软件工程技术丛书

面向模式的软件体系结构

卷2：用于并发和网络化对象的模式

Pattern-Oriented Software Architecture

Volume 2 Patterns for Concurrent and Networked Objects

Douglas Schmidt
Michael Stal
Hans Rohnert
Frank Buschmann
著

张志祥 任雄伟 肖斌 等译 贲可荣 审校

机械工业出版社
China Machine Press

Pattern-Oriented Software Architecture:
Volume 2 Patterns for Concurrent and Networked Objects

面向模式的软件体系结构
卷2：用于并发和网络化对象的模式

对于软件开发人员来说，设计运行于并发和网络化环境中的应用程序和中间件是很大的挑战。本书中所列出的模式构成了处理有关并发和网络化问题的模式语言的基础。

书中提出了17种相互关联的模式，它们包含用于构建并发和网络化系统的核心组件：服务访问和配置、事件处理、同步和并发。在多种程序设计语言（如C++、Java和C）中，这些模式有大量的例子和已知应用。

本书可用于解决特定的软件开发问题，读者也可以从头至尾地阅读，学习构建并发和网络化应用以及中间件的最佳方法。

欢迎访问本书网页：
http://www.cs.wustl.edu/~schmidt/POSA

关于作者：

本书由负责《面向模式的软件体系结构 卷1：模式系统》的三位获奖作者和加利福尼亚大学欧文分校（UCI）的Douglas Schmidt共同撰写。

封面制作 锡彬

ISBN 7-111-11686-0
9 787111 116868

华章图书

网上购书：www.china-pub.com
北京市西城区百万庄南街1号 100037
读者服务热线：(010) 68995259, 68995264
读者服务信箱：hzedu@hzbook.com
http://www.hzbook.com
ISBN 7-111-11686-0/TP · 2823
定价：59.00 元

软件工程技术丛书

设计系列

A1084808

TP311.5
25567

面向模式的软件体系结构

卷2：用于并发和网络化对象的模式

Pattern-Oriented Software Architecture

Volume 2 Patterns for Concurrent and Networked Objects

Douglas Schmidt
Michael Stal
Hans Rohnert
Frank Buschmann
著

张志祥 任雄伟 肖斌 等译 贲可荣 审校

机械工业出版社
China Machine Press

H/JS72/09

本书讲述用于创建并发和网络化系统的模式，共涉及17种模式与惯用法。这些模式构成了用于解决有关分布式、并发和网络化问题的模式语言的基础。本书的模式是创建并发和网络化系统的核心元素，包括服务访问、事件处理、并发控制、连接管理和初始化、事务、安全性等。

本书强调实际的解决方案，读者可以使用本书的方法解决特定的软件开发问题，获得对构造分布式及并发应用和中间件最佳实践的基本理解。书中含有用多种语言（C、C++和Java）编写的大量例子和已知使用，适于专业软件开发人员及计算机专业高校师生使用。

Douglas Schmidt, Michael Stal, Hans Rohnert & Frank Buschmann: Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects (ISBN: 0-471-60695-2).

Authorized translation from the English language edition published by John Wiley & Sons, Inc.
Copyright © 2000 by John Wiley & Sons , Ltd.
All rights reserved.

本书中文简体字版由约翰-威利父子公司授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。
版权所有，侵权必究。

本书版权登记号：图字：01-2002-0817

图书在版编目（CIP）数据

面向模式的软件体系结构 卷2：用于并发和网络化对象的模式 / 施密特（Schmidt, D.）等著；张志祥等译. -北京：机械工业出版社，2003.2
(软件工程技术丛书　设计系列)
书名原文：Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects.
ISBN 7-111-11686-0

I. 面…　II. ① 施…　② 张…　III. 软件工程-系统结构　IV. TP311.5

中国版本图书馆CIP 数据核字（2003）第010175号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）
责任编辑：张金梅
北京昌平奔腾印刷厂印刷・新华书店北京发行所发行
2003年8月第1版第1次印刷
787mm × 1092mm 1/16・26.75印张
印数：0 001-5 000册
定价：59.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换
本社购书热线电话（010）68326294

服务访问和配置模式

包装器外观（Wrapper Facade）（第28页）设计模式把现有的非面向对象的API所提供的函数和数据，封装在更加简洁的、健壮的、可移植的、可维护的和内聚的面向对象的类接口中。

组件配置器（Component Configurator）（第46页）设计模式允许应用程序在不必修改、重新编译、静态地重新链接应用程序的情况下，在运行时链接和解链它的组件实现。组件配置器进一步支持在不必关闭和重启运行的进程的情况下，把组件重新配置到不同的应用程序进程。

截取器（Interceptor）（第66页）体系结构模式允许透明地把服务加到框架中，并且在某些事件发生时，能自动地触发服务。

扩展接口（Extension Interface）（第85页）设计模式允许组件导出多个接口，当开发人员扩展或修改组件的功能时，此模式能防止接口的膨胀和客户机代码的破坏。

事件处理模式

反应器（Reactor）（第108页）体系结构模式，使事件驱动的应用可以多路分解并分配从一个或者多个客户机发送给应用的服务请求。

主动器（Proactor）（第130页）体系结构模式使事件驱动的应用程序能有效地多路分解和分配由于异步操作的完成而激活的服务请求，这样在不存在不利条件时能获得并发所带来的好处。

异步完成标记（Asynchronous Completion Token, ACT）（第158页）设计模式可以使应用程序能有效地多路分解和处理对调用服务的异步操作的响应。

接受器-连接器（Acceptor-Connector）（第173页）设计模式将网络化系统中合作的对等体服务的连接和初始化工作，与该服务在连接和初始化之后所执行的处理分开。

同步模式

定界加锁（Scoped Locking）（第199页）是C++惯用法，能确保当控制进入到某一范围时，自动获得锁，而当控制离开该范围时，自动释放锁，不管从该范围返回的路径是什么。

策略化加锁（strategized locking）（第205页）设计模式把同步机制参数化，这一机制保护组件的临界区免受并发访问。

线程安全接口（Thread-Safe Interface）（第212页）模式将加锁开销减至最小，并能保证组件间的方法调用不会因为想再次获得一个已经被组件拥有的锁而导致“自死锁”。

双检查加锁优化（Double-Checked Locking Optimization）（第217页）设计模式，如果代码的临界区必须在程序执行期内只以线程安全的方式获得一次锁时，该模式能够减少争用和同步开销。

并发模式

主动对象（Active Object）（第226页）设计模式将方法执行和方法调用分离，加强并发和简化对驻留在自身控制线程中对象的同步访问。

监视器对象（Monitor Object）（第245页）设计模式使并发方法的执行同步化，以确保任一时刻仅有一个方法在对象内运行。它也允许对象方法相互协调，调度方法的执行顺序。

半同步/半异步（Half-Sync/Half-Async）（第261页）体系结构模式将并发系统中的异步和同步服务处理分离，简化了编程，同时又没有降低性能。该模式介绍了两个通信层，一个用于异步服务处理，另一个用于同步服务处理。

领导者/追随者（Leader/Followers）（第273页）体系结构模式提供一个高效的并发模型。在该模型中，为了检测、多路分解、分配和处理事件源上引发的服务请求，多线程轮流共享一个事件源集合。

线程特定的存储器（Thread-Specific Storage）（第289页）设计模式允许多线程使用一个“逻辑上全局的”访问点获得一个局限于某一线程的对象，而不会导致对象访问中的加锁开销。

译者序

模式系统包括很多通用的模式，而本书集中深入地讨论用于并发和网络化的通用的、与应用领域无关的模式。这些模式是对《面向模式的软件体系结构 卷1：模式系统》一书中介绍的通用模式在两个软件开发日益重要的领域上的补充。

除了关注用于并发和网络化的与领域无关的通用模式之外，本书对已有的关于并发网络程序设计和面向对象设计的资料做了如下扩充：(1) 关于并发的网络程序设计的资料通常关注操作系统API的语法和语义。相反，本书介绍了如何在设计和实现高质量的并发和网络化的系统时有效地使用这些API。(2) 本书讨论了并发和网络应用程序的开发过程。(3) 本书中的模式并不仅仅构成一个模式分类或系统，它们相互补充，形成了用于并发和网络化软件的模式语言。本书描述了如何使用这些模式语言来构建成熟的并发和网络化软件系统和应用程序、Web服务和分布式对象计算中间件，以及底层的操作系统网络协议和机制。

本书中介绍的很多模式还可用于并发和网络化的语境之外。

本书对软件开发的初学者和专家都有帮助。它可以帮助初学者不需要积累多年的经验，就可在适当规模的项目上像专家一样工作，它可以帮助专家在大型复杂的软件设计上利用已定义属性，也可以促使他们学习其他专家的经验。

本书力图做到既是教材又是参考手册，帮助软件开发人员以一种新的方式思考软件体系结构，并提供一些技术来解决特定的不断再现的设计问题。把本书用做软件工程课程的指南，可以给学生提供大型软件设计的完整的新观点。把本书用做参考手册，可以方便查找全面的技术并随时可以使用。本书包含了涉及模式实际应用的许多指导原则和强制限制。

本书第1章、第2章由任雄伟翻译，第5~8章由肖斌翻译，其余章节由张志祥翻译。全书由贲可荣审校。孙宁、王淑雪、朱继梅等参与了本书的部分工作，在此表示感谢。

由于各种原因，译稿难免存在错误和疏漏，欢迎读者批评指正。

本书可作为计算机专业高年级本科生、计算机专业研究生的软件工程教材或参考书，亦可作为软件开发人员的参考手册。

译　者

2003年3月

译者简介

张志祥　男，1967年10月生，湖北省天门市人。现任海军工程大学副教授。

1988年和1991年在海军工程大学计算机系分别获工学学士、工学硕士学位。1991年3月至今在海军工程大学计算机系任教。2000年9月起在华中科技大学计算机学院攻读博士学位。主要研究方向：语义Web、程序设计方法。

先后参与多项军内科研项目的研制，多项成果获军队科技进步奖。在《计算机研究与发展》等刊物发表论文多篇。译著包括《净室软件工程——技术与过程》（电子工业出版社，2001年6月），《Oracle 9i宝典》（电子工业出版社，2002）等。

任雄伟　男，1970年1月生，湖北省武汉市人。现任海军工程大学讲师。

1992年在解放军通信工程学院无线通信工程专业获工学学士学位，1997年在海军电子工程学院通信与电子系统专业获工学硕士学位。1998年3月至今在海军工程大学计算机系任教。2000年9月起在华中科技大学电信系攻读博士学位。主要研究方向：多媒体通信、计算机网络。

先后参与“编队电子战训练模拟器”和“舰载电子战作战效能评估”等项目的研制。在《电视技术》、《计算机应用》等刊物发表论文多篇。

贲可荣　男，1963年8月生，江苏省海安县人。现任海军工程大学教授。

1983年在苏州大学数学系获理学学士学位，1986年在南京大学数学系获理学硕士学位。1986年8月至1990年3月在海军工程大学计算机系任教。1994年6月在国防科技大学获工学博士学位。博士期间，由国防科技大学计算机学院陈火旺院士指导，主修计算机软件。1994年12月任海军工程大学计算机系副教授。1995年起担任计算机应用技术专业硕士生导师、海军工程大学学位评定委员会委员。经教育部批准，2000年3月至2001年3月任武汉大学软件工程国家重点实验室访问学者；2000年被教育部确定为海军首批十名骨干教师之一。2000年12月晋升为教授。

主要译著有《净室软件工程——技术与过程》（电子工业出版社，2001年6月），《能力成熟度模型(CMM)：软件过程改进指南》（电子工业出版社，2001年7月），《基于项目的软件工程——面向对象研究方法》（机械工业出版社，2002年6月），《面向模式的软件体系结构卷1：模式系统》（机械工业出版社，2003年1月），审校《实用软件测试指南》（电子工业出版社，2003年1月），参加《计算机科学技术百科全书》（清华大学出版社，1998）的编写。先后承担国家自然科学基金项目、国家863项目和军队科研等五个项目。在“中国科学”、“软件学报”、“计算机科学”等刊物和会议发表论文50余篇。成果获军队科技进步奖。主要研究方向：软件可靠性、软件质量保证技术、形式化方法等。兼任中国计算机学会计算机理论专业委员会委员、中国造船工程学会电子技术学术委员会委员，《海军工程大学学报》、《舰船电子工程》等刊物编委。

译者序

模式系统包括很多通用的模式，而本书集中深入地讨论用于并发和网络化的通用的、与应用领域无关的模式。这些模式是对《面向模式的软件体系结构 卷1：模式系统》一书中介绍的通用模式在两个软件开发日益重要的领域上的补充。

除了关注用于并发和网络化的与领域无关的通用模式之外，本书对已有的关于并发网络程序设计和面向对象设计的资料做了如下扩充：(1) 关于并发的网络程序设计的资料通常关注操作系统API的语法和语义。相反，本书介绍了如何在设计和实现高质量的并发和网络化的系统时有效地使用这些API。(2) 本书讨论了并发和网络应用程序的开发过程。(3) 本书中的模式并不仅仅构成一个模式分类或系统，它们相互补充，形成了用于并发和网络化软件的模式语言。本书描述了如何使用这些模式语言来构建成熟的并发和网络化软件系统和应用程序、Web服务和分布式对象计算中间件，以及底层的操作系统网络协议和机制。

本书中介绍的很多模式还可用于并发和网络化的语境之外。

本书对软件开发的初学者和专家都有帮助。它可以帮助初学者不需要积累多年的经验，就可在适当规模的项目上像专家一样工作，它可以帮助专家在大型复杂的软件设计上利用已定义属性，也可以促使他们学习其他专家的经验。

本书力图做到既是教材又是参考手册，帮助软件开发人员以一种新的方式思考软件体系结构，并提供一些技术来解决特定的不断再现的设计问题。把本书用做软件工程课程的指南，可以给学生提供大型软件设计的完整的新观点。把本书用做参考手册，可以方便查找全面的技术并随时可以使用。本书包含了涉及模式实际应用的许多指导原则和强制限制。

本书第1章、第2章由任雄伟翻译，第5~8章由肖斌翻译，其余章节由张志祥翻译。全书由贲可荣审校。孙宁、王淑雪、朱继梅等参与了本书的部分工作，在此表示感谢。

由于各种原因，译稿难免存在错误和疏漏，欢迎读者批评指正。

本书可作为计算机专业高年级本科生、计算机专业研究生的软件工程教材或参考书，亦可作为软件开发人员的参考手册。

译　者

2003年3月

前　言

中间件是一组服务、协议和支撑工具，提供了构建现代分布式系统和应用程序的“管道”，而这些系统和应用是支持Web服务、分布式对象、协作应用程序、电子商务系统以及其他重要平台的基础设施。以前，很少听到中间件这个词，从事中间件开发的人更少。但是在过去的十年间，这个名词、有关它的研究和实践以及它的影响无处不在。但是，到目前为止，还没有一本书介绍如何构建网络化和并发的面向对象（OO）中间件，所以中间件设计还仍然像魔法一样神秘。本书阐明中间件的构建，希望能像专家一样，对于一般的设计问题、强制条件、成功的解决方案以及结论，为你提供合理的和带有经验指导的说明。

和大多数概念一样，确定中间件的范畴是很困难的。通常，中间件由构建系统和应用程序所需要的软件组成，但它并不是操作系统内核的固有部分。不过，中间件不太容易一眼就识别出来，它可能出现在库和框架、操作系统及其附件、Java虚拟机以及其他运行时系统中，也可能出现在大粒度软件组件中和像Web服务这样的部分终端产品中。

本书不是一本笼统介绍中间件，或者描述哪类应用程序和分布式系统体系结构可用中间件构造的教科书。相反，本书提出了一种模式语言，这种语言记录了构建大多数中间件具有的面向对象通信支持所用到的设计步骤。本书介绍的很多模式对于不是直接基于中间件的各层的系统和应用程序也是有用的。

本书强调实际的解决方案，而不是形式化理论。这里介绍的很多模式背后的基本思想对有经验的系统开发人员来说是熟知的——例如分配、多路分解、回调和配置，有时候是更一般的面向对象模式的变体——例如代理、适配器和外观。本书的主要贡献是基于这些思想的彻底的工程化解决方案。中间件开发人员必须解决范围很广的强制条件，包括吞吐量、响应能力、依赖性、互操作性、可移植性、可扩展性以及对遗留软件的适应等等。这些强制条件的多样性和重要性说明了中间件模式的复杂性，这一点和在较小规模的面向对象应用程序和并发程序设计中看到的不一样。

这些强制条件的多样性，以及多年的工程实践，通常产生了很多设计考虑和工程化的权衡方案，它们将思想和思想在中间件框架中的表示分开。本书所使用的模式描述格式将解决方案表示成一组具体的设计步骤，从而简化这一分开的过程。很多步骤依次调用了其他的模式。将这些综合在一起，就形成一种模式语言，开发人员在设计服务和应用程序时可以从模式转到模式。

正如作者所提到的，本书所讨论的一些思想和技术是对W. Richard Stevens那些关于网络编程的开拓性著作（如[Ste98]）所阐述思想的补充。二者主要差别在于，本书更加关注更高层次的设计问题。例如，在讨论UNIX `select()`调用的输入和输出时，本书解释如何基于`select()`和其他操作系统调用，构建可组合的和灵活的框架，如反应器。

本书的一个隐含的主题是，如何应用由当前流行的平台所提供的处理I/O、线程、同步

和事件多路分解的功能作为基础，构建更高层框架和组件。强调在UNIX和微软操作系统中使用C/C++不会减弱这一主题。而在如下情况下，Java程序员会发现一些小小的不连贯：一是Java已经直接实现了一些本书中讨论的模式（如定界加锁），或者Java已经提供了按照某些模式的特殊实现而构造的框架（如对可配置组件的JavaBeans框架支持），以及一些Java不能访问底层系统机制（如同步事件多路分解）。

但是，熟悉Java、Smalltalk以及其他面向对象程序设计语言的读者将会受惠于模式所表达的中心思想，能更好地理解如何以及为什么有一些模式已经在语言功能和库中直接得到了支持，并可能基于其他模式建立有用的组件。例如，Java一直不提供对异步I/O有用的系统结构的访问，直到有了java.nio。不过在引用了本书中对主动器模式的一段描述后，我整理了一个Java版本，它用简单的spin-loop线程在多个通道上检测I/O的可用性，来模拟多路分解步骤。这样做效率不高，但对于它的使用语境已经足够了。

近几年来，本书中介绍的一些模式，如反应器模式，已经从设计发明的描述进化为设计模式的描述。每个正在实现可移植的面向对象中间件的人，都已经写了或者使用了至少一个包装器外观。本书对以往其他几种模式的讨论还涉及其设计的一些新作用。起初不能肯定是否应该将这些描述看成是模式，模式应该是被时间证明的、独立（重新）发现的解决方案。不过，随着时间的推移，本书作者和面向对象中间件业界越来越坚信本书中介绍的模式确实抓住了关键强制条件和设计问题的本质，并已经看到所介绍的解决方案在很多不同的使用语境中被反复用到。

我建议你分享这一现象。通过阅读——特别是使用——本书中的内容，你将会明白为什么像反应器和主动器这样的模式名称在面向对象中间件开发人员之间很普及，就像装饰器和观察者在面向对象GUI开发者中很普遍一样。

Doug Lea
纽约州立大学Oswego分校

内容简介

模式已经引起了软件开发业界的重视。自从第一本开创性的著作《设计模式——可复用面向对象软件的基础》[GoF95]问世以后，软件开发人员对模式产生了很高的热情。随后的一些工作，如“程序设计的模式语言（Pattern Languages of Program Design, PLoPD）”丛书[PLoPD1][PLoPD2] [PLoPD3] [PLoPD4]以及《面向模式的软件体系结构，卷1：模式系统》[POSA1]⊖ 进一步激起了人们对模式的高度兴趣，这些兴趣最早是由更早期的对软件惯用法[Cope92]、构造建筑的模式[Ale79][AIS77]以及文化人类学模式[Bat79]的工作引起的。

本书是“面向模式软件体系结构（Pattern-Oriented Software Architecture, POSA）”系列丛书的第2卷。和第1卷《模式系统》⊖ [POSA1]一样，本书介绍了模式和最佳实践，这些模式和实践代表了用于构建产业化软件系统的具体的、经过证明了的有用技术。这些模式和最佳实践能够并且已经应用于广泛的领域，包括电信和数据通信、金融服务、医学工程、航天、制造过程控制以及科学计算。同时，它们也成为流行的分布式对象计算中间件的基础，这些中间件如CORBA[OMG98c]、COM+[Box97]、Java RMI[WRW96]和Jini[Sun99a]。

而且，本书中的所有模式和卷1都建立在同样坚实的概念基础之上。例如，它们使用同样的模式分类大纲、同样的模式描述格式、用多种编程语言（如C++、Java和C）表示实例和已有的使用。因此，本书遵循和《模式系统》一样的体系，有着同样的表现形式。

《模式系统》包括很多通用模式，与此相反，本书的内容比较集中，只讨论并发和网络化。本书中的所有模式集中在这两个领域，这样可以更深入地讨论与并发和网络化有关的问题，涉及其他不相关领域的模式时，这种讨论就不会太深入。因此，本书中的模式是对《模式系统》一书中介绍的通用模式在日益重要的软件开发领域上的补充。

我们也关注用于并发和网络化应用程序和中间件的通用的、与应用领域无关的模式。我们的目标是努力使书中的模式对日常的项目有所帮助，因此，在本书中并不讨论只针对于特殊应用领域的模式，如在[DeBr95][Mes96][ACGH+96]中的模式，只考虑属于电信领域的连网问题。

在集中讨论针对并发和网络化的通用的、与应用领域无关的模式之前，本书也对现存的有关并发网络程序设计和面向对象设计的资料做了补充：

- 关于并发的网络程序设计的文献通常关注操作系统API（如Sockets[Ste98]、POSIX Pthreads[Lew95]、Win32线程[Ric97]）的语法和语义，流行的操作系统往往提供了这些API，作为对内核层通信框架（如System V STREAMS[Ris98] [Rago93]）的访问媒介。

⊖ 我们引用《模式系统》一书时标引[POSA1]而不是标引作者。本书也一样，标引为[POSA2]。我们采用这种记法避免了在读者心目中留下只是第一作者撰写这本POSA著作的印象。

⊖ 本书中文版已由机械工业出版社引进出版。——编者注

而本书着重介绍如何在设计和实现高质量的并发和网络化系统时有效地使用这些API。

- 讨论高层软件设计和质量因素的文献[Boo94][Mey97][DLF93]一般并不关注并发和网络应用程序的开发过程。本书可以填补这一空白。

本书和《模式系统》一书的另一个不同之处在于，本书中的模式并不仅仅构成一个模式分类或模式系统。相反，这些模式相互补充，共同为并发和网络化软件的模式语言提供基础。将这些模式和其他文献中的模式结合起来，我们描述了如何使用这些模式语言来构建高级的并发和网络化软件系统和应用程序、Web服务和分布式对象计算中间件，以及底层的操作系统网络协议和机制。

我们还将对单个模式的描述与它们形成模式语言的方式的讨论分开。首先相对完整地介绍模式本身，以便可以在最有用的语境中应用这些模式。随后的一章则描述模式如何交互，以及如何用其他模式进行补充。

不过，本书中的很多模式可应用于并发和网络化语境之外，指出这一点很重要。为了说明其应用领域的广泛性，我们列出了一些其他领域（如基于组件或交互式软件系统）中的应用实例。另外，我们还给出了一些实例，说明这些模式应用于日常生活中的情形。

大家可能对一些模式很熟悉，因为在PLoP丛书[PLoPD1][PLoPD2][PLoPD3] [PLoPD4]以及《C++报告》（C++ Report）杂志已经发表了它们的初期版本。不过本书对这些早期版本做了相当大的改进：

- 首次将它们组织在一个文档中，这样有助于强调它们所表示的模式语言。
- 根据多方意见对这些模式进行了重写并做了重大修改。这些意见有些来自会议和专题学术讨论会上，有些通过电子邮件发来，有些通过大量的内部评审提出，还有我们的领导人的评审意见。
- 这些模式已经被转换成POSA模式格式，具有一致的书写风格。

本书的读者

和前一卷《模式系统》一样，本书的目的是供专业软件开发人员，尤其是那些正在构建并发和网络化系统的人员使用。本书可以帮助这些软件从业人员以一种新的方式考察软件体系结构，并帮助他们进行大规模的、复杂的中间件和应用程序的设计和编程。

本书也适合于大学高年级学生和研究生阅读，他们应具备足够的网络和操作系统基础知识，渴望掌握有效地设计和实现这样的系统所必备的核心原则、模式和技术。

结构和内容

本书可作为教科书，从头读到尾；也可以作为参考书，用于研究特定的模式在细节上的差别。

本书第1章“并发和网络化对象”，概述并发和网络化的面向对象应用程序和中间件的设计人员所面临的困难。使用了真实的例子——一个并发的Web服务器来说明该领域的一些关键问题，如服务访问和配置、事件处理、同步和并发。

第2章至第5章是本书的主要部分，这几章包含各种模式（用U2乐队的歌词来说就是

“实货，real things”[U2]），其中编纂了为开发高品质并发和网络化系统所确立的原则和技术。我们希望这些模式能成为你开发并发和网络化应用程序以及对你发现的模式建立文档时的有用的角色模型。

第6章“将模式组织在一起”讨论第2章至第5章中的模式之间相互联系的方式。我们还说明这些模式如何与文献中其他模式联系，构成用于并发网络化系统和中间件的模式语言。正如前面所提到的，一些模式还可应用于并发和网络化系统语境之外的场合。对于这些模式，我们总结了它们的应用范围。

第7章“模式的过去、现在和未来”回顾我们1996年在POSA卷1中做出的“模式向何处去”的预言，讨论过去4年间模式的实际发展方向，并分析模式和模式业界的现状。基于这些回顾，我们对自己关于模式应用和模式语言未来的研究方向的看法做了修正。

本书结束部分是对模式总的看法，常用术语的词汇表，本领域研究工作的参考文献，模式索引以及总的主题索引。

可以在http://www.posa.uci.edu/上找到与本书有关的补充材料。该URL中还包含有到ACE和TAO源代码的链接，其中包含本书所有模式的C++和某些Java实例。

毫无疑问，我们忽略了并发和网络化系统的一些方面，某些内容要随实践中应用和扩展模式语言时才会出现。如果你有任何评论、建设性的批评意见，或者是改进本书的风格和内容的建议，请用电子邮件通过patterns@mchp.siemens.de发给我们。也欢迎对我们在模式上所做的工作进行公开讨论，请用我们的邮件地址siemens-patterns@cs. uiuc.edu将反馈、评论和建议发给我们。在模式主页上可以找到订阅的方法，其URL是http://hillside.net/patterns/。该链接中还提供了有关于模式的多方面重要信息源，如已出版和将要出版的书籍，关于模式的会议和论文，等等。

致谢

很高兴地在此感谢那些帮助我们出版这本书的同仁，他们或者与我们分享了知识，或者对早期的草案进行了审查并提供了有用的反馈意见。

首先的评审是由我们尊敬的同事们做出的，他们是Regine Meunier、Christa Schwanninger、Martin Botzler、Lutz Dominick、Prashant Jain、Michael Kircher、Karl Pröse和Dietmar Schütz。他们花费了许多宝贵的时间，来评审本书手稿，对本书进行润色并定稿。还要感谢分布式对象计算（Distributed Object Computing, DOC）组的成员——Tim Harrison、Prashant Jain、Carlos O'Ryan和Irfan Pyarali，我们和他们一起合作完成了本书的六个模式的最初版本。这些研究人员和本书的四位作者一起，组成了分别位于慕尼黑的西门子公司、华盛顿大学圣路易斯分校以及加利福尼亚大学欧文分校的POSA团队。

还要感谢Peter Sommerlad、Chris Cleeland、Kevlin Henney和Paul McKenney。Peter是我们的领队，他深入地审查了我们的所有材料，特别关心正确性、完整性、一致性和质量。我们的同行评审员Chris、Kevlin和Paul向我们提供了详细的反馈意见。他们对于改进本书做出了重大贡献。

也要感谢伊利诺依大学厄巴纳-尚佩恩分校的软件体系结构小组的成员，其中包括

Frederico Balaguer、John Brant、Brian Foote、Alejandra Garrido、Peter Hatch、Ralph Johnson、Dragos Manolescu、Brian Marick、Hiroaki Nakamura、Reza Razavi、Don Roberts、Les Tyrrell、Joseph W. Yoder、Wanghong Yuan、Weerasak Witthawaskul和Bosko Zivaljevic，他们主持了关于许多POSA2模式的作者专题讨论会。他们的评论对于我们提高本书的正确性和可理解性很有帮助。

世界各地的许多同仁对本书的早期版本提供了一些反馈意见，他们是Giorgio Angiolini、Brad Appleton、Paul Asman、David Barkken、John Basrai、Joe Bergin、Rainer Blome、Don Box、Martina Buschmann、Tom Cargill、Kobi Cohen-Arazi Chuck，以及Lorrie Cranor、Robert Crell、James O. Coplien、Ward Cunningham、Mike Curtis、Gisela Ebner、Ed Fernandez、Erich Gamma、Sonja Gary、Luciano Gerber、Bob Hanmer、Neil Harrison、Michi Henning、David Holmes、Tom Jordan、Christopher Kohlhoff、Fabio Kon、Bob Laferriere、Greg Lavender、Doug Lea、John MacMillan、Mittal Monani、Duane Murphy、Jaco van der Merwe、Michael Ogg、Bill Pugh、Patrick Rabau、Dirk Riehle、Linda Rising、Eric Samuelsson、Stefan Scherer、Wolfgang Schroeder、Richard Toren、Siva Vaddepuri、John Vlissides、Roger Whitney、Johnny Willemsen和Uwe Zdun。各种模式的“致谢”一节简述了他们的建议对完善本书的帮助。

还非常感谢华盛顿大学圣路易斯分校、加利福尼亚大学欧文分校、对象计算公司以及Riverace公司的DOC小组的新老成员，他们将本书中介绍的所有模式具体化、细化和优化，在ACE和TAO中间件项目中实现为组件和框架。这个富有灵感的小组中有Everett Anderson、Alex Arulanthu、Shawn Atkins、Darrell Brunsch、Luther Baker、Matt Braun、Chris Cleeland、Angelo Corsaro、Sergio Flores-Gaitan、Chris Gill、Pradeep Gore、Andy Gokhale、Priyanka Gontla、Myrna Harbison、Tim Harrison、Shawn Hannan、John Heitmann、Joe Hoffert、James Hu、Steve Huston、Prashant Jain、Vishal Kachroo、Ray Klefstad、Yamuna Krishnamurthy、Michael Kircher、Fred Kuhns、David Levine、Ebrahim Moshiri、Michael Moran、Sumedh Mungee、Bala Natarjan、Ossama Othman、Jeff Parsons、Kirthika Parameswaran、Krish Pathayapura、Irfan Pyarali、Carlos O'Ryan、Malcolm Spence、Marina Spivak、Naga Surendran、Selcuk Uelker、Nanbor Wang、Seth Widoff以及Torben Worm。还要感谢全球几千ACE和TAO的用户，他们在过去10年间的应用完善了本书中介绍的模式和框架组件。没有他们的支持、不断的反馈和鼓励，我们不会完成这本书。

还要对Johannes Nierwetberg、Lothar Borrmann和Monika Gonauser表示感谢，感谢他们对德国慕尼黑西门子的总公司技术部软件工程实验室在管理上的支持。还要感谢慕尼黑西门子通信设备事业部的Calinel Pasteanu，他对于撰写本书的繁重的工作与“Internet时代”交付产品的压力之间的冲突给予了充分的理解。

还要感谢在模式和ACE及TAO中间件框架等研究上的同事和赞助商，特别是Ron Akers(摩托罗拉)、Al Aho(朗讯)、Steve Bachinsky(SAIC)、Detlef Becker(西门子)、Jim Blaine(华盛顿大学)、John Buttitto(摩托罗拉)、Becky Callison(波音)、Wei Chiang(诺基亚)、Russ Claus(美国宇航局)、Joe Cross(洛克希德)、Bryan Doerr(波音)、Karlheinz Dorn(西门

子)、Sylvester Fernandez(洛克希德)、Andreas Geisler(西门子)、Helen Gill(DARPA)、Trey Grubbs(Raytheon)、Jody Hagins(ATD)、Andy Harvey(Cisco)、Thomas Heimke(西门子)、Kalai Kalaichelvan(北电)、Arvind Kaushal(摩托罗拉)、Steve Kay(Tellabs)、Chandra Kintala(朗讯)、Gary Koob(DARPA)、Sean Landis(摩托罗拉)、Rick Lett(Sprint)、Joe Loyall(BBN)、Mike Masters(NSWC)、Ed Mays(美国海军)、John Mellby(Raytheon)、Dave Meyer(Virtual Technology)、Eileen Miller(朗讯)、Stan Moyer(Telcordia)、Russ Noseworthy(Object Sciences)、Guru Parulkar(Cisco)、Dan Paulish(西门子)、James Plamondon(微软)、Dieter Quehl(西门子)、Lucie Robillard(美国空军)、Allyn Romanow(Cisco)、Rick Schantz(BBN)、Steve Shaffer(柯达)、Dave Sharp(波音)、Naval Sodha(爱立信)、Brian Stacey(北电)、Paul Stephenson(爱立信)、Umar Syyid(休斯)、Dave Thomas(OTI)、Lothar Werzinger(Krones)、Shalini Yajnik(朗讯)和Tom Ziomek(摩托罗拉)。

特别感谢我们手稿的文字编辑，Wordmongers公司的Steve Rickaby，他为我们的内容增色不少。另外，要感谢编辑Gaynor Redvers-Mutton以及其他在John Wiley & Sons出版社工作的人们，是他们的帮助使本书得以出版。本书是Gaynor和Steve精心培育的我们的第二本书。他们的支持是巨大的，我们期望在将来的POSA丛书中再次与他们合作。

最后，要对已故的Richard Stevens表示深深的敬意。许多年前，他的开创性的作品就激发我们去探究网络程序设计的奇妙世界。他的精神渗透在本书中。

作者简介

Douglas C.Schmidt

Douglas Schmidt博士是加利福尼亚大学欧文分校电子与计算机工程系的副教授。他还是国家高级研究计划局（DARPA）信息技术办公室（ITO）的项目经理，在美国的中间件研究中处于领先地位。在此之前，他是密苏里州华盛顿大学圣路易斯分校计算机科学系的分布式对象计算中心的副教授和主任。他的研究方向集中在模式和模式语言、最优化原理以及面向对象技术的经验分析技术，后者用于开发能运行于高速网络和嵌入式互连系统中的高性能和实时分布式对象计算中间件。

Douglas是一位在分布对象计算模式、中间件框架、实时CORBA和开放源码开发方面的国际知名的专家。他在顶级技术杂志、会议上发表了许多文章，出版了多本专著。他曾担任《C++报告》（C++Report）杂志的主编多年，与人合著了几本在模式[PLoPD1]和框架[FJS99a][FJS99b]方面的知名书籍。除了学术研究之外，Douglas领导了ACE和TAO的开发，二者是广泛使用的开放源码的中间件框架，其中包含了很多可重用组件，这些组件是用本书中介绍的模式实现的。

在“闲暇”时间，他喜欢和妻子Sonja跳交谊舞，他还喜欢举重、弹吉他、读世界历史、开雪弗莱车。

Michael Stal

Michael Stal于1991年加入德国慕尼黑的西门子的总公司技术部。早期他在开发编译器软件和计算机图形软件方面积累了丰富的经验。他研究C++运行时类型信息，并曾经服务于C++标准化组X3J16。从1992年开始，Michael的工作集中在使用Sockets、CORBA、COM和Java开发并发和分布式的面向对象系统。Michael是西门子公司与OMG在CORBA方面的主要联系人，也是西门子总公司技术部的分布式对象计算组的负责人。他是POSA第1卷《模式系统》的作者之一。

Michael的主要兴趣是研究用于有效地开发分布式系统的方法和用于描述中间件平台体系结构的模式。他在许多杂志上发表了多篇这些方面的文章，并多次在会议上做报告。另外，他还是《Java Spektrum》的总编以及《Objektspektrum》的专栏作家和顾问委员会成员。前者是德国在Java平台方面的主要杂志，而后者是德国在对象技术方面的主要杂志。

在业余时间，Michael参加足球比赛，支持他喜爱的拜仁慕尼黑足球队，去慕尼黑周围的啤酒园观光，看管小猫别在家搞破坏，看电影，读物理、哲学和幽默方面的书。他是Douglas Adams、Scott Adams以及Terry Pratchett的影迷。

Hans Rohnert

Hans Rohnert博士是德国慕尼黑的西门子通信设备事务部的高级软件工程师。他的主要

研究方向是寻找有前途的软件技术并引入到新产品（如下一代移动电话）中。他的专业兴趣是软件体系结构、设计模式、现实世界程序设计等。他在动态图形算法、嵌入式Java虚拟机等方面做过无数次讲演。

目前Hans是一个专家组成员，该专家组的目标是定义用于小型设备的小型化KVM（键盘、视频和鼠标控制器）Java虚拟机以及类库。他从事的项目包括用于嵌入式服务器的服务端模块、C++中的工作流、对ATM交换的基本支持、CORBA客户端的Java GUI前端以及HTTP客户机。他是POSA第1卷《模式系统》的作者之一，PLoPD丛书的第四卷[PLoPD4]的主编之一。还是一名研究生的时候，他研究过组合算法。在职业生涯的早年，他在这个方面出版了一些著作、发表过一些讲演。

Hans是一个雄心勃勃的网球运动员，但输多赢少。他也喜欢爬附近的山，攀岩，去邻国滑雪。不过他最大的“爱好”是家庭，特别是在本书写作的最后冲刺阶段，家中诞生了一个婴儿。

Frank Buschmann

Frank Buschmann是德国慕尼黑西门子公司总部技术部门的高级工程师。他的研究兴趣包括对象技术、软件体系结构、框架和模式。他在这些领域发表了很多文章，这些文章可以在他与人合著的POSA第1卷中找到。Frank于1992~1996年期间是ANSI C++标准化委员会X3J16的成员。Frank发起并组织了在欧洲举办的第一次关于模式的会议——EuroPLoP 1996，他也是PLoPD丛书第三卷[PLoPD3]的主编之一。Frank参与了一些大规模工业软件项目的设计与实现，这些项目包括商务信息、工业自动化以及电信系统。

工作之余，Frank的大部分时间和妻子Martina一起享受生活，在慕尼黑的啤酒园消磨时光、玩自行车特技、滑雪、骑马，爱看支持的多特蒙德足球队比赛，一听歌剧就犯困，临睡前品苏格兰麦芽酒放松自己。

读者指南

“小猫切尔西，你能告诉我
现在我该走哪条路吗？”
“那主要看你
想到哪儿去。”猫答道。
“我并不在乎去哪里——”爱丽斯说。
“那么你走哪条路都行，”
猫答道。
“——只要我能到某个地方，”爱丽斯附带解释。
“哦，你肯定能做到，”猫答道，
“你只需多走点路。”
——刘易斯·卡洛尔，《爱丽斯漫游奇境》

本书的组织使你可以从头读到尾。不过如果知道想要读什么，可以选择自己的路线。下面的提示可以帮助决定关注哪些主题，以及阅读的顺序。

模式介绍

如果你是第一次接触模式，我们建议首先阅读[POSA1]和[GoF95]中介绍模式的内容，这两本书探讨了与软件体系结构和设计的模式有关的概念和术语。而且，本书介绍的所有模式都建立在[POSA1]定义的模式概念基础之上：

- 软件体系结构的模式定义。
- 将模式分成体系结构模式、设计模式和惯用法[⊖]等三类。
- 模式描述格式。

此外，通过使用[POSA1]和[GoF95]中介绍的模式，增强了本书中很多模式的实现。因此，为了在产品软件开发项目中指导模式的应用，我们建议要有这三本书在手。

结构和内容

本书第1章“并发和网络化对象”，描述设计并发和网络系统中的主要挑战。这一章还概述了我们介绍的模式的范围和语境。最后，这一章给出应用本书中的八个模式来开发并发Web服务器的实例分析。

本书主要内容由16种模式描述和1个惯用法构成。这些内容被分成4章，对应于在并发和网络中间件和应用程序的开发过程中的4个关键领域——**服务访问和配置**、**事件处理**、**同步**

⊖ 这些模式分类定义参见词汇表。

和并发。阅读这些材料的顺序由你决定，比如首先阅读重要的核心模式：

- 包装器外观设计模式。
- 反应器体系结构模式。
- 接受器-连接器设计模式。
- 主动对象设计模式。

本书在介绍其他12种模式和一个惯用法的时候，力求尽可能少地引用前面的内容。阅读它们的顺序可以是任意的。这些内容完善和补充了前面列出的4种模式定义的概念，并涉及很多与有效设计和实现并发和网络对象有关的问题。

你也可以从本书找到在项目中所遇到问题的解决方案。第6章“将模式组织在一起”对模式进行了综述，可以指导搜索，然后在第2章到第5章中找出作为潜在解决方案模式的详细描述。

没有一种模式是完全独立于其他模式的。因此，第6章“将模式组织在一起”也描述如何将本书中的所有模式编织在一起，形成用于构建网络应用程序和中间件的模式语言。如果你想在研究单个模式之前对书中的所有模式有一个整体的看法，建议在深入阅读第2章到第5章中的模式之前，浏览一下第6章中的模式语言介绍。

第7章“模式的过去、现在和未来”，第8章“结束语”是本书内容的终结部分。本书的其余部分是附录，包含技术术语的词汇表、图中使用符号的概述和相关工作的参考文献。

模式形式

和[POSA1]的模式形式一样，本书中的所有模式也是自含的。用这种形式，我们可以介绍模式的精髓和关键细节。我们的目标是为只想对模式的基本思想有一个总体认识的读者服务，以及为那些想深入了解模式工作细节的人们服务。

在我们的模式形式中，每一节为后续节作好准备。例如，“**例子**”一节引出“**语境**”、“**问题**”和“**解决方案**”三部分内容，这三部分总结了模式的精髓。“**解决方案**”一节引出“**结构**”和“**动态特性**”两部分，这两部分更详细地介绍模式是如何工作的，为读者阅读“**实现**”一节做好准备。

“**已解决的例子**”、“**变体**”、“**已知使用**”、“**结论**”和“**参见**”完善了对每种模式的描述。我们还提供了大量的交叉引用，帮助读者理解本书中的模式与其他出版物的模式之间的关系。

为了把模式实现活动的论述与生产软件系统联系起来，大部分实例代码受到了ACE框架[Sch97]中提供的组件的影响。因此，如果想先对所有模式有一个总体认识，需要在第一遍浏览时跳过本书的“实现”一节，当需要了解每一模式的具体实现细节时，再回过头来看它们。

尽管本书中使用的模式形式在模式描述中有重复，但是我们发现，这些重复可以通过减少“回溯”使读者更顺畅地阅读本书。

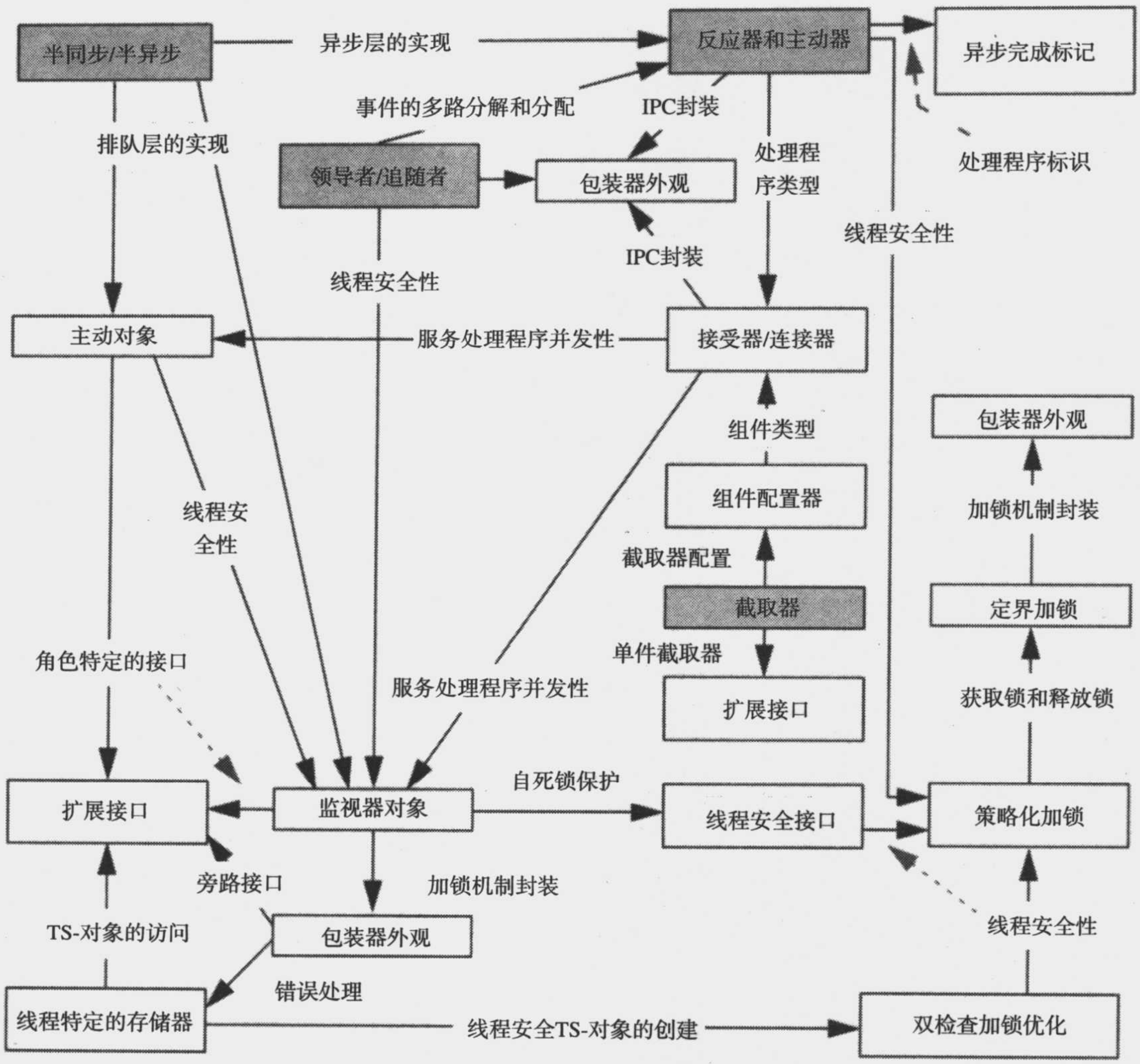
在解释模式的结构和行为的图中，我们尽可能采用标准的UML语言。不过，在极少数情况下，使用UML不能将我们的意思表达清楚。因此，我们对标准符号稍稍进行了“扩展”，这在附录B“符号”部分介绍。

阅读背景材料

许多模式，特别是反应器、主动器、半同步/半异步和领导者/追随者模式，在阅读时假设读者熟悉下列主题：

- **面向对象设计技术**，如模式[GoF95][POSA1]和惯用法[Cope92]、UML符号[BRJ98]以及结构化程序设计的原理，特别是封装和模块化[Mey97]。
- **面向对象程序设计语言的特性**，如类[Str97]、继承和多态[AG98]，以及参数化类型[Aus98]。尽管我们对大多数模式介绍了Java的已使用实例，但本书中的很多例子都是用C++写的。
- **系统程序设计概念和机制**，如进程和线程管理[Lew95][Lea99a][Ric97]、同步[Ste98]和进程间通信[Ste99]。
- **网络服务和协议**，如客户机/服务器计算[CoSte92]以及Internet协议[Ste93][SW94]。

本书还包括大量词汇和参考文献，用于解释不熟悉的术语，提供了你可能想进一步学习的主题信息源。不过本书不是有关并发和网络程序设计的入门书籍。所以，如果你对上面列出的主题不熟悉，我们建议你在阅读本书的同时，也阅读我们推荐的背景材料。


半同步/半异步
异步层的实现
反应器和主动器
异步完成标记
事件的多路分解和分配
IPC封装
排队层的实现
领导者/追随者
包装器外观
处理程序类型
处理程序标识
线程安全性
IPC封装
线程安全性
主动对象
服务处理程序并发性
接受器/连接器
组件类型
包装器外观
组件配置器
加锁机制封装
线程安全性
截取器配置
截取器
定界加锁
单件截取器
角色特定的接口
服务处理程序并发性
扩展接口
获取锁和释放锁
自死锁保护
扩展接口
监视器对象
线程安全接口
策略化加锁
旁路接口
加锁机制封装
TS-对象的访问
包装器外观
线程安全性
错误处理
线程特定的存储器
线程安全TS-对象的创建
双检查加锁优化

软件工程技术丛书书目

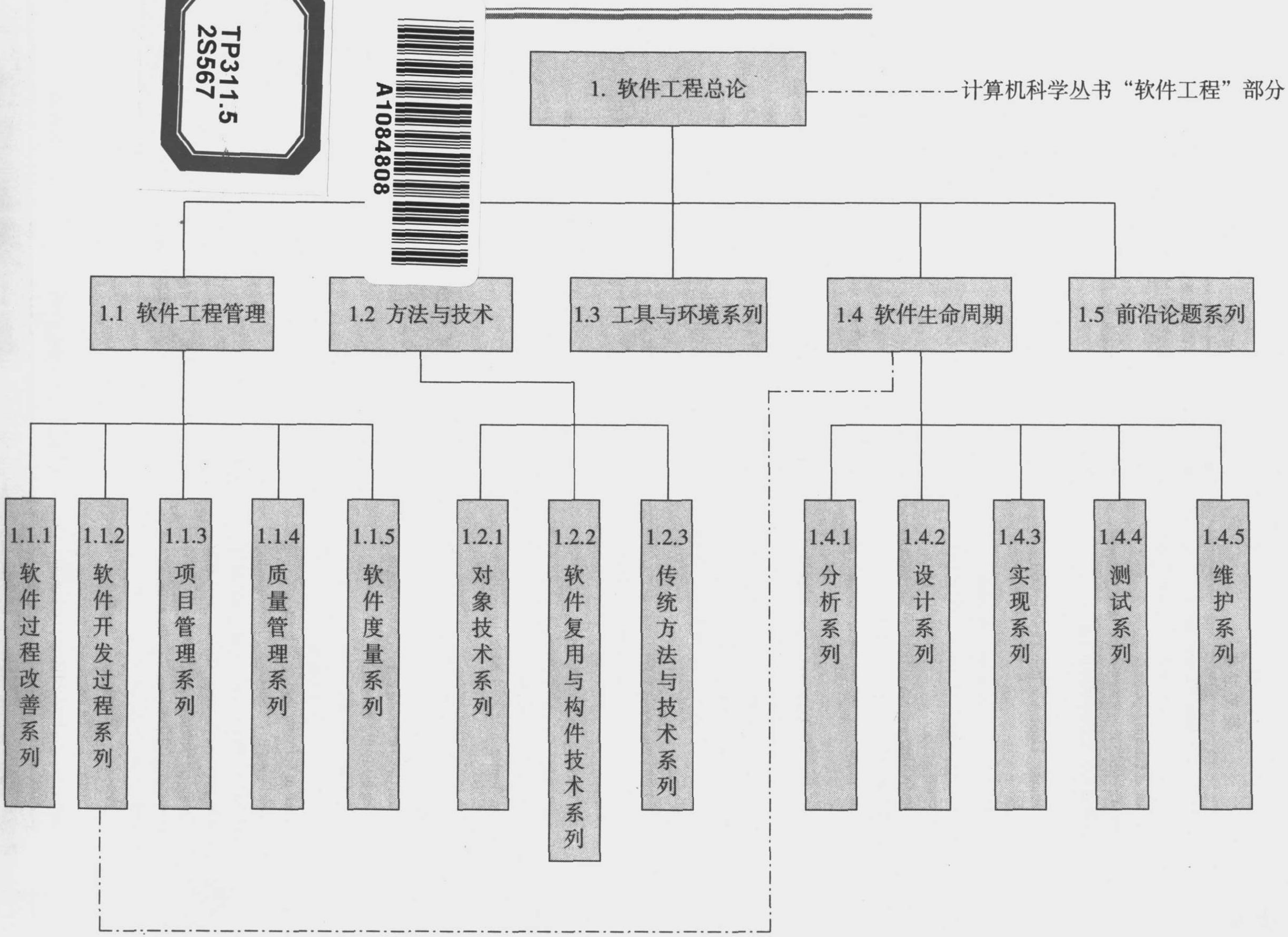
丛书编号	英文书名	中文书名	作　者
1	Object-Oriented and Classical Software Engineering, 5E	面向对象与传统软件工程(原书第5版)	Stephen R. Schach
1	Object-Oriented Software Engineering	面向对象的软件工程	Ian Sommerville
1	Software Engineering: A Practitioner's Approach, 5E	软件工程:实践者的研究方法(原书第5版)	Roger S. Pressman
1	Software Engineering, 6E	软件工程(原书第6版)	Ian Sommerville
1	Software Engineering with Java	软件工程:Java语言实现	Stephen R. Schach
1	Project-Based Software Engineering:An Object-Oriented Approach	基于项目的软件工程:面向对象研究方法	Evelyn Stiller
1.1.1	Software Process Improvement: Practical Guidelines for Business Success	软件过程改进	Sami Zahran
1.1.1	Making Process Improvement Work	软件过程改进简明实践	Neil S. Potter
1.1.2	The Road to the Unified Software Development Process	统一软件开发过程之路	Ivar Jacobson
1.1.2	The Unified Software Development Process	统一软件开发过程	Jacobson/Booch/Rumbaugh
1.1.2	The Rational Unified Process: An Introduction , 2E	Rational统一过程引论(原书第2版)	Philippe Kruchten
1.1.2	UML and The Unified Process: Practical Object-Oriented Analysis & Design	UML和统一过程: 实用面向对象的分析和设计	Jim Arlow
1.1.3	Managing Global Software Projects: How toLead Geographically Distributed Teams,Manage Processes and Use Quality Models	全球化软件项目管理	Gopalaswamy Ramesh
1.1.3	Software Project Management: A Unified Framework	软件项目管理:一个统一的框架	Walker Royce
1.1.3	How to Run Successful Projects III: The Silver Bullet	成功的软件项目管理:银弹方案(原书第3版)	Fergus O'Connell
1.1.3	Successful Software Development, 2E	成功的软件开发(原书第2版)	Scott E. Donaldson
1.1.3	Six Sigma Software Development	6σ软件开发	Christine B. Tayntor
1.1.3	IT Project Management: On Track from Start to Finish	IT项目管理: 从开始到结束的历程	Joseph Phillips
1.1.3	Successful IT Project Delivery: Learning the lessons of Project Failure	IT项目成功交付的秘诀	David Yardley
1.1.3	Software Project Management, 3E	软件项目管理(原书第3版)	Bob Hughes
1.1.3	Architecture-Centric Software Project Management	软件项目管理实用指南: 以体系结构为中心	Daniel J. Paulish
1.1.4	Handbook of Software Quality Assurance, 3E	软件质量保证(原书第3版)	Gordon G. Schulmeyer
1.1.4	Software Reliability Engineering	软件可靠性工程	John Musa
1.1.4	Implementing ISO 9001:2000 The Journey from Conformance to Performance	ISO 9001:2000 实施指南	Tom Tairnubg
1.1.4	CMMI Distilled: A Practical Introduction to Integrated Process Improvement	CMMI精粹:集成化过程改进实用导论	Dennis M. Ahern
1.1.4	CMM Implementation Guide	CMM实施与软件过程改进	Kim Caputo
1.1.4	Implementing the Capability Maturity Model	CMM实施指南	James R.Persse
1.1.4	Object-Oriented Defect Management of Software	面向对象的软件缺限管理	Houman Younessi
1.1.4	Metrics and Models in Software Quality Engineering	软件质量工程:度量与模型	Stephen H. Kan
1.1.4	Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software	软件性能工程	Connie U. Smith

