000 个组件的系统在有着不同数量 include

目录的结构上重复进行。图 7-21 展示了用 CFRONT 编译器在一个 SUN SPARC 10 工作站上以及用本地的 C++编译器在一个 HP 7000 工作站上运行该实验的结果。

作为参考，编译一个另外的只依赖一个包 include 目录的空组件，在 SUN 上编译大约要 1 CPU 秒，在 HP 上大约要 0.2 CPU 秒。随着系统规模的增加，编译开销的增加在 SUN 上是适中的，在 HP 上是微不足道的。对于 1000 个组件数量级的系统，使用单个的包 include 目录来编译一个组件的开销，在 SUN 上可能用了两倍的 CPU 时间，在 HP 上则用了 4.5 倍的 CPU 时间。对于更大型的系统，使用单个的包 include 目录的开销更加明显——在 SUN 上大约一个数量级，在 HP 上也差不多[1]。

子系统中组件的数量	包含目录的数量				包含目录的数量			
	1	10	100	1000	1	10	100	1000
10	1.0 (100%)			相对于使用单个包含目录而言	0.2 (100%)			以 CPU 秒为单位的时间
100	1.0 (100%)	1.0 (100%)			0.2 (100%)	0.2 (100%)		
1000	1.1 (100%)	1.1 (100%)	2.0 (182%)		0.2 (100%)	0.2 (100%)	0.9 (450%)	
10000	1.4 (100%)	1.4 (100%)	2.3 (164%)	15.1 (1079%)	0.2 (100%)	0.2 (100%)	0.9 (450%)	15.2 (760%)
	在 SUN SPARC 10 之上的 CPU 时间				在 HP 735 上的 CPU 时间			

图 7-21 由多 include 目录引起的编译时间/（系统开销）

减少重编译和重连接所花的时间量可以对生产率产生重大影响。幸好，我们有两个办法可以为不能购买更快速硬件部件的大型系统减少这种问题。最有效的办法是通过绝缘来减少头文件的数量，如第 6 章所讨论的那样。另一个办法效果差一些（但还是有效的），就是减少编译器在一个给定的编译过程中需要搜寻的 include 目录的数量。这样的方法之一就是，将从较低层次群（由文件 dependencies 标识）输出的头传送到一个依赖群自己的输出头目录中（也许还有定义在文件 exported 中的额外过滤）。

如图 7-22 所示，不是所有由 base 和 db 层输出的头都是 tlk 层的客户所需要的。tlk 不应该只发布自己的头，而是除了发布它自己的输出头之外，还应该重新发布较低层次输出头的必要子集。用这种方法我们可以避免强迫其客户为 base 和 db 指定不同的 include 目录。现在

① 注意，在编译依赖于一个大型子系统的组件时，实际流逝的“wall”时间甚至可能超过 CPU 时间。例如，编译一个这样的组件，它依赖一个包含 1000 个组件的系统（分布于 100 个单独的包 include 目录中），wall 时间在 SUN 上是 22.1 秒，在 HP 上是 4.8 秒。当这个系统由 10000 个组件构成时，编译一个单个组件的 wall 时间增长为 225.5 秒（SUN）和 209.2 秒（HP）。

tlk 层的客户为了访问 tlk 层的功能只需要指定一个 include 目录。在这里，绝缘再一次使我们能够减少暴露给客户的头的数量，以提高它们的编译速度。

system/base/rel1.7.1/include	system/db/rel1.7.1/include	system/tlk/rel1.7.1/include
pub_c1.h	dbi_c1.h	tlk1_c1.h
pub_c2.h	dbi_c2.h	tlk1_c2.h
usr_c1.h	dbt_c1.h	tlk1_c3.h
pub_c2.h	dbt_c2.h	tlk2_c1.h
usr_c3.h	pub_c1.h	tlk3_c1.h
	pub_c2.h	tlk3_c2.h
	usr_c1.h	dbi_c1.h
		dbt_c1.h
		pub_c1.h

图 7-22 最小化一个客户的包含头的开销

另一个可供选择的方法是，在编译之前让客户群负责将其所有需要的头“预取”进一个 include 目录中。要求客户创建一个特殊用途的目录来有效重用一个子系统，会明显地导致该子系统更不可重用。这第二种方法似乎更不友好，因为它强迫客户作更多的工作来使用这个子系统；但是它可以在一种不友好的环境中发挥它的优势。

7.6.2 补丁

对一个版本进行修改是对开发的潜在破坏，所以一旦一个版本创立了，保持其稳定性是很重要的。有时在一个稳定的版本中会发现一个不能等到下一版本才修改的严重错误。从头开始纠正这个错误和重建整个系统既具有破坏性又消耗时间，尤其是对于潜在大型的客户群来说。如果这个问题是存在于实现中的，通常比较划算的做法是修补它。

> **定 义**：**补丁（patches）是对前一次发布软件的局部修改，以修补组件内不完善或效率很低的功能。**

最简单、最安全和最常用的补丁种类包括只对组件的.c 文件进行修改。在.c 文件被修改和编译之后，产生的.o 文件可能（在 Unix 系统中）被放在一个连接命令的一个库文件之前，以取代已有的.o 文件。当然，客户可以选择是否连接进这些补丁文件——对某些客户来说，这些修改可能并不值得以损失稳定性为代价。

原 则

补丁一定不能影响任何已存在对象的内部布局。

不是每个错误都能修补。幸好，如果组件的头文件没有输出，则这样一个对象的布局只为包内的组件所知。在这样的情况下，几乎总是可以通过提供一个或多个解决该问题的补丁

文件来修正错误。但是，即使头文件被输出了，有许多错误也可以在不必重建整个系统的情况下被修补。一个组件的实现越绝缘，它就越有可能在不影响包外组件的情况下被修补。

考虑图 6-49 中所示的非绝缘类 Example，它是完全内联实现的。如果 Example 的头被输出了，就没有任何办法可以用来修补它的错误。我们对类 Example 的实现的任何修改都会迫使所有使用它的客户重编译。现在将它和图 6-51 中的完全绝缘的类 Example 相比较，图 6-51 中的 Example 没有内联函数、没有继承，只暴露了一个单个的指向其数据的不透明指针。事实上我们完全可以修补任何纯粹实现的问题，而不要求这个类的客户重编译。

理想状态下，一个补丁根本不会要求修改任何头文件。修改一个输出头文件中的信息会潜在地影响无限数量的客户；所以这样的修改最好避免。虽然危险，我们还是可以进行许多不会使我们的版本失效的修改，即使它可能意味着改变已有的输出头文件。如果我们能保证这些局部修改的影响是连接兼容的，并且不会使版本失效，我们就可以节省发布第二个版本的可观开支和工作量。

下列几种修改是相对安全的：

- 改变一个非内联函数的函数体。
- 改变.c 文件中任何有内部连接的结构。
- 给版本增加一个新的输出头文件。
- 对一个类增加一个友元声明。
- 放宽一个已有的访问限定符（例如从 protected 到 public）。
- 给一个类增加一个新的非虚函数（危险）。
- 给一个头增加一个类或自由运算符（危险）。

注意，最后四种情况需要修改头文件。在进行了这样一个修改之后，这个头文件应该被人工回溯（backdate），以防止客户进行不必要的重编译。最后两个修改是危险的，因为有可能从已被某个客户包含的头文件的函数或运算符重载中引入二义性。如果最后这个修改是对一个新的单独的头文件进行的，那么这个结构就不会有机会影响任何已存在的用法。

下列修改可能会潜在地破坏一个版本：

- 增加、重排序、修改或删除任何数据成员。
- 增加、重排序或删除任何虚函数。
- 改变任何函数的基调或返回值。
- 增加、重排序、修改或删除任何继承关系。
- 改变头中的任何有着内部连接的结构。
- 缩小一个类成员的访问权限（例如从 protected 到 private）。
- 在邻近的数据成员之间引入访问限定符[①]。

这里介绍的修改列表是不完整的，但应该给出了可以通过补丁（如果做得仔细的话）来

① 见 ellis，9.2 节，173 页，以及 11.1 节，241~247 页。

局部完成的修改种类的概念和风格。真正的要求只有：

（1）确保修补后的连接时兼容性。

（2）避免因为在头文件时间戳上的修改而引起客户重编译。

（3）确信如果我们想重建系统的话一定会成功。

创建一个有效的开发环境还有很多需要注意的问题，在此不能一一介绍。所使用的技术将取决于操作系统。类似于图 7-20 中所示的组织，已成功地应用在基于 Unix 的系统上开发超大型项目。

7.7 main 程序

当我们用 C++编写一个程序时，C++语言要求我们提供一个惟一的 main 函数定义来连接操作系统和（尤其是）处理任何命令行参数。但是，当我们投入了几十、几百甚至几千人年来创建一个 C++系统时，系统并没有一个单一的顶部。也就是说，总是由若干个可执行文件（每个都有自己的 main 过程）一起构成系统。我们的设计方法学已经创建了可重用子系统的一个层次化的集合，而不是产生单个的程序。这些子系统有很多将会用于独立的输入验证器（input verifiers）、翻译器（translators）、阅读器（viewers）、报告程序编制器（report generators）、输出分析器（output analyzers）等等。独立“main”程序的数量很可能随着系统的发展和成熟而增长。

原 则

从一个定义了 main 的编译单元中分解出独立可测试的和潜在可重用的功能，本质上能够使程序的整个实现在一个更大型的程序中重用。

一个定义 main 的编译单元（不同于分层次的测试驱动程序）的用途是，提供一个有命令行接口、解释环境变量和管理全局资源的 C++子系统——再无其他。一个常见错误是，在一个定义 main 的文件中放置了太多的代码。这样的代码不能用一个 C++测试驱动程序来增量式地测试，也不能在一个更大型的 C++程序中重用。

例如，我们来考虑一个执行某种桌面出版功能的程序——比方说一个词汇表生成器（glossary generator），如图 7-23 所示。一个词汇表生成器的功能是，读取一篇输入文档并将它存储成一组惟一的单词。这种输入被基于定义了一组拦截单词（blocking words）的第二个输入过滤。拦截单词是常用词（例如 and、this、a 等等），它们可能不适合于一个词汇表。接下来，再将留下的单词集合与第三个输入——一个词典（thesaurus）——进行比较，该词典在这种上下文中代表着到更常用或更基本术语的别名或替代形式的映射。例如，在 C++中，method（方法）是 member function（成员函数）的另一个名称。最后，未被拦截和不是别名的所有基本词都必须被定义在第四个输入——一个词典（dictionary）中。词典是从一组常用

术语到它们的各自定义的映射。词汇表生成器的输出是一组未定义的术语以及对应于被认可术语的定义（字典中）的子集（按字母顺序排列）。

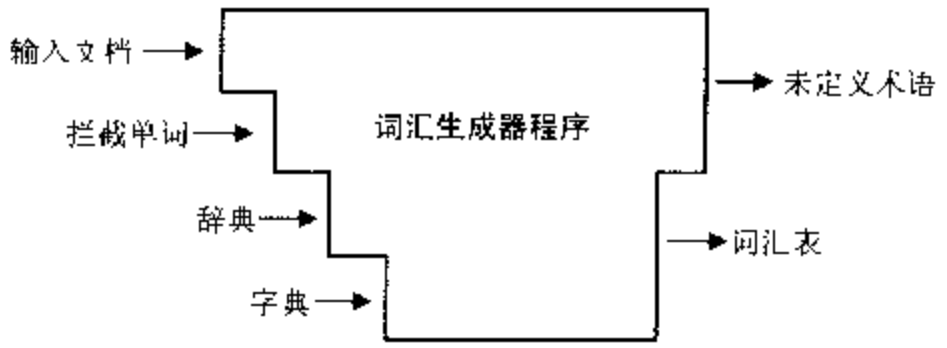


图 7-23 词汇表生成器

我们应该从哪儿开始设计这个程序呢？在一种自顶向下的方法中我们可能应该从 main 开始，对吗？也许是的。但是，我们应该在工作中努力将 main 提供的功能的实现分解，成独立可测试和潜在可重用的组件。（回忆一下 5.9 节中介绍的用来减少不好管理的大型组件复杂度的分解技术。）怎样从命令行输入信息只是我们关心的事情之一。我们应该问的另一个重要问题是，基础功能如何能够在某一天被方便地集成到一个更大型的程序中（例如，一个桌面出版框架）。为了完成一个模块化的设计，我们必须为直接最终用户同时研制基本程序接口和单独的命令行接口。

在词汇表生成器设计中的一个中心件，很可能是一个定义在词汇表生成器组件中的词汇表生成器对象，就像图 7-24 中举例说明的这一个。为了使用一个词汇表生成器对象，我们首先需要用拦截单词、别名和词典定义来编制它。在 **dtp_GlossGen** 类中的显式的操纵函数就是为这些目的而提供的。在编制完词汇表生成器之后，我们可以用 **addTextWord** 操纵函数把输入文本的单个词装载进词汇表生成器对象。一旦我们为文档装载完所有的输入文本，我们将创建一个迭代器以字母顺序访问词汇表定义。提供第二个迭代器后，我们可以对任何未定义术语进行顺序访问。完成对第一个文档的处理后，我们可能希望让若干个相关文档通过同一个生成器。操纵函数 **clearInputWords** 允许我们在保留前面编好的拦截单词、别名和定义的同时，再次开始一个新的文档。

```
// dtp_glossgen.h
#ifndef DTP_INCLUDED_GLOSSGEN
#define DTP_INCLUDED_GLOSSGEN

class dtp_GlossDefIter;
class dtp_GlossUndefTermIter;

class dtp_GlossGen_i;                    // fully insulated implementation
class dtp_GlossGen {
    dtp_GlossGen_i *d_this;

    friend dtp_GlossDefIter;
```

```
    friend dtp_GlossUndefTermIter;

  private:
    // NOT IMPLEMENTED
    dtp_GlossGen(const dtp_GlossGen&);
    dtp_GlossGen& operator=(const dtp_GlossGen&);

  public:
    // CREATORS
    dtp_GlossGen();
    ~dtp_GlossGen();

    // MANIPULATORS
    int addBlockingWord(const char *blockingWord);
    int addAlias(const char *alias, const char *keyTerm);
    int addDefinition(const char *keyTerm, const char *definition);
    int addTextWord(const char *textWord);
    void clearInputWords();
};

class dtp_GlossDefIter_i;                    // fully insulated implementation
class dtp_GlossDefIter {
    dtp_GlossDefIter_i *d_this;

 private:
    // NOT IMPLEMENTED
    dtp_GlossDefIter(const dtp_GlossDefIter&);
    dtp_GlossDefIter& operator=(const dtp_GlossDefIter&);

  public:
    // CREATORS
    dtp_GlossDefIter(const dtp_GlossGen& glossaryGenerator);
    ~dtp_GlossDefIter();

    // MANIPULATORS
    void operator++();

    // ACCESSORS
    operator const void *() const;
    const char *keyTerm();                   // Provides an association
    const char *definition();                // (keyTerm, definition) so
                                             // we choose not to define an
                                             // operator()() here.
};

class dtp_GlossUndefTermIter_i;              // fully insulated implementation
class dtp_GlossUndefTermIter {
    dtp_GlossUndefTermIter_i *d_this;

  private:
    // NOT IMPLEMENTED
    dtp_GlossUndefTermIter(const dtp_GlossUndefTermIter&);
```

```
    dtp_GlossUndefTermIter& operator=(const dtp_GlossUndefTermIter&);

  public:
    // CREATORS
    dtp_GlossUndefTermIter(const dtp_GlossGen& glossaryGenerator);
    ~dtp_GlossUndefTermIter();

    // MANIPULATORS
    void operator++();

    // ACCESSORS
    operator const void *() const;
    const char *operator()() const;    // Returns just the current undefined
};                                     // term so operator()() is ok here.

#endif
```

图 7-24　一个词汇表生成程序包装器组件的绝缘接口

我们的 main 仍然需要创建一个 dtp_GlossGen 对象，然后为了适当地编制这个对象，将输入从命令行（被命令行引用的文件）转换成 dtp_GlossGen 成员函数调用。但是，为了编制词汇表生成器，我们可能选择使用多种输入语法。所以，把程序接口和任何一种语法绑在一起都是不合适的。相反，我们创建了一个单独的组件负责读取某个给定的输入、解析那个输入，并相应地试运行该词汇表生成器组件。该词汇表生成程序组件体系结构的最高层显示在图 7-25 中。

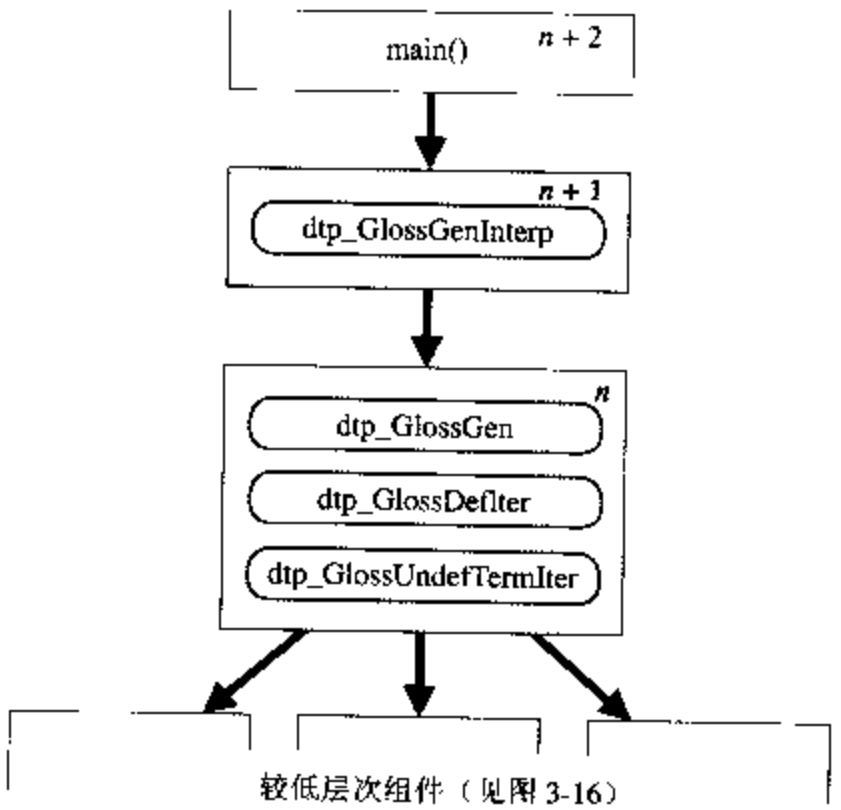


图 7-25　高层词汇表生成器体系结构

图 7-26 中所示的解释器组件的工作是，将它自己连到一个词汇表生成器对象上，然后基于一个在指定的输入文件或输入流中找到的命令，相应地试运行那个对象。解释器对象本身

用两部分信息来编制：

```
// dtp_glossgeninterp.h
#ifndef DTP_GLOSS_GEN_INTERP
#define DTP_GLOSS_GEN_INTERP

class dtp_GlossGen;
class ostream;
class istream;

class dtp_GlossGenInterp_i;
class dtp_GlossGenInterp {
    dtp_GlossGenInterp_i *d_this;

  private:
    // NOT IMPLEMENTED
    dtp_GlossGenInterp(const dtp_GlossGenInterp&);
    dtp_GlossGenInterp& operator=(const dtp_GlossGenInterp&);

  public:
    // CREATORS
    dtp_GlossGenInterp(dtp_GlossGen* glossGen);
        // create an interpreter

    ~dtp_GlossGenInterp();
        // destroy this interpreter

    // MANIPULATORS
    void setErrorStream(ostream& errorStream);
        // Set output stream to which detailed errors will be reported.
        // By default, this stream is cerr.

    // ACCESSORS
    int exercise(const char *fileName = "-") const;
        // Parses commands from the specified input file.  Returns
        // -1 on I/O error, 0 on success, and 1 on syntax error.
        // The default "-" stands for "standard input" (i.e., cin).

    int exercise(istream& input, const char *fileName = 0) const;
        // Parse commands from the specified input stream.  Returns
        // -1 if an I/O error occurs; otherwise returns the line
        // number of the first syntax error.  If successful, this
        // function returns 0.  The second argument is used only
        // for the purpose of identifying the input source when
        // formatting syntax error messages to the error stream.
};

#endif
```

图 7-26　词汇表生成器的完全绝缘的解释器组件

（1）它要操纵的词汇表生成器的地址。

（2）解释器要向其报告详细语法错误信息的错误输出流。

解释器提供了两个访问函数来试运行有关的词汇表生成器对象的功能。第一个函数只获取一个文件名称并把它打开（如果可能的话）。然后这个函数调用第二个（更基本的）函数形式，这个函数形式使用一个开放流和一个自由的“文件”名，以格式化错误信息。较低层功能被暴露在接口上，这样数据流的源不必是一个真正的文件。注意这两个成员函数没有影响解释器的状态；它们只影响词汇表生成器的状态。

最后，在 main 中所有剩下要做的事就是创建这两个对象和对一组命令行参数排序。如果没有指定命令行参数，cin 应该默认假定。词汇表生成器的一个微小的单机 main 驱动程序显示在图 7-27 中。这个驱动程序举例说明了一种可重用的模式，适合于多种独立应用程序。

```
// dtp_glossgeninterp.t.c
//
// Usage: a.out [ <file name> | - ]*
//
// Example:
//
//      john@john: a.out stuff.abc such.def -
//
//      The above command line will first read input from the file
//      "stuff.abc", then read input from the file "such.def", and
//      finally read from standard input (cin).

#include "dtp_glossgeninterp.h"
#include "dtp_glossgen.h"

const char *const defaultArgs[] = { "", "-" };   // has internal linkage
const int defaultNumArgs = sizeof defaultArgs / sizeof *defaultArgs;

main(int argc, char *argv[])
{
    int status = 0;
    const char *progName = argv[0];
    int numArgs = argc > 1 ? argc : defaultNumArgs;
    const char *const *args = argc > 1 ? argv : defaultArgs;

    dtp_GlossGen glossaryGenerator;
    dtp_GlossGenInterp interpreter(&glossaryGenerator);

    for (int i = 1; i < numArgs && 0 == status; ++i) {
        status = interpreter.exercise(args[i]);
    }

    return status;
}
```

图 7-27 词汇表生成程序和解释程序的独立 main 驱动程序

main 的所有权伴随着特权和责任。在一个给定的程序中只有一个 main。正是这段代码应该负责读取环境变量和建立全局资源。拥有 main 的人拥有全局名称空间。例如，如果包含 main 的文件去定义或存取外部全局变量、没有使用包前缀等等并没有害处。但是，为了确保我们集成任意子系统的能力，系统的其他任何部分都不应该污染全局名称空间或企图侵占全局资源。

指导方针

通常，避免赋予一个组件一种可被其他组件拥有的特权，否则将对作为一个整体的系统产生不利影响。

这种（康德式的）哲理要求我们，除了定义 main 之外，我们不应该试图做某件其他人也可能做的事情，否则会对整个系统产生消极后果。

内联函数的过度使用，只是这种可能导致微妙集成问题出现的行为之一。通过随便地声明不适当的大型成员函数为内联，我们可能常常会改善我们自己的处在绝缘中或在一个小型子系统内的对象的运行时性能。但是，这种运行时改进是以重复代码和增加可执行文件大小为代价的。

当这种私自构建的子系统被集成进较大型子系统时，增加的代码量开始显示它的不利影响。设计来改进公共使用例行程序性能的硬件机制被内联代码的过度重复所破坏。增加的程序大小减少了操作系统可保持在核心部分的可执行程序的百分比，这就导致了要增加转储。在集成的某一层次，许多这样的对象将真正开始运行得更慢（过度内联的结果），如果一些较大的函数被声明为非内联，它们会运行得更快。这种自私自利的最终结果是整个系统性能的降低。

因单个的开发者不够勤奋而可能消极影响一个集成系统的另一个例子，涉及非局部静态对象的不加选择的使用，正如 7.8 节中所讨论的那样。还有另一个在一个组件或子系统的一部分中避免这样的利己主义行为的特殊例子，将在 10.3.4.2 节中的特定类内存管理的上下文中讨论。

主要设计规则

只有定义了 main 的.c 文件才有权重新定义全局 new 和 delete。

这条哲理的一个重要的特殊情况是，只有 main 的拥有者才有权重新定义全局运算符 new 和 delete。未定义 main 的组件被禁止进行这种单方面的行为。另外，若两个独立的子系统，每个都重定义了一个惟一的资源（例如，全局运算符 new），则它们不是连接兼容的。

总结：设计一个大型系统时没有顶。main 的用途只是提供一个具有命令行接口、解释环

境变量和管理全局资源的C++子系统——别无其他。将main提供的功能分解成单独的组件有助于分层次测试，并且可以更容易地集成入更大型的系统。定义main的.c文件拥有全局名称空间，并且不必遵守某些适合于普通组件的设计规则。对于没有定义main的组件，应该小心不要接受这样的特权：如果该特权也被其他组件接受了，可能会危及整个系统的安全。

7.8 启动（start-up）

程序被第一次调用和控制线程进入main之间所经历的时间在本书中称为**启动**。正是在这段时间，在每个编译单元中的所有非局部静态对象被潜在地构造，如图7-28所示[①]。

> **定义：启动时间，也称为启用（invocation）时间，就是程序被初次调用和控制线程进入main之间的时间。**

```
// my_component.c
#include "my_component.h"
#include "pub_list.h"
#include <sys/types.h>
#include <sys/time.h>

// static object at file scope
static pub_List list;                          // constructed at start-up

// static data member initialized by function call
static time_t startUpTime = time(0);           // called at start-up

// static object in class scope
pub_List my_Class::d_List;                     // constructed at start-up

// ...
```

```
// defines class my_Class
// defines class pub_List
// declares typedef time_t
// declares ::time()
```

图7-28 在启动时非局部静态变量的初始化

因为定义在单独编译单元中的非局部静态变量的初始化顺序是依赖于实现的，必须特别小心确保这样的静态对象在被使用之前初始化。当目标是要提供一个全局可访问对象的单一实例时，我们对全局数据的反感（在2.2节已作过说明）引导我们去寻找一个替代方法。我

① 按照C++语言规范（**ellis**，3.4节，19页），一个编译单元内的所有非局部静态对象都必须在那个编译单元内定义的任何函数或对象被第一次使用之前被构造；但是在实践中，所有这样的初始化都可能并且通常也确实在启动时进行。

们不在文件作用域创建一个有外部连接的对象的实例，而通常可以用一个逻辑结构来达到我们的目的，这个逻辑结构一般称为模块（module），在C++中被作为一个只包含静态成员的类来实现[1]。

指导方针

宁愿要模块而不要对象的非局部静态实例，尤其是在以下情况下：

（1）需要在一个编译单元之外对结构进行直接访问。

（2）该结构在启动期间不需要或在启动之后不是马上需要，并且初始化结构本身的时间是显著的。

确保静态结构在它们被使用之前适当初始化的要求已被证明是必须的[2]。但这种初始化在启动时可能具有的综合影响力更未受到普遍重视。对于小程序来说，在启动时初始化一些静态结构，可能对用户对启用程序所需时间的感觉没有值得注意的影响。但是，系统越大，就会有越多的独立静态结构要求在启动时初始化。

原　则

程序中的每一个非局部静态对象结构都会潜在地增加启用时间。

因为每个定义在文件作用域或类作用域内的静态对象都在进入main前被潜在地构造，一个其组件经常定义这样的静态对象的超大型系统可能要花令人无法接受的长时间来完成启动。事实上，有文献记载过这样的超大型系统案例，由于天真地忽视了启动时的初始化开销，导致启用时间超过了10分钟！

非局部静态对象由C++运行时系统自动初始化和析构；单个组件对它们的不加选择的使用是一种利己主义行为方式，它降低了集成系统的启用性能。虽然我们没有办法阻止这些静态实例在启动时被初始化，但对于怎样和何时初始化模块却有相当大的灵活性。幸好，总是有可能将一个对象的单个全局实例转换成一个模块，当被初始化时，该模块动态分配那个对象[3]。一旦被初始化，该模块可以成功地返回一个指向它现在拥有的动态对象的引用。

---

① 一个模块也可以指一个类似于组件的物理实体，但是它有一个过程接口。注意，在ANSI C中，实现一个逻辑模块的惟一办法就是将它作为一个物理模块（即作为一个单独的在文件作用域定义了静态数据的编译单元）来实现。关于模块的更多知识，见**stroustrup**，1.2.2节，16页。

② **ellis**，3.4节，20页；**meyers**，Item 47，178页。

③ 见Singleton design pattern（**gamma**，第3章，127~138页）。

7.8.1 初始化策略

至少有四种不同的技术可用来确保一个模块在它被使用之前初始化：

- 唤醒（wake-up）初始化
- 显式的 init 函数
- 灵巧计数器
- 每次检查

这些初始化策略每一种都有自己的优缺点；最好的选择取决于如下若干因素：

- 初始化该模块所需的时间
- 该模块真正投入使用的可能性
- 每个模块函数调用所做的工作量
- 对模块函数进行调用的频率
- 直接使用该模块的组件的数量
- 在程序退出之前是否有释放/再分配资源的需求

7.8.1.1 唤醒初始化技术

到目前为止，初始化一个模块的最好方法是设法让该模块在一种被初始化的状态中“唤醒”。例如，使用这种唤醒方法，一个全局注册模块可能作为记录连接的一个链表来实现，如图 7-29 所示。

只要所有的静态数据成员都是基本类型（指针[①]、整数、双精度整数、字符数组等），它们将在装载时（即在启动之前）被初始化，不会影响启动时间。如果我们改为将一个 pub_List 对象（即不只是一个指针）作为类 ax_Registry 的一个静态成员嵌入，那么该成员将会自动初始化（在启动期间），引起运行时开销。

7.8.1.2 显式的 init 函数技术

不是所有的模块都可以唤醒初始化。更普遍的情况是，一些组件可能定义模块或包含静态结构，这些模块或静态结构必须在运行时在被使用之前初始化。使这种初始化能够完成的一种方法是，给每一个这样的组件提供一个 init 函数，如图 7-30 所示。必须在这个 init 函数被调用（至少一次）之后，才能使用该组件提供的静态结构。init 函数方法是相当灵活的，使用了它，初始化可以完全延迟到启动阶段之后，并且只有当这个组件被真正需要的时候才被调用。

① 一个指向用户自定义类型的非局部静态指针可以在装载时初始化；特别地，通常被初始化为 0。

```
// ax_registry.h
#ifndef INCLUDED_AX_REGISTRY
#define INCLUDED_AX_REGISTRY

class ax_RecordLink;
class ax_Record;

class ax_Registry {
    static ax_RecordLink *d_list_p;

  public:
    static void addRecord(ax_Record *record);
        // Add record to registry; registry now owns the record.

    static void cleanup();
        // Free all dynamicly allocated memory; reset to empty.

    // ...
};

#endif
```

```
// ax_registry.c
#include "ax_registry.h"
ax_RecordLink *ax_Registry::d_list_p = 0;
// ...
```

图 7-29　唤醒的模块已经被初始化了

虽然灵活，但这种显式的 init 函数方法很容易引起错误：客户一般会忘记在使用一个组件之前将其初始化，因而常常导致致命的运行时错误。为了缓解这个问题，我们可以提供一个与众不同的、有着一个 init 函数的包级组件（例如，ax_package），它可以初始化任何定义在这个包中的需要运行时初始化的组件。同时，它也可以为该包所依赖的所有其他包调用 init 函数。

原　则

初始化你不另外直接依赖的组件会显著地增加 CCD。

包级 init 函数方法有一些严重的缺陷。首先，确保每个被包含组件的 init 函数和这些组件依赖的每个包的 init 函数都获得包级 init 函数的调用，会有明显的维护负担。更有问题的是，初始化整个包会显著地增加耦合，潜在地在连接时拖入许多并不另外需要的组件。正是因为后面这个原因，我们最好避免使用包级 init 函数——尤其是对于有着水平依赖结构的广泛可重用的包。相反，对于那些直接依赖其他需要显式初始化的组件的组件来说，单独地初始化这样的组件更可取。客户组件可能反过来提供一个 init 函数供它自己的直接客户使用，或者可能合并一些其他的初始化技术。在一种合适的粒度水平上来维护初始化图有助于让一个系统的 CCD 保持最小化。

7.8.1.3 灵巧计数器技术

当静态对象使用其他静态对象时，初始化问题将变得更加复杂。为了举例说明，假设图7-30中的全局 pub_List 对象本身使用了一个需要运行时初始化的静态结构（例如，用于特定类内存管理，在 10.3.4 节讨论）。在 pub_List 的静态存储管理被初始化之前试图在启动时创建一个 pub_List 静态对象，很容易导致致命的运行时错误。因为这两个初始化的相对顺序是依赖实现的，必须采取特殊的预防措施。

```
// ax_table.h
#ifndef INCLUDED_AX_TABLE
#define INCLUDED_AX_TABLE

class ax_RecordLink;
class ax_Record;

class ax_Table {
    static ax_RecordLink **d_array_p;
    static int d_size;

  public:
    static void init(int size);
    static void cleanup();
    static int addRecord(const ax_Record& record);
    // ...
};

#endif
```

```
// ax_table.c
#include "ax_table.h"
#include "pub_List.h"
#include <memory.h>                    // declare memset
// ...

static pub_List s_list;                // global within this
                                       // component only

ax_RecordLink **ax_Table::d_array_p;

int ax_Table::d_size;

void ax_Table::init(int size)
{
    if (d_array_p) return;
    d_size = size;
    d_array_p = new ax_RecordLink *[size];
    memset(d_array_p, 0, size * sizeof *d_array_p);
}

// ...
```

图 7-30 给组件提供一个显式的 init 函数

为了取代易于出错的 init 函数方法，我们可能考虑使用灵巧计数器方法[1]。在这种方法中，一个初始化类的伪静态实例被放在一个组件的头文件的文件作用域中，如图 7-31 所示。这个静态实例的部分用途是计算包含这个组件的头的其他组件的数量。被一个编译单元包含的这个伪对象的每一个静态实例都将在启动时被构造（以某种顺序）。该伪对象的一个静态实例第一次被构造时，静态计数从 0 增大至 1，并且该伪对象知道要去初始化它的组件[2]。随后每一次构造一个伪实例，惟一的影响就是这个静态计数增加。

```
// pub_list.h
#ifndef INCLUDED_PUB_LIST
#define INCLUDED_PUB_LIST

// ...

class pub_List {
    // ...
};

struct pub_ListInit {
    pub_ListInit();
    ~pub_ListInit();
} pub_listInit;

#endif
```

```
// pub_list.c
#include "pub_List.h"

// ...

static int s_niftyCounter = 0;

pub_ListInit::pub_ListInit()
{
    if (0 == s_niftyCounter++) {
        // init pub_List's static constructs
    }
}

pub_ListInit::~pub_ListInit()
{
    if (0 == --s_niftyCounter) {
        // clean-up pub_List's static constructs
    }
}
```

图 7-31　使用灵巧的计数器来确保使用前的初始化

当程序退出时，过程是相反的：每个伪对象的析构函数都使静态计数减 1。当计数到达 0 时，伪对象知道可以清理掉该组件了。iostream 使用这种灵巧计数器计数来保证 cin、cout、cerr 和 clog 在使用前被初始化。

这种灵巧计数器方法的好处在于它是极简单的。想使用一个需要运行时初始化的组件而

① 灵巧计数器方法在 **ellis**（3.4 节，20~21 页）中讨论。

② 注意，定义在一个编译单元内的任何非内联函数的使用，都将触发所有定义在那个编译单元内的非局部静态实例（倘若 main()已被进入了）。

不先包含该组件的头是不可能的。这样做将导致构造一个伪对象，它反过来迫使一个未初始化的组件变成已初始化的。所有这些都发生在包含该组件的头的编译单元能够使用新提供的声明来访问该组件之前。因而一个使用这种灵巧计数器初始化方法的类，可以安全地被静态实例化[1]，即使这个类本身使用了其他也使用了这种技术的非局部静态对象[2]。

使用灵巧计数器方法的另一个好处是，只有为了连接而真正被需要的包里的那些组件才会被初始化。灵巧计数器初始化机制本身的运行时开销是可以忽略不计的，除了包含 N 个直接依赖于 M 个模块的（N 和 M 都很大）组件的不合理设计之外。通常，和构造真正初始化该组件的第一个静态对象相比，这种系统开销并不大。

使用灵巧计数器的主要缺点是，甚至只可能在运行时使用的组件也一概在启动时被初始化。对于基于请求被装入一个正在运行的程序的动态库来说，一个非局部静态初始化常常要求在启动时拖入这些库，这会破坏命令装载的目的。如果在初始化过程中所做的工作量很大（例如，装载一个多维表），考虑使用另一种技术是明智的，这种技术允许我们将初始化推迟到程序执行时。

非局部静态对象一般用来在启动时将独立具体类型的集合载入一个全局注册表（global registry）。但是，连接到一些库实现（例如 Unix 系统中的档案文件）并不会并入一个编译单元的.o 文件，除非有一个显式的指向一个被该.o 文件解析的外部符号的引用。

考虑图 7-32 中所示的系统。ax_Registry（见图 7-29）是一个模块，它充当各种派生于协议类 ax_Record 的具体记录（例如 my_Record）的一个全局仓库。因为预期将有许多不同的记录子类型，所以使用一个特殊的助手类 ax_Registrar 在启动时帮着把具体记录类型自动加入全局注册表。组件 ax_registrar 在图 7-33 中介绍。

---

① 如果该静态工具的头文件被包含在任何静态实例（在这些实例的构造中使用了该工具）的非局部定义之前，则这是成立的。

② 注意，没有独立于实现的方法可用来强制一个编译单元中的非局部静态对象的运行时初始化顺序。因此，一个为静态结构的初始化排序的灵巧计数器的使用，隐含着所有客户（直接的或间接的）对可能是一个封装的实现细节的一个编译时依赖。在对一个非局部静态对象有一个特殊依赖的地方，我们通常可以通过完整地在.c 文件中手工仿真该非局部静态初始化来绝缘客户，如下所示：

```
// a.c
#include "a.h"
#include "b.h"

inline B& force() { static B b; return b; }

static B& dummy = force();   // Optionally, force initialization on any use.

A::A() {
    static B& b = force();   // Ensure b is instantiated before it is used.

    // Use b with confidence.
}
```

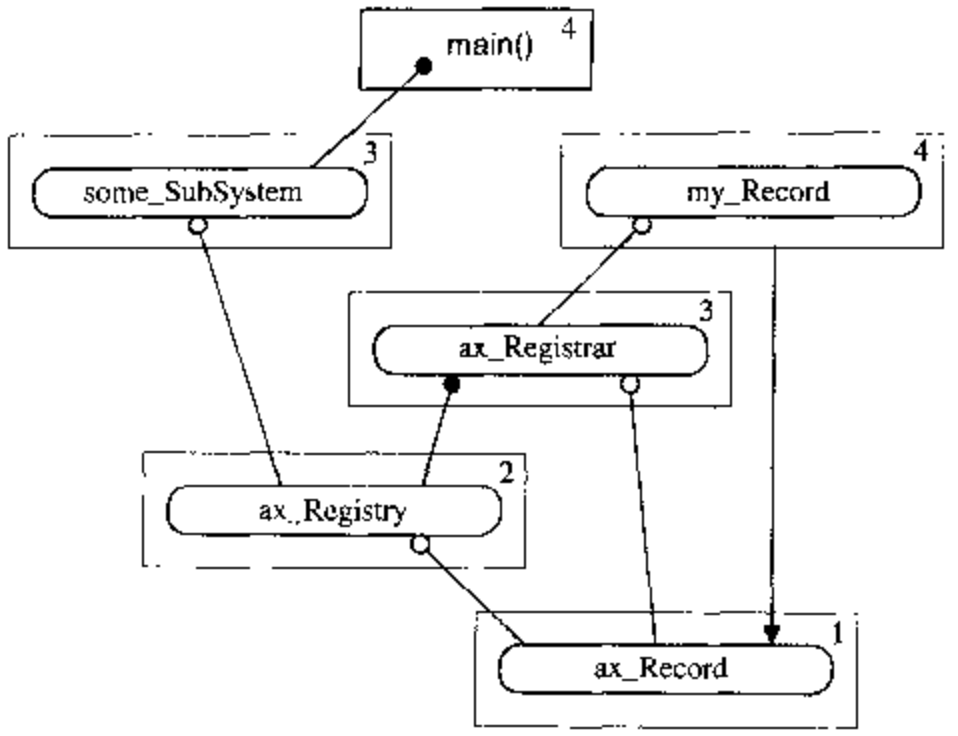


图 7-32　有自动初始化的系统的组件/类图

```
// ax_registrar.h
#ifndef INCLUDED_AX_REGISTRAR
#define INCLUDED_AX_REGISTRAR

class ax_Record;

struct ax_Registrar {
    ax_Registrar(ax_Record(*)());
    ~ax_Registrar();
};

#endif
```

```
// ax_registrar.c
#include "ax_registrar.h"
#include "ax_registry.h"

static int s_niftyCounter = 0;

ax_Registrar::ax_Registrar(ax_Record(*cfp)())
{
    ++s_niftyCounter;
    ax_Registry::add((*cfp)());
}

ax_Registrar::~ax_Registrar()
{
    if (--s_niftyCounter <= 0) {
        ax_Registry::cleanup();
    }
}
```

图 7-33　用来在启动时注册记录的 ax_Registrar 对象

为了注册一个记录类型（例如 my_Record）的一个实例，ax_Registrar 类的一个非局部静态实例被定义在组件 my_record 的.c 文件中，如图 7-34 所示。只要将 my_record.o 连接进一个可执行映像中，就足以保证它被登记在了该系统的全局记录注册表中。但是如果 my_record.o 是一个 Unix 库档案的一部分，在连接时就没有显式的引用将它收进来。也就是说，连接到定义在一个.o 文件集合中的具体记录会像预期的一样有效，但是，除非被显式地引用，否则连接到定义在一个 Unix 库档案中的同样的对象将会**没有效果**；启动之后全局注册表将是空的！

```
// my_record.h
#ifndef INCLUDED_MY_RECORD
#define INCLUDED_MY_RECORD

#ifndef INCLUDED_AX_RECORD
#include "ax_Record.h"
#endif

class my_Record : public ax_Record {
    // ...
  public:
    static ax_Record *create();
    my_Record();
    // ...
};

#endif
```

```
// my_record.c
#include "my_record.h"
#include "ax_registry.h"

static ax_Registry s_dummy(&my_Record::create);

ax_Record *my_Record::create()
{
    return new my_Record;
}

// ...
```

图 7-34 使用 ax_Registrar 来注册 my_Record

如果具体记录对象驻留在这样一个库档案文件中，那么必须有某种显式的连接时依赖才能将它们收进来。一个解决办法是提供一个空的非内联 init 函数供 main 调用。但是我们可以通过将注册过程升级到一个更高的层次（例如 main）来避免派生记录对象对注册表的依赖。这样，我们既可以提高灵活性也可以减少 CCD。图 7-35 显示了修改过的使用显式初始化的体系结构。

确定哪些特殊的派生记录类型要被并入一个给定的执行文件，（这项工作）必须在某处进行。适当的类型可以由定义 main 的组件显式地安装，或者在程序外部通过配置管理安装。一种表面上一流的初始化技术在被合并进某些库时不能正确工作的事实，强调了物理设计的重要性。

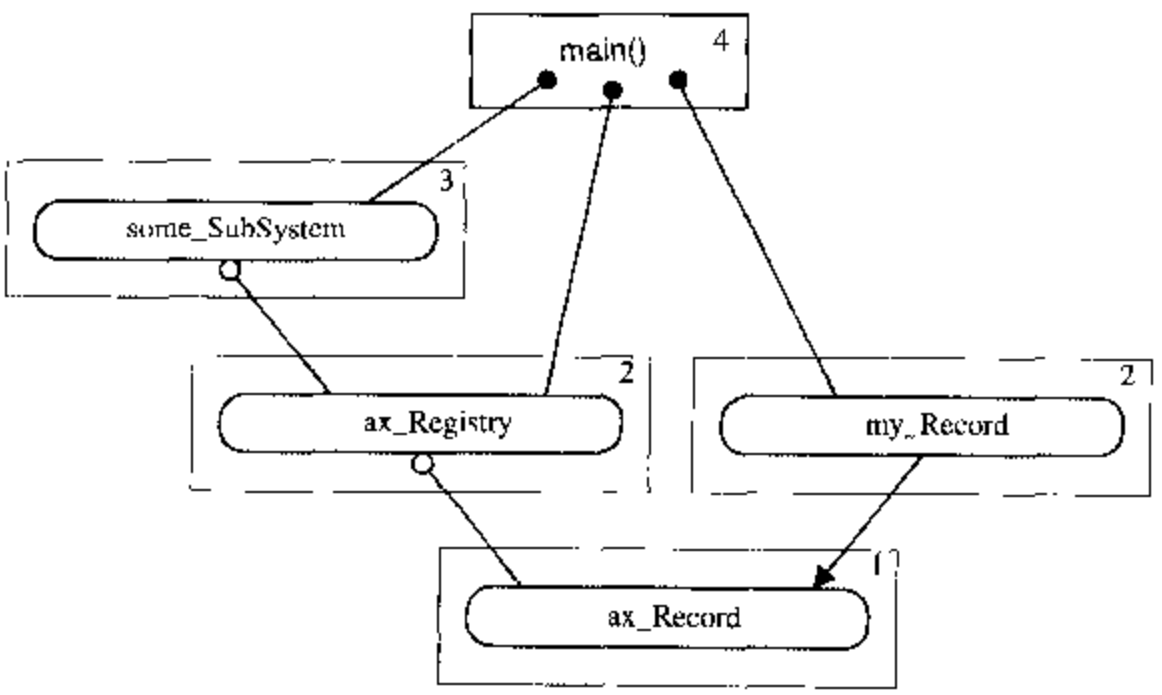


图 7-35　使用显式初始化的系统的组件/类图表

7.8.1.4　每次检查技术

程序越大，我们就越不可能使用它提供的所有功能。较少使用的子系统也许在运行时仍然需要大量的工作来初始化。在绝缘的情况下，如果一个组件的每个函数调用都已经执行了一个并非微不足道的任务，那么在每个调用上增加少量的额外运行时开销可能并不会显著影响运行时性能。

使用图 7-36 中所示的每次检查技术，我们不需要显式地初始化组件。相反，我们务必确保，依赖于内部静态构造的组件内的每个函数首先检查是否该组件已被初始化；如果没有，则立刻初始化该组件。每次检查方法的优点在于它也是极简单的（对于客户，用任何方式），并且初始化不需要在启动时出现。通过将初始化推迟到需要的时候，我们减少了启动时间，在运行时只为我们所使用的东西付出代价。iostream 使用这种技术来为流对象在其第一次被使用时分配缓冲区空间（见 10.3.3 节）。基于每个函数调用检查的缺点是，对于大量被使用的轻量级对象，这种方法通常不实用。（此外这种方法还要求）我们必须记住，无论何时将一个新的函数加入到我们的组件中，都要包含这个初始化检查。

7.8.2　清理（clean-up）

通常退出程序就足以完成普通用户所要做的事情了；但是作为负责任的开发者，我们必须始终考虑设计的易测试性。很多种方法可以检验我们的代码没有“泄漏”内存；但是，有时候很难将不明确地占据内存和一个真正的泄漏区别开来——尤其是在回归测试中。一些结构，例如多重继承，导致动态分配的内存由一个并未指向分配块起始位置的指针来管理，使得甚至复杂的工具都难以把合法使用与内存泄漏区分开来。

```
// ax_ledger.h
#ifndef INCLUDED_AX_LEDGER
#define INCLUDED_AX_LEDGER

class ax_Record;

class ax_Ledger {
    // ...
  public:
    static int addRecord(const Record& record);
    static void cleanup();
    // ...
};

#endif
```

```
// ax_Ledger.c
#include "ax_ledger.h"
// ...

static s_initFlag = 0;

static void init()
{
    s_initFlag = 1;
    // initialize component's static constructs
}

inline void s_checkInit()
{
    if (!s_initFlag) {
        init();
    }
}

int ax_Table::addRecord(const Record& record)
{
    checkInit();
    // now go ahead and add a record
}

void ax_Table::cleanup()
{
    // clean-up component's static constructs
    s_initFlag = 0;
}

// ...
```

图 7-36 每次检查，如果有必要就初始化

主要设计规则

提供一种机制来释放分配给一个组件内静态结构的任何动态内存。

通过给每个包含有可能隐匿动态分配内存的对象或静态变量的组件提供一个清理函数，有助于减轻探测内存泄漏的负担。将这个要求介绍为一条主要设计规则是因为，一个不遵守这条规则的组件可能影响需要它的任何组件的易测试性。

灵巧计数器方法的一种亦好亦坏的特性是，伪对象的析构函数可以用来自动初始化一个静态结构的清理。这对质量保证是好消息，但是会给更喜欢简单退出（因为性能原因）的用户造成一种负担。幸好，我们总是可以提供一个“开关”，用于控制清理是否真的在程序退出时出现。提供这种额外的清理能力的好处是，它是一种额外的品质度量；惟一真正的开销是额外的开发时间和接口上的少量额外复杂度。

7.8.3 回顾

总结：在启动时初始化模块和非局部静态对象，可能使启用一个大型程序的时间长得无法接受。虽然我们不能影响这些静态实例在程序中被初始化的时间点，但总是有可能将一个对象的单一全局实例转换成一个模块。一种确保没有运行时开销的初始化的有效方法是，通过使其只有基本的静态数据成员（它们在装载时被初始化），将模块或组件设计成唤醒初始化。减少启用时间的另一种方法是推迟初始化，直到真正需要时再初始化。这种被推迟的初始化可以用独立的 init 函数或将初始化检查嵌进每一个访问来实现。init 函数方法是最灵活的，也是最容易引起错误的，但是当独立访问函数是轻量级的并被频繁调用时，它可能是必要的。当试图连接存储在一个 Unix 类型的库中的自我初始化组件（对它们没有明确的连接时依赖）时，显式的初始化也是必要的。每次检查方法对客户来说是极简单的，尤其适合于当每个函数调用所做的工作量已经较大的时候。最后，如果我们知道我们很可能需要一个在启用之后立即初始化的组件，它的函数是轻量级的且被频繁调用，那么灵巧计数器方法也许是最好的选择。在所有情况下，提供一个机制来释放由静态结构拥有的任何动态内存（在程序退出之前）将有助于对内存泄漏进行回归测试。

7.9 小结

在本章中，我们将包的概念定义为物理设计的一个聚集内聚单位。包是自顶向下设计的自然结果，它既充当了体系结构设计者的一个抽象，又充当了开发者的一个分区。每个包都由协同操作组件的一个非循环层次结构组成。包中的每个组件的文件名，以及定义在那个组件中的每个全局结构，都应该以分配给那个包的注册前缀开头。该前缀的主要用途是标识一个给定的组件或类的定义可以在哪个包中找到。包前缀的一致使用分隔了全局名称空间，避

免了包集成时的名称冲突。

包内组件之间的依赖形成一个可层次化的层次结构。为了分层次地测试包内的组件，假设被包含在其他包内的组件是正确的并且每个外部组件都有一个相对于局部组件的 0 层次号。同一包内的不依赖其他组件的组件，被定义为有一个局部组件层次号 1。但是，如果包边界被移走，这些组件不一定有绝对的层次 1。系统内不依赖于任何其他组件的局部组件（称为叶子组件）有一个局部的和绝对的组件层次 1。将这些叶子组件放在使用它们的包内，有助于增进系统的模块性和可重用性。

包间依赖由组成这些包的组件之间的单个依赖的封套（envelope）定义，因为有关开发、推销、易用性、产品和可靠性方面的原因，要求包间的聚集依赖是非循环的。有着非循环依赖的包形成了一个可层次化的层次结构，完全类似于组件层次化。大部分在第 5 章讨论的减少单个组件之间的耦合的技术都可应用于作为整体的包。特别是，一般使用升级、降级和分解来减少和包间依赖有关的开发开销。

包级的绝缘包括减少为了客户使用该包必须输出的头文件的数量（或大小）。将一个包的客户与包含在该包内的一个特定组件绝缘，要求该组件本身不直接被该包（作为一个整体）的外部客户使用，所有使用了这个组件的被输出组件都将它的定义与外部客户绝缘，并且这个单独的组件没有被其他包独立重用。无论何时我们将客户和一个子系统的基础复杂性绝缘，我们就很可能已经改进了它的易用性和可维护性。

就像包将一个系统划分成可层次化的组件层次结构一样，包群将一个大型系统划分成可层次化的包层次结构。适用于单个包的组成和它们之间的相互依赖的原理，例如逻辑内聚和避免循环依赖，也同样适用于作为整体的包群。单个包的库文件可以合并成一个单个的群库文件，以方便较高层次的客户；但是，单个包库的专门装备的版本在开发过程中应该持续保持外部可访问性。用在一个系统中的每一个包都必须持续拥有一个惟一的、与任何包的分组无关的关联前缀。

内部发布是任何大型开发项目的一个必要组成部分。7.6.1 节介绍了一种能够支持分版本发布的目录结构。超大型系统可以划分成水平的包群联合（称为**层**）。一个层对应于一个给定层次上的所有群。一个可层次化的系统可以分阶段发布，从最底层（群层次 1）开始，进而到更高层次群。为了为我们的客户改进绝缘、抽象和编译时性能，我们可以选择只输出所有头文件中的一个子集（编译一个给定的包、群或层所需的）。

补丁是对前一次软件发布版本的一个局部修改。打一个补丁一般要比重新发布整个系统更便宜，也具有少得多的破坏性。我们为一个发布打补丁的能力和实现细节与客户的绝缘程度直接相关。7.6.2 节介绍了可能可以和可能不可以用补丁来实现的修改类型。

一个大型的用 C++编写的软件系统通常没有“顶”——没有单个的定义该系统的程序。main 的用途只是提供一个命令行接口、解释环境变量和管理全局资源。

将 main 提供的基础功能分解成独立可测试和重用的组件，有助于集成进更大型的子系统。只有.c 文件 main 可以采取单方面的全局行动；没有定义 main 的组件应该避免可能危及

随后的集成过程安全的利己主义行为。

启动被定义为从一个程序被调用到控制线程进入 main 的时间。正是在这个阶段，定义在整个程序的所有非局部静态对象被构造。天真地忽视这样的初始化可能导致启用时间长得不可接受。在 C++中，一个模块可以实现为一个只包含静态成员的类，并且好过实现为一个非局部静态实例，尤其是在初始化开销较大和对该对象不是立即需要的情况下。

本章还介绍了四种不同的初始化静态结构的技术：

（1）唤醒初始化：静态数据成员是一个基本类型的静态数据成员，它可以在装载时被初始化。

（2）显式的 init 函数：一个组件的 init 函数必须在该组件被使用之前显式调用。

（3）灵巧计数器：定义在组件头文件中的一个伪对象的一个静态实例保证了使用前初始化（在启动时）。

（4）每次检查：在需要时即时初始化（即在该组件中的任何函数被调用的第一时间）。

初始化技术的选择要取决于若干因素，包括：

- 该组件的“重量”
- 是否以及何时该组件可能被使用
- 初始化该组件本身的开销

对内存泄漏的有效回归测试要求我们提供一种方法来释放和静态结构有关的动态内存，即使这个特性未被应用程序本身所要求。

第3部分

逻辑设计问题

到目前为止，我们的讨论基本集中在与物理设计相关的概念（如组件、分层、绝缘和包）上。尽管好的物理设计对于大型项目的成功来说非常关键，但是，任何开发团队在开发过程的早期都应该研究基础性的逻辑设计问题。

逻辑设计是比物理设计更为成熟和更易于理解的规则。因此，这一部分的介绍将采用不同的风格。我们将尽可能引用其他可获得的书籍来减少重复。本书的第 3 部分是关于如何进行组件高效逻辑设计的简明“参考手册”。

设计模式描述协同操作组件的可重用微构架单元。在大型软件系统中要应用到无数的设计模式。虽然这种层次的逻辑设计大部分超出了本书的探讨范围，但是，许多最常用的设计模式在相关书籍中均有介绍，并且很容易获取。

在本书的这最后一部分中，我们只探讨单个组件的设计和实现。C++提供了一个几乎是压倒性的逻辑设计空间。这种特别的自由，可能使寻找一种最优的设计比通过组件实现功能更复杂。因此，我们的目标是简化每个组件的接口和消除冗余的自由度，因为这种自由度使逻辑设计空间不必要地复杂化了。

在第 8 章，我们将从高层俯视组件设计——即从逻辑角度分析。我们将研究常见的封装概念，并且指出使整体封装可能变得代价过大的条件特点。我们将指出并比较一些辅助对象实现的不同方法，这些对象仅用在一个组件的实现之内。

在第 9 章，我们将把注意力集中于在将特定的行为转化为 C++运算符和成员函数的语法时，组件接口设计者所面临的大量问题。将特定的行为实现为一个成员还是自由运算符、是否将其设置为虚拟的、如何传递进一个特定的参数以及如何返回一个值，这些只是要研究的 14 个不同问题中的一部分。我们还将讨论在接口中使用各种不同整型数（如 short、unsigned、long）的结果。接着，我们还仔细研究有关特殊情况功能的问题，如转换运算符、编译器产生的行为以及（特别是）析构函数。

在第 10 章，我们将逐个介绍在大型系统环境中的对象实现者所面临的问题，我们的研究一方面关注性能，另一方面注重可靠性。着重讨论之处包括：个体成员数据的选择和排列以及个体函数的有效实现。本章的大部分篇幅用于对对象存储的有效定制管理进行深入分析。我们发现，在避免持续运行程序中的潜在内存占用问题时，对象特定的存储管理比常规的类特定的技术更有效。最后，我们将探讨在类属的、基于模板的容器类的上下文中进行存储管理时易犯的错误，并简要地将模板的适用性和设计模式进行了比较。

8

构建一个组件

一个单独的对象通常由于太小而不能捕获一个完整的概念。一个对象若要有效，需要有自由的运算符或者完全的友元类来捕获一个抽象的基本行为。**抽象**指相互协调以服务于某种有用目的的对象和函数的抽象规范。一个组件是规范的具体表示，因此，一个**组件**也是逻辑设计的建筑块（building blook）。

封装，和绝缘一样，有不同的程度。与完全封装相关的成本通常高得难以接受。有时我们可以通过进行近乎完全的封装获得相当好的性能收益，而不会在灵活性方面有任何真正的损失。如何以及何时进行这种权衡需要仔细的考虑。

一个组件偶尔需要在它的实现辅助（不准备给客户直接使用的）对象中定义和使用。C++提供了一些实现这种类的技术，每一种技术都有其优缺点。在大多数情况下，我们有充足的理由只选择这些方法中的一种，我们需要做的只是建立一个选择标准。在这一章中，我们将研究组件接口设计的若干高层方面，讨论既适于作为整体的组件也适于它包含的单个对象的功能的类型和数量。我们将描述与完全封装相关的成本特性，并提出降低它的方法。最后，我们将考察在一个组件中实现辅助对象的众多方法，并提供一个基于特定使用模式所强调的特性来作出实现选择的基本原理。

8.1 抽象和组件

在第 3 章，我们介绍了作为物理设计的原子单位的组件。放在一个组件的头文件内的所有内容都可以立刻得到。这种物理内聚性使组件（而不是类）成为设计的最小单元，它能够跨越可执行程序而独立地重用。

组件层也是进行详细逻辑接口设计的适合层次。当你作为一个使用者利用组件实现一个列志（list）抽象时（见图 6-19），你可能要使用 List 类本身所提供的功能以外的功能。例如，

编写一个简单的输出语句

```
cout << "list = " << list << endl;
```

需要使用一个自由运算符（即运算符 <<），它不是任何类的逻辑接口的一部分。ListIter 类提供了列志抽象固有的功能，然而这种功能不是由 List 类接口直接提供的。

定 义：抽象是完成一个共同目的的一组对象和相关行为的抽象规范。

根据 Stroustrup[①]，一个 **ADT（Abstract Data Type，抽象数据类型）**的适当定义是：一个单个对象的正式抽象规范。那么一个类（接口和实现）就是这个对象的具体规范。类似地，一个抽象也是一个抽象的规范，而组件是相似的具体规范。

原 则

一个类是一个 ADT 的具体规范；一个组件是一个抽象的具体规范。

换句话说，一个**组件**不是仅仅一个类型的实现，而是一个首尾一致的功能缩影，它作为一个整体，并包含我们称为**抽象**的东西。它是一个完整的抽象，定义了一个组件要实现的系统内的功能的有用逻辑分区，而不只是一个单个的 ADT。

8.2 组件接口设计

设计良好的组件接口在质量上包括几个方面[②]。最低要求是，接口必须足以使预期的客户能有效利用设计该组件来支持的抽象。考虑一个实现集合抽象的组件。以下能力：

（1）确定集合中的成员关系；

（2）在集合成员上进行迭代；

（3）从集合中删除一个指定成员。

对任意特定客户而言可能是必要的也可能不是必要的。然而，如果没有向集合增加成员的附加能力，那么这个组件对任何人几乎都是无用的。

原 则

- 私有接口应该是充分的（sufficient）。
- 公共接口应该是完整的（complete）。
- 类接口应该是基本的（primitive）。
- 组件接口应该是最小化和便于使用的。

① **stroustrup**，1.2.3 节，18~19 页。

② **booch**，3.6 节，136 页。

如果一个组件不准备公用，那么如 1.8 节所建议的，为其已知的固定客户集有效完成工作的最小功能子集需要定义充分。而另一方面，如果要在各种不同情况下、在整个系统中广泛地重用一个组件，那么我们不可能一定事先知道会需要功能的哪个子集[1]。一个**完整**的接口，应该使给定抽象的用户所共同期待的所有操作，能够以一种有效的方式完成。我们的客户越遥远，我们就越有可能因为要使接口尽可能包罗万象而在通用性方面出错[2]。

原　则

在任何可行的地方，延缓不必要功能的实现可以降低开发和维护成本，并且可以避免过早地进行精确的接口和行为设计。

通常，一个完整的接口比用以满足任何个人用户需要的接口需要更深入的实现策略，因而实现一个完整的接口将会有更多开销。一个通用的实现可能比一个专用的版本运行得更慢，即使在使用最基本、最常用的操作时可能也是这样[3]。因此，一个完整的接口在运行时可能会有更多开销。一个更完整的接口往往也更大更复杂，混合着一些不常用的特性。一个更大更复杂的接口也使用户更难寻找和使用一些基本的功能。因此，一个完整的接口使用起来更昂贵。由于一个完整的接口在各种度量标准中都更昂贵，所以，在实现一个完整接口前，先弄清楚这样做是否具有正当理由才是明智的做法。在充分和完整两个极端间有广阔的中间地带。例如，将一个迭代器的状态赋给另一个，这种操作在实践中几乎决不会出现。因此，一个迭代器的赋值运算符通常被声明为私有并且不实现，这并不会影响组件的可用性。这种有意的缺省减少了开发时间和代码长度，而且有可能在增加功能后不会导致现存的客户重新编制他们的代码。

定　义：如果有效实现定义在一个对象上的操作意味着可以直接访问该对象的私有部分，那么该操作是**基本的**（**primitive**）。

在为一个类的接口选择功能函数时，我们的目标是努力使该函数集最小化，以**基本性**（**primitiveness**）为标准。显然，增加或删除集合中的一个成员是独立的基本操作。在集合成员上进行迭代的能力使客户能够决定集合成员，这意味着成员资格认证本身不是一个基本的操作。然而，通过迭代决定成员资格，有可能从根本上比通过对内部表示进行直接私有访问

① **偶然的重用**意味着不在组件原本准备使用的情况下使用组件。**有意的重用**意味着（值得一提）组件作者的一个愿望——提供一个完备的接口和健壮的实现。如果你要连接一个作为“可重用”组件标准库的一部分的组件（例如 STL），你是在使用（use）它还是在重用（reuse）它？isotream 又如何？

② 见 **meyers**，Item 18，62 页。

③ 例如，基于模板的、当被任意的用户自定义类型参数化时必须正常工作的容器类，不能与专为基本类型设计的容器一样，自由使用按位拷贝的例程（如 memcopy）（见 10.4.2 节）。

（如通过二进制搜索）低效得多。如果决定成员资格是项常用的操作，那么它几乎肯定是基本操作。

甚至一个潜在重要的性能益处也是将一个操作视为基本操作的正当理由。考虑图 6-9 的 String 类，我们不能确定在这个类中是否要有一个 d_length 成员。如果决定要，我们当然希望提供一个基本的 length()访问函数。如果决定不要，我们只需简单地使 length()函数将其调用随同低层表示传递给标准 C 库函数 strlen(const char*)。在后一种情况中并没有实际的性能收益；然而，除非提供 length()成员，否则当我们在这两种方法之间来回实验时，没有办法给予客户由每个实现所提供的最大利益。无论我们是否拥有一个 d_length 成员都决不应该迫使客户重新编制他们的代码；这样的考虑是微妙的封装艺术的一部分。

原　则

让功能保持在一个可行的最小范围内可以增强可用性和可重用性。

当为一个组件的接口选择函数时，我们的目标还是尽可能最小化，但也要着眼于可用性。在一个组件接口中为一个抽象提供每一个可能的操作，将增加其长度，打击其客户，削弱其可用性。例如，我们可能提供一个非基本的支持，以取代图 3-2 中的 Stack 的顶端入口。尽管对少数客户来说潜在可用，但大多数人会觉得这样的功能是不必要的。

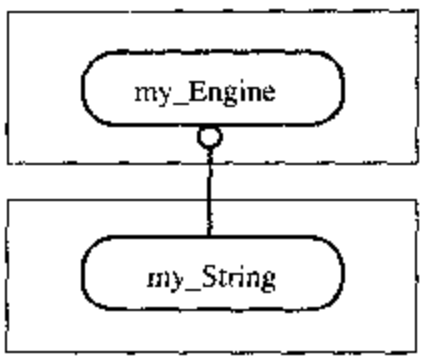
按照相同的理由，我们也可能在图 3-2 的 stack 组件中忽略等同性测试。由于这些测试被实现为自由的非友元函数，operator==和 operator!=可以改为由需要它们的开发者来实现。但是，如果许多用户正在开发将在一个大型系统中协同工作的应用程序，理想的做法是，避免让每个用户重写各个子系统内的相同的函数。这种冗余会浪费开发时间、可执行程序的大小及相应的执行时间。寻找合适的非基本功能并添加到一个组件中以使其最有用是一个设计目标，通常实现这个目标的最小接口是最好的接口。