丛书编号	英文书名	中文书名	作　者
1.1.4	Solid Software	Solid Software	Shari Pfleeger
1.1.4	Peer Reviews in Software: A Practical Guide	同级评审	Karl E. Wiegers
1.1.5	Practical Software Measurement	实用软件度量	John McGarry
1.1.5	Software Metrics: A Rigorous and Practical Approach, 2E	软件度量(原书第2版)	Norman E. Fenton
1.1.5	Software Assessments, Benchmarks, and Best Practices	软件评估、基准测试与最佳实践	Capers Jones
1.2.1	The Object Primer: The Application Developer's Guide to Object Orientation and the UML, 2E	面向对象软件开发教程(原书第2版)	Scott W. Ambler
1.2.1	UML and C++: A Practical Guide to Object-Oriented Development, 2E	C++面向对象开发(原书第2版)	Richard C.Lee
1.2.1	Object-Oriented Methods: Principles & Practices, 3E	面向对象方法:原理与实践(原书第3版)	Ian Graham
1.2.1	Principles of Object-Oriented Software Development, 2E	面向对象软件开发原理(原书第2版)	Anton Eliëns
1.2.1	Object Solutions: Managing the Object-Oriented Project	面向对象项目的解决方案	Grady Booch
1.2.1	An Introduction To Object-Oriented Programming, 3E	面向对象程序设计导引(原书第3版)	Timothy Budd
1.2.1	The Object Advantage: Business Process Reengineering with Object Technology	对象优势:采用对象技术的业务过程再工程	Ivar Jacobson
1.2.1	The Unified Modeling Language User Guide	UML用户指南	Booch/Rumbaugh/Jacobson
1.2.1	The Unified Modeling Language Reference Manual	UML参考手册	Rumbaugh/Jacobson/Booch
1.2.1	Applying UML and patterns: An Introduction to Object-Oriented Analysis and Design, 1E	UML和模式应用(原书第1版)	Craig Larman
1.2.1	Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process , 2E	UML与模式应用(原书第2版)	Craig Larman
1.2.1	Object-Oriented Analysis and Design with Applications, 2E	面向对象分析与设计(原书第2版)	Grady Booch
1.2.2	Software Reuse: Architecture, Process and Organization for Business Success	软件复用: 结构、过程和组成	Ivar Jacobson
1.2.2	Software Reuse Techniques: Adding Reuse to the Systems Development Process	软件复用技术: 在系统开发过程中考虑复用	Carma McClure
1.2.2	Practical Software Reuse: Strategies for Introducing Reuse Concepts in Your Organization	实用软件复用方法	Donald J. Reifer
1.2.2	Large-Scale Component-Based Development	大规模基于构件的软件开发	Alan W. Brown
1.2.2	Component-based Product Line Engineering with UML	基于构件的产品线工程:UML方法	Colin Atkinson
1.4.1	Object-Oriented Analysis & Design	面向对象的分析与设计	Andrew Haigh
1.4.1	Analysis Patterns: Reusable Object Models	分析模式: 可复用的对象模型	Martin Fowler
1.4.1	Requirements Analysis and System Design: Developing Information Systems with UML	需求分析与系统设计	Leeszek A. Maciaszek
1.4.1	Systems Analysis and Design in a Changing World	系统分析与设计	John W. Satzinger
1.4.1	Advanced Use Case Modeling, Vol I: Software Systems	高级用例建模 卷I: 软件系统	Frank Armour
1.4.1	Requirements Engineering: A Good Practice Guide	需求工程	Ian Sommerville
1.4.1	Software Requirements and Estimation	软件需求与估算	Swapna Kishore
1.4.1	Effective Requirements Practices	有效需求实践	Ralph R.Young
1.4.1	Applying Use Cases: A Practical Guide, 2E	用例分析技术(原书第2版)	Geri Schneider
1.4.1	Managing Software Requirements	软件需求管理:统一方法	Dean Leffingwell
1.4.1	Writing Effective Use Cases	编写有效用例	Alistair Cockburn
1.4.1	Problem Frames Analyzing and Structuring Software Development Problems	Problem Frames Analyzing and Structuring Software Development Problems	Michael Jackson

丛书编号	英文书名	中文书名	作　者
1.4.2	Object-Oriented Software Construction, 2E	面向对象的软件结构(原书第2版)	Bertrand Meyer
1.4.2	Pattern-Oriented Software Architecture, Vol I: A System of Patterns	面向模式的软件体系结构 卷1:模式系统	Frank Buschmann
1.4.2	Pattern-Oriented Software Architecture, Vol II: Patterns for Concurrent and Networked Objects	面向模式的软件体系结构 卷2:用于并发和网络化对象的模式	Douglas Schmidt
1.4.2	Server Component Patterns	面向模式的软件体系结构 卷3:服务器组件模式	Markus Volter
1.4.2	DesignPatterns: Elements of Reusable Object-Oriented Software	设计模式:可复用面向对象软件的基础	Gamma/Helm/Johnson /Vlissides
1.4.2	Patterns of Enterprise Application Architecture	企业级应用体系结构模式	Martin Fowler
1.4.2	AntiPatterns and Patterns in Software Configuration Management	软件配置管理中的模式与反模式	William J. Brown
1.4.2	Software for Use: A Practical Guide to The Models and Methods of Usage-Centered Design	面向使用的软件设计	Larry L. Constantine
1.4.2	Software Architecture: Organizational Principles and Patterns	软件架构: 组织原则与模式	David Dikel
1.4.2	The UML Profile for Framework Architectures	框架体系结构的UML档案	Marcus Fontoura
1.4.2	Software Architect's Profession: An Introduction	软件架构师入门必读	Marc Sewell
1.4.2	Business Modeling with UML: Business Patterns at work	UML业务建模: 实用业务模式	Hans-Erik Eriksson
1.4.3	Software Fabrication:Automating Application Development	软件构造:自动化软件开发	Jack Greenfield
1.4.3	Building J2EE Applications With The Rational Unified Process	用RUP构建J2EE 应用程序	Peter Eeles
1.4.3	Programming from Specifications	从规范出发的程序设计	Carroll Morgan
1.4.4	Testing IT: An Off-the-Shelf SoftwareTesting Process Handbook	实用软件测试过程之路	John Watkins
1.4.4	Lessons Learned in Software Testing	软件测试经验与教训	Cem Kaner
1.4.4	Testing Computer Software: The Bestselling Software Testing Book Of All Time, 2E	计算机软件测试(原书第2版)	Cem Kaner
1.4.4	Software Testing in the Real World: Improving the Process	软件测试过程改进	Edward Kit
1.4.4	Effective Methods for Software Testing, 2E	有效的软件测试方法(原书第2版)	William E. Perry
1.4.4	Beta Testing for Better Software	软件Beta测试	Michael R. Fine
1.4.4	A Practical Guide to Testing Object-Oriented Software	面向对象的软件测试	John D. McGregor
1.4.4	Managing the Testing Process, 2E	测试过程管理(原书第2版)	Rex Black
1.4.4	Software Testing: A Craftsman's Approach, 2E	软件测试(原书第2版)	Paul C. Jorgensen
1.4.4	Just Enough Software Test Automation	软件测试自动化	Daniel J. Mosley
1.4.4	The Craft of Software Testing: Subsystem Testing Including Object-Based and Object-Oriented Testing	软件测试实用技术	Brian Marick
1.5	JAVA Tools for Extreme Programming: Mastering Open Source Tools, Including Ant,JUnit, and Cactus	极限编程的JAVA工具	Richard Hightower
1.5	Agile Software Development Ecosystems	敏捷软件开发生态系统	Tom DeMarco
1.5	Agile Modeling: Effective Practices For eXtreme Programming and The Unified Process	敏捷建模:极限编程和统一过程的有效实践	Scott W. Ambler
1.5	Model-Driven Development: Automating Component Design, Implementation, and Assembly	模式驱动的软件开发	David Frankel
1.5	A Practical Guide to Feature-Driven Development	特征驱动开发方法:原理与实践	Steve R. Palmer

软件工程技术丛书结构图

TP311.5
2S567
A1084808



目　录

译者序

译者简介

前言

内容简介

作者简介

读者指南

第1章　并发和网络化对象 ……1

1.1　动机……1

1.2　并发和网络化软件的难题……3

1.2.1　难题1：服务访问和配置……5

1.2.2　难题2：事件处理……8

1.2.3　难题3：并发 ……10

1.2.4　难题4：同步 ……12

1.2.5　网络化软件的其他难题……13

1.3　案例分析：设计一个并发的Web服务器 ……14

1.3.1　JAWS框架概述 ……15

1.3.2　在JAWS中应用模式解决JAWS中常见的设计难题……16

1.3.3　封装低层操作系统API ……17

1.3.4　从协议处理中分离事件多路分解和连接管理……18

1.3.5　通过多线程提高服务器性能……19

1.3.6　实现一个同步化的请求队列……20

1.3.7　最小化服务器线程的开销……21

1.3.8　有效地利用异步I/O ……23

1.3.9　增强服务器的可配置性……24

1.3.10　用于实现JAWS的其他模式 ……25

1.4　小结 ……26

第2章　服务访问和配置模式……27

2.1　包装器外观 ……28

2.2　组件配置器 ……46

2.3　截取器 ……66

2.4　扩展接口 ……85

第3章　事件处理模式 ……107

3.1　反应器……108

3.2　主动器……130

3.3　异步完成标记……158

3.4　接受器-连接器 ……173

第4章　同步模式……199

4.1　定界加锁……199

4.2　策略化加锁……205

4.3　线程安全接口……212

4.4　双检查加锁优化……217

第5章　并发模式……225

5.1　主动对象……226

5.2　监视器对象……245

5.3　半同步/半异步 ……261

5.4　领导者/追随者 ……273

5.5　线程特定的存储器……289

第6章　将模式组织在一起 ……309

6.1　从独立模式到模式语言……309

6.1.1　没有模式是孤立的 ……309

6.1.2　走向模式语言 ……310

6.2　用于中间件和应用程序的模式语言……311

6.2.1　模式语言细节 ……312

6.2.2　对模式语言的讨论 ……317

6.3　并发和网络化之外……318

6.3.1　图形用户界面 ……318

6.3.2　组件 ……319

6.3.3　常规编程 ……319

6.4　模式语言与模式系统……319

第7章 模式的过去、现在和未来 ……323
7.1 在过去的4年中发生了什么 ……323
7.1.1 模式 ……323
7.1.2 模式系统和模式语言 ……324
7.1.3 方法和工具 ……324
7.1.4 算法与数据结构 ……325
7.1.5 形式化模式 ……325
7.2 模式现状……325
7.3 模式向何处去……326
7.3.1 模式 ……326
7.3.2 模式语言 ……328
7.3.3 经验报告、方法和工具 ……328
7.3.4 模式文档 ……329
7.3.5 形式化模式和模式语言 ……329
7.3.6 软件开发过程和组织 ……329
7.3.7 教育 ……330
7.3.8 长远预测 ……330
7.4 对预测未来的临别思考……330
第8章 结束语 ……333
附录A 词汇表 ……335
附录B 符号 ……347
附录C 参考文献 ……351
索引 ……373

第1章 并发和网络化对象

“除音乐之外，我们已习惯地把模式看做是
固定的事物。这样做简单而且省事，
但毫无意义。
正确方法是把相互关联的模式看做
一首舞曲的相互作用的声部，
按各种限制所固定下来。”
——文化人类学家Gregory Bateson

本章介绍与并发和网络化对象相关的主题。我们首先以这个领域内对高级软件开发技术的需求作为动机，接着，概要介绍并发和网络化的面向对象的应用和中间件的开发人员所面临的关键设计难题。为阐明模式如何能被用于解决这些问题，我们对一个面向对象的框架和使用这个框架实现的高性能Web服务器进行案例分析。其中，我们将注意力放在本书中所介绍的关键模式上，它们简化了并发和网络化应用的四个重要方面：

- 服务访问和配置。
- 事件处理。
- 同步。
- 并发。

1.1 动机

过去十年，VLSI技术和光纤的进展使计算机的处理能力以3～4个数量级增长，同时网络连接速度以6～7个数量级增长。假定这些趋势持续下去的话，那么到这个十年结束时：

- 台式电脑的时钟速度将在～100GHz上运行。
- 局域网的连接速度将在～100Gb/s上运行。
- 无线连接速度将在～100Mb/s上运行。
- 互联网的主干网连接速度将在～10Tb/s上运行。

而且，全世界将有成千上万的交互式和嵌入式计算和通信设备在运转。这些强大的计算机和网络大部分将以日用品的价格获得，大部分将用健壮的商用现成产品(common-off-the-shelf, COTS)组件构建，并且将在日益汇聚和充斥的互联网基础设施之上互操作。

为最大限度地利用这些硬件技术的进步，用于开发并发和网络化中间件和应用软件的技术的质量和生产率也必须增长。从历史上看，硬件发展趋势是更小、更快和更可靠。正如“摩

尔定律”所证实的，硬件的发展和创新也更便宜和更可预测。相反，并发和网络化软件常常发展成更庞大、更慢和更易出错。开发、验证、维护和增强这些软件也将变得非常昂贵且耗费时间。

尽管硬件的改进减轻了对某些低层软件优化的需求，但开发软件所需的生命期费用[Boe81]和工作持续地增长，尤其是任务关键型的并发和网络化应用。硬件发展的快速与软件发展的缓慢之间的不匹配有许多因素，如：

- *固有的和偶发的复杂性*。与并发和网络化软件相伴的令人烦恼的问题，是由固有的和偶发的复杂性所产生的。固有的复杂性来源于基本的领域难题，例如处理部分失效、分布式死锁和端到端的服务质量(QoS)需求。随着网络化系统在规模和功能性上的增长，它们现在必须应付更广和更难的复杂性。

  偶发的复杂性来源于软件工具和开发技术的局限，例如不可移植的编程API和拙劣的分布式调试器。具有讽刺意味的是，许多偶发的复杂性源于喜爱低层语言和工具的开发人员有意的选择，而当这些语言和工具用于复杂的并发和网络化软件时，它们的可扩展性很差。
- *不适当的方法和技术*。流行的软件分析方法[SM88][CY91][RBPEL91]和设计技术[Boo94][BRJ98]主要集中于构造具有“尽力服务（best-effort）”QoS需求的单进程、单线程应用。而高质量的并发和网络化系统的开发，尤其是那些具有严格的QoS需求的系统，例如视频会议，则要靠熟练的软件体系结构设计者和工程师的直觉和专门知识。而且，如果不花费大量的时间通过反复试验来学习，并且仔细推敲和处理与平台有关的细节，要想获得并发和网络化软件技术的经验是非常困难的。
- *不断地重复发明和重新发现核心概念和技术*。软件行业有一段对已解决的问题重新建立不兼容的解决方案的历史。例如，有许多非标准的、通用或实时的、管理同一硬件资源的操作系统。与此类似，有许多不兼容的操作系统封装库，这些库提供略微不同的但本质上实现同一特征和服务的API。

如果集中力量增强和优化较少的解决方案，那么并发和网络化软件的开发人员可能正获得硬件开发人员所享受的好处。硬件开发人员通过使用通用CAD工具、标准指令集、总线和网络协议而迅速创新。

没有单一的方案能解决困扰并发和网络化软件[Broo87]的所有难题。不过，经过过去十年的发展，已经变得很清楚的是，*模式和模式语言*有助于减轻许多固有的和偶发的软件复杂性。

模式是一个在特殊的语境下，对一个标准问题的重现的解决方案[POSA1]。模式有助于在软件设计中捕捉和复用静态和动态结构以及关键参与者之间的协作。它们对于文档化重现的微体系结构是有用的，这些微体系结构是软件组件的抽象[GoF95]，有经验的开发人员用它来解决常用的设计和实现问题。

当把有关的模式组织在一起时，它们形成一种“语言”，此语言可以：

- 为讨论软件开发问题定义一种词汇[SFJ96]。
- 为这些问题的有序分解提供一个过程[Ale79][AIS77]。

通过学习和使用模式和模式语言，开发人员常常能从一些陷阱和困境中解脱出来，而在传统意义上，要想避免这些陷阱和疏忽，只有通过长期的和昂贵的学习过程[PloPD1]。

直至前不久，用于开发并发和网络化软件的模式仍停留于个人编程中，或是专业研究人员和开发人员的头脑中，或是深藏于复杂的源代码中[Lea99a]。由于下述三个方面的原因，这种状况并不令人满意：

- 从源代码中重新发现模式是昂贵的和耗时的，因为从实现细节中分离出基本设计决策是极其困难的。
- 如果有经验的设计者的见解和理论没有用文字记载的话，那么它们将随着时间的消失而消失，因而不能帮助指导后续的软件维护和增强活动。
- 没有前期工作的指导，并发和网络化软件的开发人员面临从头开始设计复杂系统的艰巨任务[SeSch70]，而不能复用已成功的方案。

4

结果，许多并发和网络化软件系统从头开发。然而，在当今充满竞争的、上市时间紧迫的环境下，这常常产生未优化的专用解决方案。这些方案难于定制和调整，因为所有工作仅仅试图使软件运转。需求随时间发生变化，而专用解决方案的改进变得异常昂贵。但最终用户期望，软件是能用得起的、健壮的、高效的和灵活的，没有坚固的体系结构的支撑，这一切是难以实现的。

为帮助解决这些问题，本书记载用于并发和网络化软件的关键体系结构和设计模式。这些模式能够并且已经用于解决在开发面向对象的中间件框架和应用时出现的许多常见的问题。建立文档时参考本书，书中模式保存了必要的设计信息，有助于开发人员将现有的软件改进得更健壮。设计系统时参考本书，书中模式有助于指导开发人员更有效地创建新软件。

当然，模式、对象、组件和框架并不是万能药。例如，在解决所有复杂的并发和网络化软件时它们不能解除开发人员分析、设计、实现、验证和优化问题的责任。没有东西可以取代人类创造力、经验、训练、勤奋和判断力。

然而恰当使用本书描述的模式时，它可以帮助减轻前面列举的许多复杂性。尤其是模式能够做下列工作：

- 使开发人员将精力集中于更高级的软件应用体系结构和设计，例如合适的服务访问与配置、事件处理和线程模型的规范。这些是并发和网络化软件的有战略意义的重要方面。正确地处理它们，会极大地减少许多令人烦恼的复杂性。
- 再次避免开发人员专注于低层操作系统及网络协议和机制。虽然牢固掌握这些内容是重要的，但它们在整个范围中处于战术层，必须在整个软件体系结构和开发工作中被置于适当的语境。

5

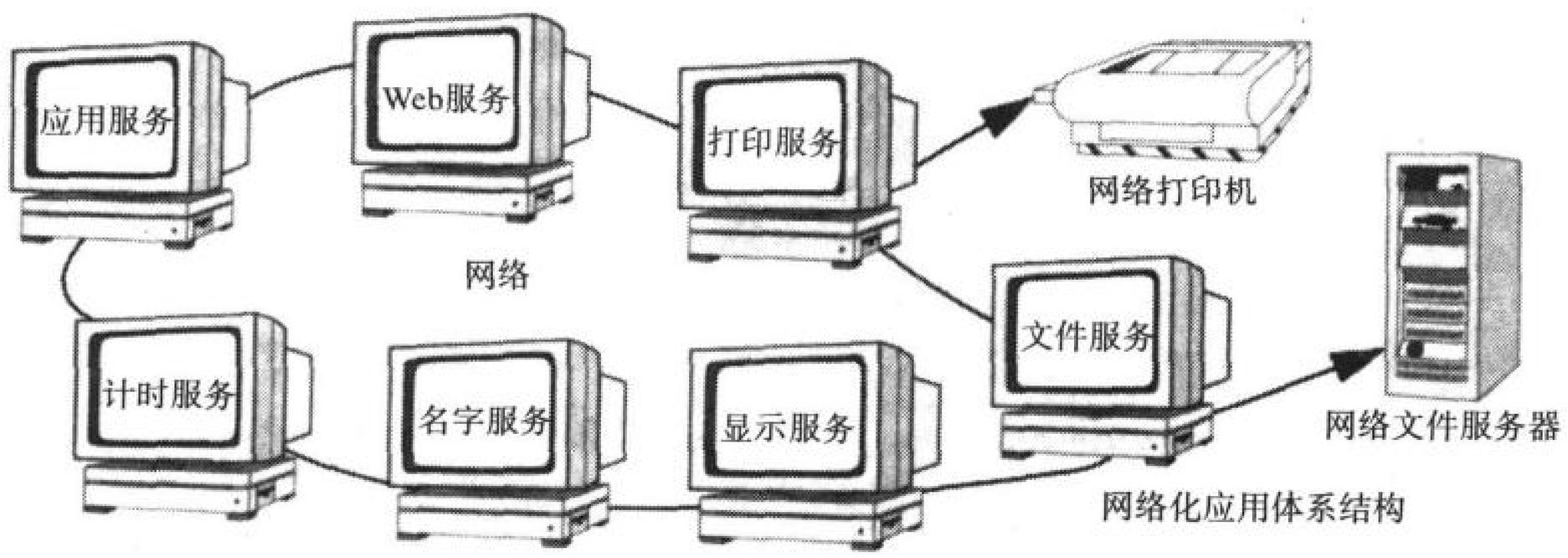
1.2 并发和网络化软件的难题

在理论上，开发使用并发和网络化服务的软件应用可以提高系统的性能、可靠性、可伸缩性和成本效益。然而实际上，开发高效的、健壮的、可扩展的又用得起的并发和网络化应用程序是极其困难的，这是由于独立应用和网络化应用在体系结构上有着很大的不同。

在独立应用体系结构中，用户接口、应用服务处理和永久数据资源驻留在一台计算机中，外设直接连接到计算机。相反，在网络化应用体系结构中，交互式表示、应用服务处理和数据

资源可以驻留在多个松散耦合的主机中，服务层通过局域网或者广域网连接在一起。

X终端的网络和“瘦客户机NetPC”是一个网络系统体系结构的例子。在这个环境中，用户界面表示被终端用户主机上的显示服务所代替。处理能力由主计算机提供，所有或者部分应用服务都在它上面运行。对常用资源的访问由一个或者多个网络文件服务器传递。其他的服务，例如命名和目录服务、计时服务、HTTP服务器、缓存和网络管理服务，都能在网络上运行，同时为应用提供附加能力（如图1-1）。



6

图 1-1

采用网络化体系结构主要有三个理由：

- *协作和互连*。Web和电子商务的爆炸性增长说明了连网的一个最普通的理由：可以连接并访问分布在全世界的海量信息和服务。在互联网上可用的即时消息和“聊天室”的流行强调另一个普通的连网动机：保持与家庭、朋友、协作者和客户的联系。
- *增强的性能、可伸缩性和容错*。网络化体系结构的性能和可伸缩性可以通过在网络中可得到的并行处理能力来增强。例如，多重计算和通信服务处理任务可以并行地在不同的主机上运行。类似地，可以在多个主机间复制各种应用服务。复制可以尽量减少单点失效，因而在部分失效的情况下提高系统的可靠性。
- *成本效益*。网络化体系结构产生能共享昂贵外设的、分散的和模块化的应用，例如大容量的文件服务器和高精度的打印机。同样，可选择一些应用组件和服务委托在具有专用处理属性的主机上运行，例如高性能的硬盘控制器、大容量的内存，或者增强的浮点性能。

尽管网络化应用具有许多优点，但它们与独立应用相比，在设计、实现、调试、优化和管理上更加困难。例如，开发人员必须面临独立应用不涉及到或不太成为问题的主题。这些主题包括：

- 建立连接和初始化服务。
- 事件多路分解和事件处理程序分配。
- 进程间通信(interprocess communication，IPC)和网络协议。
- 主存储器和次级存储器管理和缓存。
- 静态的和动态的组件配置。
- 并发和同步。

7

这些主题通常独立于特定的应用需求，因此学会处理方法可举一反三，解决更多的软件开发问题。而且，由于一些与并发和网络化系统有关的固有的和偶发的复杂性，在这些主题的语境中，会出现许多设计和编程难题：

- 与并发和网络化系统有关的常见的固有的复杂性包括管理带宽[ZBS97]、最小化延迟(等待时间)[SC99]和延迟偏差(抖动)[PRS+99]，检测部分失效并从中恢复[CRSS+98]，确定合适的服务划分和负载平衡策略[IEEE95]，确保事件的因果顺序[BR94]。类似地，在并发编程中含有的常用固有的复杂性包括消除竞争条件和避免死锁[Lea99a]，确定适合的线程调度策略[SKT96]，优化终端系统协议处理性能[SchSu95]。
- 与并发和网络化系统相关的、常见的偶发复杂性包括可移植性操作系统API的缺乏[Sch97]、不充分的调试支持和用于分析并发和网络化应用程序的工具的缺乏[LT93]，广泛的算法的（而不是面向对象的）分解[Boo94]，不断的重新发现和重复发明核心概念和常用组件[Kar95]。

因此，在这一节中，我们讨论许多与构建并发和网络化系统有关的设计和编程难题。然而，本书中的模式并不能处理与并发和连网有关的所有方面。因此，第6章“将模式组织在一起”把本书中的模式和其他文献中的相关模式关联起来。剩下的难题是将来的模式研究的问题，这在第7章“模式的过去、现在和未来”中阐述。

8

1.2.1 难题1：服务访问和配置

独立应用程序中的组件可以借助于函数调用传递参数和访问全局变量，在一个单一的地址空间范围内协作。在网络化应用程序中，组件可以使用下列手段进行协作：

- 进程间通信(IPC)机制，例如共享内存、管道和套接字[Ste98]，[⊖]它们基于类似TCP、UDP和IP[Ste93]或者ATM[CFFT97]的网络协议。
- 通信协议[Ste93]，例如TELNET、FTP、SMTP、HTTP和LDAP，它们被许多类型的服务用于给应用程序导出内聚的软件组件和功能，这些服务如远程登录、文件传输、电子邮件、Web内容发送和分布式目录。
- 使用高级通信中间件，例如COM+[Box97]和CORBA[OMG98c]，在应用级服务组件上进行远程操作。

应用程序和软件组件可以在一个网络化系统中，通过定义在任何抽象级别上的编程API来访问这些通信机制，如图1-2所示。

9

为访问这些通信机制而设计有效的API十分重要，因为这些是应用程序、组件和服务的开发人员直接编程的接口。

对于基础设施连网或者系统程序（如TELNET或者FTP）而言，传统上的服务访问包括调用下列API：

- 调用并发服务访问API来管理并发，如UNIX进程[Ste99]、POSIX Pthreads[IEEE96]或者Win32线程[Sol98]。

⊖ 在本章的剩下部分，我们在讨论中参考和使用具体的UNIX和Win32操作系统API。你手边应该有几本参考书，例如[Ste99][Ste98][Lew95][Ric97]，碰到不熟悉的主题和术语时，便于查阅。

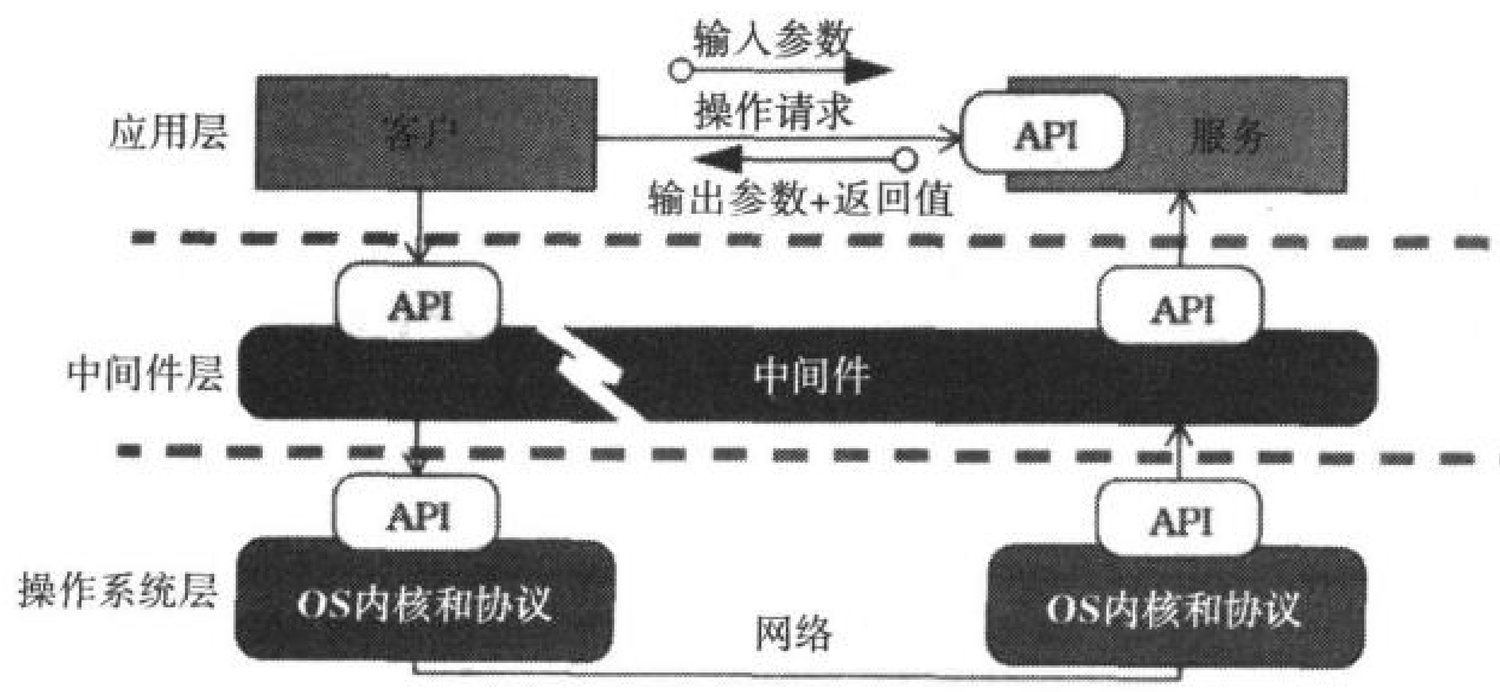


图 1-2

- 调用IPC服务访问API，例如UNIX和互联网领域套接字[Ste98]，配置位于单一主机或不同主机上的进程之间的连接和通信。

然而，当通过低层操作系统C语言API访问连网和主机服务时，会出现几个偶发的复杂性：

- *过度的低层细节*。使用操作系统API构建网络化应用程序需要开发人员详细了解许多低层细节。例如，开发人员必须仔细跟踪每个系统调用返回的错误代码，并且在应用程序代码中处理这些问题。例如，使用`wait()`系统调用的UNIX服务器开发人员必须区分由于没有子进程出现而返回的错误与从单个中断来的错误。在后一种情况，`wait()`必须重新执行。迫使应用开发人员处理这些细节会转移他们对更为关键性要点的注意力，例如服务器的语义和它的软件体系结构。
- *不兼容的高级编程抽象的不断重新发现和重复发明*。对于具有操作系统API的过度细节的一般改正方法是定义更高级编程抽象。例如，许多Web服务器创建一个文件高速缓存组件以避免每个客户机请求访问文件系统[HPS99]。然而，这些类型的抽象常常被所有开发人员或开发组独立地重新发现和重复发明。如果这个特殊的软件编程过程分化了开发人员满足客户需求的努力，那么它实际上影响了编程效率。它还会产生多余的不兼容组件，并且这些组件未被适当地记载、调试，因而在项目内和其他项目中都不容易得到重用。

10

- *出错的高可能性*。由于操作系统API差异很大且缺乏类型安全，所以对它编程单调乏味，极易出错。例如，许多网络化应用程序使用在C中定义的套接字API [MBKQ96]编程。然而，套接字端点被表示为无类型句柄。这些句柄增加了潜在的编程错误和运行时错误[Sch92]。尤其可能操作不当，例如在被动模式句柄上调用数据传输操作，而实际上只是期望用这个句柄建立连接。
- *缺乏可移植性*。操作系统API的不可移植性已是声名狼藉，甚至同一平台的不同版本也不可以移植。例如，Win32平台上的套接字API的实现(WinSock)与UNIX平台上的相比有许多不同。结果，高级的套接字操作，例如组播和广播，是不可以跨越这些平台移植的。甚至在不同Windows版本上的WinSock实现也存在不兼容的、与定时有关的程序错误，当执行非阻塞连接时，这些错误造成偶发性的失效。

- 陡峭的学习曲线。由于过多的细节层，所以掌握操作系统API所付出的劳动非常多。例如，学会如何正确地使用POSIX异步I/O编程[POSIX95]是困难的。学会如何使用异步I/O机制编写一个可移植的应用程序甚至更加困难，因为不同操作系统平台会产生巨大的差别。
- 不能按比例扩展以处理不断增长的复杂性。操作系统API为机制定义基本接口（如进程和线程管理、进程间通信、文件系统和内存管理）。然而，随着应用程序的增长，这些基本接口在大小和复杂性方面并不能协调地成比例扩展。例如，一般的UNIX进程仅允许储备约7个待解决的连接[Ste98]，但这个数目不足以支持被频繁访问的电子商务服务器，而这些服务器必须同时处理成百上千的客户机。

11

因此，基础设施连网或者系统程序的最主要的设计难题，是在不牺牲性能的前提下，最小化上述偶发的复杂性。

对于更高级的分布式对象计算应用程序而言，服务访问还包括在定义常用服务的可复用的组件上调用远程操作，例如命名[OMG97a]、交换（trading）[OMG98b]和事件通知[OMG99c]。许多组件模型，例如EJB[MaHa99]、COM+[Box97]和CORBA组件模型[OMG99a]，都允许组件根据不同因素，比如客户机所期望的版本或者客户机的授权级别等，对不同的客户机导出不同的服务角色。因此，在这个级别上，主要的设计难题在于如何确保客户机不访问无效的或未被授权的组件服务角色。

解决这个难题很重要：网络化应用程序与独立应用程序相比，安全性缺口更容易受到攻击，这是因为入侵者可以有更多的访问点进行攻击[YB99]。例如，许多共享介质的网络，如以太网、令牌环网和FDDI，在防止电缆窃听和“包探听”工具上仅提供有限的内置防护[Ste93]。还有，网络化应用程序必须防止一个主机冒充成另一个来访问未被授权的信息。尽管一些网络软件库，例如OpenSSL[OSSL00]，支持验证、授权和数据加密，但访问这些安全性服务的单一API没有得到普遍采用。

支持服务和应用程序的静态和动态演化，是网络化软件系统中的另一重大难题。演化能够以两种方式出现：

- 通常可以在运行时改变对组件服务角色的接口和组件服务角色之间的互连，新的服务角色可以实现并安装到现有的组件中。
- 通过重新配置服务负载以利用多主机的处理能力，可以提高分布式系统性能。

理想情况下，这些组件配置和重配置变化对于访问服务的客户机应用程序而言应该是透明的。因此，另一个设计难题是确保整个系统不必仅仅因为组件中的某个服务角色被重新配置或其负载被重新分布，而被关机、重新编译、重新链接或重新启动。

12

更具挑战性的是确定如何访问被“按需”配置到系统中的服务，当系统最初设计时，此服务的实现是未知的。

许多现代操作系统和运行时环境提供显式的动态链接API[WHO91]，允许按需求配置应用程序：

- UNIX定义`dlopen()`、`dlsym()`和`dlclose()` API，它们能分别用于显式地加载一个指定的动态链接库（Dynamically Linked Library, DLL）到应用程序进程，从DLL中抽取一个指

定的工厂函数，解链/卸载DLL[Ste98]。

- Win32提供LoadLibrary()、GetProcAddr()和CloseHandle() API，它们执行同UNIX DLL API一样的功能[Sol98]。
- Java的java.applet.Applet类定义init()、start()、stop()和destroy()钩子方法，分别支持动态加载的小应用程序的初始化、开始、停止和终止。

然而，按需求把服务配置到应用中需要的不仅是动态的链接机制——它需要用于协调（重新）配置策略（Policy）模式。这里的设计难题是两方面的。首先，即使应用程序也许不知道它们的详细接口，它也必须导出新的服务；其次，应用程序即使在运行时也必须把这些服务透明地、健壮地集成到它自己的控制流和处理序列中。

第2章“服务访问和配置模式”介绍了在独立的和网络化的软件系统和应用程序中，用于设计访问并配置服务和组件的有效的编程API的四种模式。这些模式是包装器外观(Wrapper Facade)、组件配置器(Component Configurator)、截取器(Interceptor)和扩展接口(Extension Interface)。

13

1.2.2 难题2：事件处理

随着系统日益变得网络化，支持事件驱动应用程序的软件开发技术也变得越来越普遍。以下三个特性使事件驱动的应用程序有别于具有传统的“自定向”控制流的那些应用程序[PLoPD1]：

- 应用程序行为被异步出现的外部或者内部事件触发。这些事件包括设备驱动程序、I/O端口、传感器、键盘或者鼠标、信号、定时器或者其他的异步软件组件。
- 大多数事件必须得到迅速的处理，以防止CPU资源匮乏，改进可察觉的响应时间，使具有实时约束的硬件避开失效的或被破坏的数据。
- 也许需要有限状态机[SGWSM94]来控制事件处理并检测非法的转换，因为事件驱动的应用程序通常只有很少或者没有对事件到达次序的控制。

因此，事件驱动的应用程序常常构造为通常所说的“控制转换”（inversion of control）[John97]的分层的体系结构[POSA1]，如图1-3所示。

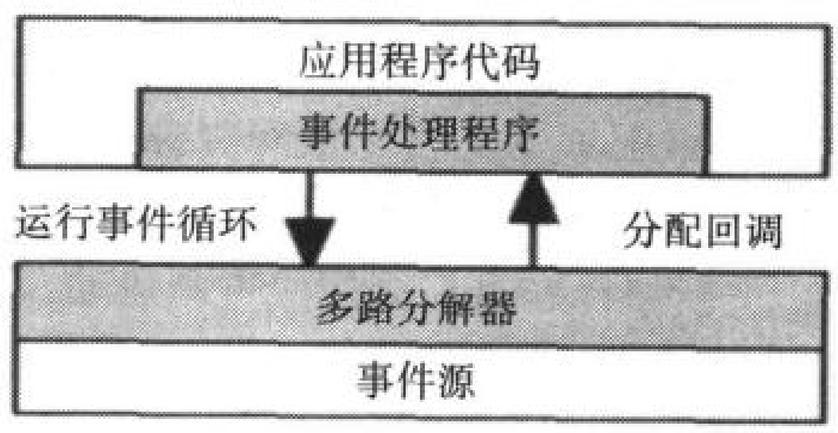


图 1-3

- 底层是事件源(event source)，它从各种硬件设备或者驻留在操作系统中的低层软件设备驱动程序那里检测和检索事件。
- 下一层是事件多路分解器(event demultiplexer)，它使用类似select()这样的函数[Ste98]，

在各种各样的事件源上等待事件的到来，接着把事件分配给它们对应的事件处理程序回调。

- 事件处理程序(event handler)和应用程序代码一起，又形成另一层，它执行响应回调的与应用有关的处理——因此有“控制转换”这一名称。

14

在这种事件驱动的体系结构中，事务分离允许开发人员将精力集中于应用层功能，而不是为每个新的系统或者应用程序反复地重写事件源和多路分解器层。

在许多网络化系统中，应用程序通过对等协议通信，例如TCP/IP[Ste93]，并使用上述分层的事件驱动体系结构实现（如图1-4）。

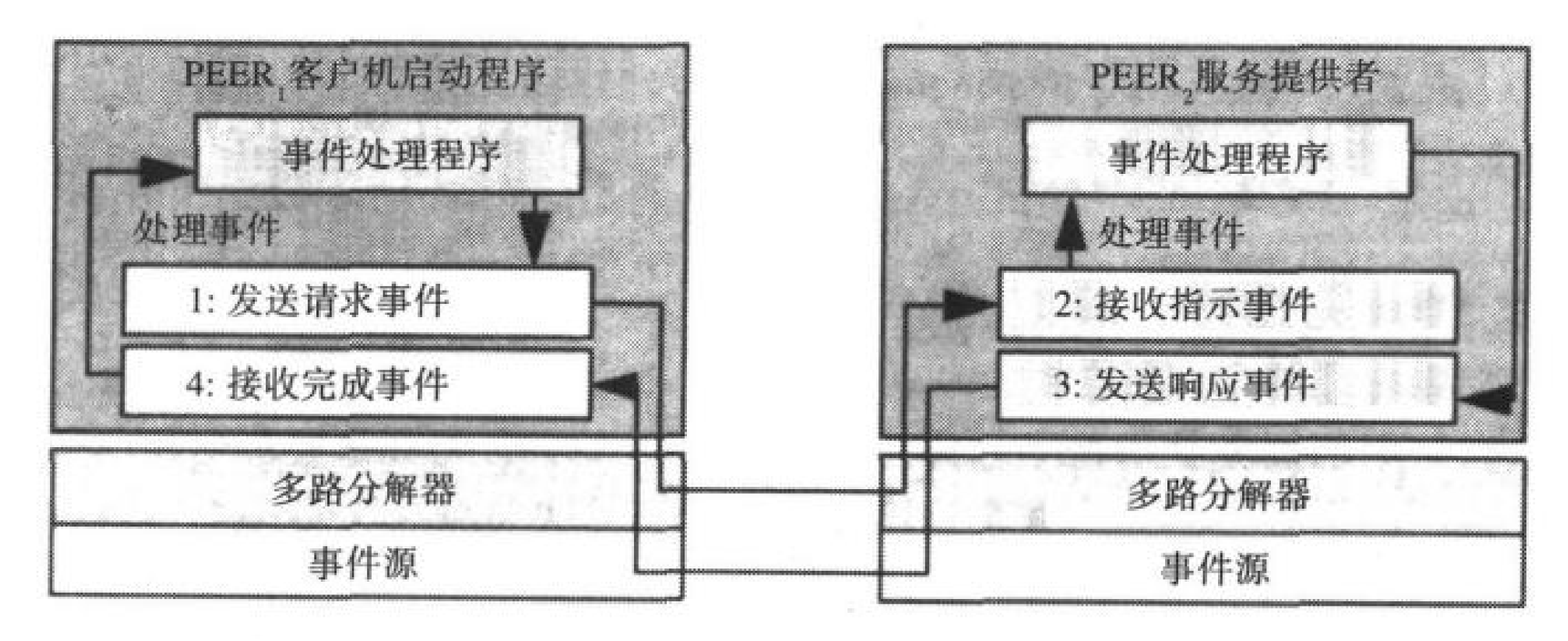


图 1-4

在该体系结构中，对等层间交换的事件扮演四个不同的角色[Bl91]：

- $PEER_1$，客户机启动应用程序，调用一个发送操作来传递请求事件给$PEER_2$，即服务提供者应用程序。事件可以包含$PEER_1$和$PEER_2$协作所必需的数据。例如，一个$PEER_1$请求可以包含一个CONNECT事件以启动一个双向连接，或者包含一个DATA事件以传递一个在$PEER_2$上远程执行的操作和它的参数。
- 通过一个指示事件通知$PEER_2$请求事件的到来。接着$PEER_2$可以调用一个接收操作去获取和使用指示事件数据来执行它的处理。$PEER_2$的多路分解层可以等待从多个对等层来的一系列指示事件。
- 当$PEER_2$完成处理指示事件后，它调用一个发送操作来传递一个响应事件给$PEER_1$，同时应答原始事件并返回任意结果。例如，$PEER_2$能把CONNECT事件作为一个初始化“握手”的一部分来应答，或者它能以一个可靠的双向远程方法调用来应答DATA事件。
- 通过一个完成事件通知$PEER_1$一个响应事件的到来。这时，它能使用一个接收操作来获取先前发送给$PEER_2$服务提供者的请求事件的结果。

15

如果在发送请求事件之后，$PEER_1$应用程序阻塞以接收包含$PEER_2$响应的完成事件，那么它被称为同步客户机。[⊖] 相反，如果$PEER_1$在发送请求后并不阻塞，那么它被称为异步客户机。异步客户机能通过异步机制接收完成事件，例如UNIX信号处理程序[Ste99]或者Win32 I/O完成

⊖ 在阅读本书中的模式时，要认识到术语“客户机”和“服务”并不是软件或者硬件组件不可改变的属性。相反，它们是在一个特殊的请求/响应交互期间扮演的角色[RG98]。例如，在一个对称的对等系统中，$PEER_1$和$PEER_2$在它们交互期间可以在不同的时间扮演客户机启动程序和服务提供者。

端口[Sol98]。

传统的网络化应用程序使用低层操作系统API检测、多路分解、分配各种类型的控制和数据事件，例如套接字[Ste98]、select()[Ste98]、poll()[Rago93]、WaitForMultipleObjects()和I/O完成端口[Sol98]。然而，使用这些低层API增加了事件驱动编程的偶发的复杂性。用这些低层API编程需要把I/O和应用程序的多路分解特征同其连接和并发机制进行耦合，从而增加了代码复制和维护工作。

第3章“事件处理模式”介绍了四种模式，这些模式描述了如何在网络化软件框架中有效地初始化、接收、多路分解、分配和处理各种类型的事件。这四种模式是反应器(Reactor)、主动器(Proactor)、异步完成标记(Asynchronous Completion Token, ACT)和接受器-连接器(Acceptor-Connector)。

1.2.3 难题3：并发

并发是涉及一系列策略和机制的术语[Ben90]，这些策略和机制能使一个或多个线程或者进程同时执行它们的服务处理任务。许多网络化应用程序，尤其是服务器，必须并发地处理从多个客户机来的请求。因此，并发网络化软件的开发人员常常需要精通各种进程和线程管理机制。

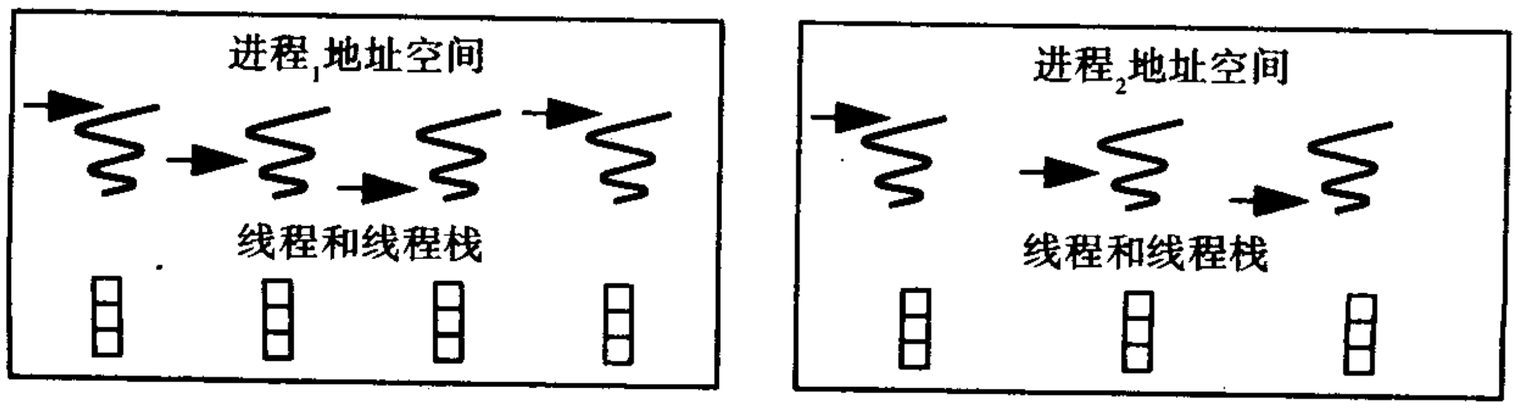
16

进程(process)是资源的集合，例如虚拟内存、I/O句柄和信号处理程序，此集合为执行的程序指令提供语境。在早期版本的操作系统中，例如BSD UNIX[MBKQ96]，进程有一个单一的控制线程。

线程(thread)是一个在进程的语境中执行的单一指令步骤的序列[Lew95]。除指令指针外，线程还包括其他资源，如运行时的函数激活记录的堆栈、一组寄存器和线程特有的数据。

使用单线程的进程将简化某些类型的并发应用，例如远程登录，这是因为在没有明显的编程人员干涉的情况下，独立的进程是不能和其他进程交互的。然而，使用单线程的进程开发网络化应用程序是极其困难的。例如，单线程的BSD UNIX服务器在不降低对其他客户机的响应性的情况下，处理一个客户机请求时，不能为阻塞延长时间。尽管使用类似信号驱动的套接字I/O或者分流多进程技术，有可能回避这些限制，但这样的程序既复杂又低效。

现代操作系统通过提供多线程的并发机制克服了单线程的进程的局限，此机制支持多个进程的创建，并且每个进程都可以包含有多个并发的线程。在这些操作系统中，在硬件保护的地址空间内，进程作为保护和资源分配单元。类似地，在被其他线程共享的进程地址之内，线程作为执行单元，如图1-5所示。



17

图 1-5

流行的线程编程模型，例如POSIX Pthreads[IEEE96]和Win32线程[Sol98]，提供四大优点：

- 它们使用硬件和软件平台的并行处理能力，透明地提高性能。
- 它们允许编程人员重叠计算和通信服务处理，明显地提高性能。
- 它们通过在一个应用程序中用独立的线程与不同的服务处理任务相关联，为交互式应用（如GUI）改进可察觉的响应时间。
- 它们通过使用同步编程抽象（如双向方法调用）允许多个服务处理任务独立地运行，从而简化应用程序设计。

然而，开发有效的、可预测的、可伸缩的和健壮的并发应用程序是非常困难的[Lea99a]。复杂性的一个来源是常见的多线程的事故，例如竞争条件和死锁[Lea99a]。复杂性的另一个来源是开发方法、工具和操作系统平台的局限，尤其是流行的硬件和软件平台的异构性，使开发在多种操作系统上运行的并发应用程序和工具变得极其复杂。

例如，轻松随意地关闭多线程的程序是困难的。问题源于不一致的跨操作系统的线程注销语义[Lew95]，例如POSIX/UNIX、Win32和类似VxWorks或者LynxOS的嵌入式实时系统。类似地，对高级线程特性的支持，例如线程特定的存储器、“独立的”线程[Lew95]、实时调度[Kan92]和调度器激活[ABLL92]等在各个操作系统上变动很大。因此通过直接编写操作系统API来编写可移植的并发应用程序是不可行的。

通用设计模式，例如适配器[GoF95]和包装器外观，可以用于保护并发软件避免上述的API的偶发的复杂性。另外，商用的“基础设施”中间件[SFJ96]，例如ACE[Sch97]和JVM，现在是广泛可用的，并已经把这些模式具体化为有效的和可复用的面向对象的操作系统封装层。然而，即使采用这种级别的中间件，仍然存在许多难题，这是由于并发应用程序开发的固有的复杂性。它们包括： 18

- 在并发应用程序中，确定一个有效的应用程序并发性体系结构[SchSu95][SKT96]，此结构最小化并发应用程序的语境交换、同步、数据复制/移动开销。
- 在不降低执行效率的情况下，设计包含同步和异步服务处理任务的复杂的并发系统以简化编程[Sch96]。
- 选择合适的同步原语，使多处理机的并发应用程序提高性能、防止资源浪费、减少维护费用[McK95]。
- 在并发[HPS99]或者实时[HLS97]应用程序中消除不必要的线程和加锁，在不危及正确性、招致死锁或者过度地阻塞应用程序处理的情况下，增强性能或者简化资源管理。

解决这些固有的复杂性不仅要求通用设计模式和可移植的基础设施中间件线程API，还要求其他东西。而且，它需要开发人员学习并把开发并发应用程序、组件、框架和系统体系结构的成功模式纳入其中。

第5章“并发模式”包括5种模式，这些模式为组件、子系统和整个应用程序定义了各种类型的并发体系结构。这5种模式是活动对象(Active Object)、监视器对象(Monitor Object)、半同步/半异步(Half-Sync/Half-Async)、领导者/追随者(Leader/Follower)和线程特定的存储器(Thread-Specific Storage)。

1.2.4 难题4：同步

许多网络化应用程序的效率、响应度和设计可以从上述的并发机制和模式中受益。例如，应用程序中的对象可以在不同的线程中并发地运行以简化程序结构。如果可用多个处理器，那么可以在编程中使线程真正利用硬件并行性提高性能。

19

除在第1.2.3节中概述的复杂性之外，并发编程也比顺序编程更为困难，这是由于同步访问共享资源的需求。例如，并发运行的线程能够同时访问同一对象或者变量，从而可能破坏了它们的内部状态。为了防止这个问题，在对象或者函数中不应同时执行的代码可以在*临界区*(critical section)范围内被同步。一个临界区是遵守下列不变式的指令序列：当一个线程或者进程在临界区执行时，没有其他的线程或者进程能够在同一临界区执行[Tan95]，如图1-6所示。

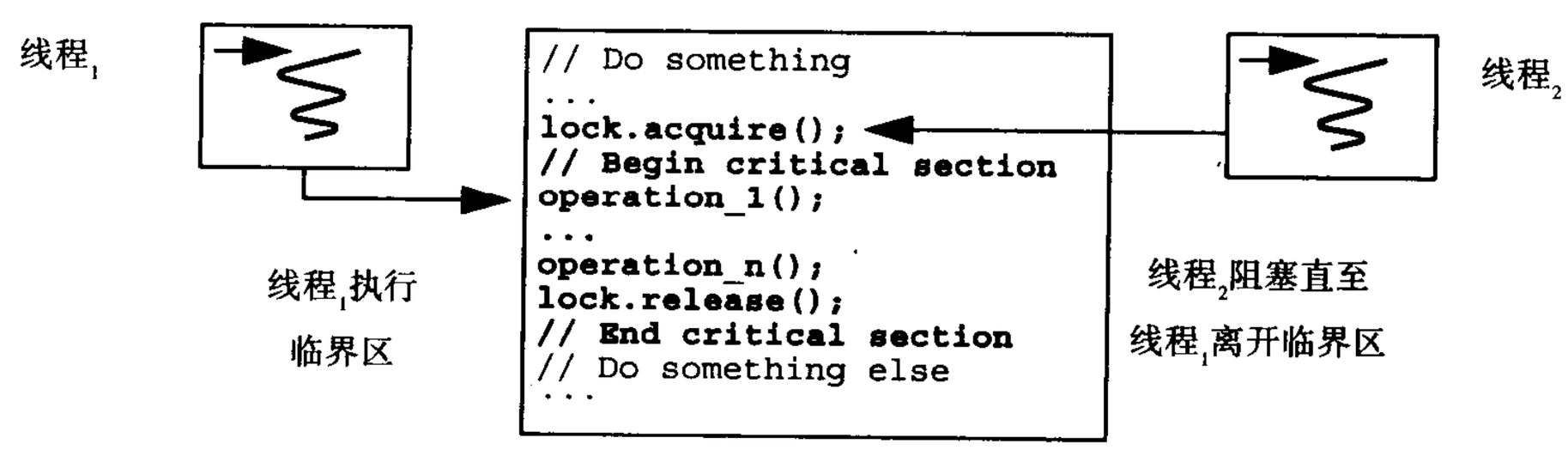


图 1-6

在面向对象的程序中实现临界区的常用方法是把某种类型的加锁对象硬编码到一个类或者组件。例如，一个互斥（mutual exclusion，mutex）对象是一种必须串行地获取和释放的锁。如果多个线程企图同时获取互斥对象，那么只有一个线程会成功。其他的线程必须等待，直至互斥对象被释放，才可以再次竞争锁[Tan92]。其他类型的锁，例如信号灯和阅读器/记录器锁，使用相似的获取/释放协议[McK95]。

不幸的是，使用低层操作系统API编写这些加锁技术有两个缺点：

- *容易出错*。要在进入临界区之前显式地获取一个锁，并且在退出临界区时显式地释放它，这相当困难。尤其是如果临界区有多个返回路径，那么必须在所有返回路径显式地释放锁。这种做法常常产生不易查觉的编程错误，因为很容易在某个返回路径中忘记释放锁，尤其是如果在阻塞中抛出异常。而如果锁没被释放的话，随后的线程进入临界区时将无限期地阻塞，从而发生死锁。

20

- *不灵活和效率低*。依赖于应用程序运行的语境，性能需求会将不同的加锁类型用于实现临界区。例如，如果一个最初使用互斥的应用程序在一个大规模的多处理程序平台上运行，那么通过把加锁机制改为阅读器/记录器锁可以提高性能。这种类型的锁允许多个阅读器线程并行地访问一个共享资源[McK95]。然而，如果加锁原语在使用的每个地方都被硬编码到软件，那么改变原语完全没有必要如此困难且耗费时间。

第4章“同步模式”描述4种模式，这些模式能减轻上述的问题，以简化并行系统中的串行化和加锁。这些模式是*定界加锁*(Scoped Locking)、*策略化加锁*(Strategized Locking)、*线程安全*

接口(Thread-Safe Interface)和双检查加锁优化(Double-Checked Locking Optimization)。

1.2.5 网络化软件的其他难题

上面所讨论的四个领域——服务访问和配置、事件处理、并发和同步——代表本书中模式所针对的网络化软件开发的核心难题。然而，网络化应用软件的开发人员必须面对其他领域中的一些问题，例如依赖性、服务命名、位置选择。尽管这些主题不是本书讨论的范围，但我们还是在此概述这些重要难题以阐明领域的范围。

依赖性（dependability）。采用网络化体系结构的一个原因是提高可靠性(reliability)和防止单点失效。具有讽刺意味的是，网络化应用程序常常需要大量的工作，才能获得与独立应用程序所提供的相同的可靠性。在独立应用程序中检测服务失效相对而言是容易的，因为操作系统了解全局的系统服务和外设的工作状态。因此，如果某个资源是不可用的，那么操作系统能迅速通知应用程序。类似地，如果某个服务或者设备失效，那么操作系统能终止一个应用程序，使之进入退出状态。 21

但是，在网络化应用程序中检测错误更加困难，这是由于对全局系统状态的不完全知识。例如，网络化应用程序被设计为容忍一定数量的等待时间抖动和非决定性。结果，直至有价值的信息丢失之后，客户机才会检测到一个异常的服务器终止。同样，在网络中服务器响应也许会丢失，造成客户机重复发送同样的请求。

有以下几个改进应用程序依赖性的技术：

- 重新激活(reactivation)。应用程序和服务可以在监视器守护进程(monitor daemon)的控制下运行，如果服务器意外地终止，那么这些守护进程会检测到并且自动地重新启动服务器[HK93]。服务器通过“脉搏”消息定期地向相关联的监视器守护进程报告它们的当前状况。如果消息没有在指定的时间间隔内到来，那么监视器守护进程就认为服务器已经异常终止，并重新激活它。
- 复制(replication)。应用程序和服务能在遍及网络的、多个方位上的拷贝管理器(replica manager)的控制下运行[GS97]。拷贝管理器可以使用“主动复制”连续地更新服务拷贝，或者正好当一个主服务失效时使用“被动复制”。复制框架[FGS98]提供各种各样的监视、成员资格、一致性和消息传送机制来帮助增强应用程序依赖性。

有大量的研究文献和工具研究改进进程[GS97][BR94][HK93]或者分布式对象[CRSS+98][FGS98]的依赖性。尽管许多研究仍在继续，但一些工作已被以模式形式记载下来[IM96][Maf96][SPM98][ACGH+96][Sta100]。随着容错的CORBA规范[OMG99g]和实现此规范的ORB的采用，更多的应用程序开发人员将能够用容错的分布式对象计算的模式记录下他们的经验。

服务命名和位置选择。独立应用程序通常通过对象和函数内存地址标识它们的组成服务。而网络化应用程序需要更精心的机制来命名、定位和选择它们的远程服务。IP主机地址和TCP端口号是被CORBA、DCOM、Java RMI、DCE和SunRPC使用的远程服务寻址方案。然而，这些低层机制对于大规模的网络化系统而言常常是不适合的，因为难以以可移植的和无歧义的方式管理它们。例如TCP端口5000不必引用在主机上由不同的厂家或者网络管理员配置的同一服务。 22

因此，分布式对象计算和RPC中间件提供位置代理者(location broker)，它允许客户机通过更高级的命名而不是通过它们的低层IP地址和TCP端口号来访问服务。位置代理者通过自动化下列任务来简化网络化系统管理，并且在全网络范围内提供更灵活和动态的服务布局：

- 名字绑定。这个任务把服务名绑定到当前的主机/进程位置。例如，SunRPC的rpcbind功能程序在单一终端系统上执行端口映射任务[Sun88]。更一般的名字绑定机制，例如DCE的单元目录服务（Cell Directory Service，CDS）[RKF92]、LDAP[HSGH99]、X.500[SS99]和CORBA命名服务[OMG97a]也都是可用的。这些服务在管理域范围内实现一个全局的名字空间，例如局域网或者企业内部网。
- 服务定位。服务或者资源常常可以通过复制在整个网络的多个位置上运行，以提高可靠性。在这种情况下，应用程序可以使用一个位置代理者以确定哪个服务提供者更适合。例如CORBA业务服务允许客户机通过一系列与服务相关联的属性来选择远程对象[OMG98b]。客户机可以通过使用这些属性来选择一个适合的资源，例如通过确定办公楼内的哪个打印机具有Postscript支持、彩色打印、1200dpi分辨率和充足的纸张来选择打印机。

23

与名字绑定和服务定位相关的模式已经出现在[POSA1][Doble96][JK00]中。

1.3 案例分析：设计一个并发的Web服务器

由于允许终端用户轻而易举地访问大量内容的Web浏览器的激增[PQ00]，Web流量的总量迅速增长。类似地，Web技术日益被用于大计算量的任务，例如医学图像处理服务器[PHS96]和数据库搜索引擎。为满足不断增长的需求，开发能为互联网和内联网用户提供有效的高速缓存和内容发送服务的并发Web服务器成为必然。

图1-7是一个有代表性的Web系统和它的组件的结构：

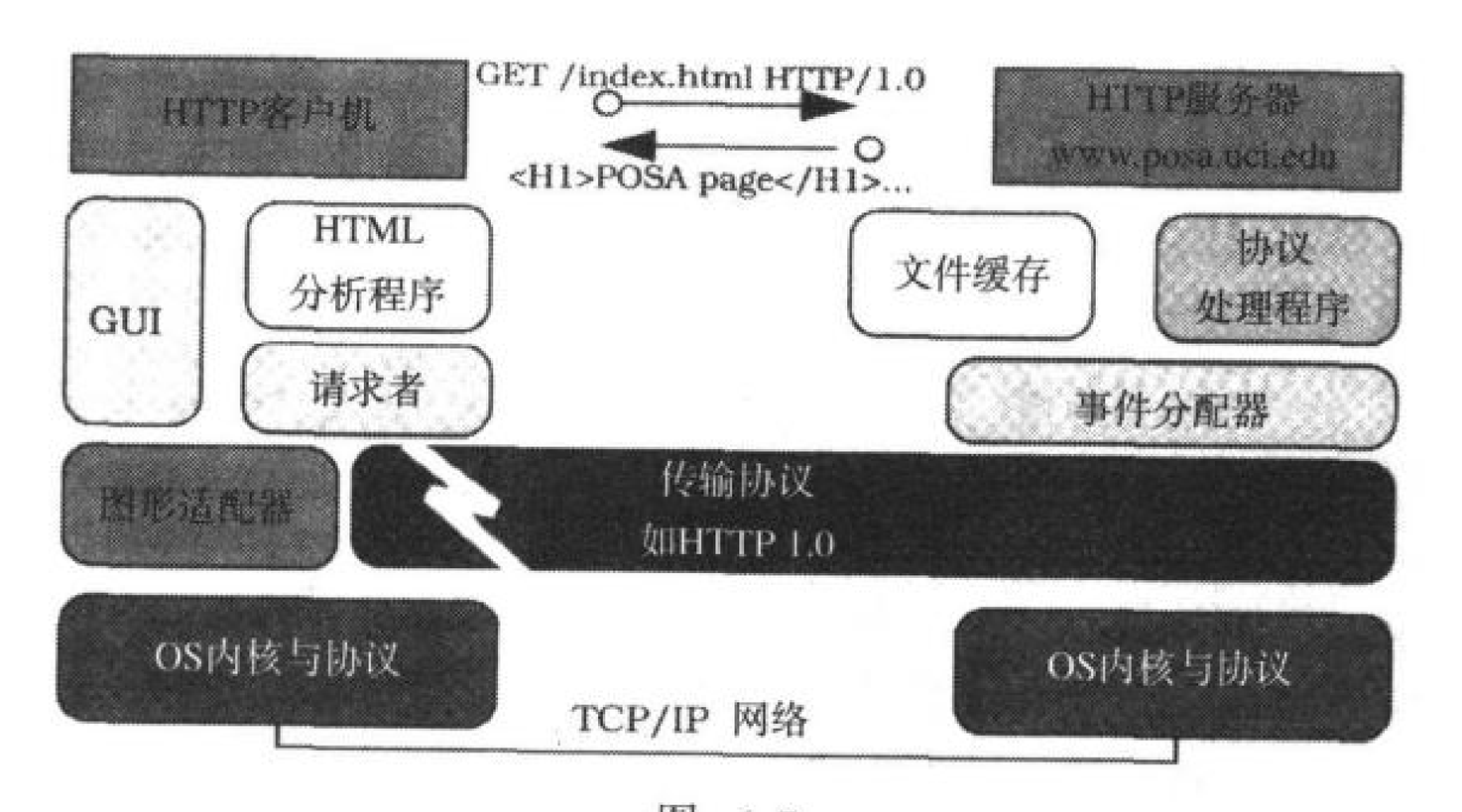


图 1-7

当HTTP客户机从HTTP服务器检索一个HTML文件时，这些Web系统组件进行如下交互。

1) 借助于Web浏览器通过GUI交互，终端用户指示HTTP客户机下载一个特殊的文件。

2) 请求者是跨越TCP/IP网络通信的HTTP客户机的活动组件。它使用合适的传输协议语法，

例如HTTP1.0[Ste96]，发送TCP连接事件和HTTP的GET请求事件，这些事件是通知服务器下载特殊文件的字符串。

3) 到达HTTP服务器的事件被事件分配器接收。它是服务器的多路分解引擎，接受TCP连接事件，协调用于接收并处理HTTP的GET请求事件的套接字句柄和线程。 24

4) 每个HTTP的GET请求事件被一个协议处理程序处理，它分析并记录请求，取出文件状态信息，更新文件缓存，把文件传输回HTTP客户机，并清除它分配的任何资源。

5) 当被请求的文件返回给客户机时，它被HTML分析程序分析，此分析程序解释并交付文件。此刻，请求者可以代表客户机发送其他的请求，例如更新客户端缓存或者下载嵌入的图像。

当开发人员创建和优化Web服务器时，他们必须避免一些常见的问题。例如，被低层的编程细节和可移植性约束所纠缠、过早地采用一个特定的服务器配置，并在大量的可供选择的设计方案面前不知所措，这些方案如：

- 并发模型，比如是每个请求一个线程还是线程池变体。
- 事件多路分解模型，例如用同步的还是异步的事件多路分解。
- 文件缓存模型，例如是未用最近最少用（least-recently used，LRU）模型还是最不常用（least- frequently used，LFU）模型。
- 内容发送协议，例如是HTTP/1.0[BFF96]、HTTP/1.1[FGMFB97]还是HTTP-NG[W3C98]。

之所以面临如此多的选择，是为确保Web服务器适合不同的终端用户需求和流量工作负载。然而，不存在单一的配置选择集对所有硬件/软件平台和工作负载都是优化的[HPS99][HMS98]。而且，在没有正确指导的情况下，在所有这些可供选择的设计中摸索是耗费时间和易出错的。

本节剩下的内容举例说明本书中的八个模式如何应用于产生一个名为JAWS的、灵活的和有效的Web服务器[HPS99]。JAWS既是一个Web服务器，又是一个框架，从中能够构建其他类型的服务器[HS98]。 25

我们选择JAWS作为应用例子有四个方面的原因：

- 它是保证质量的Web服务器[ENTERA00]，体现了开发并发和网络化软件时出现的主要难题。
- JAWS的吞吐量和可伸缩性高，而等待时间和抖动低[HPS99][HMS98]，证明面向模式的和面向框架的软件体系结构是有效的。
- JAWS框架本身是使用ACE框架[Sch97]开发的，此框架提供本书中大部分面向对象的模式的实现。
- ACE和JAWS是开放源代码的，[⊖]因此你能真切地看到如何利用模式避免重新发现和重复发明并发和网络化软件设计问题的解决方案。

1.3.1 JAWS框架概述

在JAWS中存在三个主要的框架组件，如图1-8所示。

⊖ JAWS和ACE的源代码可以在http://www.posa.uci.edu/下载。

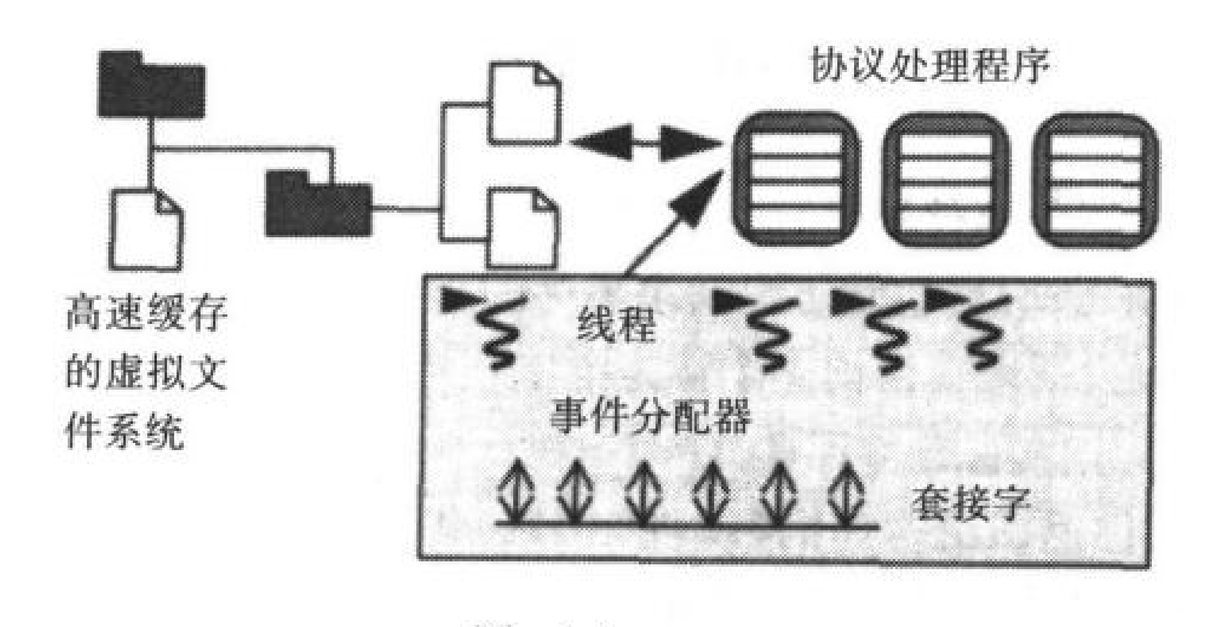


图 1-8

- 事件分配器(Event Dispatcher)。这个组件接收客户机的连接请求事件、接收HTTP的GET请求，并用它的并发策略协调JAWS的事件多路分解策略。当处理事件时，把事件分配给合适的协议处理程序。
- 协议处理程序(Protocol Handler)。这个组件实现HTTP请求事件的分析和协议处理。JAWS协议处理程序的设计允许把多个Web协议合并到一个Web服务器，例如HTTP/1.0、HTTP/1.1和HTTP-NG。为了增加一个新的协议，开发人员只需编写一个新的协议处理程序组件，并把它配置进JAWS框架。
- 高速缓存的虚拟文件系统(Cached Virtual Filesystem)。这个组件在处理HTTP的GET请求时，通过减少文件系统访问的开销来提高Web服务器性能。按照实际的或者预测的工作负载静态地或者动态地选择和配置各种缓存策略，例如最近最少用法（LRU）或者最不常用法（LFU）。

26

1.3.2 在JAWS中应用模式解决JAWS中常见的设计难题

上面对JAWS框架体系结构的概要描述了JAWS是如何构造的，但并没有解释为什么这样构造。理解为什么JAWS框架包含这些特殊的组件，为什么用那样的方法设计组件，这通常需要更深的并发和网络化软件领域底层的模式的知识，尤其是并发Web服务器。

在JAWS中使用8种模式实现主要的组件，如图1-9所示。

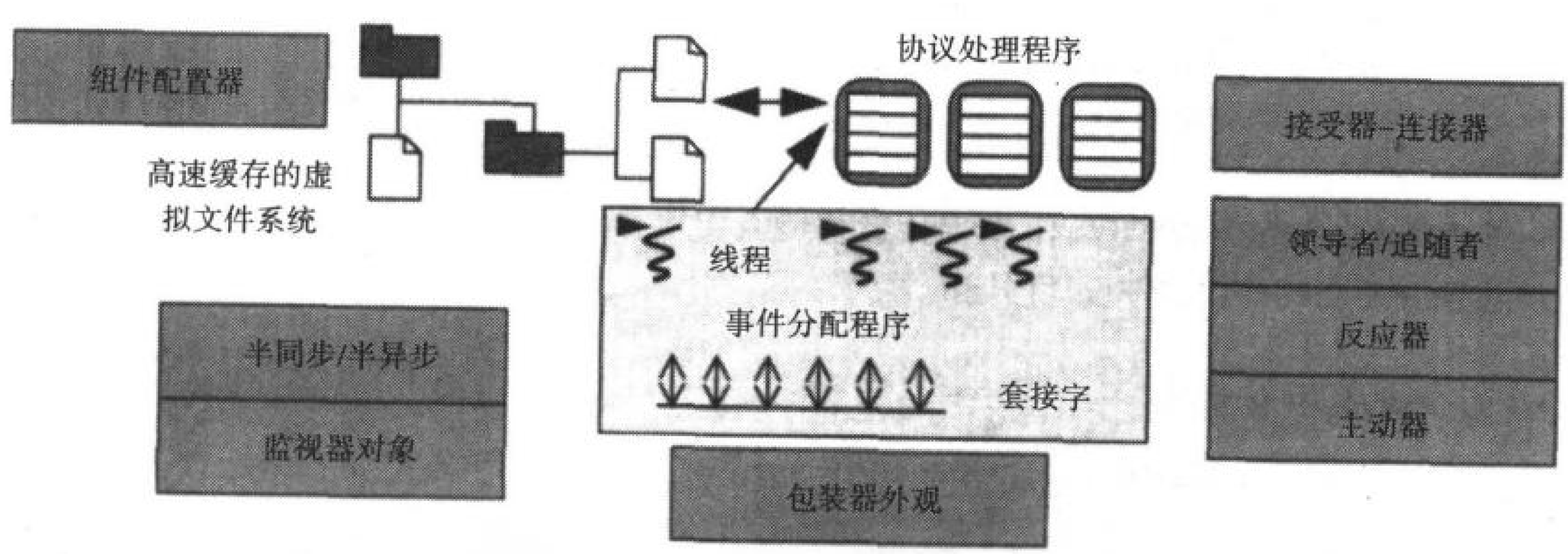


图 1-9

这些模式有助于解决在开发并发服务器时出现的下面七个常见难题：

- 封装低层操作系统API。
- 从协议处理中分解事件多路分解和连接管理。
- 通过多线程提升服务器性能。
- 实现一个同步化的请求队列。
- 最小化服务器线程的开销。
- 有效地使用异步I/O。
- 增强服务器的可配置性。

27

除使用一个基本的“语境/问题/解决方案”形式描述模式之外，我们注意兼顾讲解模式本身和显示如何使用这些模式开发并发的JAWS的Web服务器。第2章到第5章将从更通用的角度更详细地描述这些模式。

1.3.3 封装低层操作系统API

(1) 语境

Web服务器必须管理各种各样的操作系统服务，包括进程、线程、套接字连接、虚拟内存和文件。大部分操作系统，例如Win32或者POSIX，都提供C编写的访问这些服务的低层API。

(2) 问题

各种硬件和操作系统的差异使得难以直接通过对低层操作系统API编程来构建可移植的、健壮的Web服务器软件。这些API枯燥乏味，容易出错，不可移植，用它们开发Web服务器或者其他网络化应用程序效率不高。

(3) 解决方案

运用包装器外观模式来避免直接访问低层操作系统API。这种设计模式把现有的非面向对象API提供的函数和数据（例如低层操作系统API），封装在更加简洁的、健壮的、可移植的、可维护的和聚合的面向对象的类接口之内。

(4) JAWS中的使用

JAWS使用ACE定义的包装器外观以确保它的框架组件能够在多个操作系统上运行，包括Windows、UNIX和许多实时操作系统。

例如，JAWS使用ACE的`Thread_Mutex`包装器外观提供一个操作系统互斥机制的可移植的接口（如图1-10）。

28

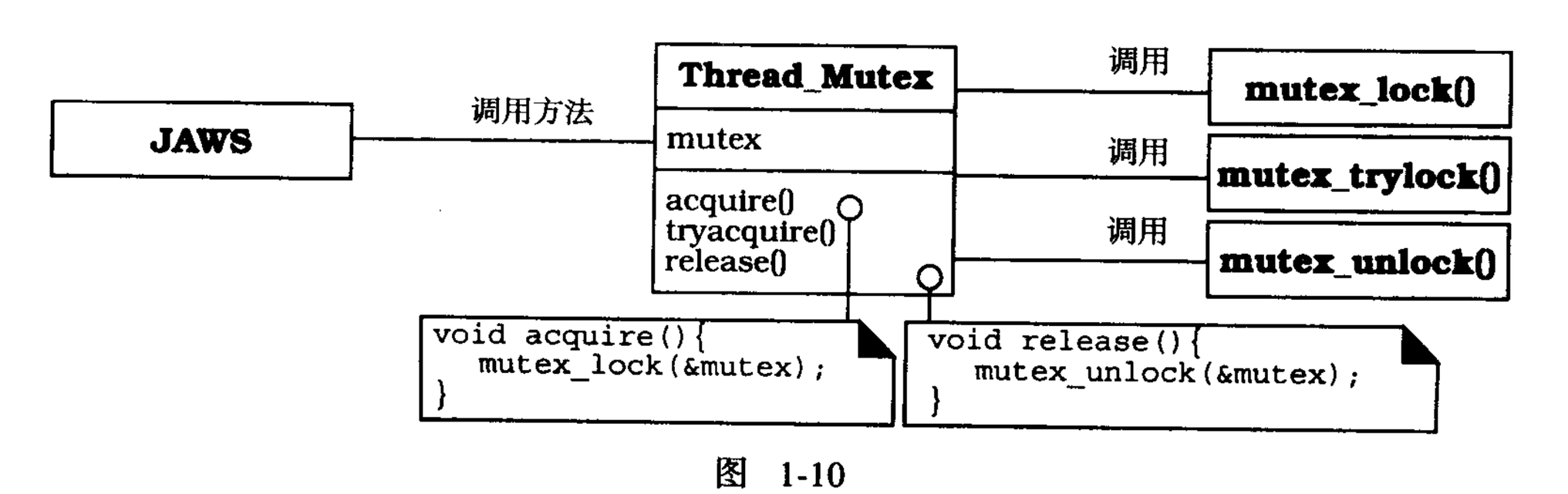


图 1-10

我们使用Solaris线程API[EKBF+92]实现框图中显示的Thread_Mutex。然而，ACE的Thread_Mutex包装器外观对于其他线程API也是可用的，例如Win32线程或者POSIX的Pthreads。在JAWS中使用的其他ACE包装器外观封装套接字、进程和线程管理、内存映射文件、显式的动态链接和定时操作等[Sch97]。

1.3.4 从协议处理中分离事件多路分解和连接管理

(1) 语境

多个客户机可以同时访问一个Web服务器，每个客户机有各自对服务器的连接。因此，一个Web服务器必须能够多路分解和处理多种类型的指示事件，这些事件可以从不同的客户机并发地到来：

- 一个连接请求，服务器通过CONNECT指示事件接收此连接请求，此事件指示服务器接收客户机连接。
- 一个下载文件的HTTP的GET请求，服务器通过READ指示事件接收此请求，此事件指示服务器接收从它的某个客户机连接中来的请求。

在一个Web服务器中多路分解事件的常用方法是使用select()[Ste98]。这个函数报告哪个套接字句柄有等待解决的指示事件，以便在不阻塞服务器的情况下调用套接字操作，例如接收客户机连接请求的accept()或者接收客户机请求的recv()。

(2) 问题

开发人员常常把一个Web服务器的事件多路分解和连接管理代码与它的执行HTTP 1.0处理的协议处理代码紧密地结合起来。然而，在这样的一个设计中，多路分解和连接管理代码不能被其他的HTTP协议作为黑盒组件复用，也不能被其他的中间件和应用程序使用，例如ORB[SC99]和图像服务器[PHS96]。而且，对事件多路分解和连接管理代码的改变，如把它改变为使用TLI[Rago93]或者WaitForMultipleObjects()值[Sol98]，将直接影响Web服务器的协议代码，并可能引入不易觉察的错误。

29

(3) 解决方案

运用反应器模式和接受器-连接器模式从HTTP协议代码中分离通用的事件多路分解和连接管理代码。反应器体系结构模式从服务中分解服务器应用程序的同步事件多路分解和分配逻辑，如HTTP协议处理，执行对事件的响应。接受器-连接器设计模式可以建立在反应器模式上，以便从处理活动中分解合作的对等服务的连接和初始化，例如HTTP客户机和服务器，一旦这些对等服务被连接和初始化，它们就执行这些处理活动。

(4) JAWS中的使用

JAWS在没有轮询它的所有事件源或者在任何单一事件源无限期阻塞的情况下，使用反应器模式处理从多个事件源来的多个同步事件。类似地，它使用接受器-连接器模式独立地改变它的连接-管理代码中的协议处理代码（如图1-11）。

在图1-11的设计中，select()同步事件多路分解器等待句柄集上事件的发生。当事件到来时，select()通知一个反应器，接着反应器多路分解和分配这些事件给一个指定的事件处理程序做进一步处理。

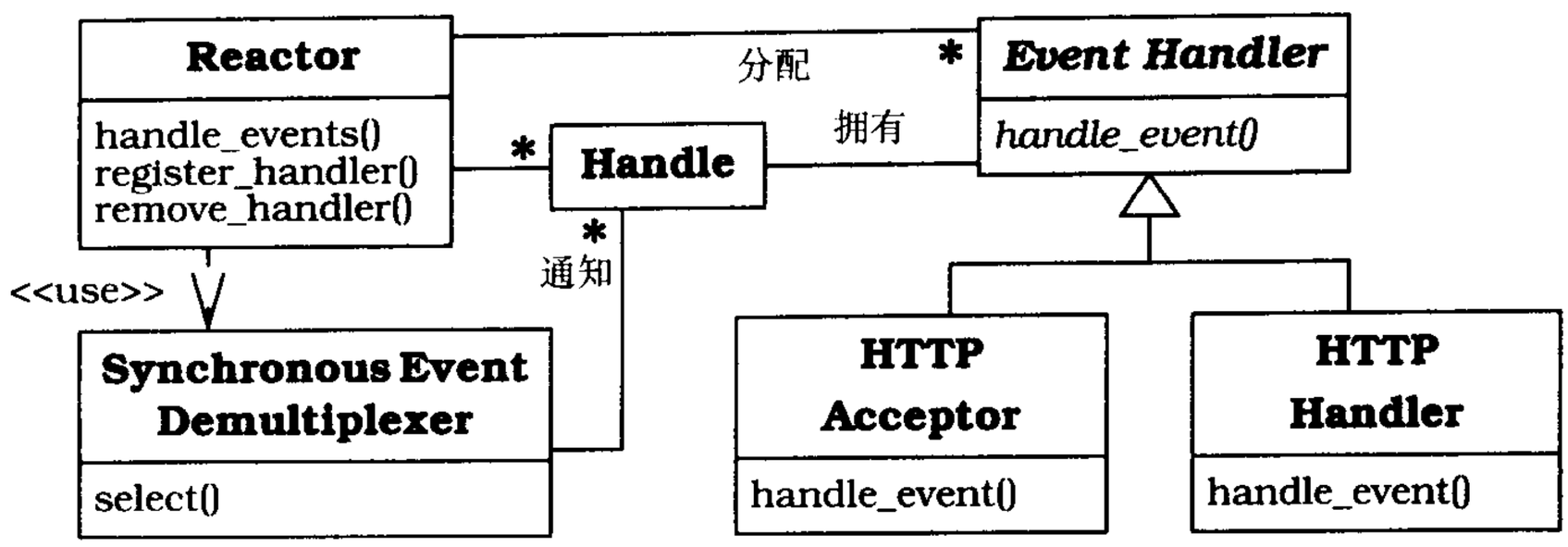


图 1-11

30

在JAWS中存在两种类型的事件处理程序：

- HTTP_Acceptor为CONNECT事件向反应器注册自己。当这些事件发生时，反应器调用HTTP_Acceptor的handle_event()钩子方法，接着handle_event()钩子方法创建、连接和激活一个HTTP_Handler。
- 每个HTTP_Handler是一个协议处理程序，它的handle_event()钩子方法负责接收和处理与其连接的客户机发送的HTTP GET请求。

通过使用反应器和接受器-连接器模式，从事件分配器中的协议无关的事件多路分解和连接管理代码中，分离出HTTP_Handler中与协议有关的处理代码。这种设计使在JAWS中维护和复用各种组件更加容易。

1.3.5 通过多线程提高服务器性能

(1) 语境

HTTP在一个使用流控制的传输协议TCP上运行，流控制确保发送方产生数据的速度不快于慢的接收方或者拥挤的网络能够缓冲和处理的速度[Ste93]。处理繁重的Web流量负载[PQ00]需要获得有效的端到端服务质量(QoS)，因此，一个Web服务器必须随着客户机数目的增长而有效地提高性能。

(2) 问题

在一个单一线程的进程内往复地处理所有HTTP GET请求并不能有效地提高性能，因为每个Web服务器CPU时间片在等待I/O操作完成上花费太多的阻塞时间。类似地，为所有连接的客户机提高QoS，整个Web服务器进程在发送文件给客户机时，不能阻塞以等待连接流控制的减少。

(3) 解决方案

通过在多个线程中并发地处理不同的HTTP请求，运用半同步/半异步模式提高服务器性能。这种体系结构的模式定义两个服务处理层（一个异步，一个同步）和一个排队层（允许服务在两个层之间交换消息）。相对于其他同步服务和异步服务（例如事件多路分解），模式允许同步服务（如HTTP协议处理）并发地运行。

31

这个解决方案有两个优点：

• 可以把线程映射给分离的CPU，通过多处理来提高性能。

• 每个线程独立地阻塞，它防止流控制的一个连接降低其他客户机接收的QoS。

(4) JAWS中的使用

JAWS可以使用半同步/半异步模式同时处理从多个客户机来的HTTP GET请求，但这种处理在各自线程的控制下并发地进行，如图1-12所示。

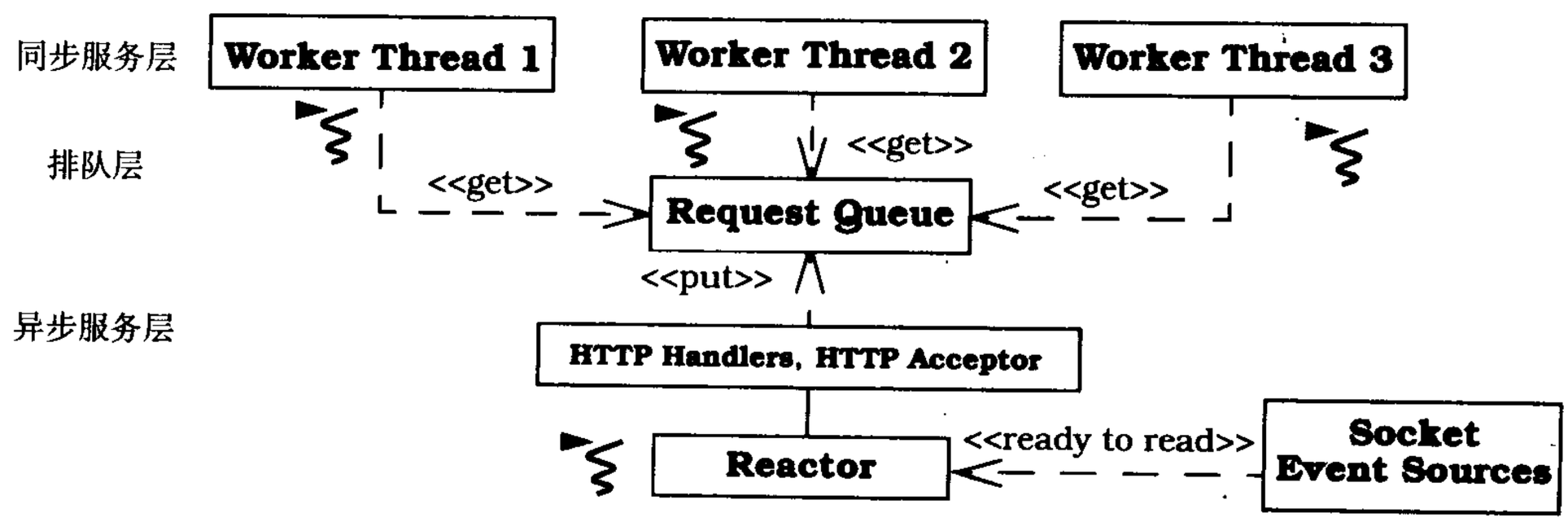


图 1-12

图1-12中的反应器的HTTP_Handler组成JAWS的“异步”层中的服务。尽管反应器并不是真正地异步，但它和异步服务共享关键属性。例如，反应器分配的HTTP_Handler在不处理其他客户机的情况下不能长久地阻塞。因此，在这种设计中，HTTP_Handler只是读进来的HTTP GET请求，并把它插入到工作者线程池服务的请求队列中。

删除这些请求的工作者线程同步地执行HTTP协议处理，然后把文件传回给客户机。如果流控制出现在它的客户机连接上，那么这个线程可能阻塞而不降低服务质量（QoS），这里的QoS是池中其他的工作者线程服务的客户机体验到的。

1.3.6 实现一个同步化的请求队列

(1) 语境

在半同步/半异步模式的中心是一个排队层。在JAWS中，反应器线程是一个“生产者”，它把HTTP GET请求插入到队列。池中的工作者线程是“消费者”，它从队列中删除请求并处理请求。

32

(2) 问题

当多个线程插入和删除请求时，请求队列的自然实现将产生竞争条件或“忙等待”。例如，如果没有被正确地同步，那么多个并发的生产者和消费者线程会破坏队列的内部状态。类似地，当队列是空或者满时，这些线程将“忙等待”，它毫无必要地浪费CPU周期。

(3) 解决方案

运用监视器对象模式来实现一个同步化的请求队列。这种设计模式同步化方法执行程序，以确保在一个对象内每次仅有一个方法运行，例如Web服务器的请求队列。另外，它允许对象的方法合作式地调度它们的执行顺序。例如，当请求队列是空或者满时，可以使用一个监视器对

象来防止线程“忙等待”。

(4) JAWS中的使用

JAWS同步化的请求队列使用一对POSIX条件变量来实现队列的非空和非满监视器条件。可以把这个同步化的请求队列集成到JAWS事件分配器的半同步/半异步线程池实现中（如图1-13）。

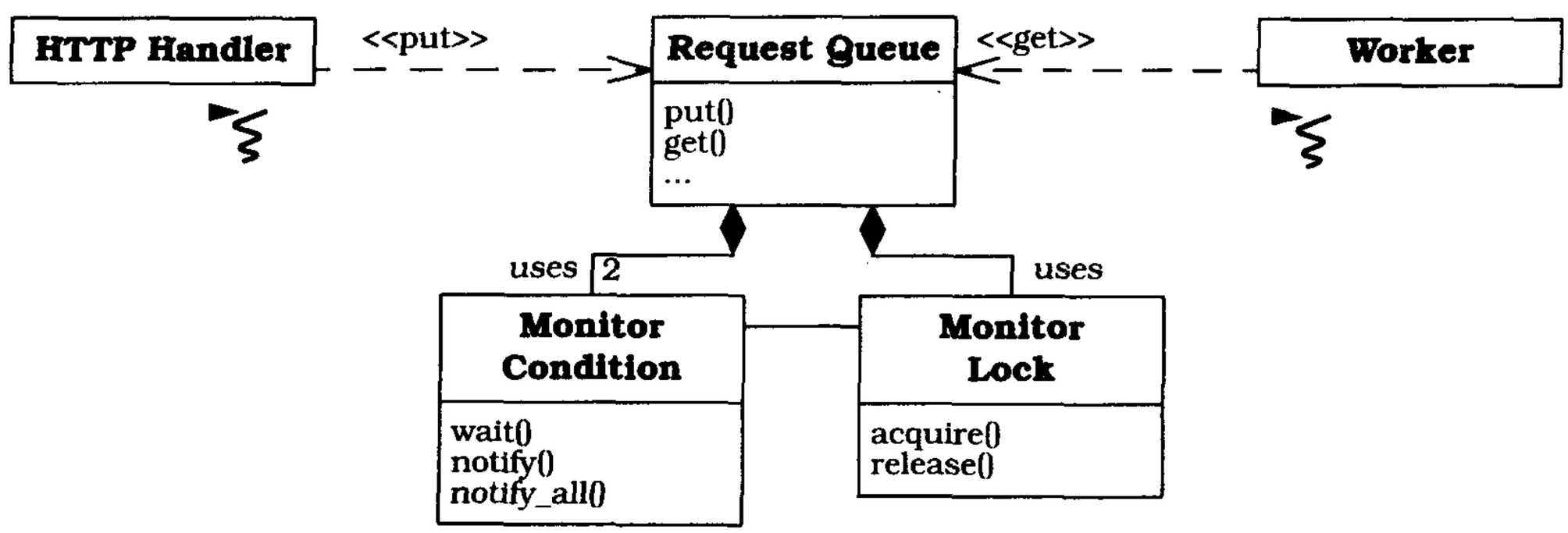


图 1-13

在图1-13中，当工作者线程试图从一个空的队列中解队一个HTTP GET请求时，请求队列的get()方法自动地释放监视器锁，同时工作者线程在非空监视器条件挂起自己。它保持挂起直至队列不再为空，即运行在反应器线程内的HTTP_Handler把一个请求插入到队列中来。 33

1.3.7 最小化服务器线程的开销

(1) 语境

在某些多线程的操作系统（例如Windows NT和Solaris）中套接字的实现，提供了一个并发的accept()优化[Ste98]以接受客户机连接请求。这种优化在三个方面提高了实现HTTP 1.0协议的Web服务器的性能：

- 操作系统允许Web服务器线程池中的线程调用同一被动模式的套接字句柄上的accept()。
- 当连接请求到来时，操作系统的传输层创建一个新的被连接的传输端点，用一个数据模式的套接字句柄封装这个新的端点，并传送句柄作为accept()的返回值。
- 接着操作系统调度线程池中的某一个线程去接收这个数据模式的句柄，线程使用这个句柄和它连接的客户机通信。

(2) 问题

在1.3.5节中描述的半同步/半异步线程模式比在1.3.4节描述的纯粹地往复的模型更具伸缩性。然而它不一定就是最有效的设计。例如，它招致一个动态的内存分配、多个同步操作、语境交换、缓存更新以便在反应器线程和工作者线程之间传递请求。这个开销使JAWS的等待时间不必要地延长，尤其是在“语境”讨论所概述的支持并发accept()优化的操作系统中。

(3) 解决方案

运用领导者/追随者模式最小化服务器线程的开销。这种体系结构的模式提供一个有效的并

发模型，在模型中多个线程轮流共享事件资源（例如一个被动模式的套接字句柄），以便检测、多路分解、分配和处理发生在事件源上的服务请求。这种模式消除了在JAWS服务器中对分离的反应器线程和同步化的请求队列的需求和开销。

34

(4) JAWS中的使用

可以通过领导者/追随者线程池设计实现JAWS的事件分配器和协议处理程序，如图1-14所示。

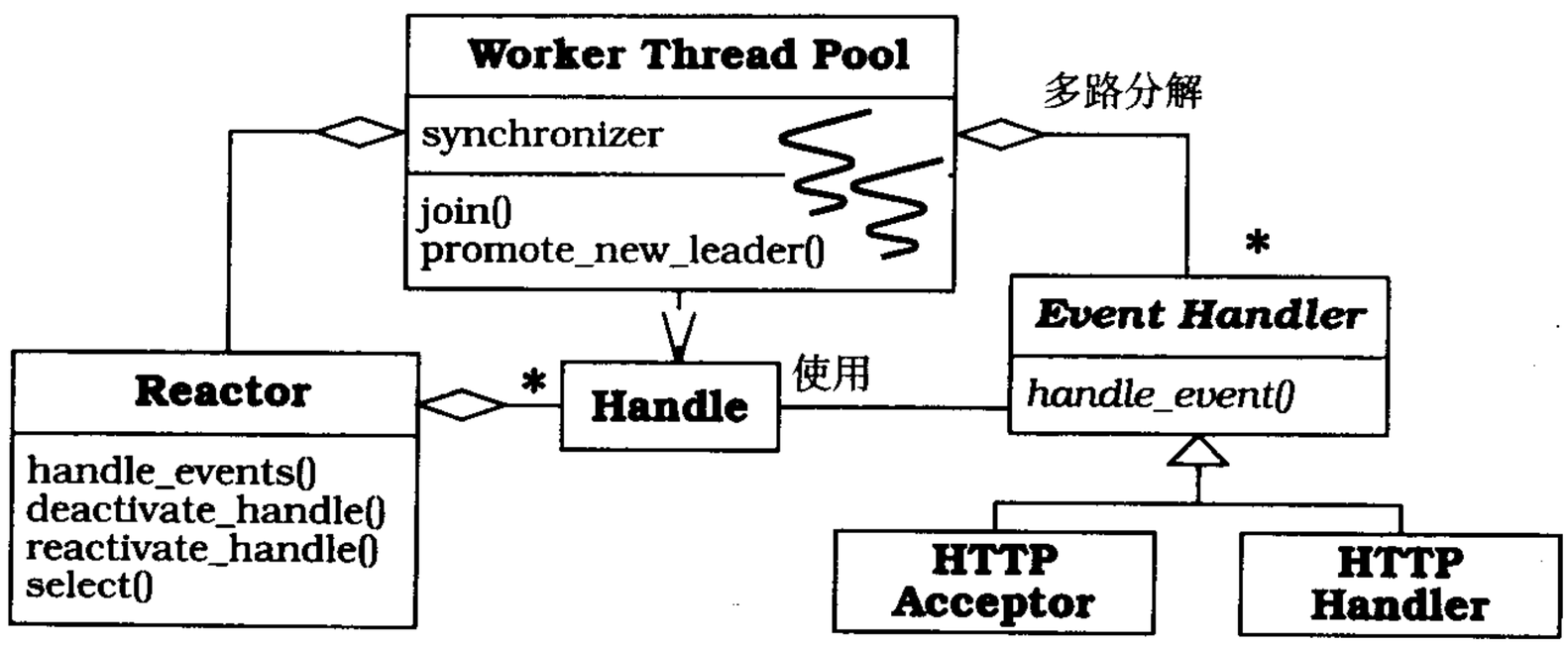


图 1-14

在这种设计中，池中多个工作者线程共享同一被动模式的套接字句柄。有两种变化需要考虑：

- 如果操作系统支持在“语境”段中描述的并发accept()优化，那么所有的工作者线程可以在同一句柄上调用accept()。接着操作系统线程调度器通过使用上面概述的步骤确定分配给HTTP_Handler的HTTP GET请求的次序。现在每个HTTP_Handler在它自己的控制线程中运行，所以它能在不阻塞正在处理客户机请求的其他线程的情况下，同步地执行它的I/O操作。
- 如果操作系统不支持accept()优化，那么可以使用一个不同的领导者/追随者实现来共享被动模式的套接字句柄。在这种设计中，每次一个线程，即领导者线程调用反应器的handle_events()方法，在被动模式的套接字句柄上等待连接的到来。其他的线程，即追随者线程在同步器上排队，等待轮到它们成为领导者。

  在当前的领导者线程接收一个最近连接的套接字句柄之后，它提升一个追随者线程成为新的领导者。接着它充当处理线程的角色，使用反应器多路分解和分配事件给一个HTTP_Handler，后者为此客户机的请求执行所有HTTP协议处理。当新的领导者线程通过反应器等待新的连接到来时，多个处理线程能并发地运行。在处理完它的HTTP GET请求之后，处理线程还原为追随者角色，并等待再次变为领导者线程。

35

领导者/追随者线程池设计是十分有效的[SMFG00]。然而，如果除未处理的性能之外还有其他需求，半同步/半异步设计对于一个Web服务器而言，仍可能是一个更适合的并发模型：

- 因为半同步/半异步设计有一个使用监视器对象模式实现的同步化请求队列，所以它能更灵活地重新排序和优先排序客户机请求。
- 因为半同步/半异步设计在Web服务器虚拟内存而不是操作系统内核[Sch97]内排队请求，所以它可以更具伸缩性。

我们在这里同时谈到这两种线程池方案，以揭示模式的使用如何使这些设计选择成为必然。

1.3.8 有效地利用异步I/O

(1) 语境

要在更有效地支持异步I/O的操作系统平台上实现一个Web服务器，同步多线程还不是最可伸缩的方法。例如，通过调用执行下列活动的异步Win32操作，可以在Windows NT[Sol98]上实现高效率的Web服务器：

- 通过`AcceptEx()`和`ReadFile()`分别处理类似TCP CONNECT和HTTP GET请求这样的指示事件。
- 通过`WriteFile()`或`TransmitFile()`对客户机异步传输所请求的文件[HPS99]。

当这些异步操作完成时，操作系统把与此关联的包含结果的完成事件分发给Web服务器。接着，Web服务器处理这些事件，并在返回事件循环之前执行相应的操作。

(2) 问题

开发具备良好的异步I/O效率和可伸缩性的软件是困难的。难题主要在于异步操作调用和它们随后的完成事件在时间和空间上的分离。

36

(3) 解决方案

运用主动器模式有效地使用异步I/O。这种体系结构的模式构造事件驱动的并发服务器应用程序，以异步地接收和处理从多个客户机来的请求。把应用程序服务分成两个部分：

- 异步执行的操作，例如接受连接和接收客户机的HTTP GET请求。
- 对应的处理异步操作结果的完成处理程序，例如在一个异步连接操作完成之后将文件传回客户机。

如同反应器模式一样，主动器模式从响应事件所执行的服务中分解出服务器应用程序的事件多路分解和事件处理程序分配逻辑。反应器和主动器之间的主要区别在于，主动器处理由异步操作产生的完成事件，而反应器处理触发同步操作的指示事件。

(4) JAWS中的使用

JAWS可以使用主动器模式执行它的协议处理程序，例如当异步地处理其他的连接相关和I/O相关的事件时，分析一个HTTP GET请求的报头。因此，JAWS可以在Windows NT上有效地实现它的事件分配器和协议处理程序组件（如图1-15）。

37

在这种设计中，JAWS通过套接字句柄启动异步Win32操作“主动地”处理服务请求。例如，`AcceptEx()`可以接受从客户机进来的连接请求，同时`TransmitFile()`可以发送文件返回客户机。Windows NT内核异步地执行这些操作。

当一个异步操作完成时，内核把包含那个操作结果的完成事件插入到一个I/O完成端口，此端口把完成事件排队。通过`GetQueuedCompletionStatus()`这个Win32函数从这个端口删除完成事件，该函数由运行JAWS事件循环的主动器调用。主动器多路分解和分配完成事件给与异步操作相关的合适的`HTTP_Acceptor`或者`HTTP_Handler`。

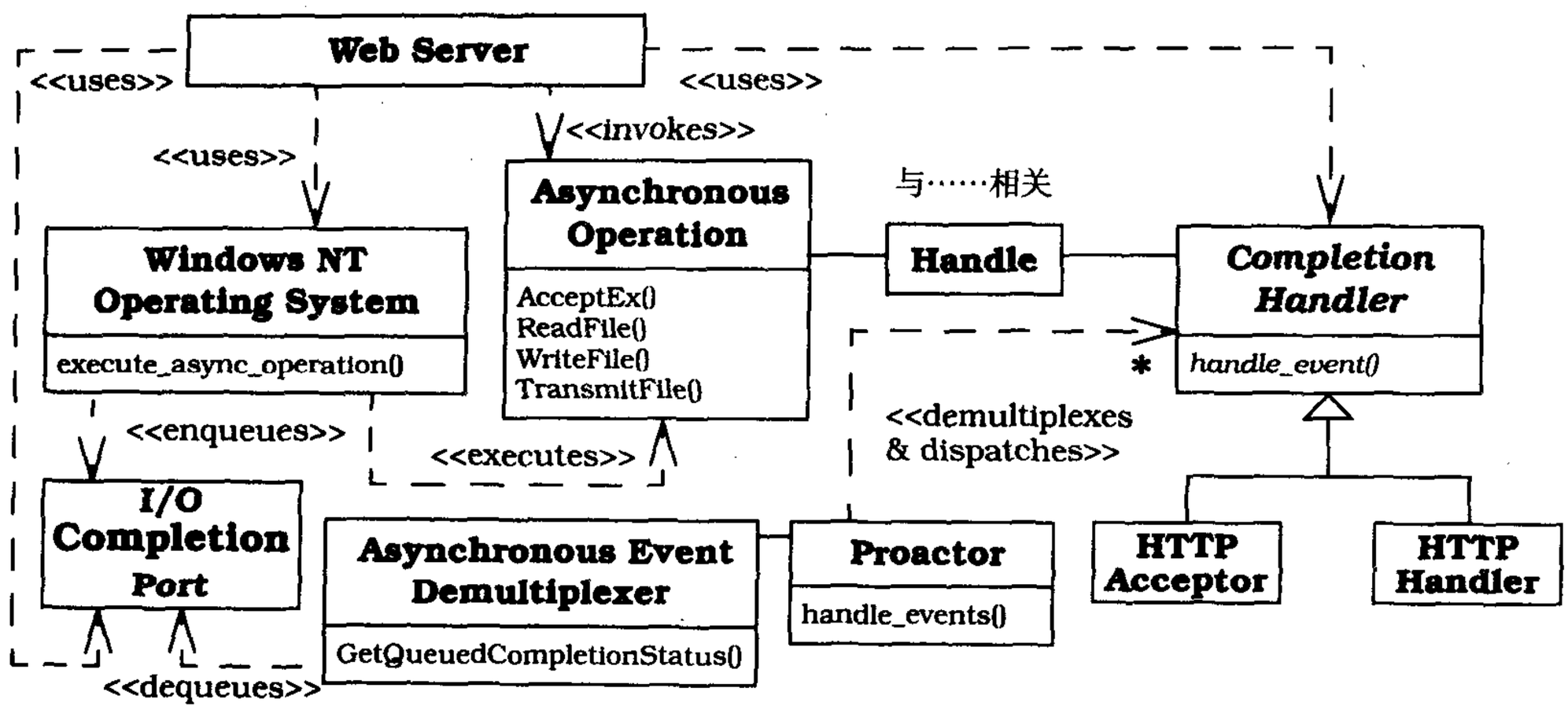


图 1-15

完成处理程序处理异步操作的结果，还有可能调用其他的异步操作。例如，异步AcceptEx()操作的完成处理程序一般会启动一个异步的WriteFile()或者TransmitFile()操作下载一个请求的文件到客户机。

在有效地支持异步I/O的平台上，Web服务器的主动器模式实现通常比半同步/半异步和领导者/追随者模式实现有效得多[HPS99]。然而，主动器模式实现起来比后两个并发体系结构更为复杂：

- 它有更多的参与者，这需要更多工作去理解和实现。
- 主动器中“控制逆转”和异步的结合需要大量的编程和调试经验。

在1.3.7节讨论过，模式的使用使我们可以跳出无关紧要的实现的细节，例如平台的线程语法、多路分解或者连接-管理API，来评估各种各样的Web服务器体系结构的得失。

38

1.3.9 增强服务器的可配置性

(1) 语境

某些Web服务器策略的实现取决于各种因素。有些因素是静态的，例如可用的CPU数和操作系统对异步I/O的支持。其他的因素是动态的，例如Web工作负载特性。

(2) 问题

不存在惟一的Web服务器配置对所有用况都是优化的。另外，一些设计决策直到运行时才能有效地做到。因此，过早地指定一个特殊的Web服务器配置是不灵活的和无效率的。例如，不需要在一个Web服务器中包含未用的协议处理程序或者缓存的虚拟文件系统组件，因为这增加了它的内存覆盖区，而且会降低它的性能。

(3) 解决方案

运用组件配置器模式增强Web服务器的可配置性。这种设计模式允许应用程序在运行时链接和解链它的组件实现。因此，可以在不必修改、重新编译、静态地重新链接或者关闭和重启一

个运行的应用程序的情况下，增加新的或增强的服务。

(4) JAWS中的使用

在安装时或者运行时期间，JAWS使用组件配置器模式动态地优化、控制和重新配置它的Web服务器策略的行为。例如，JAWS运用组件配置器模式配置它的虚拟文件系统的各种缓存策略，例如最近最少用法（LRU）或者最不常用法（LFU），如图1-16所示。

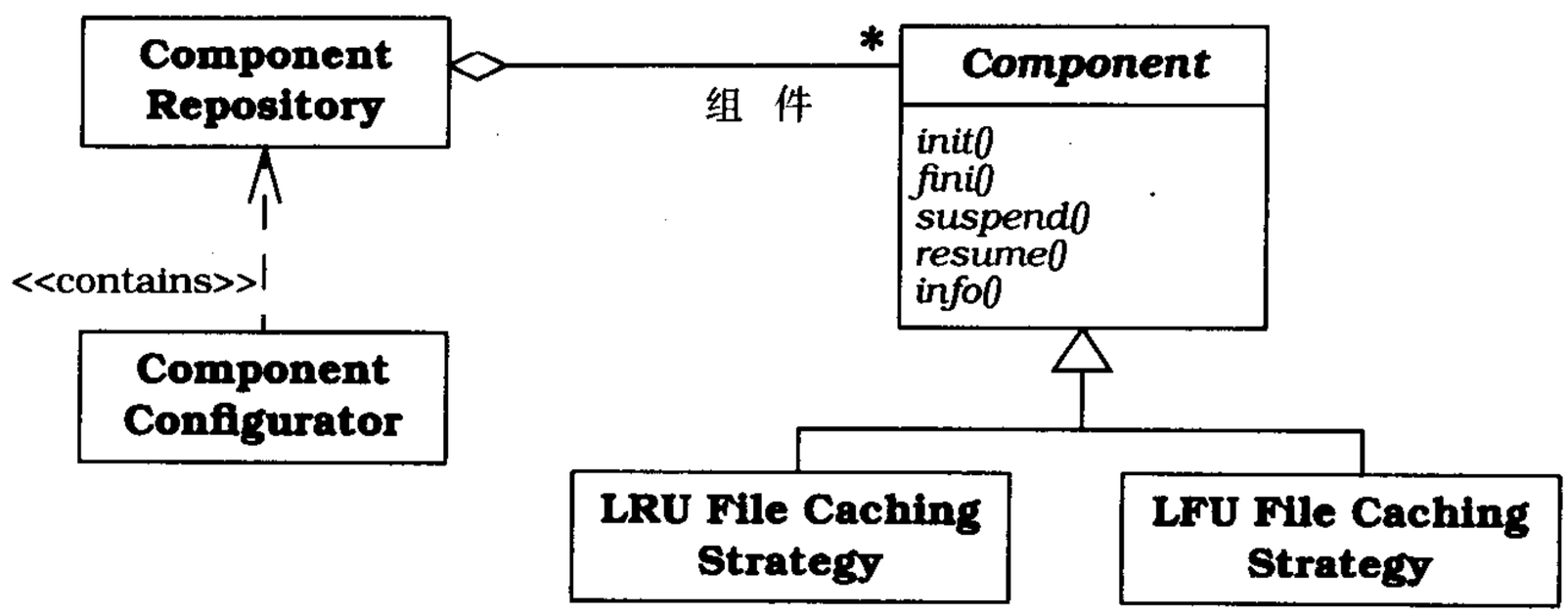


图 1-16

39

组件(component)类定义了一个一致的接口，以配置和控制它所提供的特殊的应用程序服务。具体的组件如LRU_File_Caching_Strategy类和LRU_File_Caching_Strategy类，接着实现这个接口。Web服务器管理员可以使用组件接口，根据预计的或者实际的工作负载，动态地启动、挂起、恢复和终止具体组件。

具体组件能被封装到一个适合的配置单元，例如动态链接库（DLL）。只有当前正使用的组件才需要配置到Web服务器中。这些组件在组件配置器(Component Configurator)的控制下，可以被动态地链接进/解链出应用程序。再下一步，这个对象使用组件仓库(Component Repository)，此组件仓库是一个驻留内存的数据库，它管理配置进Web服务器的所有具体组件。

1.3 10 用于实现JAWS的其他模式

JAWS的实现还用到其他一些设计模式来提高灵活性和模块性。例如，JAWS还用到本书中两个其他的设计模式和一个惯用法。

- 线程安全接口(Thread-Safe Interface)和策略化加锁(Strategized Locking)模式帮助最小化JAWS中名为缓存虚拟文件系统的文件缓存策略中的加锁开销。它们也确保组件内部的方法调用在试图重新获得一个文件缓存已经拥有的锁时，不会导致“自死锁”。
- 定界加锁(Scoped Locking)是整个JAWS上使用的C++惯用法，确保当控制进入一个范围时获得一个锁，当控制离开范围时自动地解锁，而不管退出范围的路径。