原　则

在一个组件接口中尽可能少地使用外部定义类型，可以促进在更多情况下的重用。

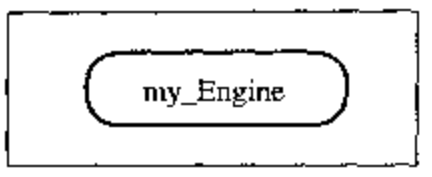
耦合这个术语对逻辑设计和物理设计都适用。物理耦合来源于将逻辑实体放在相同的组件中，或由于建立了一个组件对另一个组件的物理依赖关系而产生。逻辑耦合起源于在一个组件接口中使用由其他的组件定义或提供的类型。和物理耦合一样，逻辑耦合也最好保持最小。减少逻辑接口中使用的外部类型数量，通常可以使一个组件更易于使用和维护。

假如你正在建立一个公共的接口并需要接受字符串输入，你认为图 8-1 中的哪个接口更通用？你的客户可能有他们自己习惯使用的字符串类，每一个通用目的字符串类都将知道如何产生一个 const char*表示。图 8-1（a）的接口将迫使你的客户使用 my_String 类；而图 8-1（b）的接口则不会。




```
// my engine.h
#ifndef INCLUDED_MY_ENGINE
#define INCLUDED_MY_ENGINE

class my_String;

my_Engine {
    // ...
  public:
    my_Engine(const my_String& name);
    // ...
    void setName(const my_String& name);
    // ...
    const my_String& name() const;
    // ...
};

#endif
```

（a）在接口中使用 my_String




```
// my_engine.h
#ifndef INCLUDED_MY_ENGINE
#define INCLUDED_MY_ENGINE

my_Engine {
    // ...
  public:
    my_Engine(const char *name);
    // ...
    void setName(const char *name);
    // ...
    const char *name() const;
    // ...
};

#endif
```

（b）在接口中使用 const char*

图 8-1　避免逻辑耦合

如果我们改为选择依靠一些其他的非标准组件库模型（如接口中的 your_String），这种逻辑耦合形式的结果可能会更严重。在一个 ANSI/ISO 标准的字符串组件普遍可用之前，在一个应用广泛的接口中，选择 const char*（而不是任何其他的）字符串表示仍然是有利的。

简而言之，当我们设计一个组件接口时，我们需要问自己许多高层次的问题，最重要的问题是“组件的公共程度如何？”如果它将以许多不同和不可预测的方式重用，则它需要有一个具备合理的完整性的接口。如果这个组件是准备在一个包内私有使用的（不会被输出），那么接口只需充分就可以了。在所有情况下，如果我们将接口设计为只包含基本的功能，去掉那些有用但非基本的功能，将它们分解为独立的、非私有访问的运算符或类，就可以提高我们所设计的类的可维护性。最后，逻辑耦合常常产生不受欢迎的物理耦合；避免在组件接口中使用在该组件之外定义的、不必要的类型有助于减少这种耦合。

8.3　封装程度

封装看上去容易实现，其实不然。和完全绝缘一样，完全封装在运行时的代价也可能是非常高的。

图 8-2 是一个糟糕的封装例子[1]。它不是接受一个参数，而是每个操纵函数都返回一个指向私有数据成员的可写引用。

```
// bad_point.h
#ifndef INCLUDED_BAD_POINT
#define INCLUDED_BAD_POINT

class bad_Point {
    int d_x;    // (may change to short later)
    int d_y;    // (may change to short later)

  public:
    // CREATORS
    bad_Point(int x, int y) : d_x(x), d_y(y) {}
    bad_Point(const bad_Point& p) : d_x(p.d_x), d_y(p.d_y) {}

    // MANIPULATORS
    bad_Point& operator=(const bad_Point& p) {
        d_x = p.x(); d_y = p.y(); return *this; }
    int& x() { return d_x; }     // bad idea
    int& y() { return d_y; }     // bad idea

    // ACCESSORS
    int x() const { return d_x; }
    int y() const { return d_y; }
};

#endif
```

图 8-2 糟糕的封装例子

图 8-3 显示了 bad_Point 接口的一个小的测试驱动程序。

```
// bad_point.t.c
#include "bad_point.h"
#include <iostream.h>

ostream& operator<<(ostream& o, const bad_Point& p)
{
    return o << '(' << p.x() << ", " << p.y() << ')';
}

main()
{
    bad_Point pt(1,2);
    cout << pt << endl;
    pt.x() = 5;
    cout << pt << endl;
}
```

图 8-3 糟糕的 bad_Point 接口的测试驱动程序

① 见 **meyers**，Item 30，100~102 页。

当对图 8-2 所示的例子运行这个驱动程序时，将产生如下输出（和预期一致）：

```
john@john: a.out
(1, 2)
(5, 2)
john@john:
```

但现在假设我们将 bad_Point 类中的私有数据成员类型从 int 改为 short：

```
class bad Point {
    short d x; // OK, we changed "private" data
    short d_y; // so what?

  public:
    // ...
```

并重新运行这个测试。现在的结果是未预料到的：

```
john@johr: a.out
(1, 2)
(1, 2)
john@john:
```

问题在于接口中返回的引用（int&）和返回的数据类型（short）不一致。结果创建了一个临时 int 类型变量，并返回了一个该临时变量的可写引用。我们可以将接口函数修改为返回一个 short&类型，但这样一来我们就因实现的变化而修改了接口——从而使这个问题蔓延到了我们的客户。

一个 bad_Point（如图 4-3 中的 geom_Point）的适当的封装接口版本应该把操纵函数定义为获取新的成员值作为参数：

```
Main()
{
    bad Point pt(1,2);
    cout << pt << endl;
    //pt.x() = 5;              // Returning writable reference replaced
    pt.setX(5);                // by function taking value of x coordinate.
    cout << pt << endl;
}
```

输出结果又和预期的一样了——不论数据成员声明为 int 还是 short 类型：

```
john@john: a.out
(1, 2)
(5, 2)
john@john:
```

原　则

对封装进行的好的测试，是要看一个给定的接口是否不需做任何改变即可同时支持两种显著不同的实现策略。

在 bad_Point 例子中，做到这一点不需要任何额外的代价；然而，在有些情况下，完全的封装有可能更昂贵。考虑如图 8-4 所示的 geom_Box 的两种潜在实现策略。实现（a）存储了盒子的左下角和右上角，作为嵌入在 geom_Box 中的点，因此可能通过 const 引用返回左下角和右上角。中心点没有存储，因此需要计算并通过值返回。同样，长度与宽度也必须实时计算。然而，实现（b）存储了中心点以及 geom_Box 的宽度和高度。中心点可由 const 引用有效地返回，而左下角和右上角必须计算并通过值返回。长度和宽度不需要计算，但作为基本类型，它们通过值返回效率最高。

```
class geom_Box {
    geom_Point d_lowerLeft;
    geom_Point d_upperRight;

  public:
    // ...
    const geom_Point& lowerLeft() const;
    const geom_Point& upperRight() const;
    geom_Point center() const;
    int width() const;
    int height() const;
};
```

（a）存储左下角和右上角

```
class geom_Box {
    geom_Point d_center;
    int d_width;
    int d_height;

  public:
    // ...
    geom_Point lowerLeft() const;
    geom_Point upperRight() const;
    const geom_Point& center() const;
    int width() const;
    int height() const;
};
```

（b）存储中心点、宽度和高度

图 8-4 geom_Box 类的两种实现策略

原 则

一个完全封装的接口可能会为给定的实现带来很大的性能负担。

一种实现相对于另一种实现的部分优点在于，避免了构造最频繁访问的点的开销，而是通过引用有效地返回它。然而，严格地讲，这两种接口尽管类似，在程序设计上却并不完全相同。例如，实现（b）中有可能取得中心点的地址，而在实现（a）中不行。要封装实现的所有方面，必须通过值返回全部三个点（即从左下、中央和右上），或传递进一个过去构造的（将要对其进行赋值的）点的地址。在 geom_Box 的例子中，一个完全封装的接口会消除一个实现方法相对于另一个实现方法的大部分性能优势。

一个不完全封装的典型例子实际上能在任何通用的串类中找到，这些串类为了效率，总是向其内部的、以 null 结束的字符串表示提供如 const char*的直接访问。显然这种接口约束了内部实现，只要串对象没有修改或删除，就要迫使它维持一个合法的、以 null 结束的串表示。但是，一个封装度更高的接口，往往因开销太大或太难使用而难以为人们接受。

接口出于效率而束缚了实现的另一个例子可以在一个无界数组抽象中找到，在这个数组抽象中返回了指向一个索引对象的可写引用。如图 8-5（a）所示的一个点数组，一旦引用了

一个 geom_Point 对象，这种类型的接口就迫使实现方法为该对象维持相同的空间。任何通过数组重定位对象的做法，都将使客户保持的引用无效。

比较起来，如图 8-5（b）所示，一个幼稚的、完全的封装版本将提供函数来获取和设置一个特定的元素。注意这个接口是完全通用的。我们可以自由地在内部将点存储为两个并行的整数数组。我们可以决定对点采用某种 in-core 压缩方法。我们甚至可以考虑将大数组的一部分转储到磁盘。

原　则

传递进一个过去构造对象的地址以赋给返回值（称为**参数返回**），能在保持整体封装的同时提高性能。

尽管这个新接口不会限制我们的实现选择，但使用这种完全封装接口的运行时开销可能会昂贵得多——甚至当这两个基础实现完全一致时也是这样。对于更重的元素（即大很多的、有非内联的拷贝构造函数或在构造时需要动态分配内存的元素），一个完全封装的接口版本在运行时的开销将高得令人不敢问津。

幸运的是，接口还有另外一种完全封装形式，它能为“更重”的对象减轻一些负担，特别是在访问它们的值时。通过值返回一个对象会引起至少一个索引类型的临时变量的构造（和析构）。如图 8-5（c）所示，我们可以传递进指向已有的对象的可写指针，代替通过值返回该对象。给已有对象赋值（仅一次），通常能够相对有效地完成。

```
class geom_PointArray {
    // ...
  public:
    // ...
    geom_Point& operator[](int index);
    const geom_Point& operator[](int index) const;
    // ...
};
```

图 8-5（a）　部分封装接口（Array A）

```
class geom_PointArray {
    // ...
  public:
    // ...
    geom_Point point(int index) const;
    void setPoint(const geom_Point& point, int index);
    // ...
};
```

图 8-5（b）　不成熟的完全封装的接口（Array B）

```
class geom_PointArray {
    // ...
  public:
    // ...
    void getPoint(geom_Point *returnValue, int index) const;
    void setPoint(const geom_Point& point, int index);
    // ...
};
```

图 8-5（c） 替代的完全封装接口（Array C）

为了使这些都具体化，我开发了一个 PointArray 类的单一试验版，同时包含了所有这三种可用的访问模式。图 8-6 的内容放在驱动文件的头部，用以比较这三种操作模式的相对性能。

```
// pointarray.t.c
#include "point.h"
#include <memory.h>                    // memcpy()

class PointArray {
    Point **d_array_p;                 // array of pointers to Point objects
    int d_size;                        // current physical size of "unbounded" array
    Point d_dummy;                     // not static to avoid construction at startup

private:
  void resize(int maxIndex);  // extend array of Point pointers when needed

  PointArray(const PointArray& array);                  // not implemented
  PointArray& operator=(const PointArray& array);       // not implemented

public:
  // CREATORS
  PointArray(int size) : d_array_p(0), d_size(0), d dummy(0,0)
  {
      resize(size - 1);                                 // Factoring is good.
  }

  ~PointArray();

  // MANIPULATORS
  Point& operator[](int index)                          // ARRAY A
  {
      if (index >= d_size) {
          resize (index);
      }
      return *d_array_p[index];
  }

  void setPoint(const Point& point, int index)          // ARRAY B & C
  {
      if (index >= d_size) {
          resize (index);
```

```
    }
    *d_array_p[index] = point;
}

// ACCESSORS
int size() const { return d_size; }                    // ARRAY A, B, C
    const Point& operator[](int index) const           // ARRAY A
    {
        return index >= d_size ? d_dummy : *d_array_p[index];
    }

    Point point(int index) const                       // ARRAY B
    {
        return index >= d_size ? d_dummy : *d_array_p[index];
    }

    void getPoint(Point *returnValue, int index) const // ARRAY C
    {
        *returnValue = index >= d_size ? d_dummy : *d_array_p[index];
    }
};

PointArray::~PointArray()
{
    for (int i = 0; i < d_size; ++i) {
        delete d_array_p[i];
    }
    delete [] d array_p;
}

void PointArray::resize(int maxIndex)
{
    int newSize = maxIndex + 1;
    Point **p = d_array_p;
    d_array_p = new Point *[newSize];
    memcpy (d_array_p, p, d_size * sizeof *p);
    delete p;
    for (int i = d_size; i < newSize; ++i) {
        d_array_p[i] = new Point(0,0);
    }
    d_size = newSize;
}
```

图 8-6　实验性的 PointArray 类的实现

第一个试验是比较数组前 1000 个点的 x 坐标的相对读取效率，并在一个变量中对值进行累加。图 8-7 所示的三个数组接口都进行了这个试验。为了证明对象“重量”对接口的影响，也在这里重用了图 6-83 试验中使用的三个不同的 Point 实现[①]。

① 该图提供的数据代表了完全优化的代码。

```
main()
{
    int arraySize = 1000;
    int sum = 0;
    PointArray array(arraySize);
    const PointArray& constArray = array;  // Provide a const reference to
    // ...                                 // enable the invocation of the
                                           // const version of operator[]
// INTERFACE A:
    {
        for (int j = 0; j < arraySize; ++j) {
            sum += constArray[j].x();
        }
    }

// INTERFACE B:
    {
        for (int j = 0; j < arraySize; ++j) {
            sum += constArray.point(j).x();
        }
    }

// INTERFACE C:
    {
        Point pt(0,0);
        for (int j = 0; j < arraySize; ++j) {
            constArray.getPoint(&pt, j);
            sum += pt.x();
        }
    }
}
```

图 8-7 PointArray 的试验性的“读”驱动程序

图 8-8 提供了在同一数组中访问 Point 对象的这三种不同接口风格的比较结果。使用最初的 Point 类（行 1）（其所有函数都声明为内联函数），对于数组 B 的不成熟封装来说，完全封装的开销仅稍多一点（111%），对于数组 C 的完全封装而言，完全封装不产生新的开销（100%）。从被包含的 Point 类型中删除内联函数（行 2）将使构造及赋值 Point 对象的开销都要更大一些。数组 C（168%）相对于数组 B（271%）的部分运行时优势在于，Point 对象的赋值在每次数组访问时严格地只发生一次，不需要额外的、一般以值返回一个对象所需的构造函数（和析构函数）调用。对于一个被包含的、在构造时动态分配内存的对象（行 3），构造的开销（1673%）远远超过了在适当位置赋新值的开销。从这些数据我们可以得出这样的结论：如果我们通过参数表而不是值来返回重量级的对象，完全封装的类在性能上将有很大的收益。

第二个试验是在保持 y 坐标不变的情况下，比较数组前 1000 个点的 x 坐标的相对设置效率。注意接口 A 允许我们直接完成这个操作，而接口 B 和接口 C 迫使我们必须得到整个点的当前值。我们对三个数组接口和三个 Point 实现中的每一个都进行了图 8-9 所示的这个试验，

结果列在图 8-10 中。

行	Point 类的描述	部分封装 数组 A	完全封装 数组 B	也是完全封装 数组 C
1.	初始的 Point 类（轻量级）	0.222 （100%）	0.247 （111%）	0.222 （100%）
2.	不含内联函数（中等）	0.296 （100%）	0.802 （271%）	0.497 （168%）
3.	完全绝缘版（重量级）	0.396 （100%）	0.622 （1673%）	6.173 （169%）

在 SUN SPARC 20 上以毫秒计算的循环时间（与数组 A 用时的百分比）

图 8-8　访问一个 PointArray 元素的相对开销

```
main()
{
    arraySize = 1000;
    PointArray array(arraySize);
    PointArray& nonConstArray = array; // provide non-const reference.
    // ...

// INTERFACE A:
    {
        for (int j = 0; j < arraySize; ++j) {
            nonConstArray[j].setX(j);
        }
    }

// INTERFACE B:
    {
        for (int j = 0; j < arraySize; ++j) {
            nonConstArray.setPoint(Point(j, nonConstArray.point(j).y()), j);
        }
    }

// INTERFACE C:
    {
        Point pt(0,0);
        for (int j = 0; j < arraySize; ++j) {
            nonConstArray.getPoint(&pt, j);
            nonConstArray.setPoint(Point(j, pt.y()), j);
        }
    }
}
```

图 8-9　PointArray 的试验性“写”驱动程序

行	Point 类的描述	部分封装 数组 A	完全封装 数组 B	也是完个封装 数组 C
1.	初始的 Point 类（轻量级）	0.162	0.403	0.336
		（100%）	（249%）	（226%）
2.	不含内联函数（中等）	0.242	1.451	1.118
		（100%）	（503%）	（418%）
3.	完全绝缘版（重量级）	0.385	12.901	6.669
		（100%）	（3551%）	（1732%）

在SUN SPARC 20上以毫秒计算的循环时间(与数组A用时的百分比)

图 8-10 访问一个 PointArray 元素的相对开销

基于这个试验的结果，我们可以得出这样的结论：提供一个可写的、指向被包含对象的引用能很好地改善性能，这种改善会随着对象的增大而戏剧性地增加。对于完全绝缘的类，通过参数表返回初始值能在某种程度上减轻负担。

原 则

接受不太完全的封装有时是正确的选择。

完全封装的接口应该是一个设计目标。然而，如果性能也是设计目标，那么不论封装程度如何，都应当取消过分的实现选择。通过合理的假设，我们可以获得更好的性能而又保持我们需要的灵活性，以便在适当的限制中修改我们的实现。

作为最后的旁白，我要提及一个微妙的问题，这个问题涉及数组的非封装版本接口。[ ] 运算符有两种版本：

```
operator[](int index)
```

和

```
operator[](int index) const.
```

第一个运算符能潜在地改变数组的大小；第二个则不能。如果某个数组被实现为“稀疏”数组，一个 Point（或大很多的对象）的空间将被有意不予分配，直到被 operator[ ]的 non-const 版本引用。使用这个接口，仅仅“读”一个 non-const 数组对象的行动，就将隐含地填充（populate）它。不用运算符重载而是为这些运算符选择一个完全不同的名称会切实可行得多。这样做能使数组极不易于被微妙地错误使用而导致内存过量分配。

总结：封装是进行良好面向对象设计的基石；它的目标是将客户对跨越接口的实现细节的逻辑依赖最小化。封装越完全，实现者在他们的实现选择上就越有灵活性。然而，与绝缘类似，对一个非常低级的对象的完全封装在运行时的开销可能会非常大。

如果将性能作为设计目标，那么某些实现选择（如：对象压缩和磁盘转储）无论如何必须放弃。通过合理的假设并结合已有的经验，我们可以获得封装的大多数好处同时避免过多和不必要的运行时开销。当合适完全封装时，我们可以通过传递进一个先前构造的对象来装载而不是用值返回对象，有时这样能减少运行时开销。

8.4 辅助实现类

通常一个组件会在其实现中使用一个或多个小的辅助类，它们在组件中定义的主类的接口中是不能编程访问的。辅助类有两个区别于其他类的特征：

（1）设计辅助类是为了实现一个组件的单一目的，并且不让它在该组件外直接使用（或重用）。

（2）辅助类很小，并且可能不必直接测试。

图 6-19 所示的列表组件的 Link 类就是这样一个例子。有许多方法可以用来实现这样的实现类，它们各有优缺点。在这一节，我们将探讨一些设计选择的优缺点。

考虑如图 8-11（a）所示的一个简单的整数列表类。my_Link 类是 my_List 的实现细节，并且不能从 my_List 编程访问。在该实现中，辅助类定义被放在定义主要类的组件的头文件中。这个直截了当的做法是用辅助类实现组件的最简单和最常用的方法。

如图 8-11（b）所示，我们可以将 link 类放在它自己的组件中。这种安排的好处是允许我们独立于 my_List 测试（甚至重用）my_Link。但对于像 my_Link 这样的极小的实现类来说，由重用带来的耦合和第二个组件的额外物理复杂性使其成为不可能的选择。

我们可以将 my_List 声明为 my_Link 类的一个友元，并将该 link 类的所有函数都声明为私有，如图 8-11（c）所示。将 my_Link 作为 my_List 的“隶属”类，可以阻止 my_list 组件的客户直接使用 my_Link；然而，这样做也阻止了为了直接测试而进行的访问。

如图 8-11（d）所示，我们可以使 my_Link 类成为一个局部定义，使其完全包含在.c 文件中。这种设计能将客户程序与 my_Link 绝缘。但是，除了阻止直接测试之外，这种设计也阻止内联 my_List 的任何成员（它们实质使用了 my_Link）。如果 my_list 组件也包含了一个迭代器，那么不能内联迭代器函数也许会极大地降低运行时性能。

最后，如图 8-11（e）所示，我们可以使 my_Link 类成为私有的（或公共的）、嵌入的类，这种类的定义完全包含在 my_List 类中。这种实现不会将客户程序和 my_Link 的细节绝缘，但它允许在 my_List（和 my_ListIter）的内联成员体中使用 my_Link 的成员。使 my_Link 成为嵌入式的类避免了对全局名称空间的影响；将其设置为私有会导致其被封装，并因此而不能直接使用（或测试）它。

本节提供的 my_Link 类的各种实现选择方案的优点在图 8-12 中进行了总结。将 my_Link 放置在一个单独的组件中（实现 B）显然是最灵活的，它允许组件的开发者将辅助类的定义按需要包含在主要组件的.c 文件或.h 文件中。但是，这样做会带来一种与系统每一物理部分相关的开销。除非我们计划直接测试或独立重用辅助类，否则开发一个单独的组件来包含它可

能不太合理。

```
// my_list.h
#ifndef INCLUDED_MY_LIST
#define INCLUDED_MY_LIST

class my_Link {
    int d_data;
    my_Link *d_next_p;

  public:
    // ...
};

class my_List {
    my_Link *d_head_p;
    // ...
  public:
    // ...
};

#endif
```

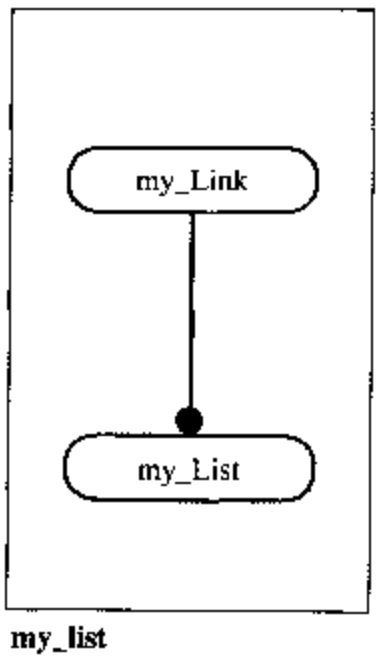


图 8-11（a） （最初的）my_Link 的文件作用域实现（A）

```
// my_list.h
#ifndef INCLUDED_MY_LIST
#define INCLUDED_MY_LIST

class my_Link;

class my_List {
    my_Link *d_head_p;
    // ...
  public:
    // ...
};

#endif
```

```
// my_link.h
#ifndef INCLUDED_MY_LINK
#define INCLUDED_MY_LINK

class my_Link {
    int d_data;
    my_Link *d_next_p;

  public:
    // ...
};

#endif
```

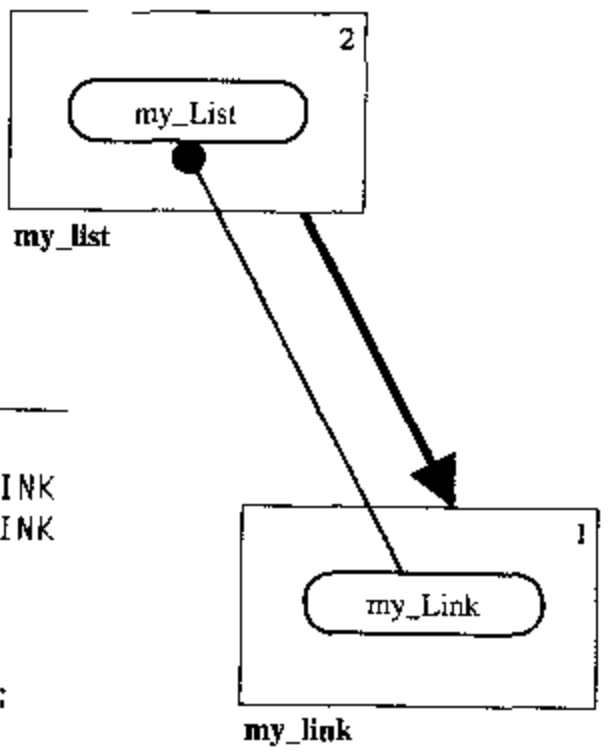


图 8-11（b） my_Link 的单独组件实现（B）

```
// my_list.h
#ifndef INCLUDED_MY_LIST
#define INCLUDED_MY_LIST

class my_List;

class my_Link {
    int d_data;
    my_link *d_next_p;
    friend my_list;
    // ...
};

class my_list {
    my_Link *d_head_p;
    // ...
  public:
    // ...
};

#endif
```

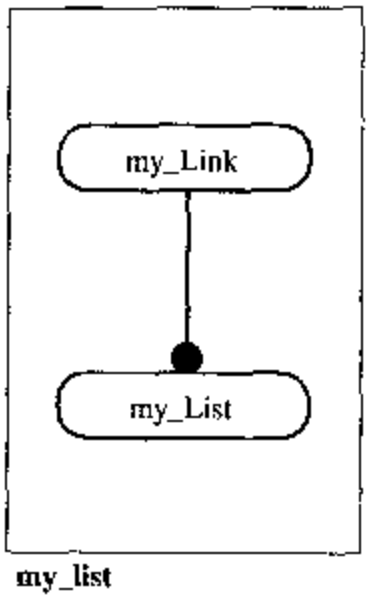


图 8-11（c） my_Link 的隶属类实现（C）

```
// my_list.h
#ifndef INCLUDED_MY_LIST
#define INCLUDED_MY_LIST

class my_Link;

class my_List {
    my_Link *d_head_p;
    // ...
  public:
    // ...
};

#endif
```

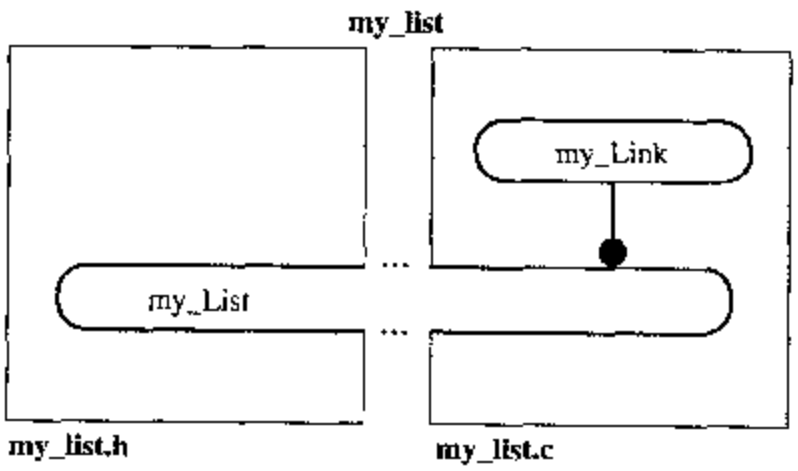


图 8-11（d） my_Link 的局部类实现（D）

嵌入式类不像定义在文件作用域中的类那样灵活。例如，嵌入式类不能前置声明[1]；这样，嵌入式类就不能与它们的外壳类（enclosing class）的客户绝缘。另外，嵌入式类型在表示法上相当不方便，会造成物理接口过分凌乱。如果我们后来决定绝缘或重用它，那么嵌入式实现的语法会阻止我们方便地将辅助类迁移到.c 文件或其他组件中去。

① ANSI/ISO 委员会已经采纳了在 C++中允许嵌入式类的前置声明的建议。见 **stroustrup94**，13.5 节，289~290 页。

```
// my_list.h
#ifndef INCLUDED_MY_LIST
#define INCLUDED_MY_LIST

class my_List {
    class my_Link {
        int d_data;
        my_Link *d_next_p;

      public:
        // ...
    };

    my_Link *d_head_p;
    // ...
  public:
    // ...
};
```

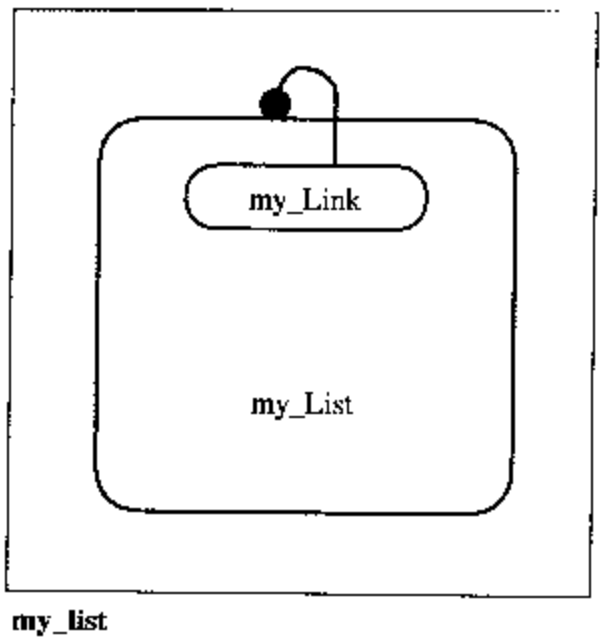
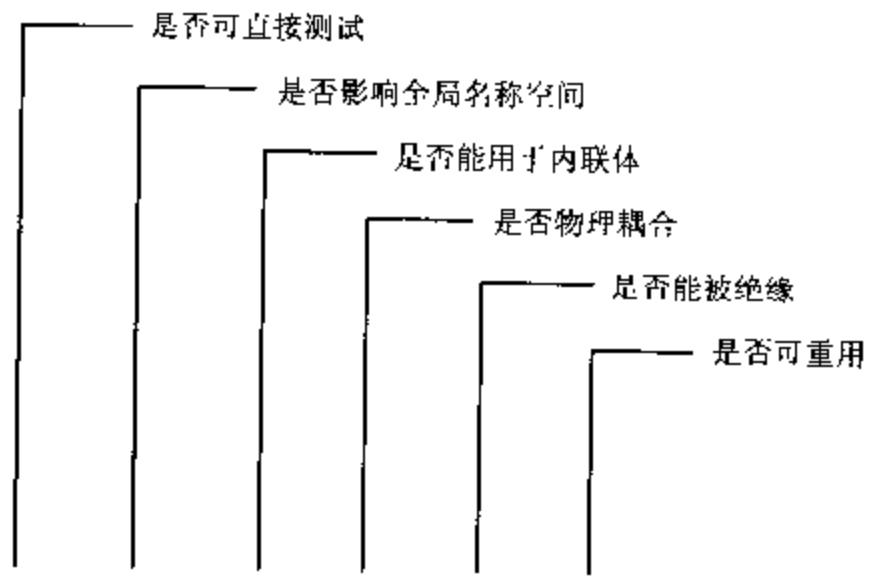


图 8-11（e） my_Link 的嵌入式类实现（E）



（A）文件作用域（初始的）	是	是	是	是	否	否
（B）单独的组件	是	是	是	否	是	是
（C）隶属类	否	是	是	是	否	否
（D）局部类	否	否*	否	是	是	否
（E）嵌入式（私有）类	否	否	是	是	否	否
（E'）嵌入式（公共）类	是	否	是	是	否	否

* 假设类不涉及有外部连接的构造

图 8-12 不同辅助类实现的优势总结

初始的实现（A）和公共的嵌入式实现（E'）有相似的特性。嵌入式公共设计的一个优点是它不会影响全局名称空间。考虑前面描述的嵌入式类的缺点，如果你要使一个嵌入式类成为公共的，为什么不能只是在它的名称前面加上前缀并在.h 文件的文件作用域中定义它（像

实现 A 那样）？

尽管没有绝缘，但私有嵌入式类是真正被封装了，并且不能被主对象的客户访问，它们也不能被直接测试。除了隶属类本身是名称空间的一部分外，隶属类实现（C）几乎和私有嵌入式类实现（E）完全相同，尽管也不能直接测试或重用。除了局部类与客户绝缘并因此不能在主类的内联函数体中使用之外，局部类实现（D）也和私有嵌入式类实现（E）类似。关于没有外部连接的类的更多情况，见 3.2 节（紧接着图 3-7 的讨论之后）。

任何给定情况下的最好选择要取决于以下三个问题：

x. 辅助组件是否需要直接测试？

在我们的列表组件中，Link 类将由在正常情况下对 List 类的测试自动地被 100%地测试；直接测试 Link 没有实在的好处。在辅助类更为复杂的情况下，我们可能应该允许直接访问，这样就会排除隶属（C）、局部（D）或私有嵌入式（E）辅助类实现方式。

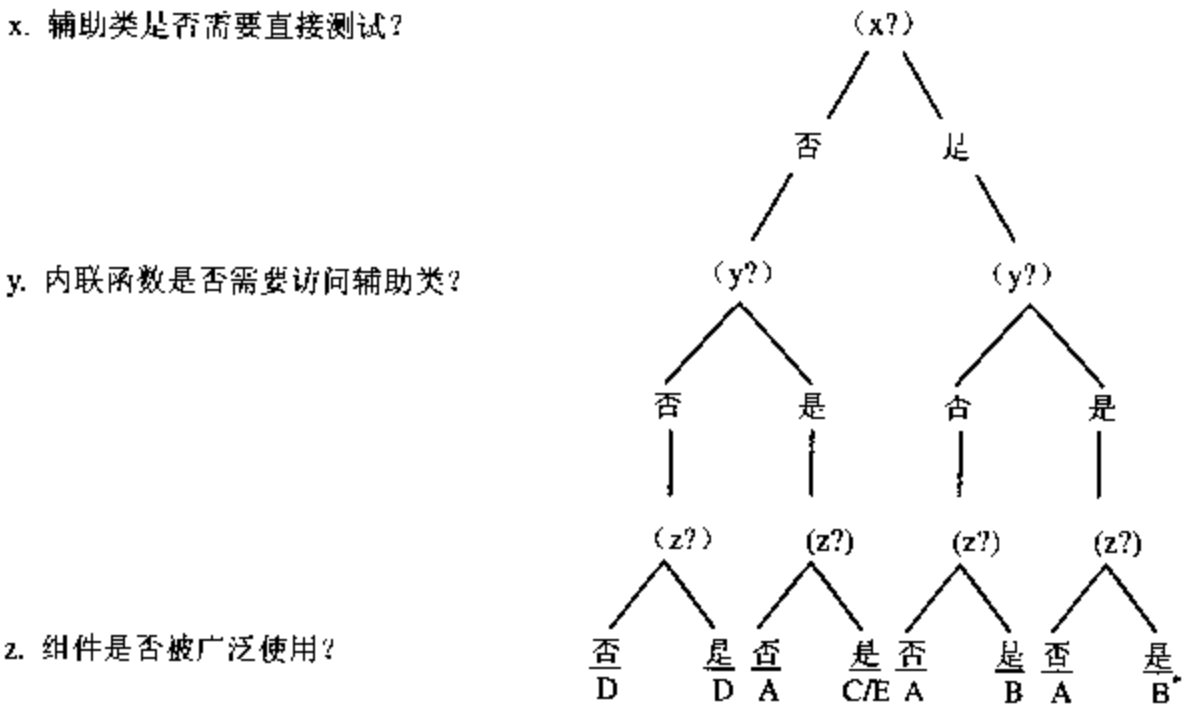
y. 组件是否要暴露需要访问辅助类的内联函数？

如果组件的头部没有定义需要实质地使用辅助类的内联函数，那么辅助类能够与客户绝缘，并且通常最好局部实现（D）。然而，如果辅助类足够复杂而需要直接测试，那么应该选用（A）或（B）实现方式。

z. 组件是否被广泛使用？

如果组件仅仅在单个子系统的实现中使用，那么没有必要考虑太多。初始的实现（A）通常是一个好选择。

图 8-13 中提供了可帮助决策的决策树。



*辅助类的头包含在主要组件的.h 文件而非.c 文件中

图 8-13 决定使用哪个辅助类实现

总结：在主类的头文件中定义辅助类是常见的；即使不是最优的，通常这样做也是恰当

的。在任何可能的情况下，我们一般愿意通过将辅助类隐藏在.c 文件，或若需要测试则将其放在单独的组件中，以将它和客户程序绝缘。对于广泛使用的轻量级的组件，我们也许要被迫使用隶属类或私有嵌入式类，以加强我们对辅助类的专有权。这一节仅起指导性作用，不能替代一般意义的应用。

8.5 小结

组件在逻辑设计和物理设计中都是有效的单元。一个抽象是与对象和（运算符）函数紧密相关的一种抽象规范；一个组件（接口和实现）是相应的具体实现。

当为一个组件建立高层次规范时，要考虑几个相互矛盾的方面。对于被设计为特定子系统的一部分的组件，我们只要求其接口对于所预期的客户来说是充分的就可以了。对于在整个大系统中会用于各种目的的组件来说，我们希望接口是完备的。“充分性”指的是，该接口对于解决某个领域的问题的特定实例来说是合适的。“完备性”指的是，该接口适合于解决那个领域中的任意一个问题。通过将所有组件接口都保持最小，可以加强可用性和可维护性。

被定义为单个对象的成员的操作应该是**基本的（primitive）**。如果一个操作的有效实现需要直接访问那个类的私有细节，那么这个操作是基本的。有用的但不是基本的操作应该根据基本函数在对象之外实现，并且不应该授予友元状态。

在一个组件的接口中使用的用户自定义类型，隐含了对该类型的强逻辑依赖。与物理耦合一样，逻辑耦合最好也要最小化。例如，为了避免不必要的逻辑耦合，应经常选择使用一个 const char *参数而不是使用某种特殊的 string 类，尤其是当接口将在许多不同的上下文中被各种客户使用的时候。

封装是一个对象的特性，它使得对象的实现被修改时不会影响其逻辑接口。有时完全封装的代价是相当昂贵的。但是，与绝缘一样，为了使之有用，封装也不需要绝对化。我们经常可以作出合理的假设：允许我们达到性能目标，并仍然保留足够的封装以允许我们在合理的限制内继续修改实现。如果要求完全封装，那么通过传递进一个前构造好的对象作为可写参数以装载结果，而不是通过值来返回结果，有时可以取得相当大的性能收益。对于管理内部动态内存的重量级对象来说，通过参数返回比通过值返回所获得的性能收益要显著得多。

当实现一个组件时，经常有必要创建一个或多个辅助类。这些类通过定义在组件中的主对象的接口是不可访问的。这些类是组件的实现细节，并且它们都足够简单，可能不需要独立测试。下列策略已被确定可用于实现辅助类：

- 在.h 文件的文件作用域中。
- 在一个单独的组件中。
- 作为一个或多个主要类的隶属类。
- 在组件的.c 文件中。
- 作为一个主要类的私有的（或公共的）嵌入式类。

图 8-12 展示了辅助类的这些实现策略的以下方面的各种优点：

- 辅助类是否可以直接测试？
- 辅助类是否会影响全局名称空间？
- 辅助类是否能用在内联函数体中？
- 辅助类是否物理耦合？
- 辅助类是否能与客户程序绝缘？
- 辅助类是否可独立地重用？

图 8-13 提供了能够用来为一个辅助类选择合适的实现（基于辅助类将被使用的上下文）的决策树。

9 设计一个函数

函数设计的目的是提供到一个抽象所定义的行为的安全、便利和有效的访问。C++语言为我们在函数层确定接口提供了很大的活动范围。是否使一个函数成为一个运算符，它是否应该是一个成员或自由运算符，参数应该如何传递，以及应该如何返回值，这些都是该层次设计的一部分。在作出设计决策时，有一些风格之外的原因在起作用，其中的许多会在本章中介绍。

C++语言提供了各具特点的基本整数类型（例如 short、unsigned、long 等）供我们使用。这些类型还代表了另一种自由度，即如果欠考虑地使用，可能使一个接口变复杂，甚至削弱这个接口。

用户自定义转换运算符允许编译器将某种类型隐式转换成一个用户自定义类型，或将一个用户自定义类型转换为某种类型。谨慎的设计需要权衡隐式转换的可能优势和因类型安全的降低而导致的二义性与出错可能。某些其他的函数，如果没有明确指定，将在需要时由编译器自动定义。决定这些编译器生成的函数定义在什么时候使用效果够好，需要深思熟虑。

本章，我们将提供一个设计组件接口的框架，但只是在一个单独函数细节的层次上。我们将研究一个组件作者可利用的过大的设计空间，并标识出已证明对他们自己的设计是有利或不利的设计决策。我们将讨论在函数接口设计空间中可以消除多少不必要的自由度而不会有任何效率上的损失。最终的框架将有助于把设计者引向更简单、更统一和更可维护的接口。

9.1 函数接口规格说明

在确定一个 C++函数的接口时，根据第 2 章介绍的基础规则，有一系列问题必须研究：

（1）运算符或非运算符函数？

（2）自由或成员运算符？

（3）虚函数或非虚函数？
（4）纯虚成员函数或非纯虚成员函数？
（5）静态或非静态成员函数？
（6）const 或非 const 成员函数？
（7）公共的、保护的还是私有的成员函数？
（8）通过值、引用还是指针返回？
（9）返回 const 还是非 const？
（10）参数是可选的还是必需的？
（11）通过值、引用还是指针传递参数？
（12）将参数作为 const 还是非 const 传递？

有两个组织问题，虽然不是逻辑接口的一部分，但也必须考虑：

（13）友元或非友元函数？
（14）内联或非内联函数？

这些问题之间相互影响；对一个问题的回答一般将隐含或至少影响对另一个问题的回答。下面我们将单独研究每一个问题，并提供进行最佳设计决定的指导方针[①]。

9.1.1 运算符或非运算符函数

除了编译器生成的运算符（例如赋值）之外，将一个函数设计成一个运算符的惟一理由是为了客户程序表示法上的方便。注意，和函数表示法不同，运算符符号不容易受上下文影响；一个从成员函数调用的运算符的最终函数调用解析，将和在文件作用域中的函数调用的解析是一样的[②]。当被正确使用时，运算符重载有着自然而明显的胜过函数表示法的优势——尤其是对于用户自定义逻辑和算术类型来说。

考虑图 9-1 中所示的对应于一个整数集合组件 pub_intset 的两种接口的两种不同使用模式。图 9-1（a）阐释了运算符表示法如何能够被有效地使用。该集合抽象的本值使得这些运算符的含义是直观的，甚至对不熟悉这个特定组件的开发人员来说也是这样。图 9-1（b）显示了使用更复杂的函数调用表示法的等价的计算[③]。

原　则

可读性（不仅仅是易用性）应该是使用运算符重载的主要原因。

---

① 也可参见 **meyers**，Item 19，70 页。

② **ellis**，13.4.1 节，332 页。

③ 我们让一些成员函数成为静态的，以激活与相应的运算符所进行的转换一样的对称的隐式参数转换（见 9.1.5 节）。图 9-1（b）中的深层嵌套的函数调用的首行缩进风格是从 LISP 和 CLOS 这样的语言中借用来的，在这些语言中这样的结构经常出现。

在这个整数集合应用程序中，运算符表示法无疑增强了可读性和易用性。我们用可读性（readability）表示一个软件工程师快速而精确地了解不熟悉的源代码体的预期行为的能力。易用性（ease of use）表示一个开发人员可以有效使用这个对象来创作新的软件的容易程度。任何有代表性的源代码实体，读比写都要多许多倍（“对于最大型和长生存期的软件系统来说，维护开销超过开发开销的倍数在 2~4 之间”[1]），所以从长远来看，支持可读性超过易用性具有实践意义。

```
#include "pub_intset.h"
#include <iostream.h>

main()
{
    pub_IntSet a, b, c, d, e, f;

    a += 1; a += 3; a += 5; a += 7;
    b += 1; b += 2; b += 3; b += 4;
    c = a + b; d = a*b; e = a - b;


    f = a*b*c + b*c*d - c*d*e;


    cout << f << endl;
}
```

（a）有运算符重载

```
#include "pub_intset.h"
#include <iostream.h>

main()
{
    pub_IntSet a, b, c, d, e, f;

    a.add(1); a.add(3); a.add(5); a.add(7);
    b.add(1); b.add(2); b.add(3); b.add(4);
    c = pub_IntSet::or(a, b);
    d = pub_IntSet::and(a, b);
    e = pub_IntSet::suo(a, b);
    f = pub_IntSet::or(
        pub_IntSet::or(
            pub_IntSet::and(
                pub_IntSet::and(a, b),
                c
            ),
            pub_IntSet::and(
                pub_IntSet::and(b, c),
                d
            )
        ),
        pub_IntSet::and(
            pub_IntSet::and(c, d),
            e
        )
    );

    pub_IntSet::print(cout, f) << endl;
}
```

（b）没有运算符重载

图 9-1 一个整数集合抽象的两种使用模式

指导方针

一个重载运算符的语义对客户应该是自然的、明显的和直观的。

---

① **sommerville**，1.2.1 节，10 页。

对于那些不熟悉你组件的开发人员来说，为（在他们看来）没有直观意义的运算符提供灵巧的和易于使用的应用程序是很容易的。但是一些一知半解的笨拙行为，例如将单目运算符~定义为一个（字符）串类的一个成员，以在合适的位置翻转该（字符）串，在一个大规模的开发环境中显然是不合适的。确定何时使用运算符表示法的最后检验，应该看是否有一个自然而直观的含义——对新客户是马上显而易见的——改进（或至少维持）了可读性[①]。

指导方针

用户自定义类型的重载运算符的语法属性，应该反映已经为基本类型定义了的属性。

在语义层次上，关于什么是和什么不是直观的，很难提出明确的指导方针。但是，在语法层次上，基于该语言中基本类型的实现，我们可以作出许多坚定而明确的声明。

原　则

让用户自定义运算符的语法属性模仿预先确定的C++运算符，以避免意外并使它们的使用更可预知。

在C++语言中，每一个表达式都有一个值。有两个基本的值类型，分别称为左值（lvalues）和右值（rvalues）[②]。左值是可以获得其地址的值。如果一个左值可以出现在一个赋值语句的“左边”，就被认为是一个可修改的左值；否则被认为是一个不可修改的左值[③]。右值不能被赋值，也不能获得它的地址[④]。最简单的左值表达式是一个变量标识符本身。这个变量如果没有被声明为const，就是一个可修改的左值。

某些运算符，例如，赋值（=）和它的变体（+=、-=、*=、/=、^=、&=、|=、~=、%=、>>=、<<=）、预递增（++x）以及预递减（--x），在应用于基本类型时都返回可修改左值。这些运算符总是返回一个指向被修改参数的可写引用。例如，这些运算符对基本类型double（如果被实现为一个C++类）的假设定义可能看起来如图9-2所示。

显示在图9-2中的其他运算符返回一个右值，因为没有适当的左值可返回。在对称双目

① 也可参见**cargill**，第5章，91页。

② 最初这些术语来源于经典C：术语左值的意思是表达式的值可以出现在赋值语句的左边，而右值可以出现在右边。随着C++和ANSI C中const的出现，左值现在分为两种：可修改的和不可修改的（见***stroustrup***，2.1.2节，46~47页）。

③ **ellis**，3.7节，25~26页。

④ 位域是一个例外，因为它们可能出现在一个赋值语句的左边，但是，根据ARM（**ellis**，9.6节，184页），一个位域的地址可能不能被提取。对于一个用户自定义类型的未命名的临时变量也同样如此（见9.1.2节）。

运算符（例如，+和*）的情况下，被返回的值既不是左参数也不是右参数，而是一个派生自两者的新值；因此必须通过值来返回[①]。相等（==、!=）和关系（<、<=、>、>=）运算符总是返回一个 int 类型的右值，不是 0 就是 1；很明显，没有一个输入参数适合在这里返回。后递增和后递减运算符是一个有趣的特例，唯有这两个运算符修改了对象但却没有返回适当的左值：

```
class double {                                  // Note: not legal C++
    // ...
  public:
    double() {}
    double(int);
    double(const double&);
    ~double() {}

    double& operator=(const double& d);
    double& operator+=(const double& d);
    double& operator-=(const double& d);
    double& operator*=(const double& d);
    double& operator/=(const double& d);

    double& operator++();                       // pre-increment ++x
    double& operator++();                       // pre-decrement --x
    double operator++(int);                     // post-increment x++
    double operator--(int);                     // post-decrement x--

    double *operator&();                        // unary address operator
    const double *operator&() const ;           // unary address operator
};

double operator+(const double& d);              // unary +
double operator-(const double& d);              // unary -

int operator!(const double& d);                 // unary logical "not"

int operator&&(const double& left, const double& right);
int operator||(const double& left, const double& right);

double operator+(const double& left, const double& right);
double operator-(const double& left, const double& right);
double operator*(const double& left, const double& right);
double operator/(const double& left, const double& right);

int operator==(const double& left, const double& right);
int operator!=(const double& left, const double& right);
int operator< (const double& left, const double& right);
int operator<=(const double& left, const double& right);
int operator> (const double& left, const double& right);
int operator>=(const double& left, const double& right);
```

图 9-2 基本类型 double 的假设实现

① 更进一步的详细解释，见 **meyers**，Item 23，82~84 页。

```
double double::operator++(int)
{
    double tmp = *this;
    ++*this;
    return tmp;
}
```

```
double double::operator--(int)
{
    double tmp = *this;
    --*this;
    return tmp;
}
```

还有一个更微妙的实例，请思考图 9-3 中显示的对应于一个通配符号表抽象的两种使用模式。在两种情况下，均构建了一个符号表（由类型 int 参数化）并加入了两个符号，符号“foo”的值通过名称来查找。因为一个指定名称的符号完全有可能不存在于表中，所以，执行查询的函数通过值或引用来返回其结果都是不合适的；因此该值应通过指针返回。（注意我们刚才随意对待封装的方式和程度）。但是，将这种用法和我们通常在把运算符[ ]应用于一个基本 int 数组时期望的用法相比较。我们期望取回一个指向索引值的引用，而不是一个可能为空的指针。图 9-3（a）中的运算符[ ]和基本类型的运算符[ ]用法之间的差异，趋向于使图 9-3（b）中的函数调用表示法在这个例子中变得更可取。为那些其语法紧密反映相应的基本语法的情况保留运算符表示法，可以增强运算符重载的效力。

图 9-4 总结了大多数 C++运算符在被应用于基本类型时的声明。（对基本运算符->、->*、()和，没提供什么见解。）

```
#include "gen_symtab.h"

main()
{
    gen_SymTab<int> s;
    s("foo", 1); // operator()
    s("bar", 2); // (bad idea)
    const int *val = s["foo"];
    // ...
```

(a) 有运算符重载

```
#include "gen_symtab.h"

main()
{
    gen_SymTab<int> s;
    s.add("foo", 1);
    s.add("bar", 2);
    const int *val = s.lookup("foo");
    // ...
```

(b) 没有运算符重载

图 9-3 一个通配符号表抽象的两种使用模式

```
                                    // operators with similar declarations
class T {
    T& operator++();                // ++x --x (prefix)
    T operator++(int);              // x++ x-- (postfix)

    T* operator&();                 // &x (unary)
    const T* operator&() const;     // &x (unary)

    T& operator=(const T&);         // = += -= *= /= %= <<= >>= &= ^= |=
};

T operator-(const T&);              // - + ~ (unary)
int operator!(const T&);            // ! (unary)

T operator+(const T&, const T&);    // + - * / << >> % & ^ |
int operator==(const T&, const T&); // == != < <= > >=
```

```
int operator&&(const T&, const T&);  // && ||

// if the type is pointer-like (i.e., P = T*)

class P {
    T& operator[](int) const;        // indexed array access (binary)
    T& operator*() const;            // pointer dereference (unary)
};

// if type is pointer-to-const-like (i.e., PC = const T*)
class PC {
    const T& operator[](int) const;  // indexed array access (binary)
    const T& operator*()const;       // pointer dereference (binary)
};
```

图 9-4 一些基本运算符属性的总结

注意，没有修改它们的参数的单目运算符不是基本成员。例如，单目运算符!可以很好地对一个用户自定义类型（例如一个 ostream）起作用，即使没有给这种类型定义!运算符：

```
#include "iostream.h"
void g(ostream& out)
{
    if (!out) {
        cerr << "output stream is bad" << endl;
        return;
    }
    // ...
}
```

上述代码能起作用，因为一个 ostream 知道如何隐式地将自己转变成定义了！运算符的基本类型（void *）。如果运算符!被设定为 void*类的一个成员，那么不可能有用户自定义转换出现，并且上述代码将导致一个编译时错误。

如果我们要禁止参数隐式转换成“自由”单目运算符!，那么我们可以简单地通过使！操作成为一个成员函数（例如，用 obj.not()代替一个运算符）来和这些指导方针保持一致[①]。

9.1.2 自由或成员运算符

决定一个运算符函数是成为一个成员函数还是自由函数，完全取决于是否要求最左边操作数的隐式类型转换。如果该运算符修改了这个操作数，这样的转换无疑是不受欢迎的。

原 则

C++语言本身可作为用户自定义运算符模仿的一个客观和适宜的标准。

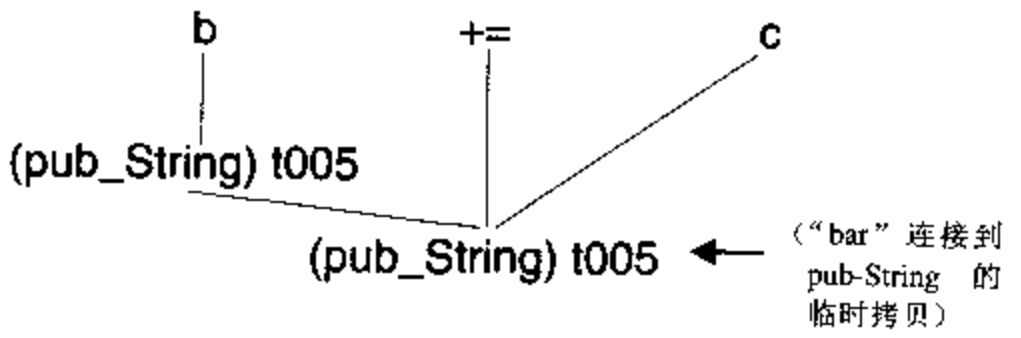
思考一下，如果我们把一个字符串类的连接运算符（+=）定义为一个自由函数（而不是

① 也可参见 **murray**，2.5 节，44 页。

模仿基本类型所采用的方法）情况会如何。如图 9-5 所示，使 operator+=成为一个自由函数，已经使它的左侧 const char*操作数能够隐式地转换为一个临时的 pub_String（在这里表示为 t005），其值为 foo。即使这个临时的变量会是基本类型的一个右值，正是这个临时的 pub_String 对象随后有值“bar”连接到 b（并且不是一个编译时错误）[1]。因为这种行为很可能会使我们的客户感到意外和恼火，所以取缔它是明智的。

```
// pub_String.h
// ...
class pub_String {
    // ...
  public:
    pub_String(const char *str);
};

pub_String& operator+=(pub_String& left, const pub_String& right);
    // free-function definition of concatenation for strings

// ...
```

```
#include "pub_string.h"

void f()
{
    pub_String a("tar");
    const char *b = "foo";
    pub_String c("bar");
    b += c; // has no effect
```




```
    a += b += c; // a now holds "tarfoobar"
                 // but b remains unaffected.
}
```

图 9-5 将 operator+=实现为一个自由函数的结果

另一方面，我们希望某些操作（例如，+和==）运作时不考虑其参数的顺序。考虑运算符+，它被用来连接两个字符串并通过值返回其结果。C++语言允许我们将运算符+定义为一个成员或非成员。对运算符==也一样。如果我们选择将这些运算符定义为成员，那么我们将使

① C++语言现在允许用户自定义类型的未命名的临时变量的修改。见 **murray**，2.7.3 节，53~55 页。

我们的客户遭遇以下反常行为：

```
void f()
{
    pub String s("foo"), t("");
    int i;

    t = s + "bar";                  // ok
    t = "bar" + s;                  // error
    i = s == "bar";                 // ok
    i = "bar" == s;                 // error
}
```

问题在于

```
pub_String::operator+(const String& right)
```

和

```
pub_String::operator==(const String& right)
```

通过一个如下形式的构造器：

```
pub_String::pub_String(const char *)
```

使一个 char *隐式转换成一个 pub_String 类，而在左边不可能有这样的转换[①]。使这些运算符成为自由运算符，可以解决这个对称问题，直到我们将转换运算符

```
pub_String::operator const char *() const
```

加入到 pub_String 类。

图 9-6 展示了一个只因加入了一个从 pub_String 到 const char *的转换（cast）运算符而造成的问题。奇怪的是，这两个明显相似的运算符（==和+）并不像我们愿意（天真地）相信的那样在重载方面是一致的。差异在于存在两种解释==运算符的方式：

（1）隐式地将 char *转换成一个 pub_String，并使用 operator==(const String&, const String&)进行比较。

（2）隐式地将 pub_String 转换成一个 const char *，并使用指针类型的内置==进行比较。

这种问题对+运算符不存在，因为在 C++中没有办法“加”两个指针类型，所以没有二义性。

在一个现实世界的字符串类中，我们绝不会依靠隐式转换来获得字符串值，以免额外的构造和析构会不适当地影响我们的性能。反之，我们会定义运算符+的独立重载版本以尽可能有效地处理这三种可能性中的每一种，从而回避这些二义性问题。

---

① **ellis**，13.4.2 节，333 页。

```
// pub_String.h
// ...
class pub_String {
    // ...
  public:
    pub_String(const char *pcc);
    // ...
    operator const char *() const;   // <== new conversion operator
};

int operator==(const String& left, const String& right);
String operator+(const String& left, const String& right);

// ...
```

```
#include "pub_String.h"

void f()
{
    pub_String s("foo"), t("");
    int i;

    t = s + "bar";          // ok
    t = "bar" + s;          // ok
    i = s == "bar";         // error (ambiguous)
    i = "bar" == s;         // error (ambiguous)
    i = strlen(s);          // ok
}
```

图 9-6 两种转换运算符导致的模棱两可

原 则

重载运算符中的不一致问题，对客户来说可能是明显的、讨厌的和高代价的。

如图 9-7 所示，为了接受==运算符左边的一个 const char *，我们不得不至少让同等运算符函数中的一个成为自由函数。

```
class pub_String {
    // ...
  public:
    pub_String(const char *pcc);
    operator const char *() const;
    int operator==(const char *pcc) const; // bad idea: (asymmetric)
        // Allows for user-defined conversion only for the
        // argument on the right side of the operator.
};

int operator==(const pub_String& left, const pub_String& right);
int operator==(const char *left, const pub_String& right);
    // Allows for user-defined conversion on both the
    // left and right arguments symmetrically.
```

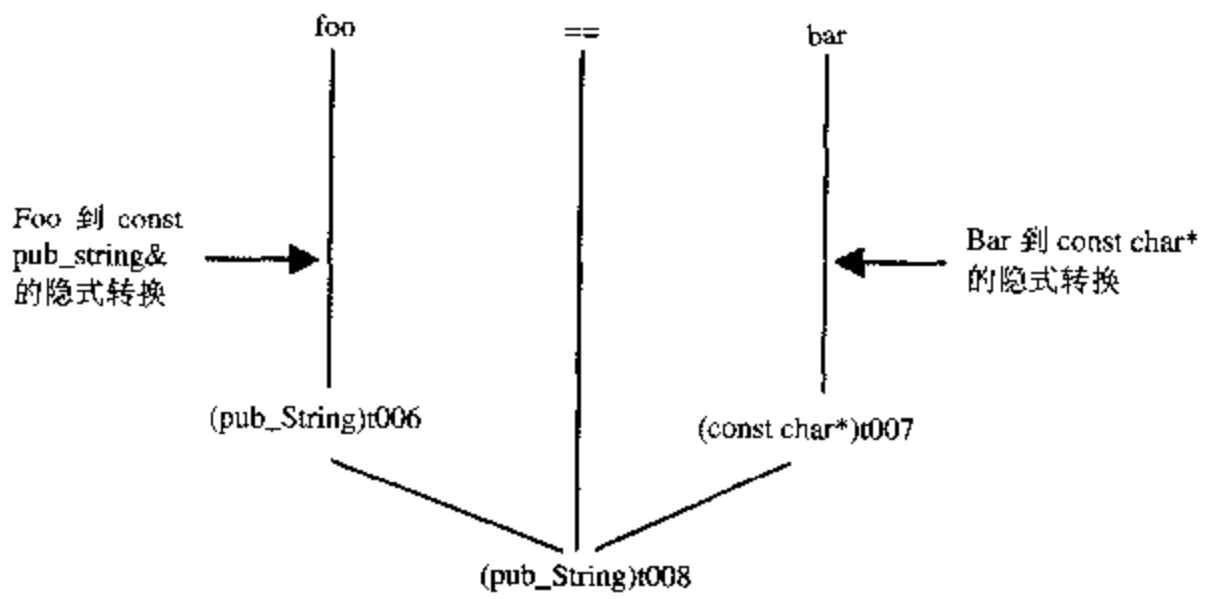
```
struct Foo {
    Foo();
    operator const pub_String& () const;
        // Implicitly convert a Foo to a pub_String.
};

struct Bar {
    Bar();
    operator const char *() const;
        // Implicitly convert a Bar to a (const char *).
};

void g()
{
    Foo foo;
    Bar bar;
    if (bar == foo) {        // ok:      Bar =to=> (const char *)
        // ...                           Foo =to=> (const pub_String&)
    }
    if (foo == bar) {        // error:   Foo =NO=> (const pub_String&)
        // ...                           Bar =to=> (const char *)
    }
}
```

图 9-7 将 operator==实现为一个成员函数的结果

当我们提供了运算符==函数的所有三个版本时，使其中一个成为成员会有什么害处呢？害处就是缺乏对称可能使我们的客户感到意外。如果一个对象可以隐式转换成 pub_String，另一个转换成一个 const char *，我们仍然期望比较顺序是不重要的。也就是说，如果 bar==foo 编译了，那么 foo==bar 也应该通过编译（并且在运行时产生同样的结果）。但是，如果

```
int operator==(const pub_String&, const char *);
```

版本不能用作一个自由函数，那么下列隐式转换就没有办法出现：



结论是 operator==应该**总是**一个自由函数，不管涉及什么其他函数。同样的推论对不修改操作数并且以值返回它们的结果的其他双目运算符也同样有效。

由 C++语言本身提出的例子是一个公平而有用的模型，客户可以使用它来推断运算符的基本语法属性和公理属性。模型化基本操作的目的不是要不必要地激活隐式转换，而是要更对称以避免使客户感到意外。如果运算符重载被用于任何更大的范围，有理由期望抽象适合在多种情况下重用。可重用组件的客户将欣赏到一个一致并且专业的接口——全无语法意外。注意 C++语言要求下列运算符是成员[1]：

=	赋值
[]	下标
->	类成员访问
()	函数调用
(T)	转换（“cast”）运算符
new	（静态）分配运算符
delete	（静态）删除运算符

9.1.3 虚函数或非虚函数

动态绑定使得通过基类而被访问的成员函数能够由对象的实际子类型来确定，这与调用中的指针类型或引用类型相反。一个函数必须声明为 virtual 才能被动态绑定。在 C++中只有成员函数才可以是虚拟的。但是，“一个运算符的多态行为要求原来的自由函数现在要变成成员函数”的结论是错误的。

原　则

不必为了获得多态行为而损害语法问题，例如双目运算符的对称隐式转换。

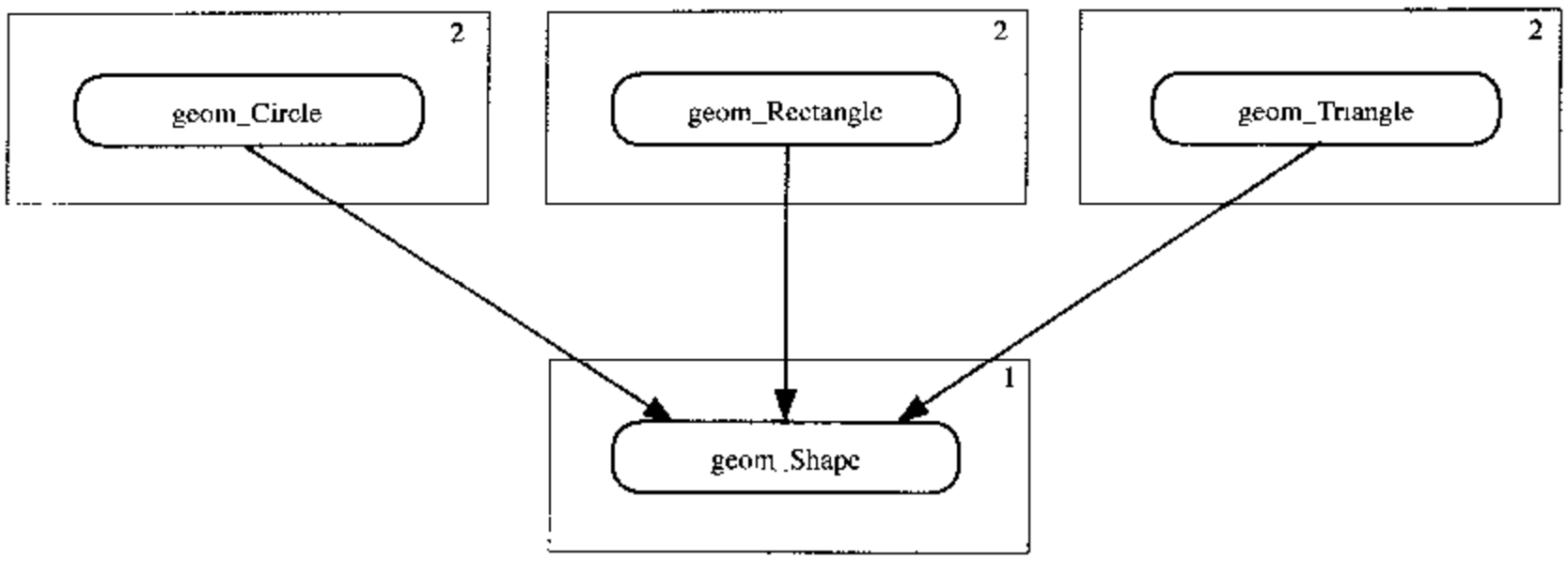
图 9-8 阐释了即使是在多态行为面前，对称运算符如何能够和应该怎样继续保持自由。不是让这六个等式运算符和关系运算符的每一个都成为类的虚成员，而是提供一个单个的虚比较成员。现在这六个运算符对于任何隐式转换都继续对称进行。