在JAWS中也使用[GoF95]中的三个模式：

- 单件(Singleton)模式确保一个类有惟一的实例，并且提供访问它的一个全局点。JAWS使用一个单件以确保在一个Web服务器进程中，只存在它的缓存虚拟文件系统的惟一实例。
- 状态(State)模式定义一个复合的对象，它的行为依赖于它的状态。JAWS中的事件分配器

40

使用状态模式无缝地支持两种不同的并发策略和同步与异步I/O。

- 策略(Strategy)模式定义一个算法簇，封装每一个算法，并使它们可互相交换。JAWS广泛地使用这个模式，例如在不影响协议处理程序软件架构的情况下，用它选择不同的HTTP协议。

[GoF95]中其他的设计模式，例如适配器、桥、工厂方法、迭代器和模板方法，还有[POSA1]中的设计模式，例如代理，在JAWS中都用到了以实现我们上面介绍的8个模式。第2章到第5章的模式说明详细描述了所有这些模式之间的关系。

与1.3.2节描述的四个体系结构的模式（反应器、主动器、半同步/半异步、领导者/追随者）不同，这些设计模式对JAWS有相对局部的影响。例如，尽管策略化加锁模式是域独立的和广泛可用的，但它所解决的问题对JAWS的Web服务器软件架构的影响不像主动器或者领导者/追随者模式那样重大。然而，必须深入理解设计模式，才能实现适应应用需求和平台特性变化的高灵活度的软件。

1.4 小结

在未来的10年里，计算能力和网络带宽将继续以令人瞩目的速度增长。然而，建立在这些硬件进展上的并发和网络化应用软件的需求、规模和复杂性也将以相似的速度增长。如果软件技术没有相应的进展，那么想以合理的时间和工作量管理生命期费用和开发高质量软件将是极其困难的。

41

并发和网络化软件的大量工作和费用，来自于不断重复发现基础模式和具体化这些模式的框架组件。使用模式和模式语言可以减少这种费用，并通过使用已被验证的体系结构和设计来产生应用和应用框架，从而提高软件的质量。可以定制这些框架来满足现有的应用程序需求，也可以扩展这些框架来满足将来的需求。

本章介绍的JAWS例子说明了通过明智地运用模式和框架，使开发并发和网络化软件的工作量大大地减少。与重新发现复杂的解决方案和从零开始构造软件相比，开发人员可以集中精力达到战略性的技术和商业目标。即使因技术或者工具的改变不能直接使用现有的组件、算法、详细设计和实现时，仍然可以重用这些制品的核心体系结构和设计模式。

JAWS例子也阐明组合多种模式解决复杂的并发和网络化应用设计问题的重要性。问题和强制条件常常是彼此相关的，当处理关键设计问题和选择实现方案时，应该考虑到这些关系。因此，不管它们各自的使用如何，没有一个模式是孤立的。相反，模式必须在所处的更大的软件架构语境中去理解。

42

第2章
服务访问和配置模式

有个人参加计算机展览会。每天进场时，
他告诉门口的警卫："我是个江洋大盗，偷东西的技能非常高明。
预先警告你，这次展览会也在劫难逃。"
警卫深受震撼，因为会场有数百万美元的计算机设备，
因此他紧紧地跟住这个人。但这个人只不过从一个展台转到另一个展台，
问一些问题，或者轻声哼唱。
当这个人离开时，警卫把他叫到一边，仔细地搜查他的衣服，
但什么东西也没发现。展览会的第二天，那个人重又到来，并责备警卫：
"我昨天拿了不少，收获颇丰，但今天要带走更多的东西。"
于是警卫盯得更紧了，但仍然一无所获。
展会的最后一天。警卫再也不能抑制住好奇，于是对那个人说：
"小偷先生，我太困惑了，我无法平静地生活，
请你告诉我，你偷的是什么东西呢？"
那个人笑着对警卫说："我在偷模式。"
——摘自《编程之道》
Geoffrey James和Duke Hillard著[JH98]

本章介绍四种模式：包装器外观(Wrapper Facade)、组件配置器(Component Configurator)、截取器(Interceptor)和扩展接口(Extension Interface)，用于设计高效的应用编程接口（API），以访问和配置独立系统和网络化系统中的服务与组件。

网络化系统本质上是异构的[HV99]。因此，研究人员和开发人员所面临的核心难题是如何有效地设计并配置应用程序，以访问演化的服务组件的接口和实现。本章介绍处理服务访问和配置的各个方面的四种模式。

- 包装器外观设计模式把现有的非面向对象的API所提供的函数和数据，封装在更加简洁的、健壮的、可移植的、可维护的和内聚的面向对象的类接口里面。我们常常应用包装器外观"包装"更低层操作系统API来提高应用程序的可移植性。它也能减轻与使用低层API编程有关的偶发的复杂性。

包装器外观模式的"实现"部分包含了用于线程(Thread)、互斥锁(Mutex Lock)、条件变量(Condition Variable)和套接字(Socket)的包装器外观的详细内容，在本书中其他模式的介绍里，相应内容得以简化。后续的模式，例如反应器(Reactor)、主动器(Proactor)、接受器-连接器(Acceptor-Connector)、策略化加锁(Strategized Locking)、主动对象(Active

Object)和监视器对象(Monitor Object)，在它们各自的实现中都使用这些包装器外观。因此，我们推荐你首先阅读包装器外观。

- 组件配置器设计模式允许应用程序在不必修改、重新编译或者静态地重新链接的情况下，在运行时链接和解链它的组件实现。具有高可用性需求的应用程序，例如执行在线事务处理或者实时工业化处理自动化的关键性任务的系统，常常需要这样灵活的配置能力。因此组件配置器处理服务配置和服务演化方面的问题。

本节的其他模式，尤其是扩展接口和截取器，不需要关闭和重新启动正在运行的应用程序进程，就能够使用组件配置器模式，把各种各样的服务角色(重新)配置到应用程序进程的组件中。

44

- 截取器体系结构模式允许透明地把服务加入框架中，并且当事件发生时，能自动地触发服务。因此，截取器为它自己的演化准备了一个框架，以适应在框架的最初开发期间未被配置甚至未知的服务。同时截取器也允许其他的应用程序把组件和服务与框架的实例集成起来。从框架实例的角度上看，这样的服务常常是“带外的”或者是应用特定的，但它对于使用此框架的应用程序的多产和适当的操作而言是重要的。
- 扩展接口设计模式在开发人员扩展或修改现有组件的服务功能时，防止引起接口的“膨胀”和客户机代码的破坏。可以把多个扩展接口放在同一组件中。扩展接口既要处理组件和服务演化的难题，又要处理客户机的供应问题，这些客户机具有与角色有关的对组件功能的访问授权。

服务访问和配置牵涉到的问题超出了本节模式所处理的范围，比如

- 借助于本地代理调节对远程服务的访问。
- 管理服务的生命期，并在分布式系统中定位服务。
- 控制操作系统和服务器所提供的计算资源，这些资源用于驻留在该服务器上的服务实现。

参考文献中的其他模式处理这些要点，例如激励器(Activator)[Sta100]、逐出者(Evictor)[HV99]、半对象加协议(Half Object plus Protocol)[Mes95]、定位器(Locator)[JK00]、对象生存期管理器(Object Lifetime Manager)[LGS99]和代理(Proxy)[POSA1][GoF95]。这些模式与本节介绍的模式相辅相成，共同描述了一些关键准则，而结构良好的分布式系统根据这些准则配置所提供的服务，并提供对服务的访问。

45

2.1 包装器外观

包装器外观（Wrapper Facade）设计模式把现有的非面向对象的API所提供的函数和数据，封装在更加简洁的、健壮的、可移植的、可维护的和内聚的面向对象的类接口中。

1. 例子

考虑一个用于分布式登录服务的服务器，它使用面向连接的TCP协议[Ste98]并发地处理多个客户机。为了登录数据，客户机必须发送一个连接请求给服务器的传输地址，此地址由TCP端口

号和IP地址组成。在登录服务器中，一个被动模式的套接字句柄工厂在这个地址上监听连接请求。套接字句柄工厂接受连接请求，并创建一个数据模式的套接字句柄，标识这个客户机的传输地址。这个句柄被传送给服务器应用程序，后者产生一个登录处理程序（logging handler）线程，处理客户机登录请求。

客户机在连接之后，发送登录请求给服务器。登录处理程序线程通过已连接的套接字句柄接收这些请求，然后在线程中处理请求，并将其写入日志文件（如图2-1）。

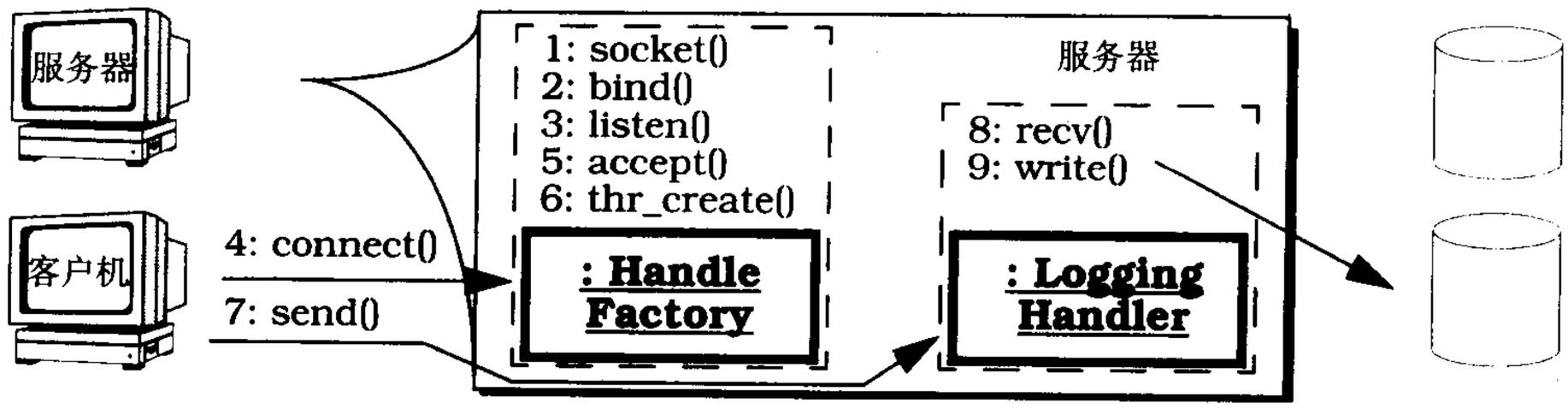


图 2-1

开发这个登录服务器，一般使用低层C语言API（如Solaris线程[EKBF+92]和套接字[Ste98]）编制服务器的线程、同步和网络通信功能。然而，如果登录服务器在多个平台上运行，那么低层API的函数和数据之间会有所不同，操作系统和编译器特性和缺陷的不一致也会引起差异。开发人员通常通过插入条件编译伪指令如C/C++ `#ifdef`来处理这些不同。例如，下页代码使用`#ifdef`实现在Solaris和Windows NT上运行的登录服务器。

47

代码显示的设计也许用于短暂的“一次性”原型[FoYo99]。然而这对于必须长期维护和增强的软件而言是远远不够的。使用条件编译伪指令以及用低层API直接编程，使应用程序代码难以理解、调试、移植、维护和演化。

如果把与平台有关的声明移入分离的配置头文件，就可以减轻一些问题，例如互斥和套接字类型。然而这种解决方案并不完全，因为用于分离针对平台的API（如线程创建调用）的用法的# ifdef，仍将损害应用程序代码。对新平台的支持也将需要修改与平台有关的声明，而不管它们是直接包含到应用程序代码中还是已分离到配置文件里。

若干有名的模式提及类似问题，但都不能帮助解决上述的问题。例如，外观(facade)[GoF95]封装面向对象的子系统，而不是封装更低层的非面向对象的API。装饰器(Decorator)[GoF95]通过透明地附加额外的责任动态地扩展对象，从而导致不必要的性能开销。桥(Bridge)和适配器(Adapter)[GoF95]也引入额外的间接层，这也导致开销。因此总的说来，这些模式不太适合封装现有的更低层的非面向对象API，它们更着重于解决方案的有效性而不是动态可扩充性。

2. 语境

可维护和可演化的应用程序，它们访问现有的非面向对象API提供的机制和服务。

```
#if defined (_WIN32)
   #include <windows.h>
   typedef int ssize_t;
#else
   typedef unsigned int UINT32;
   #include <thread.h>
   #include <unistd.h>
   #include <sys/socket.h>
   #include <netinet/in.h>
   #include <memory.h>
#endif /* _WIN32 */

// Keep track of number of logging requests.
static int request_count;

// Lock that serializes concurrent access to request_count.
#if defined (_WIN32)
   static CRITICAL_SECTION lock;
#else
   static mutex_t lock;
#endif /* _WIN32 */

// Maximum size of a logging record.
static const int LOG_RECORD_MAX = 1024;

// Port number to listen on for requests.
static const int LOGGING_PORT = 10000;

// Entry point that writes logging records.
int write_record (char log_record[], int len) {
   /* ... */
   return 0;
}

// Entry point that processes logging records for
// one client connection.
#if defined (_WIN32)
   u_long
#else
   void *
#endif /* _WIN32 */
logging_handler (void *arg) {
   // Handle UNIX/Win32 portability.
#if defined (_WIN32)
   SOCKET h = reinterpret_cast <SOCKET> (arg);
#else
   int h = reinterpret_cast <int> (arg);
#endif /* _WIN32 */

   for (;;) {
      // Ensure a 32-bit quantity.
#if defined (_WIN32)
      ULONG len;
#else
      UINT32 len;
#endif /* _WIN32 */
      char log_record[LOG_RECORD_MAX];
      // The first <recv> reads the length
      // (stored as a 32-bit integer) of
      // adjacent logging record. This code
      // does not handle "short-<recv>s".
      ssize_t n = recv (h,
               reinterpret_cast <char *> (&len),
               sizeof len, 0);
      // Bail out if we're shutdown or
      // errors occur unexpectedly.
      if (n <= sizeof len) break;
      len = ntohl (len);
      if (len > LOG_RECORD_MAX) break;

      // The second <recv> then reads <len>
      // bytes to obtain the actual record.
      // This code handles "short-<recv>s".
      for (size_t nread = 0; nread < len; nread += n) {
         n = recv (h, log_record + nread,
                  len - nread, 0);
         // Bail out if an error occurs.
         if (n <= 0) return 0;
      }
#if defined (_WIN32)
      EnterCriticalSection (&lock);
#else
      mutex_lock (&lock);
#endif /* _WIN32 */
      // Execute following two statements in a critical
      // section to avoid race conditions and scrambled
      // output, respectively.
      ++request_count;
      // A return value of -1 signifies failure.
      if (write_record (log_record, len) == -1)
         break;
#if defined (_WIN32)
      LeaveCriticalSection (&lock);
#else
      mutex_unlock (&lock);
#endif /* _WIN32 */
   }
#if defined (_WIN32)
   closesocket (h);
#else
   close (h);
#endif /* _WIN32 */
   return 0;
}

// Main driver function for the server.
int main (int argc, char *argv[]) {
   struct sockaddr_in sock_addr;
   // Handle UNIX/Win32 portability.
#if defined (_WIN32)
   SOCKET acceptor;
   WORD version_requested = MAKEWORD(2, 0);
   WSADATA wsa_data;
   int error = WSAStartup(version_requested, &wsa_data);
   if (error != 0) return -1;
#else
   int acceptor;
#endif /* _WIN32 */
   // Create a local endpoint of communication.
   acceptor = socket (PF_INET, SOCK_STREAM, 0);
   // Set up the address to become a server.
   memset (&sock_addr, 0, sizeof sock_addr);
   sock_addr.sin_family = PF_INET;
   sock_addr.sin_port = htons (LOGGING_PORT);
   sock_addr.sin_addr.s_addr = htonl (INADDR_ANY);
   // Associate address with endpoint.
   bind (acceptor, reinterpret_cast<struct sockaddr *>
         (&sock_addr), sizeof sock_addr);
   // Make endpoint listen for connections.
   listen (acceptor, 5);

   // Main server event loop.
   for (;;) {
      // Handle UNIX/Win32 portability.
#if defined (_WIN32)
      SOCKET h;
      DWORD t_id;
#else
      int h;
      thread_t t_id;
#endif /* _WIN32 */
      // Block waiting for clients to connect.
      h = accept (acceptor, 0, 0);
      // Spawn a new thread that runs the <server>
      // entry point.
#if defined (_WIN32)
      CreateThread (0, 0,
            LPTHREAD_START_ROUTINE(&logging_handler),
            reinterpret_cast <void *> (h), 0, &t_id);
#else
      thr_create
         (0, 0, logging_handler,
            reinterpret_cast <void *> (h),
            THR_DETACHED, &t_id);
#endif /* _WIN32 */
   }
   return 0;
}
```

48

3. 问题

常常使用非面向对象的操作系统API或者系统库编写应用程序。这些API访问网络和线程编程机制，也访问用户接口或数据库编程库。尽管这种设计是普通的，但它由于没有解决下列强制条件而给应用程序开发人员造成问题：

- 简洁的代码常常比冗长的代码更健壮，因为它更易于开发和维护。使用支持高级特性的面向对象语言，例如构造函数、析构函数、异常、无用单元收集等，可以减少普通编程错误。直接使用低层的、基于函数的API编程的开发人员容易重复地编写大量冗余的、易出错的软件。

49

➡在我们的登录服务器例子的main()函数中，创建和初始化接受器套接字的代码容易出错，而且，这些错误错综复杂。例如，未能初始化sock_addr为零，或者不使用htons()宏转换LOGGING_PORT数为网络字节顺序[Sch92]。在C中由于缺少构造函数和析构函数，也不能确保正确地分配和释放资源。例如，如果write_record()函数返回-1，那么就不能正确地释放串行化访问request_count的锁。 ❑

- 软件产品如果可移植或者很容易移植到不同的操作系统、编译器和硬件平台，容易取得较大的市场份额。尽管复用现有的低层API可以减少某些开发软件的工作，但直接用低层API编写的应用程序常常是不可移植的。由于软件版本间缺乏源代码级或者二进制级的兼容性，即使在同一操作系统或者编译器的不同版本中使用低层API编程也可能是不可移植的[Bcx97]。

  ➡我们的登录服务器例子在几个不可移植的操作系统线程和网络编程C API中具有硬编码依赖性。例如，Solaris的thr_create()、mutex_lock()和mutex_unlock()函数对Win32平台而言是不可移植的。尽管代码是准移植的——它也能在Win32平台上编译和运行——但仍存在种种错综复杂的移植性问题。尤其是，在Win32平台上将存在资源泄露，因为它不具有Solaris的THR_DETACHED特性的等价物，在Solaris中这种特性产生一个“分离的”线程，它的退出状态并不被线程库所保留[Lew95]。 ❑

- 改进软件可维护性可以减少软件生命期的费用。不过，直接使用低层的非面向对象API编写的程序常常难以维护。例如，C和C++开发人员常常把条件编译伪指令嵌入到应用程序源代码中，试图解决移植性问题。可是，如果在所有的点上都运用条件编译来处理与平台有关的变化，就增加了软件的实际设计复杂性[Lak95]。例如，与平台有关的细节分散到应用源文件的所有地方。 50

  ➡使用#ifdef处理Win32和Solaris间的移植性（例如Win32和Solaris的套接字类型的区别阻碍了我们对登录服务器的维护。通常，编写像这样低层C API的开发人员必须熟悉许多操作系统特质的知识，以维护和演化他们编写的代码。 ❑

- 内聚的组件更容易学习、维护和增强。然而，低层API很少分组为内聚的组件，因为类似C这样的语言缺乏诸如类、名字空间或者包等特性，从而很难确认低层API的范围。使用非内聚的独立函数API编程，也使常用代码分散在整个应用程序中，使“插入”支持不同策略和机制的新组件变得困难重重。

  ➡套接字API尤其难以学习，因为在套接字库中几十个C函数缺乏一致的命名约定。例如，socket()、bind()、listen()、connect()和accept()相关性并不明显。其他的低层网络编程API（例如TLI）通过预先加一个常用函数前缀来处理这个问题，例如在TLI的API每个函数前加一个t_前缀。然而，常用前缀的使用并不能使TLI的API比套接字更容易“插入”。它保留一个低层的基于函数的API，而不是一个更加内聚的面向对象的类接口。 ❑

通过直接编写非面向对象API开发应用程序，对于必须长期维护和演化的软件而言，是一种拙劣的设计选择。

4. 解决方案

避免直接访问非面向对象的API。对于非面向对象的API中的每组相关函数和数据，为它们

创建一个或多个包装器外观类，把这些函数和数据封装在面向对象的包装器外观所提供的更简洁的、健壮的、可移植的和可维护的方法中。

51

5. 结构

包装器外观模式中存在两个参与者：

函数是现有的非面向对象的API的构件。它们提供一种独立的服务或者机制，并管理作为参数传递或者通过全局变量访问的数据。

包装器外观是一个或多个面向对象的类的集合，这些类封装现有的函数和与它们相关联的数据。这些类导出一个内聚的抽象，由它提供特定类型的功能。每个类代表抽象中一个特定的角色。

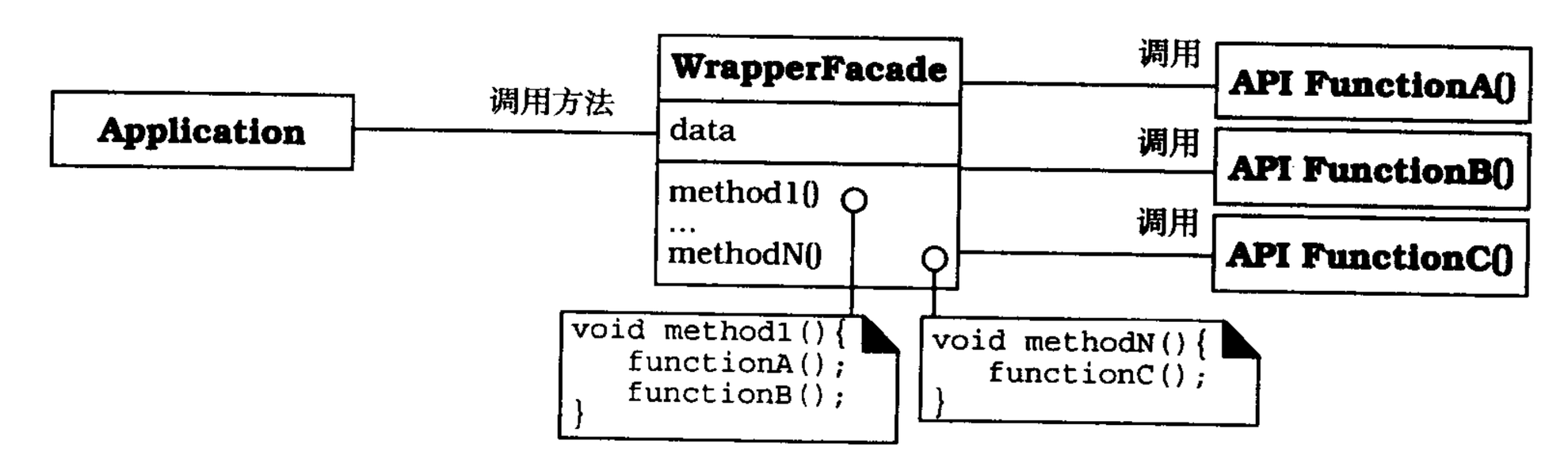
包装器外观类中的方法通常把应用程序调用转发给函数，并把数据作为参数传递。数据通常隐藏在包装器外观的私有部分，对客户机应用程序而言是不可访问的。接着编译器能强制实施类型安全，因为原始的数据类型（例如指针或者整数）是被封装在强类型的包装器外观内，如图2-2所示。

类 包装器外观	协作者 • API函数
责任 • 用一个内聚的面向对象的抽象封装非面向对象的函数和数据结构	

类 API函数	协作者
责任 • 通过一个定义良好的API提供一种服务	

图 2-2

图2-3的类框图阐明包装器外观的结构。



52

图 2-3

6. 动态特性

包装器外观模式中的协作通常是直接的（如图2-4）：

• 应用程序代码在包装器外观的实例上调用方法。

- 包装器外观方法把请求和请求的参数转发给它封装的低层的API函数，并传递底层函数所需要的任何内部数据。

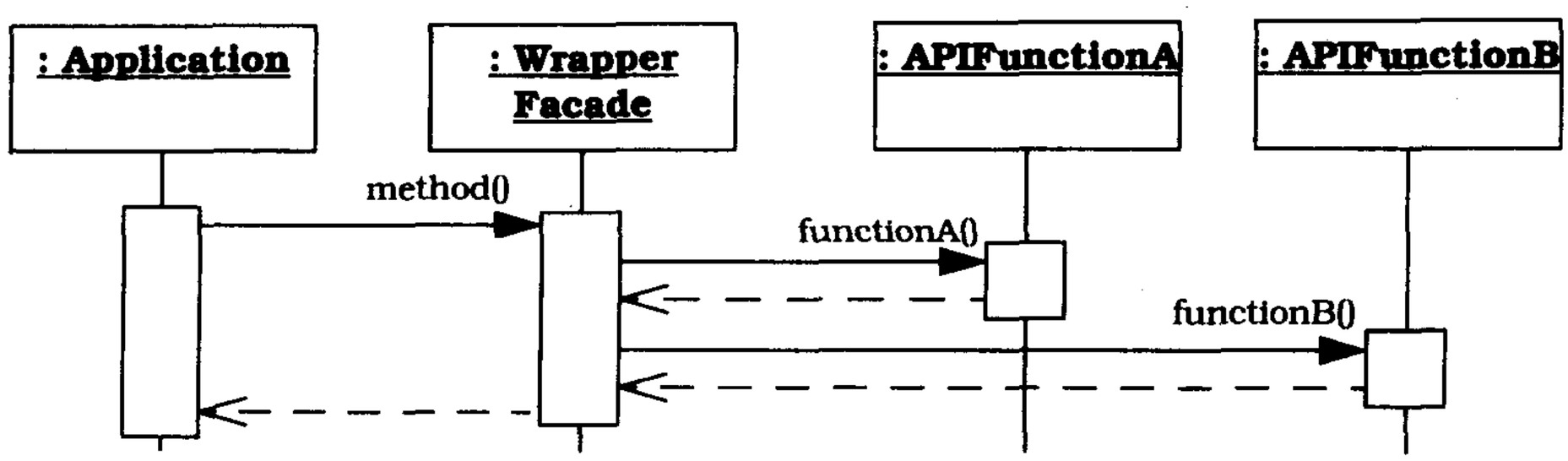


图 2-4

7. 实现

本节描述实现包装器外观模式所涉及的活动。有些活动也许需要多个迭代以标识和实现合适的包装器外观抽象。为减少本书中其他地方的重复，本节我们详细讨论用于互斥、条件变量、套接字和线程的具体包装器外观。尽管这会稍微增加本节的内容，但本书中其他模式的实现例子，包括接受器-连接器、策略化加锁、线程特定的存储器和监视器对象等，可以通过使用这些包装器外观而得以简化。

(1) *标识内聚的抽象和现有的低层API之间的关系。*成熟的低层API包含定义许多内聚的抽象的函数和数据结构，并且清楚地映射到面向对象的类和方法中。普通的例子包括用于Win32同步和线程、POSIX网络编程、X Windows GUI事件分配的C语言API。然而，由于在类似C这样的语言中缺乏数据抽象，所以在这些现有的API中并不清楚函数如何彼此相关。因此实现包装器外观模式的第一个活动是标识内聚的抽象和现有的API之间的关系。 53

➥我们的登录服务器的原始实现使用许多低层函数，它们提供多个内聚的操作系统机制，如同步和网络通信。例如，Solaris的`mutex_lock()`和`mutex_unlock()`函数和一个互斥同步抽象相关联。类似地，`socket()`、`bind()`、`listen()`和`accept()`函数在网络编程抽象中扮演多种角色。 ❑

如果现有的函数和数据结构已经被开发成即用即弃的代码，或者是以渐进的方式开发的[FoYo99]，它们也许会有一点或者完全没有内聚抽象。这时如果可能的话，在继续进行包装器外观模式的实现之前，应该重新分解代码[Opd92][FBBOR99]。

(2) *把内聚的函数组聚合进包装器外观类和方法。*这个活动定义一个或者多个类抽象，把应用程序与低层数据表示、函数语法中的任意变量以及其他的实现细节隔开。可以把它分解为五个子活动。

(2.1) *创建内聚的类。*首先，我们为每组现有的非面向对象的API定义包装器外观，这些API与一个特殊的抽象相关。用于创建内聚的类的常用标准包括下列方面：

- 把具有高度内聚性的函数和数据合并到单个类中，同时也最小化类之间不必要的耦合。内聚函数是那些管理常用数据结构的函数，例如套接字、文件或者信号集[Ste98]。

- 标识出函数和数据底层中的常用和可变方面[Cope98]。常用方面包括用于同步、线程、内存管理、寻址和操作系统平台API的机制；可变方面通常包括这些机制的实现。无论何时，函数和数据中可能的变化都应该被分解到类，由这些类在统一接口之下隔离变化，从而增强可扩展性。

通常，如果原始的API包含大量的相关函数，那么有必要创建多个包装器外观类以正确地分离业务。

54

(2.2) 把多个函数合并到一种方法之中。除把现有的函数分组到类之外，把多个单独的函数合并到每个包装器外观类中更少的方法中也是有用的。如此合并可以确保以恰当的顺序调用一组更低层的函数，如同模板方法（Template Method）模式[GoF95]一样。

(2.3) 如果可能的话，自动化创建和销毁操作。更低层的API通常需要编程人员人为地调用函数，以创建和销毁实现API实例的数据结构。然而，这个过程是易出错的，因为开发人员可能在代码的路径上忘记调用这些函数。因此一个更稳健的方法是使用由C++和Java这样的面向对象的语言提供的隐式创建和销毁操作的能力。事实上，正是自动地创建和销毁对象的能力，说明使用包装器外观模式是必要的，即使只是使用包装器外观方法把控制转发给更低层函数调用。

(2.4) 选择间接的级别。上面提到，大多数包装器外观类只是把它们的方法调用转发给低层函数。如果包装器外观方法能被隐式地或者显式地内联，那么比起直接调用低层函数来，它不存在运行时的间接开销。通过动态地使用约束方法或者某种其他形式的多态分配包装器外观实现，以增加另一间接级别也是可能的。在这种情况下，包装器外观类在桥模式[GoF95]中扮演抽象类的角色。

(2.5) 确定在哪里封装任何与平台有关的变化。包装器外观模式的一般使用是在应用程序代码中最小化与平台有关的变化。尽管包装器外观方法实现在不同的操作系统平台也许存在不同，但它们应提供一致的、与平台无关的接口。在与平台有关的变化存在的地方，可以通过条件编译或者分离目录来封装它。

55

- 条件编译　可用于在不同的包装器外观类方法实现之间作出选择。当#ifdef分散在整个应用程序代码中时，可以说条件编译的使用是粗糙的和冗长的。然而当条件编译只分布在少数与平台有关的包装器外观类或文件中，而且不被应用程序开发人员直接访问时，这是可以接受的。当条件编译和自动配置工具（例如GNU的autoconf）联合使用时，可以在一个源文件中创建与平台无关的包装器外观。因此，只要文件中所支持的变体数不大，条件编译就可以把变体局域化从而简化维护。
- 分离目录　常用于排除出不相关的包装器外观实现，从而尽量或完全避免条件编译。例如，每个操作系统平台可以有它自己的目录，包含与平台有关的包装器外观的实现。可使用语言处理工具在编译时从相关的目录中包含合适的包装器外观类。为了获得一个不同的实现，可以给编译器提供不同的包含路径。这种策略避免了上述的与条件编译有关的问题，因为它物理上把各种实现分解到分离的目录。

策略的选择依赖于包装器外观接口和实现改变的频繁程度。如果改变发生频繁，那么为每个平台更新条件编译部分也许是耗时的。类似地，即使只需对一个平台进行改变，但依赖于受

影响文件的所有文件都将被重新编译。因此，随着所支持的平台数目变大，条件编译的使用变得更加复杂。然而，无论选择何种策略，维护包装器外观实现的负担应是包装器外观开发人员的责任，而不应该是应用程序开发人员的责任。

➡为了简化我们的登录服务器实现，我们为互斥、套接字和线程定义包装器外观封装现有的低层C API。每个包装器外观阐明如何系统地处理上述的各种设计要点。我们重点为C函数定义包装器外观，因为诸如POSIX或Win32这样的流行操作系统API都是用C写的。当然，同样的设计原则和技术可以应用于其他非面向对象的语言，例如FORTRAN、Ada 83、Scheme或者Pascal，同时对于像X Windows或ODBC数据库工具这样的非操作系统API，也同样有效[San 98]。 ❑ 56

互斥包装器外观(mutex wrapper facade)。我们首先定义一个Thread_Mutex抽象，它用一致的和可移植的类接口[⊖]封装Solaris的互斥函数。

```
class Thread_Mutex {
public:
    Thread_Mutex ()
        { mutex_init (&mutex_, USYNC_THREAD, 0); }
    ~Thread_Mutex () { mutex_destroy (&mutex_); }

    void acquire () { mutex_lock (&mutex_); }
    void release () { mutex_unlock (&mutex_); }
private:
    // Solaris-specific Mutex mechanism.
    mutex_t mutex_;

    // Disallow copying and assignment.
    Thread_Mutex (const Thread_Mutex &);
    void operator= (const Thread_Mutex &);

    // Define a <Thread_Condition> as a friend so it can
    // access <mutex_>.
    friend class Thread_Condition;
};
```

注意我们如何把复制构造函数和赋值运算符定义为Thread_Mutex类中的私有方法。这个C++惯用法确保应用程序编程人员不会意外地复制或者把一个Thread_Mutex赋值给另一个[Mey98][Str97]。复制互斥实际上是一种语义无效的操作，此操作错误地被缺乏强类型的C编程API所允许。因此，我们的Thread_Mutex包装器外观提供一个互斥接口，与直接编程到更低层的Solaris同步函数相比，此接口不易出错。

通过定义一个Thread_Mutex类接口，然后编写应用程序使用它，而不是使用更低层的本地操作系统C API，可以更加容易将我们的包装器外观移植到其他平台。例如，同样的Thread_Mutex接口可以如下实现，在Win32上运行： 57

```
class Thread_Mutex {
public:
    Thread_Mutex ()
        { InitializeCriticalSection (&mutex_); }
    ~Thread_Mutex ()
```

⊖ 为了节省篇幅并集中于本质上的设计要点，本书中许多方法实现并不进行错误检查，它们也不从函数中用非void返回类型返回值或者扔出异常。很自然，软件产品一直应该检查错误，并一贯地且正确地传播错误。

```
        { DeleteCriticalSection (&mutex_); }

    void acquire () { EnterCriticalSection (&mutex_); }
    void release () { LeaveCriticalSection (&mutex_); }
private:
    // Win32-specific Mutex mechanism.
    CRITICAL_SECTION mutex_;

    // Disallow copying and assignment.
    Thread_Mutex (const Thread_Mutex &);
    void operator= (const Thread_Mutex &);
};
```

自然地，完整的Thread_Mutex实现，会把与平台有关的错误处理返回值从各种mutex_t和CRITICAL_SECTION函数的，映射给可移植的C++异常。

前面描述过，我们可以使用条件编译以及在Thread_Mutex方法实现中加入# ifdef，来同时支持多个操作系统。如果由于所支持的平台数太多使条件编译显得笨拙的话，那么还可以把不同的Thread_Mutex实现分解到分离目录。在这种情况下，也可以使用诸如编译器和预处理程序这样的语言处理工具，在编译期间把恰当的与平台有关的变化包含到应用程序中。

条件变量包装器外观(condition variable wrapper facade)。条件变量是一种同步机制，协作线程使用这种机制暂时挂起它们自己，直到包含在线程间共享的数据的条件表达式获得期望的状态[IEEE96]。这时，我们为条件变量描述包装器外观，因为它们常常和上面描述的Thread_Mutex包装器外观一起使用。尽管我们的登录服务器例子并不使用条件变量，但它被书中的其他模式所使用，例如策略化加锁、领导者/追随者和监视器对象。

58

上面提到，条件变量一直和互斥一起使用，客户机线程在估算条件表达式之前必须获取此互斥。如果条件表达式是false，那么客户机在条件变量上挂起自己，并把互斥作为整体释放，这样其他的线程便能改变共享的数据。当一个协作的线程改变这个数据时，它可以通知条件变量，作为整体恢复先前在条件变量上挂起自己的线程。接着线程重新获取与条件变量相关联的互斥。

在重新获取它的互斥之后，新恢复的线程再一次重新估算它的条件表达式。如果共享的数据已经获得期望的状态，那么线程继续进行。否则它再次在条件变量上挂起自己，直到再被恢复为止。重复这个过程，直到条件表达式变为true。

通常，当需要复杂的条件表达式或者调度行为时，结合使用条件变量和互斥比仅仅使用互斥要更适合些。正如在监视器对象模式例子中所展示的，条件变量能用于实现同步化的消息队列。在这种情况下，当消息队列满时，使用一对条件变量协作地阻塞供应者线程，同时当队列空时阻塞消费者线程。

下面的Thread_Condition类是一个使用Solaris条件变量API实现的包装器外观：

```
class Thread_Condition {
public:
    // Initialize the condition variable and
    // associate it with the <mutex_>.
    Thread_Condition (const Thread_Mutex &m) : mutex_ (m)
        { cond_init (&cond_, USYNC_THREAD, 0); }

    // Destroy the condition variable.
```

```
    ~Thread_Condition () { cond_destroy (&cond_); }

    // Wait for the <Thread_Condition> to be notified
    // or until <timeout> has elapsed. If <timeout> == 0
    // then wait indefinitely.
    void wait (Time_Value *timeout = 0) {
        cond_timedwait (&cond_, &mutex_.mutex_,
                        timeout == 0
                            ? 0 : timeout->msec ());
    }

    // Notify one thread waiting on <Thread_Condition>.
    void notify () { cond_signal (&cond_); }

    // Notify all threads waiting on <Thread_Condition>.
    void notify_all () { cond_broadcast (&cond_); }
private:
    // Solaris condition variable.
    cond_t cond_;

    // Reference to mutex lock.
    const Thread_Mutex &mutex_;
};
```

构造函数初始化条件变量，并把它和作为参数传递的Thread_Mutex关联起来。析构函数销毁条件变量，释放已分配的资源。注意，互斥不属于Thread_Condition，因此在析构函数中并不销毁它。

当被客户机线程调用时，wait()方法不可分解地执行下列两个步骤：

- 释放被关联的互斥；
- 整体自我挂起至最高超时限制，等待Thread_Condition对象被另一个线程通知。

notify()方法恢复在Thread_Condition上等待的线程。类似地，notify_all()方法通知正在Thread_Condition上等待的所有线程。在mutex_锁返回给它的客户机线程之前，因为条件变量得到通知，或者因为它的超时到期，wait()方法重新获取它。

套接字包装器外观（socket wrapper facade）。接下来的包装器外观封装套接字API。这个API比Solaris互斥API更大和更具表现力[Sch92]。因此必须定义一组相关的包装器外观类来封装套接字。我们首先定义typedef和一个隐藏某些UNIX/POSIX和Win32移植性差异的宏。

```
typedef int SOCKET;
const int INVALID_HANDLE_VALUE = -1;
```

SOCKET和INVALID_HANDLE_VALUE都已在Win32 API中定义。因此，如同在(2.5)中所讨论的一样，可以使用#ifdef，或者使用分离的与平台有关的目录来集成它们。

接着，定义一个INET_Addr类来封装互联网域地址struct：

```
class INET_Addr {
public:
    INET_Addr (u_short port, u_long addr) {
        // Set up the address to become a server.
        memset (&addr_, 0, sizeof addr_);
        addr_.sin_family = PF_INET;
        addr_.sin_port = htons (port);
        addr_.sin_addr.s_addr = htonl (addr);
    }
```

```
    u_short get_port () const
        { return ntohs (addr_.sin_port); }

    u_long get_ip_addr () const
        { return ntohl (addr_.sin_addr.s_addr); }

    sockaddr *addr () const
        { return reinterpret_cast <sockaddr *> (&addr_);}

    size_t size () const { return sizeof (addr_); }

    // ...
private:
    sockaddr_in addr_;
};
```

注意INET_Addr构造函数是如何消除多个常用套接字编程错误的。例如，它初始化sockaddr_in域为零，同时自动地使用htons()和htonl()宏确保TCP端口号和IP地址转换为网络字节顺序[Ste98]。

接下来的包装器外观类SOCK_Stream封装I/O操作，例如recv()和send()，应用程序可以在连接的套接字句柄上调用这样的I/O操作：

```
class SOCK_Stream {
public:
    // Default and copy constructor.
    SOCK_Stream () : handle_ (INVALID_HANDLE_VALUE) { }
    SOCK_Stream (SOCKET h): handle_ (h) { }

    // Automatically close the handle on destruction.
    ~SOCK_Stream () { close (handle_); }

    // Set/get the underlying SOCKET handle.
    void set_handle (SOCKET h) { handle_ = h; }
    SOCKET get_handle () const { return handle_; }

    // Regular I/O operations.
    ssize_t recv (void *buf, size_t len, int flags);
    ssize_t send (const char *buf, size_t len, int flags);

    // I/O operations for "short" receives and sends.
    ssize_t recv_n (char *buf, size_t len, int flags);
    ssize_t send_n (const char *buf, size_t len,
                    int flags);

    // ... other methods omitted.
private:
    // Socket handle for exchanging socket data.
    SOCKET handle_;
};
```

61

在(2.3)中讨论过，这个类均衡C++析构函数的语义，以确保当SOCK_Stream对象超出范围时，自动地关闭套接字句柄。另外，send_n()和recv_n()方法可以处理网络特质，例如“短”发送和接收操作。

可以通过名为SOCK_Acceptor的连接工厂创建SOCK_Stream对象，这样做可以封装套接字连接的被动建立。SOCK_Acceptor构造函数初始化被动模式的接受器套接字去监听

sock_addr地址。SOCK_Acceptor的accept()方法是一个工厂，它用一个新连接的套接字句柄初始化SOCK_Stream参数。

```
class SOCK_Acceptor {
public:
    // Initialize a passive-mode acceptor socket.
    SOCK_Acceptor (const INET_Addr &addr) {
        // Create a local endpoint of communication.
        handle_ = socket (PF_INET, SOCK_STREAM, 0);
        // Associate address with endpoint.
        bind (handle_, addr.addr (), addr.size ());
        // Make endpoint listen for connections.
        listen (handle_, 5);
    };

    // A second method to initialize a passive-mode
    // acceptor socket, analogously to the constructor.
    void open (const INET_Addr &sock_addr) { /* ... */ };
    // Accept a connection and initialize the <stream>.
    void accept (SOCK_Stream &s) {
        s.set_handle (accept (handle_, 0, 0));
    }
private:
    SOCKET handle_; // Socket handle factory.
};
```

62

注意用于SOCK_Acceptor的构造函数是如何应用实现活动(2.2)中讨论的策略的，这样确保低层socket()、bind()和listen()函数一直被一起以正确的顺序调用。

一个完整的套接字（Socket）[Sch97]包装器外观集也会包含一个SOCK_Connector，它主动地封装用于建立连接的逻辑。SOCK_Acceptor和SOCK_Connector类是具体的IPC机制，可以使用此机制实例化在接受器-连接器模式中描述的接受器类和连接器类，以执行连接的建立。

线程包装器外观(thread wrapper facade)。我们最后的包装器外观封装操作系统线程API，这些API不同的操作系统平台都提供，包括Solaris线程、POSIX Pthreads和Win32线程。这些API具有错综复杂的句法和语义上的差别。例如，能以“分离的”模式产生Solaris和POSIX线程，而Win32线程却不能。然而，提供一个Thread_Manager包装器外观，以一致的方式封装这些差别，却是可行的。下面的Thread_Manager包装器外观是一个单件（Singleton）[GoF95]，它说明了Solaris线程实现的产生方法：

```
class Thread_Manager {
public:
    // Singleton access point.
    Thread_Manager *instance ();

    // Spawn a thread.
    void spawn (void *(*entry_point_function) (void *),
                void *arg = 0, long flags = 0,
                long stack_size = 0,
                void *stack_pointer = 0,
                thread_t *t_id = 0) {
        thread_t t;
        if (t_id == 0)
            t_id = &t;
        thr_create
            (stack_size, stack_pointer,
```

63

```
                entry_point_function, arg, flags, t_id);
    }

    // ... Other methods omitted.
};
```

Thread_Manager类也提供能够移植到其他操作系统的加入和取消线程的方法。

(3) *考虑允许应用程序受控访问实现细节*。定义包装器外观的一个优点是不太容易编写不正确的或不可移植的应用程序。例如，包装器外观能使应用程序避免易出错的、与平台有关的实现细节，如一个套接句是表示为指针还是整数。然而，额外的抽象和类型安全实际上阻止编程人员以设计者未曾预先料到的有用方法使用包装器外观，这时会出现一些情况。这种经历令人沮丧，它会使编程人员不能充分利用包装器外观的其他优点。

解决“太多”抽象问题的常用方法是提供“安全舱口”(escape hatch)机制或者“开放的”实现技术，例如AOP[KLM+97]。这种设计允许应用程序以某种受控的方式访问实现细节。

➡SOCK_Stream类定义一对方法来设置和获取底层的SOCKET句柄。

```
class SOCK_Stream {
public:
    // Set/get the underlying SOCKET handle.
    void set_handle (SOCKET h) { handle_ = h; }
    SOCKET get_handle () const { return handle_; }
```

这些方法能用于设置和获取某些套接字选项，例如支持“带外”数据[Ste98]，这些选项未被原始的套接字包装器外观定义。 ❑

当然应该少用安全舱口机制，因为它们降低可移植性并增加出错的可能，从而失去包装器外观模式的主要优点。如果应用程序在类似的情况下反复使用某些安全舱口，那么也许应该给包装器外观的公共接口增加显式的方法。扩展接口模式定义增加这些新方法的技术，同时不干扰现有的客户机。

64

(4) *开发一种错误处理机制*。低层C操作系统函数API常常使用类似errno这样的返回值和整数代码，给调用它们的代码返回错误。然而，如果调用者不检查函数调用的返回状态，那么这种技术也容易出错。

更出色的报错的方法是使用异常处理。类似C++和Java这样的许多编程语言支持把异常处理作为一种基本的报错机制。某些操作系统也使用它，例如Win32[Sol98]。使用异常处理作为包装器外观类的错误处理机制有多个优点：

- *它是可扩展的*，例如通过在C++和Java中定义异常类的层次。
- *它从正常处理中清楚地分解错误处理*。错误处理信息既不被显式地传递给操作，应用程序也不会因未能检查函数返回值而偶然地忽略异常。
- *它可以是类型安全的*。在类似C++和Java的语言中，异常会被抛出，并以强类型方式被捕获。

➡可以定义下列异常类，跟踪哪种操作系统错误或者条件已发生：

```
class System_Ex : public exception {
public:
    // Map <os_status> into a platform-independent error
```

```
    // or condition status and store it into <status_>.
    System_Ex (int os_status) { /* ... */ }

    // Platform-independent error or condition status.
    int status () const { return status_; }
    // ...
private:
    // Store platform-independent error/condition status.
    int status_;
};
```

通过在所有操作系统都映射到惟一值的宏或者常量，可以定义平台无关的错误和条件。例如，前面显示的Solaris实现中Thread_Mutex::acquire()方法可以如下编写：

65

```
void Thread_Mutex::acquire () {
    int result = mutex_lock (&mutex);
    if (result != 0) { throw System_Ex (result); }
}
```

❑

但是，用于包装器外观类的异常处理的使用存在几个缺陷：

- 并不是所有的语言或者实现都提供异常处理。例如，C没有定义异常模型，某些C++编译器不能实现异常。
- 各种语言以不同的方法实现异常。因此当用不同语言编写的组件抛出异常时，集成它们是非常困难的。Windows NT采用结构化异常处理[Sol98]，如果使用这样的特性化的异常处理机制，也会降低使用这些机制的应用程序的可移植性。
- 如果在C++或者Java代码块中存在多个退出路径，那么资源管理会复杂化[Mue96]。如果语言或者编程环境不支持无用单元收集，那么必须确保当抛出异常时，删除动态分配的对象。
- 甚至当不抛出异常时，低劣的异常处理实现也会增加时间或者空间开销[Mue96]。这种开销对于必须是有效的、小内存量的嵌入式系统而言是有问题的[GS98]。

如果封装对象是内核级设备驱动程序，或者是必须在多种平台上运行的低层操作系统API[Sch92]，这些包装器外观（例如上述的互斥、套接字和线程包装器外观）采用异常处理的缺点尤其明显。用于系统级包装器外观[Sch 97]的常用错误处理机制建立在线程特定的存储器模式之上，辅以errno的使用。虽然比使用C++异常来说，这种方法显得突兀，更可能出错，但它却是有效的、可移植的和线程安全的。

66

(5) 定义相关的帮助者类(可选的)。当更低层API被封装到包装器外观类中之后，创建进一步简化应用程序开发的帮助者类就变得可行了。常常只是在已经运用包装器外观模式聚合更低层的函数及其相关联的数据到类中之后，这些帮助者类的优点才显示出来。

➡在我们的例子中，可以调整在策略化加锁模式[Str97]中定义的Guard模板类。这个类确保在一个范围内都能正确地获取和释放Thread_Mutex，而无论方法的控制流如何离开范围。在范围内Guard类构造函数自动地获取互斥，同时析构函数自动地释放互斥：

```
{
    // Constructor of <guard> automatically
    // acquires the <mutex> lock.
    Guard<Thread_Mutex> guard (mutex);
```

```
    // ... operations that must be serialized.

    // Destructor of <guard> automatically
    // releases the <mutex> lock.
}
```

当仍然使用Guard的自动化加锁和解锁协议时，我们可以容易地替换不同类型的加锁机制，因为我们把类作为Thread_Mutex包装器外观使用。例如，可以用Process_Mutex类取代Thread_Mutex类：

```
// Acquire a process-wide mutex.
Guard<Process_Mutex> guard (mutex);
```

使用更低层的C函数和数据结构而不是C++类，要获得这种程度的“可插性”会相当困难。主要问题是函数和数据缺少对内聚性的语言支持，而C++类很自然地提供这种支持。 ❑

8. 已解决的例子

下面代码是我们登录服务器的logging_handler()函数，经过“实现”一节描述的互斥、套接字和线程包装器外观重新编写。为了方便和原始代码对比，我们分双栏把它和原始代码一起展现，左栏来自例子部分，右栏是新代码。

67

```
#if defined (_WIN32)
   #include <windows.h>
   typedef int ssize_t;
#else
   typedef unsigned int UINT32;
   #include <thread.h>
   #include <unistd.h>
   #include <sys/socket.h>
   #include <netinet/in.h>
   #include <memory.h>
#endif /* _WIN32 */

// Keep track of number of logging requests.
static int request_count;

// Lock to protect request_count.
#if defined (_WIN32)
   static CRITICAL_SECTION lock;
#else
   static mutex_t lock;
#endif /* _WIN32 */

// Maximum size of a logging record.
static const int LOG_RECORD_MAX = 1024;

// Port number to listen on for requests.
static const int LOGGING_PORT = 10000;

// Entry point that writes logging records.
int write_record (const char log_record[], size_t len) {
   /* ... */ return 0;
}

// Entry point that processes logging records for
// one client connection.
#if defined (_WIN32)
   u_long
#else
   void *
#endif /* _WIN32 */
logging_handler (void *arg) {
#if defined (_WIN32)
   SOCKET h = reinterpret_cast <SOCKET> (arg);
#else
   int h = reinterpret_cast <int> (arg);
#endif /* _WIN32 */

   for (;;) {
      // Ensure a 32-bit quantify;
#if defined (_WIN32)
      ULONG len;
#else
      UINT32 len;
#endif /* _WIN32 */
            char log_record[LOG_RECORD_MAX];

      // The first <recv> reads the length
      // (stored as a 32-bit integer) of
      // adjacent logging record.
      ssize_t n = recv (h, &len, sizeof len, 0);

      if (n <= sizeof len) break; // Bailout on error.
      len = ntohl (len);
      if (len > LOG_RECORD_MAX) break;
      // Loop to <recv> the data.
```

```
#include "ThreadManager.h"
#include "ThreadMutex.h"
#include "Guard.h"
#include "INET_Addr.h"
#include "SOCKET.h"
#include "SOCK_Acceptor.h"
#include "SOCK_Stream.h"

// Keep track of number of logging requests.
static int request_count;

// Lock to protect request_count.
static Thread_Mutex lock;

// Maximum size of a logging record.
static const int LOG_RECORD_MAX = 1024;

// Port number to listen on for requests.
static const int LOGGING_PORT = 10000;

// Entry point that writes logging records.
int write_record (const char log_record[], size_t len) {
   /* ... */ return 0;
}

// Entry point that processes logging records for
// one client connection.

void *logging_handler (void *arg) {

   SOCKET h = reinterpret_cast <SOCKET> (arg);

   // Create a <SOCK_Stream> object.
   SOCK_Stream stream (h);

   for (;;) {
      // Ensure a 32-bit quantity.
      UINT_32 len;

      char log_record[LOG_RECORD_MAX];

      // The first <recv_n> reads the length
      // (stored as a 32-bit integer) of
      // adjacent logging record.
      ssize_t n = stream.recv_n (&len, sizeof len);

      if (n <= 0) break; // Bailout on error.
      len = ntohl (len);
      if (len > LOG_RECORD_MAX) break;
      // Second <recv_n> reads the data.
```

```
        for (size_t nread = 0; nread < len; nread += n) {
            n = recv (h, log_record + nread,
                      len - nread, 0);
            if (n <= 0) return 0;
        }
#if defined (_WIN32)
        EnterCriticalSection (&lock);
#else
        mutex_lock (&lock);
#endif /* _WIN32 */
        ++request_count;
        // A -1 return value signifies failure.
        if (write_record (log_record, len) == -1)
            break;
#if defined (_WIN32)
        LeaveCriticalSection (&lock);
#else
        mutex_unlock (&lock);
#endif /* _WIN32 */
    }
#if defined (_WIN32)
    closesocket (h);
#else
    close (h);
#endif /* _WIN32 */
    return 0;
}
```

```
        n = stream.recv_n (log_record, len);

        if (n <= 0) break;
        {
            // Constructor acquires the lock.
            Guard<Thread_Mutex> mon (lock);

            ++request_count;
            // A -1 return value signifies failure.
            if (write_record (log_record, len) == -1)
                break;

            // Destructor releases the lock.
        }
    }

    return 0; // Destructor of <stream> closes down <h>.
}
```


右栏的代码和左栏显示的代码处理同样的问题。例如，SOCK_Stream和Guard的析构函数将分别关闭套接字句柄和释放Thread_Mutex，而不管代码块如何退出。右栏代码更容易理解、维护和移植，因为它更简洁，并且不使用与平台有关的API。

类似于logging_handler()函数，下面的两栏表用于比较main()函数的原始代码和使用包装器外观的新代码。

```
// Main driver function for the server.
int main (int argc, char *argv[]) {
    struct sockaddr_in sock_addr;
    // Handle UNIX/Win32 portability.
#if defined (_WIN32)
    SOCKET acceptor;
    WORD version_requested = MAKEWORD(2, 0);
    WSADATA wsa_data;
    int error = WSAStartup(version_requested, &wsa_data);
    if (error != 0) return -1;
#else
    int acceptor;
#endif /* _WIN32 */
    // Create a local endpoint of communication.
    acceptor = socket (PF_INET, SOCK_STREAM, 0);
    // Set up the address to become a server.
    memset (reinterpret_cast<void *> (&sock_addr),
            0, sizeof sock_addr);
    sock_addr.sin_family = PF_INET;
    sock_addr.sin_port = htons (LOGGING_PORT);
    sock_addr.sin_addr.s_addr = htonl (INADDR_ANY);
    // Associate address with endpoint.
    bind (acceptor, reinterpret_cast<struct sockaddr *>
          (&sock_addr), sizeof sock_addr);
    // Make endpoint listen for connections.
    listen (acceptor, 5);
    // Main server event loop.
    for (;;) {
        // Handle UNIX/Win32 portability.
#if defined (_WIN32)
        SOCKET h;
        DWORD t_id;
#else
        int h;
        thread_t t_id;
#endif /* _WIN32 */
        // Block waiting for clients to connect.
        h = accept (acceptor, 0, 0);

        // Spawn a new thread that runs the <server>
        // entry point.
#if defined (_WIN32)
        CreateThread (0, 0,
            LPTHREAD_START_ROUTINE(&logging_handler),
            reinterpret_cast <void *> (h), 0, &t_id);
#else
        thr_create
            (0, 0, logging_handler,
            reinterpret_cast <void *> (h),
            THR_DETACHED, &t_id);
#endif /* _WIN32 */
    }
    return 0;
}
```

```
// Main driver function for the server.
int main (int argc, char *argv[]) {
    INET_Addr addr (LOGGING_PORT);

    // Passive-mode acceptor object.
    SOCK_Acceptor server (addr);
    SOCK_Stream new_stream;
    // Main server event loop.

    for (;;) {

        // Accept a connection from a client.
        server.accept (new_stream);

        // Get the underlying handle.
        SOCKET h = new_stream.get_handle ();

        // Spawn off a thread-per-connection.

        Thread_Manager::instance()->spawn
            (logging_handler
             reinterpret_cast <void *> (h),
             THR_DETACHED);
    }

    return 0;
}
```

注意，在右栏使用包装器外观模式的版本可以少写几十行低层的、条件编译的代码。

9. 已知使用

微软基础类(Microsoft Foundation Classes，MFC)。MFC[Pro99]提供一系列封装许多更低层C Win32 API的包装器外观。主要是提供了实现微软文档-视图体系结构的GUI组件上，它是在[POSA1]中描述的文档-视图体系结构的变体。

ACE。在“实现”部分描述的套接字、线程和互斥包装器外观是ACE框架[Sch97]组件的抽象，这些组件分别为ACE_SOCK*，ACE_Thread_Manager和ACE_Thread_Mutex类。

Rogue Wave（异常波）。Rogue Wave的Net.h++和Threads.h++类库为在许多操作系统平台上的套接字、线程和同步机制实现包装器外观。

ObjectSpace（对象空间）。对象空间系统<工具包>(System<Toolkit>)也为套接字、线程和同步机制实现了平台无关的包装器外观。

Java虚拟机和Java类库。类似AWT和Swing[RBV99]这样的Java虚拟机和各种Java类库，提供一系列封装许多低层本地操作系统调用和GUI API的包装器外观。

西门子REFORM。用于热轧机处理自动化的REFORM框架[BGHS98]使用包装器外观模式将系统的面向对象部分（例如材料跟踪和凝固点传递）与用于实际的处理控制的神经网络相隔离。这个神经网络依据它的算法性质用C编程，并包含刻画自动化处理的物理特点的数学模型。

定义在REFORM框架中的包装器外观与用于操作系统机制的包装器外观不同，因为它们封装的处理控制API位于抽象的更高级。实际上，神经网络是REFORM系统自身的一部分。然而，与复杂的、面向对象的热轧机框架的高级结构和逻辑相比，它的基于函数的C API是更低层的。因此，REFORM包装器外观具有同更低层的操作系统包装器外观类似的目的和属性：

70

- 它们提供处理控制神经网络的框架所需的面向对象部分的视图和抽象。每个使用神经网络的组件存在一个分离的包装器外观。
- 它们隐藏API变体。对于不同的与客户机有关的框架实例，存在有(略微)不同的神经网络实现。因此，在这些神经网络实现中，语义上一致的函数可以有不同的特征。然而，这些差别并不影响框架实现。
- 它们确保以正确的顺序调用更低层C函数。

论文集汇编成的著作。一个真实的包装器外观模式的例子是由论文集汇编成的著作，这些论文被组织为一个或者多个“主题”。例如，PLoPD系列[PLoPD1][PLoPD2] [PLoPD3][PLoPD4]由各自的论文组成，这些文章被组织为内聚的章节，例如事件处理、容错、应用程序框架设计或者并发。因此，对特殊主题或者领域感兴趣的读者，可以把他们的注意力放在这些章节，而不必挨篇地去找相关的论文。

10. 结论

包装器外观模式有下列**优点**：

具体的、内聚的和健壮的更高级面向对象的编程接口。可以使用包装器外观模式把更低层API封装到一个更具体和更内聚的更高级面向对象的类集中。这些抽象减少了开发应用程序的冗长乏味，因而减少了潜在的某些类型的编程错误。另外，使用封装消除了不正确使用如套接字或者文件句柄这样的非类型化数据结构而导致的编程错误。因此，应用程序代码可以使用包装

器外观正确地和一致地访问更低层的API。

可移植性和可维护性。可以实现包装器外观以避免应用程序开发人员使用更低层API的不可移植的特性。通过使用如基类、子类及其关系这样的逻辑设计实体，替换如文件和#ifdef这样的基于物理设计实体的应用程序配置策略，包装器外观模式改进了软件结构。依据逻辑设计而不是物理设计，常常可以更容易理解、维护和增强应用程序[Lak95]。 71

模块性、可复用性和可配置性。包装器外观模式创建内聚的和可复用的组件，这些组件能够使用类似继承和参数化类型这样的面向对象的语言特征，成批“插入”到其他的组件中。如果不借助于如链接器或文件系统这样粗粒度的操作系统工具，替换成组的函数是很困难的。

包装器外观模式有如下几点**不足**：

功能丧失。一旦一个抽象位于现有抽象的上层时，它就有可能丧失功能。特别是当新的抽象阻止开发人员访问底层抽象的特定能力时，上述情况就会发生。定义一个合适的覆盖所有用况的高级抽象，又不使抽象变得膨胀，是非常困难的。一个应遵循的原则是设计包装器外观时，要使它们易于正确使用，而难以不正确地使用，但还要可能以原始的设计者未曾预测的方法使用。“安全舱口”机制或者开放实现[KLM+97]技术常常能清楚地帮助减缓这些设计压力。

性能降低。包装器外观模式会降低性能。例如，如果包装器外观类用桥模式[GoF95]实现，或者如果它们在每个方法上做几个前向函数调用，那么额外的间接性也许比直接使用更低层的API编程花费更大。然而，类似C++或者某些C编译器这样的支持内联的语言，能以不大的开销实现包装器外观模式，因为编译器能内联用于实现包装器外观的方法调用。因此开销与直接调用更低层函数的开销相同。

编程语言和编译器的局限性。为精心设计的C API定义C++包装器外观是相对直接的，因为C++语言和C++编译器定义了便于跨语言集成的特性。然而由于缺乏语言支持或者编译器的局限性，用其他语言定义包装器外观或许比较困难。例如，集成C函数到类似Ada、Smalltalk和Java这样的语言，就不存在可广为接受的标准。因此编程人员或许需要使用非可移植性的机制来开发包装器外观。 72

参见

包装器外观模式与[GoF95]中的几个结构的模式有关，它们是外观、桥、适配器和装饰器。

外观（Facade）。外观的目的是提供统一接口，简化客户机访问子系统接口。包装器外观的目的更加特定：它提供简洁的、健壮的、可移植的、可维护的和内聚的类接口，这些类接口封装如操作系统互斥、套接字、线程和GUI C API这样的更低层API。通常，外观在简单的API后面隐藏了复杂的类关系，而包装器外观在丰富的面向对象的类后面隐藏了复杂的函数和数据结构API关系。包装器外观也提供能被“插入”到更高级对象或者组件中的积木式组件。

桥（Bridge）。桥的目的是从抽象的实现中分解抽象，因此外观和桥可以通过多态性独立地和动态地改变。包装器外观有相似的目的：尽可能减少间接方法和多态性的开销。然而，由于

它们封装的系统编程机制的特性，包装器外观的实现很少动态地变化。

适配器（Adapter）。适配器的目的是把一个类的接口转化为客户机所期望的另一个接口。包装器外观的一个常用应用是创建一个类集。这些类“匹配”低层操作系统API以创造对所有应用程序看起来相同的可移植的包装器外观集。尽管这种解决方案的结构既不与对象一致，也不与[GoF95]中适配器的类的形式一致，但包装器外观通过导出一个跨平台的常用的面向对象的接口，扮演类似适配器的角色。

装饰器（Decorator）。装饰器的目的是通过透明地附加责任，动态地扩展对象。而包装器外观则静态地用面向对象的类接口封装更低层的函数和数据。

73

通常，当存在现有的更低层的、非面向对象的API需要封装时，同时当解决方案的有效性比动态地扩展更重要时，应运用包装器外观作为这些模式的替代品。

层(Layer)模式[POSA1]有助于把多个包装器外观组织到一个独立的组件层中。这个层直接驻留在操作系统之上，从而使应用程序避免和它们所使用的所有低层API打交道。

致谢

感谢Brad Appleton、Mike Curtis、Luciano Gerber、Ralph Johnson、Bob Hanmer、Patrick Rabau、Roger Whitney和Joe Yoder，感谢他们对实质性地提高包装器外观模式的形式和内容所提出的宝贵意见。

在包装器外观的公开评审期间，我们为这个模式的名字展开了讨论。因为软件开发人员常常把包装器外观模式所描述的内容称做“包装器”，所以好几个人建议称它为包装器(Wrapper)。可是，术语“包装器”已在模式研究团体中过多地使用。例如，包装器在适配器和装饰器模式[GoF95]的“别名”部分列出。然而，本书的模式与这些模式并不相同，因此我们决定为这里所呈现的模式使用一个不重叠的名字。

74

2.2 组件配置器

组件配置器(Component Configurator)设计模式允许应用程序在不必修改、重新编译、静态地重新链接应用程序的情况下，在运行时链接和解链它的组件实现。组件配置器进一步支持在不必关闭和重启运行的进程的情况下，把组件重新配置到不同的应用程序进程。

1. 别名

服务配置器(Service Configurator)[JS97b]

2. 例子

分布式的时间服务[Mil88][OMG97c]为在局域网或者广域网上协作的计算机提供了准确的时钟同步。它的体系结构包括三种类型的组件：

- 时间服务器(Time Server)组件回答有关当前时间的查询。
- 职员(Clerk)组件查询一个或者多个时间服务器，采样他们当前时间的标志，计算“大致的”准确时间，并且相应地更新它们自己的本地系统时间。

- 客户机(Client)应用程序组件使用它们的职员维护的全局一致的时间信息，与其他主机的客户机一起同步其行为。

实现这种分布式时间服务的常规方法是在编译时，静态地把时间服务器、职员和客户机组件的功能配置到在网络的主机上运行的分离的进程中（如图2-5）。

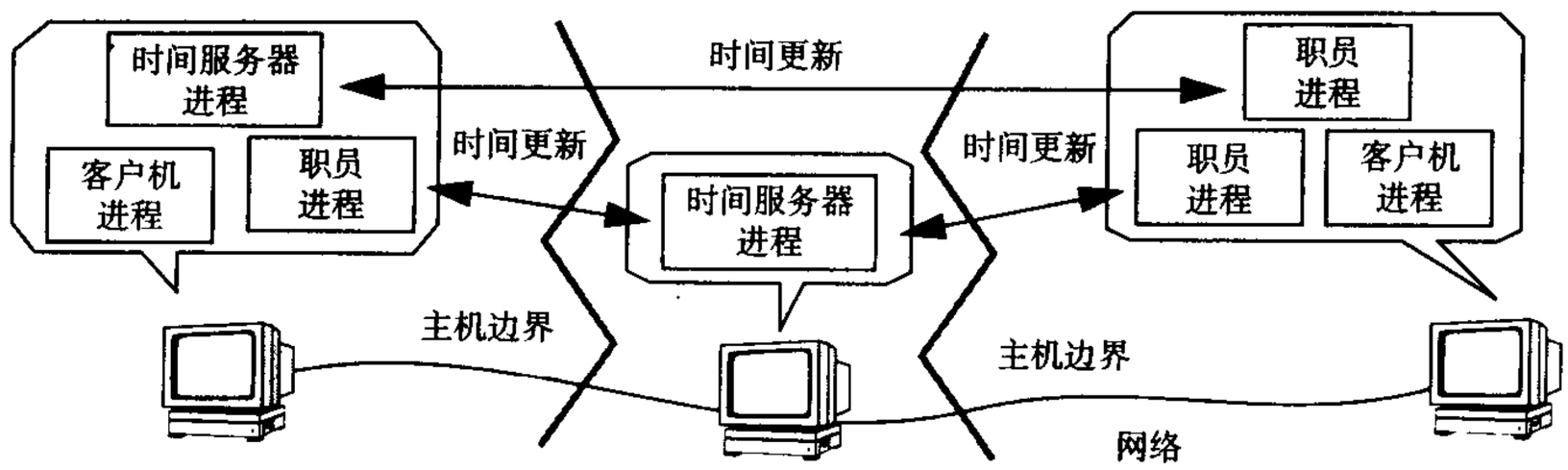


图 2-5

在这种配置中，主机运行的进程包括时间服务组件，它们处理时间更新的请求。职员组件在那些应用程序需要全局时间同步的主机的某个进程上运行。应用程序进程中的客户机组件使用它们本地的职员组件报告的同步时间执行计算。

这种方式虽然可以实现分布式时间服务的设计，但会碰到两类常规问题。

- 组件实现的选择会依赖于应用程序运行的环境。例如，如果可以使用一个WWV接收器，[⊖]那么Cristian时间服务算法[Cris89]是最合适的。否则，Berkeley算法是一种更好的选择。

  因此，改变应用程序运行的环境也需要改变时间服务组件的实现。然而，我们的设计是在编译时进程中特殊组件的实现就被静态地固化了，这使交换这个组件的实现变得困难。另外，由于每个组件和一个进程静态地耦合，因此当运行环境改变发生时，必须修改、重新编译和静态地重新链接现有的应用程序。
- 也许需要重新配置组件，以便增强类似延迟和吞吐量这样的关键的服务质量（QoS）属性。例如，可以在分布式时间服务中重新配置职员和时间服务器组件，以便它们配置在同一主机上［WSV99］。在这种情况下，通过允许职员借助于共享内存访问时间服务器的时间标记，而不是通过管道或者“回送”套接字连接交换数据，可以最小化通信开销。

  然而，如果组件被静态地配置到进程中，那么做上述的改变需要终止、重新配置和重新启动运行的时间服务进程。这些活动不仅是低效的，而且对不能容忍停机时间的系统也是不可行的。

不幸的是，类似桥(Bridge)和策略(Strategy)[GoF95]这样的模式本身没有足够的能力解决这类问题。例如，桥和策略常用于减少组件之间不必要的耦合。然而，当在我们的例子应用程序中独立地使用这些模式时，必须在编译时配置时间服务组件的所有可能实现，以便在运行时支持对不同策略的选择。这种限制对于某些应用程序而言过于僵化，而且代价太高。

⊖ WWV接收器截取美国国家标准时间研究所（NIST）广播的短脉冲，给公众提供调整标准时间。

例如，如果一个时间服务在具有严格的内存和能耗限制的个人计算设备上运行，那么当前没被使用的组件应被解链，以便减少资源消耗。类似桥和策略这样的模式并不直接涉及到这种“动态的重新配置”特性。

3. 语境

应用程序或者系统，其中的组件必须尽可能灵活和透明地被启动、挂起、恢复和终止。

4. 问题

由组件构成的应用程序必须提供一种机制，把这些组件配置到一个或多个进程中去。解决这个问题受三种强制条件的影响：

- 在许多系统和应用程序中，组件功能或者实现细节的改变是很普遍的。例如，随着应用程序的成熟，也许会发现更好的算法或者体系结构。因此，应当有可能在应用程序的开发和部署生命期期间的任何点上修改组件实现。

  对一个组件的修改应该对使用它的其他组件的实现有最小的影响。类似地，在运行时一个应用程序内，也应有可能动态地启动、挂起、恢复、终止或交换组件。这些活动应该对配置到应用程序中的其他组件有最小的影响。

- 开发人员在开发应用程序时，常常并不知道最有效地配置或者分布多个组件到进程和主机中的方法。如果开发人员过早地专注于组件的特殊配置，那么这也许会破坏灵活性，降低整个系统的性能和功能特性，并且不必要地增加资源的使用。

  另外，随着时间的推移，最初的组件配置决策也许被证明为是次优的。例如，平台的更新或者工作负载的增加，可能需要重新分布某些组件到其他的进程和主机。在这种情况下，在应用程序的开发或者部署周期中，不要冒失地修改或关闭应用程序，尽可能晚地做这些组件配置决策也许是有益的。

- 执行类似组件的配置、初始化和控制这样的普通管理任务应该是直接的，并且是与组件无关的。这些任务常常由一个中心管理员就能最有效地管理，而不是被分布到整个应用程序或者系统中。只要有可能，就应该使它们自动化，例如通过使用某种类型的脚本机制[MGG00]。

5. 解决方案

从组件的实现中分离出组件接口，使应用程序独立于组件实现被配置到应用程序进程的时间点。

详细地说：组件(Component)定义一个一致的接口，用来配置和控制它提供的特殊类型的应用程序服务或者功能。具体组件(Concrete Component)针对具体的应用程序实现这个接口。应用程序或者管理员可以使用组件接口动态地启动、挂起、恢复和终止它们的具体组件，还可以获取有关每个被配置的具体组件的运行信息。具体组件被打包进一个合适的配置单元，例如一个动态链接库（DLL）。在组件配置器的控制下，这个DLL能被动态地链接进和解链出一个应用程序，组件配置器使用组件仓库(Component Repository)来管理被配置到应用程序中的所有具体组件。

77
78

6. 结构

组件配置器模式包括四个参与者：

组件定义一个一致的接口，用于配置和控制组件实现所提供的应用程序服务或者功能特性的类型。一般的控制操作包括初始化、挂起、恢复和终止组件。

具体组件实现组件控制接口，以提供一个特定类型的组件。具体组件也实现方法以提供针对具体应用的功能，例如处理与其他的被连接的对等组件交换的数据。具体组件以某种形式（如DLL）打包，以便在运行时能被动态地链接进或解链出应用程序（如图2-6）。

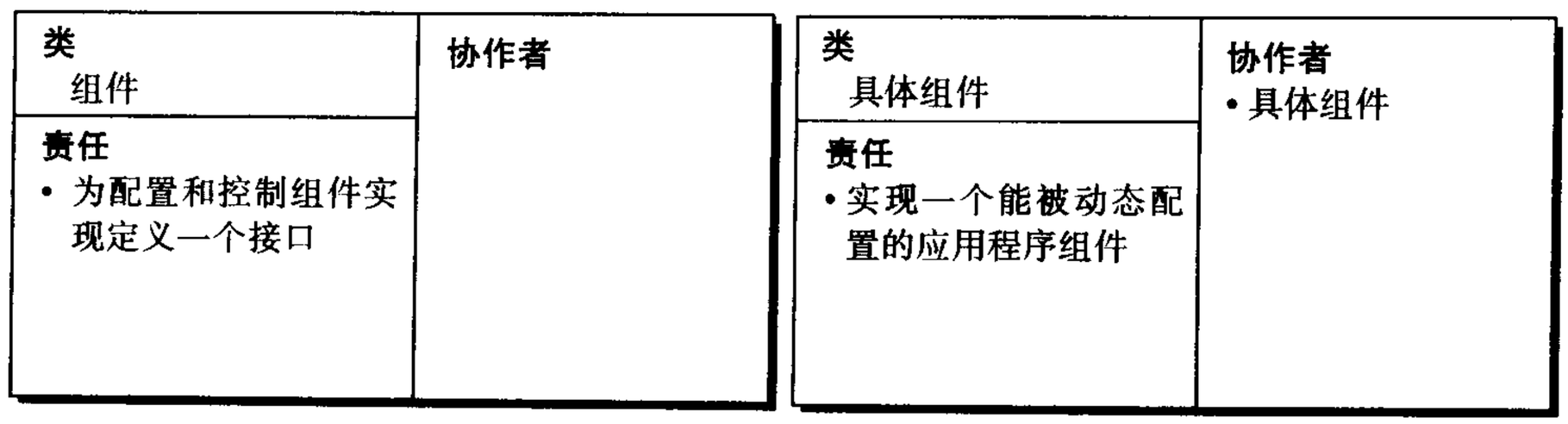


图 2-6

➡在我们的分布式时间服务中使用两种类型的具体组件：时间服务器和职员。每个具体组件都向分布式时间服务提供特定的功能。时间服务器组件接收和处理从职员来的时间更新请求。职员组件查询一个或者多个时间服务器，以确定“大致的”正确时间，并且使用这个值来更新它自己的本地系统时间。在我们的例子中有两种时间服务器实现，一个用于Cristian算法，另一个用于Berkeley算法。❑

组件仓库管理当前被配置到应用程序中的所有具体组件。这个仓库允许系统管理应用程序或者管理员通过中心管理机制，来控制被配置的具体组件的行为。 79

组件配置器使用组件仓库来协调具体组件的（重新）配置。它实现一种解释和执行脚本的机制，此脚本指定哪个可用的具体组件，通过从DLL中动态地链接和解链来（重新）配置到应用程序中（如图2-7）。

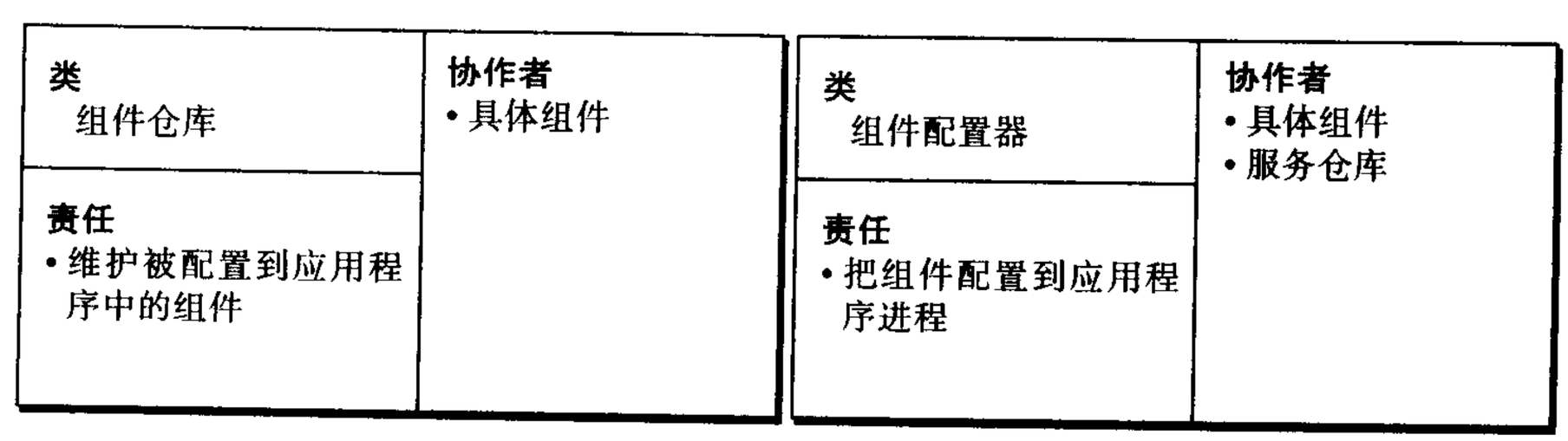


图 2-7

组件配置器模式的类图如图2-8所示。

7. 动态特性

可以用三个阶段表征组件配置器模式的行为（如图2-9）：

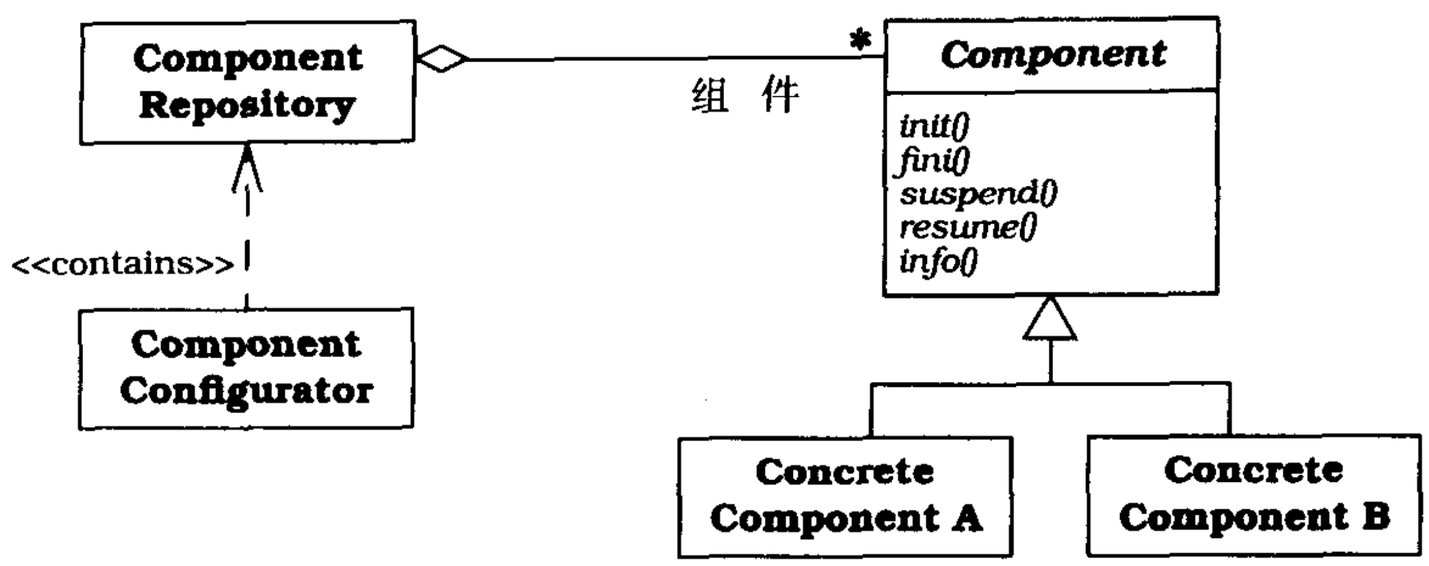


图 2-8

- 组件初始化(Component Initialization)。组件配置器动态地把一个组件链接到应用程序中并对它初始化。[⊖] 当一个组件被成功初始化之后，组件配置器把它加入到它的组件仓库中。在运行时这个仓库管理所有被配置的组件。
- 组件处理(Component Processing)。当组件被配置到应用程序中之后，组件执行它的处理任务，例如和对等的组件交换消息并执行服务请求。组件配置器能临时挂起和恢复现有的组件，比如当(重新)配置其他的组件时。
- 组件终止(Component Termination)。当不再需要组件之后，组件配置器关闭组件，同时允许组件在终止前有机会清除它们的资源。当终止一个组件时，组件配置器把它从组件仓库中删除，并把它从应用程序的地址空间中解链。

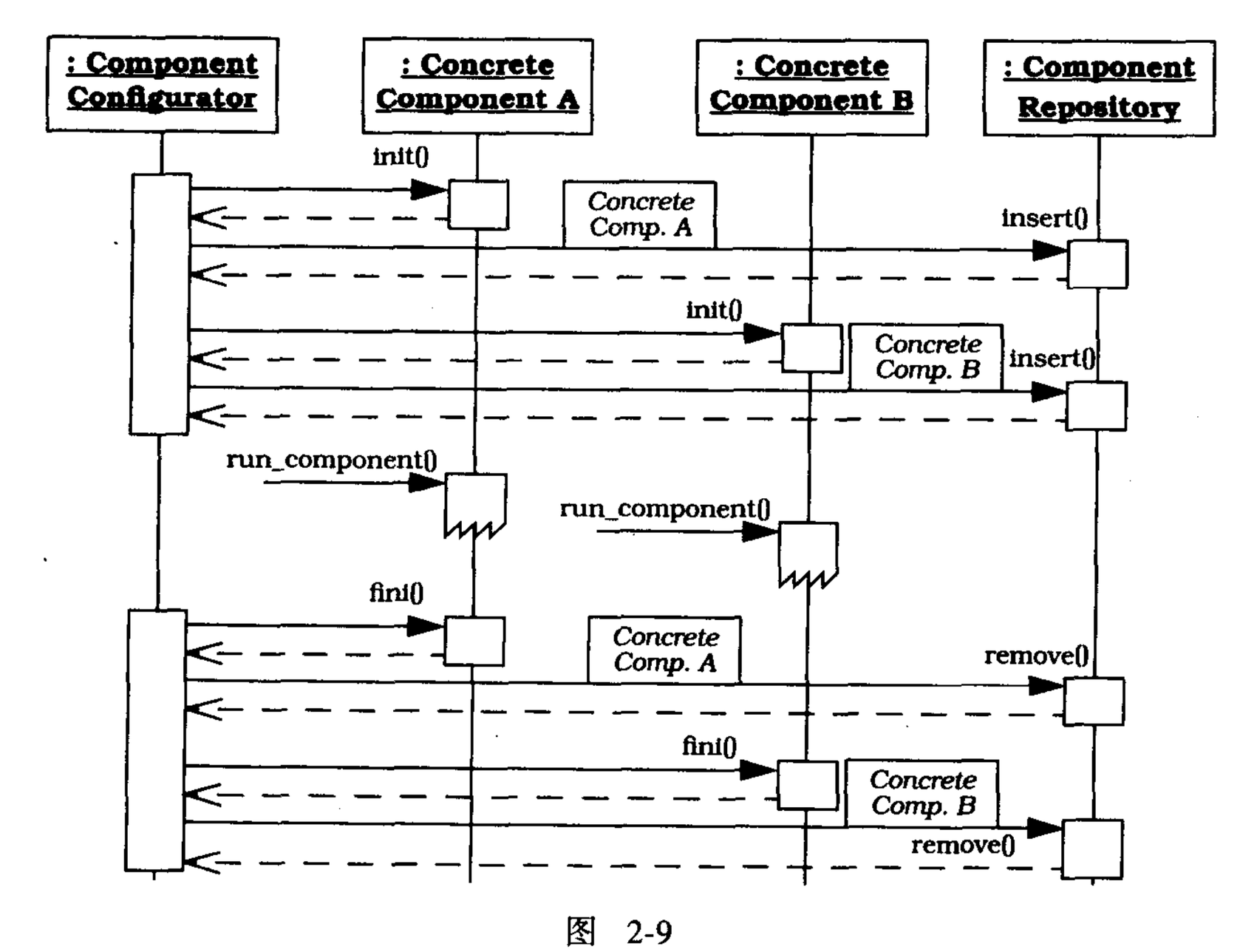


图 2-9

⊖ “实现”部分描述参数如何能被传送给组件，以及激活组件的不同方法。

图2-10状态图说明了组件配置器如何控制一个具体组件的生命期。

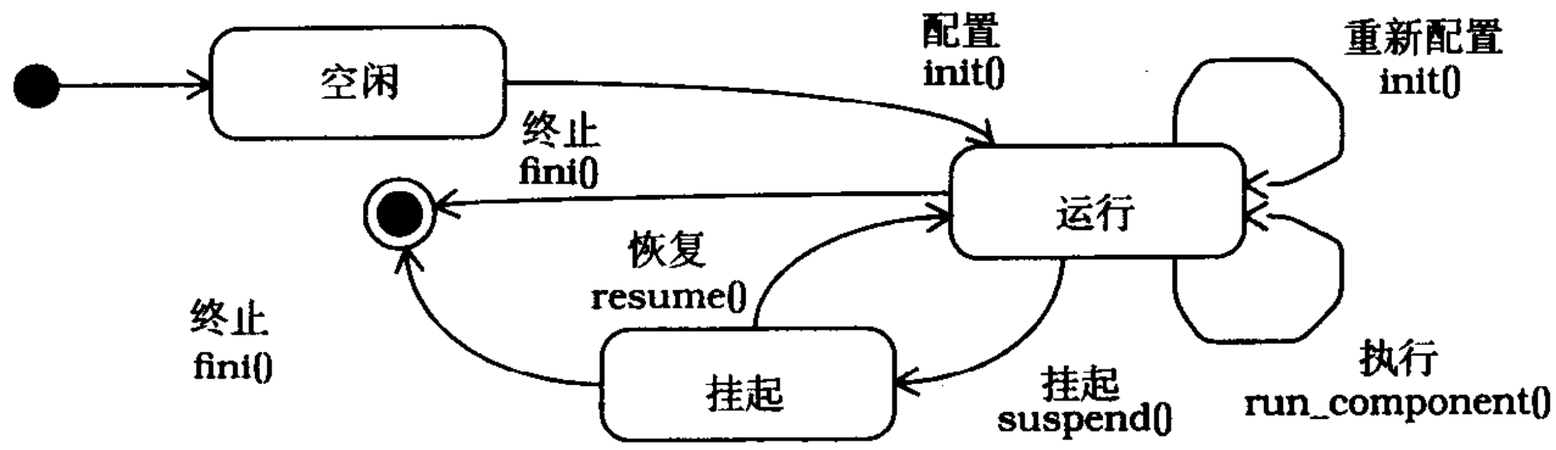


图 2-10

这个图阐明了组件配置器的事件驱动的“控制逆转”[Vlis98a]行为。例如，为响应类似CONFIGURE和TERMINATE这样的事件的发生，组件配置器调用相应的方法，本例中分别是init()和fini()。

8. 实现

可以把组件配置器模式中的参与者分解为两层：

- *配置管理层组件*(Configuration Management Layer Component)。这一层执行通用的、与应用无关的策略，这些策略安装、初始化、控制和终止组件。
- *应用层组件*(Application Layer Component)。这一层实现执行针对应用的处理的具体组件。

这部分的实现活动和配置管理层一起从“底”开始，向上工作到应用层的组件。

(1) *定义组件配置和控制接口*。组件应支持下列操作，以便组件配置器能配置和控制它们：

- *组件初始化*。初始化或者重新初始化组件。
- *组件结束*。关闭组件并清除它的资源。
- *组件挂起*。临时挂起组件执行。
- *组件恢复*。恢复被挂起的组件执行。 82
- *组件通知*。报告描述组件的静态和动态伪指令的信息。

用于配置和控制组件的接口可基于*继承策略*或者*消息传递策略*。

- *基于继承的接口*(Inheritance-based Interface)。在这个策略中，每个组件从一个常用基类继承，这个基类为各个组件的配置和控制操作包含纯虚拟钩子方法[Pree95]。
  - ➡下列抽象的Component类基于ACE框架[SchSu94]：

```
class Component : public Event_Handler {
public:
    // Initialization and termination hooks.
    virtual void init (int argc, const char *argv[]) = 0;
    virtual void fini () = 0;

    // Scheduling hooks.
    virtual void suspend ();
    virtual void resume ();

    // Status information hook.
    virtual void info (string &status) const = 0;
};
```

我们的时间服务例子所用的组件执行机制是基于在单个线程的控制范围内的反应事件处理模型，这将在反应器(Reactor)模式中进行描述。通过从反应器模式的Event_Handler参与者中继承，Component实现能向反应器注册自己，反应器接着多路分解和分配事件给组件。 ❑

- 基于消息的接口(Message-based Interface)。用于配置和控制组件的另一策略是使它们响应从组件配置器发来的一系列消息，例如INIT、SUSPEND、RESUME和FINI。组件开发人员必须编写代码分别处理这些消息，即初始化、挂起、恢复和终止一个组件。使用消息而不是继承，才有可能在C或Ada 83这样的缺乏继承的非面向对象的编程语言中实现组件配置器模式。

83

(2) 实现组件仓库。组件仓库管理所有通过DLL链接到应用程序中的具体组件实现。当组件被配置进或者配置出应用程序时，组件配置器使用这个仓库来控制它。每个组件的当前状态，比如它是活动的还是挂起的，都能在仓库中获得维护。

组件仓库可以是一个可复用的容器，例如Java的java.util.Hashtable[Sun00a]或C++标准模板库的map[Aus98]。相反，按照管理者(Manager)模式[Som97]，它可以作为一个容器实现。这个容器可以存储在主存、文件系统或者共享内存中。根据组件仓库驻留的位置，可以在应用程序范围内或者通过一个分离的进程管理它。

➡我们的Component_Repository类的接口也是基于ACE框架[SchSu94]的：

```
class Component_Repository {
public:
    // Initialize and close down the repository.
    Component_Repository ();
    ~Component_Repository ();

    // Insert a new <Component> with <component_name>.
    void insert (const string &component_name,
                 Component *);

    // Find <Component> associated with <component_name>.
    Component *find (const string &component_name);

    // Remove <Component> associated with
    // <component_name>.
    void remove (const string &component_name);

    // Suspend/resume <Component> associated with
    // <component_name>.
    void suspend (const string &component_name);
    void resume (const string &component_name);
private:
    // ...
};
```

84

❑

(3) 实现组件（重新）配置机制。在组件能够执行之前，它必须配置到应用程序的地址空间。组件配置器定义一种机制，控制把组件静态地和/或动态地（重新）配置进应用程序进程中去。组件配置器的实现包括五个子活动：

(3.1) 定义组件配置器接口。组件配置器通常作为一个单件外观（Singleton facade）[GoF95]实现。这种做法能够协调对其他组件配置器模式组件的访问，例如在实现活动（2）中描述的组

件仓库和在实现活动（3.3）中描述的解释组件配置伪指令的机制。

➡下列的C++接口是用于我们的分布式时间服务器例子的单件外观，它也是基于ACE的，

```
class Component_Configurator {
public:
    // Initialize the component configurator.
    Component_Configurator ();

    // Close down the component configurator and free up
    // dynamically allocated resources.
    ~Component_Configurator ();

    // Process the directives specified in the
    // <script_name> file.
    void process_directives (const string &script_name);

    // Process a single directive specified as a string.
    void process_directive
        (const string &directive_string);

    // Accessor to the <Component_Repository>.
    Component_Repository *component_repository ();

    // Singleton accessor.
    static Component_Configurator *instance ();
private:
    // ...
};
```

❑ 85

(3.2) 定义一种用于指定组件配置伪指令的语言。这些伪指令向组件配置器供应它在运行时定位和初始化组件实现所需的信息，也提供在初始化组件之后挂起、恢复、重新初始化和/或终止组件所需的信息。可以用多种方式指定组件配置伪指令，例如用命令行、环境变量、图形用户接口或者配置脚本。

➡为简化安装和管理，在我们的分布式时间服务器例子中的组件配置器使用与ACE所提供的类似的组件脚本机制[SchSu94]。我们称为comp.conf的脚本文件把组件配置伪指令合并到一个单独的地方，由应用程序、开发人员或者管理员集中管理。需要（重新）配置到应用程序中的每个组件由comp.conf脚本中的一个伪指令指定。

下列comp.conf脚本说明一个时间服务器如何能被动态地配置进应用程序中：

```
# Configure a Time Server.
dynamic Time_Server Component *
    cristian.dll:make_Time_Server()
            "-p $TIME_SERVER_PORT"
```

在这个comp.conf脚本中的伪指令包含一个dynamic命令，它指示解释器执行下列两个活动：

- 动态地链接cristian.dll DLL到应用程序的地址空间。
- 自动地调用make_Time_Server()工厂函数。

这个函数动态地分配一个新的时间服务器实例：

```
// Keep C++ compiler from non-portably mangling name!
extern "C"
```

```
Component *make_Time_Server () {
    // <Time_Server> inherits from the <Component>
    // class.
    return new Time_Server;
}
```

86

伪指令末尾的字符串参数“-p $TIME_SERVER_PORT”包含一个环境变量，此变量指定时间服务器组件监听并接收从职员来的连接的端口号。组件配置器把这个字符串转化为“argc/argv”格式的数组，并把它传送给时间服务器组件的init()钩子方法。如果init()方法成功地初始化组件，那么指向组件的指针被存储在“Time_Server”名字下的组件仓库中。这个名字标识出被最新配置的组件，这样代表应用程序或者管理员的组件配置器就可以动态地控制它。

在（3.3）中我们要介绍，组件配置器的伪指令解释器处理comp.conf脚本中的伪指令。每个伪指令以一个命令开始，该命令指示解释器如何配置、重新配置或者控制一个组件。如下表所示：

命　令	说　明
dynamic	动态地链接和初始化一个组件
static	初始化一个静态链接的组件
remove	从组件仓库中删除一个组件并对它解链
suspend	不删除组件，而是临时挂起它
resume	恢复先前被挂起的组件

可以使用下列BNF语法中定义的一个简单配置脚本语言编写伪指令。

```
<directive>::= <dynamic> | <static> | <suspend>
                         | <resume> | <remove>
<dynamic> ::= dynamic <comp-location> <parameters-opt>
<static> ::= static <comp-name> <parameters-opt>
<suspend> ::= suspend <comp-name>
<resume> ::= resume <comp-name>
<remove> ::= remove <comp-name>
<comp-location> ::= <comp-name> <type> <function-name>
<type> ::= Component '*' | NULL
<function-name>  ::= STRING ':' STRING '(' ')'
<parameters-opt> ::= '"' STRING '"'| NULL
<comp-name> ::= STRING
```

87

❑

(3.3) 实现一个机制，用于解析和处理组件配置伪指令。这种机制常常作为一个伪指令解释器(Directive Interpreter)实现，此解释器将组件的运行时特性与它的配置相关的特性分开。可以使用解释器(Interpreter)模式[GoF95]或者类似lex和yacc[SchSu94]这样的标准解析器-生成器工具实现伪指令解释器。

➥Component_Configurator外观类定义了两个方法，允许应用程序调用一个组件配置器的伪指令解释器。process_directives()方法能处理一连串的、存储在指定的脚本文件中的（重新）配置和控制伪指令。这种方法允许不断地存储多个伪指令，并且迭代地处理它们。反过来，process_directive()方法能处理作为字符串参数传递的单一伪指令。这种方法允许动态地创建和/或交互地处理伪指令。

❑

简单的伪指令解释器以指定的次序执行每个组件配置伪指令。这时，应用程序开发人员须要确保这个执行次序满足被配置的组件之间的任何排序依赖性。当然也能设计出更复杂的解释器和脚本语言，允许伪指令解释器自动地处理排序依赖性，例如通过使用拓扑排序。

(3.4) *实现动态的配置机制*。组件配置器使用这种机制动态地把组件链接进和解链出应用程序进程。现代操作系统，例如System V Release 4（SVR4）UNIX和Win32，通过显式的动态链接机制[WHO91]支持这个特性。

例如，SVR4 UNIX分别定义dlopen()、dlsym()和dlclose() API明确地把指定的DLL动态地链接进应用程序进程，从DLL中析取一个指定的工厂函数，并解链DLL。微软的Win32操作系统支持LoadLibrary()、GetProcAddr()和CloseHandle() API执行同样的功能。当组件配置器的伪指令解释器解析和处理伪指令时，它使用这些API动态地把DLL链接进和解链出应用程序的地址空间。

88

➡我们的Component_Configurator实现使用下列的显式动态链接API，它基于在ACE中定义的包装器外观[Sch97]：

```
class DLL {
    // This wrapper facade defines a portable interface to
    // program various DLL operations. The <OS::*>
    // methods are lower-level wrapper facades that
    // encapsulate the variation among explicit dynamic
    // linking APIs defined on different operating
    // systems.
public:
    // Opens and dynamically links the DLL <dll_name>.
    DLL (const string &dll_name) {
        handle_ = OS::dlopen (dll_name.c_str ());
    }

    // Unlinks the DLL opened in the constructor.
    ~DLL () { OS::dlclose (handle_); }

    // If <symbol_name> is in the symbol table of the DLL
    // return a pointer to it, else return 0.
    void *symbol (const string &symbol_name) {
        return OS::dlsym (handle_, symbol_name.c_str ();
    }
private:
    // Handle to the dynamically linked DLL.
    HANDLE handle_;
};
```

为说明组件配置器如何使用这个API，让我们考虑一下在（3.2）中所示的用于配置Time_Server组件的伪指令。在这个例子中，组件配置器执行七个步骤：

1) 它创建一个DLL对象，并把“cristian.dll”字符串传送给它的构造函数。

2) 接着，通过在DLL类的构造函数中调用的OS::dlopen()方法动态地把cristian.dll DLL链接进应用程序的地址空间。

3) 组件配置器接着把字符串“make_Time_Server()”传送给DLL对象的symbol()方法。

4) 这个方法使用OS::dlsym()方法在cristian.dll DLL的符号表中定位make_Time_Server入口，并返回指向这个工厂函数的指针。

89

5) 假定前四个步骤都成功的话，那么组件配置器调用工厂函数，此函数返回指向一个Time_Server组件的指针。

6) 组件配置器随后调用这个组件的init()方法，并把字符串“-p $TIME_SERVER_PORT”作为“argc/argv”格式的数组传递。init()方法是这个Time_Server组件用于初始化它自己的钩子。

7) 最后，组件配置器把被初始化的Time_Server组件存储到它的组件仓库中。

(3.5) 实现动态的重新配置机制。这种机制建立在上述的动态配置机制上，以触发组件实现的动态的重新配置（reconfiguration）。组件重新配置应该对应用程序进程中其他组件的执行影响最小。因此当实现一个动态的重新配置机制时，应当处理下列两个方面：

1) 定义重新配置的触发策略。有两种用于触发组件重新配置的策略：带内(in-band)和带外(out-of-band)：

- 带内策略通过使用如套接字连接或者CORBA操作这样的IPC机制，同步地启动重新配置。应用程序和/或组件配置器负责在指定的“重新配置点”检查这样的重新配置事件。
- 带外策略产生类似UNIX SIGHUP信号这样的异步事件，这些异步事件能中断一个运行的应用程序进程或者线程来启动重新配置。

在两种情况下，接受一个重新配置事件后，组件配置器都将解释一组新的组件配置伪指令。用于触发重新配置的带内策略通常更易于实现，因为存在竞争条件的可能性很小。然而，带内触发也许灵敏度不高，因为重新配置只能发生在指定的重新配置点。相反，带外重新配置触发更灵敏，但实现确定配置何时能发生的健壮协议比较困难。

90

2) 定义用于确保健壮的重新配置的协议。在实现重新配置机制时，另一个需要考虑的重要方面是健壮性(robustness)。例如，如果应用程序中的其他组件在使用一个正被重新配置的组件，那么组件配置器就不能执行立即删除或者挂起这个组件的请求。这时，应该允许有些组件在可以重新配置之前完成它们的计算任务。

如果有新的组件配置到应用程序中，那么其他的组件也许想得到通知，以便可以和新组件交互。类似地，当恢复一个被挂起的组件时，其他的组件也应得到通知，以便能恢复它们的计算。

组件配置器模式强调（重新）配置机制，例如如何去解释包含组件配置伪指令的脚本来动态链接和解链组件。因此，单纯地确保健壮的动态组件重新配置超出了组件配置器的讨论范围。支持健壮的重新配置需要组件配置器和与组件/配置有关的协议之间的协作。这些协议决定何时触发重新配置，链接哪个组件，和哪个组件交互，以便配置特殊的应用程序进程。

实现健壮的重新配置机制的一种方法是应用观察者(Observer)模式[GoF95]。想访问特殊组件的客户机组件称为观察者。这些观察者向组件配置器注册，后者包含一个通知者，起观察者模式的目标参与者的作用。

当计划终止一个组件时，组件配置器实现两阶段协议。第一阶段通知它的注册客户机组件“观察者”完成它们的计算。在第二阶段，在所有的客户机组件应答这个通知之后，组件配置器删除组件。当初始化新组件时，组件配置器重新通知它的注册客户机组件，指示它们可以连接新组件。

91

类似地，客户机组件可以向组件配置器注册，并且在一个特殊的组件的执行被挂起和恢复时得到通知。

➡例如，可以对Component和Component_Configurator类做下列改变，以支持基于观察者的重新配置机制。

```
class Component; // Forward declaration.

class Component_Configurator {
public:
    // Type of configuration-related events.
    enum Event_Type { INIT, SUSPEND, RESUME, FINI };

    // Register <notified_component> to receive
    // notifications when <observed_component> is
    // reconfigured or suspended/resumed.
    void register_observer
            (Component *notified_component,
             Component *observed_component);
    // ...
};

class Component : public Event_Handler {
public:
    // Hook method called back when <observed_component>
    // receives a configuration-related event.
    virtual void handle_update
            (Component *observed_component,
             Component_Configurator::Event_Type event);
    // ...
};
```

❑

(4) 实现具体组件。可以从一个在实现活动（1）中指定的类似Component类这样的常用基类中派生出具体组件类。也可以通过允许它们接收和处理组件控制消息的消息传递机制实现具体组件类。组件常常实现其他的方法，例如建立和远程对等组件的连接，处理从客户机接收的服务请求，等等。尽管组件也能和应用程序一起被静态链接，但一般情况下，组件实现驻留在DLL中。

实现具体组件包含三个子活动：

(4.1) 实现具体组件并发模型。实现具体组件的一个重要方面，涉及到组件的并发策略的选择。例如，可以使用类似反应器(Reactor)或者主动器(Proactor)这样的事件多路分解模式，也可以使用类似主动对象(Active Object)、监视器对象(Monitor Object)、半同步/半异步(Half-Sync/Half-Async)或领导者/追随者(Leader/Follower)这样的并发模式，执行配置到应用程序中的组件。 92

- 反应的/主动的执行(Reactive/Proactive Execution)。使用这些策略，就可以使用一个控制线程反应地或者主动地处理所有组件。因为能够减少甚至消除竞争条件，所以使用反应器模式实现的组件可以相对直接地进行（重新）配置和控制。然而，由于反应组件是单线程的，因此反应组件就不能同其他策略一样进行扩展。

  相反，使用主动器模式的组件可能比在支持异步I/O的平台上的反应性实现更有效。然而，由于取消异步操作的细微差别，重新配置和控制主动组件也许更复杂。详细细节参见主动器模式中有关不足的讨论。
- 多线程或多进程并发执行(Multi-threaded or multi-process concurrent)。使用这些策略，被

配置的组件经组件配置器初始化之后，就在它们自己的线程或者进程中执行。例如，组件可以使用主动对象模式并发地运行，或者按照领导者/追随者或半同步/半异步模式，在预先产生的线程或者进程池中执行。

像组件配置器那样在同一进程的一个或多个线程中执行组件，比在分离的进程中运行组件更有效。但反过来，因为操作系统和硬件保护机制能从偶发的崩溃[Sch94]中隔离每个组件，所以把组件配置到分离的进程应该更健壮和更安全。

(4.2) 实现用于组件间通信的机制。有些组件可以完全独立地运行，而另一些组件必须和其他组件通信。这时，组件开发人员必须选择一种用于组件间通信的机制。

93

常常根据通信组件是搭配式的还是分布式的，来决定对机制的选择。

- 如果组件是搭配式的，则一般有两种选择，一是对组件间指针关系进行硬编码，但这样做不太灵活，也会丧失动态组件配置的优点；另一种方法就是使用组件仓库“按名字”访问组件。

  ➡我们时间服务例子中的应用程序使用模板以类型安全的方式从单件Component_Configurator的Component_Repository中获取具体组件：

```
template <class COMPONENT>
class Concrete_Component {
public:
    // Return a pointer to the <COMPONENT> instance
    // in the singleton <Component_Configurator>'s
    // <Component_Repository> associated with <name>.
    static COMPONENT *instance (const string &name);
};
```

  instance()方法的实现如下所示：

```
template <class COMPONENT>
COMPONENT *Concrete_Component<COMPONENT>::instance
    (const string &name) {
    // Find the <Component> associated with <name>,
    // and downcast to ensure type-safety.
    Component *comp = Component_Configurator::
        instance ()->component_repository ()->find(name);
    return dynamic_cast<COMPONENT *> (comp);
}
```

  使用这种模板从组件仓库中获取组件：

```
Time_Server *time_server =
    Concrete_Component<Time_Server>::instance
        ("Time_Server");
// Invoke methods on the <time_server> component...
```

❑

- 当组件是分布式的时候，一般也有两种选择，一是低层IPC机制，如使用套接字[Ste98]或TLI[Rago93]编程的TCP/IP连接，此外是类似CORBA[OMG98a]的更高级机制。使用CORBA的优点是，通过自动地确定组件是在一起的还是分布式的[WSV99]，ORB可以透明地优化得到最快速的IPC机制。

94

(4.3) 实现重新建立组件关系的机制。在（4.2）中概述过，组件可以使用其他组件，甚至应用程序中的其他对象，以执行它们提供的服务。因此，在运行时用一个组件实现替换另一个，需要组件配置器自动地把新组件和被删除组件所使用的组件重新连接起来。

实现这种机制的策略之一是设置检查点，检查组件对它相关组件的引用，并把它存储在一个备忘录(Memento)[GoF95]中。组件在关闭之前，可以把这个备忘录传递给组件配置器。同样，备忘录可以包含从旧组件向新组件传递的其他状态信息。新组件安装后，组件配置器可以把备忘录传递给新组件。随后这个组件重新安装保存在备忘录中的连接和状态信息。

➡实现保存和重新建立组件关系的机制，需要对Component_Configurator和Component类做三个改变：

- *定义备忘录层次*。为每个具体组件类型定义一个备忘录，它保存组件类型能维护的对其他组件的引用。引用可以用组件的名字或者指针表示，这在（4.2）中概述过。所有备忘录从一个抽象备忘录派生。这允许Component_Configurator使用多态性处理任意备忘录。
- *实现用于在组件配置器中维护备忘录的机制*。在组件的重新配置期间，包含对其他组件引用的备忘录被存储在Component_Configurator中。用于在Component_Configurator内处理这个备忘录的相应的基础设施可以包含一个对备忘录的引用，也可以包含备忘录存储的引用的组件类型。
- *改变组件接口和实现*。为了在组件和Component_Configurator之间双向传递备忘录，必须改变Component接口。例如，可以把备忘录作为Component的init()方法的一个参数传递给它，并且通过Component的fini()方法中的参数返回给Component_Configurator。随后在具体组件的init()和fini()方法实现中，使用备忘录获取和保存组件对其他组件和对象的关系。

95

除组件引用之外，备忘录也会维护传递给新组件的其他状态信息。例如，Clerk组件会传递它们轮询时间服务器的频率，这样新的Clerk组件便能以相同的频率更新它们的本地系统时间。 ❑

这一节的余下部分，展示如何应用实现活动（4）和它的子活动来指导我们的分布式时间服务例子中具体组件参与者的实现。

➡在分布式时间服务中存在两类具体组件：Time_Server和Clerk。Time_Server组件接收和处理从Clerk来的时间更新请求。Time_Server和Clerk组件都是用接受器-连接器模式设计的。在实现活动（1）中讲过，用于Time_Server和Clerk的组件执行机制是基于在单个控制线程内的反应性事件处理模型，这与反应器模式一致。

Time_Server从Component类继承：

```
class Time_Server : public Component {
public:
    // Initialize and terminate a <Time_Server>.
    virtual void init (int argc, const char *argv[]);
    virtual void fini ();

    // Other methods (e.g., <info>, <suspend>, and
    // <resume>) omitted.
private:
    // The <Time_Server_Acceptor> that creates, accepts,
    // and initializes <Time_Server_Handler>s.
    Time_Server_Acceptor acceptor_;
```

```
    // A C++ standard library <list> of
    // <Time_Server_Handler>s.
    list<Time_Server_Handler *> handler_list_;
};
```

96

Time_Server是从Component继承来的，因此Component_Configurator能动态地把它链接和解链。这种设计把Time_Server的实现从配置它的时间或语境中分解出来，并且允许开发人员在不同的Time_Server算法之间快速切换。

在把Time_Server组件存储到组件仓库之前，应用程序的组件配置器单件调用组件的init()钩子方法。这允许Time_Server组件初始化它自己。

在内部，Time_Server包含Time_Server_Acceptor监听从Clerk来的连接请求。它还包含一个处理时间更新请求的Time_Server_Handler的C++标准模板库[Aus98]list。当调用Time_Server的init()方法时，创建Time_Server_Acceptor并向反应器注册。

当从Clerk发来新的连接请求时，接受器创建一个新的Time_Server_Handler，它处理随后从Clerk来的时间更新请求。当Time_Server_Acceptor调用它的init()方法时，每个处理器把自己向单件反应器注册。当时间更新请求到来时，此反应器随后分配处理器的handle_event()方法。

当组件配置器终止Time_Server时，它调用Time_Server的fini()方法。这个方法注销Time_Server_Acceptor和所有反应器来的与它相关联的Time_Server_Handler，并销毁它们。