甚至在关系运算符没有意义的时候（例如点抽象），等式运算符也常常是有意义的。有时候对一个异类集合排序可以使访问更有效。在这样的情形下，任何排序（甚至是任意排序）都可能是有用的。图 9-8 中的虚拟 classId()方法使派生类型能够定义它们自己的运行时类型标识符[2]。使用这种标识符，同一类型的 shape 可以按照它们自己的内部次序来排序，而跨越具体类型的排序可以通过一些不同的（也许是任意的）比较来定义。图 9-9 简洁地提供了一个参与了 shape 的整个排序的 geom_Circle 实现，以供读者参考。

---

① **ellis**，12.3c 节，306 页；**stroustrup94**，3.6.2 节，82~83 页。

② 见 **ellis**，10.2 节，212~213 页。




```
// geom_shape.h
#ifndef INCLUDED_GEOM_SHAPE
#define INCLUDED_GEOM_SHAPE

class geom_Shape {
  public:
    virtual ~geom_Shape();
    virtual const void *classId() const = 0;
    virtual int compare(const geom_Shape& shape) const = 0;
        // Returns negative, zero, or positive corresponding to
        // whether this geom_Shape object is less than, equal to, or
        // greater than the specified geom_Shape object, respectively.
};

inline int operator==(class geom_Shape& left, class geom_Shape& right) {
    return left.compare(right) == 0; }

inline int operator!=(class geom_Shape& left, class geom_Shape& right) {
    return left.compare(right) != 0; }

inline int operator<=(class geom_Shape& left, class geom_Shape& right) {
    return left.compare(right) <= 0; }

inline int operator<(class geom_Shape& left, class geom_Shape& right) {
    return left.compare(right) <  0; }

inline int operator>=(class geom_Shape& left, class geom_Shape& right) {
    return left.compare(right) >= 0; }

inline int operator>(class geom_Shape& left, class geom_Shape& right) {
    return left.compare(right) >  0; }

#endif
```

图 9-8 使用自由运算符的 Shape 的多态比较

```
// geom_circle.h
#ifndef INCLUDED_GEOM_CIRCLE
#define INCLUDED_GEOM_CIRCLE

#ifndef INCLUDED_GEOM_SHAPE
#include "geom_shape.h"
#endif

class geom_Circle : public geom_Shape {
    static const void *d_classId_p;
    double d_radius;

  public:
    geom_Circle(double radius) : d_radius(radius) {}
    geom_Circle(const geom_Circle& circle) : d_radius(circle.d_radius) {}
    ~geom_Circle(const geom_Circle& circle);
    geom_Circle& operator=(const geom_Circle& circle) {
                        d_radius = circle.d_radius; return *this; }
    const void *classId() const { return d_classId_p; }
    int compare(const geom_Shape& shape) const;      // virtual
    int compare(const geom_Circle& circle) const;    // non-virtual
};

inline int operator<(class geom_Circle& left, class geom_Circle& right) {
    return left.compare(right) < 0; }

// ... (definitions of other 5 symmetric operators omitted)

#endif
```

```
// geom_circle.c
#include "geom_circle.h"
const void *geom_Circle::d_classId_p = &d_classId_p; // runtime type id
geom_Circle::~geom_Circle() {} // empty & out-of-line (see Section 9.3.3)

int geom_Circle::compare(const geom_Shape& shape) const
{
    return shape.classId() == d_classId_p ?
        compare((const geom_Circle&) shape) :   // compare instances
        d_classId_p < shape.classId() ? -1 : 1; // compare types
}

int geom_Circle::compare(const geom_Circle& circle) const
{
    return d_radius < circle.d_radius ? -1 : d_radius > circle.d_radius;
}
```

图 9-9　geom_Circle 的多态比较的实现

原　则

虚函数实现行为上的变化；数据成员实现值的变化。

更一般地，虚函数被用来描述跨越派生自一个共同基类的类型的行为变化。而数据成员

不必借助于继承便足以描述值的变化[1]。例如，我们不会定义一个协议类 art_Color，然后派生出类 art_Red、art_Blue 和 art_Yellow；单个的（也许是完全绝缘的）存储了许多枚举颜色的具体类 art_Color 可能是更适当的设计。但是，虚函数是打破编译时依赖和连接时依赖的一种有效技术（见 6.4.1 节）。因为这个原因，一个单个的具体类可能派生自一个 art_Color 协议。

定义：

隐藏（hide）：一个成员函数若与在一个基类或文件作用域中声明的某个函数同名，则**隐藏**了那个函数。

重载（overload）：一个函数用定义在同一作用域的同样的名称**重载**了另一个函数的名称。

覆盖（override）：一个成员函数**覆盖**了在一个基类中声明为虚拟的同样的函数。

重新定义（redefine）：一个函数的默认定义被另一个定义不可挽回地取代。

最后，有四个类似的术语（隐藏、重载、覆盖和重新定义）通常被用来（和被误用）来描述一个函数以及它对其他函数的影响，我们在这里给出了它们的定义以供参考。只有当相同名称的不同函数在同一个作用域中被声明时，我们才认为它们被重载了。当一个派生类中的成员函数以一个在基类中声明为虚函数的同样的接口声明时，该成员函数被认为是覆盖了这个基类函数。在所有其他情况下，一个函数名称隐藏了所有在一个封闭作用域中的同名函数，不管它们的参数基调如何。隐藏在一个已命名作用域中的函数不能直接访问，但可以通过作用域标识符（::）来访问。然而，当我们重新定义一个函数时（例如，全局 new 或特定类的单目运算符&），我们取代了它的定义，以前的定义不能再通过程序来访问了[2]。

指导方针

避免将一个基类函数隐藏在一个派生类中。

我们应该小心，不要将任何基类函数的定义隐藏在派生类中。特别地，我们决不能为一个派生类中的非虚函数提供一个新的定义，因为这会使该函数对任何指针或引用（从该指针或引用调用该函数）的类型敏感。允许指针或引用的类型影响哪个行为被调用，是违反直觉、敏感和容易引起错误的[3]。隐藏定义在基类中的函数不能保护它们不被使用，而只能使这样的使用更麻烦而已。我们总是可以乱动指针或使用作用域标识符来调用隐藏的成员。一个更好的主意是，决不要首先隐藏一个成员函数。一个涉及虚函数、多重继承和运行时类型标识的设计模式示例可以在附录 C 中找到。

---

① 见 **cargill**，第 1 章，16~19 页。

② **ellis**，10.2 节，210 页和 13.1 节，310 页。

③ 见 **meyers**，Item 37，130~132 页。

9.1.4 纯虚成员函数或非纯虚成员函数

将一个虚函数声明为 pure 会迫使具体派生类的作者提供一个定义。如果不能在一个派生类中提供一个特定行为很可能是一个错误，那么这个虚函数应该在基类中声明为 pure。

协议类（见 6.4.1 节）对在继承层次结构中实现层次化和绝缘是有用的。我们要避免在协议类本身定义任何行为；使所有的成员函数（除了析构函数）成为纯虚函数，可以使我们避免定义任何行为。

一个派生自一个纯协议的抽象类有时被称为**部分实现**。没有在派生类中声明的协议函数被继承为纯虚函数。在一个部分实现中的一些函数可能有一个有用的默认行为，但是它们应该不必自动成为默认行为。如图 9-10 所示，定义一个虚函数和声明它为 pure 都迫使一个派生类作者显式地激活默认行为[1]。

```
#include <iostream.h>

struct Base {                          // *** Base Class ***
    virtual void f() = 0;
    virtual ~Base();
};

Base::~Base() {}

struct Partial : Base {                // *** Partial Implementation ***
    virtual void f() = 0;              // declaration of pure virtual function
    ~Partial();
};

void Partial::f()                      // definition of pure virtual function
{
    cout << "Partial::f" << endl;
}

Partial::~Partial() {}

struct Derived : Partial {             // *** Concrete Derived Class ***
    Derived() {}
    void f();
    ~Derived();
};

void Derived::f()
{
    cout << "Derived::f" << endl;
    Partial::f();                      // explicit call of pure virtual function
}
```

① 见 **ellis**，10.3 节，214 页。

```
Derived::~Derived() {}


main()
{                                          // *** Main Program ***
    Base *b = new Derived;
    b->f();
}

// Output:
//       john@john: a.out
//       Derived::f
//       Partial::f
//       john@john:
```

图 9-10 迫使默认行为被显式地激活

9.1.5 静态或非静态成员函数

使一个函数成为一个类的静态成员的显而易见的理由是，它不依赖于一个对象的任何特定实例：

```
class my_Widget {
    static int d_instanceCount;
    // ...
  public:
    static int instanceCount() { return d_instanceCount; }
    // ...
};
```

原 则

静态成员函数通常用于实现一个单独工具类中的非基本功能。

将功能升级到一个更高的层次，可能会要求函数成为定义在某个其他组件中的某个类型的一个静态成员函数（见图 5-15）。如果该函数是一个不要求私有访问的辅助函数，我们可以考虑使这个函数成为一个单独类的静态成员，以强调它的非基本状态。

```
class geom_Point { /* ... */ };

struct geom_PointUtil {
    static int compareMagnitude(const Point& a, const Point& b);
        // Compare the distance of each point from the origin,
        // and return a negative, zero, or positive value
        // depending on whether the magnitude of a is less than,
        // equal to, or greater than that of b, respectively.
};
```

注意，通过使 compareMagnitude 成为一个静态函数，我们保持了其参数隐式转换的对称性。如果我们代之以声明 compareMagnitude 为 geom_Point 的一个非静态成员，就有可能出现这样的情况：a.compareMagnitude(b)可以编译，而 b.compareMagnitude(a)不能（见 9.1.2 节）。

虽然很少有必要，但仅通过将静态方法移入 geom_Point 类本身的方式来保持对称时，我们可以授予私有访问权。

9.1.6 const 成员函数或非 const 成员函数

在任何合适的地方，一个成员函数应该被声明为 const，以消除对其使用的不必要的限制。有两种 const 概念：**逻辑的**和**物理的**。逻辑 const 更模糊些，指用户认为应该设定为 const 的成员函数；客户从程序上无法察觉的内部组织的改变被认为是**逻辑** const。另一方面，C++语言支持**物理** const。一个成员函数可以声明为 const，只要其（1）未修改直接包含在定义该类的结构中的位，或者（2）不返回一个指向任何嵌入那个结构的数据成员的非 const 指针或引用。

图 9-11 阐释了物理 const 在 C++中是如何实施的。一个 const 成员函数被允许修改和返回一个指向内存的可写的引用，该引用由一个对象拥有和管理。因为所有上述定义的函数都会导致或激活副作用，而客户可从程序上访问到这些副作用，所以没有一个会被认为是逻辑 const 函数。

```
class ex_String {
    char *d_str_p;

  public:
    // ...
    makeNull()             { d_str_p = 0; }      // physically non-const
    makeEmpty()     const { d_str_p[0] = 0; }   // physically const

    char *&getRepRef()     { return d_str_p; }   // physically non-const
    char * getRep() const { return d_str_p; }   // physically const
};
```

图 9-11 C++语言只实施物理 const

定义：如果一个函数只有单一参数，该参数是一个指向某个对象的引用，并且若不经过显式的转换（cast）就不能从函数体内部获得一个指向同一个对象（或它的一部分）的非 const 引用，那么这个对象是 const **正确的**（**const-correct**）。

指导方针

系统中的每一个对象都应该是 const 正确的。

① 见 **meyers**，Item 21，73~78 页。

决定什么是和什么不是一个对象的 const 行为是它的类设计的一个重要部分（见 10.3.1 节）。必须小心确保没有允许一个客户回避这个决策的漏洞。从一个 const 成员函数返回对一个对象的内部表示的可写访问，可能会使编译器在确保一个 const 对象不被修改方面出现能力短路[①]。

原　则

从一个 const 成员函数返回一个非 const 对象可能会破坏一个系统的 const 正确性。

在设计系统时，很容易因疏忽而提供一些方法，使用这些方法，可以从指向一个对象的 const 引用中获得一个指向该对象的引用的非 const 版本。例如，图 9-12 提供了用于实现一个二叉树的 te_Node 的两个定义。图 9-12（a）定义了 const 成员函数，它们返回对父母和每个孩子节点的可写访问。一个接受指向一个 const te_Node 的引用的函数，不必借助于转换（cast）就可以很容易地修改这个推测的 const 值，如下所示：

```
void f(const te_Node& readonlyNode)
{
    if (readonlyNode->child1()) {
        te_node *writableNode = readonlyNode->child1()->parent();
        writableNode->setValue(-9.99E99);
    }
}
```

```
class te_Node {
    // ...
  public:
    // ...

    // MANIPULATORS
    void setValue(double v);



    // ACCESSORS
    const char *name() const;
    te_Node *parent() const;
    te_Node *child1() const;
    te_Node *child2() const;
};
```

(a) const 不正确

```
class te_Node {
    // ...
  public:
    // ...

    // MANIPULATORS
    void setValue(double v);
    te_Node *parent();
    te_Node *child1();
    te_Node *child2();

    // ACCESSORS
    const char *name() const;
    const te_Node *parent() const;
    const te_Node *child1() const;
    const te_Node *child2() const;
};
```

(b) const 正确

图 9-12　阐释 const 正确性

① meyers，Item 29，96~99 页。

在一个系统的上下文中，一个对象若不允许指向一个对象的非 const 引用从只指向（直接或间接）同一个对象的 const 引用中获得，则该对象被认为是 const 正确的。图 9-12（b）定义了一个类，它没有提供通过间接手段从一个 const 引用获得可写访问的方法。这个实现是 const 正确的，因为它维护了只用一个 const te_Node 能做什么和不能做什么的意图。

> **定义**：在一个系统中，如果一个只有一个参数（该参数是指向一个系统内的对象的任何子集的 const 引用）的函数未通过显示转换就不能获得一个指向这些对象中的任何一个（或它们的任何部分）的可写引用，那么这个系统是 **const 正确的**。

换句话说，给定一个任意函数：

```
void f(const T1& a1, const T2& a2, ..., const TN& aN)
{
    // There is simply no way for me to get hold of a
    // writable reference to any of a1, a2, ..., aN or
    // any portion thereof (short of casting away const).
}
```

指导方针

一个系统应该是 const 正确的。

const 正确性是一种超出一个单个类或组件、应用于一个完整系统的性质。例如，图 5-8 中的类 Node 是不可信的，因为它包含一个函数

```
Edge& Node::edge(int index) const;
```

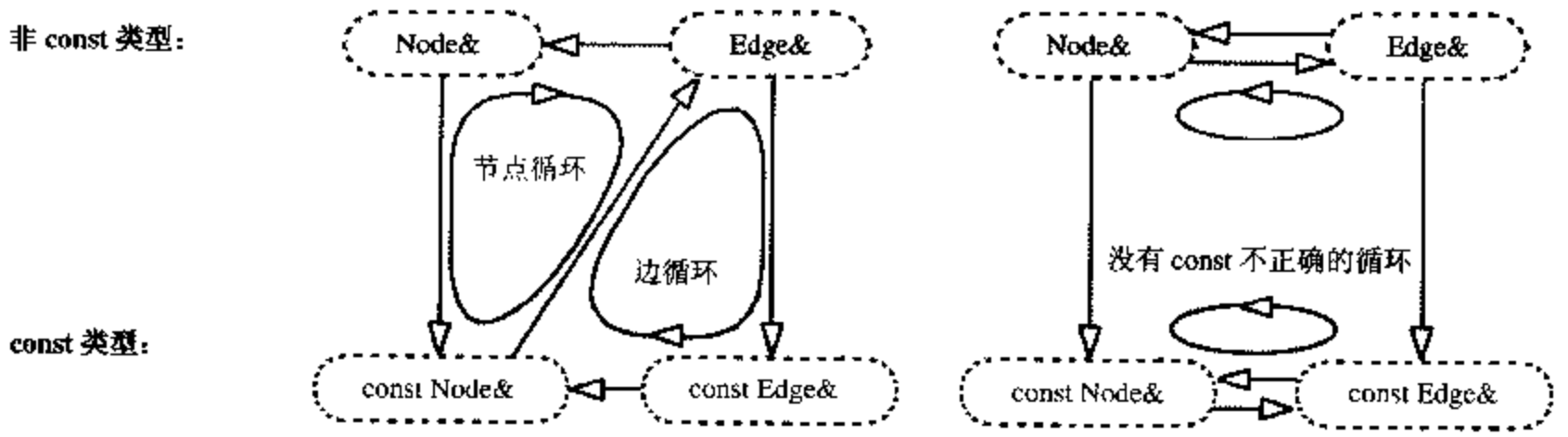
该函数允许一个客户从一个 const Node 获得一个可修改的 Edge。即使 Edge 已经像下面所显示的那样小心地让 const 永存：

```
Node& Edge::to();                          Node& Edge::from();
const Node& Edge::to() const;              const Node& Edge::from() const;
```

只要我们可以从一个非 const Edge 获得一个指向一个 Node 的非 const 引用，这个子系统就不是 const 正确的：

```
void f(const Node& readonlyNode)
{
    if (readonlyNode->numEdges() > 0) {
        Edge& writableEdge = readonlyNode->edge(0);
        Node& writableNode = (&writableEdge->to() == this) ?
                             writableEdge->to() :
                             writableEdge->from();

        // writableNode is a writable reference to readonlyNode!
    }
}
```

在一个异类的对象网络中，一个类型的一个const成员可能会被利用来破坏另一个的const正确性。这个问题可以通过图表来阐释，如图9-13（a）所示。在这个转换图中，系统中的每一个类型都有一个非const表示和一个const表示。从一个类型的非const版本（向下）转换成const版本是自动的。类型X的成员函数（它返回一个指向类型Y的指针或引用）在转换图中用适当的有向边来表示，该有向边从类型X的适当版本（即const或非const）指向类型Y的适当版本。



（a）包含const/非const循环　　（b）是const正确的

图9-13　使用图来建立const正确性（见图9-12）

因为Edge定义了一个返回指向const Node的一个引用的const成员，所以有一条从const Edge&到const Node&的有向边。这个方法的非const版本也被类似处置。问题是Node包含了一个会返回非const Edge引用的const成员——所以图9-13（a）中有向上倾斜的对角线。这个对角线引进了一个同时包含了Node的const和非const版本的循环。因此，给定一个指向一个Node的const引用，我们可以潜在地获得一个指向该Node的非const引用。一个类似的循环涉及Edge的两个版本，这样，不必使用cast，一个const Edge引用就被转换成了非const。

> **定义**：如果一个系统的转换图未包含同时涉及任何一个类型的const和非const版本的循环，那么这个系统是const**正确的**。

在图9-13（b）中显示的转换图表反映了图9-12（b）中给出的Node的定义。有Node&和Edge&之间的转换循环，也有const Node&和const Edge&之间的循环。但是，没有一个循环涉及类型的两个版本，因此这个小子系统是const正确的。这个const正确性的解释一般应用于一个完整的系统。

一个对象可能提供const信息，例如一个名称，该名称可用来在某个非const容器对象中查找同一对象的一个可写版本：

```
void g(te_Tree *t, te_Node& readonlyNode)
{
    te_Node *writableNode = t->lookup(readonlyNode.name());
```

```
    writableNode->setValue(-9.99E99);
}
```

上述情况不破坏 const 正确性，因为单独的 const 对象不足以获得它自己的一个非 const 版本。如果 te_Tree 已经被 const 引用传递，那么我们可以期望 te_Tree::lookup 成员函数的一个 const 版本将改为返回一个指向 const te_Node 的指针，以确保图 9-14 中显示的系统是 const 正确的。

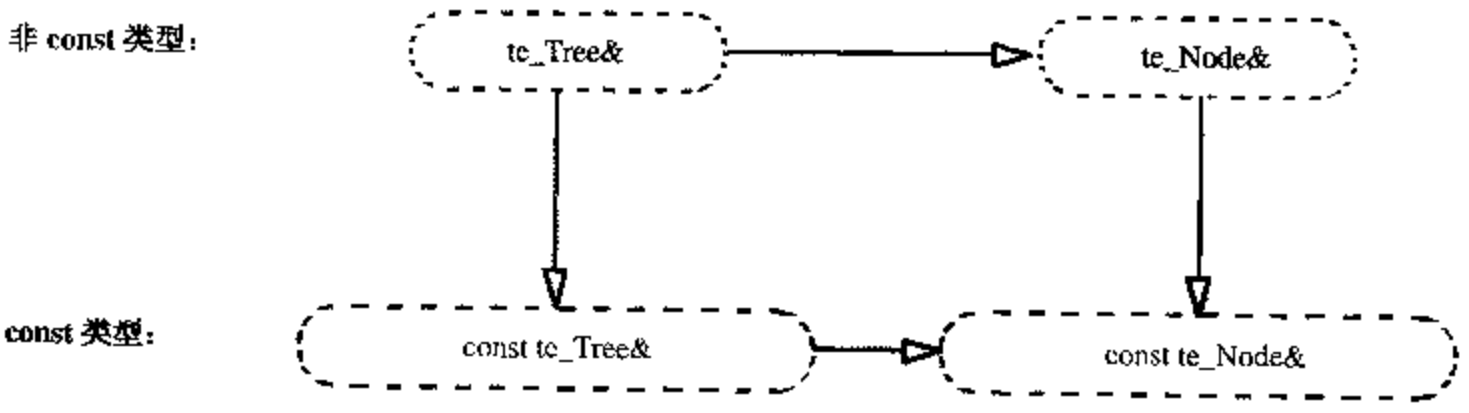


图 9-14　te_Tree 和 te_Node 的一个 const 正确的实现

有些情况下我们确实需要一个 const 成员返回一个指向另一个类型的非 const 引用或指针。在图 6-75 的 PointIterHandle 中，我们可以从指向 const 句柄的引用访问被包含的 PointIter 的一个可写版本：

```
PointIter *PointIterHandle::operator->() const;
```

这里的目的是要模拟一个通过值传递的可写指针的语义——也就是说，我们可以修改所指示的对象，但不能改变句柄本身来引用一个不同的对象[①]。但是，正如图 9-15 所阐释的，这个子系统是const正确的，因为没有办法从任何一种PointIter获得一个可写的PointIterHandle引用[②]。

忽略 const 正确性意味着一个通过 const 引用传递给函数的参数可以“合法地”在那个函数内被修改。重视 const 正确性能改进系统的一致性，并使编译器能够在编译时发现更宽泛的错误类。

指导方针

在抛弃 const 之前（至少）考虑两次。

① 使用从句柄（通过 const 引用传递给一个函数的句柄）获得的一个 const 迭代器是不可靠的。

② 注意，通过静态分析可以检验 const 的正确性；但是，因为破坏了 const 正确性可能会依赖于底层对象（实例）网络，所以可能使得一个 const 正确的系统不能被静态地证明。这样的系统维护起来更昂贵并且更难以验证。

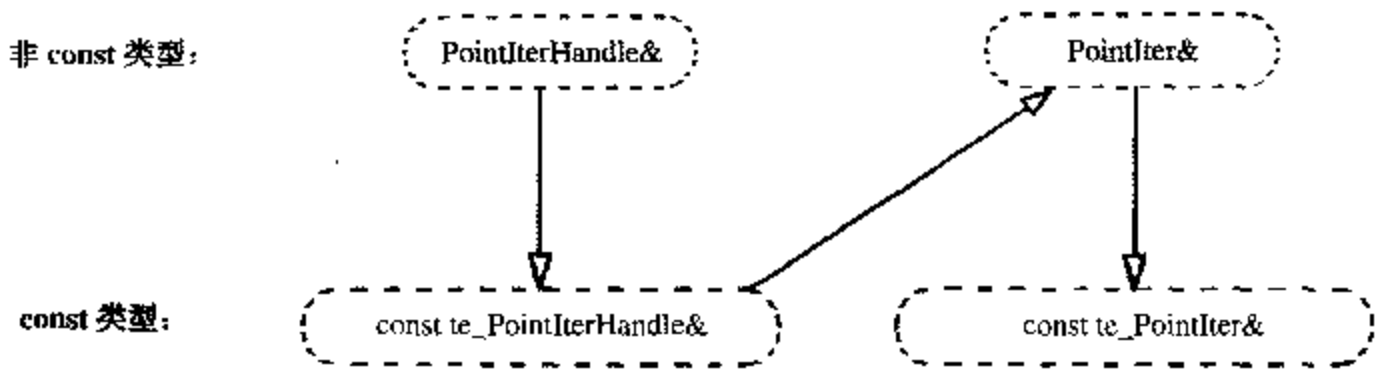


图 9-15 PointIterHandle 和 PointIter 的一个 const 正确的实现

抛弃 const 会破坏我们在本节中努力获得的所有利益。

9.1.7 公共的、保护的或私有的成员函数

准备给一般客户直接使用的成员函数必须声明为 public。像等式运算符或关系运算符这样的自由运算符可以实现为基本成员函数（见图 9-21）。如果这个基本成员函数不是公共的，那么从属的自由运算符必须声明为这个类的友元，这会减低可维护性，并使滥用成为可能（见 3.6 节）。

原 则

如果成员函数不是公共的，那么会使普通用户也要接触未绝缘的实现细节。

私有成员函数是准备在一个类的内部和给类的友元使用的。因为不鼓励组件之外的友元关系，所以，通常非虚私有成员函数相对于定义在.c 文件的文件作用域中的静态自由函数并没有多少优势（见 6.3.3 节）。私有非虚成员函数的一种常见用途是从一个另外的微小而频繁调用的内联函数中分解出复杂但很少需要的行为。例如，图 10-13 中的类 my_Stack 使用了私有非内联成员 growArray( )来实现公共的内联 push 方法：

```
inline
void my_Stack::push(int value)
{
    if (d_sp >= d_size) {
        growArray();
    }
    d_stack_p[d_sp++] = value;
}
```

如果不使用私有方法，我们将被迫要么以显著的性能代价非内联实现整个 push 方法，要么破坏封装让 growArray 方法成为公共的。设法内联实现整个 push 函数大概是不可能的。

当定义在一个派生类中的行为只被基类的成员和友元使用时，私有虚函数是有意义的。私有虚函数的一种潜在使用，可以在 6.6.3 节中描述的 Solid 类的 surfaceEquation 成员中找到

（见图 6-84）。

原　则

所有的虚函数和保护函数都是需要派生类作者考虑的事项。

保护成员函数是明确地指定给派生类作者的。保护成员函数迫使普通客户接触实现细节，通常最好避免使用（见 6.3.4 节）。虽然私有虚函数对派生类来说是不可访问的，但可以期望派生类作者给这些函数提供定义；在这个意义上，私有虚函数是例外的：

```
class Base {
  private:
    virtual void programMe();
};

class Derived : public Base {
  public:
    void programMe();     // The derived class itself cannot call this
                          // function through the base class interface;
                          // yet even clients of the derived class's public
                          // interface are able to access this function.
};
```

9.1.8　通过值、引用或指针返回

是否通过值来返回的问题涉及到该对象或参数列表内是否有某些东西适合引用。例如，下述语句没有合理的实现[1]：

```
pub_String& operator+(const pub_String& left, const pub_String& right);
```

正如我们在 8.3 节所介绍的那样，通过值返回一个对象保持了整体的封装，但运行时代价可能比返回一个指针或引用要大得多。对于非运算符函数来说，我们有额外的选择，可以通过参数列表返回一个对象。通过参数返回通常比通过值返回更有效，而且它仍然是完全封装的。图 9-16 展示了返回一个值（foo）的四种方法。

```
Foo v(...);                          // return by value
const Foo& r(...);                   // return by reference
const Foo *p(...);                   // return by pointer
int a(Foo *retVal, ...);             // return by argument

int a(Foo& retVal, ...);             // bad idea (see Section 9.1.11)
```

图 9-16　从一个函数返回一个值的四种方法

① 见 **meyers**，Item 23，82~84 页；Item 31，102~105 页。

在函数可能失败的情况下，通过值或引用返回可能不是一个可选项[1]。有时候我们可以通过指针返回值，它在失败时可以是0。另一个选择是返回一个整数状态表示成功或失败，并且通过参数列表返回对象本身。

指导方针

对于返回一个错误状态的函数来说，整数值0应该总是意味着成功。

对于返回一个int或某种枚举类型指示错误状态的函数来说，存在一种判断这个函数是否在工作[2]的方便方法，这种方法不必检查某个头文件就能确定这个特定函数的适当的成功值。按惯例，0状态表示成功，非0状态表示失败，并且特定的非0值可以给客户提供额外的信息。

原　则

通常函数成功工作只有一种方式，而它失败却有若干种方式；作为客户，我们可以不关心它为什么失败。

十分常见的是，客户不关心一个操作为什么会失败；在这种情况下一个简单的非0状态的测试就足够了，如图9-17所示的那样。在一些情况下，这个惯例可能允许我们避免包含一个额外的列举错误情况的头，从而减少不必要的编译时耦合。

对于非const成员函数来说，返回一个指向对象本身的非const引用总是一个可行的选择。返回一个指向对象内部部分的指针或引用潜在地限制了实现选择；应该仔细考虑它对封装的影响（见8.3节）。

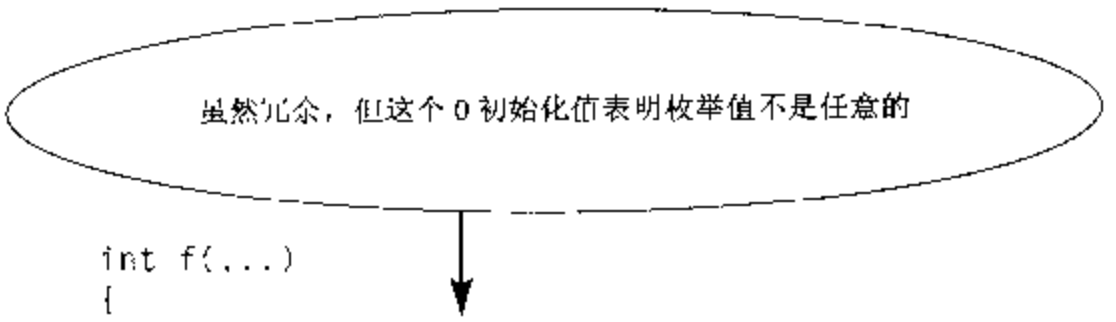
原　则

通过加载一个可修改的句柄参数返回一个动态分配对象，比通过非const指针返回那个对象更不容易产生内存泄漏。

对于像geom_Shape这样的多态对象（见图9-8），不可能通过值来返回。通过非const指针返回该对象的一个克隆（动态拷贝），会把重新定位的负担转嫁给客户，并容易产生内存泄漏。在这里，引用的使用是模糊和不适当的（见9.1.11节）。返回新分配多态对象的一个更好的方法是，传进一个指向句柄的指针（见6.5.3节），该句柄被明确地设计来保存一个指向基类geom_Shape的指针：

① 在一个类定义的“有效”状态中返回一个对象有时是有意义的。

② 注意，一个函数，如fclose，会执行一个操作并对那个操作的输出返回一个错误状态。一个函数，如isalpha，只是回答yes或no的问题，没有预想的成功或失败的概念。因此，fclose返回一个错误状态，而isalpha不会。




```
int f(...)
{
    enum { GOOD = 0, BAD, UGLY } status = GOOD;

    if (0 != g(...)) {
        status = BAD;
        // ...
    }

    if (GOOD == status && 0 != h(...
        status = UGLY;
        // ...
    }

    // ...

    return status;
}
```

图 9-17　对于所有返回状态的函数来说，0==成功

```
class geom_ShapeUtil {
    void create(geom_ShapeHandle *handle, const char *typeName);
        // Create a new shape of the type specified by typeName and
        // load it into the handle passed in via a non-const pointer.
};
```

指导方针

回答是或否问题的函数应该被适当命名（例如 isValid），并返回一个不是 0（“no”）就是 1（“yes”）的 int 值。

最后，明确地回答一个“yes”或“no”的问题的函数名若能指明这个事实——isAcute()、hasProtocol(id)、areParallel(line1,line2)等等，并且为“no”返回一个 int 值 0，为“yes”返回 1（别无其他），则对于函数是有帮助的。通过仿效像==这样的返回布尔值的内置运算符的行为，有助于使这些常用函数的语义更清晰。有时客户将在他们不需要的地方依赖这个 1 值：

```
if (1 == angle.isAcute()) { /* ... */ }
```

如果我们将这个值作为一个标记缓存在对象中，我们可能被诱惑去返回一个可能有某个非布尔值（例如 8）的屏蔽位。将一个非布尔值 x 转换为一个布尔值 y 和 y = !!x 一样简单。

isAcute 的一个适当的实现可能如下所示：

```
int geom_Angle::isAcute() const
{
    return !!(d_flags & ACUTE_MASK);
}
```

因为 ANSI/ISO 委员会已经将 bool 采纳为 C++的一个独特的整数类型，一旦这种新的基本类型成为普遍可用的，我们也许应该在这种情形下返回 bool 而不是 int[1]：

```
bool geom_Angle::isAcute()const
{
    return d_flags & ACUTE_MASK;
}
```

至少现在到一个布尔值的转换是隐式和自动的。

9.1.9 返回 const 或非 const

在努力消除不必要的限制的过程中，我们力争在维持 const 正确性的同时采用 const 操作数并返回非 const 结果[2]。

指导方针

避免将结果（从函数通过值返回的）声明为 const。

从一个函数通过值返回的结果是右值。就基本类型来说，将一个右值声明为 const 是冗余的、混乱的，并可能干扰模板实例化：

```
const int f(); // redundant use of const
```

一个用户自定义类型的右值（例如从一个函数通过值返回的一个对象）可以被非 const 成员函数操纵；一个通过 const 值返回的对象则不能。这后一种行为是语言的一个“角落”，它是有争议的，并且可能不能跨越现有的编译器和平台一致地实现。因为返回的值到底是一个拷贝，使用 const 右值与非 const 右值的概念一般是不必要的。

一般来说，通过 const 指针或引用返回一个值，比返回一个非 const 指针或引用对实现的限制更少。例如，在 8.3 节讨论的 PointArray 稀疏数组实现中，返回一个指向一个伪空对象的 const 引用是很方便的。如果该引用是非 const 的，那么我们将被迫在那个位置分配一个新对象。注意，const 成员函数一定不能返回违反 const 正确性的非 const 对象（见 9.1.6 节）。

---

① **stroustrup94**，11.7.2 节，254~255 页。

② **meyers**，Item 21，73~78 页。

9.1.10 可选参数或必要参数

只有一个函数体通常比有若干个重载版本更容易维护[1]。在大多数情况下，使用内联函数创建实际上允许可选参数位于一个参数表的中间的重载版本非常容易[2]。

原 则

默认参数可以是函数重载的一种有效替代品，尤其是在不涉及绝缘的地方。

图 9-18 将重载函数的使用与在构造函数调用中分解出公共代码的默认参数的使用进行了对比。如图 9-18（a）所示，分解若干重载构造函数的实现需要使用一个辅助函数 init，因为不能从一个构造函数有效地调用另一个构造函数[3]。这样的分解不允许我们利用构造函数的初始化列表。在图 9-18（a）中，用于从若干重载函数转发调用的内联函数的使用[有时称之为内联转发（inline forwarding）]消除了内嵌函数调用的潜在开销。同时，内联转发也否定了使单独的重载函数非内联实现的绝缘的价值。

图 9-18（b）阐释了一种比分解构造函数体更经济实惠的表示法。但是要注意，这两种实现在从一个 int 构造一个 geom_Point 时是不同的：

```
void g()
{
    geom_Point a = 5;
    a = a + 10;
}
```

图 9-18（a）中的实现不允许从一个单个的参数构造一个 geom_Point，因此阻碍了上述非直觉的初始化和隐式转换。使用两个默认参数来实现如图 9-18（b）所示的默认构造函数，会微妙地引入一个不希望的整数转换运算符，它允许上述代码悄悄地编译（见 9.3.1 节）。

默认参数可能比多重重载函数更能自我文档化、更简洁并且更容易被客户理解，因为它们在头文件中放置了更多的信息。照这样，默认参数和绝缘的目标是不一致的。若想了解有关如何在使用默认参数时减少编译时耦合的更多信息，参见 6.3.8 节。

默认参数的一个重要用途是允许开发者方便地添加参数到函数中，不会损坏任何先前存在的、使用了这些函数的程序。

指导方针

避免那些需要一个未命名临时对象的结构的默认参数。

---

① 见 **cargill**，第 2 章，32 页；第 3 章，65 页；第 4 章，87 页。

② **ellis**，8.2.6 页，142 页。

③ **meyers**，Item 24，85~87 页。

```
geom_Point {
    int d_x;
    int d_y;

  private:
    void init (x, y);

  public:
    geom_Point ();
    geom_Point (int x, int y);
    // ...
};

geom_Point operator+(const Point&,
                     const Point&);

inline
void geom_Point::init(int x, int y)
{
    d_x = x;
    d_y = y;
}

inline
geom_Point::geom_Point()
{
    init(0, 0);
}

inline
geom_Point::geom_Point(int x, int y)
{
    init(x, y);
}
```

(a) 使用重载函数

```
geom_Point
    int d_x;
    int d_y;


  public:
    geom_Point(int x = 0, int y = 0);

    // ...
};

geom_Point operator+(const Point&,
                     const Point&);

inline
geom_Point::geom_Point(int x, int y)
: d_x(x)
, d_y(y)
{
}
```

(b) 使用默认参数

图 9-18 分解公共构造函数代码

将用户自定义类型作为默认参数来传递是最麻烦的：不是所有的对象作为默认值都有意义。构建一个通过默认值传递进来的临时对象，就像通过值来传递一个对象一样，代价很高，应该避免。

9.1.11 通过值、引用还是指针来传递参数

通过值来传递用户自定义类型是不必要的，并且代价很高。这个习惯的开销非常大，以至于已经造成了这样的误解——C++语言本身很慢[①]。

① 更多的理由，见 **meyers**，Item 22，78~82 页。

次要设计规则

千万不要通过值传递一个用户自定义类型（例如 class、struct 或 union）给一个函数。

不要通过值传递一个用户自定义参数，而应该通过 const 引用来传递它。在没有全局变量时，在实践中二者语义本质上完全一样，但后者的运行时性能极佳。枚举类型和所有基本类型通过值来传递是最有效的。

当要通过参数列表来返回一个值时，有以下两种选择：

```
(1) void f(my_Object& result, ...); // return by non-const reference
(2) void f(my_Object *result, ...); // return by non-const pointer
```

在任何行得通的地方，我们更愿意使用语言本身来表达我们的意图，而不是依赖注释。C++语言定义规定，指针可以是空的而引用不可以。通过一个可修改的引用参数返回一个值使语义很清楚：接收该值的对象必须由该客户提供。任何重申这个要求的文档都是不必要和冗余的。因此没有必要对一个空引用进行测试——并且这样做无论如何也没有轻便的方法。所以，通过非 const 指针所返回的对象可被专有保存以使结果是真正可选的。也就是说，该指针总是在函数内测试，并且如果提供了一个空指针，这个结果不会被载入对象。

在另一个阵营[1]，传统理论不鼓励使用会修改它们的参数的函数：大家都知道这样的函数更难维护。记住，在历史上，一个系统的生存期内引起的大部分开销是在维护和增强上——不是最初的开发。允许函数通过引用修改它们的参数，会使维护一段不熟悉代码的软件工程师更难以判断一个（显然通过值）传递进一个函数的参数是否可能潜在地被修改。

可写引用的富有表现力的能力只有当你碰巧看到适当的头文件时才可用。只看图 9-19 中的客户代码，根本就无法弄清楚是什么导致了以“Laurel”初始化的 my_String 类型的变量 name 的值变成了（不正确的）“Hardy”。

真正修改它们的参数的函数相对较少。遵守只通过非 const 指针修改函数参数的指导方针，可以使从客户代码中发现这样的函数更容易。图 9-20 显示，对 name 操作的三个函数中只有一个能合理地修改它的值；在这个例子中，我们可以只访问一个地方而不是三个，这仅仅是因为我们遵循了这条指导方针。

指导方针

通过参数返回值要保持一致（例如，避免声明非 const 引用参数）。

① **stroustrup**，2.3.10 节，62 页。

```
void g(int i, int j)
{
    my_String name("Laurel");
    // ...
    int s = my_Stuff::funcX(name, 1);
    // ...
    int t = your_Problem::funcY(name, j);
    // ...
    int u = their_Thing::funcZ(name, i + j);
    // ...
    cout << "name = " << name << endl;
}

// Output:
//      name = Hardy
```

图 9-19 通过可写引用修改函数参数

```
void g(int i, int j)
{
    my_String name("Laurel");
    // ...
    int s = my_Stuff::funcX(name, 1);
    // ...
    int t = your_Problem::funcY(&name, j);
    // ...
    int u = their_Thing::funcZ(name, i + j);
    // ...
    cout << "name = " << name << endl;
}

// Output:
//      name = Hardy
```

图 9-20 只通过可写指针修改函数参数

图 9-21 说明了甚至在接受非 const 指针参数的函数体内，我们也只需要看函数的定义（而不需要看声明每个被调用函数的头文件），就可以推断出哪一个被调用函数可能修改那个参数。

```
void f(my_String *name, i, j)
{
    int s = my_Stuff::funcX(*name, i);          // should not mocify name
    // ...
    int t = your_Problem::funcY(name, j);       // potentially modifies name
    // ...
    int u = their_Thing::funcZ(*name, i + j);   // should not modify name
    // ...
}
```

图 9-21 通过可写指针返回的嵌套函数

从以往经验来看，通过指针传递对象和通过引用传递对象，在要进行用户自定义转换时不总是等价的。一直到 C++语言定义为 CFRONT 的 2.0 版本作出改变为止[1]，接受一个指向 your_Class 的非 const 引用的函数都允许其参数在通过参数返回赋值发生之前经历一次到临时变量的用户自定义转换。所以，值不会被返回给调用者，而且这个错误直到运行时才会被发现。比较起来，一个接受非 const 指针的函数决不会允许用户自定义转换发生；类型错误将总是在编译时就被发现。后者是我们期望的行为，它和成员运算符（例如运算符 = ）的行为是一致的，成员运算符不会隐式转换它们修改的对象（见 9.1.2 节）。

当通过非 const 指针传递要修改的对象时，标准指针转换会继续按预期进行，这一点值得注意。换句话说，传递一个派生对象的地址给一个函数（该函数接受一个指向其公共基类中的一个的指针）是有意义的，并且转换将隐式发生。要求客户程序传递可修改的参数地址，只会禁止那些不需要的用户自定义转换。幸好，当用户自定义转换导致一个临时变量被绑定到一个非 const 引用参数时，大多数当前的编译器都至少会发出警告。

C 语言不允许直接修改函数参数；C++则允许。在第一次使用 C++时，标准 C 程序员会经常忘记在一个引用参数前的适当地方插入 const 限定符，使读者很纳闷该函数的作者到底想还是不想让这个参数可修改。避免在函数参数中任何使用非 const 引用，可以使意图明确（或者使缺陷立刻显而易见）。

通过可修改的引用来传递函数参数的支持者往往会争辩：一个引用是可以自我文档化的，因为是否必须提供一个有效的对象是毫无疑问的，而一个指针参数却使这种可能性存在。再者，了解预期行为的惟一方法就是看每个被调用函数的头文件（那可能意味着许多头文件）。

迫使客户传递一个可修改参数的地址常常要求客户键入特别按键“&”。但是，在广而告之一个函数调用可能潜在地修改它的参数时，这个特别按键一字千金。并且因为大多数函数不会修改它们的参数，所以几乎所有的函数调用都可以很快排除疑虑。

通过非 const 指针返回一个参数是一种通用并且上下文独立的技术；语法异常警告客户注意参数的特殊性质。通过非 const 引用将一个可修改参数传入一个函数的任何表示法上的方便，都不如避免代价高昂的意外重要。但是，非 const 引用参数在运算符函数（例如 operator+=）和被良好确立的习惯语（例如流，除非流可以修改，否则就没有意义）中仍然拥有它们的位置。和通过一个可修改引用返回某个不知名对象的方法相比，流惯用语的上下文使得它们的用法的语义比较明确。

指导方针

避免将一个函数的任何参数的地址存储在函数结束后仍会保留的位置；改为传递该参数的地址。

① **stroustrup94**，3.7 节，86 页。

另一个相关的问题是，通过 const 引用传递一个用户自定义类型是如此的常见，以至于我们可能从不怀疑一个特定值是一个左值的重要性。考虑图 9-22 中的场景，我们定义了一个无限精度整数类型 my_BigInt，它可以从一个基本 int 类型构建。my_BigIntSet 是一个同类集合，它只存储提供给它的 add 函数的对象的地址。假设一个没有经验的用户试着创建了一个函数 g，g 加了三个整数到集合中。每个整数都被隐式转换为一个临时变量 my_BigInt，这些临时变量仅被保证在函数返回之前保持有效：从那以后直到退出创建该临时变量的作用域，其间任何时候都可以析构该临时变量[①]。如果在调用 my_BigIntSet 的 isMember 方法时第二个临时变量 my_BigInt 已不再有效，那么经过一个坏指针值的一个内存引用可能很容易导致这个程序崩溃！

```
class my_BigInt {
    // ...
  public:
    my_BigInt(int i);
    // ...
};

class my_BigIntSet {
    const my_BigInt **d_set_p;
    int d_size;                          // physical size
    int d_length;                        // cardinality
  public:
    // ...
    void add(const ni_BigInt& bi);  // bad idea: should pass by pointer
       // Stores the address of this object in the set.

        int isMember(const my_BigInt& bi) const;
            // Returns 1 if bi is a member of the set; else 0.

    };

void g()
{
    my_BigIntSet set;
    set.add(1);                          // Address of temporary my_BigInt Added
    set.add(2);                          // Address of temporary my_BigInt Added
    set.add(3);                          // Address of temporary my_BigInt Added
    set.isMember(2);                     // core dump?!
}
```

图 9-22 在一个函数中保留一个引用参数的地址

如果不仔细检查类定义，客户绝对不会注意该对象的地址（而不是该对象的一个拷贝）会被保留。如果我们把 my_BigIntSet 的 add 函数定义改为接受一个 const 指针，那么我们就警

① **ellis**，12.2 节，268 页。

告了客户，这个函数认为左值是重要的，并且同时直接在声明本身说明了这个事实[1]。my_BigIntSet 的修改过的用法模式如图 9-23 所示。

```
class my_BigIntSet {
    // ...
  public:
    // ...
    void add(const ni_BigInt *bi);
    // ...
};

void g ()
{
    my_BigIntSet set;
    set.add(1);                  // compile time error!
    set.add(&2);                 // compile time error!
    // ...
```

图 9-23 使对一个左值的需求明确化

将一个参数的地址存储到函数中是不好的方式。如果这个参数通过值来传递，那么它表现为一个局部自动变量，并且一旦该函数返回，地址就会变得无效。如果该参数是通过 const 引用来传递的，那么我们不能保证它不涉及一个临时变量。通过 const 指针而不是通过 const 引用来传递这个参数，抑制了一个临时变量的隐式创建，这是我们计划继续使用那个地址渴望的行为。当从上下文（例如一个迭代器）看显然必须存储对象的地址时，这条指导方针的例外确实会出现在非常常见的习惯用语中。注意，当一个函数为了以后的修改存储了一个非 const 参数的地址时，本节中介绍的两条指导方针（例如可修改+左值）都适用；在这种情况下，对象应该**总是**通过非 const 指针来传递。

次要设计规则

永远不要企图删除一个通过引用传递的对象。

除了修改之外，删除一个对象的函数也应该总是使用一个指向那个对象的非 const 指针，而决不要使用一个非 const 引用。为了删除一个对象，我们必须提供一个指向删除运算符的指针。提取一个要被删除的对象的地址很容易出错；一些编译器（例如 CFRONT）会产生一个警告信息，误导开发者增加一个额外的赋值（或者更糟糕，一个 cast）给一个指针变量。更值得注意的是，C++语言规范允许编译器调整一个被删除指针的值[2]（例如调整为 0）。在 C++

① 这个建议称为 Linton 惯例，在 **murray**，9.2.4 节，213~215 页中介绍。

② **ellis**，5.3.4 节，63 页。

语言中不允许空的（或无效的）引用[①]。

使用指针代替引用参数来获得本节中提及的语义特性，还有另一种维护方面的好处。如果一个先前没有修改的函数或接受一个参数的地址的函数突然改为这样做，那么那个函数的所有客户都将在被迫检查了他们的代码之后才能重新编译。这恰恰是我们所需要的！利用语法兼容性进行如此显著的语义修改，可能会导致微妙的错误和很不愉快的意外。

9.1.12 将参数作为 const 还是非 const 传递

传递给函数的指针或引用无论何时将对象称为 const，都扩大了可能利用这个函数的潜在客户的观众。

指导方针

如果无论何时一个形参（parameter）通过引用或指针传递其实参（argument）给一个函数，该函数都既不修这个改实参也不存储它的可写地址，那么这个形参就应该声明为 const。

作为一条规则，无论何时我们可以合理地把一个指针或引用参数作为 const 来传递，我们都应该这样做[②]。

指导方针

避免将通过值传递给一个函数的形参声明为 const。

一个通过值传递给函数的实参是一个拷贝。将形参声明为 const 会使它的值在该函数体内不可变：

```
void f(const int i)             // bad idea
{
    // ...
    ++i;                        // compile-time error
    // ...
}
```

一个局部拷贝是否被修改是函数的一个实现细节；将其声明为 const 就在接口上暴露了这个决定，这不仅危及绝缘而且损害可读性。这不是一个用户自定义类型的问题，因为无论如何我们决不会通过值来传递它们（见 9.1.11 节）。

① **cargill**，第 6 章，125 页；**ellis**，8.4.3 节，153 页。

② 见 **meyers**，Item 21- Item 22，73~82 页。

指导方针

考虑将那些会激活可修改访问的形参（也许除了那些有默认实参的形参）放在那些通过值、const 引用或 const 指针来传递实参的形参之前。

除了带有默认实参（在函数被使用之后加入的）的（可选）形参之外，其他所有允许其实参被修改的形参，都应该放在通过值、const 引用或 const 指针来传递其实参的形参之前。除了会使在哪里可找到可修改的实参这一点变得更加统一之外，这个建议（的其他方面）显然是随意的；但是，它是一种独立于语言的传统风格，早于 C++（甚至 C），多年来已证明是有用的。

9.1.13 友元或非友元函数

友元关系，甚至是在一个单个组件内，也会影响维护开销。

原　则

避免不必要的友元关系（甚至在同一组件内部）可以提高可维护性。

在使一个单独运算符成为一个友元之前，要考虑是否有一个基本成员函数可以用来实现该运算符。例如，我们经常可以根据基本成员运算符+=来实现自由运算符+（见 3.6 节），而不必使运算符+成为一个友元。类似地，一个公共成员函数 compare 可以用来实现所有六个自由等式运算符和关系运算符（==、!=、<=、<、>=、>）（见 9.1.2 节）。一般地，一个有私有访问权的迭代器类可以用来实现一个容器类型（例如集合、列表等）的自由输出运算符<<，从而避免了单独的友元关系，并增强了用户可扩展性。

指导方针

避免给单独的函数授予友元关系。

通常，无论何时我们决定需要一个自由运算符，我们都应该知道可以使用什么基本函数来实现这个运算符。使一个自由运算符成为一个友元可能会损害封装（见 3.6 节）。

9.1.14 内联或非内联函数

从 6.2.3 节中我们知道，内联函数会影响绝缘。除了会暴露实现之外，大型的内联函数还会增加可执行程序的大小，潜在地使得一个集成系统的运行比那些将其中一些函数声明为非内联的相同系统更慢。如果绝缘不是一个问题，那么第一个问题就是："由函数体产生的对象代码比非内联函数调用更大还是更小？"如果内联对象代码不比一个函数调用大，内联就不

会增加可执行程序的大小。

指导方针

如果函数体产生的对象代码大于同样的非内联函数调用本身所产生的对象代码，那么应该避免将该函数声明为 inline。

对于只获取和设置数据成员的函数来说，使用一个内联函数而没有首先获得性能数据通常是合理的。对于比相应的非内联函数调用产生更多对象代码的函数体来说，系统级的性能分析应该优先于定义该函数为内联的决策。传递额外的实参给一个函数，会增加为一个非内联函数调用产生的代码数量。所以，一个接受若干实参的内联函数，可能证明在建立环境文件之前稍大一些的函数体是合理的。

如果一个函数被频繁调用且性能要求很高，那么要问的第二个问题就是："可以从多少不同的地方调用该函数？"如果对该函数的访问有限制，并且已知只能从少数几个不同的地方调用，那么在可执行文件大小方面，内联不太可能是一个问题。如果该函数较大并且可以从许多地方调用，那么该函数不太可能是内联的候选者。

指导方针

若你的编译器不会产生内联函数，那么应避免将一个函数声明为 inline。

最后，声明内联只是给编译器的一个提示，没有办法确保一个函数会被真正内联。无论何时我们提取一个声明为内联的函数的地址，我们都会迫使编译器在使用该地址的编译单元中产生该函数的一个静态（非内联）版本。如果一个声明为内联的函数太大或者太复杂，那么它可能不会被内联；控制这一点的算法依赖于编译器。

当一个函数没有内联时，编译器会在每个使用该内联的编译单元中都定义一个该内联函数的静态版本。这些多重静态拷贝可能导致可执行程序变得更大且运行也更慢（和假定该函数声明为非内联相比）。幸好，通常有办法请求一个编译器报告没有被内联的函数[①]。

一个动态绑定的函数调用不能产生内联；但是，当虚调用机制因为使用作用域标识符（::）而失效时，虚函数调用可以被内联。当编译器可以确定具体对象的确切类型时（例如当函数是从对象本身而不是通过一个指针或引用调用时），虚函数调用也可以被内联。在任何一种情况下，编译器都被迫要实现虚函数的一个非内联版本，以便将虚函数的地址存储在虚表中。如果我们不小心，可能会产生这个函数的远不止一个的静态拷贝（见 9.3.3 节）[②]。

① 见 **meyers**：Item 33，107~110 页。

② 也可参见 **murray**，9.13.2 节，244 页。

9.2 在接口中使用的基本类型

在第 3 章，我们讨论了用户自定义类型方面的 uses 关系。本节我们将研究在函数接口中不同基本类型的使用。

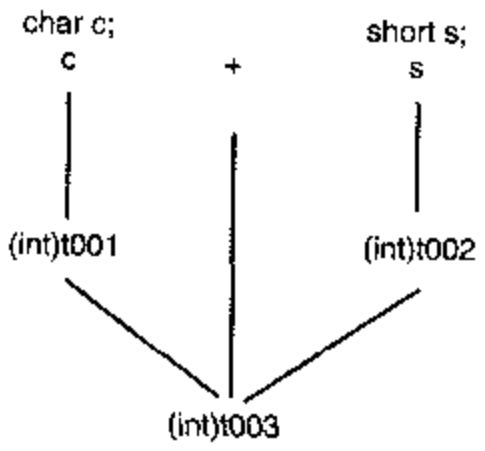
9.2.1 在接口中使用 short

C++语言要求，声明为类型 char 或 short 的变量，在参与一个表达式之前，要被自动提升为 int 类型。也就是说，除了测定它们的大小的函数（sizeof）或获取它们的地址的函数（单用&）之外，在一个表达式中不会直接使用 char 类型或 short 类型值。

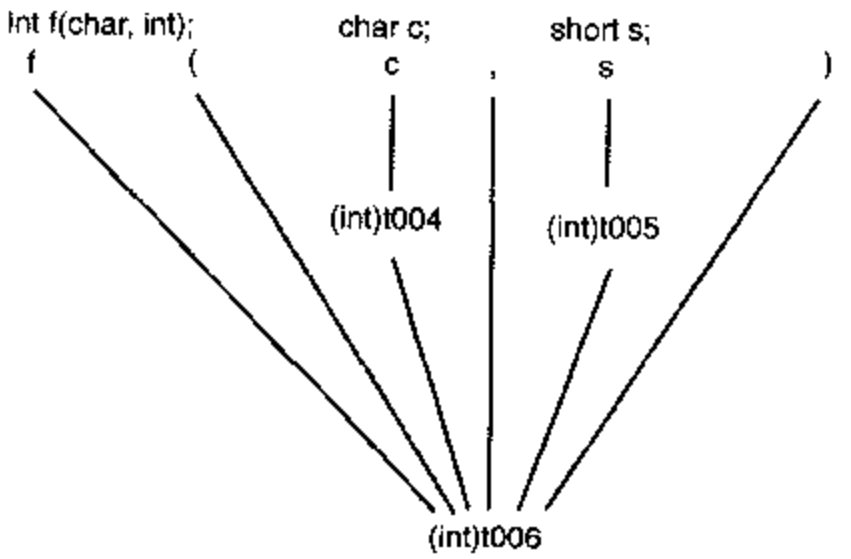
指导方针

避免在接口中使用 short；改为使用 int。

图 9-24（a）举例说明了一个 char 或 short 被用于一个双目表达式之前首先会自动提升为一个 int 类型的临时变量。如果不考虑任何重载函数调用解析，同样的到 int 值的自动提升也会隐式地出现在一个函数调用过程中，如图 9-24（b）所示。在 C++中，int 类型一般对应于底层计算机硬件支持的基本整数大小。对于大多数可商用的工作站来说，一个 int（至少）是 32 位的[1]。



（a）在双目操作中的整数提升



（b）在函数调用中的整数提升

图 9-24 整数提升的结果

图 9-25 举例说明了一个类，它在其接口中使用了 short 而不是 int。为什么我们可能要做这样一件事呢？其动机来自于一个愿望：直接在声明中表达意图而避免借助注释。如果我们

① 以后我假设机器是 32 位（或更大）的体系结构。如果你在一个 16 位的机器上或在一个嵌入式系统上工作，那么在本节中的一些语句将不能使用。

将一个参数声明为一个 short，就没人能传入任何更大的类型值，这样我们就不必自己进行检查，对吗？

```
class my_Point {
    short d_x;
    short d_y;

  public:
    // CREATORS
    my_Point(short x, short y);
    my_Point(const my_Point& p);
    my_Point();

    // MANIPULATORS
    my_Point& operator=(const my_Point& p);
    void x(short x);
    void y(short y);

    // ACCESSORS
    short x() const;
    short y() const;
};
```

图 9-25 在接口中使用 short 整数（坏主意）

原 则

在代码中直接明确地说明设计决策（而不是依赖于注释）是一个设计目标；设计能安全使用和容易维护的健壮接口偶尔会与这个目标形成竞争关系。

这个问题的实际情况是，在进行维护时，头文件中的建档信息（只有当直接查看头文件本身时才有用）的作用是有限的（见 9.1.11）。客户将传递一个整数数值或表达式，不会注意我们企图如何在头文件中将其存档；将整数声明为一个 short 只会引起截断出现在函数外部而不是内部，使得函数自身不可能发现一个溢出错误。对于客户，感觉是一样的——该函数无法工作。

考虑下列问题的答案：

（1）在接口中使用一个 short 能保证编译时出现的溢出不会在运行时出现吗？

不能保证。C++语言允许算术溢出在运行时悄悄出现。

（2）在接口中使用 short 能顾及溢出检测吗？

不能。如果接口接受了一个 int，我们至少可以发现实现范围之外的 int；传递一个 int 起码可以允许我们设置前置条件。

（3）在接口中使用 short 是否对实现的封装起作用？

将 short 暴露在接口中，限制了任何实现可提供的 short 的大小，并消除了我们检测溢出的能力；限制实现选择是降低封装的一个症状。

（4）在接口中使用 short 是提高还是降低了效率？

如果有什么不同的话，实参可能必须使其高位被屏蔽掉，这需要额外的工作，因而降低了运行时效率。

（5）在接口中使用 short 会干扰重载函数解析吗？

会。按照 C++语言的规则，将一个 int 转换成一个 short，就像将一个 int 转换成一个 double 一样，是一种标准转换。也就是说，如果两个函数都命名为 f，一个接受 short，另一个接受 double，那么函数调用 f(10)将会产生二义性。

（6）在接口中使用 short 会干扰模板实例化吗？

会。按照模板实例化的规则，一个模板的类型必须与实参严格匹配；以一个 short（例如图 9-26 中的 pub_BitVec）为参数的模板，当以一个整数数值或任意编译时常量整数表达式声明时，将无法被实例化：

```
BitVec<2> v1;                  // expecting short got int
BitVec<short(2)> v1;           // ok.
BitVec<5 - 3> v2;              // expecting short got int
BitVec<short(5 - 3)> v1;       // ok.
```

```
template<short N>
class pub_BitVec {
  int d_bits : N;

public:
  BitVec();
  int operator[](int i);
  void set(int i);
  void clear(int i);
  void toggle(int i);
};
```

图 9-26　以 short 为参数的模板类

9.2.2　在接口中使用 unsigned

C++语言要求，涉及一个 unsigned 整数的双目运算符在执行操作之前，首先要将另一个整数转换成 unsigned。通常这不是一个问题；但是，当它是一个问题的时候，要调试它一点也不容易。

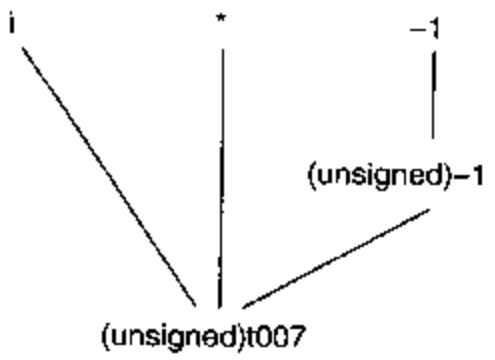
指导方针

避免在接口中使用 unsigned 整数；改为使用 int 整数。

图 9-27（a）举例说明了，当一个 signed 值和一个 unsigned 值被包含在一个双目操作中时，signed 数的位模式被悄悄地重新解释为一个 unsigned 数，没有产生任何实际的临时变量。

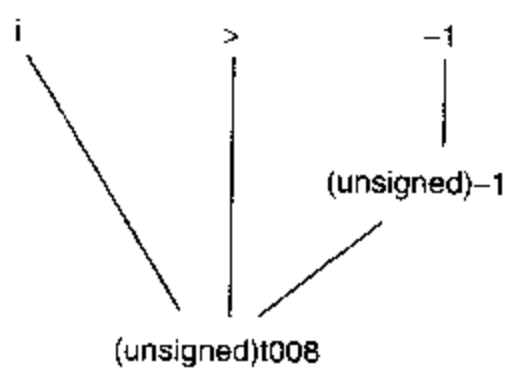
对于大多数整数表示法，结果是一个 unsigned 的最大数（例如：$2^{32}-1=4294967295$）。然后这个数字被乘以 3，导致 unsigned 溢出；于是结果输出为（3*（$2^{32}-1$））mod $2^{32}=2^{32}-3$。在图 9-27（b）中运用了完全相同的推理。unsigned 值又一次导致 signed 值的位模式被重新解释为一个巨大的 unsigned 值，我们做了比较，多数情况下将产生意想不到的结果。

```
#include <iostream.h>
main()
{
    unsigned int i = 3;
    cout << "3 *  1 - "
         << i *  1  << endl;
}

// Output:
//    john@john: a.out
//    3 * -1 = 4294967293
//    john@john:
```



（a）在算术表达式中使用 unsigned

```
#include <iostream.h>
main()
{
    unsigned int i = 3;
    cout << "(3 > -1) = "
         << (i > -1) << endl;
}

// Output:
//    john@john: a.out
//    (3 > -1) = 0;
//    john@john:
```



（b）在逻辑表达式中使用 unsigned

图 9-27　在双目表达式中混合使用 int 和 unsigned

```
class my_Array {
    int *d_array_p;
    unsigned short d_size;     // bad idea: Short used in implementation
                               //           only, but see Section 10.1.2.
  public:
    // CREATORS
    Array(unsigned int size);
    Array(const Array& array);
    ~Array();

    // MANIPULATORS
    Array& operator=(const Array& array);
    int& operator[](unsigned int i);

    // ACCESSORS
    int operator[](unsigned int i) const;
    unsigned int size() const;
};
```

图 9-28　在接口中使用 unsigned 整数（坏主意）

有人可能会争辩，认为在混用负整数和 unsigned 整数时，应该得到应得的东西。也许吧——在**我们**这样做的时候。但是请思考图 9-28 中显示的看起来无害的 my_Array 类。

作为 my_Array 类的一个客户，我已经编写了图 9-29 中的函数，它使用了 my_Array 的一个实例，并且打印它的指定宽度的向前移动平均值。作为一个负责任的开发者，我飞快地写好图 9-30 中所示的小测试程序来检验我的函数是否起作用——但它不起作用。

```
#include <assert.h>
#include <iostream.h>
void printForwardMovingAverage(const my_Array& a, int width)
{
    assert(width > 0);
    const int N = width - 1;
    int total = 0;
    for (int i = -N; i < a.size(); ++i) {
        if (i + N < a.size()) {
            total += a[i + N];
        }
        cout << i << '\t' << double(total)/width << endl;
        if (i >= 0) {
            total -= a[i];
        }
    }
}
```

图 9-29　无害的客户函数输出向前移动平均值

```
// test.c
# include <stdlib.h>  // atoi()

main(int argc, char *argv[])
{
    const int SIZE = argc > 1 ? atoi(argv[1]) : 4;
    const int WINDOW = argc > 2 ? atoi(argv[2]) : 2;
    my_Array array(SIZE);
    for (int i = 0; i < SIZE; ++i) {
        array[i] = 1;
    }
    printForwardMovingAverage(array, WINDOW);
}
```

图 9-30　函数 printForwardMovingAverage 的测试驱动函数

我所期望的对默认值（一个 SIZE 为 4 的数组，它包含所有宽度为 1 的 WINDOW 和一个宽度为 2 的 WINDOW）的输出应该如图 9-31（a）所示；但实际情况令人失望，如图 9-31（a）所示。

即使我们很懂得而不至于混合 unsigned 和 int，也不会总是检查每个返回的整数值的头。在这种情况下，size ()函数可以帮助我们。问题仍然是将一个负数与一个 unsigned int 比较通常会出错，如图 9-32 所示。

```
john@john: a.out
-1      0.5
 0       1
 1       1
 2       1
 3       0.5
 john@john:
```

(a) 期望的输出

```
john@john: a.out
john@john:
```

(b) 实际的输出

图 9-31 printForwardMovingAverage 函数的测试驱动程序的输出

```
#include <iostream.h>
main()
{
    my_Array a(10);
    cout << "size = " << a.size() << endl;
    if (a.size() > -1) {
        cout << "size is positive or zero." << endl;
    }
    else {
        cout << "size is negative!!!" << endl;
    }
}

// Output:
//      john@john: a.out
//      size = 10
//      size is negative!!!
//      john@john:
```

图 9-32 一个 unsigned int 返回值与一个负 int 值的比较

在这种情况下，我们需要做的所有修复工作是用

```
for (int i = -N; i < a.size(); ++i) {
```

替换图 9-29 中的

```
const int S = a.size(); // work-around for returning unsigned
for (int i = -N; i < S; ++i) {
```

语句。这样函数就能正确工作了。

原 则

有时注释比直接在代码中表达一个接口决策要更好（如，unsigned）。

在接口中使用 unsigned 而导致的错误具有破坏性，而且非常难以发现。如果用一个调试工具来调试这个问题，则图 9-32 中的 if 语句必须断开。推测该函数的返回值是声明为 unsigned 的，并且一个双目比较运算的结果是将其他某个负值隐式地转化为一个正数，这种推测通常

是一种误解。

考虑下列问题的答案：

（1）在接口中使用 unsigned 类型是否能确保在编译时的负数在运行时不被传递？

不能。C++语言允许位模式在运行时被重新解释。

（2）在接口中使用 unsigned 类型是否可允许存在检查负数的可能性？

是的，但是你必须在内部将 unsigned 类型强制转换到 int 类型。

（3）用 unsigned 类型是否会提高或降低运行时效率？

一般没有影响。

（4）使用 unsigned 类型会增加所能存储的正数的大小吗？

是的——增加 1 位。这额外的 1 位几乎没有用。如果这额外一位被使用，那么当 unsigned 被转换回 int 时，会有丢失数据的危险（见 10.1.2 节）。

（5）在接口上使用 unsigned 类型会增加或减少用户出错的可能性吗？

会增加出错的可能。如果没有查看头文件，就没有安全优势，因为转换是悄悄进行的。在一个包含负整数（int）的表达式中幼稚地使用一个 unsigned 返回值，会引起客户代码在运行时中断。

（6）在接口中使用 unsigned 类型是否对实现的封装或不封装起作用？

在接口中暴露 unsigned 类型可以有效地限制任何实现提供的值，因而削弱了封装。

（7）在接口中使用 unsigned 类型会干扰重载函数的解析吗？

是的，按照语言的规则，将一个 int 值转换为一个 unsigned 值是一种标准的转换，就像将一个 int 值转换为一个 double 一样。也就是说，如果两个函数的命名都是 f，其中一个接受 double 值，而另一个接受 double 值，那么 f(10)调用是二义性的。

（8）在接口中使用 unsigned 类型会干扰模板的实例化吗？

是的，我们这里所遇到的问题与我们用一个 short 参数化一个模板时所遇到的问题是出于完全相同的原因（见 9.2.1 节）。

9.2.3 在接口中使用 long

尽管在本书中我们假设一个 int 整数至少拥有 32 位，事实上只需要 16 位就可以满足 ANSI 的 int 类型的需要[①]。如果你是在 16 位的机器上工作，下面的指导方针显然是不适用的。

指导方针

在接口中避免使用 long；改为 assert(sizcof(int) >= 4)，并使用 int 或一个用户自定义的大整数类型。

① **ellis**，3.2.1c 节，28 页。

C++语言定义了一个 long 整数类型，它至少与一个 int 一样大。一个 long int 意味着“你拥有的最大的整数”；在 16 位的机器上，一个 long 整数可能是 2 个字（32 位）。在大多数可商用的 32 位工作站的编译器上，一个 long 整数是一个单个 32 位字。在 64 位体系结构中，一个 int 整数可能继续被设定为 32 位，以便与现有程序兼容，而一个 long 整数可能是 64 位。如果移植性是一个问题，那么任何假设一个 long int 是超过 32 位的做法都是失败的方法（不是你要移植到的每台机器都有一个 64 位 long int）。

图 9-33 展示了一个在接口中使用 long int 而不是 int 的组件。为什么我们要做这样一件事呢？通常这个问题的回答是：“我要保存机器所能拥有的最大的数”。对于在小机器上开发的小项目来说，这个理由可能是充分的。对于在多种平台上的工业用工作站上运行的大型项目来说，int 要么是足够大了，要么它还不够——如果你不能确定，那么它就是不够。幸好，C++语言使我们可以自己定义一个更大的整数类型。

```
class my_Point {
    long int d_x;
    long int d_y;

  public:
    // CREATORS
    my_Point(long int x, long int y);
    my_Point(const my_Point& p);
    my_Point();

    // MANIPULATORS
    my_Point& operator=(const my_Point& p);
    void x(long int x);
    void y(long int y);

    // ACCESSORS
    long int x() const;
    long int y() const;
};
```

图 9-33 在接口上使用 long 整数

无论何时我们在一个期望 int 值的地方初始化、赋值或传递一个 long int 值，我们都在强迫进行一个会潜在失去信息的标准转换（如果不是，那么在第一个地方就没有理由使用 long）。编译器可能会警告客户这是“有丢失”的转换。如图 9-34 所示。我们也许总能告诉我们的客户通过分散代码来抑制这些警告的出现——只是个玩笑！

考虑下列问题的答案：

（1）在接口中使用 long 类型能确保有比 int 更强的能力吗？

不是在所有的平台上都能。一个 int 和一个 long 经常大小是一样的。如果你依赖这种增加的能力，那么你的代码将不能移植。

```
int r(int x, int y);

void g()
{
    my_Point p(3, 2);              // fine, int converted to long.
    int j, i = p.x();              // warning: long assigned to int
    j = p.y();                     // warning: long assigned to int
    double d = r(p.x(), p.y());    // warning: argument 1: long passed as int
                                   // warning: argument 1: long passed as int
}
```

图 9-34 混合 int 类型和 long 类型

（2）在接口中使用 long 类型会妨碍可用性吗？

是的，一个潜在大量的警告信息可能会“淹没”客户程序（见问题 3）。

（3）在接口中使用 long 类型会干扰重载函数的解析吗？

是的。根据语言的规则，将一个 int 类型转换为一个 long 类型是一种标准的转换，就像将一个 int 转换为 double 一样[①]。也就是说，如果两个函数的名字都是 f，一个接受 long 类型而另一个接受 double 类型，那么 f(10)调用将是二义性的。

（4）在接口中使用 long 类型会干扰模板实例化吗？

是的。我们所遇到的问题与我们用一个 short 类型参数化一个模板时所遇到的问题出于完全相同的原因（见 9.2.1 节）。

9.2.4 在接口中使用 float、double 以及 long double

在下列三种浮点类型中，C++语言会进行浮点运算：

- float
- double
- long double

指导方针

考虑在接口中对于浮点类型只使用 double，除非有强制性的原因才使用 float 或 long double。

从历史上看，C 语言要求所有的浮点表达式都是 double 类型的，并且不支持 long double 类型。ANSI C 引入了用 float 值直接进行算术运算的能力。大多数 C 库程序调用都以 double 类型传递并返回一个浮点值。现在，大多数计算机硬件都进行了优化，使得 double 浮点运算尽可能地快。事实上，在我的机器上，一个精确的 double 乘法比一个 integer 乘法（被实现为一

① 从一个 int 到 long 的转换不是一种整数的提升，而是一种标准的转换。将一个 int 转换为一个 long 和它的（const/volatile）等价物，只是一种“没有损失”的语言中的标准转换。

个子程序）要快一个数量级。

原　则

在实际出现的大多数情况下，为了在接口中能表达整数和浮点数，所需要的惟一基本类型分别是 int 和 double。

一致性、错误检测、运算符重载以及模板实例化等在整数类型中出现的问题，在浮点运算中也同样会遇到。

9.3 特殊情况函数

有一些特殊的成员函数需要讨论。转换运算符（即，单参数构造函数和 cast 运算符）以及编译器产生的函数（例如，拷贝构造函数、赋值运算符，特别是析构函数）需要特别的说明。

9.3.1 转换运算符

隐式转换会影响类型安全，会引入二义性，并且一般来说会增加维护程序的开销。任何时候创建一个只有一个参数的构造函数，都会激活一个隐式的用户定义的转换。定义一个转换运算符而不是一个构造函数，在本书中称为 **cast 运算符**，也会激活一个隐式转换。每种形式的例子都可以在图 9-35 中找到。

```
pub_String {
    // ...
  public:
    pub_String(const char *cptr, int maxSizeHint = 0);  // "cast constructor"
    // ...
    operator const char *() const;                      // "cast operator"
};
```

图 9-35　C++中用户自定义转换运算符的两种形式

原　则

会激活隐式转换的构造函数，尤其是从广泛使用的类型或基本类型（如 int）的转换，会破坏由强类型所提供的安全性。

只接收单个参数的构造函数（有时称做 **cast 构造函数**）通过激活一个意外的转换，促使意外情况的出现。考虑图 9-36 给出的二维表组件。

```
// d2_table.h
# ifndef INCLUDED_D2_TABLE
# define INCLUDED_D2_TABLE
```

```
class d2_Entry;
class d2_RowIter;
class d2_ColIter;
class d2_Table {
    // ...
    friend d2_RowIter;
    // ...
  public:
    d2_Table();
    // ...
};

class d2_RowIter {
    // ...
    friend d2_ColIter;
    // ...
  public:
    d2_RowIter(const d2_Table& table);  // takes a d2_Table
    operator const void *() const;
    void operator++();
};

class d2_ColIter {
    // ...
  public:
    d2_ColIter(const d2_RowIter&);      // takes a d2_RowIter!!!
    operator const void *() const;
    void operator++();
    const d2_Entry& operator()() const;
};

#endif;
```

图 9-36　二维的 d2_table 组件的概貌

该组件的设计目标是，一个客户可对表使用一个行迭代器，并且对于每一个行上的位置，对那个行迭代器重新应用一个新的列迭代器。

正如图 9-37 中的函数所显示的那样，编辑器剪切和粘贴会引入错误：在指定行上的 cit(t) 本应该是 cit(rit)。只要我们的代码是“类型安全的”，在编译时，就可以更好地检测这样的错误——但不是在这里！实际发生的情况是，第二个迭代器的每次实例化都迫使 d2_Table,t，隐式转换为一个 d2_RowIter 类型的无名临时变量（它碰巧位于表的第一行上）。无法保证当列迭代器运算时，这个临时行迭代器将仍然保持为有效。但是如果它保持有效，则表的所有行的内容看起来都与第一行是相同的。

一个接受用户自定义类型的 cast 构造函数在实践上比接受单个基本类型（尤其是一个整数类型）的构造函数更不可能成为问题。例如，考虑图 9-38 所描述的情形。一个 gr_Graph 为客户程序提供通过名称或 id 号查找特殊节点的能力。不幸的是，一个 gr_NodeId 知道如何从一个任意整数来构造自己。由 gr_Graph 的重载的 lookupNode 方法提供的明显的类型安全，对在函数 f 中检测错误几乎没有帮助。问题是在所指示的行上的额外的“*”将字符串名 name[0]

转换成了第一个字符的 ASCII 的值；该值被提升为一个 int 并隐式地转换成某种伪 gr_NodeId。任何人都能猜想到下一步会发生什么。本应在编译时就检测到的错误使得 gr_NodeId 不能激活一个从整数类型的隐式转换[图 9-18（b）中，一个类似的问题是源于默认参数的使用]。

```
void g(d2_Table& t)
{
    for (d2_RowIter rit(t); rit; ++rit) {
        for (d2_ColIter cit(t); cit; ++cit) {       // <-- oops!!! "cit(t)"
                                                    // should be "cit(rit)"
            cout << cit() << endl;  // print (ith row, jth col) table entry
        }
    }
}
```

图 9-37 （误）用 d2_table 组件

```
class gr_Node;

class gr_NodeId {
    int d_index;

  public:
    // gr_NodeId(int index);        // there goes type safety
    // ...
};

class gr_Graph {
    // ...
  public:
    // ...
    const gr_Node *lookupNode (const char *name) const;
        // lookup a node in the graph by name
    const gr_Node *lookupNode (const gr_NodeId& id) const;
        // lookup a node in the graph by id
};

void f(const char *names[], const gr_Graph& g)
{
    const gr_Node *node = g.lookupNode(*names[0]);  // <-- oops!!! didn't
                                                    // want * in *names[0]
    if (node) {
        cout << *node << endl; // all is well, print the node
    }
    else {
        cout << "Program Error: What happened to the node!!!" << endl;
        assert(0);    // node with this name should be there
    }
}
```

图 9-38 （误）用 gr_Graph 去通过名称查找 gr_Node

指导方针

考虑避免 cast 运算符，尤其是对基本整数类型，改为进行显式的转换。

一般来说，显式转换函数比隐式转换更易读，并且更安全。尽管 cast 构造函数是必需的部分，但是 cast 运算符是更容易避免的隐式转换的形式——我们**总是**可以提供一个显式转换函数来完成一个 cast 运算符所做的工作。

正如我们在图 9-6 中所看到，提供给 pub_string 一个 cast 构造函数和一个 cast 运算符（转换为一个 const char *或从 const char*隐式转换）会导致二义性，这种二义性需要进一步的工作来解决。如果我们用一个成员函数（如 const char*str()const）来代替 cast 运算符，并使用相同的实现，则不会产生二义性[①]。

在很有限、很好理解的情况下，隐式转换能够增加可用性[②]。自动将 pub_String 转换为 const char*，并且将一个任意的对象转换为一个 const void*（作为有效性测试），这似乎是隐式转换的适度安全的使用。有证据表明，客户代码如果被迫调用显示的成员，例如 str()和 isValid()，那么客户代码将更清晰。由于没有出现意外的潜在可能性，使得这些转换是合理的。传递一个 pub_String 给一个期望得到一个 const char*的函数几乎总是合理的，因为这两种类型的语义是紧密耦合的。隐式地转换为一个 const void*是安全的，因为你对结果能做的惟一有用的事是将它与 0 进行比较。

清除隐式转换的误用涉及将一个对象转换为一个不相关的或广泛使用的类型——尤其是一个基本类型。例如，使一个特定的对象（如 gr_NodeId）自身转换为代表其内部序号的一个整数是完全不适当的，并且事实上消除了将该值表示为一个类的任何类型安全的优点。一个知道如何将自身转化为一个表示其大小的 double 类数值的 geom_Point 也是不适当的，尤其是因为显式转换（例如，使用 pt.magnitude()）更安全，更可读，并且是一个总是可得到的替换接口。

9.3.2 编译器产生的值语义

C++语言要求编译器在必要的情况下自动产生基本成员函数的定义，除非它们已经在类中显式地声明了（见 6.2.6 节）。这一功能最通常的用途是产生的拷贝构造函数和赋值运算符。

指导方针

为任何定义在头文件中的类显式声明（公共地或私有地）构造函数和赋值运算符，甚至在默认的实现是充分的情况下。

① 见 **meyer**，Item26，89 页。

② 见 **stroustrup**，7.3.2 节，232~233 页。

当考虑是否显式地声明一个拷贝构造函数或赋值运算符时，首先要问的问题是，我们是否准备让这个对象支持值语义（见5.9节）。对于一些对象，如图5-81中的Gnode，值语义没有意义。而对于其他一些对象，例如迭代器，值语义就有意义，但对于一个充分的接口来说，值语义通常是不必要的（见8.2节）。在这两种情况下，我们应该通过将它们声明为private并故意让它们是未定义的来显式地禁止使用这种值语义函数[①]。

如果要支持值语义，那么下一个问题是，编译器产生的构造函数和/或赋值运算符是否能正确地工作[②]。如果默认定义是不正确的，那么我们需要自己声明并定义这些成员，否则，我们必须确定默认定义变得无效的可能性，以及我们的客户程序如果对我们的接口进行一次无绝缘的修改所需的这些开销。如果所预期的开销太高，那么我们将再次选择自己定义这些操作而不是使用默认的实现。

最后，对于每一个局部的对象，只要编译器产生的实现是有意义的，并且绝缘不是一个问题，那么我们可以允许这些函数定义是默认的。特别是，对于完全在.c文件中定义的类，允许默认拷贝和赋值语义通常是合适的。但是，依赖于默认语义的输出类定义的某些客户可能还有这种钻牛角尖的疑虑：默认的实现真的足够好吗？或者只是作者未能解决这个问题？

注意，C++的一些当前的实现不允许通过函数表示法来调用产生的operator=，也不允许通过其地址来调用，这是C++语言所要求的[③]。这种编译器的失败支持了应该总是显式地声明一个暴露的类的值语义运算符的观点。

9.3.3 析构函数

析构函数是一种具有许多独特职责的函数；我们会在本节讨论它们。

次要设计规则

在每一个声明了一个虚函数的（或派生于一个声明了虚函数的类的）类中，把析构函数显式地声明为类中的第一个虚函数，并且非内联地定义它。

析构函数负责析构对象，并释放当前由对象管理的资源（如动态内存）。当一个类把一个函数声明为virtual的时候，它同时也将自己宣称为一个基类——可能有其他别的原因将一个函数声明为virtual吗？即使当基类什么也没有的时候，派生类也可以获取资源。相反地，为了确保派生类析构函数能被调用，甚至从基类指针或引用中调用，基类析构函数就必须声明为virtual[④]。

① **meyers**，Item27，92~93页。

② **ellis**，12.8节，295页；以及**meyers**，Item 11，34~37页。

③ **ellis**，12.8节，296页。

④ **ellis**，12.4节，278页。

在一个动态绑定的有效实现中，我们希望在整个程序中任何虚拟表都只有单一的拷贝。问题是："对于一个给定的类，编译器将把虚拟表定义放置在哪个惟一的编译单元中？" CFRONT 使用的技巧是（其他许多 C++实现也是这样），将外部虚拟表放置在定义了出现在该类中的（如果存在的话）的词典顺序上第一个非内联虚函数的编译单元中。

析构函数显示的代价有时确实是惊人的。图 9-39 描述了一个真实的问题，该问题在大型项目中在相当长的时间内没有检测出来。问题是这样开始的：流行的 core_String 类派生于包含虚函数的 core_String，因此当然包括一个虚拟的析构函数。core_String 没有分配任何附加的资源，甚至根本没有声明一个析构函数。编译器要求为这个派生类产生一个析构函数并把它放入该派生类的虚拟表中。

```
class core_StringBase {
    // ...
  public:
    // ...
    virtual ~core_StringBase();
    // ...
    virtual int length();
    virtual operator const char *() const;
};

class core_String : public core_StringBase {
    int d_length;

 public:
   core_String(const char *cptr);
   core_String(const core_String& string);
   core_String& operator=(const core_String& string);
   int length() { /* ... */ }
};
```

图 9-39 未能非内联定义至少一个虚函数

关于在何处放置一个惟一的全局虚表拷贝没有给定任何线索，所以编译器在每一个包含 core_String 头的编译单元中都放置一个该表的拷贝。雪上加霜的是，也没有惟一的地方来产生析构函数的一个非内联版本，因此，在每一个编译单元中析构函数的一个静态拷贝与虚表放在一起。最后，每一个内联的虚函数（例如 length）也被拒绝为其非内联实现提供惟一的"家"。每一个内联虚函数的一个静态拷贝也被放置在包含 core_String 类的每个编译单元中。

当 Unix 的"nm"工具运行在该可执行程序上时，静态名称的一个直方图中出现了数千个具有相同名称的静态函数定义，但它们都被定义在独立的编译单元中。这时，这个问题才被发现。为 core_String 声明析构函数并且非内联实现它可以解决所有的问题。这个行为是隐秘的并且是依赖于实现的，但这是我们目前的现实。

按照我们在本书中一贯遵循的风格，创建函数应该放在任何其他非静态成员函数之前。因此，遇到的第一个虚拟成员函数总是析构函数。同时，为了使析构函数的地址能放入一个

虚函数表中，必须至少保证一个析构函数的版本是非内联定义的。至少有一个虚函数声明为非内联的需求，与在类中析构函数的自然词典位置相耦合，使得析构函数的自然选择是声明为虚拟的并且非内联定义。

在很少的情况下（例如，一个深度继承的层次结构），通过内联地定义一个空的析构函数，可以提高性能，在这种情况下，类中的某个其他的函数**必须**声明为虚拟的非内联函数，以避免产生冗余的表。

指导方针

在没有另外声明虚函数的类中，显式地把析构函数声明为非虚拟的并且适当地对它进行定义（内联的或非内联的）。

对于没有另外声明虚函数的类来说，实现一个虚拟析构函数不可能是适当的。在大多数实现中，使析构函数单独成为虚拟的，会增加指针的大小，从而增加每个实例的大小。对像geom_Point这样的小对象来说，会增加50%的开销。一种防止内存泄漏的解决方案是，一个派生于一个带有非虚拟析构函数的基类的派生类应该避免管理对象被析构时必须释放的额外资源[1]。

对于不需要虚函数的类，仍然有理由要求析构函数要显式地声明。显式地调用一个基本类型的析构函数是合法的C++语句[2]：

```
int i;
i.int::~int();      // legal C++; does nothing
```

试图显式地调用一个没有显式声明析构函数并且还没有为其产生析构函数的对象的析构函数在当前的很多编译器上都不能通过编译。因为不能提取一个析构函数的地址，所以，只有当一个基类或嵌入的成员对象有一个析构函数时，才会为一个没有显式声明析构函数的类产生一个析构函数[3]。这个事实对于基于模板的容器对象显式调用参数化类型的析构函数有启示作用（图 10-33b 提供了一种有用的工作区）。为了一致性，应该在适当的地方尽可能析构对象，不管它是否定义了析构函数。

9.4 小结

C++语言在描述函数级接口方面提供了极大的灵活性。当我们设计每个函数时，我们必须研究至少14个独立的问题。每一个决定都会引起必须在上下文中解决的额外考虑：

① **meyers**，Item14，42~48页。

② **ellis**，12.4节，280页。

③ **ellis**，12.4节，277页。

（1）运算符或非运算符函数？

运算符：

- 运算符表示法提高了可用性（尤其是可读性）。

非运算符函数：

- 用户自定义类型上的操作在语法上并不能映射为基本类型上的同一操作。

（2）自由运算符或成员运算符？

自由运算符：

- 我们要为其最左参数激活使隐式用户自定义转换。
- 它是语法上对称的（例如，==、<、+）。

成员运算符：

- 我们要禁止为其最左参数进行隐式用户自定义转换。
- 它修改一个实参（例如，=、+=、*=、++）。
- 语言要求成员关系（例如，()、[]、->）。

（3）虚函数或非虚函数？

虚函数：

- 其行为在一个派生类中必须能够被覆盖。

非虚函数：

- 它是一个对称的运算符函数（例如，!=、>=、|）。
- 它是一种能支持其参数进行用户自定义转换的单目运算符函数（例如，!、+、-）。
- 其行为能被实现为成员数据的值的变化。

（4）纯虚成员函数或非纯虚成员函数？

纯虚成员函数：

- 在派生类中没有覆盖其行为很可能是一个错误。
- 从物理上拆除接口与实现之间的耦合是重要的。
- 该类将被许多客户程序使用。
- 该类定义了一个协议。

非纯虚成员函数：

- 默认的行为是有意义的。
- 在许多情况下其默认行为是正确的。
- 该类不会被广泛地使用。

（5）静态的或非静态的成员函数？

静态的：

- 它只使用 struct 来限制函数名的作用域。
- 它实现了从一个更低层次的对象中提升的非基本行为。
- 在其所有的参数中需要有关用户自定义转换的对称性。

非静态的：

- 它依赖包含在该类的特定实例中的数据。
- 它是一个运算符函数。

（6）const 或非 const 成员函数？

const：

- 它并不修饰嵌入在 class 或 struct 中的位。
- 它只修饰那些客户绝不会通过程序访问的物理值。

非 const：

- 它修饰逻辑值，即使编译器说 const 是合法的。
- 它返回一个能破坏 const 正确性的类型的非 const 版本。

（7）公共的、保护的或私有的成员函数？

公共的：

- 它是预备供一般的公共程序直接使用的。

保护的：

- 它是预备只给派生类作者使用的。

私有的：

- 它是一个非虚拟的实现细节（特别是对于分解公共内联函数的实现来说）。
- 它是一个必须编程的虚拟函数，但它不需要通过派生类访问。

（8）通过值、引用或指针返回？

值：

- 没有预先已存在的对象可返回。
- 保护整体封装，这很重要。

引用：

- 总是有某些东西要返回。
- 我们正返回对一个多态对象的访问，其内存在其他地方管理。

指针：

- **可能**有某些东西要返回。
- 函数可能失败。

实参：

- 想在一个句柄中返回一个新分配的多态对象，这个句柄将管理这个对象，从而减少了内存泄漏的可能性。
- 在返回重量级对象比通过值返回更有效的同时，要保护整体封装。
- 要返回多于一个项目。
- 状态和值都必须返回。

（9）返回 const 或非 const?

const:

- 要返回一个指向该类的一个数据成员指针或引用。
- 返回非 const 会破坏 const 的正确性。
- 要返回一个指向一个共享的伪对象的引用（例如，一个稀疏数组的 0 记录）。

非 const:

- 要通过值返回。
- 要提供对一个被包含对象的直接的可写访问（例如，在一个数组中）。

（10）实参是可选的还是必须的?

可选的:

- 我们已经添加了一个新的实参到现存代码中。
- 有单一的算法。
- 我们希望我们的代码能自我文档化。
- 默认的实参是一个无效的值。

必须的:

- 这是广泛使用的接口。
- 默认值是一个用户自定义类型。
- 绝缘是重要的。

（11）通过值、引用或指针传递实参?

值:

- 实参是一个基本类型或枚举类型。

引用:

- 实参是一个没有被修饰的用户自定义类型。

指针:

- 实参被修饰了。
- 实参的地址被提取。
- 实参被删除了。
- 实参可以省略。

（12）将实参作为 const 或非 const 传递?

const:

- 实参是通过引用传递的用户自定义类型。
- 实参从未被修饰，但通过指针传递（因为一个空值有时是合适的）。

非 const:

- 实参（例如一个枚举类型或基本类型）是通过值传递的。

（13）友元或非友元函数？

友元：

- 封装的实行不（很少）是问题。

非友元：

- 有一个适合于实现它（例如，运算符==）的可得到的成员函数（例如，compare）或友元类（例如，StackIter）。

（14）内联或非内联函数？

内联：

- 内联函数**体**的大小会比非内联函数**调用**的大小更小。
- 性能显然是关键的，并且它是合理的小型函数或者只从一些地方调用它。

非内联：

- 它不是内联的。
- 只有当动态绑定时，它才是有用的。
- 绝缘是重要的。

在函数的接口中有许多可选择的整数类型可供使用：short、unsigned、long 等。在实际中，一个 32 位的机器上，在接口中我们需要的惟一整数类型是 int，使用任何其他类型都潜在地存在低效率、不能封装、易于出错或者只是使用起来很麻烦的可能性。

在C++中有三种可选择的浮点类型：float、double 以及 long double。在 C 中，在将浮点数作为参数传递之前，传统上将所有的浮点参数都转换为 double。大多数硬件都适合于尽可能有效地处理 double 值。所有的浮点数都应该在接口中被表达为 double，除非有强制的原因要求不这么做。

转换运算符（例如，单参数的构造函数和 cast 运算符）与编译时的类型安全是相互竞争的关系。接受单个参数的构造函数会激活隐式用户自定义转换。但是这样的结构是许多接口的必要部分。另一方面，cast 运算符是容易避免的，只要提供显式的转换函数即可。有时通过隐式转换可以增强可用性。但是，在大多数情况下，隐式转换是一种倾向，尤其是当它包含一个基本整数类型时。

C++编译器会自动产生一些未定义的函数（如果需要）。不依赖默认行为有很多原因，尤其是当接口在整个系统中被广泛使用时。许多 C++实现依赖至少有一个非内联定义的虚函数。在我们的风格中，这个虚函数总是析构函数。一些当前流行的编译器不允许显式地调用没有显式声明的析构函数。在实践中，显式地定义每个函数的析构函数是明智的。对于没有虚函数的类来说，内联地定义还是非内联地定义析构函数都是合适的。对于有虚函数的类来说，应该非内联地定义析构函数。对于协议类（见 6.4.1 节），析构函数应为空。

10

实现一个对象

凭借良好的（即，小的、封装的）接口，可以使对象实现选择的狭窄空间变得较为开阔。在这里，一个设计错误的代价远比高层设计时小，因为问题只局限在整个系统的一个极小部分中。但在系统集成过程中，单个的实现技术结合在一起时，也会以若干方式对项目的整体成功产生影响。

一个程序必须在一个拥有有限资源（如：内存）的环境中运行。那些在某一时刻拥有许多被激活实例的类，会增大其对象的规模。对象的各个数据成员的大小和次序也将影响其规模。定制的内存管理技术，有时能成倍地改善运行时性能，但它们也能导致系统在运行一定时间后占用远远超过其实际需要的内存。

在这最后一章中，我们将仔细探讨关于在 C++中实现类的组织细节的一些基本原则。我们还将提出关于实现单个成员函数的一些建议。在本章的其余部分，我们将研究有关定制内存管理的一些问题。

尽管内存管理是一个复杂的课题，大多数高性能的系统也必须对其加以重视。我们将给出几个详细的例子和实验，用以比较一些普通内存管理组织的相对优点。特别地，我们还考虑探讨特定类管理技术的性能优势，然后讨论当使用这些技术的类被集成到更大的系统时可能造成的潜在问题。接着，我们将介绍特定对象内存管理——一个更为可取的选择方案，它避免了许多特定类管理技术集成时出现的问题，而又基本达到了与基于类的技术相同的运行时性能。最后，我们将讨论在模板上下文中的内存管理，并提供详细的例子来说明如何实现真正通用的容器类。

10.1 数据成员

这一节，我们讨论关于在类中对数据成员进行选择和排序的逻辑和组织问题。

10.1.1 自然对齐

许多普通的基于RISC的微处理器，依赖于被自然对齐的基本类型的实例。被**自然**对齐意味着内置类型的实例（如int、double和char*）不能简单地存储于任意地址，而应排列在N字节地址界上，其中N是对象的大小。

> **定义**：如果一个基本类型实例的大小能整除其地址值，那么它是**自然对齐的**（**naturally aligned**）。

由于sizeof(char)等于1，一个char变量可以存储在内存的任意地址处。如果sizeof(short)等于2，它就不能存储在“奇数”地址；而是应该存储在“偶数”地址，有时被称为“半字界（half-word boundary）”（在32位机器上）。integer和指针，它们的大小通常和机器字的大小相关，需要存储在一个字界中（如32位机器上的4字节界）。double变量通常比一个机器字大，一般存储在两字界中[1]。

> **定义**：一个聚集类型的实例，如果其对齐要求最严格的子类型的排列能整除聚集的地址，那么它是**自然对齐的**。

一个给定类型的数组实例有着和该类型本身相同的对齐要求。满足一个用户自定义类型的自然对齐要求，意味着满足其要求最高的嵌入式子类型的对齐要求。图10-1给出了典型的32位机器上的一些自然对齐的例子。

假设 sizeof(char *) = sizeof(int) = 4 并且 sizeof(double) = 8

```
struct A {
    char d_c;
};
```

```
  sizeof(A): 1
alignment(A): 1
```

```
struct B {
    char d_c;
    char d_ac[3];
};
```

```
   sizeof(B): 4
alignment(B): 1
```

```
struct C {
    double d_d;
    char *d_pc_p;
    int d_i;
};
```

```
  sizeof(C): 16
alignment(C):  8
```

图10-1 各种用户自定义类型的大小和自然对齐

① 通常double被保存于奇数字界（与偶数字界相反）也不会造成太严重的后果。然而，在一些体系结构中，如果没有对double类型进行自然对齐，可能会造成性能的严重下降。

原　则

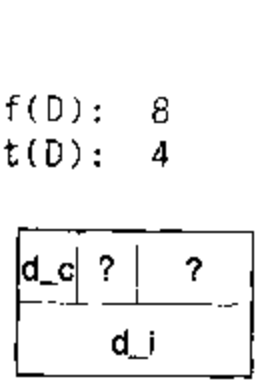
声明数据成员的次序能够影响对象的大小。

C++语言保证在没有干涉访问限定符（如，public、protected 和 private）的情况下，将根据它们在结构中的被声明次序为非常静态数据成员的内存分配递增的地址值；但是，它们不必是连续的[1]。一个结构内的对齐可能导致结构的中间部分及尾部脱节（但在头部绝对不会）。作为一项规则，当组织一个类或结构的布局时，人们可以假设是采用了自然对齐；但是不应该依赖它。

假设 sizeof (int) = 4 以及 sizeof (double) = 8

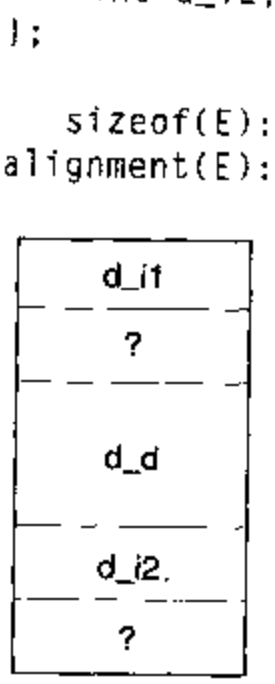
```
struct D {
    char d_c;
    int d_i;
};
```

```
sizeof(D):  8
alignment(D):  4
```




```
struct E {
    int d_i1;
    double d_d;
    int d_i2;
};
```

```
sizeof(E): 24
alignment(E):  8
```




```
struct F {
    int_d_i1;
    int d_i2;
    double d_d;
};
```

```
sizeof(F): 16
alignment(F):  8
```

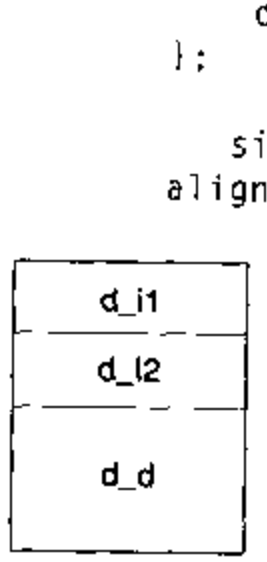


图 10-2　有孔的用户自定义类型

图 10-2 给出了三种用户自定义类型的大小、自然对齐和相应的对象布局。类型 D 因为其第二个数据成员必须存储于一个字界内所以中间有一个孔。类型 E 有两个孔：第一个孔是因为 double 型的 d_d 必须从一个双字（8 字节）界开始存储；尾部的第二个孔是为了确保 E 对象数组的每个元素也是对齐的：

① **ellis**，9.2 节，173 页。

```
假设: E a[N], b;  // N 's compile-time const with value > 0
那么: assert(sizeof a == N * sizeof b);
```

当某类型会同时有许多实例处于活动状态时，认真考虑其数据成员的声明次序（以减少对象大小）是很重要的。我们可以对图 10-2 中类型 E 的数据成员进行重新组织，以消除孔洞；结果就是类型 F，它小了 33%。

我们无论何时用重载全局运算符 new 适当地分配一个对象（使用布放语法），我们都必须确保这样做是在一个合适的对齐位置进行的。我们可以假设全局 new 返回满足最严格的可能边界的地址，但我们必须小心谨慎，以避免如下代码：

```
#include <new.h>  // declare placement syntax
// ...
char *buf = new char[sizeof(Stack[100])];
Stack *p = new(&buf[1]) Stack;              // bus error!
```

附录 B.2 的图 B-5 给出了在一个缓冲区中确保对齐正确的例子。

10.1.2 在实现中使用的基本类型

在 9.2 节中，我们论证了对在接口中使用的基本类型进行选择限制是一种明智的做法。在实现中使用特殊的基本类型会产生一些独特的问题。

指导方针

只有在已知这样做安全时，才能为了优化在实现中用 short 代替 int。

```
class win_Point {
    short d_x;
    short d_y;

  public:
    win_Point(int x, int y);
    // ...
};
```

图 10-3 在实现中使用 short

图 10-3 显示了在对象实现中合理使用 short 的例子，其中假设这个类的规范说明了 x 和 y 的值在任何时候都取在[0、…、1023]的范围内。在 sizeof(short)等于 2 的 32 位体系结构中，我们需要将整个对象存储在 32 位寄存器中。在空间和运行时性能两方面的潜在性能要求都可能证明在这种情况下使用 short 是合理的。（一个更为简单的移植方法，见 10.1.3 节）。除了低

级的位移操作[1]外，unsigned 类型的合理使用非常难以获得。

指导方针

即使在实现中也尽量不要考虑使用 unsigned。

```
class pub_Array {
    int *d_array_p;
    unsigned int d_size;

  public:
    pub_Array(unsigned int size);
    // ...
};
```

（a） 一个数组类

```
class core_String {
    char *d_string_p;
    unsigned short d_length;

  public:
    // ...
    int length() const;
    // ...
};
```

（b） 一个串类

图 10-4 在实现中对 unsigned 和 short 的不恰当使用

图 10-4 显示了对 unsigned 和 short 使用不当的两个类（假设在一个标准 32 位体系结构中）。在图 10-4（a）中，内部大小被设为 unsigned 类型，以容纳至多 2^{32} 个整数的数组，大概是为了避免溢出的可能。这个决策也已促使类的设计者在接口中暴露了 unsigned int 。即使忽略对接口的负面影响，在此例中使用 unsigned 的理由也是很不合理的。首先，new 运算符绝不可能去为 2^{32} 个相邻整数大小的对象寻找空间（至少在可预见的将来）。其次，除非一个指针变量大于一个 int，否则虚地址空间限制了整数的总数：$2^{32} \div \text{sizeof(int)} \leqslant 2^{30}$。换句话说，一个最大能容纳 $2^{31}-1$ 绝对值的（带符号的）int 类型变量太大了。

原 则

在实现中使用 unsigned 类型以“提高一点性能”，是基本的整数类型没有大到足够安全的标志。

在图 10-4（b）中，core_String 类定义其内部大小为 short，因为它认为其串长不会超过数千。这样，内部变量仅在此值超过 32767 时才被设为 unsigned 类型。除了 9.2.2 节讨论的在可维护性方面的损失之外，在所有（除了反常的）情况下，如果不知道 32767 是否够大，那么 65535 也是值得怀疑的。为了“以防万一”而将一个 short 型值设为 unsigned 型是在赌运气——通常使用 int 比冒险要好。在这种情况下错误使用 short 则更加荒谬，因为自然排列会造成孔洞，而这个孔洞本来可以保存一个 int 变量的另一半；在此使用一个 short 变量不能节省

① **ellis**，5.8 节，74 页。

任何空间[1]。

在优化任何局部代码的过程中，在对象中使用可选择的基本整型（如，short、char）以优化存储的决定最好推迟到该对象正在工作、已进行了功能测试、并已得到性能分析数据后再进行。一套完全的回归测试将有助于确保我们没有为优化而牺牲实现的正确性[2]。

10.1.3 在实现中使用 typedef

typedef 通常能帮助表达复杂函数的声明。在某些采用精确位数表示的基本类型的定义中，typedef 也是非常有用的。

有时我们确切地知道自己需要多少位。例如：当我们要持久地保存要跨越异构平台共享的信息（在磁盘）时，我们需要确保我们的基本数据类型有正合所需的、不多不少的精确性。图 10-5（a）显示了一个在系统范围内有效的头文件，它将类型的定义和绝对大小隔离开来。为了保证采用绝对大小的对象移植到一个新平台时被正确处理，我们只需改变这一文件。如图 10-5（b）显示了一个 geom_Point 类，该类的每一个坐标正好需要 32 位。一般而言，一个 int 类型的大小相当一个机器字。即使在 64 位体系结构的机器中，我们也只需要 32 位，以便和其他体系结构兼容——为什么要浪费空间呢？

```
// sys_type.h
#ifndef INCLUDED_SYS_TYPE
#define INCLUDED_SYS_TYPE

struct sys_Type {
    typedef signed char Int8;
    typedef short Int16;
    typedef int Int32;
    typedef unsigned char Uint8;
    typedef unsigned Uint16;
    typedef int Uint32;
    typedef float Float32;
    typedef double Float64;
};

#endif
```

（a）系统范围定义文件

```
// geom_point.h
#ifndef INCLUDED_GEOM_POINT
#define INCLUDED_GEOM_POINT

#ifndef INCLUDED_SYS_TYPE
#include "sys_type.h"
#endif


class geom_Point {
    sys_Type::Int32 d_x;
    sys_Type::Int32 d_y;

  public:
    Point (int x, int y);
    // ...
};

#endif
```

（b）固定大小 geom_Point 类

图 10-5 使用 typedef 来指定实现中的固定大小

① 关于在实现中不恰当地使用固定大小数组的进一步讨论见 **murray**，9.2.2 节，210~212 页。

② 也见 **murray**，9.2-9.10 节，234~235 页；以及 **cargill**，第 7 章，138 页。

倘若你认为只有函数是值得测试的，那么考虑图 10-6 所示的 sys_type 组件的测试驱动程序。在运行任何其他驱动程序之前首先试运行这个程序，可以确保依靠定长数据类型的组件没有“欺骗它们自己”。我们已经将我们的配置假设条件隔离地存放在一个单独的文件中。由于一般信息不是来自于任何高层可扩展的（可变的）组件集，而是产生于低层的编译器和机器的体系结构（固定的），因此编译时耦合在这里不是问题[1]。

```
// sys_type.t.c
#include "test_util.h"          // define TEST_ASSERT, etc.
#include "sys type.h"
main()
{
    TEST_BEGIN
    TEST_ASSERT(1 == sizeof(Int8));
    TEST_ASSERT(2 == sizeof(Int16));
    TEST_ASSERT(4 == sizeof(Int32));
    TEST_ASSERT(1 == sizeof(Uint8));
    TEST_ASSERT(2 == sizeof(Uint16));
    TEST_ASSERT(4 == sizeof(Uint32));
    TEST_ASSERT(4 == sizeof(Float32));
    TEST_ASSERT(8 == sizeof(Float64));
    TEST_END
}
```

图 10-6 测试一个 typedef

10.2 函数定义

一旦进入了实现函数的阶段，则我们的大部分决定都是局部化的决定。因此做出一个不良决定的代价会比较小，因为改变它通常不会影响大量代码。即使是这样，在编写函数体时仍有一些一般的准则要牢记在心。

10.2.1 自我断言

将函数行为无定义的条件建成文档是开发接口的一个重要部分。在接口中进行注释是消极的，并且不会在运行时发出程序出错的警告。如 1.3 节所讨论的那样，我们可以结合使用注释和 assert 语句来得到轻量级的可维护代码。

在较长的函数中，有时存在几种能引向相同语句的途径，通常这种语句设定了内部假定条件。图 10-7 展示了一种情况，在其中可以不进入 if 语句，也可以不进入 while 语句。在任

① 6.2.9 节研究枚举和编译时耦合。

何一种情况下，if 语句之后规定的条件必须保持为真，并被一个 assert 语句支持。

```
// ...

if (!q) {
    while (p && 0 != strcmp(name, p->name())) {
        p = p->next();
    }
}

// Either q is true, or p now points to the first element
// with the specified name if one exists.
assert (q || !p || 0 == strcmp(name, p->name()));

// ...
```

图 10-7 注释/断言一个内部变量

这些内部自检查不光能在运行时检测错误。清楚地标识一个假设的习惯能使思路清晰，使函数的逻辑流程更易于被他人理解[1]。

在监测关键功能的系统里，即使在产品代码中我们也可能不太愿意删去 assert 语句；但是，当一个编程错误导致一个对象进入不一致状态时迫使整个程序中止，这显然也是不能接受的。一个 assert 语句的自然扩展是抛出一个派生于的基本类的异常，如 sys_ProgrammingError。在这种情况下，容错系统可以被设计成尽可能捕获这样的错误并从中恢复，而不是自动退出。若错误未能被捕获，会最终造成系统中止。要注意，使用异常时我们实际上将处理错误的责任“升级”到了一个更高层的组件上（见 5.2 节）。

10.2.2 避免特殊情况

获得代码覆盖率是一个衡量测试有效性的一般标准。但函数中的路径越多，就越难保证函数在所有情况下都可靠。

原 则

自然地包括其边界条件的算法，通常比将边界条件作为特殊情况处理的算法更简单、更短小、更易于理解和测试。

例如，当遍历一个需要修改的链表时，开发者有时会选择使用一对指针。这种方法需要将空链表作为一种特别情况对待（或总是保持一个伪首连接）。我们不使用指向当前连接的指针作为状态变量，而考虑改为维护当前连接的地址，如在图 5-83 中显示的对 PtrBagManip 类

[1] 也见 **murray**，9.2.1 节，208~210 页。

所做的那样。

```
class PtrBagManip {
    PtrBag *d_bag_p;
    PtrBagLink *d_Link_p;
    PtrBagLink *d_PrevLink_p;

  public:
    // ...
    void remove();
    // ...
};

void PtrBagManip::remove()
{
    PtrBagLink *tmp = d_Link_p;
    d_link_p = d_link_p->next();
    if (d_prevLink_p) {
        d_prevLink_p->nextRef() = d_link_p;
    }
    else {
        d_bag_p->d_root_p = d_link_p;
    }
    delete tmp;
}
```

```
class PtrBagManip {
    PtrBagLink **d_addrLink_p;

  public:
    // ...
    void remove();
    // ...
};

void PtrBagManip::remove()
{
    PtrBagLink *tmp = *d_addrLink_p;
    *d_addrLink_p = (*d_addrLink_p)->next();
    delete tmp;
}
```

（a）将边界条件当作一个特殊情况看待　　（b）将边界条件作为主要算法的一部分看待

图 10-8　避免特殊情况的代码

图 10-8（a）显示了一个实现，它同时维护一个指向当前连接的指针和一个指向过去连接的指针；在当前连接被删除时，d_prevLink_p 指针将被用于更新过去连接的 d_next_p 字段。如果当前连接正好是链表的首连接，d_prevLink_p 将为 0，我们就需要更新链表的根节点，因此我们保留了一个可写的指向 PtrBag 自身的指针。

图 10-8（a）的实现很直截但并不精练，它被不必要的状态和算法的复杂性搞得很费解。图 10-8（b）中的实现完成了同样的功能。删除此实现中的第一个元素和删除任何其他元素的方法一样。

实现（b）只需要一个状态变量，删除一个节点的复杂度大大降低了。

原　则

通过增加一个额外的间接层能解决很多问题。

维护一个指针的**地址**而不是维护指针本身的技术，是处理各种链表型结构的简洁但有力的惯用法：

```
struct Link { Link *d_next_p; Link(Link *next) : d_next_p(next) {} // ....

class List { Link *d_head_p; void modify(); // ...

static int q(const Link& x);      // 1 if some condition on x holds; else 0

void List::modify()               // some function that modifies the list
{
    Link **ppl = &d_head_p;       // starting at the beginning
    while (*ppl && !q(*ppl)) {    // while not at end and item not found
        ppl = &(*ppl)->d_next_p; // advance the (address of the) pointer
    }
    assert(!*ppl || q(*ppl));     // ppl points to last or desired Link *

    // ...                        // insert before, remove, etc. using *ppl
```

额外的间接层允许我们向一个已排序的链表插入一个元素，而不需要维护两个指针或将空链表作为特殊情况：

```
*ppl = new Link(*ppl);            // inserting before an item is easy
```

从一个链表删除一个特定的连接也很容易：

```
if (*ppl) {                       // if found we can unlink and delete item
    Link *item = *ppl;
    *ppl = (*ppl)->d_next_p;
    delete item;
}
```

这个惯用法（或它的 for 循环的等价物）被用于实现私有功能拷贝和图 6-19（b）所示的链表类的结束，并被扩展用于实现下一节的一个基于哈希的符号表（图 10-11）。

10.2.3 用分解代替副本

在 5.6 节，我们讨论了小函数的重用会导致物理耦合，这与分解实现所取得的利益不相当。然而，在单个的设计优良的组件中，几乎没有理由要复制代码。构造、析构和赋值将经常共享一个通用的算法。如图 6-19 所示的 List 类，定义小的更基本的函数集以便从这个基本的公共功能中分解出共性是有用的，如图 10-9 所示。

默认构造函数：		init();	
拷贝构造函数：		init();	copy();
赋值运算符：	clean();	init();	copy();
析构函数：	clean();		

图 10-9 分解出共同的构造函数/赋值功能

注意，有趣的是，赋值运算符不是完全基本的；它可以根据基本的析构函数和拷贝构造函数来实现。

```
#include "new.h"                    // declare placement syntax

T& T::operator=(const T& that)
{
    if (this != &that) {            // check for x = x
        T::~T();                    // destroy object in place
        new(this) T(that);          // construct object in place
    }
    return *this;                   // return reference to this object
}
```

这个观察对基于模板的容器类有启示作用（见 10.4.2 节）。

在一个基本符号表的组件实现中可以找到另一个分解的例子。一个符号表是一个相关联的数组规范，该数组支持从关键字（最可能是一个字符串）到一些值（在这种情况下，是用户自定义类型 my_Value）的映射。

图 10-10 显示了一个符号表的简单实现的头文件。这个实现使用封闭的哈希表，因而使用动态分配的 my_SymTabLink 指针（d_table_p）数组来实现，指针的大小得自 maxEntriesHint (d_size)。该实现一共提供了四个基本操作：未找到则增加、不论找到与否都设置、如找到则删除并报告、查找。如图 10-11（a）所示，每项这样的操作都能被单独实现。然而，每项这样的功能基本上都需要定位指向符号的指针（实现为 my_SymTabLink）。注意，只有 remove 方法需要指针的地址；add 和 set 总是能为一个给定的哈希槽在链表的头部增加一个新符号，而 lookup 绝对不会增加一个符号。

原　则

在一个组件中分解出一般可重用的功能，可以减小代码长度并提高可靠性，而不会损失太多的运行时性能。

```
// my_symtab.h
#ifndef INCLUDED_MY_SYMTAB
#define INCLUDED_MY_SYMTAB

class my_Value;
class my_SymTabLink;
class my_SymTabIter;

class my_SymTab {
    class my_SymTabLink **d_table_p;        // closed hash table
    int d_size;                             // size of hash table
    friend my_SymTabIter;
```

```
private:
  my_SymTab(const my_SymTab&);                    // not implemented
  my_SymTab& operator=(const my_SymTab&);         // not implemented

public:
  // CREATORS
  my_SymTab(int maxSymbolsHint = 0);              // see Section 10.3.1
      // Optionally specify approx. number of entries (default ~500).
  ~my_SymTab();

  // MANIPULATORS
  my_Value *add(const char* name);
      // Adds a symbol to the table only if name is not already present.
      // Returns a pointer to the internal value if added, and 0 otherwise.

  my_Value& set(const char* name);
      // Adds a symbol to the table if not already present.  Returns a
      // reference to the internal value of a symbol with specified name.

  int remove(const char *name);
      // Removes a symbol from the table.  Returns 0 if the symbol with
      // the specified name was found, and non-zero otherwise.

  // ACCESSORS
  my_Value *lookup(const char *name) const;
      // Returns a pointer to an existing symbol's value, or 0 if
      // a symbol with the specified name cannot be found.
};

my_SymTabIter { /* ... */ };

#endif
```

图 10-10　基本符号表抽象的部分头文件

```
// my_symtab.c
# include "my_symtab.h"

static int hash(const char *name) { /* ... */ }




my_Value *my_SymTab::add(const char *name)
{
    int index = hash(name) % d_size;
    my_SymTabLink *&slot = d_table_p[index];
    my_SymTabLink *p = slot;
```

```
        while (p && 0 != strcmp(p->name(), name)) {
            p = p->next();
        }
        if (!p) {
            slot = new my_SymTabLink(name, slot);
            return &slot->value();
        }
        else {
            return 0;
        }
    }


my_Value& my_SymTab::set(const char *name)
{
    int index = hash(name) % d_size;
    my_SymTabLink *&slot = d_table_p[index];
    my_SymTabLink *p = slot;

    while (p && 0 != strcmp(p->name(), name)) {
        p = p->next();
    }
    if (!p) {
        p = slot = new my_SymTabLink(name, slot);
    }
    return p->value();
}

// my_symtab.c
# include "my_symtab.h"

static int hash(const char *name) { /* ... */ }

static my_SymTabLink *&
locate(my_SymTabLink **table, int size, const char *name)
{
    int index = hash(name) % size;
    my_SymTabLink **pp = &table[index];
    while (*pp && 0 != strcmp((*pp)->name(), name)) {
        pp = &(*pp)->next();
    }
    return *pp;
}


  my_Value *my_SymTab::add(const char *name)
  {
      my_SymTabLink *& p = locate(d_table_p, d_size, name);
      if (p) {
          return 0;
      }
```

```
    p = new my_SymTabLink(name, p);
    return &p->value();
}
```

```
my_Value& my_SymTab::set(const char *name)
{
    my_SymTabLink *& p = locate(d_table_p, d_size, name);
    if (!p) {
        p = new my_SymTabLink(name, p);
    }
    return p->value();
}
```

```
int my_SymTab::remove(const char *name)
{
    enum { FOUND = 0, NOT_FOUND };

    int index = hash(name) % d_size;
    my_SymTabLink **pp = &d_table_p[index];
    while (*pp && 0 != strcmp((*pp)->name(), name)) {
        pp = &(*pp)->next();
    }
    if (*pp) {
        my_SymTabLink *q = *pp;
        *pp = q->next();
        delete q;
        return FOUND;
    }
    else {
        return NOT_FOUND;
    }
}
```

```
my_Value *my_SymTab::lookup(const char *name) const
{
    int index = hash(name) % d_size;
    my_SymTabLink *p = d_table_p[index];

    while (p && 0 != strcmp(p->name(), name)) {
```

```
            p = p->next();
        }
        return p ? &p->value() : 0;
    }
```

(a) 特定功能的重新实现

```
int my_SymTab::remove(const char *name)
{
    enum { FOUND = 0, NOT_FOUND };
    my_SymTabLink *& p = locate(d_table_p, d_size, name);
    if (!p) {
        return NOT_FOUND;
    }
    my_SymTabLink *q = p;
    p = q->next();
    delete q;
    return FOUND;
}
```

```
my_Value *my_SymTab::lookup(const char *name) const
{
    my_SymTabLink *& p = locate(d_table_p, d_size, name);
    return p ? &p->value() : 0;
}
```

(b) 更一般功能的重用

图 10-11 实现一个符号表类

如图 10-11（b）所示，我们能够更简洁地实现相同的功能。尽管不算太好，但实现更短小了且性能相当。如果经性能分析之后，我们发现 lookup 耗费了大量的时间，那么我们可能有理由选择独立于更通用的 locate 函数重新实现这个函数（但不要在性能分析之前这样做）[1]。

有些情况下，**符号表**这个术语不太恰当。尽管这个抽象在某种意义上是符号的集合，但在该组件内的一个符号概念是隐含的；在这个符号表组件的逻辑接口上没有暴露“符号”类型。如 10.3.5 节所讨论的那样，一个有效的符号表对实现来说并非完全不重要。

① 见 **cargill**，第 7 章，138 页。

10.2.4 不要过于聪明

在大学毕业后的第一份工作中，我被告知有一个传奇的程序员，他对 FORTRAN 的精通在同行中无人能及。这个爱动脑筋的人关注到语言的一些最难理解的结构没有得到应有的使用。因此他研究这些结构中的每一个是否在产品软件中被充分地展示，即使这对他自己和别人都意味着要做更多工作。我希望我能遇上那个人就好了，唉，就在我加入公司之前他被赶走了。

指导方针

在设计一个函数、组件、包或完整的系统时，使用最简单的有效技术。

任何看得懂这本书的人显然不是编程新手。实际上，你可能对 C++语言内外都了解。要想成为有经验的 C++软件工程师，就必须对该语言有很好的了解——这一点是毫无疑问的。但对语言规则的了解仅仅是设计大型系统所需全部知识的一小部分。掌握带有注释的 C++参考手册（ARM）将使你了解每个 C++结构起什么作用，但不会使你知道它们在日常编程中的应用频率。使用最普通的方法来有效解决一个问题，将使软件更易于理解和维护。正如医学院的人常说的："当你见到蹄印，应该联想到马而不是斑马。"

10.3 内存管理

高性能设计的一个重要方面是内存管理。在一定情况下，通过在更严格的上下文中进行内存分配，能获得真实的性能收益。同时，内存是全局资源；个别类不恰当地使用内存管理（如过分使用内联函数），将对集成系统的性能产生负面的影响。

C++在相当程度上涉及了内存管理的语法和语言问题；幸好已有书籍很好地包含了这些细节[1]。在这一节中，我们提出了一些关于定制内存管理设计的问题。我们明确地区分逻辑和物理状态值。我们分析了某些特定物理参数对性能的影响，给出了一个特定类内存管理器的几个变量；讨论了它们的相对性能和对系统的影响；另外，还研究了可测试性的问题。我们认为，在大多数情况下使用特定对象而不是特定类来处理内存管理是更可取的。

10.3.1 逻辑与物理状态值

在一个对象直接管理动态内存的很多实现中，存在两种截然不同的、定义对象"状态"的数据成员：逻辑的和物理的。

① 例如，见 **meyers**，Item 5- Item 10，18~33 页。

> **定 义**：如果一个和某对象状态相关的值有助于预期的语义（即 ADT 的基本行为），那么它是逻辑值；否则就是物理值。

原 则

对一个完全封装的接口来说，每个可编程访问的值都是一个逻辑值。

逻辑值是根据语义而不是通过程序可访问能力来定义的，这是因为封装经常被以效率的名义破坏。仅仅因为一个对象在实践中未能封装一个特定的值，并不能判定它有助于基本的行为。虽然一个组件开发者的意图没有通过组件接口能访问的值那样易于客观度量，但是这个基本行为倾向于不像依赖程序可访问值的集合那样依赖实现选择。

指导方针

避免允许通过程序对物理值进行访问。

考虑如图 10-12 的 pub_String 类实现。这个串的状态包括一个受管字符数组的地址及其内容、该数组的物理大小和包含在数组中的串的逻辑长度。串长有助于一个串抽象的语义，因此是一个逻辑值（不论它是被保存还是需要计算）。与此相比，用来保存该串的内部数组的大小不必正好大于该长度；它的精确值对于对象的物理状态很重要，但对其逻辑状态则不然。

封装的目标是隐藏所有的物理状态，同时使逻辑状态可通过接口随时得到。对于该串类，动态数组的内容从位置 0 到串的长度都是其逻辑状态的一部分，数组的剩余部分和数组本身的地址是其物理状态的一部分。

```
class pub_String {
    char *d_str_p;              // dynamic array of characters
    int d_size;                 // physical size of array
    int d_length;               // logical length of string

  public:
    pub_String(const char *str, int maxLengthHint = 0);
    // ...
    pub_String& operator+=(const pub_String& str);
    // ...
    int length() const { return d_length; }
    operator const char *() const { return d_str_p; }
};
```

图 10-12　一个串类的一个可能的实现

根据 8.3 节的讨论，这个串类没有被完全封装，因为它的 cast 运算符暴露了一个物理值（即数组的地址）。作为一个规则，我们希望避免通过对象接口对物理值进行编程访问。效率，

甚至实用性有时会与这个目标相抵触。然而，什么会和什么不会被认为是物理值，这取决于抽象的程度（见5.10节）。

有时出于（已察觉的）性能原因，我们不得不让客户协助我们的类去完成其工作。显然在接口中暴露通用串内部数组的大小是不合适的。但是，我们可以为那些事先知道串会变得多大的客户提供一种机制，以给对象一个“提示（hint）”。

例如，在图10-12中，pub_String类的构造函数的第二个参数可选的能够为实现提供一个提示。注意，该提示没有限制这个pub_String对象需要容纳的串的长度；客户总是有权利超过该提示所指的值的大小而不会出错。如果考虑提示，串数组的物理大小在构造时将被设为maxLengthHint+1。现在，运算符+=的后续使用可能不需要重新设定串的大小，从而潜在地提高了运行时性能。

原　则

提示是只写的（write-only）。

在成员函数中作为参数传递的提示仍然只不过是个提示而已（正如register或inline）。如何处理提示是对象自己的事。对象绝对有忽略提示的自由，并且应该没有程序性的方法可以确定已提供提示的效果。相反，一个不准确的提示值可能严重降低对象的性能，但是它应该不会影响任何逻辑行为。

原　则

最好的**提示**不直接被绑定到特定的实现。

最好的提示和抽象本身相关，而不与特定实现相关。注意，提示被称为maxLengthHint而不是sizeHint。

一个参数是否应被作为提示对待，取决于抽象的程度。对于一个哈希表抽象，我们大概可以明确槽的数目，并提供对该值的访问方法。对于一个基于哈希表的符号表抽象（图10-10）实现，我们可以改为提供maxSymbolsHint提示。

指导方针

调用一个const成员函数的结果不应该改变对象中的**任何**可编程访问的值。

逻辑值和逻辑常量（见9.1.6节）紧密相关。如果一个成员函数被声明为const，那么它有责任不去修改任何逻辑值。基于相同原因，一个const成员应该避免改变任何恰好可以编程访问的物理值（例如，串类中数组的地址）。

```
pub_String str("foo", 1000);
// ...
const char *pcc = str;
```

```
if (str.length() < 10) {                    // const member length() resizes array
    cout << pcc << endl;                    // ?? behavior is undefined!
}
```

例如，假设在一些实现中，一个对 length()的调用导致动态数组重分配（如定位到一个“更合适”的大小）。那么调用这个 const 成员函数的结果将使 const char* 指针无效，并且导致后续的输出操作也是未定义的。

原　则

如果一个支持值语义的类型有两个实例，两个实例各自所有的逻辑值都相等，那么这两个实例是相等的（==）；只要有任何一个单个的逻辑值不相等，这两个实例就是不相等的（!=）。

逻辑值也直接和相等的概念相关。对于任何支持值语义的对象（见 5.9 节），截然不同的对象可以定义为拥有相同的值。也就是说，相等的定义是指逻辑相等而不是物理相等。在 pub_String 对象的例子中，如果两个串有相同的长度 n，并且它们各自的内部数组的头 n 个相应的字符的内容相等，那么这两个串是相等的。因为 pub_String 暴露了它的一些物理状态，我们能够在逻辑上相等的 pub_String 对象中找出不同点：

```
pub_String s1("foo");
pub_String s2("foo");

const char *p1 = s1;
const char *p2 = s2;
const char *p3 = s1;

s1 == s2                    // yes
p1 == p2                    // no
p1 == p3                    // yes
```

10.3.2 物理参数

物理值的设计空间很大。在这一节，我们来研究简单的、基于数组的整数栈的组织设计空间，该整数栈的头文件如图 10-13 所示。

```
// my_stack.h
#ifndef INCLUDED_MY_STACK
#define INCLUDED_MY_STACK

class my_Stack {
    int *d_stack_p;                         // dynamic array
    int d_size;                             // physical size
    int d_sp;                               // logical depth
```

```
  private:
    void growArray();                          // increase physical size
    my_Stack(const my_Stack&);                 // not implemented
    my_Stack& operator=(const my_Stack&);      // not implemented

  public:
    my_Stack(int maxDepthHint);                // hint: probable maximum depth
    ~my_Stack();
    void push(int value);
    int pop() { return d_stack_p[--d_sp]; }
    int top() { return d_stack_p[d_sp-1]; }
    int isEmpty() const { return d_sp <= 0; }
};

inline
void my_Stack::push(int value)
{
    if (d_sp >= d_size) {
        growArray();                           // note use of private member function
    }
    d_stack_p[d_sp++] = value;
}

#endif
```

图 10-13 一个基于数组的栈组件的头文件

首先，让我们考虑期望的最大栈深提示。如前所述，一个提示仅是一个建议；栈实现可以选择完全忽略它。另一方面，提示可能不对；实现一定不能因为超过提示的最大深度而出错。但是，如果栈被给定了一个有关在任何时候被要求能容纳的元素数目的严格上界，实现将能够利用这个信息去改善性能。

图 10-14 举例说明了影响动态数组管理的 my_Stack 函数的实现。在文件头部，两个枚举的物理参数值，INITIAL_SIZE 和 GROW_FACTOR，刻画了这个组件中的性能权衡。在没有提示的情况下，数组的初始大小将被置为 INITIAL_SIZE。无论何时栈满了，都将分配一个新的、大小为过去数倍的数组；这个倍数值由 GROW_FACTOR 指定。

预先知道数组的大小可以使实现避免重分配。改进的程度不一定有它初次出现的那样大。通过用 GROW_FACTOR 方法改变数组大小，处理栈深为 N 的栈，只要进行 d_stack_p 数组的 $O(\log(N))$次重分配。而且，为达到这个大小所需要进行的整数拷贝数量仅为 $O(N)$，而不是人们可能简单地认为的 $O(N\log(N))$[①]。

① 见 **murray**，8.3.2 节，173~174 页。

```
// my_stack.c
#include "my_stack.h"
#include <memory.h>  // memcpy()
#include <assert.h>

enum { INITIAL_SIZE = 1, GROW_FACTOR = 2 };

my_Stack::my_Stack(int size)
: d_size(size > INITIAL_SIZE ? size : INITIAL_SIZE)
, d_sp(0)
{
    d_stack_p = new int[d_size];
    assert(d_stack_p);
}

my_Stack::~my_Stack()
{
    delete [] d_stack_p;
}

void my_Stack::growArray()
{
    int *p = d_stack_p;
    d_size *= GROW_FACTOR;
    d_stack_p = new int[d_size];
    assert(d_stack_p);
    memcpy(d_stack_p, p, d_sp * sizeof *d_stack_p);
    delete [] p;
}
```