我们提供两个Time_Server组件实现：

- 第一个组件实现Cristian算法[Cris89]。在这种算法中每个Time_Server是被动实体，响应Clerk的查询。尤其是，Time_Server并不主动地查询其他机器来确定自己的时间标记。
- 第二个组件实现Berkeley算法[GZ89]。在这种算法中，Time_Server是主动组件，它周期地轮询网络上的每台机器，以确定它的本地时间。基于它接收的响应，Time_Server计算出一个对正确时间的总的标记。

97

和上面的Time_Server一样，Clerk从Component类继承：

```
class Clerk : public Component {
public:
    // Initialize and terminate a <Clerk>.
    virtual void init (int argc, const char *argv[]);
    virtual void fini ();

    // <info>, <suspend>, and <resume> methods omitted.

    // Hook method invoked by a <Reactor> when a timeout
    // occurs periodically. This method contacts several
    // <Time_Server>s to compute its local notion of time.
    virtual void handle_event (HANDLE, Event_Type);
private:
    // The <Clerk_Connector> that connects and
    // initializes <Clerk_Handler>s.
    Clerk_Connector connector_;

    // A C++ standard library <list> of <Clerk_Handler>s.
    list<Clerk_Handler *> handler_list_;
};
```

由于Clerk是从Component继承的，因此组件配置器可以动态地链接和解链Clerk。类似地，组件配置器可以通过调用它的init()、suspend()、resume()和fini()钩子方法来配置、控制和重新配置它所管理的Clerk。

Clerk组件建立和维护与Time_Server的连接，并查询它们来计算当前的时间。Clerk的init()方法动态地分配Clerk_Handler，后者通过Clerk_Connector向连接的Time_Server发送时间更新请求。它也向反应器注册Clerk，以便定期接收超时事件，例如每五分钟一次。

在超时期间，反应器通知Clerk的handle_event()钩子方法。这个方法指示Clerk的Clerk_Handler向它们连接的时间服务器请求当前时间。Clerk接收和处理这些服务器应答，接着对应地更新它的本地系统时间。当Client向Clerk组件请求当前时间时，它们接收一个本地缓存的时间值，这个时间值已经和全局时间标记同步。Clerk的fini()方法关闭和清除它的连接器和处理器。

98

时间服务的两个可选的实现在两个DLL内提供。cristian.dll包含一个创建运行Cristian算法的组件的工厂。同样地，berkeley.dll包含一个创建运行Berkeley算法的组件的工厂。

9. 已解决的例子

在这一部分中，我们展示分布式时间服务器实现例子如何使用基于显式动态链接[SchSu94]的配置机制和comp.conf配置脚本来运用组件配置器模式。展示的步骤如下：

- 首先展示配置机制如何通过脚本支持把Clerk和Time_Server组件动态地配置到应用程序进程中。
- 接着展示这些特性如何允许Clerk组件改变用于计算本地系统时间的算法。尤其是，在已经选择一个新算法后，在不影响组件配置器控制的其他类型组件执行的情况下，单件Component_Configurator如何能动态地重新配置Clerk组件。

配置分布式时间组件应用程序有两种通用的策略：搭配式和分布式。我们概述每个策略，以便阐明如何动态地（重新）配置和运行可以使用组件配置器的应用程序。

搭配式配置(Collocated configuration)。这种配置使用comp.conf脚本把Time_Server和Clerk放在同一进程内。

99

通常的main()程序使用Component_Configurator对象的process_directives()方法动态地配置组件，随后运行应用程序的事件循环。事件循环是基于反应器模式的。

```
int main (int argc, char *argv[]) {
    Component_Configurator server;

    // Interpret the comp.conf file specified in argv[1].
    server.process_directives (argv[1]);

    // Reactor singleton perform component processing and
    // any reconfiguration updates.
    for (;;)
        Reactor::instance ()->handle_events ();
    /* NOTREACHED */
}
```

process_directives()方法在解释下列comp.conf配置脚本的同时，把组件动态地配置到服务器进程：

```
# Configure a Time Server.
dynamic Time_Server Component *
    cristian.dll:make_Time_Server()
        "-p $TIME_SERVER_PORT"

# Configure a Clerk.
dynamic Clerk Component *
    cristian.dll:make_Clerk()
        "-h tango.cs:$TIME_SERVER_PORT"
        "-h perdita.wuerl:$TIME_SERVER_PORT"
        "-h atomic-clock.lanl.gov:$TIME_SERVER_PORT"
        "-P 10" # polling frequency
```

comp.conf中的伪指令指定Component_Configurator如何在同一应用程序进程中使用Cristian算法动态地配置搭配式的Time_Server和Clerk。Component_Configurator把cristian.dll DLL动态地链接进应用程序的地址空间，并调用合适的工厂函数来创建新的组件实例。在我们的例子中，这些工厂函数称为make_Time_Server()和make_Clerk()，它们定义如下：

```
Component *make_Time_Server () { return new Time_Server; }
Component *make_Clerk () { return new Clerk; }
```

100

在每个工厂函数返回其新分配的组件之后，comp.conf脚本中指定的初始化参数被传递给各自的init()钩子方法。这些方法执行相应的与组件有关的初始化，参见实现活动（4）。

分布式配置(Distributed configuration)。为减少应用程序的内存覆盖，我们也许想在不同的进程中搭配使用Time_Server和Clerk。由于组件配置器模式的灵活性，所有需要用于分布这些组件的工作是把comp.conf脚本分为两部分，并且分别在不同的进程或者主机中运行它们。一个进程包含Time_Server组件，另一个进程包含Clerk组件。

图2-11展示在同一进程中搭配的Time_Server和Clerk的配置是怎样的，同时也展示重新配置分离后的新配置。注意，由于组件配置器模式在组件被配置时，已及时把它们的处理行为分解出来，因此组件自身不必改变。

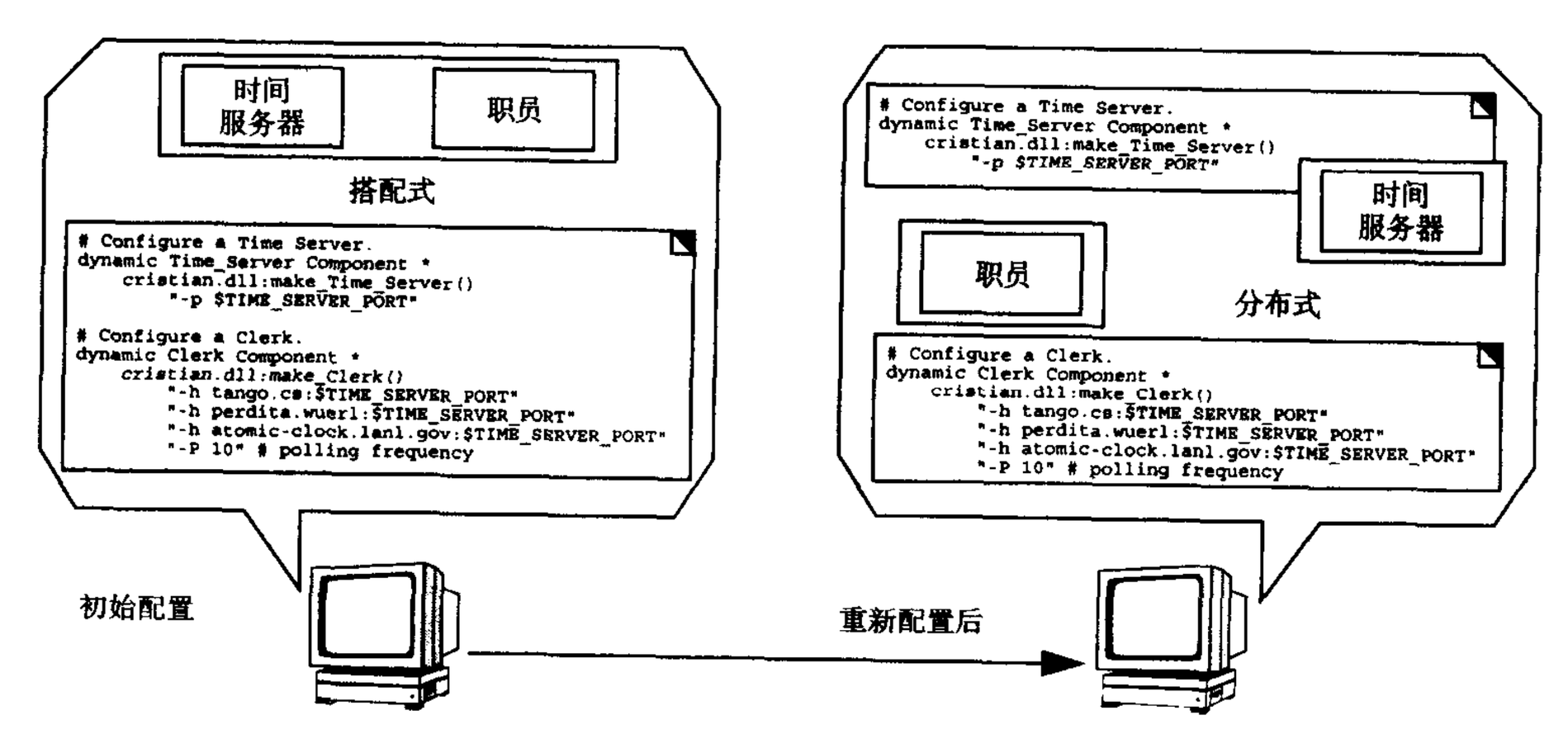


图 2-11

重新配置应用程序的组件。现在考虑一下如果我们要在分布式时间服务中改变实现组件的算法，那么会发生什么呢？例如，为了利用环境中的新特色，也许需要从Cristian算法切换到Berkeley算法。例如，如果Time_Server驻留的机器有一个WWV接收器，那么Time_Server扮演被动实体，Cristian算法也许是合适的。相反，如果Time_Server驻留的机器没有WWV接收器，那么Berkeley算法的实现也许更合适。 101

理想状况下，应该能在不影响分布式时间服务的其他组件执行的情况下，改变Time_Server算法实现。使用组件配置器模式完成这一切，只需简单地对分布式时间服务配置活动做一点小修改。

1) *修改现有的comp.conf脚本*。我们首先对comp.conf脚本做下列改动：

```
# Shut down <Time_Server>.
remove Time_Server
```

这个伪指令指示Component_Configurator关闭Time_Server组件，并且从Component_Repository中删除它，如果不再有对它的引用，那么解链cristian.dll。

2) *通知组件配置器重新解释comp.conf脚本*。接着必须指示Component_ Configurator处理被更新的comp.conf脚本。这可以通过带内触发，例如套接字链接或者CORBA操作，也可以通过带外触发，例如UNIX SIGHUP信号。无论使用哪种触发策略，Component_Configurator接收到一个重新配置事件之后，会再次咨询comp.conf脚本，并通过调用它的fini()方法关闭Time_Server组件。在这个步骤期间，其他组件的执行应不受影响。

3) *初始化重新配置*。现在可以重复步骤(1) 和(2) 来把Berkeley Time_Server组件实现重新配置进应用程序中。必须在comp.conf脚本中加入一个新的伪指令，以便从berkeley.dll DLL中动态地链接Berkeley Time_Server组件：

```
# Configure a Time Server.
dynamic Time_Server Component *
    berkeley.dll:make_Time_Server()
        "-p $TIME_SERVER_PORT"
```

最后，产生一个事件去触发进程中的Component_Configurator重读它的comp.conf脚本，并把被更新的Time_Server组件增加到Component_Repository。 102

在成功调用组件的init()方法之后，组件立即开始执行。

从例中看到，动态更换新组件实现可以容易地做到，这说明了组件配置器模式所提供的灵活性和可扩展性。特别是，当Component_Configurator删除或重新配置Time_Server组件时，不会影响应用程序中其他已配置的组件。

10. 已知使用

Windows NT的服务控制管理器（SCM）。SCM允许主SCM进程使用在实现活动（1）中所描述的基于消息的策略，自动地启动和控制管理员安装的服务组件。主SCM进程通过给系统服务组件传递各种各样的控制消息来启动和管理它们，例如PAUSE、RESUME和TERMINATE，每个服务组件必须处理这些控制消息。基于SCM的服务组件在单服务或者多服务的服务器进程内作为分离的线程运行。安装的每个服务组件负责配置它们自身，并且监视任何通信终端，

这些终端会比套接字端口更通用。例如，SCM能控制命名管道和共享内存。

现代操作系统的**设备驱动程序**。大多数现代操作系统如Solaris、Windows NT和Linux，支持动态配置的内核级设备驱动程序。可以通过钩子，例如定义在SVR4 UNIX[Rago93]中的init()、fini()和info()函数，动态地把这些驱动程序链接进和解链出系统。这些操作系统应用组件配置器模式，允许管理员在不必关闭操作系统、重新编译和静态地重新链接新驱动程序以及重新启动的情况下，重新配置操作系统内核。

Java小应用程序。小应用程序机制支持动态的下载、初始化、启动、停止和终止Java小应用程序。Web浏览器实现实际下载小应用程序并准备它们执行的基础结构软件。类java.applet.Applet提供在针对应用的子类中重载的空方法init()、start()、stop()和destroy()。因此Java使用在实现部分描述的基于继承的策略。浏览器在恰当的时间调用上面提到的四个生命期的钩子方法。它们使小应用程序有机会提供适当时可调用的客户行为。

103

例如，一旦小应用程序被加载，浏览器调用init()钩子。一旦建立完成，浏览器将调用start()钩子，小应用程序也将开始它的应用逻辑。当用户离开Web站点时，浏览器调用stop()钩子。注意，start()和stop()能被重复调用，例如当用户多次访问和离开Web站点时。一旦小应用程序被回收就调用destroy()钩子，同时释放所有资源。通过在小应用程序中创建多个线程，并且使它们作为普通Java应用程序调度，可以在小应用程序中获得更精细的生命期行为。在[JS97b]中提供了关于其他组件配置器模式如何用于Java小应用程序的例子。

dynamicTAO 映像（reflective）ORB[KRL+00]实现组件配置器的集合，此集合允许组件在分布式系统内传输、加载和卸载模块到ORB运行时系统，并监视和修改ORB配置状态。每个组件配置器负责处理dynamicTAO的某个方面的（重新）配置。例如，它的TAOConfigurator组件配置器包含的钩子，可以关联到并发和调度策略的实现，还可以关联安全和监视截取器。另外，DomainConfigurator提供把组件向/从dynamicTAO加载/卸载的普通服务。它是所有其他组件配置器派生的基类，例如TAOConfigurator。

ACE[Sch97]。自适应通信环境（ADAPTIVE Communication Environment，ACE）框架提供一套C++机制，使用在实现活动（1）中描述的基于继承的策略来动态配置和控制组件。ACE服务配置器框架[SchSu94]扩展了Inetd、Listen和SCM所提供的机制，支持通信服务组件自动的、动态的链接和解链。

ACE提供的服务配置器框架受到在现代操作系统中用于配置和控制设备驱动程序的机制和模式的影响。然而，ACE并不针对内核级设备驱动程序，而是重点解决应用程序级组件的动态配置和控制。这些ACE组件常常和反应器、接受器-连接器、主动对象模式一起实现通信服务。

104

足球比赛中，教练可以替换队员的次数有限。教练就是组件配置器，他决定替换哪个队员，同时队员体现了组件的作用。所有队员遵守同一有关替换的协议，替换动态地发生，也就是在替换期间比赛并不停止。当队员看见带他们号码的牌子挥动时，就离开球场，新的队员立即投入比赛。教练手中当前11名队员的名单对应于组件仓库。重新配置脚本并不是总由教练编写：

一些主场球迷叫喊着要求某个队员上场比赛——甚至要求教练下课。

11. 结论

组件配置器模式提供下列**好处**：

一致性。组件配置器模式强制采用一致的配置和控制接口来管理组件。这个一致性允许把组件作为积木块看待，它们可以作为组件集成到一个更大的应用中。对所有组件强制采用一个通用接口使它们的“观感”对其配置活动而言是一样的，从而利用“最小意外原则”简化了应用程序开发。

集中化管理。组件配置器模式把一个或多个组件分为一个管理单元。这种合并通过自动地执行常用的组件初始化和终止活动，例如打开/关闭文件和获取/释放锁，使开发工作得到简化。另外，模式通过确保每个组件支持相同的配置管理操作（如init()、suspend()、resume和fini()）来集中化组件管理。

模块性、可测试性和可复用性。通过从组件配置到进程的方式中分解出组件的实现，组件配置器模式提高了应用程序的模块性和可复用性。因为所有组件有一致的配置和控制接口，所以，整块应用程序能更容易被分解为可以独立开发和测试的可复用组件。这种关系的分离有利于更大规模的复用并简化后续组件的开发。 105

配置动态性和控制。组件配置器模式可以在不需要修改、重新编译或静态地重新链接现有代码的情况下，动态地重新配置组件。另外，在不重新启动组件或者和它搭配的其他主动组件的情况下，常常也可以执行组件的（重新）配置。这些特性有助于为应用程序定义的组件配置框架创建基础设施。

调整和优化。组件配置器模式通过从组件执行机制中分解出组件功能，为开发人员增加组件配置的选择范围。例如，开发人员能自适应地调整服务器并发策略，以满足客户机需求与可用的操作系统处理资源相匹配。普通的执行方案有：当客户机请求到来时产生一个线程或者进程，或者在组件创建时预先产生一个线程或者进程。

组件配置器模式有几个**不足**：

缺乏决定性和排序相关性。使用组件配置器模式难以决定或者分析应用程序的行为，直到在运行时配置它的组件。这对于某些类型的系统，尤其是实时系统，是一个问题，因为当一个动态配置的组件和某些其他的组件一起运行时，它的行为可能无法预料。例如，新近配置的组件也许消耗过多的CPU周期，因而使其他组件的处理时间不足，并导致它们超过最后时限。

减少安全性或可靠性。使用组件配置器模式的应用程序也许比同等的静态配置的应用程序更不安全或更不可靠。说它不安全，是因为冒名顶替者能够伪装成DLL中的组件。说它不可靠，是因为某个特殊的组件配置可能会对组件执行有不利的影响。例如，一个有错误的组件会破坏受损的状态信息，这些信息为它和其他配置进同一进程的组件所共享。 106

增加运行时开销和基础设施复杂性。组件配置器模式在执行组件时增加了抽象和间接的级数。例如，组件配置器首先初始化组件，接着把它们链接进组件仓库，这在对时间要求严格的

应用程序中会导致过多的开销。另外，当使用动态链接实现组件时，许多编译器在调用方法和访问全局变量时增加额外的间接级[GLDW87]。

过度窄小公共接口。组件的初始化或者终止也许会过于复杂或者与它的语境过于耦合，以至于通过常用组件控制接口（如init()和fini()）很难以一致的方式执行。

参见

组件配置器模式的目的与配置(Configuration)模式[CMP95]类似。配置模式从协议和服务自身的执行中，分解出在分布式应用程序中与配置协议及服务相关的结构问题。配置模式已经用于一些框架，支持用积木式组件构造分布式系统。

用类似的方法，组件配置器模式从组件处理中分解出组件初始化。二者主要的不同之处在于，配置模式集中于与协议和服务相关的主动组件链，而组件配置器模式集中于在传输终端之间处理交换请求的组件的动态初始化。

致谢

感谢Giorgio Angiolini为我们提供有关这种模式早期版本的反馈意见。另外，感谢服务配置器(Service Configurator)模式的原始版本的合著者Prashant Jain，原始版本是这里的组件配置器模式的文字基础。Fabio Kon写作了dynamicTAO的已知使用的部分。

107

2.3 截取器

截取器(Interceptor)体系结构模式允许透明地把服务加到框架中，并且在某些事件发生时，能自动地触发服务。

1. 例子

MiddleSoft公司正在开发一种名为MiddleORB的对象请求代理（Object Request Broker，ORB）中间件框架，此框架是代理(Broker)模式[POSA1]的一种实现。MiddleORB提供简化分布式应用程序开发的通信服务。除核心通信服务（例如连接管理和传输协议）之外，使用MiddleORB的应用程序也需要其他服务，例如事务和安全、负载平衡和容错、审核、登录、如共享内存这样的非标准通信机制，以及监视和调试工具（如图2-12）。

为满足广泛的应用程序需求，MiddleORB体系结构必须支持对这些扩展服务的集成。一个妥善处理这个需求的策略是，把尽可能多的服务集成到默认的MiddleORB配置中。然而，这个策略常常是不可行的，因为并非所有的ORB服务都能在它的开发期间预料到。随着分布式应用程序的演进，ORB框架会不可避免地扩展，包含新的特性。这样逐渐的增长会使ORB设计和维护变得十分复杂，同样也会增加它的内存覆盖，即使其中许多特性并不是在所有时间被所有的应用程序使用。

109

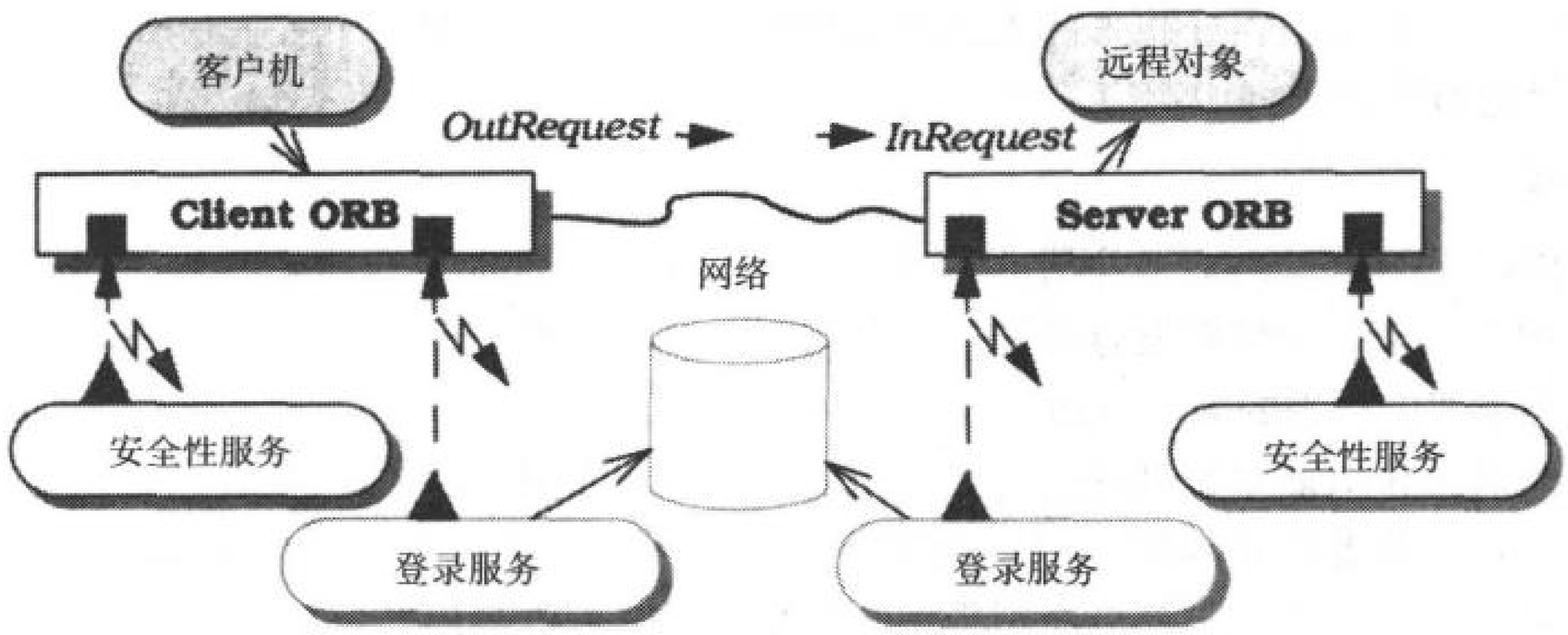


图 2-12

另一种策略是使MiddleORB框架尽可能保持简单和简洁。在这种模型中，如果应用程序开发人员需要在框架中得不到的服务，那么他们可以用自己的客户机和服务器代码实现这些服务。然而，这个策略会要求开发人员实现更多代码，而这些代码与他们的应用逻辑无关。

另外，因为有些的服务必须和核心的ORB特性紧密地交互，所以它们不能在应用程序客户机和对象级单独实现。例如，安全服务应和ORB基础结构一起集成。否则，应用程序能伪装成特权用户，并获得对受保护系统资源的未经授权的访问。

显然，上述两种策略都不能令人十分满意。如果使用第一种策略，则MiddleORB将太大，太不灵活；而如果使用第二种策略，则应用程序将变得过于复杂，并且可能不安全或者易出错。因此必须设计一种更好的策略，把针对具体应用的服务集成到MiddleORB中。

2. 语境

开发能被透明地扩展的框架。

3. 问题

框架（如ORB）、应用服务器和针对具体领域的软件架构[SG96]，不能预先知道它们必须提供给用户的所有服务。使用最初设计时并不支持的新服务来扩展某些类型的框架也会是不灵活的，尤其是黑盒框架[HJE95]。类似地，依靠应用程序自己实现所有必需的服务常常是不理想的，因为这样做破坏了复用的许多好处。因此，框架开发人员必须处理下列三种强制条件：

- 在不需修改核心架构的情况下，框架应允许集成附加服务。

➡例如，在不修改它的内部设计结构[OMG98d]的情况下，扩展MiddleORB来支持类似Kerberos或SSL[OSSL00]这样的安全服务，应该是有可能的。 ❑ 110

- 把与应用有关的服务集成到框架中，应不影响现有的框架组件，也应不需要对使用框架的现有应用程序的设计或实现做改变。

➡例如，为MiddleORB增加负载平衡，这对于现有的MiddleORB客户机和服务器应用程序影响很小。 ❑

- 使用框架的应用程序也许需要监视和控制框架的行为。

➥例如，一些应用程序也许想通过映像(Reflection)模式[POSA1]控制MiddleORB的容错策略，以指导它对错误条件的响应。 ❑

4. 解决方案

通过预定义的接口向框架注册“带外”服务，从而允许应用程序透明地扩展框架，当某些事件发生时让框架自动地触发这些服务[⊖]。另外，开放框架的实现[Kic92]，这样带外服务便能访问和控制框架的行为的某些方面。

详细地说：为由框架处理的指定的事件集，指明或者揭示一个截取器回调接口。应用程序能从这个接口派生具体截取器，以应用特定的方式实现处理这些事件并发的带外服务。为每个截取器提供一个分配器，它允许应用程序向框架注册它们的具体截取器。当指定的事件发生时，框架通知合适的分配器调用注册的具体截取器的回调。

定义语境对象以允许具体截取器内省（introspect）和控制框架的内部状态和响应事件的行为的某些方面。语境对象提供访问和修改框架内部状态的方法，因而开放它的实现。当框架分配语境对象时，能把它们传递给具体截取器。

111

5. 结构

具体框架(Concrete framework)实例化一个一般的、可扩展的体系结构，来定义由特殊系统提供的服务，例如ORB、Web服务器或者应用服务器（如图2-13）。

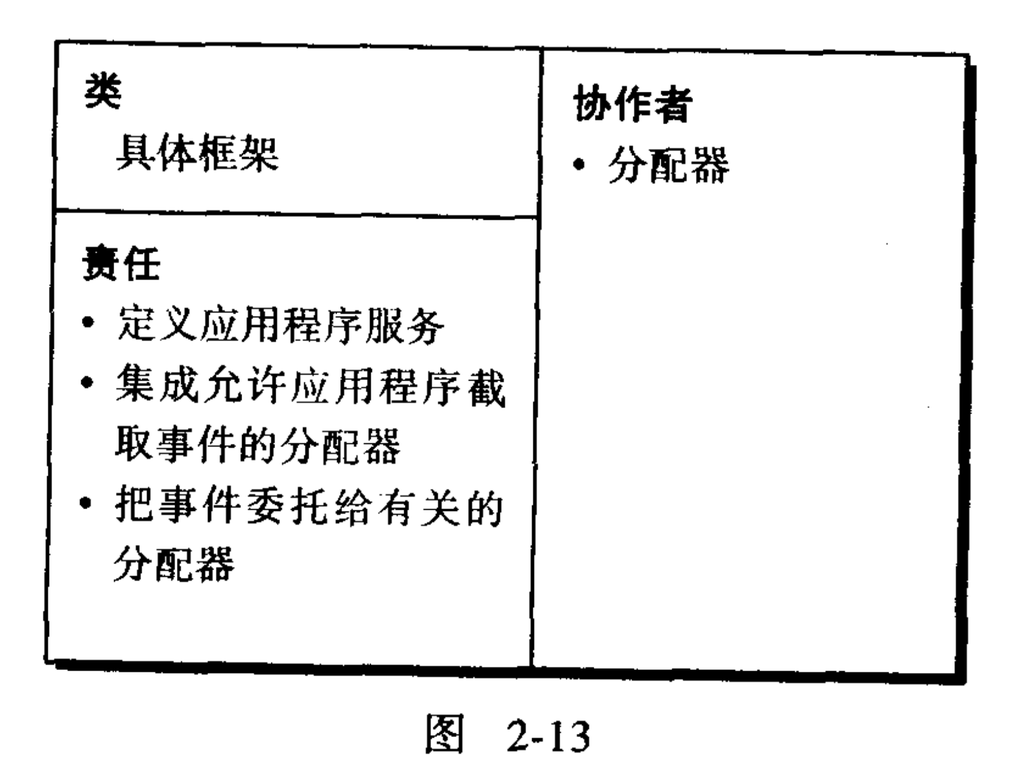


图 2-13

➥在MiddleORB中有两种类型的具体框架可用，一种用于客户机，一种用于服务器：[⊖]

- 客户机应用程序使用客户机具体ORB框架的编程接口来访问远程对象。这个具体框架提供常用服务，例如绑定到一个远程对象、发送请求给对象、等待应答和把应答返回给客户机。
- 服务器具体ORB框架提供互补的服务，包括注册和管理对象实现，在传输终点上监听，接收请求，分配这些请求给对象实现，以及返回应答给客户机。 ❑

⊖ 在这个语境中，事件意味着应用级事件，例如在ORB框架内请求和响应的发送。这些事件常常只在框架实现内可见。

⊖ ORB支持对等通信。这样“客户机”和“服务器”是相应于在特殊的请求/响应交互期间扮演的角色的相关术语，而不是特殊系统组件的基本属性。

截取器与具体框架揭示的特殊事件或者事件集相关。截取器定义钩子方法[Pree95][GHJV95]的特征描述，当相应的事件发生时，具体框架将通过一个指定的分配机制自动地调用这些钩子方法。具体截取器对截取器接口特例化并实现它们的钩子方法，以针对应用处理这些事件。 112

➥在MiddleORB例子中，我们指定一个包含多个钩子方法的截取器接口，当客户机应用程序调用远程操作和相应的服务器接收新的请求时，客户机和服务器具体ORB框架分别自动地分配这些钩子方法（如图2-14）。

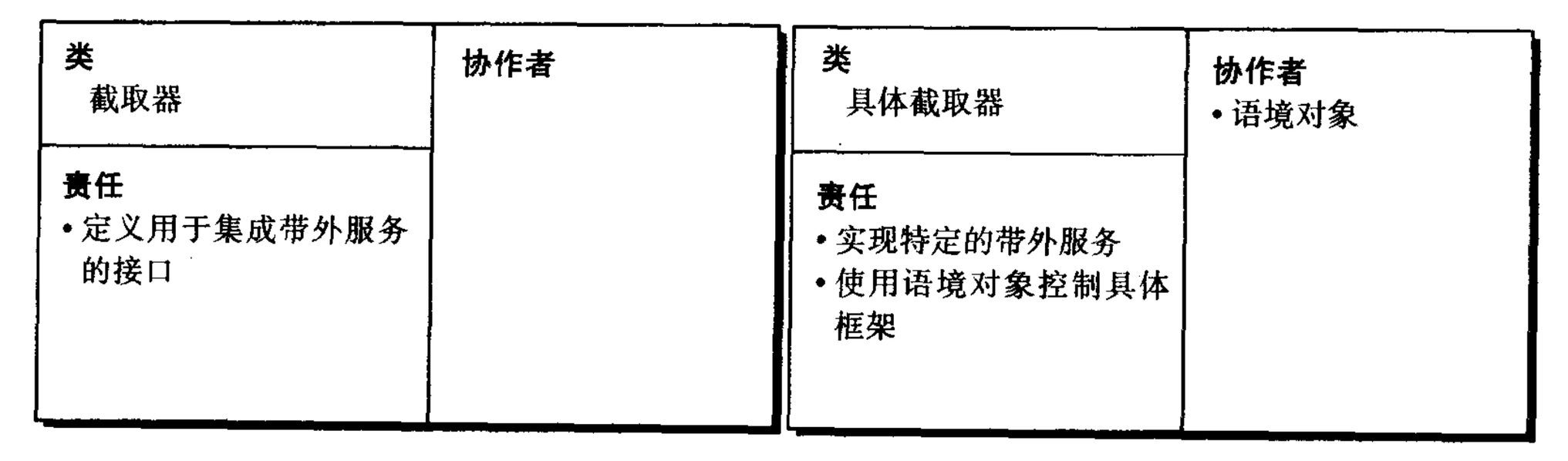


图 2-14

为了允许截取器处理特殊事件的发生，用一个具体框架定义分配器，用于配置和触发具体截取器。一般每个截取器有一个分配器。分配器定义注册和删除方法，应用程序使用它向具体框架预订和解预订具体截取器。

分配器也定义另一接口，当具体截取器注册的指定事件发生时，具体框架调用此接口。当具体框架通知分配器这样一个事件发生时，分配器调用所有向它注册的具体截取器回调。分配器在一个容器中维护它的所有已注册的截取器。

➥在我们的MiddleORB例子中，客户机具体ORB框架实现一个分配器，它允许客户机应用程序截取某些事件，例如向远程对象输出的请求和对象输入的应答。在服务器具体ORB框架中，服务器使用相应的分配器截取有关事件，例如输入的客户机请求和输出的对象应答。可以在ORB中的不同层定义其他的分配器以截取其他类型的事件，例如连接和消息传输事件。 ❑

具体截取器可以使用语境对象访问和控制具体框架的某些方面。语境对象能提供存取器方法(accessor method)从具体框架中获取信息，并提供变异器方法(mutator method)控制具体框架的行为。具体框架可以实例化语境对象，并把它和每个回调调用一起传递给具体截取器。在这种情况下，语境对象能包含与触发它的创建的事件有关的信息。 113

相反，当语境对象向分配器注册时，它能被传递给截取器。这种设计提供较少信息，但同时也导致较少开销（如图2-15）。

➥在我们的MiddleORB例子中，截取器接口定义某种方法，当客户机具体ORB框架处理一个输出请求时，它自动地分配此方法。把包含有关当前请求信息的语境对象的参数传递给这些方法。每个语境对象定义存取器和变异器方法，分别允许具体截取器查询和改变ORB状态和行为。

例如，语境对象中的存取器方法能返回远程操作的参数。使用语境对象的变异器方法，客

户机应用程序的具体截取器能给不同的对象重定向操作，这个特征能用于实现定制的负载平衡和容错服务[ZBS97]。 ❑

类 分配器	协作者 • 截取器 • 应用程序
责任 • 允许应用程序注册和移去具体截取器 • 当事件发生时分配已注册的具体截取器回调	

类 语境对象	协作者 • 具体框架
责任 • 允许服务从具体框架获取信息 • 允许服务控制具体框架的确定行为	

图 2-15

应用程序在具体框架上运行并复用它提供的服务。应用程序也能实现具体截取器并把它们向具体框架注册以处理某些事件。当这些事件发生时，它们触发具体框架及其分配器，去调用具体截取器回调，后者执行针对具体应用的事件处理（如图2-16）。

114

类 应用程序	协作者 • 分配器 • 具体截取器
责任 • 在具体框架上运行 • 实现具体截取器并把它们向分配器注册	

图 2-16

图2-17阐明截取器模式中参与者的结构。

6. 动态特性

用于截取器模式的典型场景阐明了应用程序如何实现具体截取器，并把它向相应的分配器注册。当具体框架通知它感兴趣的事件出现时，分配器接着调用截取器回调（如图2-18）：

115

- 应用程序实例化实现特定截取器接口的具体截取器。应用程序把这个具体截取器向合适的分配器注册。
- 具体框架随后接收隶属于截取的事件。在这个场景中，对于每种事件都可得到一个特殊的语境对象。因此具体框架实例化一个针对具体事件的语境对象，它包含与事件有关的信息，也包含访问和潜在地控制具体框架的功能。
- 具体框架通知合适的分配器有关事件的发生，并把语境对象作为参数传递。
- 分配器在其已注册的具体截取器的容器中迭代，并调用它们的回调钩子方法，把语境对象作为参数传递。

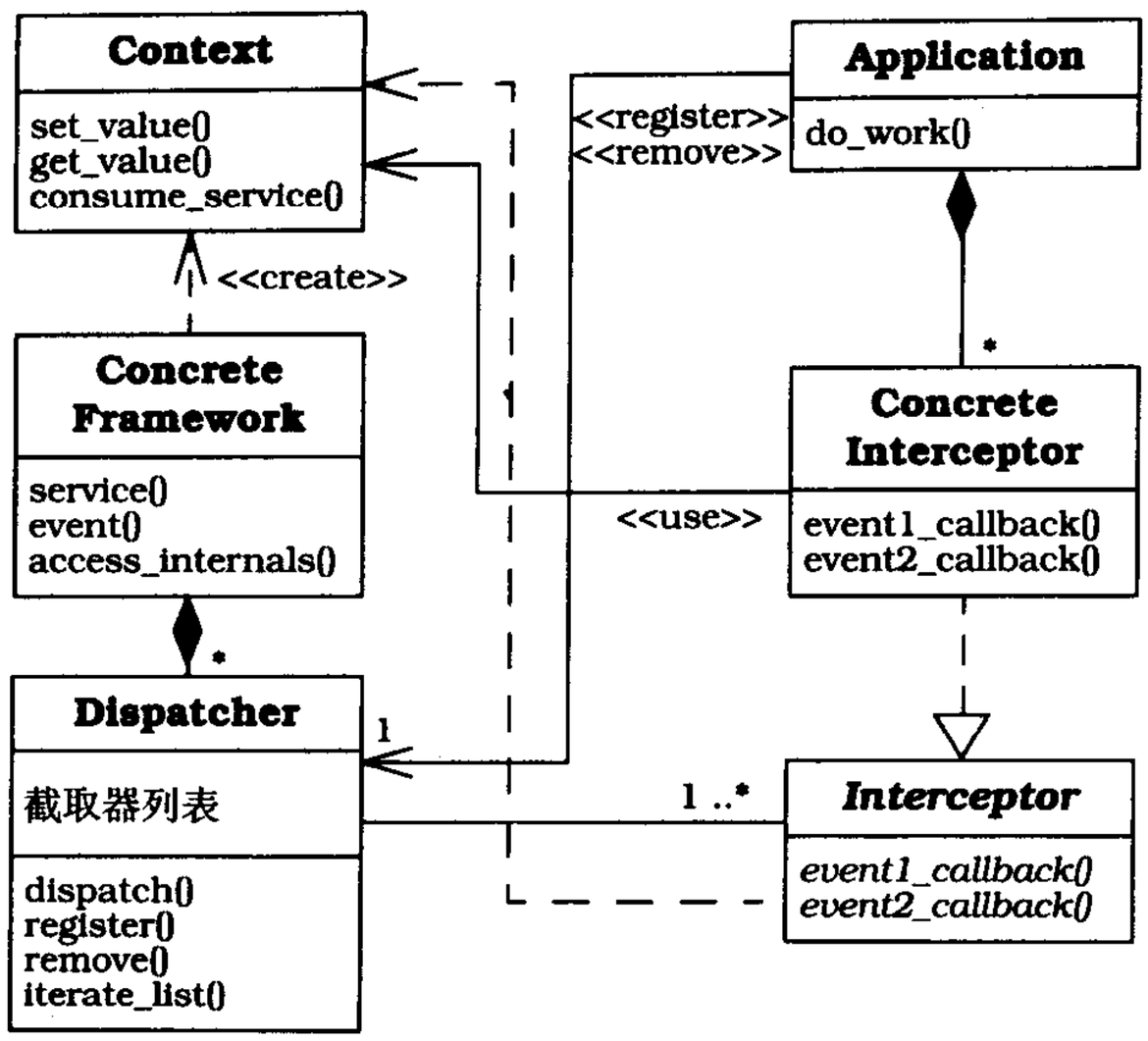


图 2-17

- 每个具体截取器能够使用它的语境对象去获取有关事件或者具体框架的信息。在处理这个信息之后，具体截取器能选择性地调用语境对象上的方法，从而控制具体框架的行为和它后续的事件处理。
- 在所有的具体截取器回调方法都返回后，具体框架继续它的正常操作。

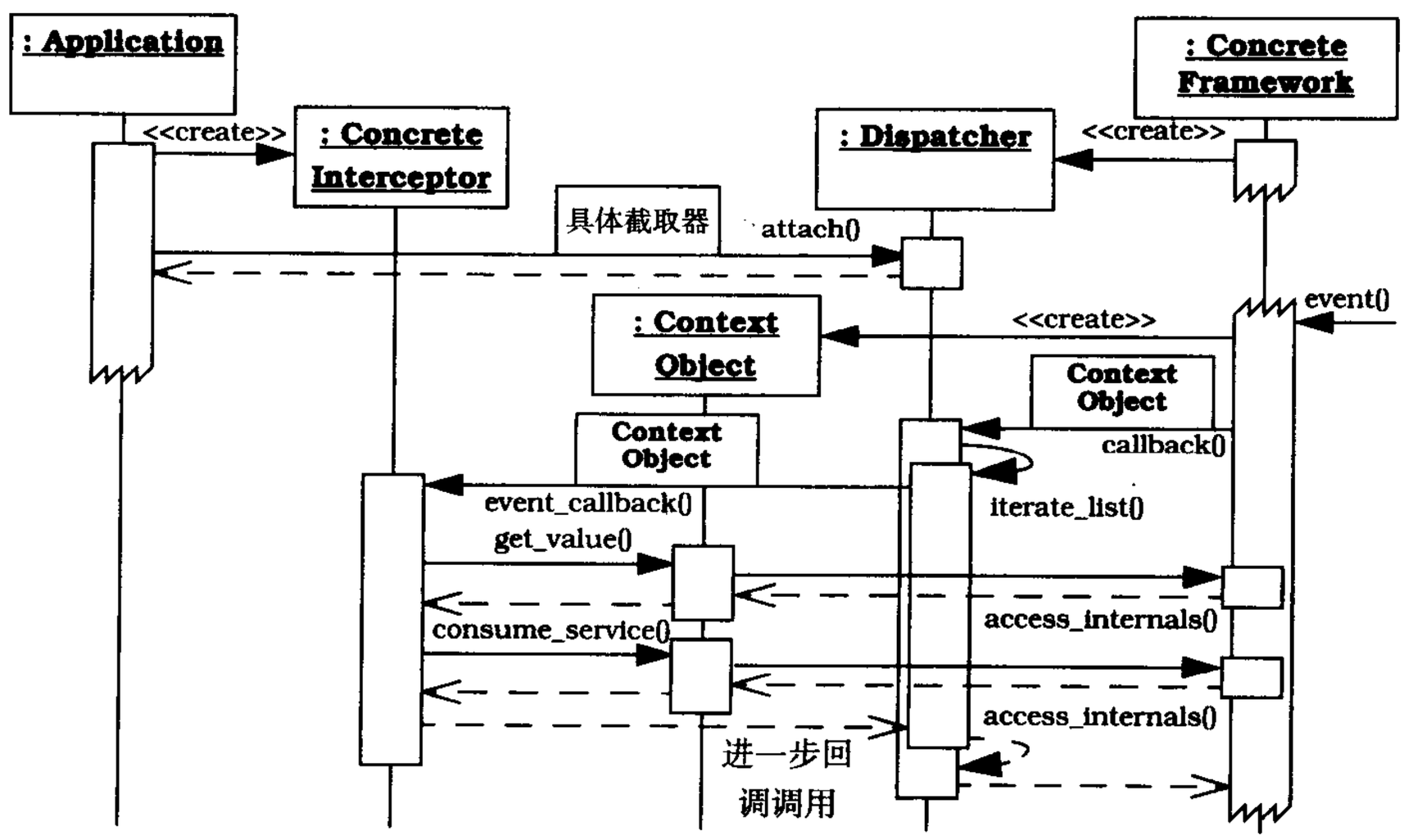


图 2-18

116

7. 实现

以下7个实现活动描述了用于实现截取器模式的一种常用的方法。

(1) 如果还没有这样的模式的话，那么使用状态机或者等价的表示法来建立具体框架内部行为的模型。这种建模不必捕获具体框架的所有抽象，但应记载与截取有关的特性。为了尽量减小任意给定的状态机的复杂性，具体框架的建模部分应能由更小的状态机构成，这些小的状态机一起形成组合的状态机。[⊖]

每个较小的状态机代表了具体框架的一个特殊方面。一旦具体框架的动态方面建模为一个状态机，那么就使用这个模型来确定何处以及何时能够截取确定事件。

在ORB中间件和许多其他的基于组件的系统中，至少存在两种类型的具体框架，一个起客户机作用，一个起服务器作用。在这种情况下，具体框架应被建模为分离的状态机。通常，状态机建模有助于标识何处放置截取器，以及如何在具体框架中定义它们的行为。

➥下面考虑一下被MiddleORB定义的客户机具体ORB框架。在ORB开始工作时，初始化这个框架以继续处理客户机请求，直到它被关闭。客户机具体ORB框架为客户提供两种类型的服务：

- 当客户机绑定给一个新的远程对象时，具体框架创建一个连接到对象的代理。
- 如果绑定操作成功的话，客户机就能发送请求给远程对象。使用预先建立的连接，每个请求被列集(marshal)和分发给远程对象。当成功分发之后，具体框架等待对象的响应消息，当响应消息到来时就散集(demarshal)它，并把结果返回给客户机，同时变迁到空闲状态。 ❑

117

附加的错误状态意味着碰到问题时的情况，例如通信错误或列集错误，如图2-19所示。注意此图只阐明了客户机具体ORB框架的内部组合状态机的一部分。

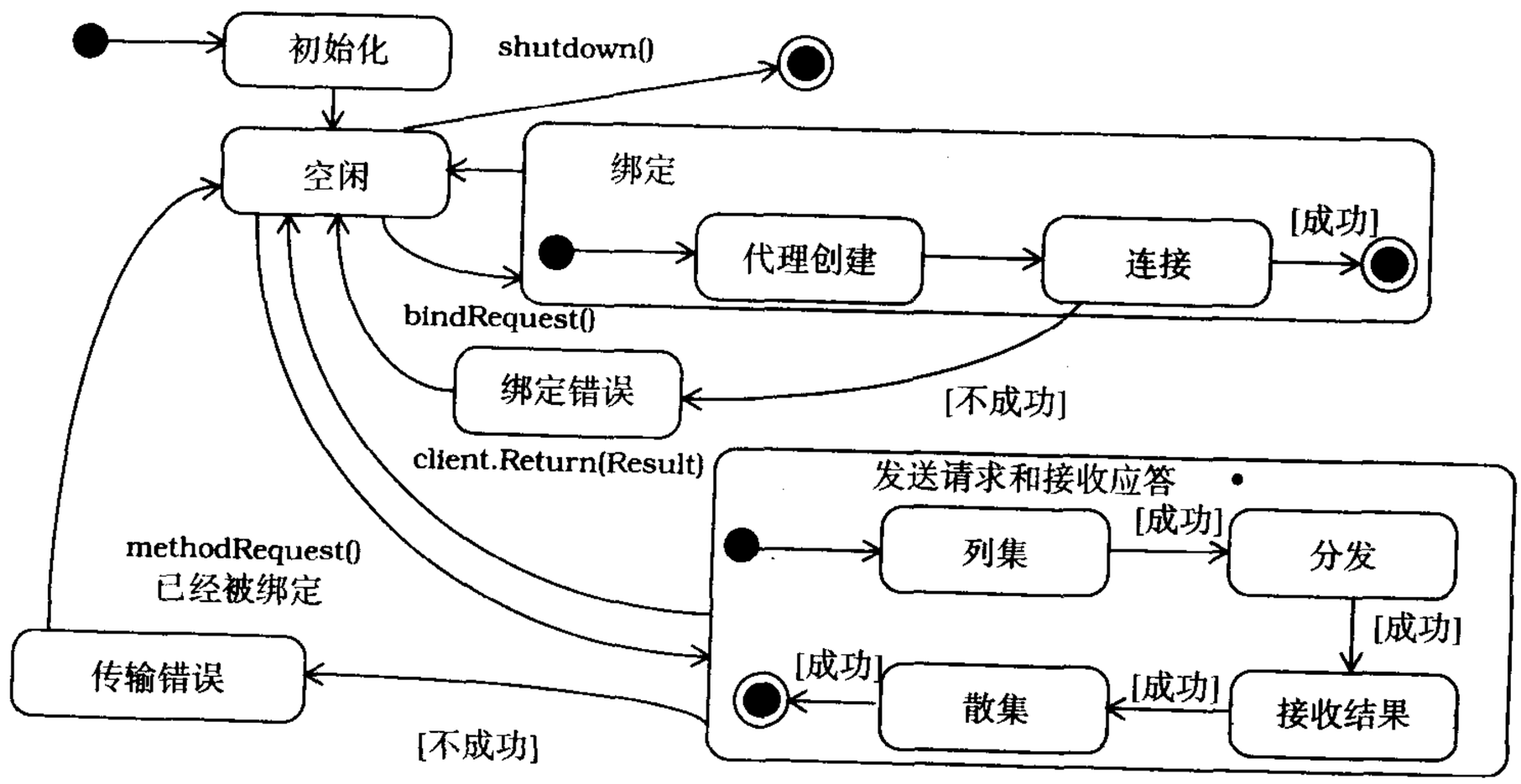


图 2-19

(2) 标识和对截取点建模。可以把这个实现活动划分为四个子活动：

(2.1) 标识具体框架状态变迁，此变迁对外部应用程序而言也许是不可见的，但它隶属于截取。例如，客户机也许想截取输出请求，这样它就能动态地增加类似登录或改变某些请求参数

⊖ 有关组合状态机的更多细节可在《UML用户指南》[BRJ98]书中获得。

这样的功能。我们把这些状态变迁称为“截取点”(interception point)。

(2.2) 把截取点划分为阅读器和记录器集。阅读器集(reader set)包括其中的应用程序仅从具体框架访问信息的所有状态变迁。相反，记录器集(writer set)包括其中的应用程序能修改具体框架行为的所有状态变迁。

(2.3) 把截取点集成到状态机模型。通过引入中间状态，可以在状态机中对截取点建模。如果状态变迁取决于截取，那么在最初变迁的源状态和汇状态之间放置一个新的截取状态。这个截取状态触发相应的截取器。对于属于记录器集的截取点，我们引入附加的状态变迁，采用如下属性： 118

- 截取状态是开始节点。
- 目标节点是代表截取之后的具体框架后续行为的状态。

许多基于组件的分布式系统定义对等的具体框架，例如客户机和服务器ORB，这些对等具体框架依据层(Layer)模式[POSA1]组织。当在某一个具体框架中标识截取点时，在位于同一逻辑层的其他对等具体框架中引入有关的截取点。例如，如果客户机ORB截取输出请求，则很可能服务器ORB也应截取输入请求。当集成分层的服务时，例如在客户机端增加安全标记并加密输出请求数据，在服务器上需要相应的截取器抽取安全标记，并解密输入数据。

➥通过使用图2-19所示的客户机具体ORB框架的状态机模型，我们能标识下表所示的潜在的截取点：

截 取 点	描 述	阅读器/记录器
Shut-down	具体框架正关闭它的操作。客户机也许需要执行一些清除工作，例如释放它们先前分配的资源	阅读器
Binding	客户机应用程序正绑定到一个远程对象。具体框架初始化一个新的代理并建立通信信道。监视服务应截取这个事件，以可视化新的客户机/对象关系	阅读器
PreMarshalOutRequest	客户机应用程序发送请求给远程对象。截取器应用于改变目标对象或参数值以支持负载平衡、确认某些前提或加密参数	阅读器+记录器
PostMarshalOutRequest	客户机具体ORB框架列集数据，但还未发送它。客户机也许对监视活动感兴趣，比如启动定时器来计算来回等待时间	阅读器
PreMarshalInReply	应答刚到达，而具体框架还未散集数据。客户机也许喜欢监视这个事件，或者停止一个来回等待时间定时器	阅读器
PostMarshalInReply	客户机具体ORB框架已列集应答。截取器会评估后置条件或者改变结果。例如，如果结果是服务器端的截取器加密的，那么它会解密它	阅读器+记录器

119

如果客户机截取异常，例如失败的连接事件，那么它也许需要额外的截取点。服务器具体ORB框架也能定义对等的截取点。 ❑

(2.4) 把截取点划分到不相交的截取组。为了处理事件，具体框架常常执行一系列相关的活动，每个活动可以和一个截取点相关联。为了强调每个活动之间的关系，把一系列语义相关的截取点合并到一个截取组(interception group)或许是有用的。

例如，与发送请求有关的所有截取点能形成一个截取组，而与接收请求有关的所有截取点能形成另一个组。这些截取组可以尽可能减少在实现活动（4）中所示的必要的截取器和分配器

的数目。

为了标识截取组，分析用于截取点的状态机，这些截取点位于状态机的同一区域并参与同一活动。例如，由起源于特定状态、结束于特定状态或者结束于特定的相邻状态的变迁触发的截取点，可以作为同一截取组的一部分候选考虑。

120

➡在MiddleORB中，PreMarshalOutRequest和PostMarshalOutRequest截取点都参与发送请求。因此这些截取点能组成OutRequest截取组。这个截取组把所有与发送请求的活动有关的事件分成一组，以使这些事件区别于其他类似InRequest、OutReply或InReply这样的截取组。 ❑

(3) *指定语境对象*。语境对象允许截取器访问和控制框架的内部状态及其对特定事件的响应行为。我们可以运用三个子活动指定语境对象。

(3.1) *确定语境对象语义*。语境对象提供有关截取点的信息，它也可以定义控制框架后续行为的服务。具体截取器针对具体应用使用信息和服务来处理截取点。为语境对象定义的存取器和变异器方法可以基于具体框架提供给截取器的信息，同时也基于框架“开放”的程度。

- 如果截取点属于阅读器集，那么要确定具体框架应为截取器处理的每个事件提供什么信息。例如，如果语境对象提供某个特别的远程操作调用的信息，那么它可以包含对调用的目标对象的引用，也包含操作的名称和参数值。
- 如果截取点属于记录器集，那么要确定如何“开放”具体框架的实现，以便具体截取器能控制选定的它的行为特性[Kic92]。例如，如果语境对象提供某个特别的远程操作调用的信息，它可以包含能修改操作的参数值的方法。当然这里要平衡两个设计强制条件，即“开放的可扩展性”与“易出错的截取代码”。

121

尽管具有开放实现的具体框架能有功能强大的截取器，但对于恶意地或偶然地破坏具体框架的健壮性和安全性的截取器而言，这些框架也是更加脆弱的。因此一些截取设计不允许在语境对象内有变异器功能。

(3.2) *确定语境对象类型的数目*。以下两种策略用于选择语境对象的数目和类型：

- *多接口*(multiple interface)。如果在具体框架中的截取点包含多种需求，那么可以为不同的截取点定义不同类型的语境对象。这个策略是灵活的，因为它允许对特殊截取点的细粒度控制。然而它增加了具体截取器的开发人员必须理解的接口的数目。
- *单接口*(single interface)。指定具有单接口的通用语境对象是可能的。使用单接口减少了语境对象接口的数目，但也会产生一个膨胀的、复杂的语境对象接口。

通常，当客户机应用程序截取多种不同框架事件时，多接口是有用的。然而在其他方面，由于单接口的一致性，它也许更为可取。

➡当MiddleORB客户机ORB框架截取输出客户机请求时，应用程序也许想访问和/或控制下列方面：

- 读取并改变目标对象引用，以实现容错或者负载平衡。
- 读取并修改参数值，以加密数据、验证选定的参数，或审慎地改变行为[POSA1]。
- 增加新的数据给请求以发送带外信息，例如安全标记或事务语境。
- 集成定制的参数列集器和散集器。