图 10-14　一个基于数组的栈组件的实现文件

由于 GROW_FACTOR 的指数特性，它的精确值对性能的影响程度不如对空间的影响大。值为 2 的 GROW_FACTOR 意味着将按需要为数组重新分配最多两倍的空间。如果我们将 GROW_FACTOR 的值改为 4，则意味着数组的 75%将不会被使用。通过修改图 10-15 所示的栈实现，用一个线性的 GROW_SIZE 代替指数性的 GROW_FACTOR，是一个有更高的潜在存储效率但健壮性却差得多的方法。

```
// my_stack.c
#include "my_stack.h"
#include <memory.h>  // memcpy()
#include <assert.h>

enum { INITIAL_SIZE = 0, GROW_SIZE = 100 };          // modified

my_Stack::my_Stack(int size)
: d_size(size > INITIAL_SIZE ? size : INITIAL_SIZE)
```

```
, d_sp(0)
{
    d_stack_p = d_size > 0 ? new int[d_size] : 0;     // modified
    assert(d_stack_p || d_size <= 0);                 // modified
}

my_Stack::~my_Stack()
{
    delete [] d_stack_p;
}

void my_Stack::growArray()
{
    int *p = d_stack_p;
    d_stack_p = new int[d_size += GROW_SIZE];         // modified
    assert(d_stack_p);
    memcpy(d_stack_p, p, d_sp * sizeof *d_stack_p);
    delete [] p;
}
```

图 10-15 使用一个固定的 GROW_SIZE 代替一个自适应的 GROW_FACTOR

为了说明这些物理参数在模拟操作下的相对效果，我开发了如图 10-16 所示的性能测试驱动程序。使用模式被有意设计为使栈快速增长。在循环的每次迭代中，循环次数的当前值被三次压入栈中，然后栈被一次弹出。注意栈深在每次迭代中加 2。

改变个别物理参数值，运行图 10-16 所示测试驱动程序的结果如图 10-17 所示。第一行代表对八个物理配置中的每一个进行 4000 次循环迭代。每个后续的行分别代表的迭代次数增长一倍，直至 8192000 次迭代。最前面的两列分别代表使用固定 GROW_ SIZE 的两种可相互替代的方法。其中的非自适应方法将不可避免因为大得出乎意料的栈而展现二次运行时特性；对 GROW_ SIZE 的较大值来说，它将极大地浪费小堆栈的内存空间。

当你预先知道这个值时，对正确的最大深度进行提示显然是一种胜利。在缺少信息的情况下，任何 INITIAL_SIZE 的特定值要么浪费内存，要么因太小而不起作用。由于数组可以适应几何大小的变化，所以运行时性能不是 GROW_FACTOR 实际值的敏感函数；但是，大于 2 的增长因子将产生过大的空间需求。注意表的最后一行，一个大小为 4 或 8 的增长因子，实际要快于推测一个刚刚越过最终值一半的大小然后不得不将那一半拷贝到一个新数组的方法（推测 16384002 的正确值产生最优结果 7.0）[①]。

```
// my_stack.t.c
#include "my_stack.h"
#include <stdlib.h>    // atoi()
```

① 在这个初始化大小中的额外的 2 是要允许队列在相同大小的一个循环的最后两个迭代上 push 三次并 pop 一次，同时不会迫使一个重分配动作发生。

```
#include <iostream.h>

main(int argc, const char *argv[])
{
    register int repeat = argc > 1 ? atoi(argv[1]) : 0;
    cout << "repeat = " << repeat << '\t';
    my_Stack s;

    for (register int i = 0; i < repeat; ++i) {
        s.push(i);
        s.push(i);
        s.push(i);
        int value = s.pop();
    }
}
```

图 10-16　my_stack 的微不足道的性能“加强”测试驱动程序

(GROW_SIZE):	100	10000						
INITIAL_SIZE:	0	0	1	128002	1024002	81920022	1	1
GROW_FACTOR:			2	2	2	2	4	8
迭代	I	II	III	IV	V	VI	VII	VIII
4000	0.0							
8000	0.4							
16000	1.8							
32000	9.5	0.0						
64000	51.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0
128000	254.6	0.6	0.1	0.1	0.1	0.1	0.1	0.1
256000	1074.5	2.7	0.3	0.2	0.2	0.1	0.3	0.2
512000		11.0	0.7	0.6	0.4	0.4	0.4	0.4
1024000		42.9	1.4	1.2	1.0	0.8	1.1	0.9
2048000		170.8	2.7	2.4	2.3	1.6	2.0	2.2
4096000			5.4	5.1	5.2	3.3	4.6	3.9
8192000			10.7	10.7	10.5	9.0	8.3	7.2

在 SUN SPARC 20 工作站上运行（优化）的 CPU 秒数

图 10-17　各种物理参数设置的相对性能

INITIAL_SIZE 和 GROW_FACTOR 方法非常通用，被应用在多种需要根据要求动态扩展的、数组型的对象上。在最初的开发中，选择 INITIAL_SIZE 为 1 并且 GROW_FACTOR 为 2 是保证完全试运行新代码的好办法。以后经过性能分析，如果需要可调整这些数字来提高边

际性能。

注意，这里提供的使用模式偏好 GROW_FACTOR 方法。对于经过较长时间缓慢增长才达到稳定运行时大小的对象，GROW_SIZE 有时被证明是更好的。GROW_SIZE 的主要缺点是，我们不能对剩余内存的使用设置上限，并将它设为所需实际内存的大范围百分比（像我们能对 GROW_FACTOR 所做那样）。如果以 50%未用内存作为上限太高，我们总是可以求助于带有小数的 GROW_FACTOR。例如，值为 1.1 的 GROW_FACTOR 可以将潜在的过度内存使用限制在 10%，而保留其适应几何变化的特性。惟一需注意的是必须结合使用 INITIAL_SIZE 和 GROW_FACTOR，确保首次调整使当前值最少加 1。除非在任何给定的时候都有大量激活的 my_Stack 对象，否则不必对过度内存使用如此关注。

10.3.3 内存分配器

内存分配器（memory allocators）是一个有丰富理论的课题，但远远超出了本书范围。在本节，我们只限于讨论我们在后面章节中要用到的两个非常基本又很有用的分配器。

```
// pub_blocklist.h
#ifndef INCLUDED_PUB_BLOCKLIST
#define INCLUDED_PUB_BLOCKLIST

class pub_BlockLink;

class pub_BlockList {
    pub_BlockLink *d_blockList_p; // linked list of allocated blocks
    pub_BlockList(const pub_BlockList&);             // not implemented
    pub_BlockList& operator=(const pub_BlockList&); // not implemented

  public:
    // CREATORS
    pub_BlockList();
        // Create an empty list of allocated blocks of memory.

    ~pub_BlockList();
        // Destroy this object and all associated blocks of memory.

    // MANIPULATORS
    void *allocate(int bytes);
        // Allocate block of memory of specified number of bytes.

    void release();
        // Free all blocks of memory allocated through this object.
};

#endif
```

图 10-18 简单的块分配器组件的接口

如图 10-18 所示的 pub_BlockList 分配器提供了一个便利的机制，可用来追踪对象的 allocate 函数申请的全部全局内存。本质上，分配器拥有并管理着内存，这样我们不必单独对其追踪。一旦我们知道所有内存不再被使用，我们可以调用 release 成员函数进行释放。当 pub_BlockList 对象被析构时，它自动释放所有由其分配的内存。为了完整性，图 10-19 提供了 pub_BlockList 块内存分配器的一个很小的实现。

```
// pub_blocklist.c
#include "pub_blocklist.h"
#include <assert.h>

// (LOCAL) AUXILIARY CLASS
class pub_BlockLink {
    void *d_block_p;            // block of storage
    pub_BlockLink *d_next_p;    // pointer to next link (or null)
    pub_BlockLink(const pub_BlockLink&);
    pub_BlockLink& operator=(const pub_BlockLink&);

  public:
    pub_BlockLink(void *block, pub_BlockLink *next) :
                                d_block_p(block), d_next_p(next) {}
    ~pub_BlockLink() { delete [] d_block_p; }
    pub_BlockLink *next() const { return d_next_p; }
};

pub_BlockList::pub_BlockList() : d_blockList_p(0) {}

pub_BlockList::~pub_BlockList() { release(); }

void pub_BlockList::release()
{
    while (d_blockList_p) {
        pub_BlockLink *q = d_blockList_p;
        d_blockList_p = d_blockList_p->next();
        delete q;
    }
}

void *pub_BlockList::allocate(int bytes)
{
    void *p = new char[bytes];
    assert(p);
    d_blockList_p = new pub_BlockLink(p, d_blockList_p);
    return p;
}
```

图 10-19　简单的块分配器组件的实现

原 则

将全局运算符 new 和 delete 工具化，是在系统中理解和测试动态内存分配行为的简单但有效的方法。

复杂的内存分配器的一个共同特性是：复杂并极易出错，但它们都有使其难于测试且测试开销很大的微小接口。幸运的是，作为测试程序开发者，我们有主程序（见 7.7 节）。因此，我们知道总是能够利用重定义全局的 new 和 delete 来满足我们自己的要求[1, 2]。

图 10-20 显示了怎样利用全局的 new 和 delete 来赢得分配器正如所期望的那样工作的信心。在每一次调用中，重定义的全局运算符 new 简单地声明它已被调用以及它所需的内存大小。每个对重定义全局运算符 delete 的调用返回要释放的地址。对于测试自身，测试驱动程序 main 创建了一个 pub_BlockList 分配器的实例，使用它分配两个块，并释放这些块，再分配了另一个块，并允许分配器超出范围。

```
// pub_blocklist.t.c
#include "pub_blocklist.h"
#include <iostream.h>
#include <stdio.h>
#include <malloc.h>

void *operator new(size_t sz)
{
    void *p = malloc(sz);
    printf("\t...new(%d): %x\n", sz, p);
    return p;
}

void operator delete(void *addr)
{
    printf("\t...delete(%x)\n", addr);
    free(addr);
}

main()
{
    cout << endl << "TEST DRIVER FOR: pub_blocklist" << endl << endl;
    {
        printf("pub_BlockList b();\n");
        pub_BlockList b;

        printf("b.allocate(100);\n");
        b.allocate(100);
```

① **ellis**，5.3.3 节，60 页。

② 也可参考 **murray**，9.7.2 节，226~229 页。

```
            printf("b.allocate(200);\n");
            b.allocate(200);

            printf("b.release();\n");
            b.release();

            printf("b.allocate(100);\n");
            b.allocate(100);

            printf("b.pub_BlockList::~pub_BlockList()\n");
        }
        printf("end of test\n");
    }
```

图 10-20　使用了工具化的 new 和 delete 的微小的开发测试驱动程序

图 10-21 给出了在我的计算机上运行这个简单驱动程序的结果。

```
john@john: a.out
        ...new(64): 6520
        ...new(64): 65f8
        ...new(64): 6640
        ...new(64): 6688
        ...new(1024): 66d0

TEST DRIVER FOR: pub_blocklist
pub_BlockList b();
b.allocate(100);
        ...new(100): 6ad8
        ...new(8): 6b48
b.allocate(200);
        ...new(200): 6b58
        ...new(8): 6c28
b.release();
        ...delete(6b58)
        ...delete(6c28)
        ...delete(6ad8)
        ...delete(6b48)
b.allocate(100);
        ...new(100): 6ad8
        ...new(8): 6b48
b.pub_BlockList::~pub_BlockList()
        ...delete(6ad8)
        ...delete(6b48)
end of test
john@john:
```

图 10-21　在我的机器上运行微小的开发测试驱动程序的输出

通过“监听”，我们一定能了解到整个系统运行的许多信息。注意，甚至在测试标志显示

之前，已经发生了五次分配（从未被释放的）。头四次是包含 iostream.h 及其对 cin、cout、cerr、和 clog 的灵巧计数器初始化（见 7.8.1.3 节）的直接结果。发生第五次分配是第一次使用 cout 的结果（7.8.1.4 节讨论了一个每次检查初始化的例子）。

原　则

当工具化全局的 new 和 delete 时，使用 iostream 会引起令人不快的负作用。

因为 iostream 使用了全局运算符 new，所以当重定义全局运算符 new 时，我们希望避免使用 iostream。回归到使用更基本的 stdio，可以避免在工具化全局运算符 new 和 delete 时的递归。

通过对其余输出的粗略检查，我们知道块分配器的简单实现工作正常。每次分配都分配了两个块：一个给内存自身，一个给管理它的连接。对两个块的释放导致过去分配的、四个相同地址被删除。最后，在我的计算机上重新分配一个块，结果像第一次那样返回了相同的地址——这是内存被真正释放的信号。但是，在其他平台上一个不同的地址并不一定意味着有问题。注意，一个更复杂的分配器可能对连接和块自身只进行一次内存分配。

第二种分配方案，被称为**缓冲池分配（pool allocation）**。它很适合为大量固定大小的对象分配内存。通过在实现中使用块分配器以保持对大的内存块进行独立追踪，能够实现缓冲池分配。在 10.3.4.2 节的特定例子的上下文中，将对缓冲池分配进行进一步讨论，但在这里提供它只是作为参考（图 10-22）[①]。

```
// pub_pool.h
#ifndef INCLUDED_PUB_POOL
#define INCLUDED_PUB_POOL

class pub_BlockList;
class pub_Pool {
    pub_BlockList *d_blockAllocator_p;          // * is for insulation
    const int d_objSize;
    const int d_chunkSize;
    int d_instanceCount;                        // help detect memory leaks
    struct Link { Link *d_next_p; } *d_freeList_p;
    void replenish();

  private:
    pub_Pool(const pub_Pool&);                  // not implemented
    pub_Pool& operator=(const pub_Pool&);       // not implemented

  public:
    // CREATORS
    pub_Pool(int objectSize, int chunkSize = 0);
        // Create an allocator for a pool of specified size.
```

① 也可参考 **stroustrup**，13.10.3 节，472~474 页（注意那一节的原始印刷包含实质的编码错误）。

```
        // Optionally specify chunkSize in terms of the number
        // of objects for which space is to be allocated each
        // time the pool is replenished.

    ~pub_Pool();
        // Destroy pool AND ALL ASSOCIATED BLOCKS OF MEMORY.

    // MANIPULATORS
    void *alloc();
        // Allocate block of memory of specified number of bytes.

    void free(void *obj);
        // Return the address to the local free pool.

    void dryUp();
        // Release all dynamically allocated memory from this pool.
};

inline
void *pub_Pool::alloc()
{
    if (!d_freeList_p) {
        replenish();
    }
    Link *p = d_freeList_p;
    d_freeList_p = p->d_next_p;
    ++d_instanceCount;                          // help detect memory leaks
    return p;
}
inline
void pub_Pool::free(void *obj)
{
    Link *p = (Link *) obj;
    p->d_next_p = d_freeList_p;
    d_freeList_p = p;
    --d_instanceCount;                          // help detect memory leaks
}

#endif
```

图 10-22（a）　一个缓冲池分配器接口

```
// pub_pool.c
#include "pub_pool.h"
#include "pub_blocklist.h"
#include <assert.h>

enum { DEFAULT_CHUNK_SIZE = 100 };

pub_Pool::pub_Pool(int objSize, int chunkSize)
: d_freeList_p(0)
, d_objSize(objSize >= sizeof(Link) ? objSize : sizeof(Link))
, d_chunkSize(chunkSize > 0 ? chunkSize : DEFAULT_CHUNK_SIZE)
```

```
, d_blockAllocator_p(new pub_BlockList)
, d_instanceCount(0) // help detect leaks
{
    assert(objSize > 0);
    assert(d_blockAllocator_p);
}

void pub_Pool::dryUp()
{
    d_blockAllocator_p->release();
    d_freeList_p = 0;
}

pub_Pool::~pub_Pool()
{
    if (0 == d_instanceCount) {                    // help detect leaks
        delete d_blockAllocator_p;
    }
    assert(0 == d_instanceCount);                  // really help detect leaks
}

void pub_Pool::replenish()
{
    int size = d_chunkSize * d_objSize;
    char *start = (char *) d_blockAllocator_p->allocate(size);
    assert(start);
    char *last = &start[(d_chunkSize - 1) * d_objSize];
    for (char *p = start; p < last; p += d_objSize) {
        ((Link *)p)->d_next_p = (Link *)(p + d_objSize);
    }
    ((Link *)last)->d_next_p = 0;
    d_freeList_p = (Link *) start;
}
```

图 10-22（b） 使用一个块分配器的缓冲池分配器实现

10.3.4 特定类内存管理

C++语言允许我们在逐个类的基础上，通过重新定义特定类（class-specific）的运算符 new 和 delete 来接管动态内存分配过程。任何人如果分配类的动态实例，将自动接到一个指向由特定类分配器所提供的内存的指针。

有时一个较高层的管理器对象将使用一个较低层的、其实例在整个管理器生存期内要反复动态构造和析构的类。应用这些从属对象的特殊用途的分配器，能够不止一倍地提高管理器的整体速度。

在这一节中，我们在 geom_Point 实例的一个简单的**优先级队列（也称为堆）**的上下文中仔细分析了特定类内存分配策略的应用。我们还研究了一些关于泄漏检测和大系统集成的潜在问题。10.3.5 节中介绍的特定对象分配策略提供了通用的集成问题解决方案。

图 10-23 给出了 my_PointQueue 类的接口。这个点队列的构造函数允许我们提供关于我们期望队列增长到多大的物理提示。点通过 push 方法加入到队列，其优先级由方法的第二个参数 cost 给出。通过 pop 方法，当前 cost 值最小的点被参数返回，与之相关的 cost 值通过值返回。

```
// my_PointQueue
#ifndef INCLUDED_MY_POINTQUEUE
#define INCLUDED_MY_POINTQUEUE

class geom_Point;
class my_PointQueueEntry;

class my_PointQueue {
    my_PointQueueEntry **d_heap_p;          // physical array
    int d_size;                             // physical size
    int d_length;                           // logical length

  private: // not implemented
    my_PointQueue(const my_PointQueue&);
    my_PointQueue& operator=(const my_PointQueue&);

  public:
    // CREATORS
    my_PointQueue(int maxLengthHint = 0);  // physical hint (see Section 10.3.1)
    ~my_PointQueue();

    // MANIPULATORS
    void push(const geom_Point& point, double cost);
    double pop(geom_Point *returnValue);
        // Undefined if length is 0.

    // ACCESSORS
    int length() const { return d_length; }
};

#endif
```

图 10-23 优先级队列组件 my_pointqueue 的接口文件

这个优先级队列的实现使用了一个辅助类——my_PointQueueEntry。my_PointQueueEntry 由 geom_Point 派生，保存相关的 cost 值。这个辅助类被局部地定义在组件的.c 文件中。图 10-24 提供了该优先队列的完全实现以供读者参考；但是，我们关心的是内存管理问题而不是数据结构本身[①]。

① 见 **aho83**，4.10-4.11 节，135~145 页。

与队列操作相关的大量运行时开销，存在于后续的 push 和 pop 期间 my_PointQueueEntry 实例的动态分配和回收上。另外，和 10.3.2 节中讨论的 my_Stack 对象的相应操作相比，优先队列的每次 push 和 pop 所做的工作都是实质性的。与分配及其他的开销相比，这里的 STACK_SIZE 和 GROW_FACTOR 的特定值效果事实上是不可检测的；但对于大的队列，由于使用固定 GROW_SIZE 而导致的性能下降仍然比较明显。

```
// my_pointqueue.c
#include "my_pointqueue.h"
#include "geom_point.h"
#include <memory.h>             // memcpy()

// STATIC FILE SCOPE DEFINITIONS WITH INTERNAL LINKAGE
inline int parent(int i) { return (i + 1) / 2 - 1; }
inline int firstChild(int i) { return (i + 1) * 2 - 1; }
enum { INITIAL_SIZE = 1, GROW_FACTOR = 2 };

// (LOCAL) AUXILIARY CLASS
class my_PointQueueEntry : public geom_Point {
    double d_cost;

  private:
    my_PointQueueEntry(const my_PointQueueEntry&);
    my_PointQueueEntry& operator=(const my_PointQueueEntry&);

  public:
    // CREATORS
    my_PointQueueEntry(const geom_Point& point, double cost)
    : geom_Point(point), d_cost(cost) {}
    ~my_PointQueueEntry() {}

    // ACCESSORS
    double cost() const { return d_cost; }
};

my_PointQueue::my_PointQueue(int size)
: d_length(0)
, d_size(size > INITIAL_SIZE ? size : INITIAL_SIZE)
{
    d_heap_p = new my_PointQueueEntry *[d_size];
}

my_PointQueue::~my_PointQueue()
{
    while (--d_length >= 0) {
        delete d_heap_p[d_length];
    }
    delete [] d_heap_p;
}
```

```
void my_PointQueue::push(const geom_Point& point, double cost)
{
    if (d_length >= d_size) {
        my_PointQueueEntry **tmp = d_heap_p;
        d_heap_p = new my_PointQueueEntry *[d_size *= GROW_FACTOR];
        memcpy(d_heap_p, tmp, d_length * sizeof *d_heap_p);
        delete [] tmp;
    }

    my_PointQueueEntry *newNode = new my_PointQueueEntry(point, cost);
    int n = d_length++;

    // SIFT UP

    while (n > 0) {
        int p = parent(n);
        if (d_heap_p[p]->cost() <= newNode->cost()) {
            break;
        }
        d_heap_p[n] = d_heap_p[p];
        n = p;
    }

    // n is now the index at which to insert the new node
    d_heap_p[n] = newNode;
}

double my_PointQueue::pop(geom_Point *returnValue)
{
    assert(length() > 0);
    *returnValue = *d_heap_p[0];
    double cost = d_heap_p[0]->cost();
    delete d_heap_p[0];

    int n = 0;
    my_PointQueueEntry *newNode = d_heap_p[--d_length];

    // SIFT DOWN

    while (1) {
        int c = firstChild(n);
        if (c >= d_length) {
            break;                          // no children
        }

        if (c + 1 < d_length) {             // two children
            if (d_heap_p[c+1]->cost() < d_heap_p[c]->cost()) {
                ++c;                        // adjust to second child
            }
        }

        // c is index of minimum child, whether first or second

        if (d_heap_p[c]->cost() >= newNode->cost()) {
```

```
            break;                          // all children are not smaller
        }
        d_heap_p[n] = d_heap_p[c];
        n = c;
    }

    // n is now the index at which to insert the new node

    d_heap_p[n] = newNode;
    return cost;
}
```

图 10-24 优先级队列组件 my_pointqueue 的实现文件

10.3.4.1 增加定制的内存管理

默认情况下，每个 my_PointQucueEntry 对象的分配都由用途广泛的、与全局运算符 new 和 delete 相关联的全局分配器完成。但是，利用一个特定类的所有实例有相同大小（即类的 sizeof）这个事实，我们可以创建自己的分配方案，以提高性能。

图 10-25 显示了我们可以怎样开发一个定制的内存管理系统，以降低我们必须访问的相对较慢的全局分配器的频率。我们可以为许多实例分配足够的内存（由 CHUNK_SIZE 指定），而不从每次为一个实例申请内存空间。每个大的块从概念上被分为 CHUNK_SIZE 个更小的连续块，然后它们被合成为一个链表，并植根于静态类成员 s_freeList_p。

现在首次调用特定类的 new 运算符将产生一个大的全局分配器；但是，接下来的 CHUNK_SIZE-1 次分配，能够简单地通过从自由链表中断开一个可用的块来完成。当删除一个 my_PointQueueEntry 类的实例时，它的固定大小的内存块被压入自由链表的前部，以便在后续的重分配中使用[①]。

```
// ...
#include <stddef.h>  // size_t
// ...
enum { INITIAL_SIZE = 1, GROW_FACTOR = 2, CHUNK_SIZE = 100 };
struct Link { Link *d_next_p; };                    // no external linkage

// AUXILIARY CLASS (WITH SOME EXTERNAL LINKAGE)
class my_PointQueueEntry : public geom_Point {
    static Link *s_freeList_p;                        // external linkage
    static void replenish();                          // external linkage
    double d_cost;

  private:
```

① 也可参考 **murray**，9.12.3 节，238~242 页。

```
    my_PointQueueEntry(const my_PointQueueEntry&);              // not impl.
    my_PointQueueEntry& operator=(const my_PointQueueEntry&);   // not impl.

  public:
    // STATICS
    void *operator new(size_t)
    {
        if (!s_freeList_p) {
            replenish();
        }
        assert(s_freeList_p);
        Link *p = s_freeList_p;
        s_freeList_p = p->d_next_p;
        return p;
    }
    void operator delete(void *addr, size_t)
    {
        ((Link *) addr)->d_next_p = s_freeList_p;
        s_freeList_p = (Link *) addr;
    }

    // CREATORS
    my_PointQueueEntry(const geom_Point& point, double cost) :
                                                geom_Point(point), d_cost(cost) {}
    ~my_PointQueueEntry() {}

    // ACCESSORS
    double cost() const { return d_cost; }
};

Link *my_PointQueueEntry::s_freeList_p = 0;

void my_PointQueueEntry::replenish()
{
    int size = CHUNK_SIZE * sizeof(my_PointQueueEntry);
    char *start = new char[size];
    char *last = &start[(CHUNK_SIZE - 1) * sizeof(my_PointQueueEntry)];
    for (char *p = start; p < last; p += sizeof(my_PointQueueEntry)) {
        ((Link *)p)->d_next_p = (Link *)(p+sizeof(my_PointQueueEntry));
    }
    ((Link *)last)->d_next_p = 0;
    s_freeList_p = (Link *) start;
}
```

图 10-25　典型的定制的特定类内存管理方案

为了说明在仿真操作下这个分配方案的相对有效性，我开发了一个性能测试驱动程序（如图 10-26 所示），它和用于测试图 10-16 中 my_Stack 的那个程序相似。在循环的每次迭代中，有任意相关值的三个点被压入到队列中，然后有最低值的当前点被弹出。注意，在每次迭代

中，队列的长度增加 2。

```
// my_pointqueue.t.c
#include "my_pointqueue.h"
#include "geom_point.h"
#include <iostream.h>
#include <stdlib.h>          // atoi()

main(int argc, const char *argv[])
{
    int repeat = argc > 1 ? atoi(argv[1]) : 0;
    cout << "repeat = " << repeat << "\t";

    my_PointQueue q;
    geom_Point p(0,0);

    for (int i = 0; i < repeat; ++i) {
        int x = i * i * i * i * i;
        q.push(p, x % 9999);
        q.push(p, x % 7777);
        q.push(p, x % 3333);
        double cost = q.pop(&p);
    }
}
```

图 10-26　my_PointQueue 的微小的性能“加强”测试驱动程序

图 10-27 显示了只改变 CHUNK_SIZE 参数时，运行图 10-26 所示的驱动程序的结果。第一行代表对八个物理配置中的每一个进行循环的 1000 次迭代。每个后续行将迭代的次数翻一番，直至 1024000 次迭代。头两列分别代表使用固定 GROW_SIZE 和预分配队列数组的最大值的效果（如，我们在 10.3.2 节 my_Stack 所做的那样）。我们再次看到（列 I），非适应的方法（没有特定类内存管理）应用于较大规模的队列会产生严重的性能问题。另一方面，提前预分配整个队列数组（列 II），与 10.3.2 节提倡的简单 grow-factor 方法（列 III）相比仅有微不足道的运行时性能收益。

通过增加特定类的内存管理，一次只为两个 my_PointQueueEntry 对象分配内存空间（列 IV），相对于用全局运算符 new 分配单个 my_PointQueueEntry 对象的方法（列 III），我们获得了显著的性能提高。一次为四个 entry 分配内存（列 V）则进一步改善了性能。注意，使用块分配时，不仅运行时性能提高了，而且管理个体的、全局分配的内存块所需的实际的空间开销也减少了。

当 CHUNK_SIZE 值达到 100 时（列 VII），我们可以看到逐渐减少的返回值。我们在至少 *N* 次局部分配中分摊全局分配的开销，这样可以使一次局部分配的有效开销（最多）比全局分配开销的 1/*N* 多一点。如果回收与分配更紧密、更均匀地交错，那么这种特定类的分配

技术会更有吸引力（并且全局分配器越慢，改进的效果越好）。

(GROW_SIZE):	100							
INITIAL_SIZE:	0	1024002	1	1	1	1	1	1
GROW_FACTOR:		2	2	2	2	2	2	2
CHUNK_SIZE:				2	4	10	100	1000
迭代	I	II	III	IV	V	VI	VII	VIII
1000	0.0	0.0	0.0					
2000	0.1	0.1	0.1	0.0	0.0	0.0	0.0	0.0
4000	0.4	0.2	0.2	0.1	0.1	0.1	0.1	0.1
8000	1.2	0.5	0.5	0.3	0.3	0.3	0.3	0.2
16000	4.1	1.1	1.1	0.6	0.6	0.6	0.5	0.5
32000	10.9	2.4	2.5	1.3	1.2	1.2	1.2	1.2
64000	29.9	5.0	5.1	2.7	2.6	2.5	2.5	2.5
128000	87.2	10.6	11.1	5.7	5.5	5.5	5.3	5.3
256000	296.1	22.3	22.6	11.9	11.3	11.3	10.9	10.6
512000		47.1	47.9	24.2	23.6	23.6	23.1	22.6
1024000		101.2	104.3	51.1	48.3	48.3	47.9	47.3

在 SUN SPARC 20 工作站上运行（优化）的 CPU 秒数

图 10-27 CHUNK_SIZE 的各种值的相对性能

10.3.4.2 独占内存

定制的特定类分配器常常存在以下问题：它们往往是在整个程序的执行期间收集内存，但它们从不将内存返还给全局分配器。在这个分配方案中，组件的静态 s_freeList_p 部分被激活并初始化为 0。一旦程序开始运行，静态自由链表的内存块将由多个 my_PointQueue 实例共享，这样就没有单一的队列对象可以释放公共缓冲池。这个实现违反了 7.7 节中的主要设计规则，这个规则要求我们在程序退出之前提供某种途径去释放分配给静态变量的内存。因此，我们很难使用自动化工具来确定程序是否已经丢失了这块内存。

原 则

从不返还其内存的特定类分配方案，使得对内存泄漏的自动检测变得更加困难。

为了遵循设计规则，我们可以独立跟踪分配给 my_PointQueueEntry 对象的内存块的情况，然后一旦我们确认已没有 my_PointQueueEntry 对象的未完成实例，我们就可以将这些块返回

给全局分配器。图 10-28 显示了一个完成该目标的、my_PointQueueEntry 辅助类的被分解的重新实现。my_PointQueueEntry 类的实例通过重用图 10-22 中预包装的 pub_pool 组件来管理。pub_Pool 类型的静态 s_allocator 成员在启动时就初始化了。my_PointQueueEntry 的动态实例的所有内存都来自于这个静态分配器对象，而不是来自于全局分配器。程序结尾处没有保留 my_PointQueue 的任何实例；因此当缓冲池被回收时，静态的 pub_Pool 对象释放该内存，并将它返回给全局分配器的做法是安全的。

```
// ...
#include "geom_point.h"
#include "pub_pool.h"
// ...
enum { INITIAL_SIZE = 1, GROW_FACTOR = 2, CHUNK_SIZE = 100 };

// AUXILIARY CLASS (WITH SOME EXTERNAL LINKAGE)
class my_PointQueueEntry : public geom_Point {
    static pub_Pool s_allocator;                          // external linkage
    double d_cost;

  private:
    my_PointQueueEntry(const my_PointQueueEntry&);              // not impl.
    my_PointQueueEntry& operator=(const my_PointQueueEntry&);   // not impl.

  public:
    // STATICS
    void *operator new(size_t)
    {
        return s_allocator.alloc();
    }
    void operator delete(void *addr, size_t)
    {
        s_allocator.free(addr);
    }

    // CREATORS
    my_PointQueueEntry(const geom_Point& point, double cost) :
                                            geom_Point(point), d_cost(cost) {}
    ~my_PointQueueEntry() {}

    // ACCESSORS
    double cost() const { return d_cost; }
};

pub_Pool my_PointQueueEntry::s_allocator(sizeof(my_PointQueueEntry),
                                                          CHUNK_SIZE);
```

图 10-28 重用 pub_Pool 分配器来实现 my_PointQueueEntry

如果你仔细研究图 10-22（a）中 pub_Pool 分配器的列表，就会发现它保存有其内部实例的计数。当存在未完成的的实例时，试图回收这个缓冲区将被特定的分配器认为是个程序错误。在这样的情况下，缓冲区拒绝返回任何它分配的块——故意泄漏内存。如果泄漏了一个实例，也就泄漏了所有的实例；现在我们可以更容易地用自动工具检测到这个问题。要强调的是，该错误条件是用 assert 语句“建档”在 pub_Pool 的析构器中的。

值得指出的是，my_PointQueueEntry 的这个实现是安全的，因为 pub_Pool 的静态实例与管理 my_PointQueue 对象存在于相同的编译单元中，因此能保证在其使用前被初始化。但是，考虑图 10-29 的例子，并假设 my_PointQueueEntry 被改为定义在与 my_PointQueue 隔开的编译单元中。如果在启动时 main.o 的静态初始化发生在 my_pointqueueentry.o 的静态初始化之前，这个行为将是未定义的（可能也不太恰当）。无论何时一个分配器的静态实例和定义管理对象的编译单元在物理上相分离，我们就需要采取额外的预防措施来确保启动时创建的管理对象实例的行为正确[①]。

```
// main.c
#include "my_pointqueue.h"

int f()
{
    my_PointQueue q;
    q.push(geom_Point(1,2), 3.0);
    return 4;
}

int i = f();    // occurs at startup

main() {}
```

图 10-29　因静态初始化而产生的微妙的未定义行为

原　则

特定类的内存分配器倾向于占用全局分配的内存，因此增加了整个内存的使用。

在隔离的情况下，特定类的内存分配器往往工作得很好。这样的实现不会违反任何设计规则，但在大型系统的上下文中，特定类的分配方法在集成方面会导致潜在的严重后果。在程序执行过程中，要构造和析构实例；在某些时候，可能会存在一个给定类的众多实例——而在另一些时候则相对很少。特定类分配会导致整个内存需求增长，因为在程序整个执行期间每个类都顽固地保留着它曾使用过的最大内存。存储在私有分配器中的内存不能通用。随

① 见 **meyers**，Item 47，178~182 页。

着时间的积累，每个单个的缓冲池都将达到其“高水位”。

图 10-30 显示了一个包含多个静态缓冲池的系统的典型的内存使用方式。启动时缓冲池最初是空的。在第一阶段的处理中，池 A 和池 B 使用相当频繁，而池 D 和池 E 很少使用。至阶段 II，不再需要与池 A 和池 B 相联系的实例。池 C 使用频繁，池 D 使用适中。在阶段 III，只大量需要使用池 E 的实例。即使系统中的活动对象永远不需要超过 25MB，总的内存使用也已经超过该值三倍了。如果只有少许类使用特定类分配策略，那么可能不会有问题。但是在一个大型系统环境中，很难做到“少许”。

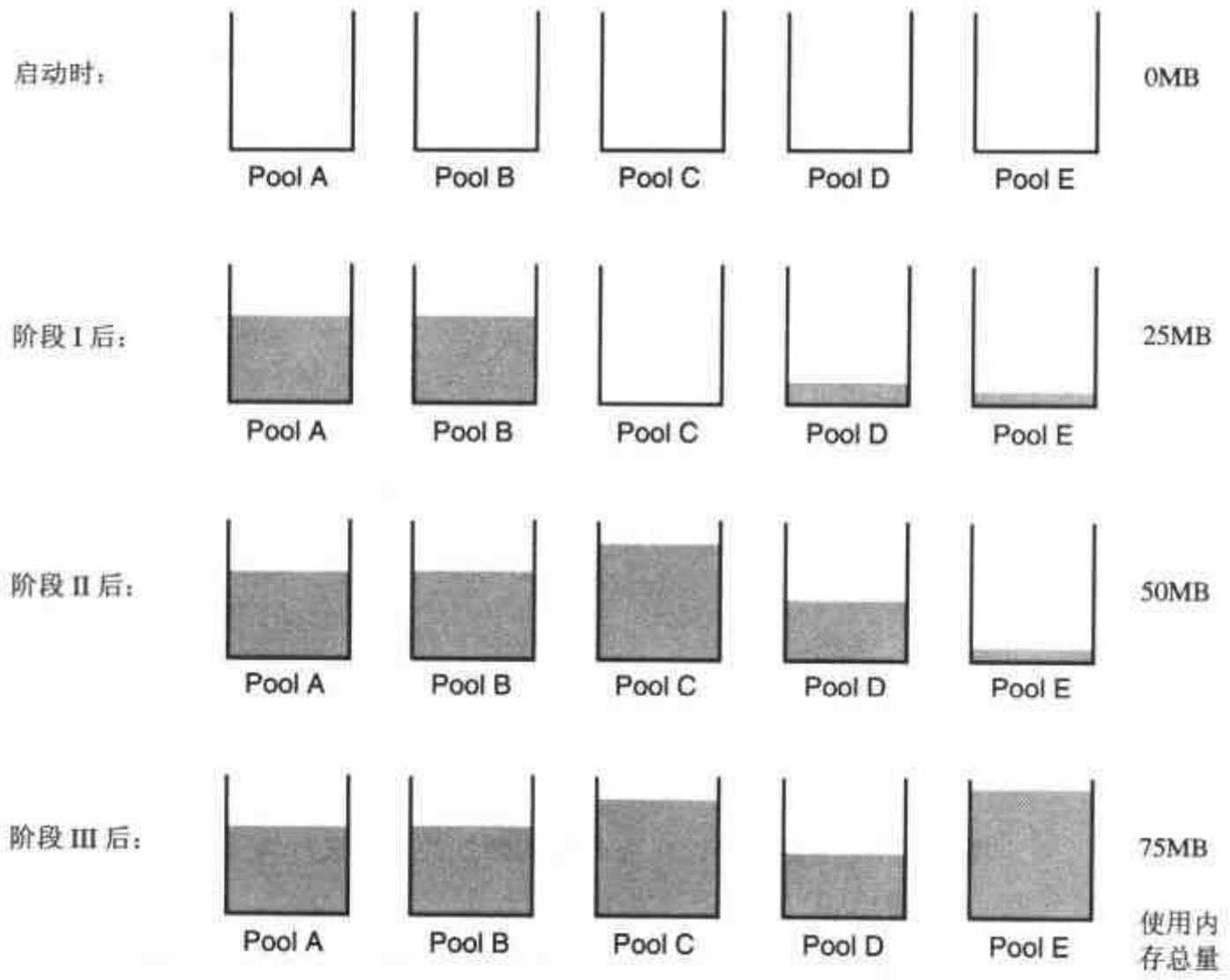


图 10-30 由于特定类分配而引起的典型内存使用模式

通过扩展先前的实现，以设立如图 10-31 所示的实例计数方案，我们可以试图改善这种情况。理论依据在于，当实例计数达到 0 时，释放缓冲区中的所有块并重新开始是安全的做法。但是，这个类的一个单个的静态实例，或一个使用这个类的一个实例的类，就足以使整个缓冲池中的内存在程序运行期间都被牵制在私有的静态分配器中。

```
// AUXILIARY CLASS (WITH SOME EXTERNAL LINKAGE)
class my_PointQueueEntry : public geom_Point {
    static pub_Pool s_allocator;                    // external linkage
```

```
    static int s_instCount;                          // new; external linkage
    double d_cost;

  private:
    my_PointQueueEntry(const my_PointQueueEntry&);              // not impl.
    my_PointQueueEntry& operator=(const my_PointQueueEntry&);   // not impl.

  public:
    // STATICS
    void *operator new(size_t)
    {
        ++s_instCount;                                          // new
        return s_allocator.alloc();
    }
    void operator delete(void *addr, size_t)
    {
        s_allocator.free(addr);
        if (--s_instCount <= 0) {                               // new
            s allocator.dryUp();                                // new
        }                                                       // new
    }

    // CREATORS
    my_PointQueueEntry(const geom_Point& point, double cost) :
                                                  geom_Point(point), d_cost(cost) {}
    ~my_PointQueueEntry() {}

    // ACCESSORS
    double cost() const { return d_cost; }
};

pub_Pool my_PointQueueEntry::s_allocator(sizeof(my_PointQueueEntry),
                                                              CHUNK_SIZE);
int my_PointQueueEntry::s_instCount = 0;                          // new
```

图 10-31　为特定类分配器提供某种自动清理服务

原　则

对特定类的内存管理方法不加选择地使用，是一种以自我为中心的形式，它会对一个集成系统的整体性能产生负面影响。

问题的根源在于，我们在依赖低级的对象去做超过它合理能力的事。这个问题的解决方案——像此书的许多其他解决方案一样——也是将分配责任升级到一个能使其更有效执行的更高层次上。

10.3.5 特定对象内存管理

类的独立实例应有独立的行为[1]。这个观念既可以应用于功能方面，又可以应用于组织方面。特定类的内存管理方法不知道什么时候删除其缓冲池的一部分是安全的。另一方面，每个客户对象都知道将应用从属对象的上下文，这样就可以更清楚地知道什么时候不再需要它们。

原 则

一个特定对象（object-specific）的内存分配方案有足够的上下文可以知道什么时候不再需要一些实例的子集而可以释放它们（这些实例子集是分配给特定对象并由它来管理的）。

升级分配责任的概念和 5.8 节关于链表的讨论相类似，在那个例子中连接递归地删除其自身（见图 5-71）。在那儿我们还做了一个一般雇员不相互雇用和解雇的类比。在这里我们将管理者奉若神明，并假设我们的整个命运完全掌握在我们的创造者手中。

指导方针

特定对象的内存管理比特定类的内存管理要好。

在 my_PointQueue 的例子中，我们可以通过将分配的责任从辅助类 my_PointQueueEntry 升级到容器对象 my_PointQueue 自身，来解决内存耦合问题。为了将最初的（未优化的）my_pointqueue 组件（如图 10-23 和图 10-24 所示）转换为基于每个客户的辅助类的分配实例，我们需要有针对每个客户对象的惟一的缓冲池分配器。因此我们将向 my_pointqueue.h 文件的 my_PointQueue 类中增加一个 pub_Pool *数据成员，如下所示：

```
// my_pointqueue.h
// ...
class pub_Pool;                     // <- add this
// ...
class my_PointQueue {
    pub_Pool *d_allocator_p;        // <  add this
    // ...
```

注意，我们明智地避免了直接将分配器作为数据成员嵌入，这样就避免了使我们的客户必须包含 pub_Pool 头文件。分配器自身由 my_PointQueue 实例管理。附带地说一下，下面的指导方针只是一般常识。

① 见 **cargill**，第 6 章，119 页。

指导方针

使用一个非 const 指针数据成员来保存**被管理的**对象。

我们将需要使用全局运算符 new 的**布放语法（placement syntax）**[1]（在 new.h 中声明），以便将 my_PointQueueEntry 类的实例布放在由缓冲池分配器提供的定制大小的内存块上。我们仍然需要包含缓冲池的定义，并说明希望的 CHUNK_SIZE：

```
// my_pointqueue.c
// ...
#include "pub_pool.h"                          // <- add this
#include <new.h>                               // <- add this
// ...
enum { INITIAL_SIZE = 1, GROW_FACTOR = 2, CHUNK_SIZE = 100 };
// ...                                         // ^^^ add this ^^^ (again)
```

构造函数现在必须分配一个 pub_Pool 来初始化 d_allocator_p 数据成员：

```
my_PointQueue::my_PointQueue(int size)
: d_length(0)
, d_size(size > INITIAL_SIZE ? size : INITIAL_SIZE)
, d_allocator_p(new pub_Pool(sizeof(my_PointQueueEntry))) // <- add this
{
    d_heap_p = new my_PointQueueEntry *[d_size];
}
```

现在我们需要修改析构函数。即使被管理的对象有一个空的析构函数，我们仍然需要显式地释放每个指针；这个特别的缓冲池分配器本身在对被我们释放的每个实例进行计数，以帮助检测内存泄漏。

将

```
my_PointQueue::~my_PointQueue()
{
    while (--d_length >= 0) {
        delete d_heap_p[d_length];
    }
    delete [] d_heap_p;
}
```

改为

```
my_PointQueue::~my_PointQueue()
{                                      // This loop turns out not to be needed for
    while (--d_length >= 0) {          // a pool that does not track instances.
        d_heap_p[d_length]->my_PointQueueEntry::~my_PointQueueEntry();
        d_allocator_p->free(d_heap_p[d_length]);
```

① **ellis**，5.3.3 节，60 页；也可参见 **murray**，9.5 节，222 页。

```
    }
    delete [] d_heap_p;
    delete d_allocator_p;
}
```

最后两个改变涉及了 push 和 pop 方法。若要 **push** 一个新值，我们应该将执行 push 操作的那一行改变为使用运算符 new 的布放语法：

将

```
my_PointQueueEntry *newNode = new my_PointQueueEntry(point, cost);
```

改为

```
my_PointQueueEntry *newNode =
    new(d_allocator_p->alloc()) my_PointQueueEntry(point, cost);
```

最后，我们需要将删除 entry 的那一行改为 **destroy** 而不是 **delete** 该对象，然后返回其内存给特定对象分配器：

将

```
delete d_heap_p[0];
```

改为

```
d_heap_p[0]->my_PointQueueEntry::~my_PointQueueEntry();
d_allocator_p->free(d_heap_p[0]);
```

如果 entry 对象的析构函数不做任何事情，可以选择 destroy 该对象；然而，它必须总是被释放。如果我们想用另一种缓冲池分配器（例如不进行实例追踪的分配器）代替 pub_Pool，所有留在队列中的 entry 都可以简单地通过删除分配器而更有效地被释放。

次要设计规则

避免依赖一个对象在初始化过程中定义数据成员的顺序。

注意，d_heap_p 不是 my_PointQueue 构造函数初始化列表的一部分。也就是说，我们本可以这样编写代码：

```
my_PointQueue::my_PointQueue(int size)
: d_length(0)
, d_size(size > INITIAL_SIZE ? size : INITIAL_SIZE)
, d_heap_p(new my_PointQueueEntry *[d_size])          // bad idea
{
}
```

但是，数据成员初始化的次序由它们在类定义中被声明的次序决定，而不是由它们在初

始化列表中出现的次序决定[1]。在类定义中（见图 10-23），不能保证 d_size 在或将一直在 d_heap_p 之前声明。这种错误很微妙而且难以调试。

我们通常可以直接使用已知合法的引入表达式（如，size），以便不依赖这种次序。在这种情况下，size 与一个条件表达式混合在一起产生了 d_size（正是我们想得到的）。我们不是重复条件表达式，而是将初始化代码放在赋值体中（在性能上没有额外的开销）[2]。

原　则

如果能够利用有关特定用户使用模式的知识，我们通常可以为其管理的对象编写更有效的分配程序。

回到我们的主题：在大多数情况下，特定类和特定对象基于缓冲池的分配策略有类似的运行时性能特性。但是，用特定对象分配策略，管理对象的每个实例都是自治的，即，my_PointQueue 的一个实例不会影响其另一个实例。当析构一个 my_PointQueue 对象时，其所有资源都归还给系统以便重新配置。我们已经达到了我们的运行时性能而又没有过分占用珍贵的全局资源——内存。

作为第二个例子，考虑我们可能如何为一个符号表优化内存分配（见图 10-10）。如果一个符号表被实现为一个符号大小固定的哈希表，那么我们可能使用缓冲池分配。更困难的问题是如何提高分配串的效率。创建一个特定类的串分配器可能仅仅比一个完全通用的分配器好一点点。

另一方面，如果把串的内存与符号表对象关联起来，我们就更容易编写一个优化的、特殊用途的串分配器。例如，我们可以使用特定对象块分配器得到大块的内存，然后分离出大小正合需要的部分。因为分配器是在管理对象的上下文中设计的，所以我们能够知道它的一些预定使用模式方面的信息。例如，我们可能知道符号很少（或从不）会从表中删除。使用通用的特定类串分配方案，不论我们需要与否，我们都将不得不负担分别删除串的开销。在符号表中，因为我们很清楚地知道，当符号表自身被析构时，它们将被块分配器覆盖，所以我们可以采用不对这些串进行处理的简单方法。

指导方针

考虑提供一种方法在块分配和动态内存的单独分配之间进行切换。

依赖特定对象的块分配器来改善性能并不是十分安全的。如果使用不小心，块分配器有一个屏蔽内存泄漏的坏习惯，这样内存泄露就不能被内存分析工具检测到。再次考虑 my_PointQueue 的特定对象缓冲池分配策略。假设我不慎忘记了在对象被弹出时立即释放它：

① 关于如何处理这个问题的一个不同意见，见 **meyers**，Item 13，41~42 页。

② 见 **meyers**，Item 12，37~41 页。

```
double my_PointQueue::pop(geom_Point *returnValue)
{
    assert(length() > 0);
    *returnValue = *d_heap_p[0];
    double cost = d_heap_p[0]->cost();
    d_heap_p[0]->my_PointQueueEntry::~my_PointQueueEntry();
    // d_allocator_p->free(d_heap_p[0]); // oops! (forgot this line)

    // ...
}
```

定义：当程序失去了对动态分配的内存块进行释放的能力时会发生内存泄漏。

严格地说，上面的编程错误不会导致内存泄漏，因为对象的块分配器仍然持有一个指向内存的指针；但实际上，对象已经失去了这块内存的使用权。如果我们正在使用一个追踪实例的分配器，如 pub_Pool，我们会发现当析构 my_PointQueue 对象时，会立即发生“泄漏”。另外，内存仅在对象自身被析构时才返回，并且出错了我们也决不会知道；然而，对象所需要的内存数量可能变得极大。

内存管理是件非常复杂的事情。如果我们想从中受益，就应该警惕，以免受其害。一种解决的方法是提供某种切换机制，以允许测试工程师取消优化的分配策略，并恢复到对全局运算符 new 和 delete 的单独调用上来。如图 10-32 所示，通过使用条件编译能够得到期望的效果。同样的效果在运行时可以用静态的过程接口来实现，或在启动时通过读取环境变量实现。

```
#ifndef DEBUG_ALLOC
    my_PointQueueEntry *newNode =
        new(d_allocator_p->alloc()) my_PointQueueEntry(point, cost);
#else
    my_PointQueueEntry *newNode = new my_PointQueueEntry(point, cost);
#endif

// ...

#ifndef DEBUG_ALLOC
    d_heap_p[0]->my_PointQueueEntry::~my_PointQueueEntry();
    d_allocator_p->free(d_heap_p[0]);
#else
    delete d_heap_p[0];
#endif
```

图 10-32 用优化之外的条件编译来检测内存泄漏

提供非优化运行的能力有两个优点：

（1）它允许计算机辅助软件工程（Computer-Aided Software Engineering，CASE）工具，

如 Purify（或甚至是运算符 new 和 delete 的工具化版本），准确标出内部内存泄漏的位置。

（2）它使你确切知道你的内存管理策略真正为你节省了多少时间和空间。

10.4 在大型工程中使用 C++模板

在概念上，模板使重用的可能性产生了质的飞跃——特别是对那种普通的、低级的容器对象来说更是如此。模板的表达能力使它们成为 C++语言的一个受欢迎的、期望值很高的特点。不幸的是，许多现存的编译器受到严重的性能和/或者耦合问题的困扰，这些问题使得将模板引入到大型工程环境后会变得非常不可靠。在这一节，我们的讨论围绕两个 C++编译器模板实现策略的问题，然后探讨前几章涉及到的与基于模板的容器类开发有关的内存管理的问题 。

10.4.1 编译器实现

大多数编译器用以下两种方式之一实现模板：

（1）CFRONT 型：当在程序中遇到模板时，创建一个系统级的库，以提供编译单元共享的信息。

（2）MACRO 型：用户必须可以使用模板组件的头部分和实现部分的源代码。

在 CFRONT 采取的方法中，执行一个“模拟的”连接，以决定哪些未定义的符号可以通过模板实例化来解析。在正常的模式中，解析这些符号可能（并且一般也会）导致新的未定义符号从新实例化的函数体中产生。这个过程一直重复到模拟的连接再也找不到任何新符号为止。模拟连接的数目能够和函数调用层次结构的深度一样大，这使得这种模式的模板开发在连接时间方面代价太大。在哥伦比亚大学，跟我学习面向对象设计课程的学生遇到过这样的情况——在 SUN SPARC 2 工作站上使用 CFRONT 3.0，完成了一个链表模板组件的编译并将它与其测试驱动程序连接用了 15 分钟。

像避免实质使用参数化对象的模板并连接预实例化库这样的技术，有助于减少增加的连接时开销，但不能消除它。不幸的是，在一些平台中仅仅一个模板的使用就足以极大地影响连接时间。

MACRO 方法不会受过长连接时间问题的困扰，但它产生了一个与绝缘有关的问题。必须向客户提供实现源码，也就意味着代码不再被认为是私有的了。

尽管模板有明显的价值，在编写本书时（1996 年 5 月）我还是不断听到有关大型工程中使用模板的可怕事例，它们证明了我的经验：使用模板会增加开发时间，尤其是连接时间。

---

① 也可参见 **stroustrup**，第 8 章，255~292 页；以及 **murray**，第 7-8 章，141~203 页。

随着 ANSI/ISO 委员会最近对标准模板库（STL）的采纳，对于能够向模板提供有效而且健壮的支持的大众化编译器的需求日益迫切。

作为一个满怀期待的 C++程序员，我热切地期望 C++编译器的下一次浪潮，希望它能够为大型工程更有效地解决这些缺陷[2]。

10.4.2 在模板中管理内存

编写一个好的模板容器类比为一个特定对象编写一个等价的容器类要困难得多。在编写模板时，我们必须注意到参数类型可以是基本数据类型，也可以是自我管理动态内存的用户自定义类型。从一个基本类型中直接派生是不可能的，一般也不可能使用位拷贝（如，memcpy）来移动用户自定义类型。在后面的部分，我们将探讨一些涉及在模板类中进行内存管理的复杂问题。

原 则

通过在一个伪参数化类型中嵌入实际参数类型（如 gen_StackItem<T>嵌入到 gen_Stack<T>），我们可以在继承方面像处理用户自定义类型那样处理基本类型，并且允许它有寻址和分配功能，这些功能在所有用户自定义类型中都是没有的。

如图 10-33（a）所示，为了解决怎样从基本参数类型派生的问题，我们总是可以构造一个它包含（HasA）参数化类型的伪模板结构。然后我们可以如图 10-33（b）所示那样，从这个伪类型中派生，增加特定类的 new 和 delete 运算符，重新建立运算符的一个私有地址（单目的&），或对一个协同运作的用户自定义类型做任何其他我们想做的事。

原 则

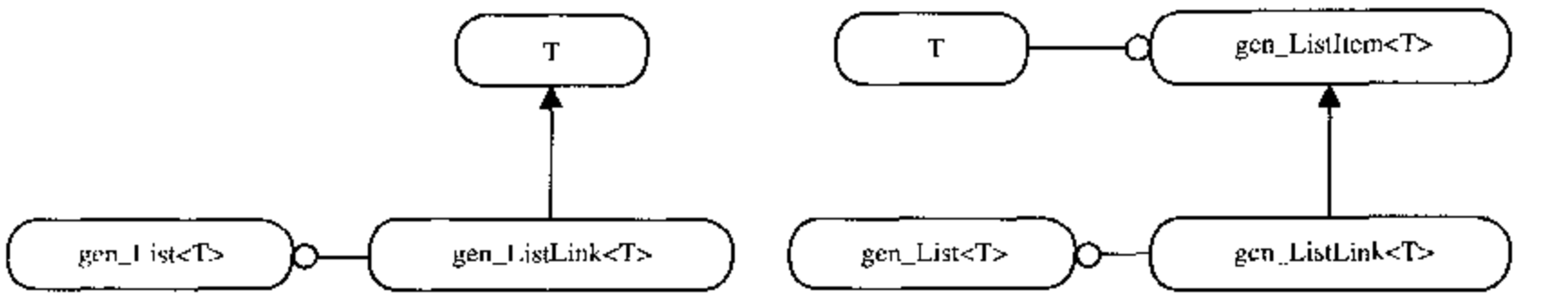
一般来说，一个对象不能使用位拷贝来拷贝或移动。

一般情况下，用 memcpy 将对象拷贝到新的位置是危险的，因为对象可能包含指向它所拥有的另一个下属对象的指针或引用。位拷贝之后，一个对象将有两个实例，每个实例都认为自己单独负责相同的下属对象内存的删除。析构两个实例将导致两次删除下属对象（即一个编程错误）。

甚至用位拷贝移动初始对象一般也不安全。要小心不要去调用初始对象的析构函数，以保证下属对象不会被删除两次，但不能寻址有自引用指针的繁琐对象（如，一个在链表类中嵌入了连接的循环链表）。当用 C++模板实现容器类时，牢记避免对象的位拷贝尤其重要。

① 更多有关 STL 的信息，见 **musser**。

② 对于模板的潜在编译器实现策略的更多信息，见 **stroustrup94**，15.10 节，365~378 页。




```
class gen_ListLink : public int {
    // not legal C++
    // ...
};
```

```
struct gen_ListItem {
    int d_data;
};

class gen_ListLink : public gen_ListItem {
    // works for all types in C++
    // ...
};
```

（a）对<int>不工作　　（b）完全通用的实现

图 10-33　从任意的参数化类型中激活派生

memcpy 是否可用于拷贝一个特定对象显然是一个实现细节，并且可以修改。允许一个对象（模板）以这种方式依赖于另一个对象（参数化类型）的实现，不仅破坏了封装性，而且在升级系统时容易出现不易发现的细微的内存错误。

原　则

一般来说，不能使用对象的赋值运算符将对象拷贝或移动到未初始化的内存。

为了说明这个观点，让我们考虑将图 3-2 所示的整型栈转换为一个基于模板的、任意对象的栈。不是直接跳到模板表示法，而是从 typedef(T)代替每个出现的初始栈项目类型（int）开始，这样做通常有很用，如图 10-34 所示。从一开始就使用模板表示法将使我们很难调试模板，并很可能增加我们的开发时间。一旦我们已经使该栈工作在一些基本类型（如，int 或 double）和至少一个重要的用户自定义类型（如，pub_String）上了，转换到 C++模板表示法将是件简单的事情。

```
// stack.h
#ifndef INCLUDED_STACK
#define INCLUDED_STACK

typedef int T;                    // <= Stack item type T, currently an int

class StackIter;

class Stack {
```

```
    T *d_stack_p;
    int d_size;
    int d_sp;
    friend StackIter;

  public:
    Stack();
    Stack(const Stack &stack);
    Stack& operator=(const Stack &stack);
    ~Stack();
    void push(const T& value);
    T pop();
    const T& top() const;
    int isEmpty() const;
};
// ...
#endif
```

图 10-34 将一个类转换为一个模板类（见图 3-2）

我们要做的第一件事是决定怎样分配数组：

```
// stack.c
#include "stack.h"
#include <memory.h>  // memcpy()

enum { START_SIZE = 1, GROW_FACTOR = 2 };

Stack::Stack()
: d_stack_p(new T[START_SIZE])                    // oops!
, d_size(START_SIZE)
, d_sp(0)
{}
```

这样做不好！我们是在物理数组中隐式地分配和**初始化**所有对象。这将非常低效——特别是对重量级的类型而言。作为替换方法，试试这个：

```
Stack::Stack()
: d_stack_p((T*) new char[START_SIZE * sizeof *d_stack_p])  // ok
, d_size(START_SIZE)
, d_sp(0)
{}
```

下一步，我们要设法实现该栈的拷贝构造函数。初始代码如下所示：

```
Stack::Stack(const Stack& s)
: d_stack_p(new T[s.d_size])
, d_size(s.d_size)
, d_sp(s.d_sp)
{
```

```
    memcpy(d_stack_p, s.d_stack_p, d_sp * sizeof *d_stack_p);
}
```

我们现在将上面的代码“明智”地改为：

```
Stack::Stack(const Stack& s)
: d_stack_p((T*) new char[s.d_size * sizeof *d_stack p])
, d_size(s.d_size)
, d_sp(s.d_sp)
{
    for (int i = 0; i < d_sp; ++i) {
        d_stack_p[i] = s.d_stack_p[i];                    // oops!
    }
}
```

我们刚才引入了一个不易察觉的程序错误，这个错误要到我们用一个含有有意义的析构函数的类型实例化栈时才会被发现。注意左边的栈项目是未初始化的垃圾。如果要用 pub_String 参数化我们的模板，考虑下面的 pub_String 赋值运算符在上面的循环中会如何表现：

```
pub_String & pub_String::operator=(const pub_String& string)
{
    if (this != &string) {
        delete d_string_p;                                // yowza!
        d_string_p = init(string.d_string_p);
    }
    return *this;
}
```

次要设计规则

当为一个通用的、参数化的容器模板实现内存管理时，若赋值的目标是未初始化的内存，则要小心不要使用被包含类型的赋值运算符。

由于左边是未初始化的内存，因而其位置的内容可能与右边的 pub_String 对象的地址不符；如果对应于 d_string_p 的内存位置恰好不为 0，那么内存分配系统将会崩溃。当编写基于模板的容器类时，用赋值来将参数类型拷贝到未初始化的内存是一个令人吃惊的常见错误。

指导方针

当为一个完全通用的、参数化的容器类实现内存管理时（该容器类管理其包含对象的内存），假定参数化类型只定义了拷贝构造函数和析构函数——别无其他。

只有使用其拷贝构造函数才能合理地拷贝一个任意对象。要移动一个任意对象只能通过

先使用其拷贝构造函数，然后显式地调用其析构函数才行[1]。

```
#include <new.h>  // declare placement syntax

Stack::Stack(const Stack& s)
: d_stack_p((T*) new char[s.d_size * sizeof *d_stack_p])
, d_size(s.d_size)
, d_sp(s.d_sp)
{
    for (int i = 0; i < d_sp; ++i) {
        new(&d_stack_p[i]) T(s.d_stack_p[i]);    // ok
    }
}
```

当析构栈时，我们应该准确地控制哪些包含的实例要析构而哪些不要析构。初始代码的行为

```
Stack::~Stack()
{
    delete [] d_stack_p;                          // no good!
}
```

现在将是未定义的，并能够解决内存泄漏问题。我们必须像下面这样显式地析构每个活动的栈项目：

```
Stack::~Stack()
{
    for (int i = 0; i < d_sp; ++i) {
        d_stack_p[i].T::~T();
    }
    delete [] (char *) d_stack_p;                 // ok
}
```

这个例子的余下部分沿着同样的路线进行。push 方法需要布放语法；pop 方法需要对参数类型的析构函数（图 10-35 提供了一个完全的栈模板组件作为参考）进行显式的调用。这个例子的寓意是，当编写模板时，我们被迫考虑一个全新的约束尺度，这个约束尺度不只是为特定的类型而存在。我们作出的任何关于参数类的假设即使不影响模板的正确性，也将影响其有用性。但是，更专用的模板类（如：gen_Set 和 gen_OrderedList）指定它们各自的参数类型所要求的函数和操作是合理的。

---

① 注意大多数（但不是全部）带有一个拷贝构造函数的对象同时也实现一个赋值运算符。但是，一个赋值运算符可以用一个析构函数和一个拷贝构造函数来模拟（见 10.2.3 节）。

```
// stack.h
#ifndef INCLUDED_STACK
#define INCLUDED_STACK

template<class T> class StackIter;

template<class T>
struct StackItem {
    T d_item;
    StackItem(const T&);
    ~StackItem(){}  // Some compilers need this (see Section 9.3.3).
};

template<class T>
class Stack {
   StackItem<T> *d_stack_p;
    int d_size;
    int d_sp;
    friend StackIter<T>;

  public:
    Stack();
    Stack(const Stack<T>& stack);
    ~Stack();
    Stack& operator=(const Stack<T>& stack);
    void push(const T& value);
    T pop();
    const T& top() const;
    int isEmpty() const;
};

template<class T>
class StackIter {
    StackItem<T> *d_stack_p;
    int d_sp;
    StackIter(const StackIter<T>&);              // not implemented
    StackIter& operator=(const StackIter<T>&);  // not implemented

  public:
    StackIter(const Stack<T>& stack);
    ~StackIter();
    operator const void *() const;
    const T& operator()() const;
    void operator++();
};

#endif
```

图 10-35（a） 模板 stack 组件的接口（stack.h）

```
// stack.c
#include "stack.h"
#include <new.h>      // declare placement syntax
#include <memory.h>   // memcpy()
#include <assert.h>

enum { START_SIZE = 1, GROW_FACTOR = 2 };

template<class T>
StackItem<T>::StackItem(const T& item)
: d_item(item)
{
}

template<class T>
Stack<T>::Stack()
: d_stack_p((StackItem<T> *) new char[START_SIZE * sizeof *d_stack_p])
, d_size(START_SIZE)
, d_sp(0)
{
}

template<class T>
Stack<T>::Stack(const Stack<T>& s)
: d_stack_p((StackItem<T> *) new char[s.d_size * sizeof *d_stack_p])
, d_size(s.d_size)
, d_sp(s.d_sp)
{
    for (int i = 0; i < d_sp; ++i) {
        new(d_stack_p + i) StackItem<T>(s.d_stack_p[i]);
    }
}

template<class T>
Stack<T>::~Stack()
{
    for (int i = 0; i < d_sp; ++i) {
        d_stack_p[i].StackItem<T>::~StackItem();
    }
    delete [] (char *) d_stack_p;
}

template<class T>
Stack<T>& Stack<T>::operator=(const Stack<T>& s)
{
    if (&s != this) {
        for (int i = 0; i < d_sp; ++i) {
            d_stack_p[i].StackItem<T>::~StackItem();
        }
```

```
            if (d_size < s.d_size) {
                delete [] (char *) d_stack_p;

                d_stack_p = (StackItem<T> *)
                                        new char[s.d_size * sizeof *d_stack_p];
                d_size = s.d_size;
            }
            d_sp = s.d_sp;
            for (int i = 0; i < d_sp; ++i) {
                new(d_stack_p + i) StackItem<T>(s.d_stack_p[i]);
            }
        }
        return *this;
    }

    template<class T>
    void Stack<T>::push(const T& v)
    {
        if (d_sp >= d_size) {
            StackItem<T> *p = d_stack_p;
            d_size *= GROW_FACTOR;
            d_stack_p = (StackItem<T> *) new char[d_size * sizeof *d_stack_p];
            for (int i = 0; i < d_sp; ++i) {
                new(d_stack_p + i) StackItem<T>(p[i]);
                p[i].StackItem<T>::~StackItem();
            }
            delete [] (char *) p;
        }
        new(d_stack_p + d_sp++) StackItem<T>(v);
    }

    template<class T>
    T Stack<T>::pop()
    {
        assert(d_sp > 0);
        StackItem<T> tmp = d_stack_p[d_sp - 1];
        d_stack_p[--d_sp].StackItem<T>::~StackItem();
        return tmp.d_item;
    }

    template<class T>
    const T& Stack<T>::top() const
    {
        assert(d_sp > 0);
        return d_stack_p[d_sp - 1].d_item;
    }

    template<class T>
    int Stack<T>::isEmpty() const
    {
```

```
    return d_sp <= 0;
}

template<class T>
StackIter<T>::StackIter(const Stack<T>& stack)
: d_stack_p(stack.d_stack_p)
, d_sp(stack.d_sp)
{
}

template<class T>
StackIter<T>::~StackIter()
{
}

template<class T>
StackIter<T>::operator const void *() const
{
    return d_sp > 0 ? this : 0;
}

template<class T>
const T& StackIter<T>::operator ()() const
{
    assert(d_sp > 0);
    return d_stack_p[d_sp - 1].d_item;
}

template<class T>
void StackIter<T>::operator++()
{
    assert(d_sp > 0);
    --d_sp;
}
```

图 10-35（b） 模板 stack 组件的实现（stack.c）

指导方针

尽可能，在分解的可重用 void * 指针类型的顶部，使用内联函数来实现模板，以重建类型的安全性。

模板的丰富表达能力隐藏了这样的事实，即为每个实例化模板类型的非内联函数生成完全相同的对象代码。只要有可能，我们更喜欢分解出共同功能，而不是让可执行程序的大小不必要地增大。像 GnodePtrBag（图 5-82）这样的容器类非常适合用于模板的实现：所有的实际代码都能够被分解为单个的非模板类型（如，PtrBag），并且模板能够提供指针类型安全

（通过内联函数），而不会产生任何冗余的对象代码[1]。如果始终遵循这一指导方针，可能会在很大程度上减少可执行程序的大小。

10.4.3 模式与模板

当我们通过使用模板努力以具体形式捕获可重用行为的任何时候，我们都是在提高系统的模块性和可靠性。如果能获得标准库，如 STL，将使重用的好处更加直接而明显。一般的容器类如数组、集合、AVL 树和优先队列，它们的实现本质上独立于它们所包含对象的类型，模板对于描述这些类属的容器类特别有用。

尽管模板是能很好地处理某些问题的强大工具，但并不是所有的共性都能够具体地以模板的形式加以描述。一个更高层次的概念是**设计模式**[2]。

> **定义：设计模式**是类或对象的抽象组织，它被反复证明在解决多领域的相似问题上是有效的。

设计模式是嵌入在更大设计中的小型体系结构。最常用的模式之一是迭代器模式，但若只有迭代则无助于模板实现。设计模式抽象出重复出现的设计结构。一个模式一般会跨越几个组件。设计模式不像抽象那么具体，它捕获更高层的行为，这些行为只描述了系统的一部分，而不需要描述详细的操作。由于设计模式表达的是高层的信息，它能以表面不同的方式被反复重用。

所有有经验的开发者，不论他们是否意识到，其实都在他们的设计中使用了模式。设计模式的使用是经验的自然结果：我们回头查看在相似的情况下我们是如何做的，并重复以前的工作。随着时间的流逝，我们开始收集用于组织对象的策略，并总结一个给定的策略可以应用在什么情况下。

那么为什么确定设计模式这么有用呢？简单来说，因为它能方便和加快开发过程。经验需要大量的时间和努力来积累。数学界从接受负数到 17 世纪微积分的发现，花费了世界上最聪明的头脑超过 1000 年的时间。然而这样精选出来的许多数学信息至今都以一种有组织的形式，一般在高中毕业以前传授给学生。

正如更为确定的学科如数学那样，面向对象的软件工程也应如此。向计算机专业人员提供各种有用的解决方案，并指出应用它们的条件，实践者可以推动技术的进步，而不是不断地重新发明车轮。

当你需要决定一种解决体系结构问题的方法时，如果你对如何在几个可选择的设计模式中挑选出对你的特定系统最有效的模式缺乏经验，那么拥有一些设计模式的目录是很有价值

---

① **stroustrup**，8.3.2 节，264~265 页；也可参见 **murray**，8.6.1 节，188~190 页。

② 见 **gamma**。

的。如果某种方案已经建档了，那么不将该信息结合到你的决策中是愚蠢的。在一些没有可接受的解决方案的情况下，有一个已知有用的配置“辞典”常常能帮助我们及时地得到合适的体系结构。

原 则

设计模式是在体系结构层次上交流可重用概念和思想的有效方法。

除了自身是有效的解决方案之外，设计模式还提供了一个词典——一组使体系结构设计师们能够方便地在较高层次上交流各自思想的词汇。虽然取一个好记忆的名称很有用，但有时不管怎样取名，总比在你每次需要就模式与他人交流时都要对其进行描述要好。

在本书中，我们已经重复使用了几种简单的设计模式。例如：

- 提供非基本函数的集合，作为单独工具类的独立于实例的成员函数（在C++中是static），有助于促进组件的层次化和类接口的最小化。
- 从一个（抽象）类中析取一个纯接口并将这个**协议**类置于一个组件中（这一组件和留下的（部分的）、现在派生于它的实现相分离），将允许客户使用这个接口，而不会在编译或连接时被迫依赖于任何特定的实现。
- 在一个完全定义在其组件的.c文件内的struct中嵌入一个具体类的私有成员，可以使该类将客户程序与其实现完全绝缘（专用于C++语言）。
- 通过在相同组件中提供一个单独的迭代器友元类来封装原理类的各部分或成员的排序细节，可以提高可维护性，而且不会限制并发客户的数目。

对于熟悉它们的人来说，每一个术语如工具、完全绝缘、协议和迭代器等，会直接提供大量的抽象组织信息，这些信息可以排除附加的描述。对于那些不熟悉术语的人来说，这些术语中的每一个都提供了一个名称，它能够用来检索模式的细节。

原 则

设计模式，与设计过程自身一样，能处理逻辑和物理问题。

这本书中描述的许多模式是专用于C++语言的，并由实现一个可靠的物理设计的愿望所驱动。这些都是**物理设计模式**。一个可层次化的组件集合，其中每个组件都由一个接口（.h）文件和一个实现（.c）文件组成，是这些模式中最基本的。

比较而言，逻辑设计模式显然是独立于语言的，它用于解决部分逻辑设计过程出现的问题（如迭代器）。

由Gamma、Helm、Johnson和Vlissides编著的《Design Patterns: Elements of Reusable Object-Oriented Software》的这本书，与本书在同一套丛书中出版，也是其逻辑/体系的补充。其中介绍了23种有用的设计模式，均源自实际运行的系统，并提供了足够的细节，可以立即使用。与所有其他好的设计一样，每个模式都自然地将实现建议为可层次化的协作组件层次

结构。作为一个例子，我已经提供了一个附加的很有用的模式，我称之为**协议层次结构**（**protocol hierarchy**）（见附录 A）。

10.5 小结

在这一章，我们讨论了在 C++中属于单个对象实现的许多不同主题。对齐需求能导致在对象的物理布局中出现空隙。重新安排内部数据有时能减小实例大小，这对于希望在一个时刻有多个实例的类来说非常重要。

在最小化对象规模的工作中，我们有时会使用 short 或 char 基本类型代表整数成员数据，但只有当我们确定不会发生溢出时才可以这么做。试图用 unsigned 得到额外的位，这样做说明整数类型还没有大到足够安全的地步。总的来说，使用 unsigned 易于出错，不予推荐。

Typedef 声明能够被用作特定大小（如，Int32、Float64）的基本类型的别名，这些基本类型取特定大小是为了使在各平台间共享的对象实现保持一致性，如在面向对象数据库中所做的那样。

在实现个别功能的层次上，作出错误决策的代价相对较小，因为这种决策的影响通常局限在系统的很小一部分当中。但是，下面的做法是被推荐的：

- 使用 assert 语句来帮助确认你在整个复杂功能的实现中做出的假设并建档。
- 尽量将边界条件融入到基本算法中，而不是将其作为特殊情况对待。
- 在单个组件中，尽量避免重复代码块；而是将这些代码块重新分解为局部的可重用子程序。
- 一般来说，尽量寻找能有效解决问题的最简单方法。

逻辑状态值有助于一个对象的基本行为，而物理状态值只是特定实现选择的一个技巧。作为一个规则，我们应该努力避免提供对物理值的编程访问。

提示是客户为使对象改善性能而向该对象提供的附加信息。就像 inline 或 register，提示仅仅是一种建议，不应该影响一个对象的逻辑状态。

定制的动态内存管理实现起来可能很复杂，但对获得高性能又经常是必要的。管理动态分配内存的设计空间可以相当大。在 10.3.2 节中，我们研究了扩大一个内部数组规模的两种方法：

- GROW_FACTOR：通过乘法因子增大数组规模。
- GROW_SIZE：通过加入一个固定增量来增大数组规模。

GROW_FACTOR 方法在数组大小范围变化很大时在性能方面更为健壮。在数组极大时，GROW_SIZE 方法的运行时性能低得令人难以接受。尽管 GROW_SIZE 方法潜在地需要更少的空间，但是，大小为 G 的 GROW_FACTOR 将绝不会允许数组超出优化分配的范围多于一个 G 因子的大小的分配。除了不合理的情况，GROW_FACTOR 几乎总是实现数组扩张的首选方法。

在 10.3.3 节，我们研究了两种内存分配器：

- **块分配器**追踪大小潜在变化的每个分配块，并提供一种机制，一旦不再需要内存时，将其全部回收。
- **缓冲池分配器**保持固定大小块的自由缓冲池。当池为空时，调用块分配器来补充缓冲池。当缓冲池被析构时，所有的块被回收。

我们观察到，通过工具化全局运算符 new 和 delete 来开发和使用内存分配器有很多益处。注意，我们将在全局内存运算符函数体中使用在 stdio.h 文件中声明的功能而不是在 iostream.h 文件中声明的功能，因为 iostream 依赖全局的 new 运算符来进行它自己的动态分配。

特定类内存分配意味着类本身包含着静态变量，以帮助管理实例化对象的内存，这常常是通过保留一列未使用的内存块来实现的，这些内存的大小适合该特定对象。这种优化的结果使运行时性能翻倍并非不寻常。

特定类内存管理方案也有一个重大的缺点，因为它们很少向运行时系统返回它们所分配的内存。尽管这块内存没有泄漏，但它不能被普遍使用。很多有特定类内存管理器的大型系统，会占用远远超过其需要的内存。

特定对象内存管理解决了很多问题（这些问题对于特定类的方法来说是不可避免的），并在大多数情况下与特定类对象内存管理有基本相同的运行时性能特性。管理对象有充足的上下文知道受管理对象的使用模式可能是怎样的。管理类因此比下属类更易于优化地管理对象的内存。另外，当管理类被析构时，将向运行时系统返回所有与下属对象相关的内存，于是这些内存又重新可以通用了。结果形成了一个更健壮的系统，它不会在任何时候牵制（远远）超过其需要的内存。

块分**配**技术在提高运行时效率方面是有效的，但它们会掩饰代码错误，这些错误导致不能重用已被析构的个别实例。这样的错误不是内存泄漏问题，但它们能微妙地降低复杂对象的空间性能特性。提供一种机制在块分配和个别分配之间进行切换，将使我们能够对这个易产生错觉但很普遍的问题进行试验，这个试验可以通过工具化全局的 new 和 delete 运算符来完成，也可以使用可以商业购买的工具来进行。

模板是 C++语言表示能力的重要部分。很多已有的编译器实现有严重的缺陷：要么连接时间过长，要么它们的实现不能与客户程序绝缘。STL 的采用对编译器提出了更高的要求，要求它对大型项目中的模板提供有效和健壮的支持。

实现一个基于模板的容器类显然要比实现一个特定对象的等价容器困难得多。重用的模板类通常必须和基本类型以及用户自定义类型一起工作。直接从一个基本类型中继承是不可能的，一般而言，正确地使用在用户自定义类型上的位拷贝（例如，memcpy）也是不可能的。一个常见的错误是使用参数化对象的赋值运算符，将对象拷贝到未初始化的内存中。当我们实现最通用的容器类时，可以假设参数化对象有一个拷贝构造函数和一个析构函数，但是没有其他更多的东西。

模板语法改善了源代码的模块性，但是如果幼稚地使用它，可能会产生许多的冗余对象

代码。通过在通用指针算法的顶部实现模板来重建类型安全，我们可以实质地减少产生过度代码的潜在可能性，并且可以减少必须经过一段难以忍受的连接时间的潜在可能性。

模板解决了某一类问题，但是并不是所有的共性都能够使用模板来表达。模板是一个高层的设计抽象，用来描述一个嵌入在更大型设计内部的、经常重现的微体系结构。设计模式提供了一个可用于有效表达可重用设计概念的辞典。《Design Patterns: Elements of Reusable Object-Oriented Software》一书，是本系列的一部分，对在工业中最经常用到的 23 个模式进行了归类。设计模式的另一个有用的例子，协议层次体系结构在附录 A 中提供，供读者参考。

附录 A

协议层次结构设计模式

对于大型的、高质量的软件系统的有效开发来说，认识及重用一些通用的设计模式是有必要的。本部分将描述一个用于面向对象数据库应用程序的重要模式：协议层次结构。这种模式表示格式类似于《Design Patterns: Elements of Reusable Object-Oriented Software》[1]中分类和解释的23种其他有用模式的表示格式。注意，在那本书中用名称和页码标识这些模式以便于引用（例如，Composite(163)出现在书中的第163页）。

类结构的协议层次结构

目的

把一些松散关联的对象类型的异构集合组织成为一个逻辑层次结构，使每一个不同的和动态的客户群体都能处理一个最大化的实例子集，该实例集是从包含大量不同类型的可扩展类型集合中安全而一致地创建的。

也称做

安全转换（Safe Casting）

---

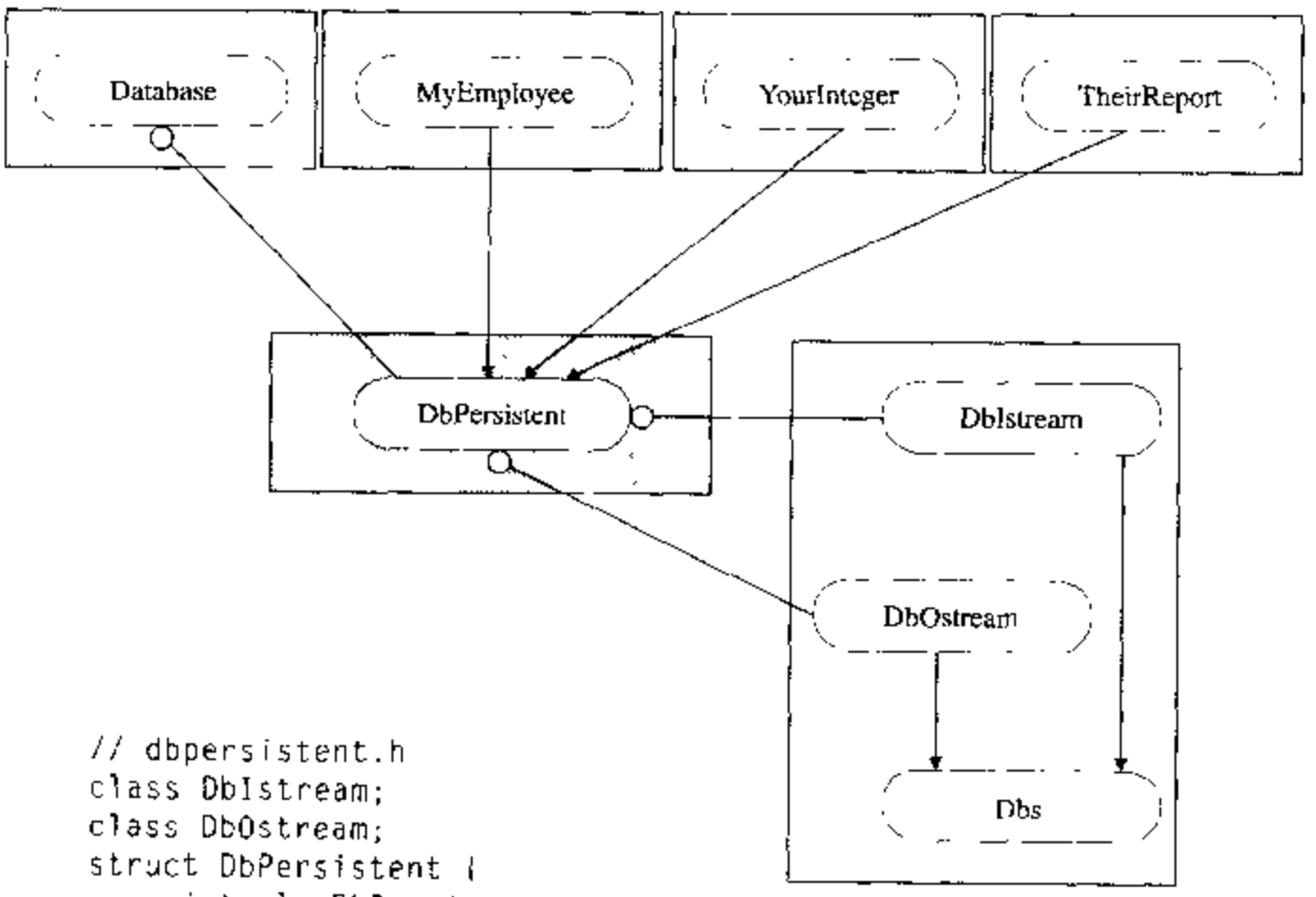
① **gamma**。

动机

客户经常会写一些基于类型集合的应用程序。把每一个类型作为一种特殊情况来处理是很乏味的，并且难以扩展和维护。如果客户具有统一处理广泛对象类型的能力，那么就可以很容易地写出通用的应用程序，并且这些通用应用程序更易于维护。

数据库和图形应用程序经常要求它们所操作的对象实现了某些通用的功能。对于一个对象来说，为了能透明地参与一个特定的应用程序，必须派生自声明了被要求功能的协议类。

例如，一个对象数据库要求每一个具体持久对象实现以一种独立于机器的格式“将自己写”到一个字节流中，并且如下功能：能够将自己从这样一个字节流中读回来。这个通用的功能可被分解成如下的抽象基类（DbPersistent）：




```
// dbpersistent.h
class DbIstream;
class DbOstream;
struct DbPersistent {
    virtual ~DbPersistent();
    virtual DbIstream& restore(DbIstream& fromBuffer) = 0;
    virtual DbOstream& save(DbOstream& toBuffer) const = 0;
    virtual const char *className() const = 0;
};
```

现在客户（例如，Database）能够统一地对待所有的持久对象，而不必向下转换成特定的类型。

一个潜在的问题是：除了所要求的功能之外，不同的类型可能很少有共性。例如对于一个对象数据库来说，一些持久对象除了都允许它们被存储到磁盘之外，没有什么共同之处。

调和这些不同功能的一个办法就是让根基类包含定义在任意子类中的所有操作的联合。

这个办法会导致一个“胖接口”[1]。胖接口意味着操作可通过基类进行调用，而对一些继承的对象来说是无意义的。在这样的情况下，首先检查特定的对象是否支持该操作，或者测试方法的返回状态以检查它是否成功是必要的。在以上任何一种事件中，我们都不能够统一地对待所有的对象，并且有可能会因为疏忽而调用一个不支持的操作。

除了失去统一性和编译时类型安全之外，胖接口方法意味着在一个子类中增加任何新的操作，都将会迫使基类、所有现存的子类以及它们各自所有的客户程序重新编译。

为每一个类型定义一个枚举值，也不是解决办法。这种技术，有时候称为“类型交换（switching on type）”，会迫使所有的客户知道每一个类型。增加一个新类型意味着修改某个低层次组件，该组件枚举了所有可用的类型，再次会强迫所有现存子类型和它们的客户程序进行重编译（见 6.2.9 节）。但是，为了使客户程序能操纵新类型，若在每一个 switch 中增加相应的 case，则每个客户程序都将被迫修改源代码。因此，类型转换使得扩展类型集合的代价非常昂贵。

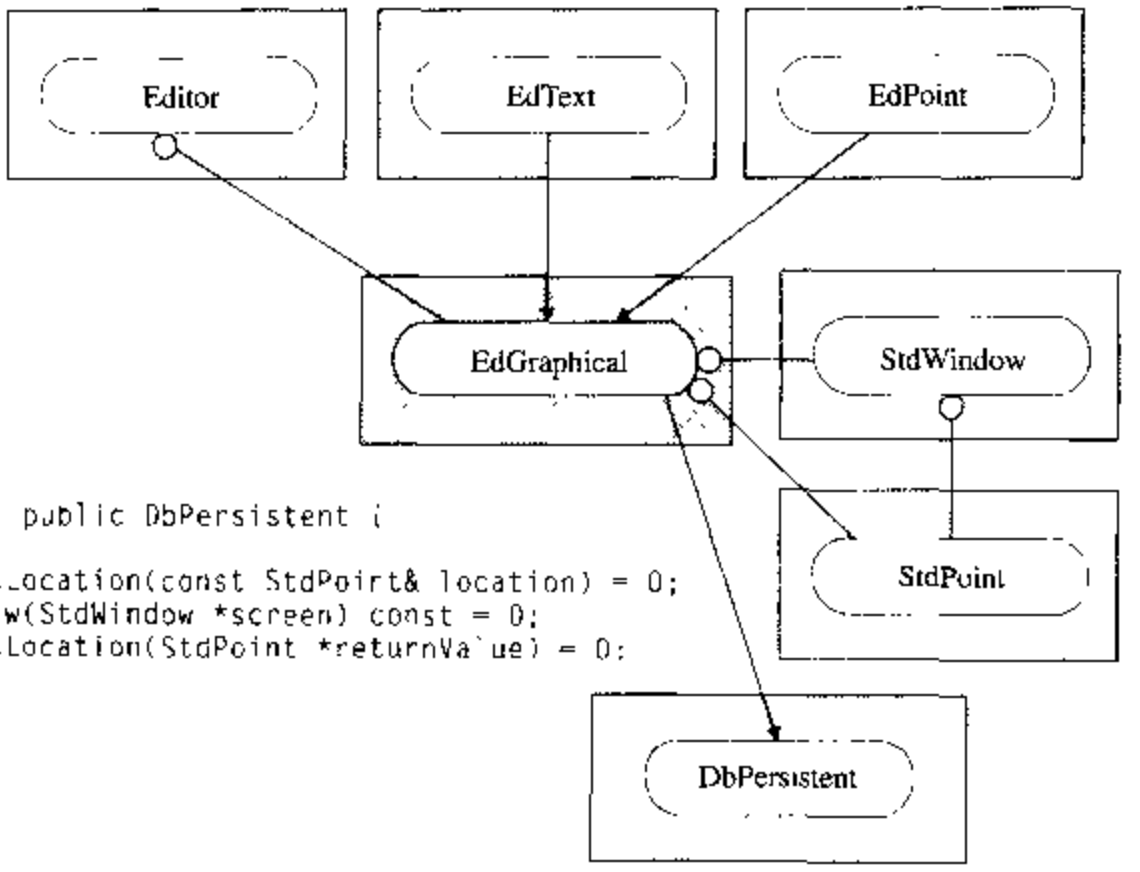
在一些类似的模式中，例如 Composite(163)，有时会提供一个函数——例如，const Derived *Base::derived() const ——来测试基类，看它是否是一个特定的派生类型的基类，是则返回一个指向那个类型的有效指针，否则返回 0。（如果可用，使用 dynamic_cast 结构更可取。）使一个基类型在它的接口中返回一个派生类型，将会在基类型和派生类型之间创建一个循环依赖。增加一个新的派生类型也会要求基类增加一个其它的测试成员函数，这将会迫使所有的其它派生类和它们的所有客户程序进行重新编译。这些新的类型不会被现存的客户自动处理，因此必须修改这些客户程序，以适应新类型。

我们使用一个纯粹抽象（协议）类型（它们的父母共享通用接口特性）的层次结构来组织我们的松散相关的具体类型，但是不包含任何实现。协议的层次结构充当一种决策树，它允许客户程序下沉到它们的应用程序所要求的操作特定层，而不必自始至终转换为一个特定的具体类型。然后，每个客户程序都将能够使用一个通用协议在一个相关对象的最大集合上操作，同时保持着高度的编译时类型安全。此外，我们还可以在不影响现有协议、具体对象或者客户程序的情况下增加新的协议。

扩展这个数据库的例子，考虑一个操作持久 graphical 对象的编辑子系统。每个 graphical 对象有一个位置，并且能够以统一的方式移动。每个 graphical 对象必须能够在一个给定的屏幕上给自己着色。因此，为 graphical 编辑对象引入如下的协议是有意义的：

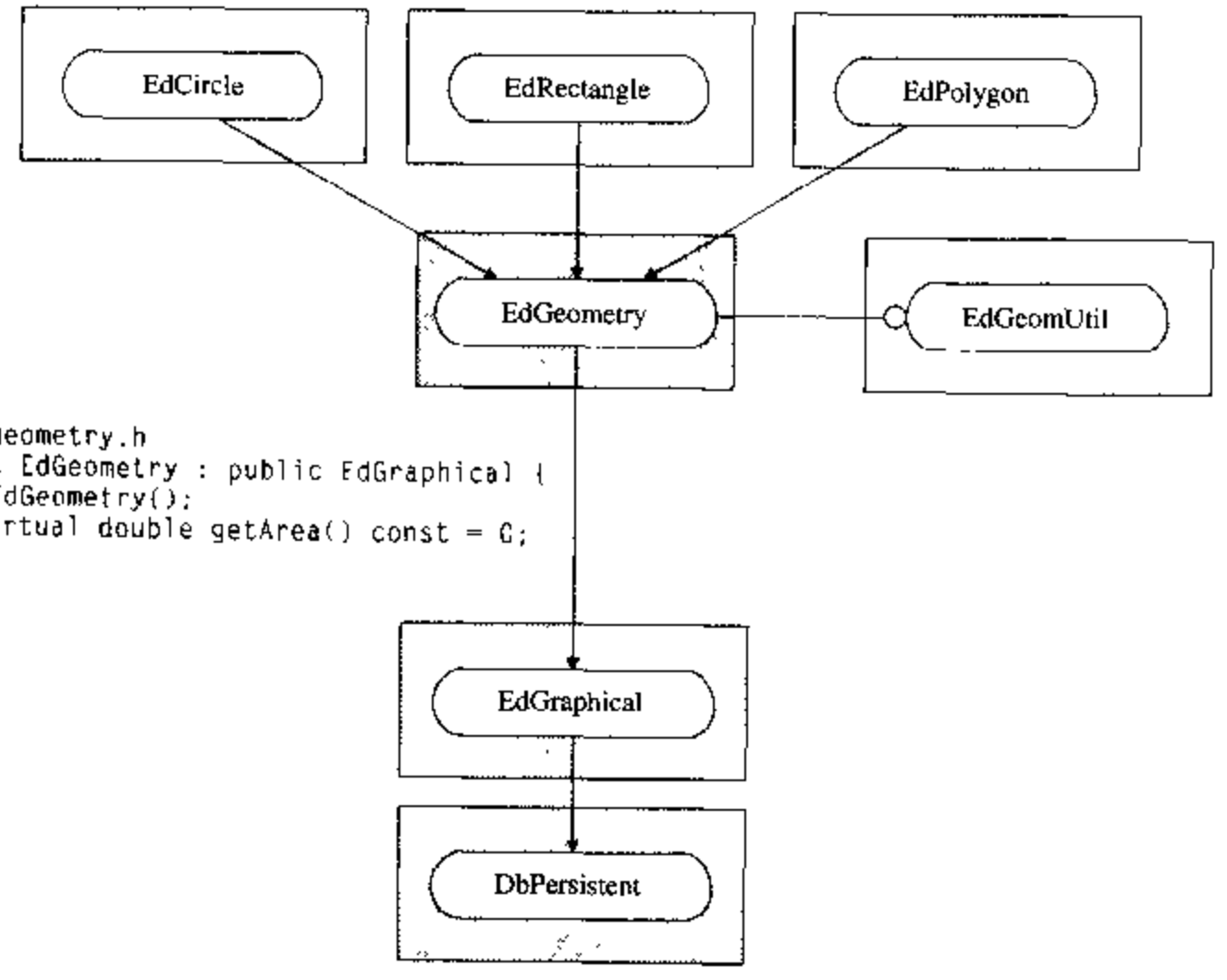
---

① **stroustrup**，13.6 节，452~454 页。




```
// edgraphical.h
class StdPoint;
class StdWindow;
struct EdGraphical : public DbPersistent {
    ~EdGraphical();
    virtual void setLocation(const StdPoint& location) = 0;
    virtual void draw(StdWindow *screen) const = 0;
    virtual void getLocation(StdPoint *returnValue) = 0;
};
```

某些操作对于 graphical 对象的一个大的子类（称为 geometric 对象——即，代表闭合形状的对象）是有意义的。例如，我们可以合理地要求计算一个多边形或圆形的面积，但不能请求文本。这些共性用如下协议来捕获：




```
// edgeometry.h
struct EdGeometry : public EdGraphical {
    ~EdGeometry();
    virtual double getArea() const = 0;
};
```

现在一个包含非基本几何工具函数（要求面积概念有意义）的 struct（例如，EdGeomUtil）接受类型为 EdGeometry 而不是 EdGraphical 的参数。

但是，有其他的方法可用于修改各种不同的几何对象，这是完全不同的事情。例如，我们能够改变一个圆的半径，但不能改变一个多边形的半径。我们可以把一个顶点加到一个多边形上，但不能加到一个圆形上。在这种情况下，我们可能被迫将所有的方法集中到一个具体类中，或使用一个更加灵活的、基于运行时的方法。但是，在实践中大部分应用程序可能使用一个更加通用的协议接口来有效地进行操作。

这些协议中的每一个都是一个不包含数据的纯粹抽象类，并且没有定义实质功能。换句话说，就是每一个协议只从它的父母处继承接口[1]。这个特性是模式的一个重要组成部分，因为它避免了将任何实现负担施加给客户或继承类作者。为了使协议层次结构模式能进行工作，从一个协议派生的具体对象的开发者必须保证，所有指定的行为都以与协议及其所有祖先的文档相一致的语义来实现。

适用性

在下列情况下使用协议层次结构：

- 类型的一个不同集合只从一个公共基类中继承其接口的小子集。
- 从一个根基类派生而来的中间抽象类型可以允许较宽范围的对象由客户统一处理。
- 把一个对象的特定子类作为一个特殊情况来处理的能力，能显著地改善整体性能[2]。
- 对象集合是可扩展的，并且可期望它们的增长能超过系统的生命期。我们必须能够在不影响现存的协议、具体类或客户程序的情况下扩展层次结构。
- 被每一个对象请求的基本操作是基本的和稳定的。如果操作集合被期望扩展，并且这些新的操作能够以具体对象的公共接口的形式来实现，那么考虑使用 Visitor(331)模式，或者增强协议层次结构。
- 现存的协议层次结构是相对稳定的。新的叶子类型的增加不会造成层次结构的一个实质性的重新分解；然而，额外的中间抽象协议类可以插入协议层次，代价是重新编译所有派生的协议和叶子类，以及它们各自的所有客户程序，但是不必重新编写现有代码。

结构

一个协议层次结构使用继承来创建中间协议，以使客户在可能的地方统一地处理松散相关的具体叶子类型。

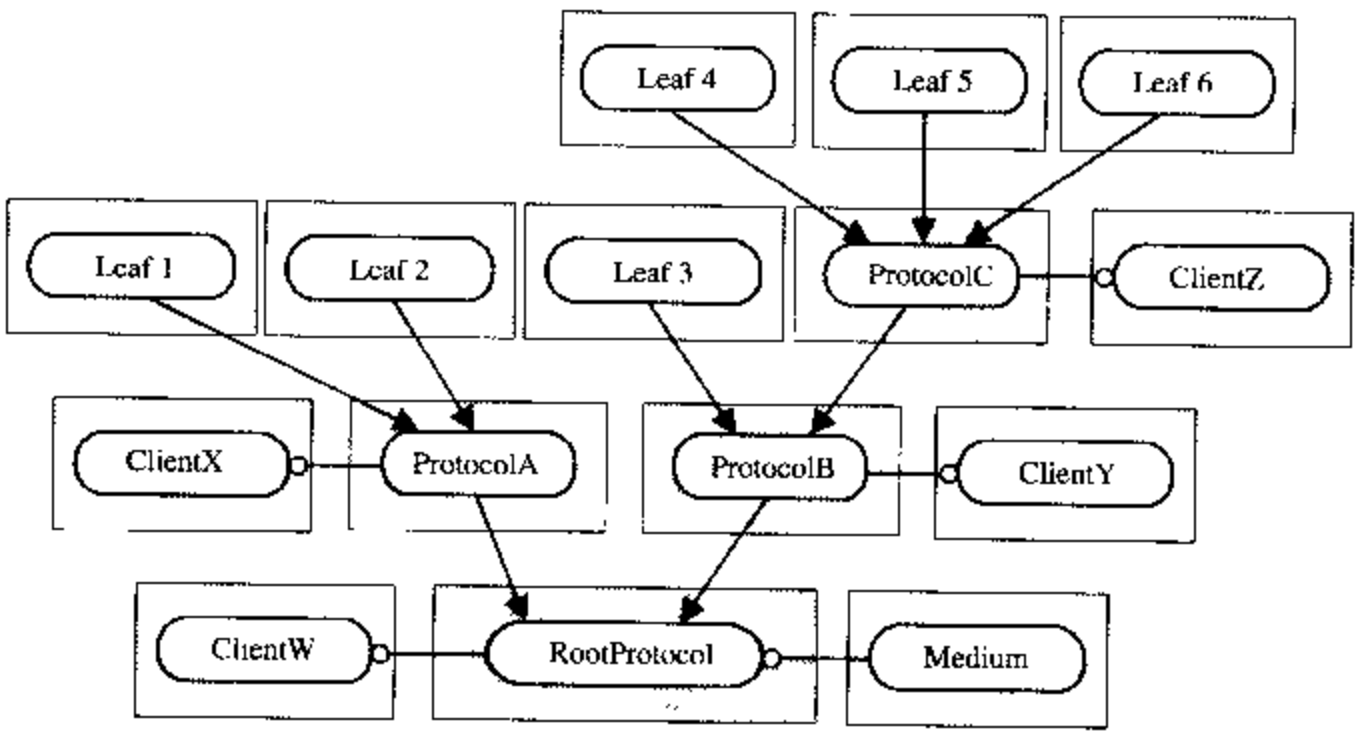
---

① 扩展协议层次结构使之拥有一个以上的根协议（见实现注释 5）是可能的。

② **stroustrup**，13.5 节，442~443 页。

参与者

- RootProtocol (DbPersistent)
  ——协议层次结构的根。
- Medium (DbIstream, StdWindow)
  ——协议接口中使用的一个类型。
- Protocol (EdGraphical , EdGeometry)
  ——定义一个通用的中间接口，以允许客户统一地使用派生的具体叶子类型。
- Leaf (MyEmployee , EdPoint, EdCircle)
  ——一个具体类型，它实现协议中定义的功能（它直接或间接地继承该协议）。
- Client (Editor, Database, GeomUtil)
  ——一个应用程序，使用适当的协议与对象的最通用集合进行交互，这些对象支持应用程序被要求的行为。



协作

- 客户程序统一地在所有通过协议接口从一个合适的协议派生的对象上进行操作。
- 通过尝试将对象 dynamic_cast 转换到期望的 Protocol 类型，一个客户程序能够判定一个类型为 Root Protocol 的对象是否适合它的要求。如果转换成功，那么对象是兼容的；否则，对象是不兼容的。
- 在没有实现这个语言特性的地方，每个（抽象）协议和（具体）叶子类可能需要提供一种操作来模拟一个 dynamic_cast（见实现注释 2）。
- 每个具体叶子类必须执行它从中直接或间接派生而来的协议指示的所有操作。

结论

使用协议层次结构模式有如下的好处和可靠性：

1. **避免胖接口**。对于那些可供选择的接口是根接口（该根接口是所有派生类型的接口的联合）的系统来说，当提升类型安全时，协议层次结构减少了编译时耦合。

2. **避免特殊的向下转换**。协议层次结构提供了一个中间级接口的组织良好的、可重用的集合，这些中间级接口避免了总是向下转换到一个特定的**具体**类型。

3. **是可层次化的**。协议层次结构可以避免使基类知道它的派生类的任何情况（见实现注释 2）。它也避免了创建一个闭合系统，在闭合系统中，只有那些能修改基类源代码的人才能够扩展系统。

4. **是自然绝缘的**。对具体叶子类的封装实现细节所做的改变，决不会导致在层次结构中只与抽象协议类型交互的客户程序被迫重新编译（见 6.4.1 节）。

5. **是模块化的和易于扩展的**。在一个设计良好的协议层次结构中增加新的具体对象，不会影响其它的组件（例如，具体类型、协议或者客户程序）。即刻移走一个现存协议下的新的具体类型，可以使客户程序解除协议（或任何其基类协议），以便于透明地操纵新类型的实例。

6. **能够改善效率**。如果度量证明系统中有一个瓶颈，并且一个叶子类或协议类十分普遍，那么为这些类型提供一个更为有效、有特定用途的算法可能是合理的。动态转换至少需要一个虚拟函数调用，因此有不小的开销。但是，当一个客户程序需要做几个函数调用时，将所有方式转换为具体叶子类型，通过使内联函数直接使用，有时可以显著地改善运行时性能（见 6.6.1 节）。

7. **需要继承层次结构**。这个方法要求所有参与进协议层次结构的对象通过继承（至少）一个根基类而相互关联（和 Visitor(331)模式相对，那个模式不要求这些对象相互关联）。另外，也要注意，在协议层次中声明的固定方法集对应于由具体叶子对象中的虚拟函数定义的基本行为。（与之相比，Visitor 模式的可扩展操作必须根据这些对象的公共接口来实现。）

8. **需要一些全局知识**。想要提出一个有用的协议层次结构，有必要知道哪些应用程序会使用中间级协议以及将参与进层次结构的对象种类。

9. **重构的代价昂贵**。在现存的层次结构中增加中间级协议，将要求重编译所有从新的协议派生（直接或间接）而来的类型，以及它们的所有客户程序。但是与其它技术（例如，类型交换）相比，客户程序不会被迫重构它们的代码，以利用那些被引入层次结构的新的对象类型。另一方面，从层次结构中删除一个协议可能导致对现存客户程序库的严重后果。一般来说，删除一个在大系统中已经建立的协议是不可行的。

实现

在实现协议层次结构模式时需要考虑如下问题：

1. **协议分解的是接口，而不是实现**。接口越通用，下层数据结构就越复杂，以便能表示任何实例。这是一个严酷的现实。例如，表示一个任意的多边形会要求无限大的空间。相反，

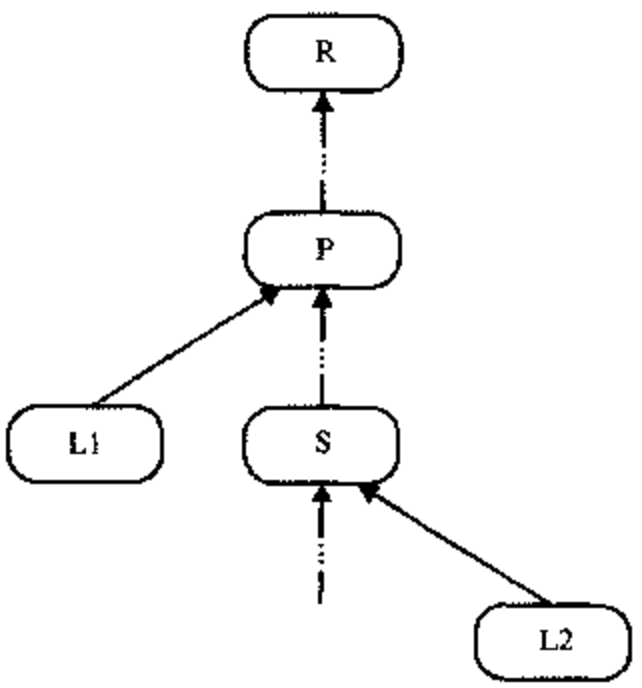
一个矩形（也是一种多边形，有着垂直/水平的角）可以一直只用两个点来表示（例如，d_lowerLeftCorner 以及 d_upperRightCorner）。为了避免负担更多带有不必要通用实现的特定具体对象，强制规定：协议不定义任何数据成员或是任何行为（在实现注释 2 中需要的例外）。

有时候我们会碰到一个所有派生类都支持的协议属性（例如，颜色）。我们可以定义一个并行（实现）层次结构来包含这些属性——由 Bridge(151)模式建议，而不是将这些属性定义为协议的一个数据成员。在这样做的时候，我们能够保持接口与实现的分离，达到完全的绝缘，并且仍然为实现复杂具体类型提供支持。

2. **动态转换**。C++语言草案标准现在以内置 dynamic_cast 和 typeid 操作的形式支持运行时类型信息（RTTI，Run-Time. Type. Information）[①]。当 C++的 dynamic_cast 特性不可用时，必须提供额外的语言支持，以便安全地转换成在协议层次结构中更特定的协议和具体类型[②]。一个模拟 dyanmic_cast 的有效方法——最初称为“安全转换”——要求带有根协议 R 的协议层次结构中的每个类 T（抽象的或具体的）定义一对静态函数[③]：

```
class T {
    // ...
  public:
    static T *T::dynamicCast(R& object);
    static const T *T::dynamicCast(const R& object);
    // ...
};
```

仅当一个实际对象是 T 类型的或以 T 作为（直接或间接）基类时，每个函数返回一个指向一个类型 T 的对象的指针；否则返回一个 0 值。层次结构中的每一个 dynamicCast 函数的参数总是根类型 R 的，以便最大化那些可以安全地重新解释的对象类型的数量。



① **stroustrup**，14.2 节，306~336 页。

② **stroustrup**，13.5 节，442~451 页。

③ 这个函数的两个版本都需要，这是为了保持 const 正确性（见 9.1.6 节）

当一个客户希望确定一个派生自协议 P 的特定实例是否也派生自一个子协议 S 时，对应于协议 S 的 dynamicCast 功能可以按如下方式使用：

```
void client(const P& p)
{
    const S *s;
    if (s = S::dynamicCast(p)) {  // assignment (=) is intentional
        // p is a kind of S; treat specially using valid s.
        s->functionDefinedOnlyForTypeS();
    }
    else {
        // p is not an S; treat generically using p.
        p.functionDefinedForTypeP();
    }
}
```

实现 dynamicCast 功能需要一个虚拟函数 hasProtocol 的支持，它在根协议 R 中被声明为 public。hasProtocol 的用途是确定这个具体对象是否是一个从特定类派生而来的叶子类型的实例（或者类本身的一个实例）。因为一些 C++编译器没有实现 typeid 操作[1]，所以在运行时可能没有内置方式可以识别一个对象的类型。为了弥补这个不足，我们使用了下列技巧[2]。我们可以把每个类 T 和该类的一个惟一静态成员的地址联系起来。

```
// t.h
class T {
    static const void *const s_typeId;    // external linkage
    // ...
};

// t.c
const void *const T::s_typeId = &s_typeId;// external linkage
// ...
```

但是为了改善绝缘和避免外部符号，我们可以通过移动 s_typeId 到.c 文件中，作为其文件作用域中一个静态（例如内部连接）变量，获得同样的效果（假设每个组件有一个层次结构类型）[3]：

```
// t.c
const void *const s_typeId = &s_typeId;    // internal linkage
```

如果它的参数等于对象本身的 s_typeId，或者等于该对象的某个基类的 s_typeId，那么虚

---

① **stroustrup**，14.2.5 节，316~317 页。

② **ellis**，10.2 节，212~213 页。

③ 注意一个在文件作用域中声明了 const 的指针——例如，void *const p；——有内部连接。也要注意类型 const void *的一个指针能够拥有任意（非成员）对象的地址，包括任意指针对象的地址值（例如，const void **）。

拟函数 hasProtocol(const void *)将返回 1。每个派生类型 T 的 hasProtocol 函数的定义（通常）按如下方式实现[1]：

```
int T::hasProtocol(const void *typeId) const
{
    return typeId == s_typeId
        || DirectBase1::hasProtocol(typeId)
        || DirectBase2::hasProtocol(typeId) // For multiple roots
        // ...                              // see implementation
        || DirectBaseN::hasProtocol(typeId) // note 5.
    ;
}
```

在上面的例子中，T 必须直接从协议 DirectBase1 派生而来。在存在多个根的情况下（见实现注释 5），一个类可能从多于一个协议派生而来，因此必须检查它的每个（多重继承）直接基类，它们被标识为 DirectBase2、…、DirectBase*N*。

一个任意派生类型 T 的 DynamicCast 静态成员函数，现在可以如下方式实现：

```
T *T::dynamicCast(R& object)
{
    return object.hasProtocol(s_typeId) ? (T *) &object : 0;
}
const T *T::dynamicCast(const R& object)
{
    return object.hasProtocol(s_typeId) ? (T *) &object : 0;
}
```

因为只有层次结构 R 的根由一个特定的 dynamicCast 函数使用，所以只有 R::hasProtocol 需要是公共的。在派生类中的所有 hasProtocol 函数都可能声明为保护的，以帮助加强正确的使用。

3. **用一个静态 cast 来优化**。有时候一个客户有足够的全局上下文可以知道一个 dynamicCast（或者 dynamic_cast）将获得成功。在这样的情况下，我们有时可以通过借助一个非安全类型的转换（或者 static_cast）来显著改进性能[2]。这样的优化本质上容易出错。一个合理的折衷是，按如下形式组合两个 cast 形式和一个断言语句：

```
void knowledgeableClient(const R& object)
{
    const T *p;
    assert(p == T::dynamicCast(object));
    p = (T *) &object;
```

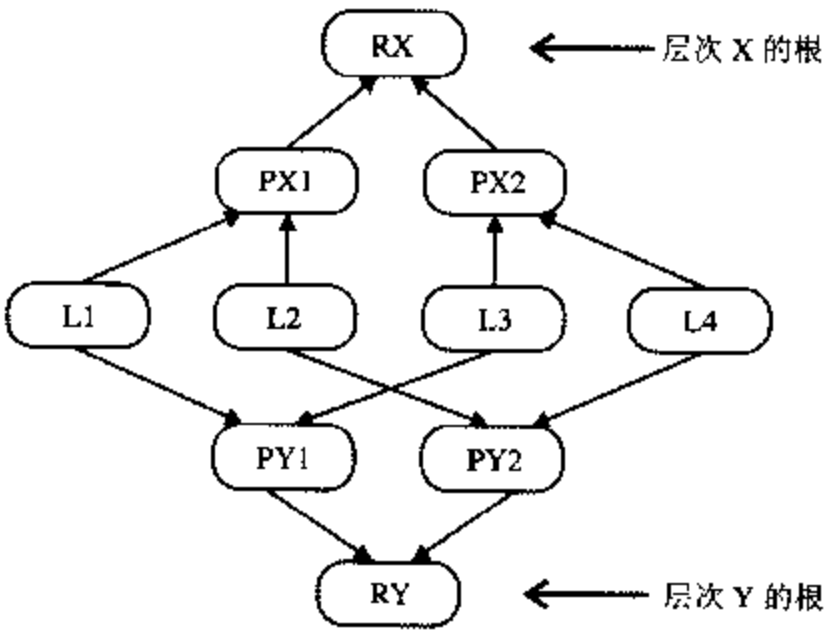
---

① 在一些实现中，hasProtocol 函数接受一个从惟一地址构建的用户自定义类型（例如，TypeId），这是为了预防传递一个任意地址给该函数。

② 在继承层次结构中类指针上的 Dynamic_cast 和 static_cast 的使用在 **stroustrup94** 中讨论，14.3.2.1 节，330 页。

```
        // use valid pointer of type (const T *)
        // ...
    }
```

在开发期间，编程错误会被检测并立即报告。可以很容易地从产品代码中删除冗余的错误检查，以消除这种不必要的性能负担。



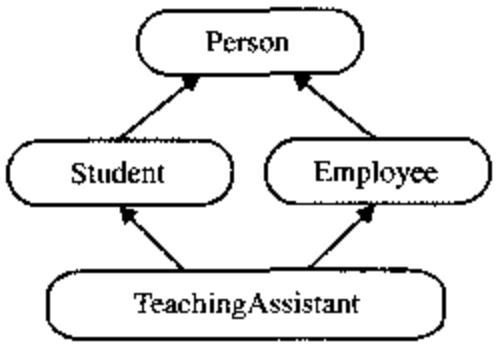
4. **实现协议析构器**。虽然一个协议没有它自己的实例数据，但是它的（空）析构器应该声明为 virtual，并定义为非内联，这样可以避免增加不需要的析构函数和协议类虚拟表的静态拷贝，如注释 2 第 2 点（见 9.3.3 节）。如果 dynamic_cast 中所描述的那样被模拟，那么 hasProtocol 函数可以设定这个角色，这样就不必定义协议类的析构器了。

5. **多根协议**。一个协议拥有多个父协议在技术上是可能的；一个协议层次结构拥有多个根在技术上也是可能的。

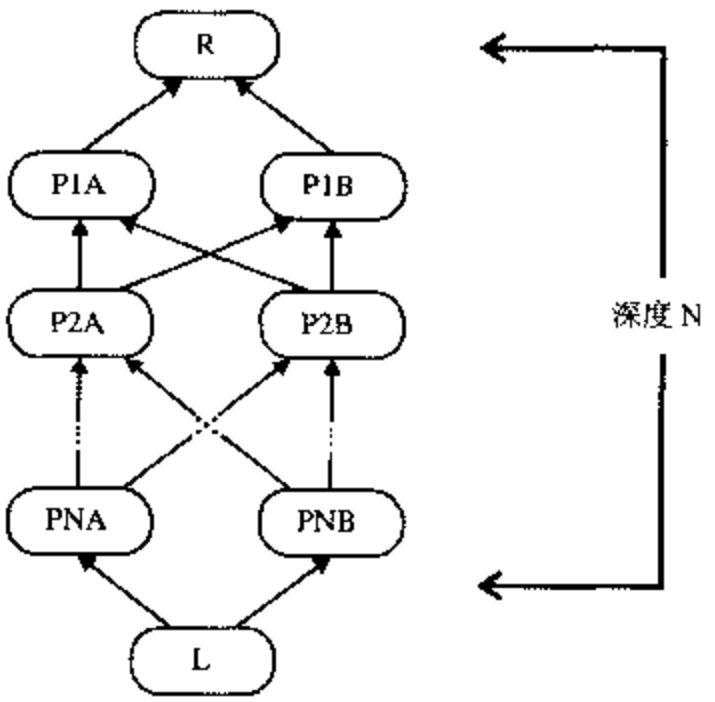
在上面的例子中，L1~L4 表示了四个具体叶子类。协议 PX1（或 PX2）派生自根协议 RX，可用来统一地处理 L1 和 L2（或 L3 和 L4）。由于提供了第二个根 RY，使我们可以派生两个新的协议（PY1 和 PY2），这些协议允许我们统一地处理 L1 和 L3（或 L2 和 L4）。

由于下列多种原因，在一个协议层次结构中多个根的个数受到限制：

- 每个类要求每个根有两个 dynamicCast 函数：一个用于 const 对象，一个用于 non-const 对象。
- 每个新的根增加一个额外的虚拟表指针指向每个具体的对象实例。
- 在一个多根协议层次结构中寻找一个具体叶子类的正确位置，比在只有一个根的协议层次结构中寻找要更困难。
- 使用协议来获得所有的任意组合，将要求一个指数级数量的根。例如，可以基于电路元素是否属于图形的、电的或复合的进行分类。如果这三个属性是独立的，那么为每个组合提供协议将需要八个根。我们可以在运行时将这些属性聚集起来放进目录，而不是在编译时将这些属性组织起来放进一棵决策树。我们将使用一个函数（例如 hasCategory）（而不是一个 hasProtocol 函数）来确定一个给定的对象是否适合于一个给定的应用程序。这个更动态的方法解决了一个问题，这个问题与通过 Decorator(175) 模式解决的问题相类似。
- 虚拟基类的使用将允许一个协议以多种方式从一个根协议中派生。



在上面的例子中，TeachingAssistant 从 Person 协议派生而来，沿着两条不同的路径：Student 和 Employee。C++语言不允许传统的转换（cast）在存在虚拟基类的情况下起作用（虽然当派生类可以被无二义性地确定时，dynamic_cast 将会起作用）[①]。由于继承层次结构再次集中到一个节点，进行一个 dynamicCast（如上文第 2 点那样实现）的运行时开销将潜在地变成协议层次结构深度的指数：



在上面的协议层次结构中，有 N 个层次将叶子类 L 与根协议 R 分开。假设 cast 最终失败，一个搜索特定协议的简单深度优先搜索将遍历从 L 到 R 的 2^N 条不同路径。

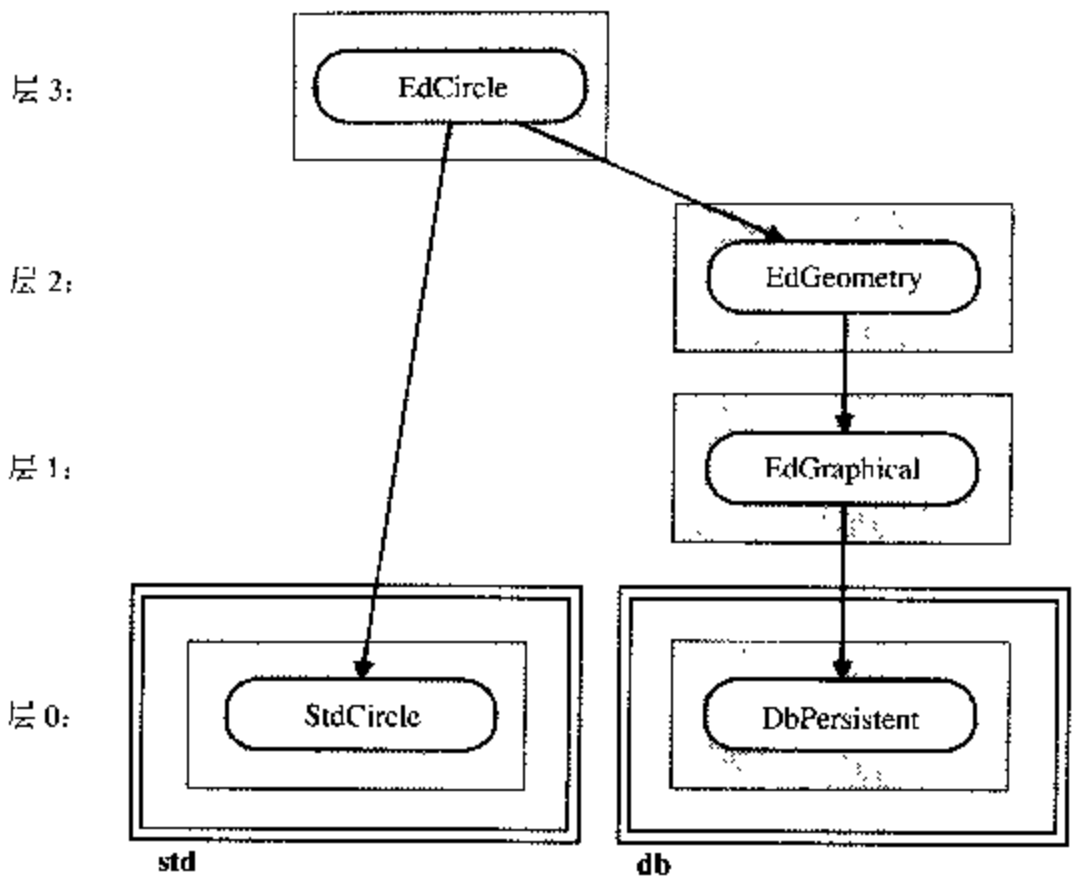
6. **多重继承**。多重继承的一个有效使用是将一个定义了非虚拟功能的现存（低层）类融进一个协议层次结构。在这种方法中，我们通过继承现存具体类和适当的抽象协议来派生一个新的具体叶子类。然后我们使用具体基类为了实现协议的虚拟函数而提供的功能。以这种方式使用多重继承是 Adapter(139)模式的类形式的一个实例[②]。

例如，通过从抽象 EdGeometry 协议和一个具体 StdCircle 类型（这些类型没有虚函数，或许有许多内联函数，由一个低层标准库包，例如 std）中继承，我们可以选择实现叶子类型

① **stroustrup94**，14.2.2.3 节，312~313 页。

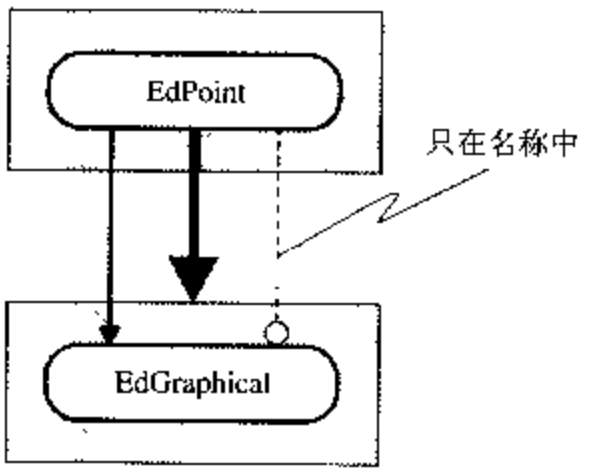
② 使用多重继承的对象 I/O 的简单例子可以在 **stroustrup94** 中找到，14.2.7 节，322 页。

EdCircle。

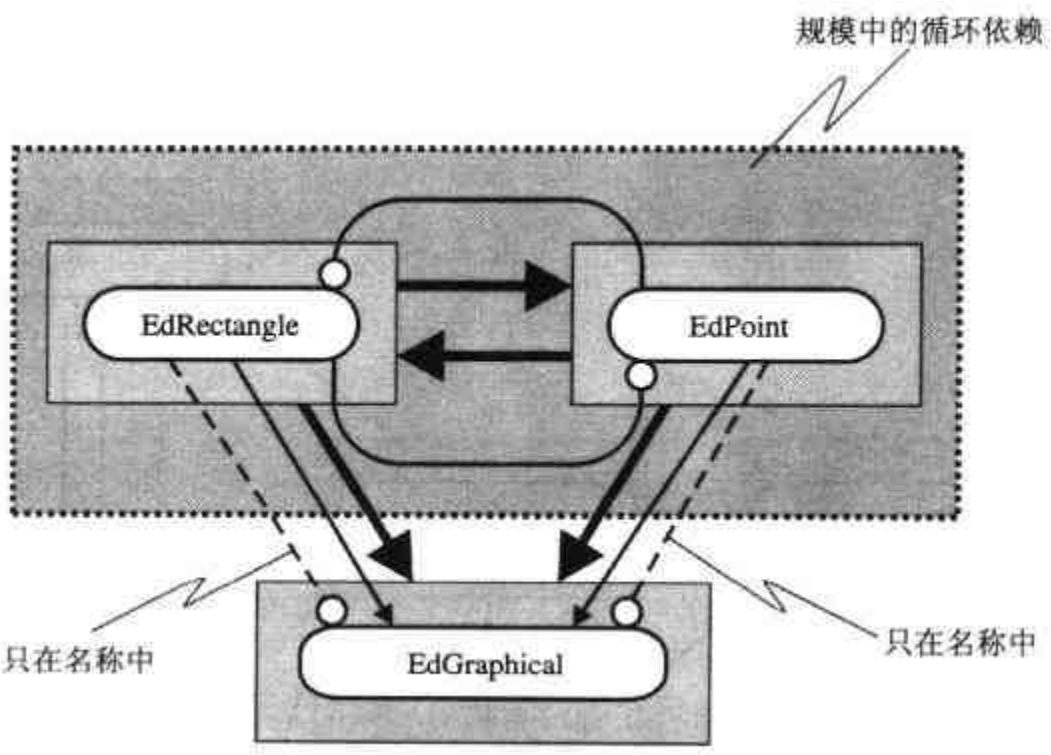


类 EdCircle 只需要实现定义在协议层次结构中的虚函数，通常根据在低层 StdCircle 类中提供的函数（或许有不同的名称）来实现。

7. **中间媒介**。作为一个规则，不使用协议层次结构中一个类型作为一个中间媒介（即，不在相同层次结构中的其它类型的接口中使用）是一个明智的选择。例如，考虑 EdGraphical 的接口。为了得到/设定一个图形对象的位置，我们需要提供某种指针对象。EdPoint 是一种 EdGraphical。如果我们为了这个目的要提供一个 EdPoint，那么我们会在 EdPoint 和 EdGraphical 之间创建一个循环依赖。

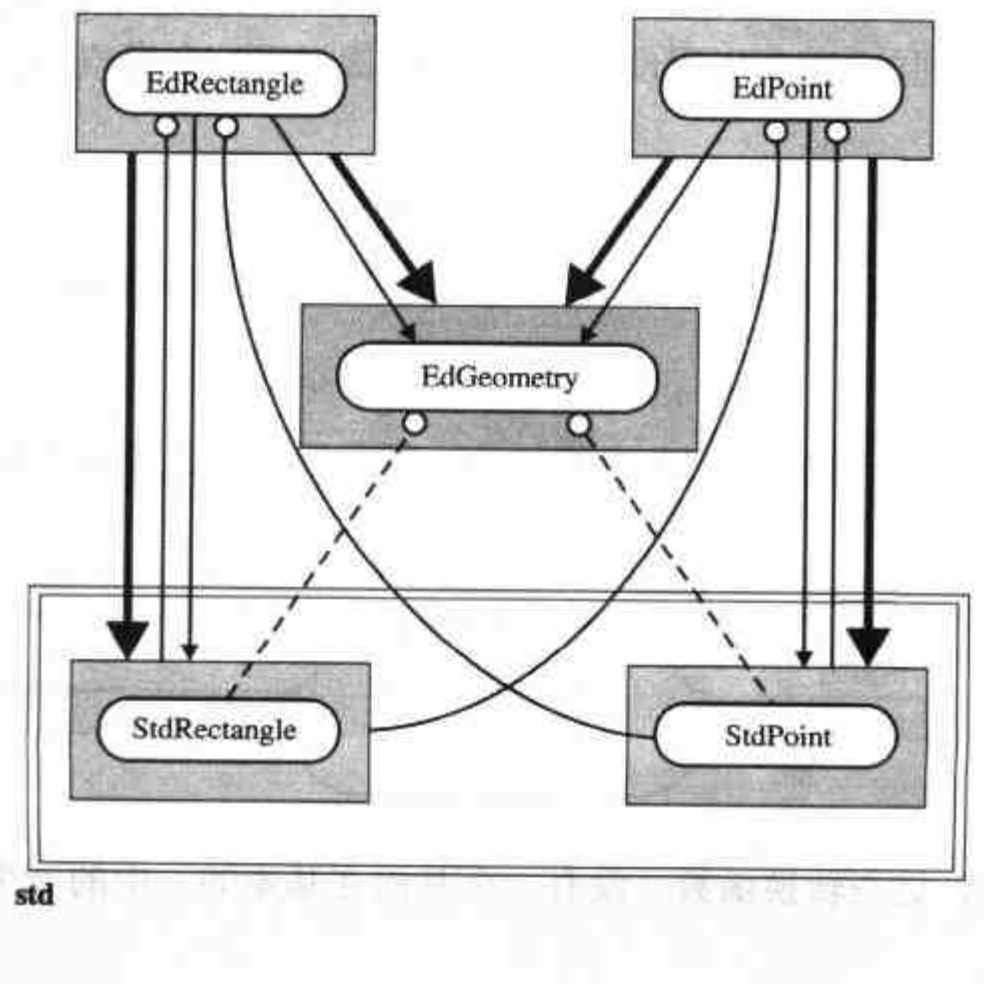


因为 EdGraphical 只是一个协议，所以它对 EdPoint 的依赖仅仅是名称上的。但是，考虑当两个或更多具体叶子类型出现在 EdGraphical 协议中时会发生什么。假设 EdGraphical 也提供功能来返回图形对象的边界框（作为一个 EdRectangle）。