这些活动对应于在实现活动（2.3）中的表中所概述的PreMarshalOutRequest和Post-Marshal-OutRequest截取点指定的活动。因此我们引入两个对应的语境对象类型Unmarshaled- Request和MarshaledRequest。接口UnmarshaledRequest如下构成： 122

```
public interface UnmarshaledRequest {
    public String getHost (); // get host
    public void setHost (String host); // set host
    public long getPort (); // get server port
    public void setPort (long newPort); // set new port
    public String getObjName (); // get object name
    public void setObjName (String newName); // set name
    public String getMethod (); // get method name
    public void setMethod (String name); // set method
    public Enumeration getParameters ();// get parameters
    public Object getArg (long i); // get i_th arg
    public void setArg (long i, Object o); // set i_th arg
    public void addInfo (Object info); // add extra info.
    // ...
}
```

❑

(3.3) 定义如何将语境对象传递给具体截取器。语境对象由具体框架实例化，并且通过下列两种策略之一传递给具体截取器：

- 按注册(per-registration)。在这种策略中，当语境对象向分配器注册时，向截取器传递一次语境对象。
- 按事件(per-event)。在这种策略中，每次回调调用时语境对象一起被传递给具体截取器。

“按事件”策略允许具体框架提供有关特定事件发生的细粒度信息。相反，“按注册”策略仅提供特定事件类型的所有发生所共有的一般信息。然而，由于语境对象的重复创建和删除，“按事件”策略会招致更大的开销。

(4) 指定截取器。截取器定义通用的接口，当截取点被触发时，具体框架通过分配器使用该接口调用具体截取器。为每个在实现活动（2.4）中标识的截取组定义一个截取器。因此每个从特殊的截取器派生的具体截取器负责处理一个指定截取组的所有截取点。

对于同一个截取组中的每个截取点，截取器定义一个指定的回调钩子方法。因此在截取点和截取器钩子方法之间存在一对一的关系。通常截取器对应于观察者模式[GoF95]中的观察者(Observer)，在那里它的回调钩子方法扮演针对具体事件的更新方法的角色。如果应用在实现活动（3）中描述的“按事件”语境对象策略，那么可以把语境对象作为参数传递给具体截取器回调钩子方法。依据在实现活动（6）中描述的策略，这些方法能返回结果或者引发异常。 123

➡在实现活动（2.4）中，我们标识截取组OutRequest。下面举例说明这个截取组的一个常用的截取器接口：

```
public interface ClientRequestInterceptor {
    public void onPreMarshalRequest
        (UnmarshaledRequest context);
    public void onPostMarshalRequest
        (MarshaledRequest context);
}
```

对于每个与OutRequest截取组相关联的截取点，ClientRequestInterceptor定义了一

个分离的钩子方法，分配器在合适的截取点回调此方法。 ❑

(5) 指定分配器。为每个截取器定义一个分配器接口，应用程序可以使用该接口向具体框架注册和删除具体截取器。另外，框架使用这个接口分配在截取点上注册的具体截取器。这包括两个子活动：

(5.1) 指定截取器注册接口。分配器在观察者模式[GoF95]中扮演主题(subject)角色。它为扮演观察者角色的截取器实现一个注册接口。应用程序传递一个对具体截取器的引用给注册方法，此方法依据管理者(Manager)模式[Som97]在容器中存储引用。

为了实现不同的回调策略，应用程序可以传给分配器附加的参数。例如，正如在实现活动（6）中所描述的，当为同一截取点注册多个截取器时，它可以传递一个决定调用次序的优先值。分配器给应用程序返回一个惟一标识已注册的截取器的密钥。当应用程序删除它先前注册的截取器时，它把这个密钥传递给分配器。

124

为了使截取器注册自动化，并隐蔽它的实现，具体框架可以实现帮助者(helper)类，此类提供截取器接口的“空操作”实现。这些类的构造函数自动地向具体框架注册实例。应用程序从合适的帮助者类派生具体截取器实现，重载它的方法，并且调用基类构造函数隐式地注册它们的截取器。

通常，指定的分配器能将相应事件类型的每次出现，从具体框架转发给已注册的处理这些事件的具体截取器。因此常常使用单件(Singleton)模式[GoF95]实现分配器。

➡在下列ClientRequestDispatcher类中定义的方法，允许应用程序向MiddleORB具体框架注册和删除ClientRequestInterceptor实例：

```
public class ClientRequestDispatcher {
    // Interceptors are stored in a Java vector and called
    // in FIFO order.
    Vector interceptors_;

    synchronized public void
    registerClientRequestInterceptor
        (ClientRequestInterceptor i) {
        interceptors_.addElement (i); // Add interceptor.
    }

    synchronized public void
    removeClientRequestInterceptor
        (ClientRequestInterceptor i) {
        // Remove interceptor.
        interceptors_.removeElement (i);
    }
    // ...
}
```

❑

(5.2) 指定分配器回调接口。当一个截取事件发生时，具体框架通知它的分配器。当分配器得到通知后，它就调用其已注册的具体截取器的相应钩子方法。对于相关联的截取器提供给分配器的具体框架，分配器常常提供同一个接口。

125

这种相似性有两种理由：

• 通过允许分配器在不传输任何参数的情况下，把事件通知有效地委托给它的注册截取器，

使性能更具效率。

- 如果分配器的公共接口改变了，它能局部化和最小化所需的修改。这种修改可能是增加新的截取点到与分配器回调接口相关联的截取组。在这种情况下，会增加一个额外的钩子方法给回调接口。

➡在MiddleORB中，内部分配器ClientRequestDispatcher也实现接口ClientRequestInterceptor:

```
public class ClientRequestDispatcher
    implements ClientRequestInterceptor { /* ... */ }
```

因此MiddleORB客户机具体ORB框架可以使用这个接口中的回调钩子方法，用它通知分配器有关与客户机请求相关的所有事件。 ❑

(6) *在具体框架中实现回调机制*。当截取事件出现时，具体框架通知相应的分配器。接着分配器依次调用所有被注册的具体截取器回调的钩子方法。因此，需要一种从具体框架向它的分配器和从分配器向被注册的截取器传播事件的机制。可以通过两次运用观察者模式[GoF95]实现这种机制。

观察者模式的第一个应用出现在具体框架到达截取点的时候。此时，它创建合适的语境对象，并通知分配器有关事件的出现，依据观察者模式的术语，具体框架是*主题*，并且被一个分配器观察。

当具体框架通知分配器时，它可以把语境对象作为参数传递，或者使用预先分配的单件语境对象，此对象作为对具体框架的接口。在第一种策略中，所有事件相关的信息被封装在语境对象中，而第二种策略需要具体框架来存储所有的必要信息。策略的选择依赖于实现活动（3.3）中所描述的具体框架的设计。 126

观察者模式的第二个应用出现在分配器得到通知之后。在这一点上，它在所有已在这个截取点注册的截取器上迭代，并且调用它们接口中合适的回调方法，把语境对象作为参数传递。因此分配器也是被具体截取器观察的*主题*。

分配器的内部回调机制可以用迭代器(Iterator)模式[GoF95]实现。类似地，分配器可以运用策略（Strategy）模式[GoF95]，允许应用程序在几个截取器回调次序之间做出选择。

- 简单的调用策略包含“先进先出”（FIFO）或“后进先出”（LIFO）排序策略，分别以截取器被注册的次序调用它们或者以相反次序调用它们。当使用截取器模式实现一个特殊的“截取器栈”时，可以使用一个组合的FIFO/LIFO方法来处理遍历栈的消息。在客户机上，可以使用FIFO策略把消息往栈下传递。服务器可以使用LIFO策略从栈中得到消息。
- 更复杂的排序回调策略以优先级次序分配具体截取器。在这种策略中，当向分配器注册一个具体截取器时，它传递一个优先参数。当传播一个事件时，分配器以更高优先权第一的原则调用截取器。
- 另一种复杂的回调策略是基于责任链(Chain of Responsibility)模式[GoF95]。如果具体截取器能处理它的分配器分发的事件，那么它返回相应的结果；否则它返回一个特殊值或者引发异常，以指示它对截取事件不感兴趣。在这种情况下，回调分配机制要求链中的下一个

截取器处理事件。当某一个截取器处理完事件之后，该过程停止。

如果截取器碰到阻止它成功完成工作的错误条件，那么它可以调用异常或者返回错误值，把这些错误传递给处理器。在这种情况下，必须准备具体框架处理这些错误。

127

➡当客户机具体ORB框架处理请求时，它实例化一个语境对象，并且通知相应的分配器在被注册的截取器上迭代，以调用合适的事件处理钩子方法，例如OnPreMarshalRequest()。

```
public class ClientRequestDispatcher {
    // ...
    public void
    dispatchClientRequestInterceptorPreMarshal
        (UnmarshaledRequest context) {
        Vector interceptors;
        synchronized (this) { // Clone vector.
            interceptors = (Vector)
                interceptors.clone ();
        }
        for (int i = 0; i < interceptors.size (); ++i) {
            ClientRequestInterceptor ic =
                (ClientRequestInterceptor)
                interceptors.elementAt (i);
            // Dispatch callback hook method.
            ic.onPreMarshalRequest (context);
        }
    }
    // ...
}
```

❑

(7) 实现具体截取器。具体截取器可以从截取器派生，并且针对具体应用实现相应的截取器接口。具体截取器可以使用它接收的语境对象作为参数来

- 获取关于发生的事件的附加信息。
- 控制具体框架的后续行为，正如在实现活动（3）中所描述的。

可应用扩展接口模式使应用程序中不同截取器类型的数目最少。每个截取器接口变成单一截取器对象的扩展接口。因此同一“物理”对象能用于实现不同的“逻辑”截取器。

128

➡客户机应用程序可以提供它自己的ClientRequestInterceptor类。

```
public class Client {
    static final void main (String args[]) {
        ClientRequestInterceptor myInterceptor =
            // Use an anonymous inner class.
            new ClientRequestInterceptor () {
            public void onPreMarshalRequest
                (UnmarshaledRequest context) {
                System.out.println
                    (context.getObj () + " called");
                // ...
            }
            public void onPostMarshalRequest
                (MarshaledRequest context) { /* ... */ }
        };
        ClientRequestDispatcher.theInstance ().
            registerClientRequestInterceptor
                (myInterceptor);
        // Do normal work.
    }
}
```

在这个实现中，客户机的main()方法创建一个匿名的ClientRequestInterceptor内部类的实例，并把它向ClientRequestDispatcher类的单件实例注册。无论何时客户机具体ORB框架碰到一个客户机请求事件时，它就通知分配器，接着分配器回调被注册的截取器的合适的钩子方法。本例中，一个方法被调用之后且在被列集之前截取器在屏幕上打印一条消息。 ❑

8. 已解决的例子

应用程序可以使用截取器模式把定制的负载平衡机制集成到MiddleORB。由于使用了截取器，这种机制对于客户机应用程序、服务器应用程序和ORB基础结构自身而言是透明的。在这个例子中，客户机应用程序插入了一对具体截取器：

- 绑定截取器(Bind Interceptor)。当一个客户机绑定到远程对象时，绑定截取器确定在CORBA对象上的后续调用是否应被负载平衡。在预定义的服务器机器上都能自动地复制所有类似“负载平衡”的对象 [GS97]。可以在ORB的实现仓库中维护有关负载平衡、服务器和可用的被复制对象的信息[Hen98]，并且在驻留内存表中缓存它们。有关当前系统负载的信息能够驻留在分离的表中。 129
- 客户机请求截取器(Client Request Interceptor)。当一个客户机在远程对象上调用操作时，就分配客户机请求具体截取器。这个截取器检查对象是否被复制。如果它被复制的话，那么截取器就发现了一台具有较轻负载的服务器机器，并把请求转发给合适的目标对象。可以使用策略(Strategy)模式[GoF95]配置用于测量当前负载的算法。这样客户机开发人员可以在不影响ORB基础设施或者客户机/服务器应用逻辑的情况下，透明地替代他们自己的算法。

图2-20举例说明在绑定截取器复制一个对象并放在多个服务器上以平衡负载之后，客户机请求截取器执行的场景。

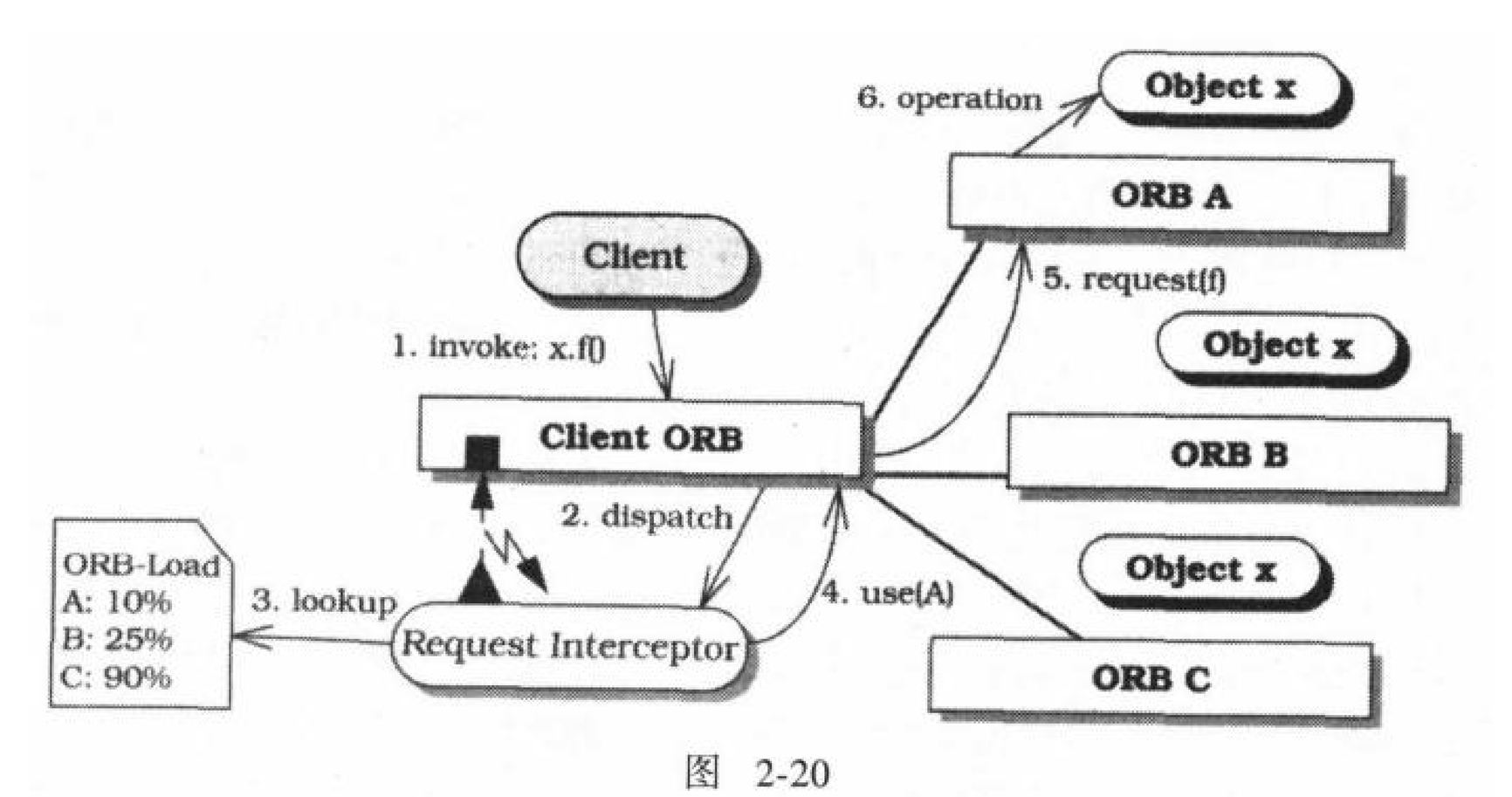


图 2-20

这个场景包括以下三个步骤：

- 客户机在被复制的对象上调用操作（1）。

130

- 客户机请求截取器截取这个请求（2）。接着它查询包含对象副本的表，以标识具有最轻负载的服务器（3）。在对象被复制之前，绑定截取器已经创建了这个表。
- 客户机ORB转发请求给具有轻负载的服务器（4）。接着服务器的ORB把它发送给驻留在这个服务器上的对象实现（5），并且分配它的操作（6）。

9. 变体

截取器代理(Interceptor Proxy)变体，也称为委托器(Delegator)。常常在分布式系统的服务器端使用这个变体来截取远程操作。服务器具体框架自动地实例化驻留在服务器上的本地对象实现的代理[POSA1]。这个代理实现和对象相同的接口。当代理被实例化时，它接收一个对实际服务器对象的引用。

当客户机发出一个请求时，服务器的代理截取输入的请求，并且执行一些预处理功能，例如启动一个新的事务或者确认一个安全标记。接着代理把请求转发给本地服务器对象，此对象在代理建立的语境中执行它的处理操作（如图2-21）。

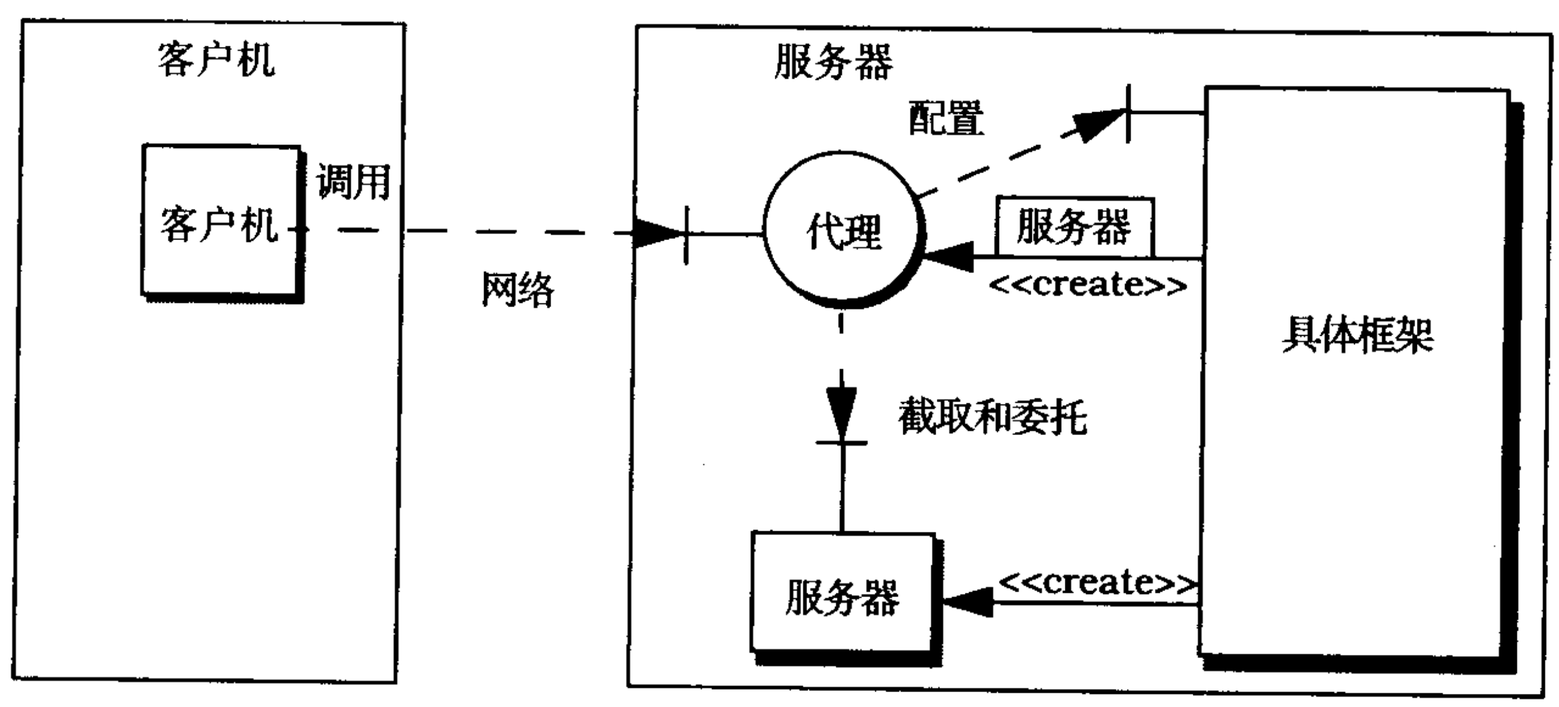


图 2-21

在完成对象处理之后，代理执行所需的任何后处理，如果有结果的话，就把结果返回给客户机。客户机和服务器对象都感觉不到截取器代理的存在。

每个分配器一个截取器(Single Interceptor-per-Dispatcher)。这种变体允许仅有一个截取器向指定的分配器注册。当拥有多个截取器毫无意义时，这个限制能够简化模式的实现，在这种情况下，不需要为具体框架保留所有截取器构成的集合。

131

➡在MiddleORB中，可以有一个截取器接口用于动态地改变具体框架的传输协议[Naka00]。最多只应有一个改变具体框架默认行为的截取器。因此，没有理由注册一个不同截取器的链，在这些链中的每个截取器都负责改变传输协议。 ❑

截取器工厂(Interceptor Factory)。当具体框架多次实例化同一类，并且类的每个实例都受到截取时，可以使用这个变体。应用程序向具体框架注册截取器工厂，而不是显式地为每个对象向分配器注册截取器。因此，对于具体框架所实例化的每一对象，它也使用提供的工厂实例化一个具体截取器。

➡在MiddleORB中，对于服务器具体ORB框架创建的每个对象实现，可以有一个不同的截取器。

另外，客户机具体ORB框架可以使用一个工厂为每个代理来实例化一个单独的客户机截取器。 ❑

隐式的截取器注册(Implicit Interceptor Registration)。与通过分配器显式地注册截取器不一样，具体框架能够动态地加载截取器。存在两种实现这个策略的方法：

- 具体框架在预先定义的位置中搜索截取器库。接着它把这些库加载到具体框架，并确保在安装和分配事件给它们之前，它们支持所需的截取器接口。
- 具体框架可以使用运行时配置机制（如组件配置器模式定义的）来动态链接截取器。在这种设计中，具体框架内的组件配置器组件解释脚本，此脚本指定链接哪个截取器，在哪里发现包含这些截取器的动态链接库（DLL），和如何初始化它们。接着组件配置器链接指定的DLL，并向具体框架注册包含在它们中的截取器。

10. 已知使用

基于组件的应用服务器。此服务器用于服务器端组件，例如EJB[MaHa99]、CORBA组件[OMG99a]或者COM+[Box97]，并且实现截取器代理变体。为帮助开发人员只关注他们的与应用有关的业务逻辑，引入特殊的具体框架——在此称为“容器”——将组件和与系统有关的运行环境隔离。组件不必实现它们所有的基础框架服务，例如事务、安全性或持久性，而是使用与配置有关的属性声明它们的需求。图2-22阐明这个容器的体系结构： 132

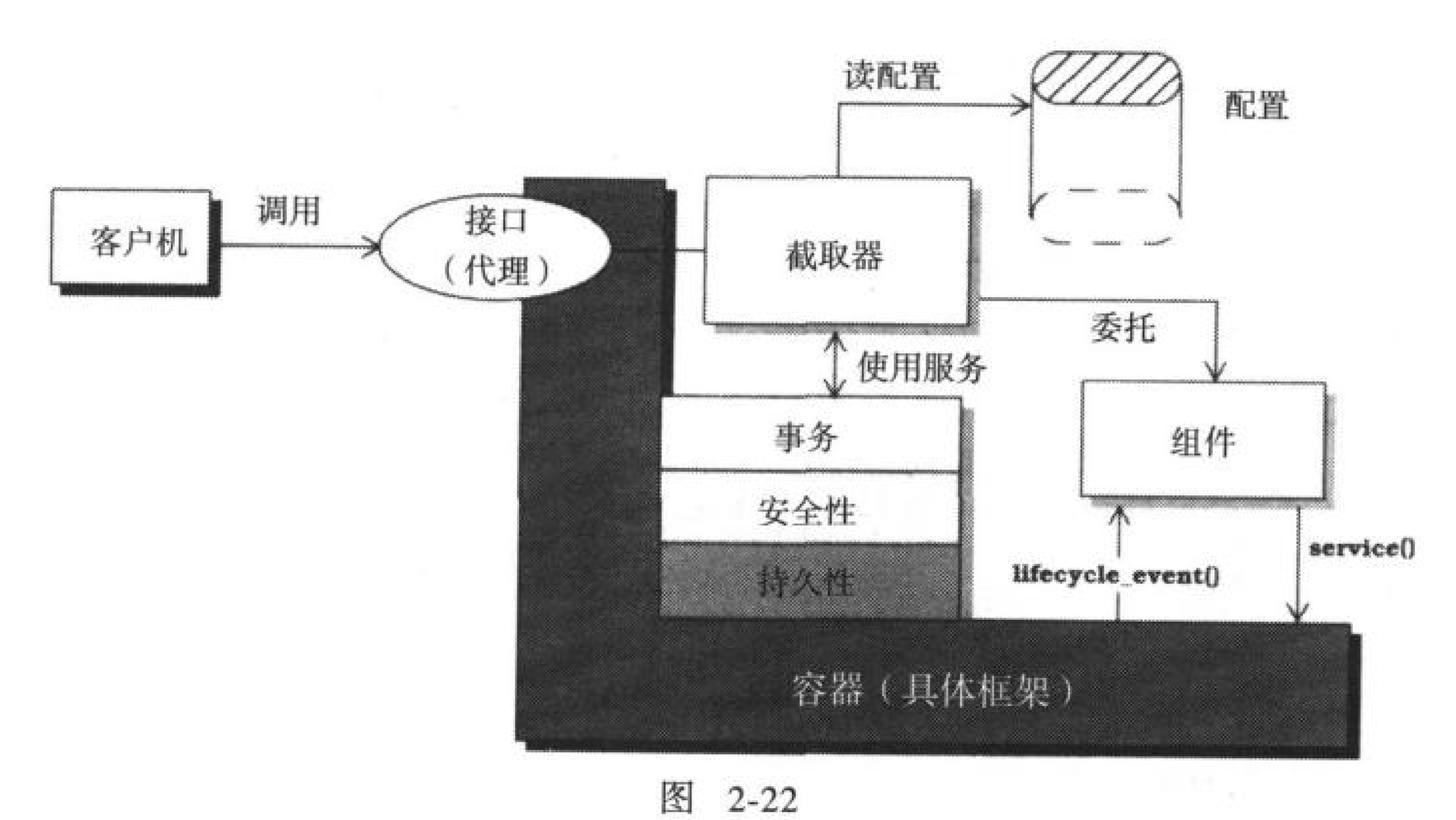


图 2-22

在实例化一个新的组件之后，具体框架还实例化一个截取器代理，并把它和那个组件关联起来，例如可以在它的初始化期间通过向代理提供一个组件引用来实现。在任何客户机请求到来后，代理检查组件的与配置有关的属性，并执行它期望的服务，例如初始化新的事务。

应用服务器常常使用标准截取器模式的一个实例，向组件通知有关生命期事件，例如连接初始化和终止、组件激活和钝化或者与事务有关的事件。

CORBA实现[OMG98c]。例如TAO[SLM98]和Orbix[Bak97]运用截取器模式，以便应用程序开发人员能集成附加的服务来处理指定类型的事件。截取器通过从发送和接收请求以及应答所需要的传统ORB通信机制中分离出请求处理，增强ORB的灵活性。 133

例如，Orbix定义过滤器概念，此过滤器基于“灵活绑定”[Shap93]的思想。通过从预定义的基类中派生，开发人员能截取事件。常见的事件包括客户机启动的传输和远程操作的到达，以及对象实现启动的传输和应答的到达。开发人员可以选择在列集之前或者之后是否截取请求或结果。Orbix编程人员可以使用同一过滤机制来建立多线程服务器[SV96a][SV96b][SV96c]。其他的ORB，例如Visibroker，实现截取器模式的截取器工厂变体。

CORBA可移植截取器规范[OMG99f]。OMG引入它来标准化对符合CORBA实现的截取器的用法。可移植截取器和客户机与服务器之间的通信紧密相连。因此在它们被交换时，它们能够影响CORBA请求和应答的内容，下面是两个这样的例子：

- 客户机端安全截取器在离开客户机进程之前，能透明地给请求增加授权信息。在接收服务器中与之匹配的服务器端安全截取器就能够验证如下事实：在请求被分配之前，客户机被授权在目标对象上调用请求。
- 事务截取器是可移植截取器的另一例子。在事务截取器离开客户机之前，它在请求中增加事务ID。接着相应的服务器端事务截取器确保将请求分配给特定事务语境中的目标对象。

容错ORB框架。已经在大量的容错ORB框架中应用截取器模式，例如永久系统(Eternal System)[NMM99][MMN99]和CORBA容错规范[OMG99g]。永久系统截取客户机通过更低层I/O子系统进行系统调用，并把这些系统调用映射给可靠的多播子系统。永久系统并不修改ORB或者CORBA语言映射，从而确保应用程序容错的透明性。

134

AQuA框架[CRSS+98]也提供一个截取器模式的变体。AQuA入口扮演CORBA对象和总体组通信子系统之间的媒介，并把GIOP消息翻译为组通信原语。AQuA使用质量对象（Quality Object，QuO）[ZBS97]框架以允许应用程序指定它们的可靠性需求。

COM[Box97][HS99a]。编程人员可以使用截取器模式在它们的组件中实现标准接口IMarshal。IMarshal提供定制的列集功能而不是标准列集，这样做有很多原因。例如，使用定制的列集能有效地发送复杂的数据，例如跨网络的图结构。

当COM运行时系统（run-time system）从一个组件向驻留在另一执行环境的客户机传输一个接口指针时，它查询相应的组件以找出截取器接口IMarshal的一个实现。如果组件确实实现了IMarshal，那么COM运行时系统使用这个截取器接口的方法向组件请求指定的信息，以允许它将数据表面化为流对象。

Web浏览器。Web浏览器实现截取器模式以帮助第三方卖方和用户集成它们自己的工具和插件。例如，Netscape Communicator和Internet Explorer允许浏览器为处理指定的媒质类型注册插件。当来自于Web服务器的媒质流到达时，浏览器抽取里面的类型。如果浏览器本来并不支持这种类型，那么它检查是否注册了一个用于此类型的插件。接着浏览器自动地调用合适的插件来处理数据。

dynamicTAO reflective ORB[KRL+00]支持用于监视和安全的截取器。使用组件配置器动态地把特殊的截取器实现加载到dynamicTAO。使用组件配置器在dynamicTAO中安装截取器，允许应用程序在运行时交换监视和安全策略。

135

地址改变的普通邮件转发。当人们从一个住处移到另一个住处时，就出现截取器模式的现实生活中的例子。此模式可以指导邮局截取寄向最初住处的普通邮件，并把它透明地转发给新

的住处。在这种情况下，邮件的内容没有被修改，仅仅改变了目标地址。

11. 结论

截取器模式提供下列**优点**：

可扩展性和灵活性。通过定制和配置截取器和分配器接口，具体框架的用户能够在不改变具体框架体系结构或者实现的情况下，增加、修改和删除服务。

事务分离。在不影响现有应用程序代码的情况下，可以透明地增加截取器，因为截取器被从应用程序行为中分解出来。截取器可以看做是织入应用程序的一个*方面*(aspect)[KLM+97]，这样开发人员便能集中注意力于应用逻辑而不是在基础设施服务上。截取器模式也有助于将负责开发和配置应用逻辑的编程人员与编写截取器代码的编程人员分开。

支持对框架的监视和控制。截取器和语境对象有助于从具体框架中动态地获取信息，也有助于控制它的行为。这些能力有助于开发人员建立管理工具、调试器和其他的高级服务，例如负载平衡和容错。

当客户机调用远程操作时，可以自动通知截取器。通过使用语境对象，截取器能够将在一个方法请求中指定的目标对象由最初的目标改变为提供被请求服务的另一个服务器。服务器的选择可以依赖于各种动态因素，例如当前的服务器负载或可用性。如果一个框架不能成功地完成一个请求，那么可以激活另一个截取器来重新发送请求给提供相同服务的备份服务器，从而通过复制[OMG99a]改进容错性能。

层对称性。为实现分层的服务，开发人员可以为具体框架所提示的相关事件引入对称的截取器。例如，在一个CORBA环境中，开发人员可以编写客户机端截取器，以创建安全标记并自动地把这些标记添加到输出请求中。同样，他们也可以编写对称的服务器端截取器，在转发输入请求给实际对象实现之前抽取这些标记。 136

可复用性。通过从其他应用程序代码中分离出截取器代码，可以跨应用地复用截取器。例如，用于把信息写入日志文件的截取器可以在需要相同类型的登录功能的其他应用程序中被复用。

截取器模式也有下列**不足**：

使设计问题复杂化。对使用特定具体框架的应用程序的需求预期并不容易，从而难以确定提供哪个截取器分配器。通常，提供低效的分配器会降低具体框架的灵活性和可扩展性。相反，提供太多分配器会导致庞大的、低效的系统，此系统难以实现、使用和优化。

当具体框架定义许多不同的截取器接口和分配器时，也会出现类似的问题。在这种情况下，截取器实现必须处理所有这些异构的可扩展性机制。如果存在太多不同的机制，那么难以学习和使用它们。相反，仅提供一个通用的截取器和一个通用的分配器会导致接口的膨胀或者复杂的方法型构。通常，如果没有关于常见应用程序用法的知识，找到正确的平衡是困难的。

恶意的或易出错的截取器。如果一个具体框架调用截取器，而截取器未能返回，那么整个应用程序会阻塞。为了防止阻塞，具体框架可以使用可配置的超时值。如果截取器在一个指定的时间之后没有返回控制，那么一个独立的线程能中断截取器的执行。然而这种方法会使具体框架的设计复杂化。

例如，也许需要复杂的功能使具体框架从超时中恢复而不泄漏资源或破坏重要数据结构。截取器也可以执行非预期的活动或者产生运行时错误。防止这些问题是很困难的，因为具体框架和截取器通常在同一地址空间运行。

137

潜在的截取级联。如果截取器应用语境对象以改变具体框架的行为，那么它也许会触发新的事件，从而启动底层状态机中的状态转移。这些状态转移会导致具体框架调用一连串的触发新事件的截取器等。截取级联会导致严重的性能瓶颈或者死锁。具体框架提供的截取器分配器越多，截取级联的风险越大。

参见

模板方法(Template Method)模式[GoF95]为算法定义了一个构架，称之为“模板方法”，可以改变在算法中的不同步骤。这些变体的执行被委托给钩子方法，钩子方法可以在客户机提供的子类中被重载。因此模板方法可以看做是一个轻量级的具体框架，同样把钩子方法看做是轻量级的截取器。模板方法模式可用于特定的抽象层局部地应用截取，而截取器模式把截取提升为基础的设计特性，此特性跨越框架体系结构中的多个层次。

责任链模式[GoF95]定义不同的处理程序，可以在请求的发送方和接收方之间插入这些处理程序。和使用截取器模式一样，这些处理程序能用于在发送者和接收者之间集成附加的服务。然而在责任链模式中，转发请求，直到某一个中间处理器处理该请求。与此相反，截取器模式中的分配器常常把事件转发给已为它注册的所有具体截取器。

因此，为仿效截取器模式，责任链中的每个中间处理程序必须处理并且转发请求。然而截取器和责任链在其他两个方面有所不同。正如模式的名字所暗示的，责任链中的事件处理程序是链接在一起的。与此相反，框架中的具体截取器不必链接在一起，而是在一个分层的体系结构[POSA1]的多种抽象级别上被关联。责任链中的事件处理程序也不能控制其他事件处理程序或应用程序组件的后续行为。相反，截取器模式的一个关键特性是当一个指定的事件发生时，它能控制具体框架后续行为。

138

管道和过滤器模式[POSA1]为处理数据流定义了一个体系结构，在此体系结构中每个处理步骤被封装在一个过滤器组件中，通过管道在相邻的过滤器之间传递数据。如果具体框架被结构化为一个管道过滤器体系结构，该体系结构具有作为端点的客户和对象，那么管道和过滤器链中的每个管道定义一个潜在的位置，在此位置可以在相邻的过滤器之间插入截取器。在这种情况下，截取器的注册由管道和过滤器链的重新配置组成。

语境对象是从源过滤器向截取器传递的信息。截取器负责以合适的格式发送信息给汇点过滤器。然而，在管道和过滤器模式中，通过管道链接过滤器，而在截取器模式中，在不同层上的具体截取器常常是独立的。另外，管道和过滤器为完整的应用程序“管线”定义了一个基础的计算模型，而截取器用来实现任何类型的具体框架的“带外”服务。

代理模式[GoF95][POSA1]为一个对象提供一个代理或者占位符以控制对它自己的访问。尽管代理可以把附加的功能集成到一个系统，但它们只能用于系统可见的对象。相反，截取器允许外部组件访问和控制内部的和其他“不可见”组件。正如在“变体”部分所描述的，为实例化截取器代理变体，可以实例化具有增强功能（如语境对象）的代理模式。

观察者[GoF95]和出版者-订阅者(Publisher-Subscriber)[POSA1]模式有助于合作组件的状态

同步。这些模式单向传播状态变化，当主题的状态改变时，出版者可以通知一个或者多个观察者/订阅者。与截取器模式相比，观察者和出版者-订阅者模式并不指定观察者/订阅者应如何访问出版者的功能，因为它们只定义从出版者到订阅者的单向通信。这些模式也强调事件通知功能，而截取器模式则关注服务到框架的集成。 139

事件对象和语境对象之间的不同也说明了这些差异。在事件对象常常包含与当前事件有关的值的同时，语境对象还提供一个额外的编程接口来访问和控制具体框架。因此观察者和出版者-订阅者(Publisher-Subscriber)模式可以看做是截取器模式的变体，其中，语境对象对应于从扮演主题角色的框架向扮演观察者/订阅者角色的截取器传递的事件类型。

映像(Reflection)模式[POSA1]提供了一种机制用于改变软件系统的结构和行为。基级(base-level)对象的层包含应用逻辑。由元级(meta-level)提供有关系统属性的信息，并允许开发人员控制基级的语义。映像模式和截取器之间的关系有两个方面：

- 截取提供实现映像机制的方法。例如为实例化映像模式，可以引入分配器，它通过向元级注册截取器来帮助开发人员引入新的行为。因此截取能看做是轻量级映像方法，它更易于实现，且消耗较少的CPU和内存。而且，截取仅揭示系统底层的某些内部状态，而映像常涉及更广的范围。
- 映像能定义一种截取机制类型。映像的主要目的是允许应用程序观察它们自己的状态，从而能动态地改变它们自己的行为。与此相反，截取器的主要目的是允许其他应用扩展和控制具体框架的行为。

反应器模式多路分解和分配服务请求，这些请求从一个或多个客户机并发地分发给应用程序。反应器模式主要考虑处理与系统有关的事件，而截取器模式有助于截取与应用有关的事件。常常实例化反应器模式以处理在通信框架低层发生的系统事件，而截取器模式用于框架和应用程序之间的多个层次。

致谢

感谢Fabio Kon写的已知使用中的dynamicTAO部分。 140

2.4 扩展接口

扩展接口(Extension Interface)设计模式允许组件导出多个接口，当开发人员扩展或修改组件的功能时，此模式能防止接口的膨胀和客户机代码的破坏。

1. 例子

考虑一个通信管理网络（Telecommunication Management Network, TMN）[ITUT92]框架，可以定制该框架，以监视和控制远程网络元素（如IP路由器和ATM交换机）。依照模型-视图-控制器(Model-View-Controller)模式[POSA1]，每种类型的网络元素建模为多个部分的框架组件。视图和控制器位于管理应用控制台。视图在控制台上呈现网络元素的当前状态，控制器允许网络管理员管理网络元素。

模型驻留在网络元素上，并且与视图和控制器通信，以接收和处理命令，例如向管理应用控制台发送有关网络元素的状态信息的命令。TMN框架中的所有组件按层次进行组织。图2-23所示的UniversalComponent接口提供每个组件所需的常用功能，例如显示网络元素的关键属性和访问它的邻居。

141

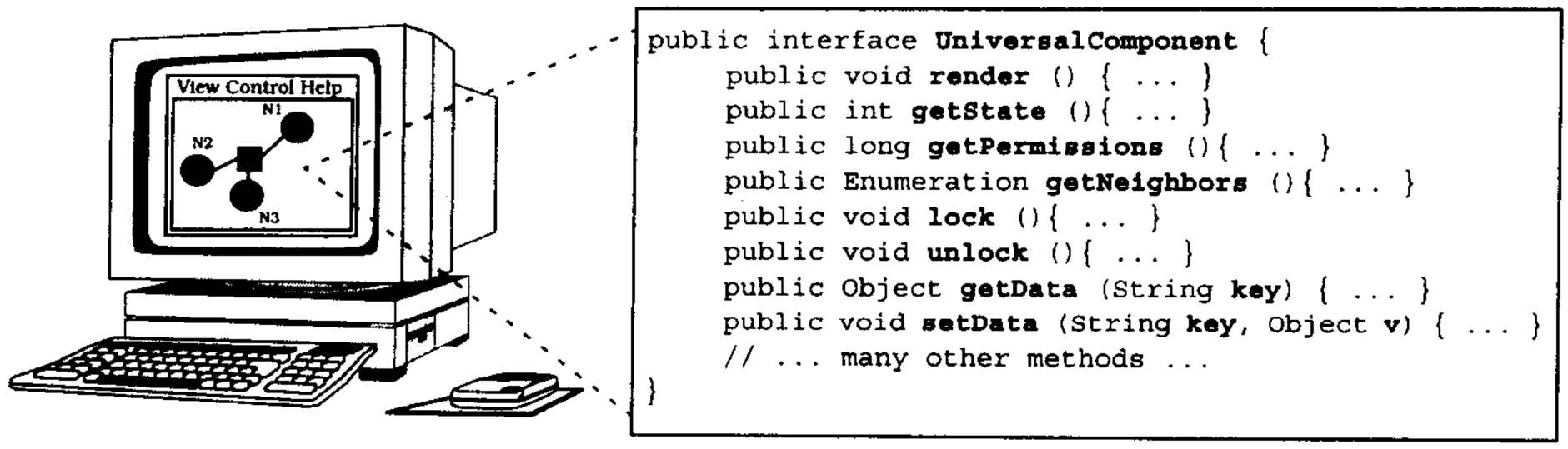


图 2-23

在理论上讲，如果从不改变上面所显示的UniversalComponent接口，那么这种设计也许是合适的，因为它会允许客户机应用程序通过一致的接口访问大量网络元素。然而，实际上随着TMN框架变得越来越流行，管理应用程序开发人员将要求把新功能特性和新方法（如dump()和persist()）添加到UniversalComponent接口中。

随着时间的推移，这些增加的要求会因增加在最初框架设计中并未考虑的功能而使接口膨胀。如果直接把新方法添加到UniversalComponent接口中，那么必须更新所有的客户机代码并且重新编译它们，这将是繁琐的和易出错的。因此关键的设计难题是确保对TMN框架的逐步扩展不使它的接口膨胀或者破坏它的客户机代码。

2. 语境

一个应用程序环境，其中组件接口会随着时间而演化。

3. 问题

适应应用程序需求的改变常常需要对组件功能的修改和扩展。有时可以在发布组件给应用程序开发人员之前预计到所有的接口的改变。在这种情况下，可以应用“Liskov替换原则”[Mar95]。这个原则定义稳定的基接口，此接口的方法能通过子类和多态性单独地扩展。

然而在其他情况下，设计稳定的接口是困难的。因为在把组件分发和集成到应用程序之后，需求能以非预期的方式改变。如果不能仔细处理这些改变的话，它们就会破坏现有的使用组件的代码。另外，如果新功能仅被少数应用程序所使用，那么所有其他的应用程序必然招致不必要的时间和空间开销以支持它们并不需要的组件服务。

为了避免这些问题，把组件设计为支持预料到和预料不到的改进也许是必需的。这要求解决以下四种强制条件：

142

- 当组件接口没有改变时，对组件实现的修改应不破坏现有的客户机代码。
  ➡如果UniversalComponent接口实现在外部存储中持久存储它们的状态，只要组件的接口不改变，即使以不同的方式重新实现该功能，客户机也不应该受影响。 ❑

- 当开发人员用外部可见的新服务扩展组件时，不应破坏现有的客户机代码。理想情况下，应该没有必要重新编译客户机代码。

  ➡或许需要为UniversalComponent接口增加一个登录服务，这样管理应用程序和网络元素就可以到中心仓库登记信息。知道UniversalComponent原始版本的现有客户机应不受这个改变的影响，而新的客户机应能利用新的登录功能。❑

- 组件功能的改变或者扩展应该是相对直接的，既不膨胀现有的组件接口，又不动摇现有组件的内部体系结构。

  ➡当增加上述的登录服务时，应使对现有的UniversalComponent接口实现的改变减至最少。❑

- 应该可以使用同一接口远程地或者局部地访问组件。如果组件和它们的客户机跨越网络节点分布，那么应能分离组件的接口和实现。

  ➡在我们的TMN系统中，管理应用程序能从对远程网络元素的位置透明的访问中获益。因此从它们的物理实现中分离网络元素管理组件的接口是可能的。这些接口可以在全网络分布。❑

4. 解决方案

编写客户机程序以通过独立的接口来访问组件，每个接口对应组件扮演的角色，而不是编程使客户机使用单一组件，此组件把它的所有角色合并到一个单一接口或者实现。 143

详细地说：通过扩展接口导出组件功能，每个语义相关的操作集对应于一个接口。组件必须至少实现一个扩展接口。为了给组件增加新的功能，或者修改现有的组件功能，导出新的扩展接口而不是修改现有的接口。而且，编程使客户机通过它的扩展接口而不是它的实现来访问组件。因此，客户机仅依赖于对组件的不同角色，组件中的每个角色由一个独立的扩展接口来表示。

为使客户机能创建组件实例并且获取组件扩展接口，我们引入额外的间接方法。例如，为创建组件实例的每个组件类型引入一个相关联的组件工厂。确保它返回一个最初的接口引用，客户机能使用它来获取其他组件的扩展接口。类似地，确保每个接口从定义所有组件（如用于获取特殊扩展接口的机制）公共功能特性的根接口(root interface)继承。所有其他的扩展接口从根接口派生，这确保它们最少提供根接口所导出的功能。

5. 结构

扩展接口模式的结构包含四个组成部分：

组件(Component)聚合和实现多种类型的与服务相关的功能。常常把这个功能划分为几个独立的角色，每个角色定义一个语义相关的操作集（如图2-24）。 144

➡TMN框架中的组件扮演各种角色，例如存储和获取网络元素的状态或者管理组件的内部状态的持久性(persistence)。❑

扩展接口(Extension interface)导出经过选择的组件实现的面(facet)。对应于组件所实现的每个角色[RG98]都有一个扩展接口存在。另外，扩展接口隐式地指定描述客户机应如何使用组件功能的协定。这个协定定义了调用扩展接口方法的协议，例如可接受的参数类型和调用方法的次序。

类 组件	协作者
责任 • 扮演不同角色 • 实现扩展接口 • 返回对组件工厂的初始接口	

图 2-24

➡TMN框架中的组件能够实现IStateMemory接口，此接口允许组件在内存中维护它们的状态。持久管理器，例如CORBA持久状态服务[OMG99e]可以在不需要组件揭示它们的描述性细节的情况下使用IStateMemory接口管理组件持久性。

如果新增加的网络元素组件也实现IStateMemory，那么持久管理器可以在不需做任何改变的情况下管理它们的持久性。IStateMemory接口包含一些方法，为读和写组件的状态作一些准备工作，还包括它的读和写操作。因此接口和它的用户之间的这种隐式协定规定必须在readState()或者writeState()之前调用prepare()方法。 ❑

根接口(root interface)是一个特殊的扩展接口，它提供三种类型的功能特性：

- 核心功能特性(Core functionality)，所有扩展接口都必须支持它们（如允许客户机获取它们请求的接口的功能特性）。这个功能特性定义组件必须实现的允许客户机获取接口和在接口之间导航的基本机制。
- 领域无关功能(Domain-independent functionality)，例如管理组件生存期的方法。
- 与领域有关的功能特性(Domain-specific functionality)，所有组件在一个特殊领域范围内应该提供的功能。

145

尽管根接口必须实现核心功能特性，但它不必支持领域无关或者领域有关的功能特性。然而，所有扩展接口必须支持根接口定义的功能。因此，每个扩展接口可以扮演根接口的角色，它保证每个扩展接口能代表客户机请求返回任何其他的扩展接口（如图2-25）。

类 根接口	协作者 • 组件
责任 • 定义每个扩展接口必须提供的功能特性	

类 扩展接口	协作者 • 组件
责任 • 定义由组件提供的角色特定的接口	

图 2-25

➥在TMN框架中，UniversalComponent接口可以定义为根接口。与“例子”一节中介绍的多面的——和不断膨胀的——UniversalComponent接口不一样，这个根接口仅定义了在TMN框架中所有组件常用的最小方法集。 ❑

客户机只能通过扩展接口访问组件提供的功能。客户机获取一个初始扩展接口的引用之后，它能够使用这个引用来获取组件支持的任何其他扩展接口。

➥管理应用程序控制台客户能够使用TMN框架中的组件在屏幕上可视化地呈现网络元素的状态和它们的关系，也可以使用持久存储器来存储和获取它们的状态。 ❑

为获取初始引用，客户机同与特定组件类型相关联的组件工厂交互。这个组件工厂将它的处理特性与组件的创建和初始化特性分离出来。当客户机创建一个新的组件实例时，它把这项任务委托给合适的组件工厂。 146

当成功地创建组件之后，组件工厂将对扩展接口的引用返回给客户。组件工厂也许允许客户机请求特定类型的初始扩展接口。工厂也可以提供一些功能定位和返回现有组件实例的引用（如图2-26）。

类 客户机	协作者 • 扩展接口 • 根接口 • 工厂
责任 • 实现应用特定的功能特性 • 访问工厂以创建新的组件 • 使用扩展接口访问组件功能特性	

类 组件工厂	协作者 • 组件
责任 • 定义创建组件的功能 • （可选地）包含定位已有组件的功能特性	

图 2-26

类图图2-27阐明扩展接口模式中的参与者。此图强调组件之间的逻辑关系而不是物理关系。例如，正如在实现活动（6.1）中描述的那样，可以使用多继承或者嵌套类实现扩展接口。这种实现细节对客户机而言是透明的。

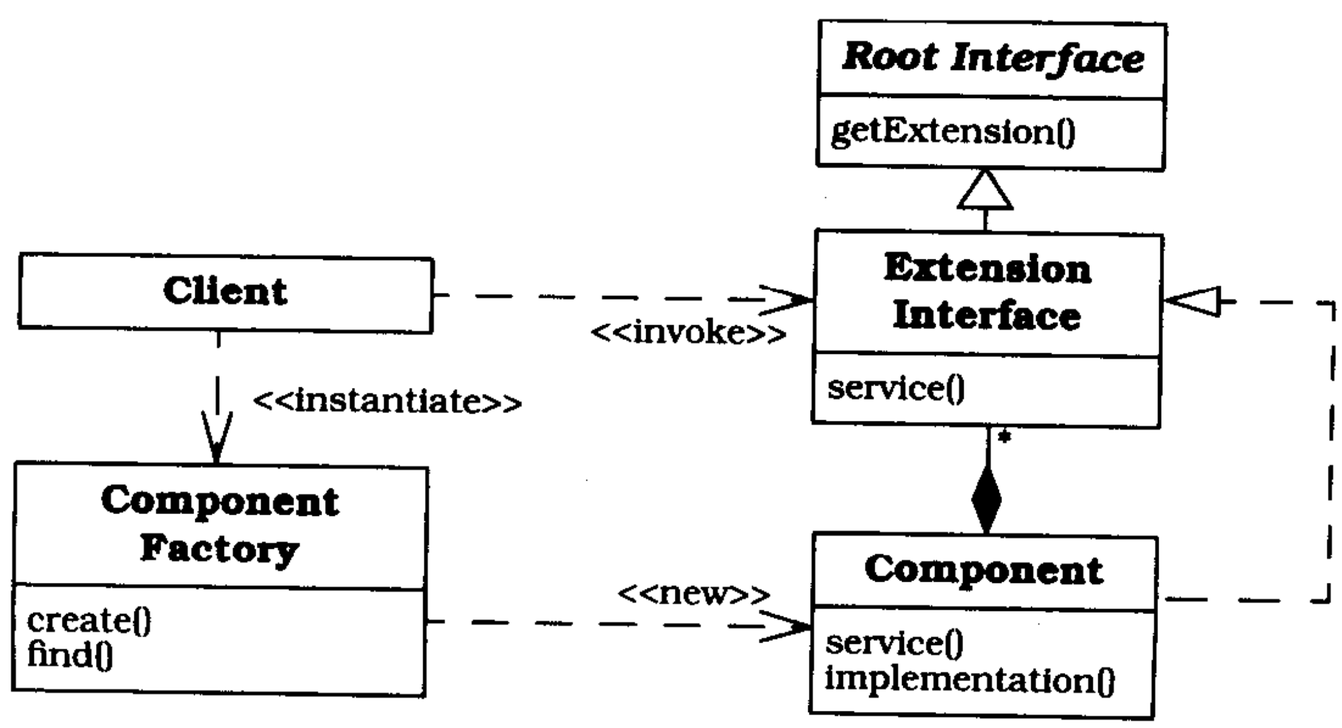


图 2-27

6. 动态特性

我们使用两个场景说明扩展接口模式中的关键协作。**场景I**描述客户如何创建新的组件并获取初始扩展接口（如图2-28）：

- 客户机请求组件工厂创建一个新组件，并返回对特殊扩展接口的引用。
- 组件工厂创建新组件并获取对其根接口的引用。
- 组件工厂向根接口查询被请求的扩展接口，接着将扩展接口的引用返回给客户机。

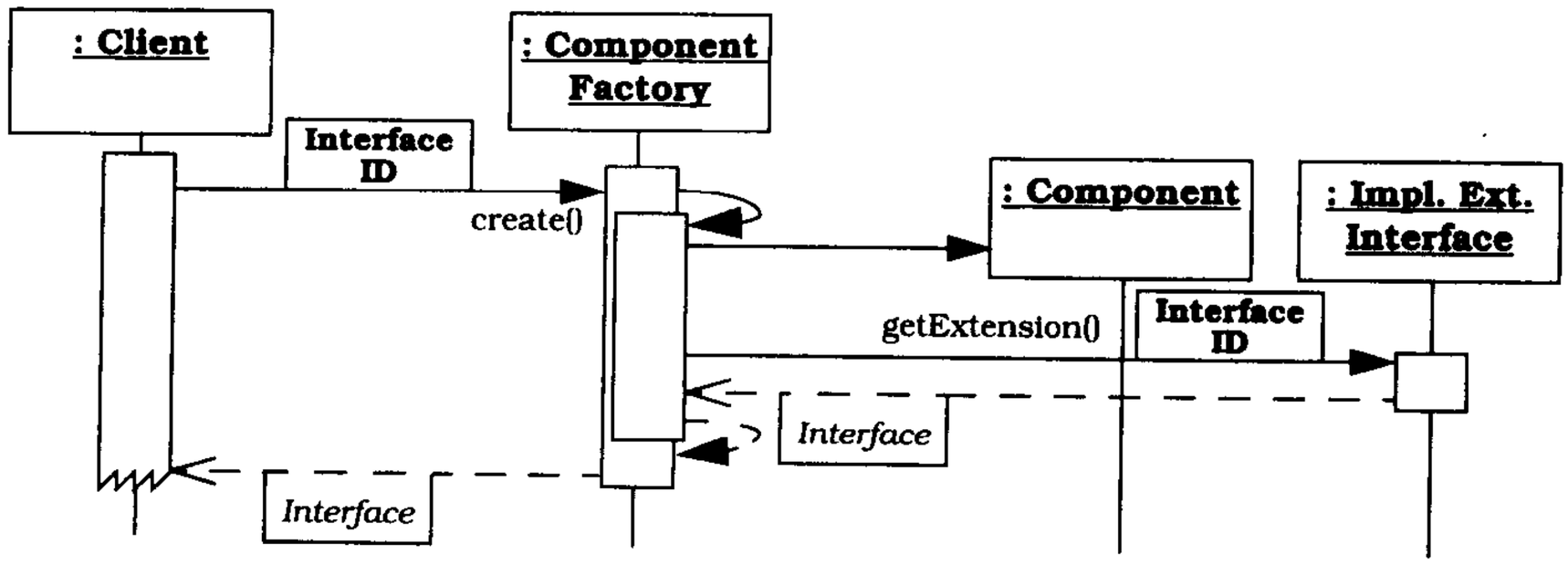


图 2-28

注意工厂可以给客户机返回任何接口，而不是获取一个特定的扩展接口。然而这样的设计会导致在分布式系统中额外的往复，这增加访问所需接口的开销。

场景II描述客户机和扩展接口之间的协作（如图2-29）。注意组件实现自己对客户机而言是不可见的，因为它只处理扩展接口：

- 客户机调用扩展接口A上的方法，此接口可以是根接口也可以是扩展接口。
- 在组件范围内扩展接口A的实现执行被请求的方法，如果有结果的话，返回结果给客户机。
- 客户机调用扩展接口A上的getExtension()方法，并传给它客户机感兴趣的指定扩展接口的参数。getExtension()方法定义在根接口中，因此它被所有扩展接口所支持。组件范围内扩展接口A的实现定位被请求的扩展接口B，并将扩展接口B的引用返回客户机。
- 客户机调用扩展接口B上的方法，接着在组件实现中执行此方法。