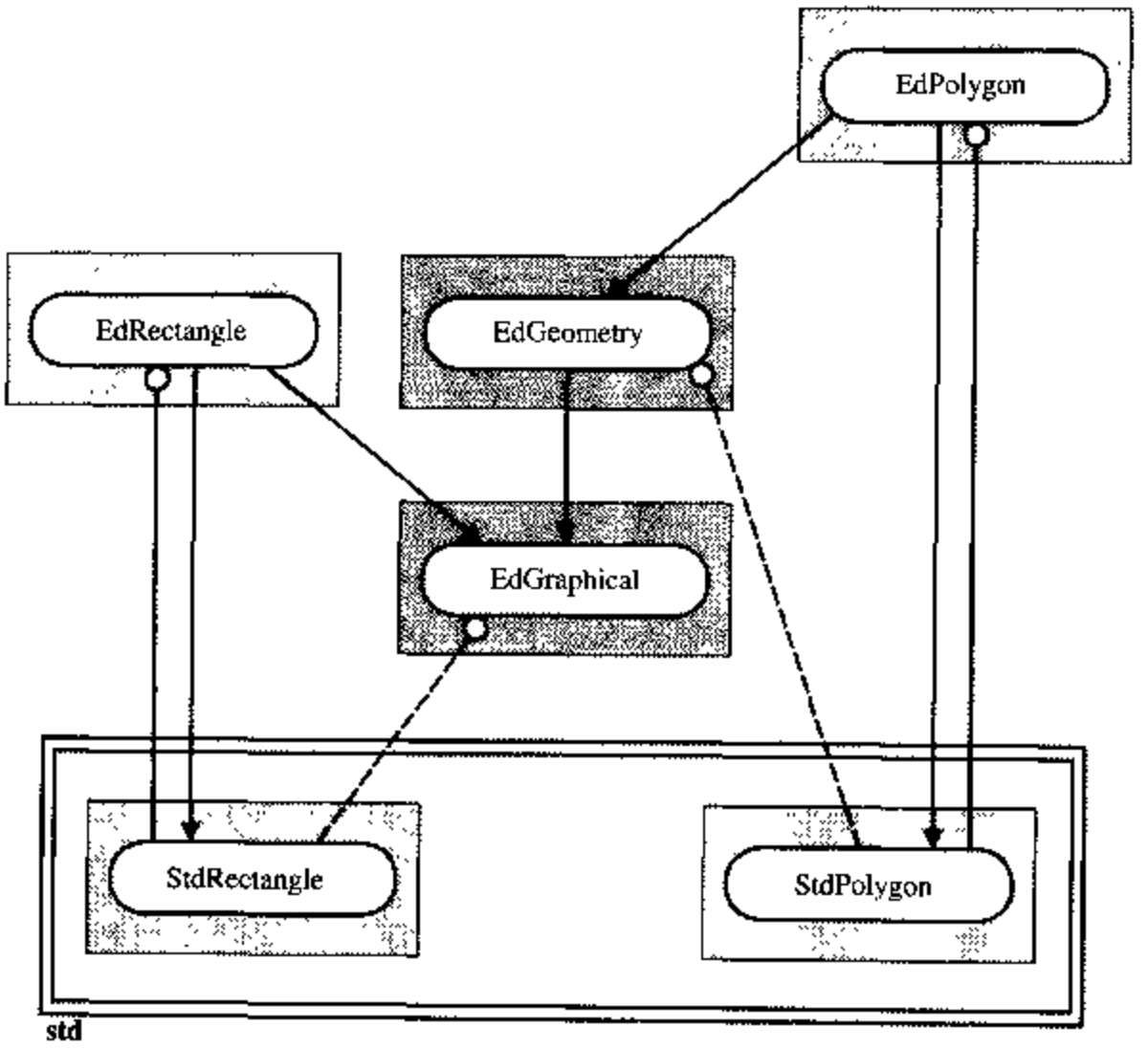


因为 EdRectangle 和 EdPoint 都需要实现在 EdGraphical 协议中声明的所有功能，所以每一个都必须实质地使用协议中命名的类型。这样的使用会导致循环物理依赖，而且这些循环物理依赖会随着新叶子类型在协议中命名而变得更大。

通过从在较低层库实现的具体类型和适当的协议（如实现注释 6 所建议的）中继承来实现一个叶子类，可以解决层次化的问题。低层具体类型被用来在协议层次结构的成员之间传递复杂信息。公共继承自动允许像使用低层库类型那样使用协议层次结构中的每个具体类型。提供一个构建器来从一个低层库类型中创建一个具体协议类型，可以允许协议和非协议类型进行无缝地交互。



有时，一个具体叶子类型的协议所要求的功能不能直接以相应低层类型的形式来实现。因为这样一个实现将破坏低层包的内部层次化。

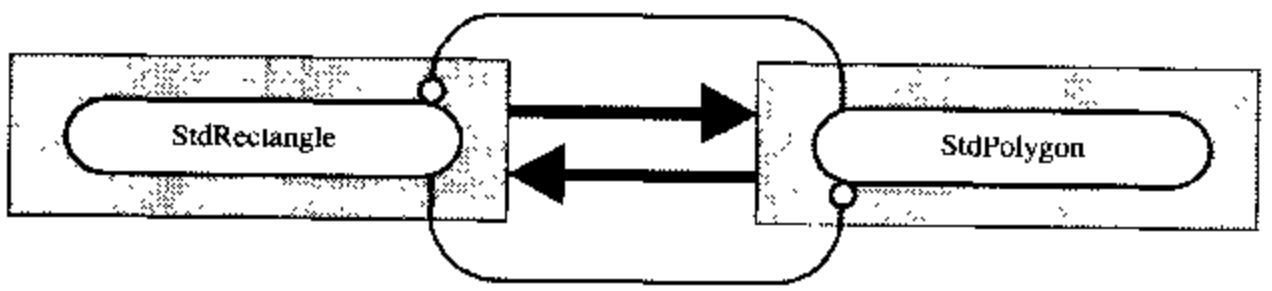


再次，假设上图中的 EdGraphical 提供了能返回图形对象的边界框的功能，但是这次是作为一个 StdRectangle 返回的。进一步假设 EdGeometry 提供了一个模拟给定的 StdPolygon 图形的功能：

```
class StdRectangle {
    // ...
    void cvtToPolygon(StdPolygon *);
    // ...
};
```

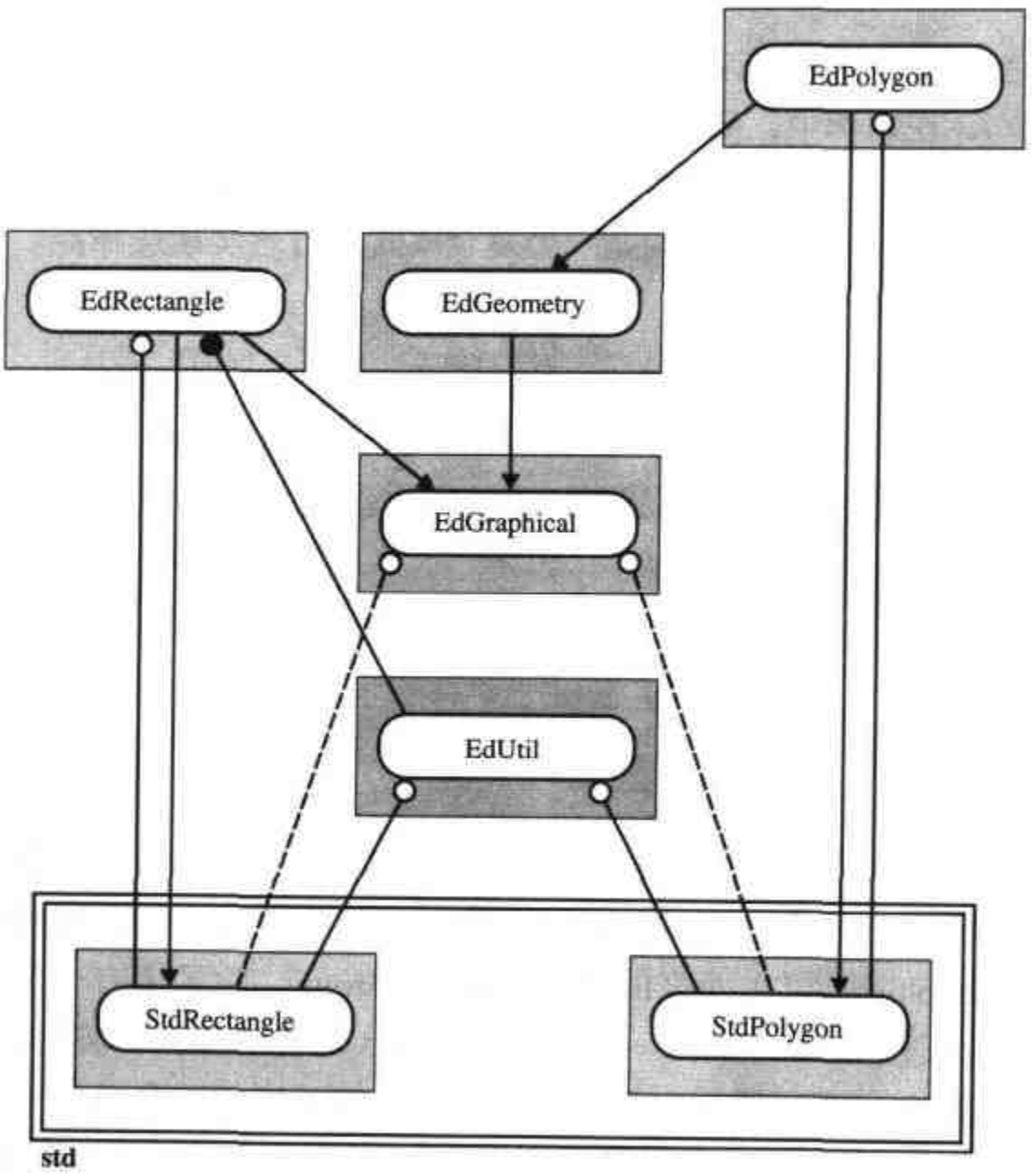
```
class StdPolygon {
    // ...
    void getBbox(StdRectangle*);
    // ...
};
```

若允许这些转换函数驻留在它们各自的类中，则会导致下面的循环依赖：



为了达到层次化，这些转换函数（没有一个是内在基本的）中的一个或两个必须升级到

std 包的物理协议层次结构中一个更高层次（见 5.2 节）：



在上面修改后的系统中，cvtToPolygon 和 getBox 现在都作为 EdUtil 工具类的静态成员驻留，并且操作只使用 StdRectangle 和 StdPolygon 的公共接口。

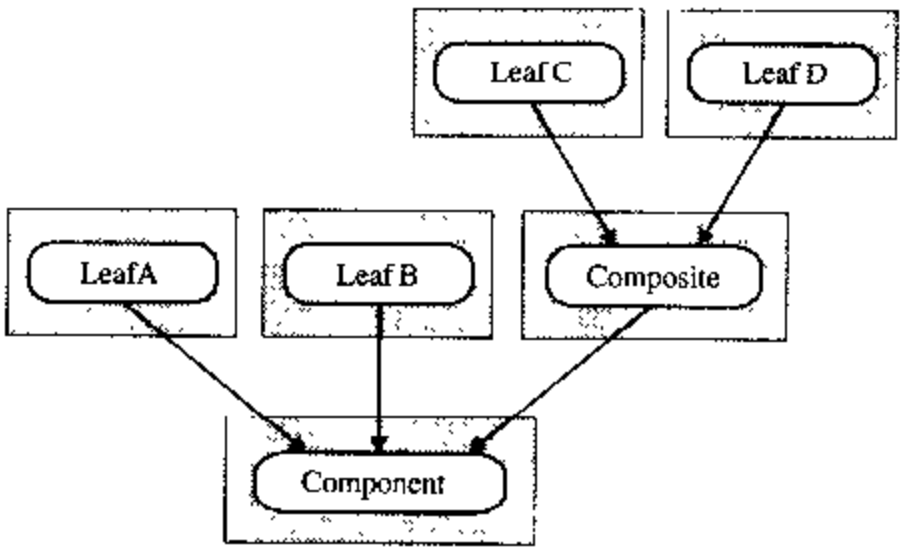
8. **持久性**。对于**持久**性对象来说，谁应该定义编码和解码操作呢？例如，如果 EdPoint 实现了 encode 和 decode，那么 EdPoint 将需要对实际数据成员进行私有访问。如果我们使用多重继承技术（在实现注释 6 中建议的），那么 StdPoint 将被迫授予 EdPoint 远距离友元关系，这是不受欢迎的（见 3.6 节）。允许这种远距离友元关系甚至会要求修改 StdPoint 的源代码。替代的方法是，用低层库对象封装基本功能的实现（这些基本功能可将对象的状态信息转换为一个字节流，或将一个字节流转换为对象的状态信息），这种做法将是有意义的。

持久类型能够跨越多个平台共享。我们希望这些持久类型使用同样大小的整数和浮点数，而不考虑平台。如 10.1.3 节中建议的那样，我们可以定义一个非常低层的包括标识一个特定大小的整数类型和浮点数类型（例如，Int16、Int32、Float64）的 typedef 声明的特定平台类。把我们的系统移植到一个新平台，要求将这些 typedef 声明重新映射到相应的基本数据类型

（例如，short、int、double），但避免了对每个持久对象类中的这些基本类型定义进行条件编译。

9. **包前缀**。在实践中，在协议层次结构中用通用前缀标识类（不考虑名称空间问题）是方便的（见 7.2 节）。在本例中，所有的编辑器协议和具体类型都以 Ed 开头。这些前缀有助于刻画与参与了某种基础应用的类型相关的共性，并使这种共性更加明显。

10. **复合对象**。一个协议层次结构是一个复合模式的自然实现选择：



上例中，LeafA 和 Leaf B 都是具体类，它们只实现了每个组件所要求的基本功能。Composite 协议提供了特定于复合组件的额外功能。例如，我们可以创建图形对象的一个复合，称它为从协议 EdGraphical 派生而来的 EdGraphicalGroup。除了所有一般的图形操作外，EdGraphicalGroup 还支持像 add、remove、numMembers 这样的操作。被包含的组件可能实现为图形对象的动态分配（深度）拷贝，这需要 Graphical 协议中的一个 clone 方法[1]。动态分配、多态对象的安全操作应该使用一个句柄类来完成（见 6.5.3 节，9.1.6 节中图 9-15 显示的例子，以及 9.1.8 节）。

代码示例

当一个松散相关的对象类型的集合可以通过一个共同的基类型得到，但若没有单个子类型的更特定的信息，就不能处理这些对象类型时，协议层次结构是重要的。数据库应用是一种典型的情况：根类型要求进一步的细化，以便对象在各种应用中都是有用的。

```
double getTotalArea()
{
    Database db("data.dat");  // open an existing database

    double totalArea = 0.0;   // Used to accumulate area of all
```

① **stroustrup**，6.7.1 节，218 页。

```
                                // geometric objects in the database.
    EdGeometry *p;              // *may* hold valid kind of EdGeometry

    for (DatabaseIter it(db); it; ++it) {

        DbPersistent& object = it(); // illustrates type of "it()"

        if (p = EdGeometry::dynamicCast(object)) {
            totalArea += p->getArea();
        }
        else {
            cout << "object was not geometric" << endl;
        }
    }

    return totalArea;

}
```

上例中，我们创建了一个 Database 对象 db，并将其与存储在磁盘文件 data.dat 中的信息相关联。我们创建了一个数据库迭代器，它在数据库的所有对象上进行迭代。迭代器所返回的类型是协议层次结构的根 DbPersistent，它不支持 getArea()操作。只有那些从 EdGeometry 中派生而来的持久对象才支持这个功能。使用 dynamic_cast（或者 EdGeometry 的 dynamicCast），我们能够探测出一个给定的持久对象是否也是一个几何对象。如果是，我们计算它的面积；如果不是，我们对那个结果打印出一个信息。通过把 dynamicCast 放在 if 语句中，我们可以确定我们不会执行该 if 程序体，除非从 DbPersistent 中派生而来的具体对象也是从 EdGeometry 中派生而来的。

使用多重继承来创建协议层次结构中的一个叶子类，这是从基础可重用数据类型中去除一个特定层次结构中的耦合的有效方法。例如，考虑标准圆类型的接口：

```
// stdcircle.h
#ifndef INCLUDED_STDCIRCLE
#define INCLUDED_STDCIRCLE

#ifndef INCLUDED_STDPOINT
#include "stdpoint.h"
#endif

class ostream;
class DbOstream;
class DbIstream;

class StdCircle {
    StdPoint d_center;
    int d_radius;

  public:
    // CREATORS
    StdCircle();
    StdCircle(const StdPoint& center, int radius);
```

```
    StdCircle(const StdCircle& circle);
    ~StdCircle();

    // MANIPULATORS
    StdCircle& operator=(const StdCircle& circle);
    DbIstream& decode(DbIstream& fromBuffer);
    void setCenter(const StdPoint& location);
    void setRadius(int length);

    // ACCESSORS
    DbOstream& encode(DbOstream& toBuffer) const;
    double getArea() const;
    const StdPoint& getCenter() const;
    int getRadius() const;
};

ostream& operator<<(ostream& o, const StdCircle& circle);
inline StdCircle::StdCircle() {}
inline StdCircle::~StdCircle() {}
// ...
inline
DbOstream& operator<<(DbOstream& toBuffer, const StdCircle& circle)
{
    return circle.encode(toBuffer);
        // Special-purpose primitive functionality implemented
        // directly in the low-level type in order to support the
        // DbPersistent protocol while preserving data hiding.
}

inline
DbIstream& operator>>(DbIstream& fromBuffer, StdCircle& circle)
{
    return circle.decode(fromBuffer);  // See above comment.
}

#endif


// stdcircle.c
#include "stdcircle.h"
#include "dbstream.h"  // object I/O capability
#include <iostream.h>
#include <math.h>      // define M_PI

// ...

DbIstream& StdCircle::decode(DbIstream& fromBuffer)
{
    return fromBuffer >> d_center >> d_radius;
}

DbOstream& StdCircle::encode(DbOstream& toBuffer) const
{
    return toBuffer << d_center << d_radius;
}

double StdCircle::getArea() const
{
```

```
    return M_PI * d_radius * d_radius;
}
```

但是，持久性提出了一个重要的特殊情况。为了有效地将一个对象保存和恢复到磁盘中，需要访问该对象的内部状态变量。为了避免远距离友元关系，有必要在低层类型本身直接实现基本的 encode 和 decode 功能，这是必需的。与此相对，在由每个基本类型所提供的公共功能的顶部为所有的图形类型实现通用 draw 功能是可能的[①]：

```
// edcircle.h
#ifndef INCLUDED_EDCIRCLE
#define INCLUDED_EDCIRCLE

#ifndef INCLUDED_EDGEOMETRY
#include "edgeometry.h"
#endif

#ifndef INCLUDED_STDCIRCLE
#include "stdcircle.h"
#endif

struct EdCircle : EdGeometry, StdCircle {
    static const EdCircle *dynamicCast(const DbPersistent& object);
    static EdCircle *dynamicCast(DbPersistent& object);

    // CREATORS
    EdCircle();
    EdCircle(const StdPoint& center, int radius);
    EdCircle(const StdCircle& circle);
    EdCircle(const EdCircle& circle);
    ~EdCircle();

    // MANIPULATORS
    EdCircle& operator=(const EdCircle&);
    EdCircle& operator=(const StdCircle&);
    void setLocation(const StdPoint& location);
    DbIstream& restore(DbIstream& fromBuffer);

    // ACCESSORS
    const char *className() const;
    void draw(StdWindow *screen) const;
    double getArea() const;
    void getLocation(StdPoint *returnValue) const;
    DbOstream& save(DbOstream& toBuffer) const;

protected:
    int hasProtocol(const void *protocol) const;
```

① 非基本操作，例如 draw，能够以低层基础类型的公共接口的形式来实现，这些低层基础类型是 Visitor(331)模式所支持的扩展功能的一种。

```
};

#endif


// edcircle.c
#include "edcircle.h"

const void *const s_typeId = &s_typeId; // local linkage


// STATICS

const EdCircle *EdCircle::dynamicCast(const DbPersistent &p)
{
    return p.hasProtocol(s_typeId) ? (EdCircle *) &p : 0;
}

EdCircle *EdCircle::dynamicCast(DbPersistent &p)
{
    return p.hasProtocol(s_typeId) ? (EdCircle *) &p : 0;
}


// CREATORS

EdCircle::EdCircle() {}

EdCircle::EdCircle(const StdPoint& center, int radius)
: StdCircle(center, radius)
{}

EdCircle::EdCircle(const StdCircle& circle)
: StdCircle(circle)
{}

EdCircle::EdCircle(const EdCircle& circle)
: StdCircle(circle)
{}

EdCircle::~EdCircle() {}

// MANIPULATORS

DbIstream& EdCircle::restore(DbIstream& fromBuffer)
{
    return StdCircle::decode(fromBuffer);
}

void EdCircle::setLocation(const StdPoint& location)
{
    StdCircle::setCenter(location);
}
```

```
// ACCESSORS

const char *EdCircle::className() const
{
    return "EdCircle";
}

void EdCircle::draw(StdWindow *) const
{
    // Implement draw capability using utilities
    // supplied by the editor package on top of
    // the public functionality provided by
    // the low-level, StdCircle Type.
}

double EdCircle::getArea() const
{
    return SLdCircle::getArea();
}

void EdCircle::getLocation(StdPoint *returnValue) const
{
    *returnValue = getCenter();
}

DbOstream& EdCircle::save(DbOstream& toBuffer) const
{
    return StdCircle::encode(toBuffer);
}


// PROTECTED

int EdCircle::hasProtocol(const void *typeId) const
{
    return typeId == s_typeId || EdGeometry::hasProtocol(typeId);
}
```

已知的使用

一些图形和数据库应用的 Schema 部分使用了协议层次结构。C 语言采用 dynamic_cast 运算符来支持这种使用[①]。在由 Mentor Graphic 公司 IC 部提供的 ICGen CAD 框架产品中，采用协议层次结构（circa 1992）来实现持久基本类型的一个固定集合[②]。这个基本类型的固定集合形成了描述 IC 元素的一个扩展集合的基础。这些元素参与了它们自己的层次结构。例如，

① **stroustrup94**，14.2.1 节，307~308 页。

② **soukup**，第 9 章，案例 4，397~402 页。

ElemGraphical、ElemGeometry 和 ElemSingleGeom 是一个元素层次结构中的系列协议。ElemSingleGeom 表示一个在集成电路的单个层次上的、只有单个基本几何图形的部件（例如，ElemPolygon），而 ElemGeom 允许在多个层次上有若干个几何图形（例如，ElemWire）。

实现持久性的这个方法的一个简化版本在哥伦比亚大学的研究生计算机科学课程**面向对象设计和编程**中一直被作为一个编程练习使用（自 1993 年以来）。

相关的模式

协议层次结构与 Composite(163)相类似，设计它的目的是为了统一地处理各种类型。Composite 是专门用于解决递归包含的一个**对象**模式[①]，而协议层次结构是一个适用于更通用应用程序（包括复合）的**类**模式。

协议层次结构也可以用于实现一系列 Iterator(257)类型，这些类型由每个迭代器所返回的类型（例如，IntIter、DoubleIter、EdPersistentIter、EdGraphicalIter 以及 EdGeometryIter）刻画。所有的迭代器功能（除了返回当前对象的方法），都能以一种通用的基协议（例如 Iterator）统一地描述。用这种方法，与具体容器类相关的具体迭代器可以在一个组件的.c 文件中完整地定义。这种迭代器能够从对象中获取，并且通过一个通用协议来使用，同时在定义它的.c 文件之外不必暴露具体迭代器的定义。

将现存具体类型集成到一个协议层次结构中的这种多重继承的使用，是 Adapter(139)模式的类形式的一个实例。

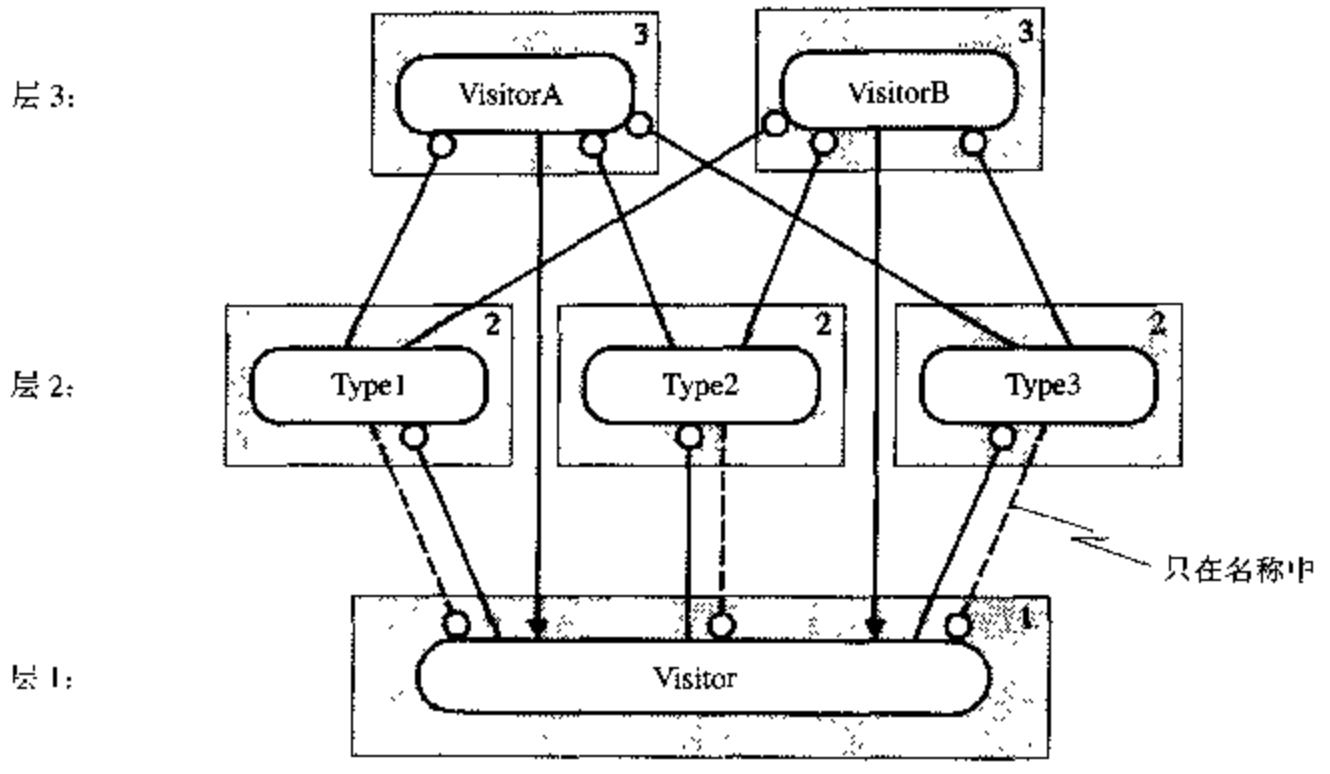
维护纯粹抽象类型的一个继承层次结构类似于 Bridge(151)模式的精神，在 Bridge(151)模式中，一个接口层次结构分离了客户程序和物理实现。一个并行的层次结构可以用来收集那些在一个协议层次结构中的具体叶子对象之间共享的实现。

协议层次结构遇到了一个 Decorator(175)巧妙回避了的问题：组合爆炸的潜在可能。一个协议类层次结构的候选对象模式，可能涉及使用在运行时建立的属性目录。

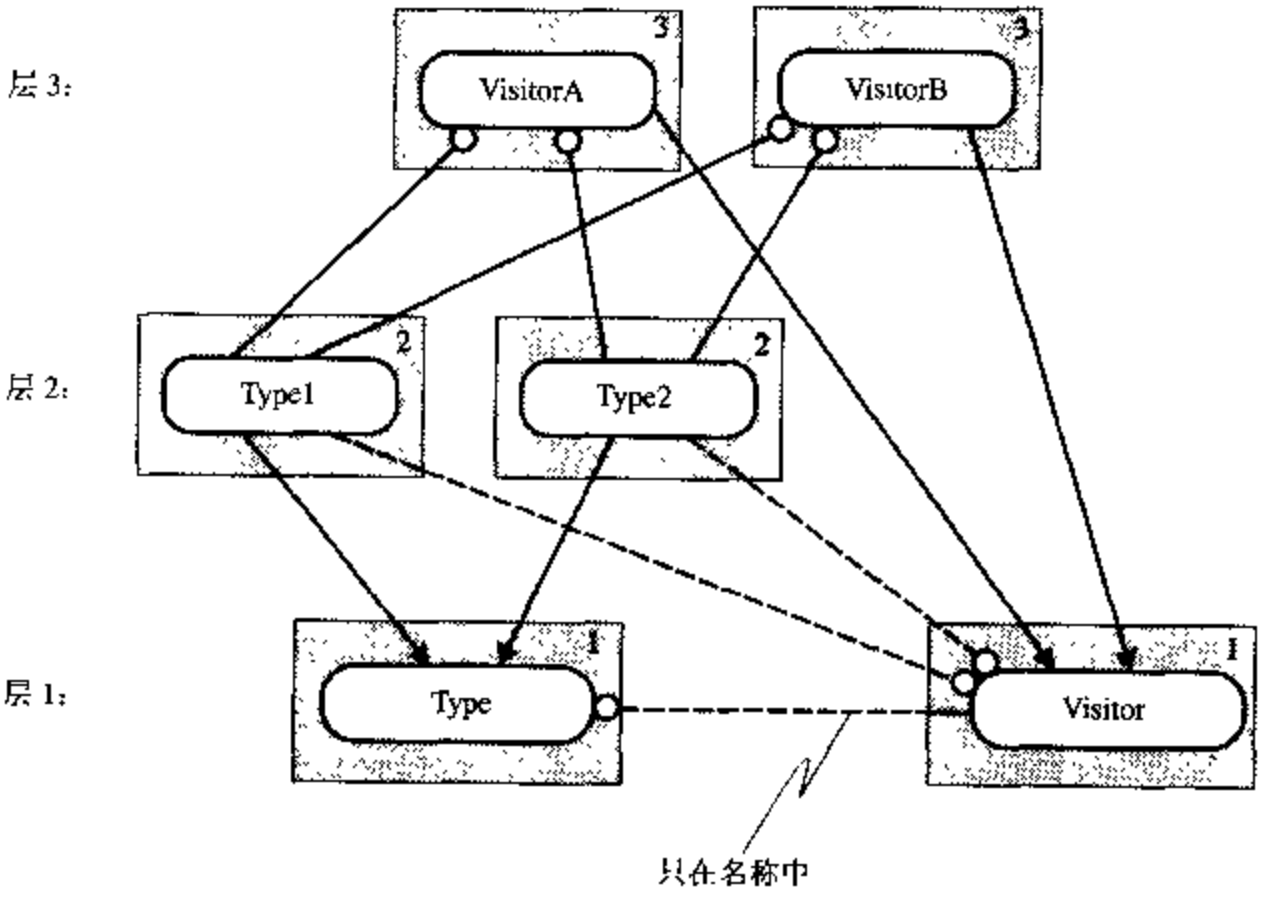
当操作是可扩展的而对象集合是固定的时候，Visitor(331)提供了一种候选的协议层次结构。Visitor 假设可扩展的操作不是基本的——即，可以根据每个具体对象的公共接口来实现。与此相对的是，协议层次结构允许透明增加新的对象类型和协议，但是基本操作集合（虽然它可能是基本的）是固定的。虽然 Visitor 不要求集合中的对象借助继承相关联，但是 Visitor 和协议层次结构都是可扩散性的（invasive）——这两种模式都要求所有参与进集合的类型要满足特定的要求。这两者中，和 Visitor 相关的物理耦合潜在地更复杂（就 CCD 而言，见 4.12 节），虽然始终是可层次化的（见图 5-64）：

---

① **gamma**，第 1 章，10 页。



如上图所示，每个具体 visitor 依赖于每个对象类型，并且所有类都依赖于 Visitor 基类，但是在物理依赖图中没有循环。增加一个通用基类来支持迭代，应该不会影响层次号，并且不必借助于动态 cast 操作来实现：



附录 B 实现一个与 ANSI C 兼容的 C++接口

有时，一个系统客户会选择（不管什么原因）使用 C 而不是用 C++来编写他们的应用程序。如果支持这样的客户也是我们系统的一个需求，那么可能要实现一个与 ANSI C 兼容的过程接口。我们选择 ANSI C 而不是 K&R（经典）C，是因为 ANSI C 与 C++是语义兼容的，并且因为经典 C 的类型太弱，不能允许任何程度合理的类型安全。由于我曾为大型 C 和 C++系统编写过这样的接口，所以我愿与大家一起分享一些在实现一个过程接口时可能有帮助的实现细节。

B.1 内存分配错误探测

图 B-1 提供了一种在探测和报告内存分配错误（这些错误是由那些使用大规模集成电路计算机辅助设计（ICCAD）产品的过程接口的客户所造成的）方面已经证明是高效的内存分配器风格的接口。分配器的接口绝缘了它的大部分实现，但是要注意，分配器本身是与客户程序完全绝缘的，这是通过在我们的系统内创建和析构对象的过程接口函数来实现的。虽然有这个分配器可能未能探测到的、在一些计算机体系结构上不正确使用过程接口函数的不正常情况，但是，在实践中这样的情况发生的可能性很小。实现一个可移植的“防弹”解决方案的代价是更高的复杂度或运行时开销。

```
// pi_alloc.h
#ifndef INCLUDED_PI_ALLOC
#define INCLUDED_PI_ALLOC
```

```
#ifndef INCLUDED_STDDEF
#include <stddef.h>                    // declares size_t
#define INCLUDED_STDDEF
#endif

class pi_Alloc {
    pi_Alloc();                        // not implemented

  public:
    // STATIC METHODS
    static void *allocate(int size, pi_Alloc *dummy);
        // Allocate specified number of bytes (with largest alignment).
        // The dummy second argument should always be null (its only
        // purpose is to encode the pi_Alloc type into the signature of
        // overloaded version of global new defined below).

    static void assertValid(void *p);
        // Use isAllocated() to determine if specified object is valid
        // and if not, take appropriate action (e.g., print a message).

    static void deallocate(void *addr);
        // Attempt to free space; try to print error if not successful.

    static int isAllocated(void *p);
        // Return 1 if memory is properly allocated, and 0 otherwise.

    static int numBlocks();
        // Return the number of outstanding allocated blocks.

    static int numBytes();
        // Return the number of outstanding allocated bytes.
};

inline void *operator new(size_t size, void *(*f)(int, pi_Alloc*))
{
    // This operator returns the result of applying its second
    // argument to its first.  The second argument of the supplied
    // function pointer is only to ensure that someone else using
    // this same technique ir a different procedural interface does
    // not bump into this definition of global operator new.

    return (*f)(size, (pi_alloc *)0);
}                                      // internal linkage

static struct pi_LeakReporter {
    pi_LeakReporter();
    ~pi_LeakReporter();
} pi_leakReporter;                     // internal linkage

#endif
```

图 B-1　一个内存分配器组件接口

图 B-2 描述了这个分配器的实现的基本思想。当这个分配器请求一个新对象的内存时，动态分配的内存比请求的要稍微多些[图 B-2（a）]。然后在移交一个指向所请求的内存分区的指针之前，分配器用自带的 magic 位模式（它的基调）以及所请求的字节数量[图 B-2（b）]初始化新分配内存的起始部分。

当对象被析构时，首先检查基调，以确定它是有效的。如果它是，则检查字节字段，以

便于调整要分配的字节数量。然后修改基调中的位模式（例如，补充）以使内存失效，以防止重复地析构（或者进一步使用）同一个对象[图 B-2（c）]。在指针返回到自由存储之前，它被调整回实际区块的首部[图 B-2（d）]。

当内存被返回到给自由存储时，不能保证修改过的基调仍保持完整。但是请注意，释放内存时，如果客户程序不拥有内存，那么，仅通过增加基调的大小，就能使一些“随机”位模式准确匹配基调的概率降到最低。实际上，如果在内存释放后，修改过的基调碰巧保持完整，那么就能更精确地报告这种特定的错误。通过特地寻找修改过的基调，我们有时能将这个错误报告为“提前释放要释放的对象”，而不是更通用的“释放未分配的存储地址”。通过使用同样的技术，任何试图使用一个分配地址作为其它过程接口的参数的错误都可以很容易地探测出来。然而，一旦内存被分配给另一个用户对象，我们就不能只使用这种简单策略来探测出指针是无效的。

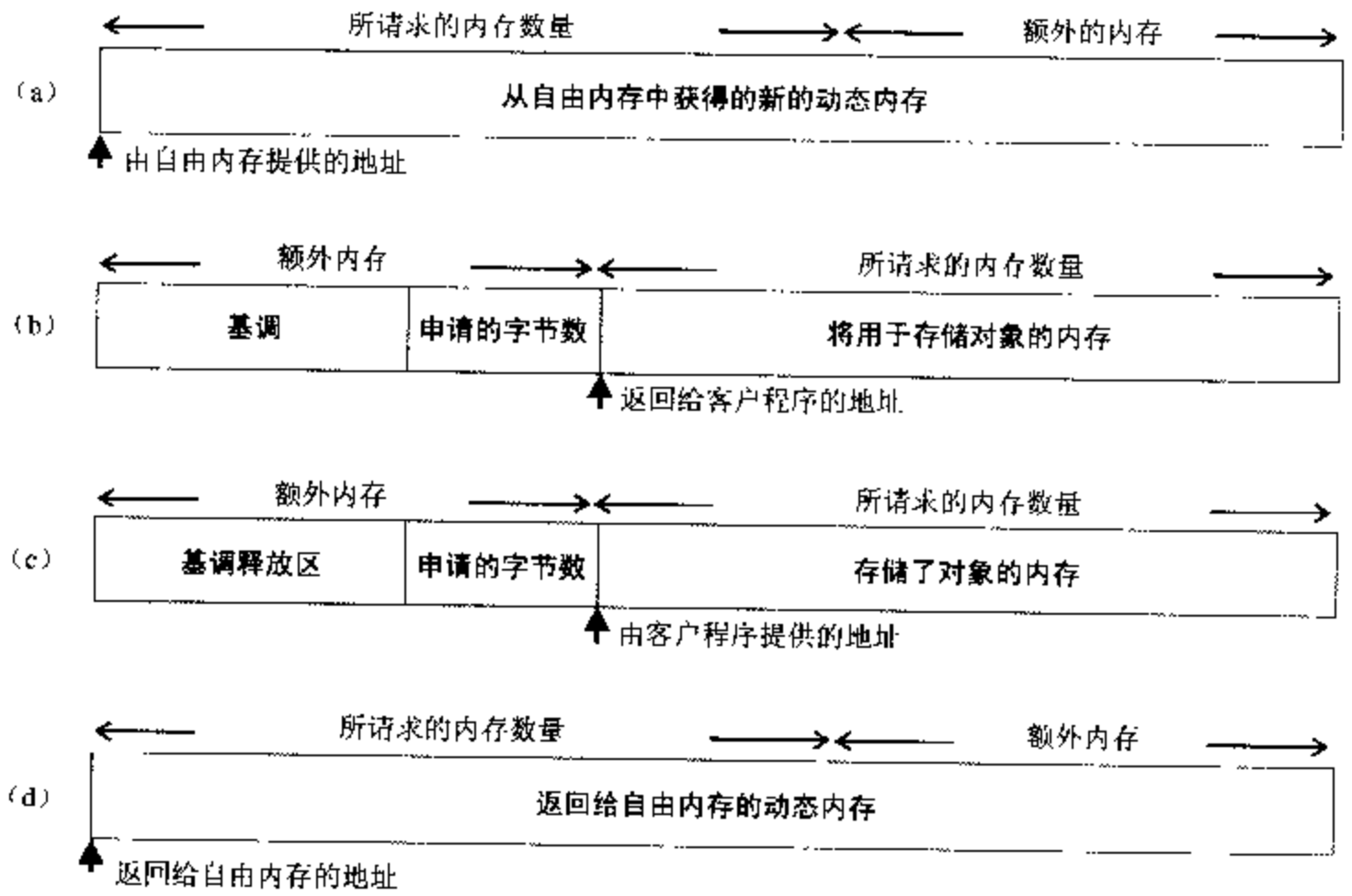


图 B-2 分配器错误探测方案的描述

图 B-3 显示了图 6-64 中的 pi_stack 创建函数以及操纵函数和访问函数的实现。我们重载全局运算符 new 以接受类型为 void *(*)(int, pi_Alloc *)的第二个参数。这与 pi_Alloc::allocate 是这种类型的情况不相符。通过调用类的构建器以及执行到新创建的实例类型的显式转换，全局 new 帮助协调类型安全。重载的全局 new 使用 pi_Alloc::allocate 函数来获得将被分配的内存。pi_Alloc::allocate 的第一个参数来自重载的 new 本身，它规定了所请求的内存数量。pi_Alloc::allocate 的第二个参数是一个哑参数（不使用），它只是将用户自定义类型 pi_Alloc

编码到重载的全局操作符 new 的基调中。通过这种方式，独立使用同样的分配策略的其他人可以定义分配器 my_Alloc，以及定义全局操作符 new 的相应重载版本，而不会与我们的版本冲突。

```
// pi_stack.c
extern "C" {
#include <pi_stack.h>
}

#include "pi_alloc.h"
#include "stack.h"

// CREATORS

extern "C"
Stack *pi_createStack()
{
    return new(pi_Alloc::allocate) Stack;
}

extern "C"
void pi_destroyStack(Stack *thisStack)
{
    if (pi_Alloc::isAllocated(thisStack)) {
        thisStack->Stack::~Stack();
    }
    pi_Alloc::deallocate(thisStack);
}

// MANIPULATORS

extern "C"
Stack *pi_StackAssign(Stack *thisStack, const Stack *thatStack)
{
    pi_Alloc::assertValid(thisStack);
    pi_Alloc::assertValid(thatStack);
    return &(*thisStack = *thatStack);
}


extern "C"
void pi_StackPush(Stack *thisStack, int value)
{
    pi_Alloc::assertValid(thisStack);
    thisStack->push(value);
}

extern "C"
int pi_StackPop(Stack *thisStack)
{
    pi_Alloc::assertValid(thisStack);
    return thisStack->pop();
}
```

```
// ACCESSORS

extern "C"
int pi_StackTop(const Stack *thisStack)
{
    pi_Alloc::assertValid(thisStack);
    return thisStack->top();
}

extern "C"
int pi_StackIsEmpty(const Stack *thisStack)
{
    pi_Alloc::assertValid(thisStack);
    return thisStack->isEmpty();
}

extern "C"
int pi_StackIsEqual(const Stack *leftStack, const Stack *rightStack)
{
    pi_Alloc::assertValid(leftStack);
    pi_Alloc::assertValid(rightStack);
    return *leftStack == *rightStack;
}
```

图 B-3 pi_stack 功能的实现

正如大家在图 B-3 中所看到的那样，使用这个分配器几乎不会比使用全局 new 本身更难。当试图去释放一个对象时，必须首先通过显式调用析构器来析构它（假设该对象是我们可以析构的）。pi_destroyStack 函数正是做这个的。对于访问和操纵功能，我们只使用由 pi_Alloc 提供的 assertValid 函数来验证被传递进去的对象是否有效。如果无效，分配器采取某种合适的操作。如果我们的客户是 C++客户，我们可能抛出一个异常（例如，pi_UserError）。但是因为我们的客户可能是 ANSI C 用户，所以我们只是打印一个信息并且不中断程序。最后，注意每个函数声明为带有 extern "C"连接，以使它能与 ANSI C 和 C++语言兼容。当使用这个分配器时，运行图 6-65 中糟糕代码的结果如图 B-4 所示。

```
john@john: a.out
pi: PROGRAMMING ERROR -  deallocating previously deallocated object
pi: MEMORY LEAK -- 1 block, 12 bytes
john@john:
```

图 B-4 探测内存分配编程错误

我们已经扩展了这个分配器，不仅要探测非法内存分配，而且要探测我们客户部分的失败，以便于在它们后面清除。用于触发退出报告的技术与 7.8.1.3 节中讨论的自动初始化是相关的。通过在 pi_alloc 组件的头内部创建一个带有内部连接的静态对象，我们可以确保在使用我们的内存分配器尝试释放一个对象之后，最后的 pi_LeakReporter 对象将被析构。图 B-5 提供了 pi_alloc 的一个完整的实现，以供参考。注意，检查的程度以及我们是否就泄露的内存报

告错误信息，可以通过编译时分支、环境变量、甚至它自己的过程接口很容易地进行控制。

```
// pi_alloc.c
#include "pi_alloc.h"
#include <iostream.h>

        // -*-*-*-*- class pi_Alloc -*-*-*-*-

static int s_numBlocks = 0;             // internal linkage

static int s_numBytes = 0;              // internal linkage

static void oneMoreBlock(int bytes)
{
    ++s_numBlocks;
    s_numBytes += bytes;
}                                       // internal linkage

static void oneLessBlock(int bytes)
{
    --s_numBlocks;
    s_numBytes -= bytes;
}                                       // internal linkage

int pi_Alloc::numBlocks()
{
    return s_numBlocks;
}

int pi_Alloc::numBytes()
{
    return s_numBytes;
}

union Align {
    struct {
        int d_magic;
        int d_bytes;
    } d_data;
    long int d_longInt;
    long double d_longDouble;
    char *d_pointer;
};                                      // internal linkage

enum {
    ALLOCATED_MEMORY = 0xA110CAED,
    DEALLOCATED_MEMORY = ~ALLOCATED_MEMORY
};                                      // internal linkage

void *pi_Alloc::allocate(int size, pi_Alloc*)
{
    // Note: the second argument is never used and is present
```

```
    // only to establish the uniqueness of this overloaded
    // version of global operator new, which takes this function
    // as its second argument.

    Align *align = (Align *) new char[sizeof(Align) + size];
    align->d_data.d_magic = ALLOCATED_MEMORY;
    align->d_data.d_bytes = size;
    void *addr = ++align;
    oneMoreBlock(size);
    return addr;
}

static void report(const void *addr, const char *gerund)
{
    // Note: a gerund is the present progressive tense of
    // a verb -- i.e., a verb ending in -ing.

    cerr << "pi: PROGRAMMING ERROR -- " << gerund << ' ';
    if (!addr) {
        cerr << "null memory address";
    }
    else {
        Align *align = (Align *) addr;
        if (DEALLOCATED_MEMORY == align[-1].d_data.d_magic) {
            cerr << "previously deallocated object";
        }
        else {
            cerr << "unallocated memory address";
        }
    }
    cerr << endl;
}                                       // internal linkage

void pi_Alloc::assertValid(const void *addr)
{
    if (!isAllocated(addr)) {
        report (addr, "using");
    }
}


void pi_Alloc::deallocate(void *addr)
{
    if (!isAllocated(addr)) {
        report (addr, "deallocating");
        return;
    }

    Align *align = (Align *) addr;
    int size = align[-1].d_data.d_bytes;
    oneLessBlock(size);
    align[-1].d_data.d_magic = DEALLOCATED_MEMORY;
    delete [] (char *) --align;
};
```

```
int pi_Alloc::isAllocated(const void *addr)
{
    if (!addr) {
        return 0;
    }
    Align *align = (Align *) addr;
    return ALLOCATED_MEMORY == align[-1].d_data.d_magic;
}


       // -*-*-*-*- class pi_LeakReporter  *-*-*-*-

static int s_niftyCounter = 0;                  // internal linkage

pi_LeakReporter::pi_LeakReporter()
{
    ++s_niftyCounter;
}

pi_LeakReporter::~pi_LeakReporter()
{
    if (--s_niftyCounter <= 0) {
        int b = pi_Alloc::numBlocks();
        int y = pi_Alloc::numBytes();
        if (b > 0 || y > 0) {
            cerr << "pi: MEMORY LEAK -- "
                << b << " block" << (b != 1 ? "s" : "") << ", "
                << y << " byte" << (y != 1 ? "s" : "") << endl;
        }
    }
};
```

图 B-5　内存分配器组件实现

B.2　提供一个主过程（仅对 ANSI C）

一些 C++编译器要求定义 main 的组件用 C++编译器来编译，并且要求使用 C++连接器。这种需求的原因涉及将构造函数和析构函数调用连接在一起的过程（这些构造函数和析构函数调用在文件作用域内声明的用户定义类型实例）。为了使 ANSI C 客户能成功编译你的接口，你也许必须给它们提供一个预编译文件 main.o，该文件除了传递一个调用给另一个定义有 extern "C"连接的“main”（例如，pi_main）之外，不做其他事情：

```
extern "C" int pi_main(int argc, char *argv[]);
main(int argc, char *argv[])
{
    return pi_main(argc, argv);
}
```

用 ANSI C 编写程序的客户现在必须像对待 main 一样来对待 pi_main——即，客户必须编写和编译它们自己的 pi_main 函数。不幸的是，你也必须提供给你的客户一个连接你的库的方法，以便于你的文件作用域对象在进入 main()之前被创建，并且在离开 main 后被析构。一种方法是保证他们使用的连接程序与你的一样，但是也有另外的复杂一点的方法。不同的编译器使用不同的策略来初始化静态对象。不是所有的编译器都会对名称冲突使用同样的技术来获得类型安全连接。因为不同的编译器以不同的方式来完成这些事情，所以来自两个不同的编译器的 C++库可能不兼容。假设两个独立的公司都为基于 C++的大型子系统提供 ANSI-C 兼容接口。如果它们都试图使用上面的技术来提供它们的一个预编译 main，那么客户可能将无法将这两个子系统连接到一个单独的可执行单元中。这个问题是严重的，它使得基于 C++的库集成远比传统 C 库的集成困难得多。惟一可靠的方法是确定你所使用的所有 C++库都由同一编译器的同一版本来编译，并且你本身可以访问这个编译器。

附录 C

一个依赖提取器/分析器包

正如 3.5 节中所讨论的那样，提取和分析设计依赖是管理大型系统开发的一个必备和基本的部分。在体系结构已定义和开发工作已起步之后，从组件间提取出实际的物理依赖，并且以一种适当格式（见 4.7 节）来展示它们，通过这样的方法来**关闭循环**是非常有用的。简单的物理度量，如在 4.12 和 4.13 节中所讨论的那些，可用来在整个开发过程中刻画软件的物理结构。本附录描述了支持这项能力的三个 Unix 风格命令的使用和实现。

adep——创建别名，将文件聚合进内聚组件。

cdep——从一个文件集合中提取编译时依赖。

ldep——分析一个组件集合中的连接时依赖。

在大型软件系统的整个开发过程中很难过分强调主动使用这样的物理设计工具的重要性。

在下面的小节里，我们将使用这三个命令来检验和刻画 idep（Implementation DEPendencies）包的内部组件间依赖，这些命令在 idep 包之上实现。然后，以一种类似于其它 Unix 命令的形式（称为 *manpages*）表示这些命令的一个更正式的描述。

最后，我们将展示 idep 包体系结构的一个整体视图。这些命令以及整个下层 idep 包的源代码可以从 Addison-Wesley 网站下载（网址是 http://www.aw.com/cp/lakos.html），也可借助匿名 ftp 从 ftp.aw.com 的目录 cp/lakos 中获得——既作为层次化的一个具体数学定义，也可作为可层次化组件的包的一个例子。我真诚地希望有一天这些（或类似的）物理工具能加入到所有非小型软件系统的开发过程中。

C.1 使用 adep、cdep 和 ldep

在本节中，我们将分析和刻画在一个包括了 11 个组件的小包（idep）内部的物理依赖。

为了完成这个工作，我们使用 adep、cdep 和 ldep 命令来执行以下步骤：

- 列出当前（包）目录的源文件。
- 提取这些文件间的编译时依赖。
- 识别并且将具有非匹配根名称实现文件与相应的组件头文件配对。
- 检验是否每个实现文件都在其第一个包含命令中指定了相应的头文件（以确保每个组件头都可以隔离地编译）。
- 以层次化的顺序排序和列出包中的组件。
- 将外部目录依赖并入有助记忆的包组名。
- 以文本形式列出规范组件依赖图（已删去冗余依赖）。

首先，idep 的源目录包含下列文件：

```
idep_adep.c              idep_cdep.c              idep_nameindexmap.h
idep_aliasdep.h          idep_compiledep.h        idep_nimap.c
idep_aliastable.h        idep_fdepitr.c           idep_string.c
idep_aliasutil.h         idep_filedepiter.h       idep_string.h
idep_altab.c             idep_ldep.c              idep_tokeniter.h
idep_alutil.c            idep_linkdep.h           idep_tokitr.c
idep_binrel.c            idep_namea.c
idep_binrel.h            idep_namearray.h
```

理想的情况下，每个组件的.c 文件的根名称将严格地匹配相应.h 文件的根名称。在一些系统上（包括我的）有一个陈旧的规定：可以单独存储在一个档案文件中的.o 文件名有 14 个字符的最大长度限制。为了满足这个要求，有时候必须精简组件的名称（但决不是包前缀）。例如，在 idep 包中，11 个组件中 9 个组件精简了所属的.c 文件名：

idep 头文件名	Idep 实现文件名	
idep_aliasdep.h	idep_adep.c	（缩写）
idep_aliastable.h	idep_altab.c	（缩写）
idep_aliasutil.h	idep_alutil.c	（缩写）
idep_binrel.h	idep_binrel.c	
idep_compiledep.h	idep_cdep.c	（缩写）
idep_filedepiter.h	idep_fdepitr.c	（缩写）
idep_linkdep.h	idep_ldep.c	（缩写）
idep_namearray.h	idep_namea.c	（缩写）
idep_nameindexmap.h	idep_nimap.c	（缩写）
idep_string.h	idep_string.c	
idep_tokeniter.h	idep_tokitr.c	（缩写）
0 2 4 6 8 10 12 14	0 2 4 6 8 10 12 14	

我们的第一步是使用 cdep 命令来提取这些文件间的编译时依赖。我们需要规定一个目录顺序进行搜索，以便找到相关的头文件。这些目录可以通过使用-I<dir>选择项分别指定，或

是一次使用全部选项-i<dirlist>。为了使这个命令正确工作，我们一般会设置我们的当前工作目录为 idep 源目录，并且指定当前目录（.）为将要搜索的目录之一。

idep 包只依赖于标准编译器提供库。下面列出了一个名为 searchpath 的局部文件的内容，该文件内容包括要在其中寻找属于该当前目录的头文件的目录，以及要在其中寻找 ATT_3.0 C++编译器和下层 C 编译器标准库的目录。

```
# searchpath

                                                    # current directory
/usr/lang/ATT_3.0/include/                          # C++ library
/usr/lang/ATT_3.0/include/cc/                       # C library
/usr/lang/ATT_3.0/include/sys/                      # C++ system library
/usr/lang/ATT_3.0/include/cc/sys/                   # C system library
```

执行下列命令将为所有 idep 头文件和实现文件提取出编译时依赖（基于预处理器包含目录），并且将这些依赖格式化到一个名为 deps 的局部文件中。

```
john@john: cdep -isearchpath *.[ch] > deps
```

deps 文件通常十分长，并且一般不支持人工检查。省略的 deps 文件内容如下：

```
idep_adep.c
    idep_aliasdep.h
    idep_aliastable.h
    idep_aliasutil.h
    idep_filedepiter.h
    idep_namearray.h
    idep_nameindexmap.h
    idep_tokeniter.h
    /usr/lang/ATT_3.0/include/ctype.h
    /usr/lang/ATT_3.0/include/cc/ctype.h
    /usr/lang/ATT_3.0/include/string.h
    /usr/lang/ATT_3.0/include/memory.h
    /usr/lang/ATT_3.0/include/cc/memory.h
    /usr/lang/ATT_3.0/include/cc/string.h
...30 lines omitted...
    /usr/lang/ATT_3.0/include/cc/malloc.h

idep_aliasdep.h

idep_aliastable.h

idep_aliasutil.h

idep_altab.c
    idep_aliastable.h
    /usr/lang/ATT_3.0/include/string.h
    /usr/lang/ATT_3.0/include/memory.h
    /usr/lang/ATT_3.0/include/cc/memory.h
...30 lines omitted...
    /usr/lang/ATT_3.0/include/cc/malloc.h
```

```
...350 lines omitted...
```

下一步是从 idep 源目录的各个文件中创建组件，这可以通过将相应的实现文件和头文件配对来进行。adep 命令提供了一些方法，例如，键入下列命令可以标识出所有这样的本地源文件：这些文件是主文件名相同的头文件和实现文件。

```
john@john: adep *.[ch]
idep_adep.c
idep_aliasdep.h
idep_aliastable.h
idep_aliasutil.h
idep_altab.c
idep_alutil.c
idep_cdep.c
idep_compiledep.h
idep_fdepitr.c
idep_filedepiter.h
idep_ldep.c
idep_linkdep.h
idep_namea.c
idep_namearray.h
idep_namcindexmap.h
idep_nimap.c
idep_tokeniter.h
idep_tokitr.c
@john@john:
```

通过肉眼检查将上面这些文件配对并不困难。然而，我们的目标是要在一个 **aliases** 文件中捕捉这种文件对，该 aliases 文件将实现文件的根映射到它相应的.h 文件的根。为了实施这项任务，-s 选项压缩了后缀并且重排序了相临的名称（例如，idep_namea 和 idep_namearray），这样，较长的（组件）名称会出现在较短的（别名）名称之前：

```
john@john: adep -s *.[ch] > aliases
john@john:
```

现在我们可手动编辑局部 aliases 文件，使得它看起来像下面这样：

```
idep_aliastable idep_altab
idep_aliasdep idep_adep
idep_aliasutil idep_alutil
idep_compiledep idep_cdep
idep_filedepiter idep_fdepitr
idep_linkdep idep_ldep
idep_namearray idep_namea
idep_nameindexman idep_nimap
idep_tokeniter idep_tokitr
```

为了确保每个组件头在编译时都是自满足的，每个实现文件都应该在它的第一个包含命令中命名相应的头文件（见 3.2 节）。使用 adep 的-v（验证）模式，我们可以很容易地判定实现文件是否已经遵守了这个规则：

```
john@john: adep -v *.c
Error: corresponding include directive for "idep_adep.c" not found.
Error: corresponding include directive for "idep_altab.c" not found.
Error: corresponding include directive for "idep_alutil.c" not found.
Error: corresponding include directive for "idep_cdep.c" not found.
Error: corresponding include directive for "idep_fdepitr.c" not found.
Error: corresponding include directive for "idep_ldep.c" not found.
Error: corresponding include directive for "idep_namea.c" not found.
Error: corresponding include directive for "idep_nimap.c" not found.
Error: corresponding include directive for "idep_tokitr.c" not found.
john@john:
```

不带组件文件别名使用 adep 的验证模式（如上所示），将直接提醒我们对于这个包中的很多组件来说，相应的头文件和实现文件的根名称不是一样的。合并上述到名文件的内容可以联合相应的头文件和实现文件名，而 adep 现在安静地返回一个成功的没有文件名返回的 0 状态：

```
john@john: adep -v -aaliases *.c
john@john:
```

假如作者不小心弄反了 idep_namea.c 文件中的前面两个包含命令，那么下列信息将会被报告给标准错误，并且返回一个非 0 状态：

```
john@john: adep -v -aaliases *.c
Error: "idep_namea.c" contains corresponding include as 2nd directive.
john@john:
```

但是，可以用一个较为容易的办法来提取这些别名，如果我们已经一致地遵守了“在实现文件的第一个包含命令中命名相应的头文件”这一设计规则：

```
john@john: adep -e *.c > aliases
john@john:
```

-e（提取）模式根据“相应头文件优先”设计规则从每个特定的实现文件中提取第一个包含头文件的根名称，并且自动地将它与实现文件的根名称配对。其结果与上面通过使用 adep 的缺省模式和一个文本编辑器来手工创建的结果是一致的。虽然一些明显的错误将被报告给标准错误，但是这里假设这个设计规则已经被一致地遵守了。否则，通过提取在第一个包含命令中发现的名称而获得的别名将是不正确的。

我们现在可以基于文件（deps）和组件文件名别名（aliases）之间的组件编译时依赖分析组件间的连接时依赖了：

```
john@john: ldep -ddeps -aaliases
ALIASES:
    idep_cdep -> idep_compiledep
 idep_alutil -> idep_aliasutil
  idep_namea -> idep_namearray
   idep_adep -> idep_aliasdep
```

```
idep_fdepitr -> idep_filedepiter
   idep_ldep -> idep_linkdep
  idep_altab -> idep_aliastable
  idep_nimap -> idep_nameindexmap
 idep_tokitr -> idep_tokeniter

LEVELS:
0. /usr/lang/ATT_3.0/include/
   /usr/lang/ATT_3.0/include/cc/
   /usr/lang/ATT_3.0/include/cc/sys/
   /usr/lang/ATT_3.0/include/sys/

1. idep_aliastable
   idep_binrel
   idep_filedepiter
   idep_namearray
   idep_string
   idep_tokeniter

2. idep_aliasutil
   idep_nameindexmap

3. idep_aliasdep
   idep_compiledep
   idep_linkdep

SUMMARY:
   11 Components          3 Levels          4 Packages
   35 CCD           3.18182 ACD       1.09308 NCCD

john@john:
```

上述信息（格式化到标准输出）提供了别名、包中组件的层次化顺序、以及帮助刻画包内依赖的统计量[①]。该包中有 11 个带有非循环物理依赖的局部组件，定义了四层。在 0 层的实体没有指定的依赖，并且被假设是这个包所依赖的外部包。第 1 层的组件只依赖于 0 层包。在没有循环依赖的情况下，在 $N>1$ 层中的每个组件，依赖至少一个 $N-1$ 层的组件，并且可能依赖在更低层的其它组件，但是不依赖第 N 层或更高层的组件。

测试一个局部组件要求一个或多个局部组件（包括组件本身）被连接到一个测试驱动程序上。组件依赖（Component Dependency，CD）是为了使用一个给定组件所需要的组件数量。当 CD 是在一个应用程序中使用一个组件的开销（连接时间、磁盘空间）度量时，它不太可能反映维护实现该组件的那个组件的子系统的相对开销。

在一个子系统中增量式测试每个组件，要求一个或多个局部组件（包括正在测试的组件）连接到一个单独的驱动程序，并且单独运行。一个子系统的累积组件依赖（Cumulative

① 注意，仅知道组件层次对于计算 CCD 来说是不够的。为了计算 CCD，需要详细的组件依赖图。因此，在这个简短列表的概述中提供的 CCD 不能仅凭借它提供的层次信息来进行验证。

Component Dependency，CCD）是那个子系统中每个组件的 CD 之和。CCD 是衡量一个子系统内部组件间耦合的尺度。CCD 的值在 N 到 N^2 之间，这里 N 是指局部组件的数量。一个等于 N 的 CCD 值意味着是独立组件的一个水平子系统。等于 N^2 的值意味着是一个完全相互依赖的子系统。一个如 $N\log(N)$的值意味着是一个树状依赖图。一般来说，一个较低的 CCD 表明是一个更松散耦合、更灵活、更可理解的子系统，其中的组件能够更独立地测试和重用。

提供的其他统计量包括平均组件依赖（Average Component Dependency，ACD）和标准 CCD（Normalized CCD，NCCD）。ACD 就是 CCD 和局部组件数量 N 的比率，范围在 1 和 N 之间。NCCD 是 CCD 和有着相同局部组件数量的（理论）平衡二叉依赖树的比值。有一个比 1 小的 NCCD 的设计被认为是更加水平和松散耦合的。比 1 大的 NCCD 的设计则被认为是更加垂直和紧密耦合的。一个显著比 1 大的 NCCD 是过度或循环物理依赖的标志。一个实现了一个应用程序特定工具（例如 idep）的高质量包体系结构的 NCCD 的典型值，范围在 0.85 和 1.10 之间。注意在该工具的这个版本中，当计算依赖性度量（例如，CCD、ACD 或 NCCD）时，没有考虑 0 层包。

在默认情况下，只有组件（并且不是它们的实际依赖）被格式化为层次化顺序的标准输出。指定-l 选择项会导致组件和它的非冗余依赖被格式化到标准输出。对于冗余，我们的意思是，如果 B 依赖 A，并且 C 依赖 B 和 A，那么 C 对 A 的直接依赖是冗余的，可以忽略掉，不会影响系统的 CCD。除了包括（冗余）传递依赖的所有依赖被格式化到标准输出之外，提供-L 选择项（而不是-l）会有类似的效果。

别名不仅仅可用来将组件头文件和实现文件配对。例如，四个标准 C++/C 包含目录中的每一个都能被赋予一个单一名称的别名，C++LIB，如同在局部文件 merge 中所描述的那样：

```
# merge
C++LIB                                  # new name for all C/C++ libraries
/usr/lang/ATT_3.0/include/              # C++ library
/usr/lang/ATT_3.0/include/cc/           # C library
/usr/lang/ATT_3.0/include/sys/          # C++ system library
/usr/lang/ATT_3.0/include/cc/sys/       # C system library
```

下面的命令产生了带有标准编译器包含库的长列表，这些包含库被结合进一个单独的包里（在 C.3 节中提供了等效的图形表示）：

```
john@john: idep -ddeps -aaliases -l -amerge
ALIASES:
/usr/lang/ATT_3.0/include/cc/sys/ -> C++LIB
                        idep_cdep -> idep_compiledep
                     idep_alutil -> idep_aliasutil
      /usr/lang/ATT_3.0/include/ -> C++LIB
                     idep_namea -> idep_namearray
                      idep_adep -> idep_aliasdep
                   idep_fdepitr -> idep_filedepiter
                      idep_ldep -> idep_linkdep
  /usr/lang/ATT_3.0/include/sys/ -> C++LIB
```

```
                        idep_altab -> idep_aliastable
                        idep_nimap -> idep_nameindexmap
    /usr/lang/ATT_3.0/include/cc/ -> C++LIB
                       idep_tokitr -> idep_tokeniter
LEVELS:
0.            C++LIB

1.   idep_aliastable 0. C++LIB

         idep_binrel 0. C++LIB

    idep_filedepiter 0. C++LIB

      idep_namearray 0. C++LIB

         idep_string 0. C++LIB

      idep_tokeniter 0. C++LIB


2.    idep_aliasutil 1. idep_aliastable
                     1. idep_string
                     1. idep_tokeniter

   idep_nameindexmap 1. idep_namearray


3.     idep_aliasdep 1. idep_filedepiter
                     2. idep_aliasutil
                     2. idep_nameindexmap

     idep_compiledep 1. idep_binrel
                     1. idep_filedepiter
                     1. idep_string
                     1. idep_tokeniter
                     2. idep_nameindexmap

        idep_linkdep 1. idep_binrel
                     2. idep_aliasutil
                     2. idep_nameindexmap

SUMMARY:
         11 Components              3 Levels                  1 Package
         35 CCD               3.18182 ACD               1.09308 NCCD
```

对循环而言，适用于一个非循环依赖图的简单层次定义并不适用。为了使这个工具甚至对于循环物理依赖也可通用，我们必须引入更一般的层次化定义。每个组件被赋予一个权重（weight），定义为最大循环的大小（该组件是该循环的一个成员）。对于一个非循环设计，每个组件的权重是 1。每个包含 N 个组件的最大循环的成员被定义在比该循环的任何成员所依赖的最高（非成员）组件的层次还要高 N 的层次上。如果给定这个定义，在一个循环依赖设计中就有可能有空的层次。

例如，假设我们通过处理（除 deps 之外）局部依赖文件 extra 引入一个“额外的”依赖：

```
# extra
idep_string
    idep_linkdep
```

通过下列命令，可检测到隐含循环并报告给标准错误：

```
john@john: ldep -ddeps -aaliases -l -amerge -dextra -x
Warning<1>: The following 3 components are cyclically dependent:
    idep_aliasutil
    idep_linkdep
    idep_string

LEVELS:
0.              C++LIB

1.    idep_aliastable     0. C++LIB

          idep_binrel     0. C++LIB

     idep_filedepiter     0. C++LIB

       idep_namearray     0. C++LIB

       idep_tokeniter     0. C++LIB


2. idep_nameindexmap      1. idep_namearray


3.
4.
5.     idep_aliasutil<1>  1. idep_aliastable
                          1. idep_binrel
                          1. idep_tokeniter
                          2. idep_nameindexmap
                          5. idep_linkdep<1>
                          5. idep_string<1>

         idep_linkdep<1>  5. idep_aliasutil<1>

          idep_string<1>  5. idep_aliasutil<1>
6.       idep_aliasdep    1. idep_filedepiter
                          5. idep_aliasutil<1>

      idep_compiledep     1. idep_filedepiter
                          5. idep_aliasutil<1>


SUMMARY:
          1 Cycle                3 Members
         11 Components           6 Levels              1 Package
         51 CCD            4.63636 ACD          1.59278 NCCD
```

```
john@john:
```

在上述例子中，CCD 已从初始的 35 上升到了 51，并且 NCCD 表明其内部耦合的程度（1.59，而不是最初的 1.09）明显高于一个树形依赖图。在 idep 包中局部层次的数量现在是 6，而不是 3，这反映了一个耦合更加紧密、灵活性更少的物理设计。空层次 3 和 4 是包含 idep_aliasutil、idep_string 和 idep_linkdep 的循环的直接结果。顺便说一下，-x 选择项可用来压缩别名的打印。

```
# bigcycle
idep_filedepiter
    idep_namearray

idep_namearray
    idep_tokeniter

idep_tokeniter
    idep_binrel

idep_binrel
    idep_aliastable

idep_aliastable
    idep_linkdep

idep_linkdep
    idep_aliasdep

idep_aliasdep
    idep_compiledep

idep_compiledep
    idep_filedepiter
```

引入在上面局部文件 bigcycle 中描述的循环依赖，将导致整个包变得在物理上相互依赖：

```
john@john: ldep -ddeps -aaliases -l -amerge -dextra -x -dbigcycle
Warning<1>: The following 11 components are cyclically dependent:
    idep_aliasdep
    idep_aliastable
    idep_aliasutil
    idep_binrel
    idep_compiledep
    idep_filedepiter
    idep_linkdep
    idep_namearray
    idep_nameindexmap
    idep_string
    idep_tokeniter

LEVELS:
0.              C++LIB
```

```
1.
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.      idep_aliasdep<1>  0. C++LIB
                          11. idep_aliastable<1>
                          11. idep_aliasutil<1>
                          11. idep_binrel<1>
                          11. idep_compiledep<1>
                          11. idep_filedepiter<1>
                          11. idep_linkdep<1>
                          11. idep_namearray<1>
                          11. idep_nameindexmap<1>
                          11. idep_string<1>
                          11. idep_tokeniter<1>

          idep_aliastable<1> 11. idep_aliasdep<1>

           idep_aliasutil<1> 11. idep_aliasdep<1>

              idep_binrel<1> 11. idep_aliasdep<1>

           idep_compiledep<1> 11. idep_aliasdep<1>

         idep_filedepiter<1> 11. idep_aliasdep<1>

            idep_linkdep<1> 11. idep_aliasdep<1>

           idep_namearray<1> 11. idep_aliasdep<1>

        idep_nameindexmap<1> 11. idep_aliasdep<1>

            idep_string<1> 11. idep_aliasdep<1>

           idep_tokeniter<1> 11. idep_aliasdep<1>


SUMMARY:
            1 Cycle                 11 Members
           11 Components            11 Levels                1 Package
          121 CCD                   11 ACD             3.77894 NCCD

john@john:
```

上面的例子表示了一个最糟糕的情况，其中包里的每个组件都直接或间接地依赖每个其它的组件。ACD 和 CCD 分别等于它们的最大值（N=11 以及 N^2）。NCCD 为 3.77，这清楚地说明了该包相对于同等大小的树状体系结构的过度耦合。如果作者没有进行一些有意识的努

力来最小化物理依赖的话，那么一个子系统表现出几乎最坏的相互依赖并非不寻常。

总结：下面是一个命令列表，我们执行这些命令来获得层次化组件列表以及规范组件依赖图的文本表示：

- 提取编译时文件依赖，放入一个局部 deps 文件：

```
cdep -isearchpath *.[ch] > deps
```

- 列出不配对的组件文件名：

```
adep *.[ch]
```

- 将未配对的根文件名输出到一个 aliases 文件：

```
adep -s *.[ch] > aliases
```

- 验证第一个#include 指令是否命名了相应的头文件：

```
adep -v -aaliases *.c
```

- （可选）自动提取组件文件别名：

```
adep -e *.c > aliases
```

- 以层次顺序列出包的组件：

```
ldep  ddeps -aaliases
```

- 列出带有帮助记忆的包组名称的规范组件依赖：

```
ldep -ddeps -aaliases -l -amerge
```

这些命令的详细描述以及附加选项在下面一节中介绍。

C.2 命令行文档

本节描述三个 Unix 风格命令的功能以及这些程序的输入和输出文件格式。

名称

adep——创建别名，将文件组织进内聚组件。

说明

adep [-s] [-a*aliases*] [-f*filelist*] [-X*fn*] [-x*xFile*] *filename**

```
adep -v [-aaliases] [ ffilelist] [-Xfn] [- xxFile] cfilename*
adep -e [-aaliases] [-ffilelist] [-Xfn] [-xxFile] cfilename*
```

描述

adep 从文件集合中创建、验证以及提取别名。adep 的这三个主要形式如上面的说明所示。

adep 的第一个形式试图将头文件和实现文件配对。对于每个在命令行中指定的 filename，文件的后缀部分被删除了。每个未配对（在应用了一些别名之后）文件的名称将被打印到标准输出（每行一个）。如果映射到单个组件名的文件名多于两个，那么会产生一个警告到标准错误，但不会影响返回状态。在这个模式里，如果不能打开一个或多个文件以进行读访问，那么 adep 返回一个负值；否则返回 0。

adep 的第二个形式验证（在应用了一些别名之后）命令行中指定的每个实现文件是否都包含一个文件（作为其第一个指令），该文件的名称应与指定文件的根名称相匹配。如果不是，报告一个错误信息给标准错误。在这个模式里，如果不能打开一个或多个文件以进行读访问，那么 adep 返回一个负值。另外，如果一个或多个指定的实现文件有不完整的或是丢失的 #include 指令，那么 adep 返回一个正值。否则，adep 返回一个 0 状态。

adep 的第三种形式通过假设每个指定实现文件中的第一个#include 指令定义了相关头文件名称，试图为未配对的实现文件提取别名。任何一个还未建立（通过同样的根名称或借助别名）的映射都被作为名称/别名对格式化到标准输出（每行一个）。如果有多于一个文件映射到同样的头文件，那么会产生一个警告信息到标准错误，但不会影响返回值。在这个模式里，如果不能打开一个或多个文件以进行读访问，那么 adep 返回一个负值。另外，如果一个或多个指定的实现文件没有#include 命令，那么 adep 返回一个正值。否则，adep 返回一个 0 状态。

如果没有给出 filename 参数，并且没有使用-f*filelist* 选择项，则 adep 从标准输入读取要处理的文件名的列表。

选择项

-s　为未配对的名称压缩后缀打印。在第一个（缺省）模式中，未配对文件名将带着后缀以字母表顺序打印。指定这个选择项会导致压缩后缀，并且会调整输出顺序，这样两个初始匹配的相邻名称中较长的那个会出现在另一个之前（为了便于使用一个文本编辑器来创建一个别名文件）。例如：

```
adep -s my_long*.[ch]
```

不加 -s	加 -s
my_longlostlove.h	my_longlostlove

```
my_longlslov.c                  my_longlslov
my_longnaitr.c                  my_longnaitr
my_longnamea.c                  my_longnamearray
my_longnamearray.h              my_longnamea
my_longnameiter.h               my_longnameiter
```

-v 验证每个指定的实现文件是否在它的第一个#include 指令中包含了相应的头文件。在这个模式中，比较实现文件名和在第一个#include 指令中找到的头文件名。如果在应用了一些相关的别名之后，一个实现文件没有包含有着和它的第一个#include 指令相对应名称的头文件，那么将报告一个错误信息给标准错误。例如：

```
// wrongorder.c
#include <iostream.h>
#include "wrorgorder.h"
// ...
```

```
// missing.c
// ...
```

```
adep -v wrongorder.c missing.c

stderr:   Error: "wrongorder.c" contains corresponding
          include as 2nd directive.
stderr:   Error: corresponding include directive for
          "missing.c" not found.
```

-e 使用在第一个#include 指令中发现的名称，为每个未配对文件名提取一个别名，并且将这些别名打印到标准输出（每行一个）。如果（在应用了一些别名之后）在第一个#include 指令中命名的文件的根已经和实现文件的根相匹配了，那么不打印别名。例如：

```
// my_longnamea.c
#include "my_longnamearray.h"
// ...
```

```
// my binaryrel.c
#include "my_binaryrelation.h"
#include <iostream.h>
// ...
```

```
adep -e my long*.c my_bir*.c

stdout:   my_longnamearray my_longnamea
stdout:   my_binaryrelation my_binaryrel
```

注意，带有误放或丢失#include 指令的文件将破坏这个命令模式的效果：

```
adep -e wrongorder.c missing.c

stderr:   Error: "missing.c" contains no include directives.
stdout:   iostream wrongorder
```

-aaliases 指定一个包含组件名称别名的文件。*aliases* 文件包含了序列的一个集合(见文件格式)。在每个序列中的第一个名称标示出主要的名称。序列中的其余名称是那个主要名称的同义名。例如，假设当前目录包含了如下文件：

```
my_longlostlove.h    my_longnaitr.c     my_longnamearray.h
my_longlslov.c       my_longnamea.c     my_longnameiter.h
```

下面的 aliases 文件把每个.c 文件的根名称映射到其对应.h 文件的根名称。

```
# aliases
my_longlostlove my_longlslov
my_longnamearray my_longnamea
my_longnameiter my_longnaitr
```

-ffilelist 指定一个包含要处理文件名称序列的文件。效果是似乎这些单个文件名中的每一个都已经被指定为一个命令行参数了（一个空 *filelist* 将抑制从标准输入读取文件名)。

-Xfn 指定一个要在处理期间忽略的文件名称：

```
adep -e -Xmy_main.c *.[ch]
```

-xxFile 指定一个包含了一个文件名序列的文件，这些文件名在处理期间将被忽略。效果是这些单个文件名中的每一个好像都已经通过使用-X*fn* 选择项指定了。

错误

试图从没有在第一个#include 指令中一致地命名相关头文件的实现文件中提取相关的头文件，则会导致错误的结果。

参考

cdep、ldep、文件格式

名称

cdep——从一个文件集合中提取编译时依赖。

说明

cdep [-I*dir*] [-i*dirlist*] [-f*filelist*] [-x] *filename**

描述

为命令行中指出的每个 *filename* 确定所有被这个文件直接或间接包含的头文件，并且格式化这些依赖到标准输出。用来搜索#include 文件的目录序列（包括当前目录“.”）必须使用如下所示的-I*dir* 和-i*dirlist* 选择项的组合来指定：

```
john@john: cdep -I. -isearchpath *.[ch] > deps
```

例如，上面的命令行使用在文件 searchpath 中指定的目录名来寻找定义在当前目录每个.h 和.c 文件中的头文件，并且将这些结果放置在文件 deps 中。如果没有给出 filename 参数，并且没有使用-f*filelist* 选择项，那么 cdep 从标准输入读取要处理的文件名列表。输出格式是一个以空行终止的名称序列——每个输入文件占一个序列。在每个序列中的第一个名称标示输入文件，并且随后的每个名称标示输入文件在编译时所依赖的一个文件（见文件格式）。如果不能打开一个或多个文件进行读访问，那么该命令返回一个负值；否则 cdep 返回一个 0 状态。

选择项

-I*dir* 指定一个包含目录的名称以便将该目录追加到搜索路径上。例如，-I，将追加当前目录到搜索路径上。

-i*dirlist* 指定一个包含了一个目录名序列的文件将目录追加到搜索路径上。该选项的作用类似于通过-Idir 选项分别指定每个目录。

-f*filelist* 指定一个包含了一个文件名（要进行处理的）序列的文件。效果是将这些单独的文件名每一个都已经指定为一个命令行参数（一个空 filelist 将抑制从标准输入读入文件名）。

-x 不递归地检查嵌套的#include 命令。只有那些由输入文件直接包含的文件才会在输出中指定。

错误

甚至那些已经#ifdef 掉或通过/*...*/注释掉的包含命令也将被作为有效解释。

参考

adep、ldep、文件格式

名称

ldep——分析一个组件集合中的连接时依赖。

说明

```
ldep [-Udir] [-uun] [-aaliases] [-ddeps] [-l | -L] [-x | -X] [-s]
```

描述

ldep 处理处于别名上下文中的文件间编译时依赖的一个集合，以推导出组件间的连接时依赖。组件以从 0 层开始的层次化顺序被格式化到标准输出（每行一个）。层次上的一个变化都用一个额外的空行来标示。

在默认情况下，对目录（除了当前目录）中组件的依赖被视为对目录本身的依赖（见下面的-U 和-*uun* 选择项）。

在默认的情况下，别名、非别名以及刻画了组件依赖的统计量的一个概述都被格式化到一个标准输出（见下面的-x 和-X 选择项）。这些统计量包括：

组件 0 以上层次的局部组件的数量（即，有至少一个依赖的组件数量）。

层次 局部组件依赖图的高度。

包 0 层实体的数量（即，不带依赖的实体数量）。

CCD 连接和测试 C_i 所需的局部组件数量在所有局部组件 C_i 上的总和。

ACD CCD 和局部组件数量的比值。

NCCD 该 CCD 和一个拥有相同局部组件数量的理论平衡二叉依赖树的 CCD 的比值。（注意，对于大部分高质量包体系结构来说，这个数值不会大于 1.00）。

在循环组件依赖的情况下，每个不同的最大循环的成员都被分别标识并报告给标准错误。一个附加的概述行在其它行之前被格式化到标准输出，包括以下信息：

循环 在组件依赖图中不同最大循环的数量。

成员 参与进循环的组件总数。

一个包含了 N 个组件的最大循环中的每个成员，被定义为处在此循环中的任何成员都要依赖的最高的（非成员）组件所处层次还要高 N 层的层次上。

这个命令不带参数。除非-d*deps* 选择项已被调用，否则依赖本身将来自标准输入。如果不能打开一个或多个文件进行读访问，那么 ldep 命令返回一个负值。另外，如果在组件依赖图中至少有一个循环被检测出来，那么 ldep 返回一个正的状态。否则，ldep 返回一个 0 状态。

选择项

-U*dir* 指定一个外部目录，其中的文件要单独处理。缺省情况下，所有对当前目录（.）

之外的文件的依赖都被当作是对包含文件的目录（包）的依赖。例如，假设

```
ldep -ddeps
```

产生了下面的结果：

```
LEVELS:
0. /usr/lang/ATT_3.0/include/
   /usr/lang/ATT_3.0/include/cc/
   /usr/lang/ATT_3.0/include/sys/
   /usr/lang/ATT_3.0/include/cc/sys/

1. idep_aliastable
   idep_binrel
   idep_namearray
   . . .
```

然后

```
ldep -ddeps -U/usr/lang/ATT_3.0/include/sys
```

可能产生在指定目录中标识单个组件（例如，stdtypes、types 以及 wait）的下面的列表。

```
LEVELS:
0. /usr/lang/ATT_3.0/include/
   /usr/lang/ATT_3.0/include/cc/
   /usr/lang/ATT_3.0/include/sys/stdtypes
   /usr/lang/ATT_3.0/include/sys/types
   /usr/lang/ATT_3.0/include/cc/sys/
   /usr/lang/ATT_3.0/include/sys/wait

1. idep_aliastable
   idep_binrel
   idep_namearray
   . . .
```

-u*un* 为那些需要单独处理的组件指定一个包含目录的文件。作用是类似于使用-U*dir* 选择项指定单个目录。

-a*aliases* 指定一个包含组件名称别名的文件。*aliases* 文件包含一个序列的集合（见文件格式）。在每个序列中的第一个名称标示主要名称。序列中其余的名称是那个名称的同义词。例如，假设当前目录包含了下面列出的文件：

```
my_longlostlove.h    my_longnaitr.c    my_longnamearray.h
my_longlslov.c       my_longnamea.c    my_longnameiter.h
```

下面的 *aliases* 文件把每个.c 文件的根名称映射到其对应.h 文件的根名称。

```
# aliases
my_longlostlove my_longlslov
my_longnamearray my_longnamea
```

```
my_longnameiter my_longnaitr
```

然后映射将像下面这样显示出来：

```
my_longnaitr -> my_longnameiter
my_longnamea -> my_longnamearray
my_longlslov -> my_longlostlove
```

我们也可以使用别名来将几个组件并入一个单独的实体，这需要把所有的组成文件名映射到一个单个名称上。例如，下面的 aliases 文件将每个组件文件名称映射到一个单一的标识符：

```
# aliases
MY_LONG
my_longlostlove my_longlslov
my_longnamearray my_longnamea
my_longnameiter my_longnaitr
```

这个映射的结果如下所示：

```
my_longnamearray -> MY_LONG
    my_longnaitr -> MY_LONG
    my_longnamea -> MY_LONG
 my_longnameiter -> MY_LONG
 my_longlostlove -> MY_LONG
    my_longlslov -> MY_LONG
```

-d*deps* 指定一个包含一个编译时文件依赖列表的文件。*deps* 文件包含了一个以空行终止的名称序列的集合。每个序列的第一个名称标示依赖文件，每个随后的文件标示那个文件所依赖的文件（见文件格式）。如果这个选择项未被调用，那么文件依赖本身要从标准输入读取。

-l|-L 提供一个包含特定组件依赖的长列表。在缺省情况下，仅仅组件（而不是它们的依赖）被按照层次顺序格式化到标准输出。指定-l 选择项会导致组件和它的非冗余依赖（即，除了传递依赖）以 *deps* 格式发送到标准输出（见文件格式）。由一个附加的空行伴随着层次的一个变化。提供-L 选择项有一个相似的效果，除下述情况之外：包括（冗余）传递依赖的所有依赖都被格式化到标准输出。

-x|-X 压缩不重要信息的打印。在缺省情况下，别名、非别名、层次号以及概述会和组件名称一起，按照以 0 层开始的层次顺序打印到标准输出。指定-x 选择项可以压缩别名和非别名的打印。而指定-X 选择项可以压缩除了组件名本身之外的所有信息。（-X 选择项是用来驱动一个图形显示的；注意，一个附加的空行标示一个层次上的变化。）

-s 不要删去后缀——单独考虑每个文件。（这个选择项有时侯是有用的，有助于判断组件间循环依赖的成因。）

错误

为了让ldep产生正确的结果，假设没有实现文件使用了局部声明来访问带有外部连接（见1.1.2节）的非局部实体（例如，全局变量或自由函数）。extern关键词的任何使用都是可疑的。

CCD忽略了对其它包的依赖的权重。

一个不带#include命令的局部组件empty将被赋予一个0层次号，并且将被误认为是一个外部包。为那个组件加上一个对“.”的虚构的依赖就可以解决这个问题：

```
# artificial dependencies
empty
```

参考

adep、cdep、文件格式

文件格式（FILE FORMATS）：

下面描述特定的文件格式。每个格式都支持注释的概念。注释是一个以#符号开始的标记，它隐藏所有的标记，直到遇见一个*新行*：

```
this is valid input text #this is a comment
```

这些格式中的许多只不过是一个以空格分隔的标记列表。这样的格式以符号<list>来标识。

aliases：以空格分隔的标记序列。在每个序列中的第一个标记标示主要的名称。每个随后的标记为那个名称标示出一个同义词。如果第一个标记单独出现在一行中，那么该序列以一个空行终止。否则该序列以一个新行终止。在一个新行前面放一个单独的反斜杠\，可继续该逻辑行。在同一行跟在反斜杠后面有一个注释，并不会干涉逻辑行的继续。例如：

```
# aliases
C++                                     # standard libs
/usr/lang/ATT_3.0/include/              # directory
/usr/lang/ATT_3.0/include/cc/           # subdirectory
/usr/lang/ATT_3.0/include/sys           # subdirectory
/usr/lang/ATT_3.0/include/cc/sys        # subdirectory

idep_nameindexmap idep_nimap
idep_namearray \     # this is line continuation
idep_namea
```

产生了下面的别名：

```
   /usr/lang/ATT_3.0/include/sys -> C++
                     idep_namea.c -> idep_namearray
      /usr/lang/ATT_3.0/include/ -> C++
/usr/lang/ATT_3.0/include/cc/sys -> C++
                      idep_nimap -> my nameindexmap
   /usr/lang/ATT_3.0/include/cc/ -> C++
```

deps: 以空格分隔的标记序列，由一个空行终止。在每个序列中的第一个标记标示根文件、每个随后的标记标示那个根文件所依赖的一个文件。例如，

```
john@john: cdep -isearchpath idep_tokeniter.[ch]
```

在我的系统上产生了下列依赖文件：

```
idep_tokeniter.c
    idep_tokeniter.h
    /usr/lang/ATT_3.0/include/ctype.h
    /usr/lang/ATT_3.0/include/cc/ctype.h
    /usr/lang/ATT_3.0/include/string.h
    /usr/lang/ATT_3.0/include/memory.h
    . . .
    . . . (30 lines omitted)
    . . .
    /usr/lang/ATT_3.0/include/malloc.h
    /usr/lang/ATT_3.0/include/cc/malloc.h

idep_tokeniter.h
```

注意：文件 idep_tokeniter.c 有一些直接或间接的编译时依赖，而文件 idep_tokeniter.h 却没有。

dirlist: <list>中每个标记都标示要追加到搜索路径上的一个目录名称：

```
# my search path
                                  # current directory
/usr/john/app/include             # app include directory
/usr/lang/ATT_3.0/include         # standard C++ library
/usr/lang/ATT_3.0/include/cc      # standard C library
```

filelist: <list>中每个标记都标示一个将要处理的文件名。

```
# my files
idep_nameindexmap.t.c             # test driver
idep_nameindexmap.h idep_nimap.c
idep_namearray.h idep_namea.c
```

un: <list>中每个标记都标示要单独处理的文件的一个外部目录名称。

```
# my unalias directories
/usr/john/app/include             # In case we want to
/usr/lang/ATT_3.0/include         # know exactly what
```

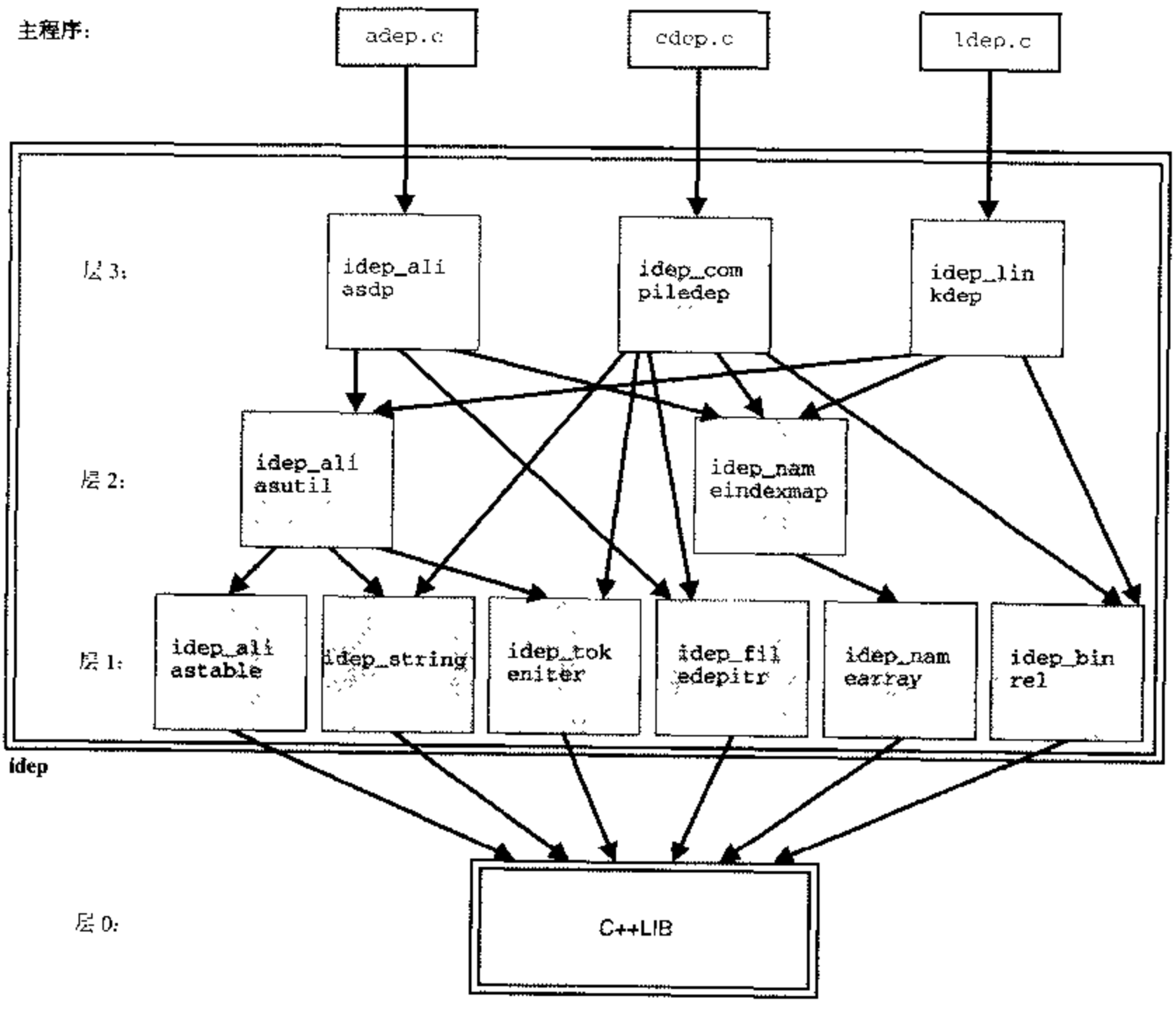
```
/usr/lang/ATT_3.0/include/sys    # non-local components
                                 # we depend on.
```

xFile: <list>中每个标记都标示在处理期间将被忽略的一个文件的名称。

```
# ignore these files
idep_nameindexmap_t.c            # test driver
main.c                           # main program
```

C.3 ldep包体系结构

在本节中，我们将简单讨论idep包的物理体系结构，以及三个Unix风格命令（adep、cdep和ldep）的实现：



上图描述了 idep 包内部的组件依赖[①]。注意，在组成 idep 包的 11 个组件中有 6 个是叶子组件（只依赖标准的编译器供应库）。这些组件中的每一个都可以隔离地进行有效测试，并且可以独立于包中的其它组件而重用。

idep_aliastable 实现了一个从一个名称到另一个名称的简单（哈希表）映射。中间客户没有与它的实现绝缘；然而，该表和定义在这个组件中的迭代器类都没有暴露在任何更高层包装器组件的接口中。如果要在这个包的外部使用，那么一个完全绝缘的实现将是首选的。（注意：一旦一个标准的 C++组件库变得普遍可用，那么像这样的组件实现（如这一个）就可能变得不合时宜。）

idep_string 定义了一个不带装饰的基本字符串类以及一个简明的实现。该字符串的实现没有绝缘，但是如果这个组件在它的包外部被独立使用的话，那么该字符串就应该绝缘。

idep tokeniter 是一个简单的语法解析器类，它读入一个标记流，返回以空格分隔的标识符以及 *Newline* 字符。该组件的内在可重用性与它的权值（由 I/O 产生）促进了它的完全绝缘的实现。

idep_filedepiter 定义了一个类，该类依次从一个文件中提取出#include 命令中的所有头文件名。出于与 idep_tokeniter 同样的原因，该类也有一个完全绝缘的实现（虽然 idep_filedepiter 的使用被更直接地绑定到一种特定的应用程序）。

idep_namearray 不需要格外的功能就实现了一个名称的可扩展数组。它的实现是如此的合乎标准，以至于它似乎不需要绝缘。（注意：一旦一个标准的组件库变得普遍可用，就有可能不再需要这个组件。）

叶子组件的最后一个，idep_binrel，定义了一个低层布尔矩阵类，用来表示集合之间的二元关系。对数组进行有效（内联）访问的潜在需要阻碍了一个完全绝缘的实现。

有两个组件驻留在第二层，idep_aliasutil 是一个用来从一个文件或输入流中读取别名的工具组件。当对别名输入的语法分析与 idep_aliastable 组件逻辑内聚时，分析功能单独引入了对 idep_tokeniter 和 idep_string 的附加依赖。把这个语法分析能力放在 idep_aliastable 组件本身，会迫使 idep_aliastable 的所有客户连接到 idep_string 和 idep_tokeniter 上，而不管它们是否需要从一个文件中分析别名。将这个分析功能升级到一个更高层的工具组件上，可以避免将这两个附加的依赖强加到 idep_aliastable 的所有客户上。

另一个第二层的组件 idep_nameindexmap，支持在名称和非负连续整数之间的快速双向转换。这个组件在它的（完全绝缘的）实现中使用了 idep_namearray。

第三层包含了三个绝缘的包装器组件，它们向一个更高层的包或向实现了一个 Unix 风格命令（例如，adep.c、cdep.c 或者 ldep.c）的主程序展示组合应用程序功能。每个包装器组件提供了一个（完全绝缘的）主要类，作为将要被执行计算的累加信息的环境类。在环境对象

---

① 通过使用带有-x 选项的 ldep 来只创建组件依赖，可以生成这张图，然后把结果输出导入一个简单的图形显示引擎。这个引擎将允许用户交互地在层次内水平移动组件，并且将保存组件图标的位置，以方便对体系结构进行增量式改变。

被完全配置之后，会调用适当的处理行为，以产生所期望的结果。

idep_aliasdep 定义了一个单个包装器类，用于为相应的组件头文件和具非匹配根名称的实现文件配对，以及用于验证实现文件内部的包含命令是否被正确放置。这个类的操纵函数被用来编制处理的环境。因为实际的处理并不影响这个对象的逻辑状态，所以这个类中的每个处理函数都声明为一个 const 成员。

idep_compiledep 定义了一个用于计算文件间编译时依赖的主要包装器类。当客户与这个包装器组件中使用的低层组件进行任何接触时，这个组件也提供了两个完全绝缘的迭代器类，允许客户检索下层的依赖信息。而且，这些类在它的逻辑接口中都只使用基础 C++类型，因此进一步减少了对实现选择上的约束，并且增强了可重用性。

在三个绝缘的包装器组件中，idep_linkdep 是最复杂的，它定义了主要的 idep_LinkDep 类以及七个迭代器类。这个功能的大部分是通过把工作托付给更低层对象来实现的。这个局部实现的功能被分离在计算组件的层次化和格式化输出之间。通过将这个包装器分解成两个组件，可以减少这个组件的复杂性（以及改善可维护性）。一些局部定义的数据结构可能被降级到一个新的（更低层次的）idep_linkdepimp 组件中，在那里它们可以被更直接地测试。

剩下的 idep_linkdep 组件仍然是相当的长，但是它所直接实现的复杂功能的数量显著地减少了。

作为一个整体，这个包从一开始就被设计为支持三个包装器组件，每一个都重用大多数下层实现组件。由于设计不够谨慎而且准备不足，这个包的 NCCD（1.09）所反映的耦合程度，比通常期望的设计良好的应用程序所具有的耦合程度要更高。为了正确地评价这些度量，我们必须考虑重用。假如这个包仅仅实现一个单个的包装器（例如，idep_linkdep），那么 NCCD 将是 0.83——远小于（理论）平衡二义依赖的值 1.00。一定要记住，NCCD 刻画了组件之间的依赖，并且可能与可靠性和可重用性有一个反相关性，但是，它本身并不是设计质量的一个绝对度量。

C.4 源代码

实现了三个 Unix 型命令（adep、cdep 和 ldep）的上程序的完全源代码、idep 包中组件的头文件以及它们的相应实现文件都可以从 Addison-Wesley 网站获得，网址是 http://www.aw.com/cp/lakos.html，也可借助匿名 ftp 从 ftp.aw.com 的目录 cp/lakos 获得。最初，下面的命令可以用来编译和连接这三个工具（在我的基于 Unix 的系统上）：

```
john@john: CC -c idep_*.c
john@john: CC -o adep adep.c idep_*.o -lm
john@john: CC -o cdep cdep.c idep_*.o -lm
john@john: CC -o ldep ldep.c idep_*.o -lm
```

一旦编译过了，可以调整 cdep 以便自动地为 Unix 风格的 makefiles 产生任意的头文件依赖。

附录 D 快速参考

下面是本书讨论的所有定义、设计规则、指导方针以及原则。

D.1 定义

第 1 章：

- 声明把一个名称引入程序；定义则提供了程序内一个实体（例如，类型、实例、函数）的一个独特描述。
- 如果一个名称对于它的编译单元来说是局部的，并且在连接时不可能与其它编译单元中的名称相冲突，那么这个名称有**内部连接**。
- 在一个多文件程序中，如果一个名称在连接时可以和其它编译单元互相影响，那么这个名称就有**外部连接**。

符号	含义
X	X 是一个逻辑实体（例如，类）
X	X 是一个物理实体（例如，文件）
B —IsA→ A	B 是 A 的一种
B —Uses-In-The-Interface— A	B 在 B 的接口中使用 A
B —Uses-In-The-Implementation— A	B 在 B 的实现中使用 A

- 如果在声明一个函数时提到了某个类型，那么就是在该函数的接口中使用了该类型。

- 如果在一个类的（公共）成员函数的接口中使用了某类型，那么就是在这个类的（公共）接口中使用了该类型。
- 如果在某函数的定义中提到了某类型，那么在这个函数的实现中就使用了该类型。
- 如果一个类型（1）被用在某个类的一个成员函数中，（2）在某个类的一个数据成员的声明中被提到，或者（3）是某个类的一个私有基类，那么在这个类的实现中就使用了这个类型。
- Uses-In-The-Implementation（在实现中使用）关系的特定种类：

名称	含义
Uses	该类有一个成员函数命名了该类型。
HasA	该类嵌入了该类型的一个实例。
HoldsA	该类把一个指针（或引用）嵌入了该类型。
WasA	该类私有继承于该类型。

- 如果某个类在它的实现中实质地使用了某个类型，则该类分层于该类型之上。

第2章

- 若不能通过某个类的逻辑接口编程访问或检测到其包含的实现细节（类型、数据或函数），则称这些实现细节被那个类封装了。

第3章

- 一个**组件**（**component**）就是物理设计的最小单位。
- 一个组件的逻辑接口就是可被客户通过编程访问或检测到的东西。
- 在一个组件的.h 文件的文件作用域中定义的任何类或声明的任何自由（运算符）函数的公共（或保护）接口中，如果使用了一个类型，那么称**在这个组件的接口中使用了这个类型**（**Used-In-The-Interface**）。
- 一个组件的**物理接口**就是它的头文件中的**所有东西**。
- 如果在一个组件的任何地方通过名称提到了一个类型，那么称**在这个组件的实现中使用了这个类型**（**Used-In-The-Implemention**）。
- 如果编译或连接组件 y 时需要组件 x，则组件 y **依赖**（**DependsOn**）组件 x。
- 如果编译 y.c 时需要 x.h，那么组件 y 展示了对组件 x 的**编译时**依赖。
- 如果对象文件 y.o（通过编译 y.c 产生）包含未定义的符号，为了这些符号，可能在连接时直接或间接地调用 x.o 来帮助解析这些符号，那么组件 y 展示了对组件 x 的一种**连接时依赖**。
- 若通过一个组件的逻辑接口，不能通过程序访问或探测到一个包含的实现细节（类型、数据或函数），则称这些实现细节被那个组件**封装**了。

第4章

- **回归测试**指的是这样的规程：运行一个程序（该程序被给定了一个有固定期望结果集的特定输入），比较其结果，以便验证程序从一个版本升级到另一个版本时能够继续执行期望的行为。
- **隔离测试**是指这样的规程：独立于系统的其它部分对单个组件或子系统进行测试。
- 一个可被赋予惟一的层次号的物理依赖图称为是**可层次化的**（levelizable）。

层次 0：一个在我们的软件包之外的组件。

层次 1：一个没有局部物理依赖的组件。

层次 *N*：一个组件，它在物理上依赖于层次 *N*–1 上（但不是更高层次上）的一个组件。

- 一个组件的层次是最长路径的长度，该路径是指从那个组件穿过（局部）组件依赖图到外部（或由编译器提供的）库组件的集合（可能为空）的路径。
- **分层次测试**是指在每个物理层次结构上测试单个组件的惯例。
- **增量式测试**是指这样的测试惯例：只特意测试真正在被测试组件中实现的功能。
- **白盒测试**是指通过查看组件的基础实现来检验一个组件的期望行为的惯例。
- **黑盒测试**是指只基于组件的规范（即不用查看其基础实现）来检验一个组件的期望行为的惯例。
- **累积组件依赖**（CCD）就是在一个子系统内的所有组件 C_i 之上，对每个组件 C_i 在增量式地测试时所需要的组件数量进行求和。（即为了增量式地测试子系统中所有组件所需的组件数量的总和。）
- **平均组件依赖**（ACD）定义为：一个子系统的 CCD 与系统中的组件数量 *N* 的比值：

$$\mathrm{ACD}(\text{子系统})=\frac{\mathrm{CCD}(\text{子系统})}{N_{\text{子系统}}}$$

- **标准累积组件依赖**（NCCD）定义为：包含 *N* 个组件的子系统的 CCD 值与相同大小的树形系统的 CCD 值的比值。

$$\mathrm{NCCD}(\text{子系统})=\frac{\mathrm{CCD}(\text{子系统})}{\mathrm{CCD}_{\text{平衡二叉树}}(N_{\text{子系统}})}$$

第5章

- 如果一个子系统可编译并且单个组件（包括.c 文件）的包含命令隐含的图是非循环的，则称这个子系统是**可层次化的**（levelizable）。
- 如果组件 y 处在比组件 x 更高的层次上，并且 y 在物理上依赖 x，则称组件 y **支配**（dominate）组件 x。
- 如果编译函数 f 的函数体时要求已经提前看到了类型 T 的定义，则称函数 f **实质**（**in size**）使用了类型 T。

- 如果编译函数 f 以及 f 可能依赖的任何组件时不要求已经提前看到了类型 T 的定义，则称函数 f **只在名称**上（**in name only**）使用了类型 T。
- 如果编译组件 c 时要求必须提前看到了类型 T 的定义，则称组件 c **实质**使用了类型 T。
- 如果编译组件 c 以及 c 可能依赖的任何组件时不要求已经提前看到了类型 T 的定义，则称组件 c **只在名称上**使用了类型 T。
- 在分层次系统中，**封装一个类型**（定义在头文件内的文件作用域中）意味着隐藏了它的使用（use）而不是隐藏了类型本身。

第 6 章

- 一个被包含的实现细节（类型、数据或函数），如果它们被修改、添加或删除时不会迫使客户程序重新编译，则称这样的实现细节被绝缘了。
- 满足下列条件的**抽象类**是一个**协议类**：

1. 既不包含也不继承那些包含成员数据、非虚拟函数或任何种类的私有（或保护的）成员的类；

2. 它有一个非内联虚析构函数（定义有一个空实现）；

3. 所有成员函数（除了包含被继承函数的析构函数）都被声明为纯虚的，并任其处于未定义状态。

- 一个具体类如果满足下列条件，就是完全绝缘的：

1. 只包含一个数据成员，它表面上是不透明的指针，该指针指向一个定义那个类的实现的 non-const struct（定义在.c 文件中）；

2. 不包含任何其它任何种类的私有的或保护的成员；

3. 不继承任何其它类；

4. 不声明任何虚拟的或内联的函数。

- 在本书中，**handle** 是一个类，它维持一个指向一个对象的指针，该对象可以通过 handle 类的公共接口编程访问。
- **轻量级（Light-weight）**是一个术语，其含义依赖应用它的上下文：

1. 不依赖于（许多）其它的组件；

2. 创建/析构都不昂贵；

3. 不分配额外的动态内存；

4. 有效地利用内联函数来访问/操纵嵌入数据。

第 7 章

- 一个**包（package）**就是被组织成一个物理内聚单位的组件集合。
- 如果包 x 中的一个或多个组件依赖（DependsOn）另一个包 y 中的一个或多个组件，

则称包 x **依赖**（**DependsOn**）包 y。

- **包群**（**package group**）就是一个**包**的集合，它被组织成一个物理上内聚的单位。
- 如果包群 g 中的一个或多个包**依赖**（**DependsOn**）另一个包群 h 中的一个或多个包，则称包群 g **依赖**（**DependsOn**）包群 h。
- 一个**层**（**layer**）对应于系统的一个给定层次上的所有包群。
- **补丁**是对前一次发布软件的局部修改，以修补组件内不完善或效率很低的功能。
- **启动时间**[也称为启用（invocation）时间]就是程序被初次调用和控制线程进入 main 之间的时间。

第 8 章

- **抽象**是完成一个共同目的的一组对象和相关行为的抽象规范。
- 如果有效实现定义在一个对象上的操作意味着可以直接访问该对象的私有部分，那么该操作是**基本的**（primitive）。

第 9 章

- **隐藏**（**hide**）：一个成员函数若使用了在一个基类或文件作用域中声明的某一函数的同样的名称，则**隐藏**了那个函数。

重载（**overload**）：一个函数用定义在同一作用域的同样的名称**重载**了另一个函数的名称。

覆盖（**override**）：一个成员函数**覆盖**了在一个基类中声明为虚拟的同样的函数。

重新定义（**redefine**）：一个函数的默认定义被另一个定义不可挽回地取代。

- 如果一个对象只有单一参数，该参数是一个指向那个对象的引用，并且若不经过显式的转换（cast）就不能从函数体内部获得一个指向同一个对象（或它的一部分）的非 const 引用，那么这个对象是 const **正确的**（**const-correct**）。
- 如果一个只采用 const 引用参数的函数（该引用参数指向系统内对象的任何对象子集）不能获得一个指向这些对象中的任何一个（或它们的任何部分）的可写引用，那么该系统是 const **正确的**。
- 如果一个系统的转换图未包含同时涉及任何一个类型的 const 和非 const 版本的循环，那么这个系统是 const **正确的**。

第 10 章

- 如果一个基本类型实例的大小能整除其地址值，那么它是**自然对齐的**。
- 一个聚集类型的实例，如果对对齐要求最严格的子类型的对齐地址能整除聚集的地址，那么它是**自然对齐**的。
- 如果一个和某对象状态相关的值有助于预期的语义（即，ADT 的基本行为），那么它

是逻辑值；否则就是物理值。

- 当程序失去了对动态分配的内存块进行释放的能力时会发生内存泄漏。
- **设计模式**是类或对象的抽象组织，它被反复证明在解决多种领域的相似问题上是有效的。

D.2 主要设计规则

第2章

- 保持类数据成员的私有性。
- 避免带外部连接的数据出现在文件作用域内。
- 在.h 文件的文件作用域内避免使用自由函数（运算符函数除外）；在.c 文件中避免使用带有外部连接的自由函数（包括运算符函数）。
- 在.h 文件的文件作用域内避免使用枚举类型、用户自定义类型和常量数据。
- 除非是作为包含卫哨，否则在头文件中应避免使用预处理宏。
- 只有类、结构、联合和自由运算符函数应该在.h 文件的文件作用域内**声明**；只有类、结构、联合和内联函数（成员或自由运算符）应该在.h 文件的文件作用域内**定义**。
- 在每个头文件的内容周围放置一个惟一的并且可预知的（predictable）（内部的）包含卫哨。

第3章

- 在一个组件内部声明的逻辑实体不应该在那个组件之外定义。
- 每个组件的.c 文件都应该将包含它自己的.h 文件的语句作为其代码的第一行有效语句。
- 在一个组件的.c 文件中，避免使用有外部连接并且没有在相应的.h 文件中明确声明的定义。
- 避免通过一个局部声明来访问另一组件中带有外部连接的定义；改为包含那个组件的.h 文件。

第7章

- 为每个全局标识符都附加上它的包前缀。
- 为每一个源文件名附加上它的包前缀。
- 避免包之间的循环依赖。
- 只有**定**义了 main 的.c 文件才有权重新定义全局 new 和 delete。
- 提供一种机制来释放分配给一个组件内的静态结构的任何动态内存。

D.3 次要设计规则

第2章

- 在每个头文件预处理器的包含命令周围，放置一个冗余（外部）包含卫哨。
- 使用一种一致的方法（例如，d_前缀）强调类成员函数。
- 使用一种一致的方法（例如，第一个字母大写）标识类型名称。
- 使用一种一致的方法（例如，所有的字母都大写以及下划线）标识不变的值，如枚举值、const 数据和预处理器常量。

第3章

- 组成一个组件的.c 文件和.h 文件的根名称应该严格匹配。

第9章

- 千万不要通过值传递一个用户自定义类型（例如 class、struct 或 union）给一个函数。
- 千万不要企图删除一个通过引用传递的对象。
- 在每一个声明了一个虚函数的（或派生于一个声明了虚函数的类的）类中，把析构函数显式地声明为类中的第一个虚函数，并且非内联地定义它。

第10章

- 避免在初始化过程依赖一个对象中数据成员被定义的顺序。
- 当为一个通用的、参数化的容器模板实现内存管理时，若赋值的目标是未初始化的内存，则要小心不要使用所包含类型的赋值运算符。

D.4 指导方针

第2章

- 给接口建立文档以便其他人可以使用；至少请另一个开发者检查每个接口。
- 明确地声明条件（在该条件下行为没有定义）。
- 标识符名称必须一致；使用大写字母或下划线（但不是同时用两种）来分隔标识符中的单词。
- 以相同方式使用的名称必须一致；特别是对于递归设计模式（recurring design patterns）

（例如迭代），要采用一致的方法名称和运算符。

第3章

- 客户应该包含直接提供了所需的类型定义的头文件；除了非私有继承，应避免依赖一个头文件去包含另一个头文件。
- 只有当组件 x 直接实质地使用了定义在 y 中的一个类或自由运算符函数时，x 才应该包含 y.h。
- 避免把（远距离的）友元关系授权给定义在另一个组件中一个逻辑实体。

第4章

- 避免组件之间的循环物理依赖。

第7章

- 通常，避免给予一个组件一种也可被其他组件获得的特权，否则将对作为一个整体的系统产生不利影响。
- 宁愿要模块而不要对象的非局部静态实例，尤其是在以下情况下：

1. 在一个编译单元之外需要对结构的直接访问。

2. 该结构在启动期间不需要或不是在启动之后马上需要，并且初始化结构本身的时间是显著的。

第9章

- 一个重载运算符的语义对客户应该是自然的、明显的和直观的。
- 用户自定义类型的重载运算符的语法属性应该反映已经为基本类型定义了的属性。
- 避免将一个基类函数隐藏在一个派生类中。
- 系统中的每一个对象都应该是 const 正确的。
- 一个系统应该是 const 正确的。
- 在抛弃 const 之前考虑（至少）两次。
- 对于返回一个错误状态的函数来说，整数值 0 应该总是意味着成功。
- 回答是或否问题的函数名称应该适当措辞（例如 isValid），并返回一个不是 0（“no”）就是 1（“yes”）的 int 值。
- 避免将从函数通过值返回的结果声明为 const。
- 避免那些需要一个未命名临时对象的结构的默认参数。
- 通过参数返回值要保持一致（例如，避免声明非 const 引用参数）。

- 避免将一个函数的任何参数的地址存储在函数结束后仍会保留的位置；改为传递该参数的地址。
- 无论何时一个形参通过引用或指针传递其实参给一个函数，如果该函数既不修改那个实参也不存储它的可写地址，那么那个形参就应该声明为 const。
- 避免将通过值传递给一个函数的形参声明为 const。
- 考虑将那些会激活可修改访问的形参（也许除了那些有默认实在参数的形参）放在那些通过值、const 引用或 const 指针来传递实在参数的形参之前。
- 避免给单独的函数授予友元关系。
- 如果函数体产生的对象代码大于同样的非内联函数调用本身所产生的对象代码，那么应该避免将该函数声明为 inline。
- 若你的编译器不会产生内联函数，那么应避免将函数声明为 inline。
- 避免在接口中使用 short；改为使用 int。
- 避免在接口中使用 unsigned 整数；改为使用 int 整数。
- 在接口中避免使用 long；改为 assert(sizeof(int) >= 4)并使用 int 或一个用户自定义的大整数类型。
- 考虑在接口中对于浮点类型只使用 double，除非有强制性的原因才使用 float 或 long double。
- 考虑避免“cast”运算符，尤其是基本整数类型，改为进行显式的转换。
- 为任何定义在头文件中的类显式地声明（公共地或私有地）构造函数和赋值运算符，甚至在默认的实现是充分的情况下。
- 在没有另外声明虚函数的类中，显式地把析构函数声明为非虚拟的并且适当地对它进行定义（内联的或非内联的）。

第 10 章

- 只有在已知这样做安全时，才能在实现中用 short 代替 int 作为优化措施。
- 即使在实现中也尽量不要考虑使用 unsigned。
- 在设计一个函数、组件、包或完整的系统时，使用最简单的有效技术。
- 避免允许通过程序对物理值进行访问。
- 调用一个 const 成员函数的结果不应该改变对象中的**任何**可编程访问的值。
- 特定对象的内存管理比特定类的内存管理要好。
- 使用一个非 const 指针数据成员来保存**被管理的**对象。
- 考虑提供一种方法在块分配和动态内存的单独分配之间进行切换。
- 当为一个完全通用的、参数化的容器类实现内存管理时（该容器类管理其包含对象的内存），假定参数化类型只定义了拷贝构造函数和析构函数——别无其他。

- 在任何可能之处，在分解的可重用 void * 指针类型的顶部，使用内联函数来实现模板，以重建类型的安全性。

D.5 原则

第 2 章

- assert 语句的使用有助于为用户实现编码时的假设建立文档。

第 3 章

- 逻辑设计只研究体系结构问题；物理设计研究组织问题。
- 一个组件就是设计的适当的基本单位。
- 通过确保一个组件自己分析自己的.h 文件——不要外部提供的声明和定义，可以避免潜在的使用错误。
- 一个编译时依赖几乎总是隐含一个连接时依赖。
- 组件的 DependsOn 关系是传递性的。
- 定义了某个函数的组件**通常**会物理依赖于定义了被那个函数使用的类型的组件。
- 如果一个组件定义了某个类，该类是一个（IsA）或有一个（HasA）用户自定义类型，那么那个定义类的组件总是会在编译时依赖那个定义类型的组件。
- 假如系统编译成功的话，仅凭由 C++预处理器#include 命令产生的包含图，就足以推断出系统内部的所有物理依赖。
- 一个组件内部的友元关系是那个组件的一个实现细节。
- 给定义在同一个组件内的类授予（局部的）友元关系不会破坏封装。
- 在同一组件中为容器类定义一个迭代器类可以在保持封装的同时，使组件具有用户可扩展性，改进可维护性和加强重用性。
- 对一个定义在系统的单独物理部分的逻辑实体，授权（远距离）友元关系，破坏了授予该友元关系的那个类的封装。
- 友元关系影响访问特权但不隐含依赖。

第 4 章

- 对于测试来说，软件中的一个**类**类似于现实世界中的**实例**。
- 对整个设计的层次结构进行分布式测试，比只对最高层接口进行测试高效得多（就每美元测试费用而言）。
- 独立测试减小了一部分与软件集成相关的风险。

- 隔离测试组件是确保可靠性的有效办法。
- 每个有向非循环图都可被赋予惟一的层次号；一个有循环的图则不能。
- 在大多数现实世界情况下，如果大型设计要被有效地测试，它们必须是可层次化的。
- 分层次测试需要为每个组件提供一个独立的测试驱动程序。
- 只测试在一个组件中**直接实现的**功能，能够使测试的复杂性与组件的复杂性相当。
- 完全的回归测试是昂贵的但也是必需的。建立彻底的回归测试的适当时间，与要测试的子系统的稳定性密切相关。
- 组件之间的循环物理依赖抑制了理解、测试和重用。
- N 代表系统中组件的数量。

$$\text{CCD}_{\text{循环依赖图}}(N) = (\text{组件的总数}) \cdot (\text{测试一个组件的连接时开销}) = N \cdot N = N^2$$

- 非循环物理依赖可以明显减少与开发、维护和测试大型系统相关的连接时开销。
- CCD 的主要用途是，对一个给定体系结构的较小改动引起的整个耦合结构的变化进行量化。
- 使给定组件集合的 CCD 最小化是一个设计目标。

第 5 章

- 允许两个组件经由#include 命令彼此“知道”隐含了循环物理依赖。
- 相关抽象接口中的内在耦合使它们更难被层次分解。
- 如果同层次的组件是循环依赖的，那么就有可能把互相依赖的功能从每一个组件升级为一个潜在的新的更高层次的组件（依赖于每一个初始的组件）的静态成员。
- 在大型的、低层次子系统中的循环物理依赖最容易增加维护系统的整体开销。
- 如果同层次的组件是循环依赖的，那么就有可能把互相依赖的功能从每一个组件降到一个潜在的新的较低级（共享）组件中，每一个原来的组件都依赖于这个新组件。
- 将共有代码降级可促成独立重用。
- 可以将升级策略与基础设施降级结合起来，以增强独立重用。
- 把一个具体的类分解为两个包含更高和更低层次功能的类，可以促进层次化。
- 将一个抽象的基类分解成两个类——一个定义一个纯粹的接口，另一个定义它的部分的实现——可以促进层次化。
- 把一个系统分解成更小的组件，既可以使它更灵活，也可以使它更复杂，因为现在要处理更多的物理部件了。
- 只在名称上使用了对象的组件可以独立于被命名的对象而被彻底测试。
- 如果一个被包含的对象拥有一个指向它的容器的指针，并且要实现实质地依赖那个容器的功能，那么我们可以通过以下方法来消除相互依赖：（1）让被包含类中的指针不透明；（2）在被包含类的公共接口上提供对容器指针的访问；（3）将被包含类的受影

响的方法升级为容器类的静态成员。

- 哑数据可以用来打破 **in-name-only** 依赖、促进易测试性和减少实现的大小。但是，不透明指针可以同时保持类型安全和封装；而哑数据通常是不能的。
- 与一些重用形式相关的额外耦合可能会超过从该重用获得的利益。
- 提供少量的冗余数据可以使一个对象的使用只在名称上，从而消除连接到那个对象类型的定义的开销。
- 将子系统打包，使其连接到其它子系统的开销最小化，这是一个设计目标。
- 不加选择地使用回调可能导致难以理解、调试和维护的设计。
- 对回调的需求，可能是糟糕的整体体系结构的一个症状。
- 建立较低层次对象的分等级的所有权，可以使一个系统更容易理解和更可维护。
- 将独立可测试实现细节分解出来并降级，能够减少维护一个循环依赖类集合的开销。
- 在循环物理依赖不可避免的地方，将其升级到尽可能高的层次可减少 CCD，甚至可以使循环能够被一个单个的、大小便于管理的组件代替。
- 授权友元关系不会产生依赖，但是为了保持封装可能会引起物理耦合。
- 什么是和什么不是实现细节，取决于物理层次结构内部的抽象级别。
- 将封装所在的层次升级，能够消除对一个子系统内协同操作的组件授予私有访问权的需求。
- 私有的头文件不是封装的适当替代品，因为它们禁止并排（side-by-side）重用。
- 包装器组件可以用来封装一个子系统内的实现类型的使用，但允许其它类型通过它的接口。

第 6 章

- 以一个基类的保护成员的形式为派生类的作者提供支持，会将派生类的未绝缘的实现细节暴露给基类的公共客户。
- 给较高层次的客户程序授予修改较低层次共享资源的接口的权利，会隐含地耦合所有的客户程序。
- 一个协议类几乎是一个完美的绝缘器。
- 一个协议类可以用来消除编译时依赖。
- 只保留一个指向包含一个类的所有私有成员的结构的不透明指针，会使一个具体的类能够将其客户程序与其实现绝缘。
- **所有**完全绝缘的类的物理结构外表上都是一样的。
- 所有完全绝缘的实现都可以在不影响任何头文件的情况下进行修改。
- 没有什么办法可以从一个组件的外部通过编程来确定一个组件是否为一个包装器。
- 一个定义在 A 包装器组件中的类型，无论什么时候被传递进一个定义在 B 包装器组件

中的类型中，B 组件将不能访问基础的被包装的实现对象，而只能访问包装器的公共功能。

- 在一个过程接口中，使客户程序只显式地析构那些他们显式创建的对象，这样可以减少所有权关系上的混乱，并且可以提高性能。
- 对于一个组件的实现不进行绝缘的决策可能是基于该组件不是很广泛地使用的认识。
- 除非性能已知不是一个问题，否则避免用极小的访问函数（它们在整个系统中广泛使用）对低层次类的实现进行绝缘可能是明智的。
- 绝缘轻量级的、广泛使用的并且通常通过值返回的对象，会显著地降低整个运行时性能。
- 对于大型的、广泛使用的对象来说，要尽早进行绝缘，如果有必要，以后可以有选择性地删除该绝缘。
- 有时整体绝缘的运行时开销不会比部分绝缘大。
- 有时获得最后百分之十的绝缘要以增加十倍的运行时间为代价。

第 7 章

- 前缀的主要用途是惟一地标识组件或类在其中定义的物理包。
- 理想状态是，包前缀不仅表示组件或类被定义在其中的物理库，还将隐含着内聚的逻辑特征和组织特征。
- 给被一个多包子系统的客户直接使用的组件子集指定一个单一包前缀并不是一定可能的。
- 在把一个新组件加入到包中时，这个组件的逻辑特性和物理特性都应该考虑。
- 最小化输出头文件的数量和大小，可以提高可用性。
- 在一个包群内将协议降级和将包装器升级，有助于避免输出（展示）包和非输出（实现）包之间的循环依赖。
- 将源代码修改之后重编译所需的时间最小化，可以显著地减少开发开销。
- 补丁一定不能影响任何已存在对象的内部布局。
- 从一个定义了 main 的编译单元中分解出独立可测试的和潜在可重用的功能，本质上能够使程序的整个实现在一个更大型的程序中重用。
- 程序中的每一个非局部静态对象的结构都潜在地会增加调用时间。
- 初始化你不另外直接依赖的组件会显著地增加 CCD。

第 8 章

- 一个**类**是一个 ADT 的具体规范；一个**组件**是一个抽象的具体规范。
- 私有接口应该是充分的。

公共接口应该是完整的。
类接口应该是基本的（primitive）。
组件接口应该是最小和便于使用的。

- 在任何可行的地方，延缓不必要功能的实现可以降低开发和维护成本，并且可以避免过早地进行精确的接口和行为设计。
- 让功能保持在一个可行的最小范围内可以增强可用性和可重用性。
- 在一个组件接口中尽可能少地使用外部定义类型，可以促进在更多情况下的重用。
- 一个对于封装的良好的测试，是要看一个给定的接口是否无需做任何改变就可同时支持两种显著不同的实现策略。
- 一个完全封装的接口可能会为给定的实现方法带来很大的性能负担。
- 传递进一个已构造对象的地址以接受返回值（称为**参数返回**），能在保持整体封装的同时提高性能。
- 接受不太完全的封装有时是正确的选择。

第9章

- 可读性（胜过易用性）应该是使用运算符重载的主要原因。
- 让用户自定义运算符的语法属性模仿预先确定的 C++运算符，以避免意外并使它们的使用更可预知。
- C++语言本身可作为用户自定义运算符模仿的一个客观和适宜的标准。
- 在重载运算符中的不一致问题，对客户来说可能是明显的、讨厌的和高代价的。
- 语法问题，例如双目运算符的对称隐式转换，不必为了获得多态行为而让步。
- 虚函数实现行为上的变化；数据成员实现值的变化。
- 静态成员函数通常用于实现一个单独工具类中的非基本功能。
- 从一个 const 成员函数返回一个非 const 对象可能会破坏一个系统的 const 正确性。
- 如果成员函数不是公共的，那么会使普通用户也要接触未绝缘的实现细节。
- 所有的虚函数和保护函数都是要由派生类作者考虑的事项。
- 通常函数成功工作只有一种方式，而它失败却有若干种方式；作为客户，我们可以不关心它为什么失败。
- 通过加载一个可修改的句柄参数返回一个动态分配对象，比通过非 const 指针返回那个对象更不容易产生内存泄漏。
- 默认参数可以是函数重载的一种有效选择，尤其是在绝缘未涉及到的地方。
- 避免不必要的友元关系（甚至在同一组件内部）可以提高可维护性。
- 在代码中直接显式地说明设计决策（而不是依赖注释）是一个设计目标；设计能安全使用和容易维护的健壮接口偶尔会与这个目标形成竞争关系。

- 有时注释比直接在代码中表达一个接口决策要更好。
- 在实际出现的大多数情况下，为了在接口中能表达整数和浮点数，所需要的惟一基本类型分别是 int 和 double。
- 能激活隐式转换的构造函数，尤其是从广泛使用的类型或基本类型（如 int）的转换，会破坏由强类型所提供的安全性。

第 10 章

- 数据成员声明的次序能够影响对象的大小。
- 在实现中使用 unsigned 类型以“提高一点性能”，是基本的整数类型没有大到足够安全的标志。
- 自然包括其边界条件的算法，通常比将边界条件作为特殊情况处理的算法更简单、更短小、更易于理解和测试。
- 通过增加一个额外的间接层能解决各种问题。
- 在一个组件中分解出一般可重用的功能，可以减小代码长度并提高可靠性，而运行时性能的损失不多。
- 对一个完全封装的接口来说，每个可编程访问的值都是一个逻辑值。
- **提示**是只写的。
- 最好的提示不直接和特定的实现相联系。
- 如果一个支持值语义的类型有两个实例，它们各自所有的逻辑值都相等，那么这两个实例是相等的（==）；如果它们各自的任何一个单个的逻辑值不相等，那么这两个实例是不相等的（!=）。
- 将全局运算符 new 和 delete 工具化，是在系统中理解和测试动态内存行为的简单但有效的方法。
- 当利用全局的 new 和 delete 时，使用 iostream 会引起令人不快的负作用。
- 从不返还其内存的特定类分配方案，使得对内存泄漏的自动检测变得更加困难。
- 特定类的内存分配器倾向于占用全局分配的内存，因此会增加了整个内存的使用。
- 对特定类的内存管理方法不加选择地使用，是一种以自我为中心的表现，会对一个集成系统的整体性能产生负面的影响。
- 一个特定对象的内存分配方案有足够的上下文可以知道什么时候不再需要一些实例的子集而可以释放它们（这些实例子集是分配给特定对象并由它来管理的）。
- 如果能够利用有关特定用户使用模式的知识，我们通常可以为其管理的对象编写更有效的分配程序。
- 在一个伪参数化类型中嵌入实际参数类型（如，将 gen_StackItem<T>嵌入到 gen_Stack<T>），可以让我们在继承方面像处理用户自定义类型那样处理基本类型，并且允许它

有寻址和分配功能，这些功能在所有用户自定义类型中是没有的。

- 一般来说，一个对象不能使用位拷贝来拷贝或移动。
- 一般来说，不能使用对象的赋值运算符拷贝或移动该对象到未初始化的内存。
- 设计模式是在体系结构层次上交流可重用的概念和思想的有效方法。
- 设计模式，就像设计过程一样，能处理逻辑和物理问题。

参考文献

aho

Aho, Alfred V., John E. Hopcraft, and Jeffrey D. Ullman, 1974,
The Design and Analysis of Computer Algorithms,
Addison-Wesley, Reading, Massachusetts.

aho83

Aho, Alfred V., John E. Hopcraft, and Jeffrey D. Ullman, 1983,
Data Structures and Algorithms,
Addison-Wesley, Reading, Massachusetts.

booch

Booch, Grady, 1994,
Object-Oriented Analysis and Design with Applications,
Second Edition,
Benjamin/Cummings, Redwood City, California.

cargill

Cargill, Tom, 1992,
C++ Programming Style,
Addison-Wesley, Reading, Massachusetts.

ellis

Ellis, Margaret, A. and Bjarne Stroustrup, 1990,
The Annotated C++ Reference Manual,
Addison-Wesley, Reading, Massachusetts.

gamma

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides, 1995,
Design Patterns,
Elements of Reusable Object-Oriented Software,
Addison-Wesley, Reading, Massachusetts.

marick

Marick, Brian, 1995,
The Craft of Software Testing,
Subsystem Testing Including Object-Based and Object-Oriented Testing,
Prentice Hall, Englewood Cliffs, New Jersey.

meyer

Meyer, Bertrand, 1988,

Object-oriented Software Construction,
Prentice Hall, Englewood Cliffs, New Jersey.

meyers

Meyers, Scott, 1992.
Effective C++,
50 Specific Ways to Improve Your Programs and Designs,
Addison-Wesley, Reading, Massachusetts.

murray

Murray, Robert, B. 1993,
C++ Strategies and Tactics,
Addison-Wesley, Reading, Massachusetts.

musser

Musser, David R. and Atul Saini, 1996,
STL Tutorial and Reference Guide,
C++ Programming with the Standard Template Library,
Addison-Wesley, Reading, Massachusetts.

perry

Perry, Dewayne E. and Gail E. Kaiser, 1990,
Adequate Testing and Object-Oriented Programming,
SIGS Publications, New York, New York.

plauger

Plauger, P. J., 1992,
The Standard C Library,
Prentice Hall, Englewood Cliffs, New Jersey.

sommerville

Sommerville, Ian, 1992,
Software Engineering,
Fourth Edition,
Addison-Wesley, Reading, Massachusetts.

soukup

Soukup, Jiri, 1994,
Taming C++,
Pattern Classes and Persistence for Large Projects,
Addison-Wesley, Reading, Massachusetts.

stroustrup

Stroustrup, Bjarne, 1991,
The C++ programming Language,
Second Edition,
Addison-Wesley, Reading, Massachusetts.

stroustrup94

Stroustrup, Bjarne, 1994,
The Design and Evolution of C++,
Addison-Wesley, Reading, Massachusetts.