148

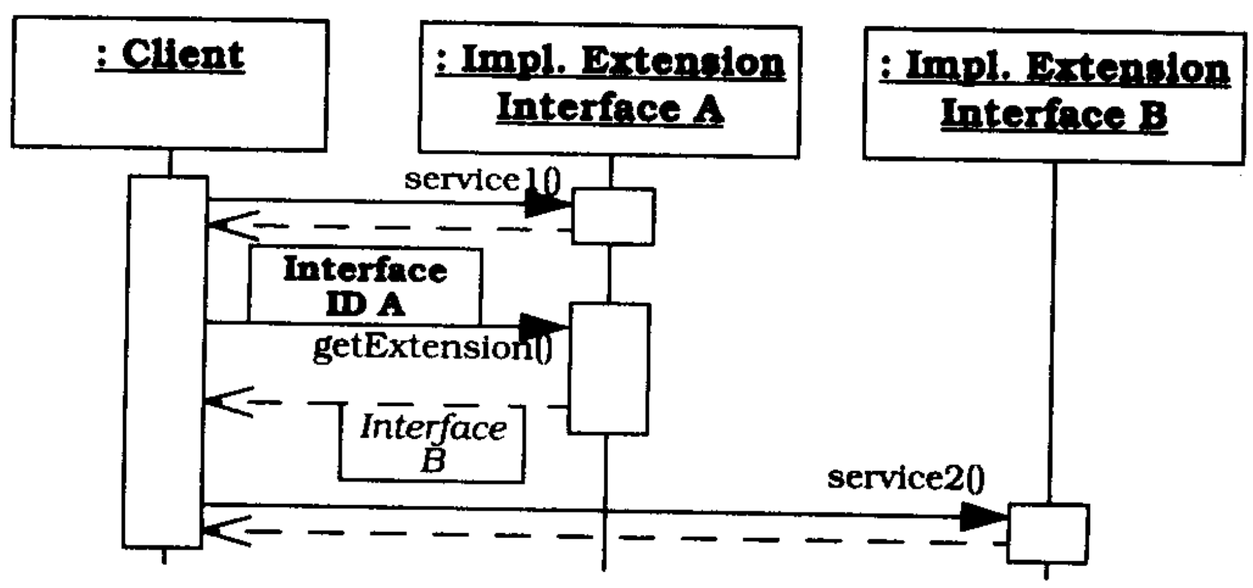


图 2-29

7. 实现

本节描述与实现扩展接口模式有关的活动。这种模式对于用过微软的组件对象模型（Component Object Model, COM）[Box97]、Enterprise JavaBeans (EJB)[MaHa99]或者CORBA组件模型（CORBA Component Model, CCM）[OMG99a]编程的人应该是熟悉的，因为它吸收并推广了支持这些组件技术的核心概念。

(1) *确定设计和长期应用需求的平稳性*。在运用扩展接口模式之前，有必要确定它是否真正需要。尽管这种模式对某些强制条件而言是强有力的解决方案，但实现并不容易。如果随意应用，就会使软件设计极端复杂化。

因此我们推荐仔细考虑在“问题”一节概述的强制条件。在运用这种模式之前，应确保软件系统中已解决了这些问题。例如，可能一个接口所需的所有方法都可以在系统开发期间确定，也可能当应用需求变化时，接口将不随时间改变。在这种情况下，使用Liskov替换原则[Mar95]比使用扩展接口模式会更简单一些。 149

(2) *分析领域并指定与领域有关的组件模型*。假定扩展接口模式是必需的，那么下一个活动涉及分析与领域相关的应用需求。尤其是，这个活动关注与应用有关的实体的标识，例如TMN例子中的网络元素、某一实体提供给系统的角色、以及支持不同角色的功能。结果是标识要实现哪个组件，也标识组件必须提供的功能的领域模型。

➡对于管理应用控制台，被控制的每种类型的实体作为独立的被控对象(managed object)[ITUT92]实现，该被控对象是对硬件单元（例如，路由器、计算机、网桥或者交换器）的抽象。被控对象也能代表软件元素，例如应用程序、端口或者连接。管理应用使用被控对象控制和监视网络元素的状态，显示调试信息，或者在管理控制台上使系统行为可视化。 ❑

在设计一个领域模型之后，需要指定组件模型以实现被标识的组件：

- 如果组件限制在单一应用领域或者相关领域的较小集合，那么考虑定义一个与领域有关的组件模型，此组件模型是为正被开发的应用程序或者应用程序族定制的。
- 相反，如果组件必须运用于宽范围的应用程序，或者甚至跨越多个领域，那么考虑使用现有的组件技术，例如微软COM[Box97]、Enterprise JavaBeans [MaHa99]或者CORBA组件模型[OMG99a]。

在后一种情况，可以跳过下一个实现活动，因为这些组件模型定义了指定的基础设施。

(3) *指定根接口*。确定实现活动(2)中所标识的每种类型的功能特性是否应该是根接口的一部分，或者被分离到扩展接口。 150

按照这个标准迭代进行三个子活动：

(3.1) *指定核心功能特性*。当定义根接口的核心功能特性时，必须处理几个问题：

- *扩展接口获取*。根接口最少必须包含向客户机返回扩展接口引用的方法。这个方法返回的信息类型很大程度上依赖于编程语言。例如，Java客户机期望获取一个对象引用，而对于C++而言，指针是合适的选择。
- *惟一的命名*。必须使用整数或者字符串命名扩展接口。编程人员和简单的管理工具更容易读取字符串，但整数值所占空间更小，能更有效地处理。为防止名字冲突，应该能够使用算法生成接口标识符。例如，微软COM使用128位基于网络接口地址、日期和时间的全局

惟一标识符（Globally Unique Identifier, GUID）。

- 错误处理。组件开发人员必须确定当客户机请求一个并不支持的扩展接口时，组件应该做什么。例如，组件可以返回一个错误值，或者引发一个异常。在包装器外观模式的“实现”一节讨论了几个用于实现错误处理机制的策略，并对它们进行了权衡比较。

(3.2) 指定与领域无关的服务。除定义获取扩展接口的方法之外，根接口也能为各种领域无关的服务提供方法。这里有两种可能：

- 引用计数(Reference counting)。在不支持自动无用单元收集的编程语言（如C或者C++）中，客户机负责删除它们不再需要的扩展接口。然而，多个客户机也许共享同一扩展接口。因此组件可以提供引用计数机制，以防止意外删除用于实现扩展接口的资源。

  引用计数使组件能够跟踪访问指定的扩展接口的客户机数。当一个扩展接口不再被任何客户机引用时，组件接口的实现所使用的资源能自动地释放。计数指针惯用法[POSA1][Cope92]提供几个实现引用计数机制的选项。

151

- 运行时映像(Run-time reflection)。领域无关服务的另一个例子是运行时映像机制。这种机制允许组件发布有关它们支持的指定角色、扩展接口和方法的信息。使用这一知识，客户机能动态地构造和发送方法调用[GS96]。这允许编写脚本语言在运行时把组件集成到现有的客户机应用程序。可以使用映像体系结构模式[POSA1]实例化映像机制。

(3.3) 指定与领域相关的功能特性。如果所有实现根接口的组件提供这个功能的话，那么根接口也能导出领域相关的功能。

➡在管理应用控制台，也可以把绘画功能移到根接口。 ❑

关于指定根接口中哪个与领域相关的服务的决定通常应被推迟到实现活动（4）之后。理想情况是，所有与领域相关的功能应该驻留在独立的扩展接口。如果所有组件最后实现了特定的扩展接口，那么就重新分解当前的解决方案[Opd92][FBBOR99]，并把那个特定的扩展接口方法移到根接口中。

➡在TMN框架例子中，用惟一的整数常量标识扩展接口。我们使用Java作为实现语言，因为它提供自动的无用单元收集，这简化了内存管理。因此在根接口中所需的惟一常用功能——我们称之为IRoot——是允许客户机获取它们需要的任何接口的方法。

```
// Definition of IRoot:
public interface IRoot {
    IRoot getExtension (int ID) throws UnknownEx;
}
```

152

IRoot作为所有扩展接口的通用基接口使用。如果组件不支持一个特定的接口，那么它会抛出一个UnKnownEx异常。

```
// Definition of UnknownEx:
public class UnknownEx extends Exception {
    protected int ID;
    public UnknownEx (int ID) { this.ID = ID; }
    public int getID () { return ID; }
}
```

把被请求接口的惟一标识符作为参数传递给UnKnownEx构造函数。这样客户机能够确定哪

个接口引发异常。 ❑

包含在根接口中包含与领域相关的功能的另一个潜在的候选机制是持久性机制。然而，有许多不同的用于处理持久性的策略和方针，例如数据库或者“flat”文件中的管理组件状态，这使预测所有可能的使用情况变得困难。因此，组件可以通过实现特定的扩展接口来选择支持，它们认为是合适的任何持久性机制。

(4) *引入通用扩展接口*。通用扩展接口包含功能角色，此角色必须由多个组件提供，并且不包含在根接口中。应该为每个角色定义一个独立的扩展接口。例如，可以定义扩展接口来处理组件的持久特性，这在实现活动（3.3）的结尾讨论过。

➡管理应用控制台通过被控对象来帮助控制和监视远程网络实体。被控对象作为组件实现，该组件给管理应用控制台发送信息，并从管理应用控制台接收命令。 153

因此每个被控对象实现下列接口IManagedObject：[⊖]

```
// Definition of IManagedObject:
import java.util.*;

public interface IManagedObject extends IRoot {
    public void setValue (String key, Object value);
    public Object getValue (String key) throws
            WrongKeyEx;
    public void setMultipleValues
        (Vector keys, Vector values);
    public Vector getMultipleValues
        (Vector keys) throws WrongKeyEx;
    public long addNotificationListener
        (INotificationSink sink);
    public void removeNotificationListener (long handle);
    public void setFilter (String expr);
}
```

这个例子说明了在管理控制台上被可视化的被控对象。因此引入两个附加的扩展接口IDump和IRender，它们由在控制台上输出调试信息或者显示其所有组件实现。

```
// Definition of IDump:
public interface IDump extends IRoot {
    public String dump ();
}

// File IDraw.java.
public interface IRender extends IRoot {
    public void render ();
}
```

❑

如果所有组件必须支持一个特殊的通用扩展接口，那么它可以灵活地重新分解在实现活动（3.3）中指定的根接口，并在那里集成这个功能。然而注意，重新分解根接口会由于功能特性增加而过度膨胀或者破坏现有的应用程序，因而抵消扩展接口模式的优点。 154

(5) *定义与组件有关的扩展接口*。在实现活动（3）和（4）中指定需要导出一般组件功能的扩展

⊖ 注意组件应提供客户机本地访问的接口，例如IManagedObject，而组件的实际实现驻留在远程网络节点。通过使用代理模式[POSA1][GoF95]，使分布性对于客户机而言是透明的。为了澄清这个问题，我们假定在这个例子中所有接口具有本地实现。有关如何引入代理以支持分布式环境的信息，请参见*分布式扩展接口*变体。

接口。这个实现活动定义附加的接口，此接口是与某一组件有关的，或者只可用于受限范围的组件。

➡对于TMN框架，我们指定扩展接口IPort和IConnection。在某一主机上代表端口的被控对象实现IPort：

```
// Definition of IPort:
public interface IPort extends IRoot {
    public void setHost (String host);
    public String getHost ();
    public void setPort (long port);
    public long getPort ();
}
```

同样地，代表在两个端口之间连接的对象实现IConnection。

```
// Definition of IConnection:
public interface IConnection extends IRoot {
    public void setPort1 (IPort p1);
    public IPort getPort1 ();
    public void setPort2 (IPort p2);
    public IPort getPort2 ();
    public void openConnection () throws CommErrorEx;
    public void closeConnection () throws CommErrorEx;
}
```

❑

(6) 实现组件。组件的实现包括五个子活动：

(6.1) 指定组件实现策略。这个活动确定依据下列三个策略应该如何链接扩展接口实现：

- 多重继承(Multiple inheritance)。在这个策略中，组件类继承它的所有扩展接口。
- 嵌套类(Nested classes)。在这个策略中，扩展接口可以实现为组件类内的嵌套类。组件类实例化每个嵌套类的单件实例[GoF95]。一旦客户机查询一个特定的扩展接口时，getExtension()方法实现返回合适的嵌套类对象。
- 独立接口类(Separate interface classes)。扩展接口可以使用桥或者适配器模式[GoF95]实现独立于组件自身的独立类。当应用扩展接口模式重新分解并没有实现模式的一个现有组件时，这个策略特别有用。

  在CORBA IDL映射中为Java和C++定义的“捆绑”（tie）适配器［SV98a］是组件实现策略的一个示例。在CORBA中捆绑适配器继承一个自动生成的服务者（Servant）类，重载所有的纯虚方法，并把这些方法委托给另一个C++对象，即所谓的“捆绑对象”。服务器应用程序开发人员定义捆绑对象。

不管选择哪种组件实现策略，客户机都不受影响，因为它仅通过对外部接口的引用访问组件。

(6.2) 实现获取扩展接口的机制。当代表客户机实现获取扩展接口的一般方法时，应该确保方法实现符合三个约定：

- 自反性(Reflexivity)。当客户机向扩展接口A查询同一扩展接口A时，客户机必须总能接收到同一引用A。
- 对称性(Symmetry)。如果客户机能从扩展接口A获取扩展接口B，那么它也必须能从扩展接口B获取扩展接口A。
- 传递性(Transitivity)。如果客户机能从扩展接口A获取扩展接口B，从扩展接口B获取扩展接口C，那么它必须可以从扩展接口A直接获取扩展接口C。

155

遵守上述约定确保客户机总可以从组件的指定的扩展接口导航到同一组件的任何其他扩展接口。换句话说，每个扩展接口能通过导航和每一个其他扩展接口连接。 156

(6.3) 实现引用计数机制（可选的）。如果根接口需要在实现活动（3.2）中讨论的引用计数机制，那么在必须被这种机制管理的组件实现中指定资源。实现引用计数机制有两个常用的选择：

- 如果每个接口实现需要或者使用独立的资源，或者如果对每个接口存在独立的实现，那么为每个扩展接口引入独立的引用计数器。如果一个特定的引用计数器下降为零，那么释放所有被相应的扩展接口所使用的资源。在最后一个引用计数器降为零之后，释放所有与组件相关联的资源。
- 如果所有接口实现共享同一资源，那么为整个组件引入全局引用计数器。在这个全局引用计数器为零之后，释放组件的资源。

第一个选项比第二个选项能更有效地优化资源管理，因为在第二个选项中，所有资源必须一直是可用的。相反，在第一个选项中，扩展接口和它们所需要的资源是按需激活和停用（deactivated）的。只有被客户机实际使用的扩展接口和资源才是激活的。然而，维护与扩展接口有关的引用计数器的缺点在于组件范围内它们的复杂实现。

➡在TMN框架中，可以按需运用引用计数来激活和停用扩展接口实现。这避免了不必要的资源（如内存或者套接字句柄）投入。例如，当管理应用客户机访问一个引用计数器是零的扩展接口时，组件能够透明地激活接口实现和它的资源。当没有客户机访问扩展接口时，可以选择性地停用和释放相应的实现和资源。COM[Box97]组件模型实现了这个策略。 ❑ 157

(6.4) 选择并发策略。在并发或者网络化系统中，多个客户机能同时访问一个特殊的扩展接口。不同的扩展接口实现也许在组件实现范围内共享状态和资源。因此，组件实现范围内的临界区分和状态必须被串行化以防止客户机的并发访问造成破坏。

主动对象和监视器对象并发模式，也有定界加锁、策略化加锁和线程安全接口同步模式，为保护组件范围内的临界区分和状态定义多种策略和机制。

(6.5) 使用所选择的组件实现策略实现扩展接口功能。这种实现活动大部分是与领域有关的或者与应用有关的，因此没有需要处理的通用问题。

➡在TMN框架例子中，我们使用多接口继承实现组件，这里的组件并不需要显式的引用计数，因为Java提供了自动的无用单元收集功能。

为简单起见，我们不阐明组件并发策略。为了惟一地标识不同的扩展接口，我们定义一个InterfaceID类，它枚举所有接口标识符，在下面的类型中将接口标识符定义为整数：

```
// Definition of InterfaceID:
public class InterfaceID {
    public final static int ID_ROOT   = 0;
    public final static int ID_MANOBJ= 1;
    public final static int ID_DUMP   = 2;
    public final static int ID_RENDER= 3;
    public final static int ID_PORT   = 4;
    public final static int ID_CONN   = 5;
}
```

一个更复杂的实现要使用接口标识符仓库。在这种情况下，当不同的组件提供商定义不同的接口时，通过工具可以自动地产生惟一标识符以防止名字冲突。我们也可以使用String而不

158 是int作为标识符类型。这样可以提高组件系统的可读性和可调试性，但也使内存开销更大，查询时间更长。

在管理应用控制台中有一类组件表示两个端口之间的连接。这个组件支持扩展接口IManagedObject、IRender、IConnection和IDump。我们使用Java接口继承实现所有的扩展接口：

```
// Definition of ConnectionComponent:
public class ConnectionComponent implements
IManagedObject, IRender, IDump, IConnection {
    // <table> contains all properties.
    private Hashtable table = new Hashtable ();

    // <listener> contains event sinks.
    private Hashtable listeners = new Hashtable ();
    private long nListeners = 0;

    private IPort port1, port2;
    private String filterExpression;

    // <IRoot> method.
    public IRoot getExtension (int ID)
        throws UnknownEx {
        switch(ID) {
            case InterfaceID.ID_ROOT:
            case InterfaceID.ID_MANOBJ:
            case InterfaceID.ID_DUMP:
            case InterfaceID.ID_RENDER:
            case InterfaceID.ID_CONNECT:
                return this;
            default:
                throw new UnknownEx (ID);
        }
    }
```

请留意getExtension()接口是如何使用switch语句以确定组件支持哪个接口的。如果标识符类型定义为String而不是int，那么我们可以使用不同类型的查询策略，例如线性查找、动态散列或完全散列[Sch98a]。

159

```
// Definition of IManagedObject:
public void setValue (String key, Object value) {
    table.put (key, value);
}

public Object getValue (String key)
    throws WrongKeyEx {
    WrongKeyEx wkEx = new WrongKeyEx ();
    if (!table.containsKey (key)) {
        wkEx.addKey (key); throw wkEx;
    }
    return table.get (key);
}

// Additional methods from <IManagedObject>.
public void setMultipleValues
    (Vector keys, Vector values) { /* ... */ }
public Vector getMultipleValues
    (Vector keys) throws WrongKeyEx { /* ... */ }
public long addNotificationListener
```

```
        (INotificationSink sink) { /* ... */ }
public void removeNotificationListener
        (long handle) { /* ... */ }
public void setFilter (String expr) { /* ... */ }

// <IDump> and <IRender> methods.
public String dump () { /* ... */ }
public void render () { /* ... */ }

// <IConnection> methods.
public void setPort1 (IPort p1) { port1 = p1; }
public IPort getPort1 () { return port1; }
public void setPort2 (IPort p2) { port2 = p2; }
public IPort getPort2 () { return port2; }
public void openConnection () throws CommErrorEx { }
public void closeConnection () throws CommErrorEx { }
}
```

❑

(7) 实现组件工厂。每个组件类型必须实现一个工厂，客户机要使用这个工厂来获得组件类型的实例。这包括三个子活动：

(7.1) 定义组件工厂和组件之间的关联。对于每个组件类型，都可以定义一个单件[GoF95]组件工厂来创建这个组件类型的实例。可以运用两种策略来实现这种关联：

- 每种组件类型对应一个组件工厂。在这种策略中，为每个组件类型定义一个独立的工厂接口，并使用它实例化组件类型。一个组件类型能够提供一个具有单一方法create()的组件工厂。另一种组件类型可以选择创建组件的不同方法。可以使用工厂方法(Factory Method)模式[GoF95]实现这种策略。不过它需要客户机处理许多不同的组件工厂接口。 160
- 所有组件类型对应一个组件工厂接口。在这种策略中，只有一个必须由所有具体组件工厂实现的组件工厂接口。这种设计使客户机能以一致的风格创建不同的组件。例如，当客户机创建一个新组件时，它只需知道如何去调用通用的组件工厂接口。可以使用抽象工厂(Abstract Factory)设计模式[GoF95]实现这种策略。

(7.2) 确定工厂将导出的功能特性。无论在实现活动(7.1)中选择的策略是什么，当指定一个特殊组件工厂的接口时，必须处理下列要点：

- 可以有一个或多个不同方法用于创建新的组件。这些创建方法与面向对象的编程语言（如C++或Java）中的构造函数相似，因为它们实例化和初始化组件类型。构建一个新组件实例或许需要不同类型的初始化信息。对于每一个选择，引入一个独立的创建方法，这个方法具有它自己的初始化参数集，这个集合也可能是空集。
- 有一些方法可以用于发现现有的组件而不是为每次调用而创建组件。如果已有组件实例，并且能惟一地标识它们，那么可以使用管理者模式[Som97]实现一个工厂。在这种情况下，向find()方法传递一组条件作为参数（如与每个组件相关联的主关键字）。接着它获取一个或多个依附于条件变量的组件。
- 客户机能指定组件使用策略。例如，一个策略为某一类组件提供一个单件实现。另一个策略会确定是否期望某个组件持久地维护它的状态。
- 支持组件的生命期管理是组件工厂接口的另一候选功能。例如，可以在组件工厂包含释放已有组件的方法。 161

➡在TMN框架中，我们为每个被控对象提供一个作为单件实现的独立的组件工厂。接口IFactory是通用的，并且被所有具体组件工厂实现所支持。它包含create()方法，客户机使用此方法实例化一个新组件，并给调用者返回IRoot接口：

```
// Definition of Factory:
public interface Factory {
    IRoot create ();
}
```

每个具体组件工厂必须实现这个工厂接口：

```
// Definition of ConnectionFactory:
public class ConnectionFactory implements Factory {
    // Implement the Singleton pattern.
    private static ConnectionFactory theInstance;

    private ConnectionFactory () { }

    public static ConnectionFactory getInstance () {
        if (theInstance == null)
            theInstance = new ConnectionFactory ();
        return theInstance;
    }

    // Component creation method.
    public IRoot create () {
        return new ConnectionComponent ();
    }
}
```

❑

(7.3) 引入组件工厂发现者。随着组件类型数目的增长，会出现如何发现相关联的组件工厂的问题。解决这个问题的一种方法是定义一个全局组件工厂发现者。这个发现者会维护组件类型和它们的组件工厂之间的关联，正如在实现活动（7.1）中所指出的。

为一个特殊的组件类型获取组件工厂，客户机必须告诉组件工厂发现者它们需要的组件类型。因此，必须惟一地标识组件类型。实现这个标识的常用方法是为每个组件类型引入一个主关键字(primary key)类型。这个主关键字类型有助于惟一地把组件实例和主关键字类型的实例关联起来。

162

例如，每个组件实例可以和一个整数值惟一地关联。可以把这个整数值作为一个变量传递给组件工厂的特殊find()方法，此方法使用主关键字去获得相关联的组件实例。为此组件工厂可以运用管理者模式[Som97]，并将主关键字值映射为组件实例。为了简化客户机编程，可以使用同一主关键字类型标识组件实例和扩展接口，如在实现活动（3.1）中所示的。例如，在微软COM中，全局惟一标识符（GUID）标识扩展接口和组件类型。

当客户机从组件工厂发现者请求一个指定的组件工厂时，工厂发现者返回组件工厂的接口。通过使用这个接口，客户机能实例化它们所需要的组件。如果系统中只有一个全局组件工厂发现者，那么就使用单件模式[GoF95]实现。

组件工厂发现者可以提供一个交易机制[OMG98b]。在这种情况下，客户机并不给组件工厂发现者传递一个具体组件类型。相反，它们指定一些属性，组件工厂发现者利用这些属性获取一个合适的组件工厂。例如，客户机可以指定扩展接口的特定属性，此属性是组件工厂发现

者所感兴趣的。接着组件工厂发现者定位一个实现所有被请求的接口的组件类型。

➡在TMN系统中，管理应用客户机不必知道所有组件工厂。因此我们引入一个组件工厂发现者负责管理具有组件-工厂关联的散列表。客户机仅需知道一个组件工厂发现者位于何处即可。要惟一地标识组件，可以运用在实现活动（6.5）中用于接口的相同策略。 163

引入包含整数值的类ComponentID，其中每一个整数值对应一个单一的组件工厂：

```
// Definition of ComponentID:
public class ComponentID {
    public final static int CID_PORT = 0;
    public final static int CID_CONN = 1;
}
```

将组件工厂发现者实现为单件。它包括两个公共可访问的方法。客户机或者组件必须调用registerFactory()方法向组件工厂发现者注册组件工厂。findFactory()方法用于搜索现有的组件工厂。

```
// Definition of FactoryFinder:
import java.util.*;

public class FactoryFinder {
    // ID/factory associations are stored in a hash table.
    Hashtable table = null;
    // Implement the Singleton pattern.
    private static FactoryFinder theInstance;

    public static FactoryFinder getInstance () {
        if (theInstance == null) {
            theInstance = new FactoryFinder ();
        }
        return theInstance;
    }

    private FactoryFinder () {
        table = new Hashtable ();
    }

    // Component factory is registered with the finder.
    public void registerFactory (int ID, Factory f) {
        table.put (new Integer (ID), f);
    }

    // Finder is asked for a specific component factory.
    public Factory findFactory (int ID)
        throws UnknownEx {
        Factory f = (Factory) table.get
            (new Integer (ID));
        if (f == null) throw new UnknownEx (ID);
        else return f;
    }
}
```

❑ 164

(8) 实现客户机。客户机使用组件提供的功能特性，它们也可以作为这些组件的容器。[⊖] 为

⊖ 一般地，组件被加载到运行时环境的地址空间，此环境向组件提供类似CPU时间和内存这类资源。这种运行时环境通常称为容器(container)，因为它为组件屏蔽了像操作系统这样的底层基础结构的细节。在非分布式的情况下，客户机可以包含组件，并因此扮演容器角色。

实现客户机，要执行下列步骤：

- 首先确定它们需要哪个组件功能特性。例如，确定是否存在这样一些组件，其中包括期望客户机提供的部分或所有功能。
- 标识哪些组件应合在一起，并确定哪些组件能使用其他组件。
- 确定在客户机应用范围内是否存在一些子系统，这些子系统会在其他应用中使用，把这些子系统分离出来作为新的组件类型。

在解决这些问题之后，通过在上面实现活动中概述的分析方法，使用被标识的组件集成客户机应用程序。

➡在例子中，为局部化TMN系统的初始化，我们在客户机中提供一个类Component Installer，它创建所有必需的组件工厂，并且把它们向组件工厂发现者注册：

```
class ComponentInstaller {
    static public void install () {
        // First, get the global factory finder instance.
        FactoryFinder finder =
            FactoryFinder.getInstance ();
        // Ask the factory finder for the comp. factories
        PortFactory pFactory =
            PortFactory.getInstance ();
        ConnectionFactory cFactory =
            ConnectionFactory.getInstance ();

        // Register both component factories.
        finder.registerFactory
            (componentID.CID_PORT, pFactory);
        finder.registerFactory
            (componentID.CID_CONN, cFactory);
    }
}
```

165

客户机应用程序的主类中定义了方法dumpALL()和drawALL()。两种方法都被传递一个组件数组作为参数。接着它们分别在数组上迭代，分别调用方法dump()和render()，查询每个组件的扩展接口IDump和IRender。这个例子显示，可以使用接口继承而不是实现继承来支持多态性。

```
// This client instantiates three components: two ports
// and a connection between them.
public class Client {
    private static void dumpAll (IRoot components[])
        throws UnknownEx {
        for (int i = 0; i < components.length; ++i) {
            IDump d = (IDump)
                components[i].getExtension
                    (InterfaceID.ID_DUMP);
            System.out.println (d.dump ());
        }
    }

    private static void drawAll (IRoot components[])
        throws UnknownEx {
        for (int i = 0; i < components.length; ++i) {
            IRender r = (IRender)
                components[i].getExtension
```

```
                    (InterfaceID.ID_RENDER);
            r.render ();
        }
    }
```

main()方法是进入客户机应用程序的入口点。它首先使用上面介绍的初始化组件初始化TMN系统，接着它获取代表端口和端口之间连接的所需的组件工厂：

```
public static void main (String args[]) {
    Factory pFactory = null;
    Factory cFactory = null;

    // Register components with the factory finder.
    ComponentInstaller.install ();

    // access factory finder.
    FactoryFinder finder =
        FactoryFinder.getInstance ();

    try {
        // Get factories.
        pFactory = finder.findFactory
                    (componentID.CID_PORT);
        cFactory = finder.findFactory
                    (componentID.CID_CONN);
    }
    catch (UnknownEx ex) {
        System.out.println (ex.getID () +
                            "not found!");
        System.exit (1);
    }

    // Create two ports and a connection.
    IRoot port1Root = pFactory.create ();
    IRoot port2Root = pFactory.create ();
    IRoot connectionRoot = cFactory.create ();
```

166

注意客户机可以对port1Root和port2Root进行类型转换，而不是调用getExtension()方法，因为组件使用接口继承来实现扩展接口。然而，这种设计会紧密地耦合客户机实现和组件实现。例如，如果后来我们重新构造组件使用Java内嵌类而不是多接口继承，那么所有客户机代码会崩溃。

```
try {
    // Initialize port 1.
    IPort p1 = (IPort) port1Root.getExtension
                    (InterfaceID.ID_PORT);
    p1.setHost ("Machine A");
    p1.setPort (PORT_NUMBER);
    // ...Initialize port 2 and connection...

    // Build array of components.
    IRoot components[] = { c, p1, p2 };
    // Dump all components.
    dumpAll (components);
    // Draw all components.
    drawAll (components);
} catch (UnknownEx error) {
    System.out.println ("Interface "
        +error.getID () + " not supported!");
```

```
            } catch (CommErrorEx commError) {
                System.out.println ("Connection problem");
            }
        }
    }
```

167

❑

8. 已解决的例子

给客户分发基于组件的管理应用控制台不久，TMN框架开发人员收到两个改变请求。第一个请求需要TMN框架中的每个组件从一个持久数据库中加载和存储它的状态。第二个请求需要一个新的具有星型连接拓扑结构的组件。这种拓扑结构表示一系列网络元素都连接到一个中心元素，形成星型的形状。

为满足这些改变请求，TMN框架开发人员可以运用扩展接口模式：

- 为支持从持久数据库中加载和存储组件状态，定义一个新的称为IPersistence的扩展接口：

```
public interface IPersistence extends IRoot {
    public PersistenceId store ();
    public load (PersistenceId persistenceId);
}
```

接着改进每个现有的组件以实现这个接口。详细地说，一个组件实现者必须把定义在新的接口中的所有方法增加到组件实现中。用新的扩展接口扩展一个组件所需的工作量直接依赖于增加给组件的特定扩展接口。持久性例子只需要一些数据库调用来实现新的接口。

- 为支持星型连接拓扑，我们定义一个IConnectionStar接口：

```
public interface IConnectionStar extends IRoot {
    public void setAllPorts (IPort ports[]);
    public void setPort (long whichPort, IPort port);
    public IPort getPort (long whichPort);
}
```

接着所有的TMN框架组件必须实现IRender、IDump、IConnectionStar、IManagedObject和IPersistence接口。

168

用新接口的标识符扩展在实现活动（6.5）中定义的InterfaceID类。

如果客户机需要访问新的功能，那么它可以从组件中获取任何扩展接口，向组件查询新的扩展接口，并且使用新的服务：

```
IRoot iRoot = /* ... */; // use any component interface
try {
    PersistenceId storage = /* ... */;
    IPersistence iPersistence =
        iRoot.getExtension (InterfaceID.ID_PERSISTENCE);
    PersistenceId id = iPersistence.load (storage);
} catch (UnknownEx ue) {
    // Provide exception handling code here when
    // <getExtension> fails to return <IPersistence>.
}
```

9. 变体

扩展对象(Extension Object)[PLoPD3]。在这种变体中不需要组件工厂，因为每个组件负责将接口引用返回给客户，扩展对象非常适合于使用单一的面向对象编程语言（如C++或Java）建造的组件，在那里组件从所有它们实现的接口派生。类型安全的向下类型转换可用于获取组件

接口。在这些与语言有关的实现中，不需要组件工厂，因为组件类直接映射到语言类，而语言类本身就负责实例创建。

分布式扩展接口(Distributed Extension Interface)。这种变体以另外有一类参与者（即服务器）为特征，该服务器驻留组件的实现。每个服务器包含工厂，也包含所有被支持的扩展接口实现。一个服务器可以保持多个组件类型。在分布式系统中，客户机和服务器并不共享同一地址空间。向定位器服务注册和注销它的组件是服务器的任务，这样客户机或者工厂发现者便能获取远程组件。

在分布式系统中，接口和实现存在着物理上的分离。可以引入客户机代理把客户机透明地依附于远程扩展接口[POSA1][GoF95]。客户机端代理实现的扩展接口与它们代表的组件相同。它们也可以把方法调用通过网络转发给远程对象，使客户避免繁琐的和易出错的通信机制。可以定义代理以便客户机能应用上面概述的*扩展对象*(Extension Object)变体。为了改善性能，客户代理能依据半对象加协议(Half Object plus protocol)模式[Mes95]，提供处于同一地点的 [WSU99] 通用扩展接口的本地实现，以减少网络流量。 169

在分布式对象计算中间件[OMG98c]中，可以通过接口定义语言(Interface Definition Language, IDL)编译器自动地实现代理。IDL编译器解析包含接口定义的文件，并生成执行各种网络编程任务（例如列集、散集和错误检查[GS98]）的源代码。接口定义语言的使用简化了用不同编程语言编写的组件和客户机的连接。为确保这种程度的分布性和定位透明性，底层的组件基础设施可以实例化代理者(Broker)体系结构模式[POSA1]。

具有访问控制的扩展接口(Extension Interface With Access Control)。在这种变体中，客户机必须向扩展接口证实自己。这种方法可以限制客户机对扩展接口的访问。例如，可以授权一个管理员访问组件的所有接口，而允许另一客户机调用一个提供特定功能的部分接口的方法。

非对称扩展接口(Asymmetric Extension Interface)。这种变体指定一个特异接口，此接口负责提供对其他所有接口的访问。与对称情况相反，非对称情况中客户不能从一个扩展接口导航到任何其他扩展接口。它们必须转而使用特异扩展接口来导航到任何其他扩展接口。组件自己可以提供这个接口，正如扩展对象变体中定义的那样。

10. 已知使用

微软的COM/COM+技术是基于扩展接口的[Box97]。每个COM类实现必须提供称为`IClassFactory`的工厂接口，此工厂接口定义了实例化新的类实例的功能。当COM运行期激活组件实现时，它获取一个指向相关工厂接口的指针，客户机使用这个接口可以创建新的组件实例。

每个COM类实现一个或多个从称为`IUnknown`的公共根接口派生的接口。`IUnknown`接口包含方法`QueryInterface(REFIID, void**)`，此方法允许客户机获取组件导出的特定扩展接口。`QueryInterface()`的第一个参数是确定哪个扩展接口返回给客户机的惟一标识符。如果组件实现了被客户机请求的接口，那么它在第二个参数中返回一个接口指针，否则返回一个错误。 170

这个活动称为*接口协商*(interface negotiation)，因为客户机可以查询组件以确定它们是否支持特定的扩展接口。COM/COM+实现分布式扩展接口变体，也使得可以用微软支持的任何编程语言开发客户机和组件，这些编程语言包括Visual Basic、C、C++和Java。

CORBA 3[Vin98]引入一个CORBA组件模型（CCM）[OMG99a]，在此模型中每个组件可以提供多个接口。客户机首先获取一个特异接口，即组件的所谓“等价”接口。接着它们使用指定的“提供”方法来导航到一个扩展接口，在CCM中这个扩展接口称为“面”（facet）。每个CCM接口必须实现方法get_component()，此方法与上面介绍的COM的QueryInterface()方法类似。因此总可以从一个面导航回到组件的等价接口。

为获得一个对现有的组件的引用，或者创建一个新的组件，客户机访问一个所谓“home”接口，此接口与一个组件类型相关联。这个接口代表组件工厂接口，正如CORBA组件和Enterprise JavaBeans(EJB)所定义的。由ComponentHomeFinder实现CCM中的工厂发现者，而为了同一目的，EJB依赖于Java命名和目录接口（Java Naming and Directory Interface, JNDI）。CORBA组件和EJB的以Java为核心的子集[MaHa99]使用非对称扩展接口变体。

OpenDoc[OHE96] 引入了使用扩展给对象增加功能的概念。在根接口中提供了获取扩展的功能，也提供了引用计数的功能。OpenDoc实现扩展接口的扩展对象变体。

171

11. 结论

扩展接口模式有下列**优点**：

可扩展性。扩展组件的功能只需增加新的扩展接口，现有的接口保持不变，这样对现有的客户机不会造成不利影响。开发人员可以通过使用多个扩展接口，而不是把所有方法合并到一个基接口来防止*接口膨胀*(interface bloating)。

事务分离。把语义相关的功能分组到独立的扩展接口。通过为每个角色定义一个独立的扩展接口，组件能为相同或者不同的客户机扮演不同角色。

多态性。在不需要继承一个公共接口的情况下支持多态性。如果两个组件实现同一扩展接口，那么某一扩展接口的客户机不必知道哪个组件实际提供功能。类似地，多个组件可以实现相同的接口集，因而允许它们透明地交换组件实现。

将组件和它们的客户机去耦合。客户机访问扩展接口而不是组件实现。因此在组件实现和它的客户机之间不存在（紧）耦合。这样在不破坏现有客户机代码的情况下，可以提供新的扩展接口实现。通过使用代理[POSA1][GoF95]，甚至有可能将组件的接口与组件的实现分开。

支持接口聚合和授权。组件能聚合其他的组件，并像提供自己的接口一样提供被聚合的接口。聚合的接口把所有的客户机请求委托给实现接口的被聚合的组件。这样聚合接口可以作为每个被聚合组件的标识，并且复用它们的代码。然而，这种设计的前提条件是聚合接口组件和组成它的被聚合组件通过getExtension()方法协作。

然而，扩展接口模式也有如下**不足**：

172

增加了组件设计和实现工作。开发和部署组件并非易事。当扩展接口模式没有被透明地集成在某一编程语言时，组件编程工作是特别繁琐的。例如，使用Java或C++实例化本模式是相对直接的。然而，由于缺乏类似继承或者多态这样的关键语言特色，在C中实现本模式就相当复杂。

增加了客户编程复杂性。扩展接口模式需要客户机确定哪个接口适合它们的特殊用况。因此客户机必须在使用扩展接口之前执行多步协议来获得对扩展接口的引用。客户机也必须跟踪大量记录细节，例如接口或者实例标识符和引用计数，这会使客户机的核心应用逻辑难以理解。

附加的间接方法和运行时开销。客户机从不直接访问组件，这会轻微地降低运行时效率。类似地，在多线程或者分布式环境中被初始化的组件的运行时引用计数是复杂的并且潜在地低效率。不过，在一些情况下，这个额外的间接方法是可以忽略的，尤其当跨越高延迟网络访问组件时。

参见

组件和客户机可以不驻留在同一地址空间，可以不用同一编程语言编写或者不以二进制形式部署，但仍有必要使它们互相连接。在这个语境中，可以应用代理(Proxy)模式[POSA1][GoF95]使组件实现与组件接口去耦。对于一个更复杂和更灵活的解决方案，可以运用代理者(Broker)模式[POSA1]。在这种模式中组件作为服务器，而代理者的职责之一是提供全局可用的工厂发现者服务。

在[PLoPD3]中引入了扩展接口模式的扩展对象变体。一旦使用底层编程语言的对象模型实现非分布式组件扩展机制时，就可以应用这种变体。在这种情况下：

- 组件和组件工厂直接映射到编程语言类。
- 组件接口映射到编程语言接口。
- 用类型安全的向下类型转换实现对组件接口的获取。

173

致谢

我们很高兴与Erich合作描述这个模式。Erich在[PLoPD3]中发表了关于扩展对象变体的文章，启发我们在这里对更通用扩展接口模式的文档化。我们也要感谢Don Box，又称为“COM 青年”[Box97]，他为我们提供了很多有关Microsoft COM范型的内幕，并且他在Irvine的California大学的研究生院与Doug Schmidt共事时还指出了网络计算和分布式对象计算的可信任的优点。

174

第3章
事件处理模式

"从已知的事物推测未见的事物
探索事物的内涵
由模式判断整体……
具备了这些天赋大体才可以说形成了经验"
亨利·詹姆斯(1843-1916)——英国作家

本章介绍四种模式，它们描述了如何初始化、接收、多路分解、分配和处理网络化系统中的事件。这四种模式分别是反应器（Reactor）、主动器（Proactor）、异步完成标记（Asynchronous Completion Token）和接受器-连接器（Acceptor-Connector）。 175

事件驱动的体系结构正在越来越深入地应用在网络化软件应用系统中。本章的四种模式有助于简化开发灵活高效的事件驱动应用程序。第一种模式用于开发同步服务提供程序：

- 使用反应器结构化模式，事件驱动的应用可以多路分解并分配从一个或者多个客户机发送给应用的服务请求。反应器模式引入的结构"逆转"了应用中的控制流，这就是所说的好莱坞原则（"不要打电话给我们，我们会打电话给你的"）[Vlis98a]。

  由指定的组件"反应器"，而不是应用程序负责同步等待指示事件，将它们多路分解给负责处理这些事件的事件处理程序，然后在事件处理程序上分配合适的钩子方法。尤其是，由反应器分配对某一事件做出反应的事件处理程序。这样应用程序开发人员只负责实现具体事件处理程序并重用反应器的多路分解和分配机制。

虽然反应器模式的编程和使用相对直观，反应器模式的可用性还是受到了一些限制，特别是它不能同时支持大量的客户机和/或耗时长的客户机请求，因为它在事件多路分解层串行化了所有的事件处理程序的处理过程。本章介绍的第二种模式能帮助运行在有效支持异步I/O的平台上的事件驱动应用减少这种限制：

- 主动器结构模式使事件驱动应用能有效地多路分解和分配由完成的异步操作所触发的服务请求。在不存在不利条件时，它获得了并发的性能优势。

  在主动器模式中，客户机和完成处理程序所代表的应用组件称为主动性实体。和被动地等待指示事件的到达并做出响应的反应器模式不同，主动器模式中的客户机和完成处理程序通过在一个异步操作处理器中主动地初始化一个或者多个异步操作请求，引起应用程序内部的控制流和数据流。 176

  异步操作完成后，异步操作处理器和指定的主动器组件协作，将产生的完成事件（completion event）多路分解给相关的完成处理程序，并分配这些处理程序的钩子方法。完成处理程序处理一个完成事件后，就主动地激活一个异步操作请求。

本章还介绍了另外两种设计模式，它们和前面介绍的两种设计模式联合应用，可以解决更

广泛的事件驱动应用中更具体的问题。

对于用主动器实现任务多路分解的优化，下面一种模式特别有用，因为它考虑到了异步应用设计的一个重要特征：

- 异步完成标记设计模式使应用程序能对它在服务中调用异步操作而引起的响应进行有效地多路分解和处理。

本章的最后一个模式常和反应器模式结合使用，用于网络应用程序中：

- 一旦完成了连接和初始化，接受器-连接器设计模式将网络化系统中同级服务的连接和协作初始化与随后进行的处理分开。接受器-连接器允许应用程序配置它们的连接拓扑结构，进行这种配置不依赖于应用程序所提供的服务。该模式可以置于反应器或者主动器之上，处理那些和建立服务间连接关系有关的事件。

本章介绍的所有四种模式都可以与第5章“并发模式”中介绍的模式结合应用。相关文献中介绍的事件处理模式还包括事件通知（Event Notification）模式[Rie96]、观察者（Observer）模式[GoF95]以及出版者-订阅者（Publisher-Subscriber）模式[POSA1]等。

177

3.1 反应器

事件驱动的应用程序可以使用反应器（Reactor）结构化模式，多路分解并分配从一个或者多个客户机发送给应用的服务请求。

1. 别名 分配器（Dispatcher），通知器（Notifier）

2. 例子

考虑一个用于分布式登录服务的事件驱动服务器。远程客户机应用程序使用该登录服务在一个分布式系统中记录其状态信息。通常这些状态信息包括错误通知、调试跟踪和性能诊断。登录记录被发送到中央登录服务器，该服务器将记录写到不同的输出设备上，如控制台、打印机、文件、网络管理数据库等（如图3-1）。

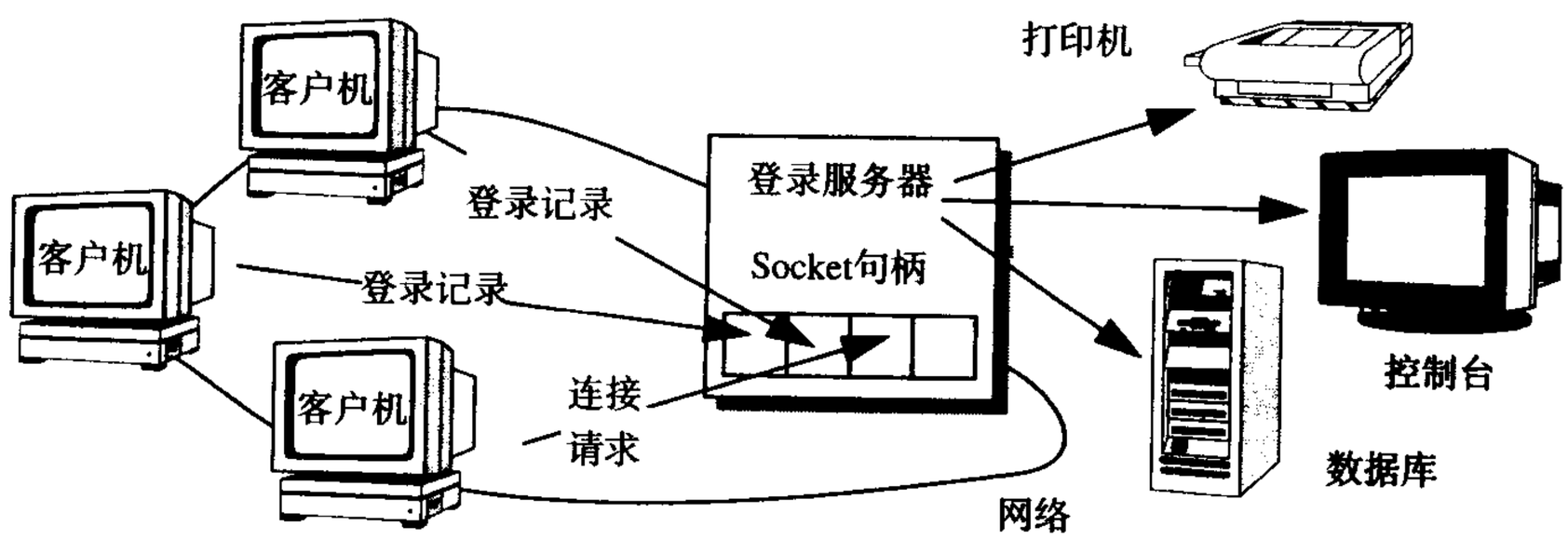


图 3-1

客户机使用面向连接的协议（如TCP[Ste98]）与登录服务器通信。因此客户机和登录服务被绑定到由一个全关联（full association）指定的传输端点（endpoint）上，全关联由IP地址和

TCP端口组成，能惟一标识客户机和登录服务。

多个客户机可以同时访问登录服务器，每个客户机都维持一个与登录服务器的连接。用CONNECT事件表示客户机连向服务器的一个新请求，READ事件表示在登录服务器中处理记录的请求，它会使登录服务器从一个客户机连接中读入新的输入。客户机发出的登录记录和连接请求可以并发地到达登录服务器。 179

实现登录服务器的一个方法是使用一种多线程模型。例如，服务器可以使用“每个连接一个线程”的模型，为每个连接分配指定的控制线程，处理到达后的登录记录。但是，多线程模型有如下不足：

- 由于CPU之间的语境切换、同步和数据移动，使线程方法效率不高，不易扩展。
- 线程方法需要在服务器所有代码中使用复杂的并发控制方案。
- 线程方法并不适用于所有的操作系统，并不是所有的操作系统都提供可移植的线程语义。
- 在制定线程策略来进行并发服务器的优化时，最好根据像CPU的数目这样的可用的资源，而不是根据并发服务的客户机数。

由于上述这些缺点，在对登录服务器进行开发时，多线程模型是一个效率不高和相当复杂的解决方案。而为了确保对所有的连接客户机的服务质量，登录服务器必须高效地和公平地处理请求，特别是不能只为一个客户机服务而忽视别的客户机。

3. 语境

同时接收多个服务请求，并且依次同步地处理它们的事件驱动应用程序。

4. 问题

在分布式系统，特别是服务器[⊖]中的事件驱动应用，必须随时准备同时处理多个服务请求，即使这些请求在应用程序中最终还是被串行处理的。和登录服务器例子中的CONNECT和READ事件一样，用特殊的指示事件标识每个请求的到达。因此，在顺序执行特殊的服务之前，一个事件驱动应用程序必须先将并发到达的指示事件多路分解和分配给相应的服务实现。 180

要有效地解决这些问题，需要考虑以下四个强制条件：

- 为了提高可扩展性和响应性能，应用程序不应该被单个指示事件源所阻塞，而排斥其他的事件源，否则会降低服务器对客户机的响应能力。
- 为了使吞吐率最大，如“例子”一节所介绍的，应避免CPU之间的任何不必要的语境切换、同步和数据移动。
- 新的或改进的服务应该很容易地集成已有的指示事件多路分解和分配机制。
- 应用程序代码应尽量不受多线程和同步机制的复杂性的影响。

5. 解决方案

同步等待多个事件源（如连接的socket句柄）的指示事件的到达。将对事件多路分解以及分配处理事件的服务的机制进行集成；将事件多路分解和分配机制从服务中对指示事件的与应用有关的处理分离出去。

⊖ “已知的使用”一节列出了一些反应器模式的已知使用，其中反应器模式用于实现应用程序的事件处理，这些应用程序同时起着客户机和服务器的作用。

细节：对于应用程序提供的每个服务，引入一个单独的事件处理程序处理某一事件源的某一类型的事件。在反应器中注册所有的事件处理程序，反应器使用一个同步事件多路分解器等待一个或多个事件源的指示事件。发生指示事件后，同步事件多路分解器通知反应器，后者同步地分配与事件相关的事件处理程序，以便这些事件处理程序可以执行请求的服务。

6. 结构

反应器模式中有5个主要组成部分：

操作系统提供句柄（handle）来识别像网络连接或打开文件这样的事件源，事件源产生指示事件并对它们进行排队。指示事件可以来自于外部，如由客户机发送给服务的CONNECT事件或者READ事件，也可以来源于内部，如超时事件。一旦一个事件源产生了一个指示事件，该指示事件就被送入相关句柄的队列中，句柄被标记为“就绪”。这样，就可以在句柄中执行一个像accept()或者read()这样的操作，而不会阻塞调用线程。

181

➡在登录服务器中使用socket句柄来标识一个接收CONNECT或者READ指示事件的传输端点。一个被动模式的传输端点和其相应的socket句柄侦听CONNECT指示事件。这样登录服务器为每个连接的客户机维护一个单独的连接，也就维护了一个单独的socket句柄。 ❑

同步事件多路分解器（synchronous event demultiplexer）是一个函数。调用该函数，可以等待发生在一组句柄（句柄集）上的一个或多个指示事件。该函数在开始一直阻塞着，直到句柄集上的指示事件通知同步事件多路分解器“该句柄集上的一个或多个句柄变为‘就绪’”，这就意味着在没有阻塞的情况下可以启动一个对句柄的操作。

➡select()是一个用于I/O事件[Ste98]的常用同步事件多路分解器函数，包括UNIX和Win32平台的很多操作系统都支持它。select()调用指明句柄集中哪些句柄存在待处理的指示事件。在不阻塞调用线程的情况下，可以在这些句柄上同步地调用操作。 ❑

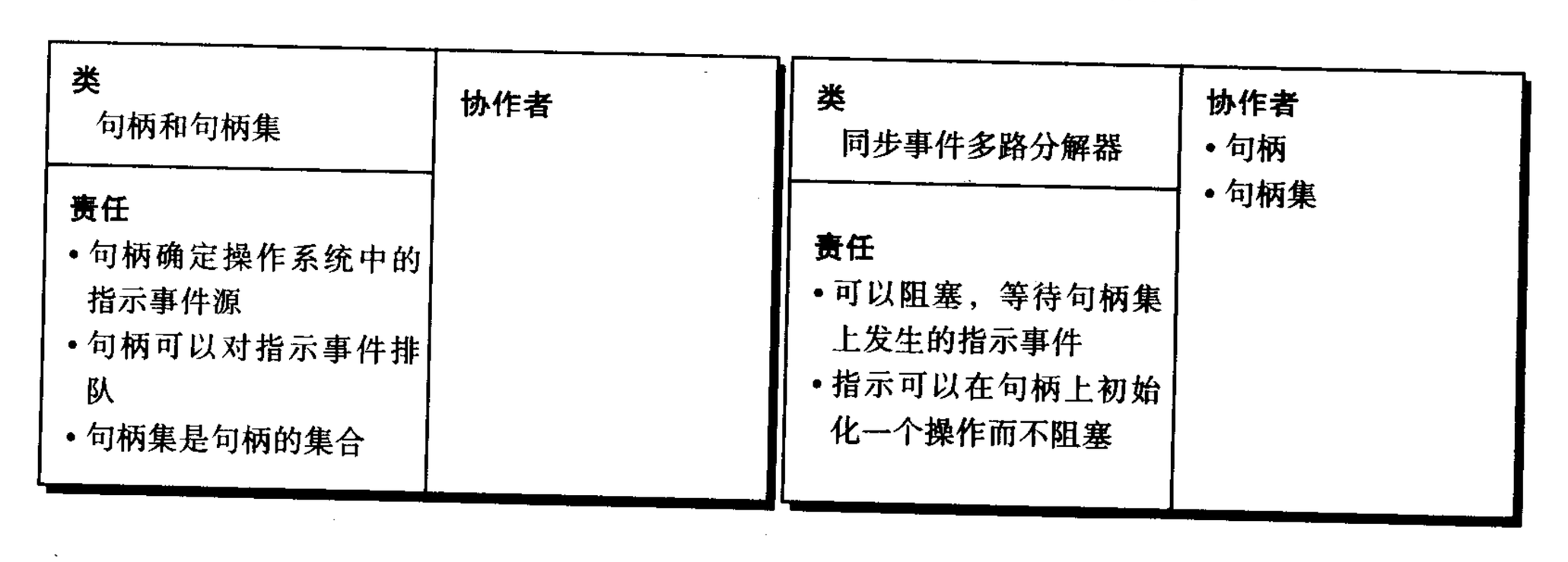


图 3-2

事件处理程序定义一个由一个或多个钩子方法组成的接口[Pree95][GoF95]。钩子方法代表操作，这些操作可用于处理与应用有关的发生在与某事件处理程序相关联的句柄上的指示事件。

182

具体事件处理程序（concrete event handler）是实现应用程序所提供的特定服务的事件处理

程序。每个具体事件处理程序都和一个句柄相关，句柄决定应用程序中的服务。特别是具体事件处理程序实现了钩子方法，这些钩子方法负责处理通过对应的句柄接收来的指示事件。将服务结果写到句柄上，并返回给调用者（如图3-3）。

➡登录服务器包括两类具体事件处理程序：登录接受器和登录处理程序。登录接受器使用接受器-连接器模式建立和连接登录处理程序。登录处理程序负责接收和处理与之连接的客户机发来的登录服务请求。 ❑

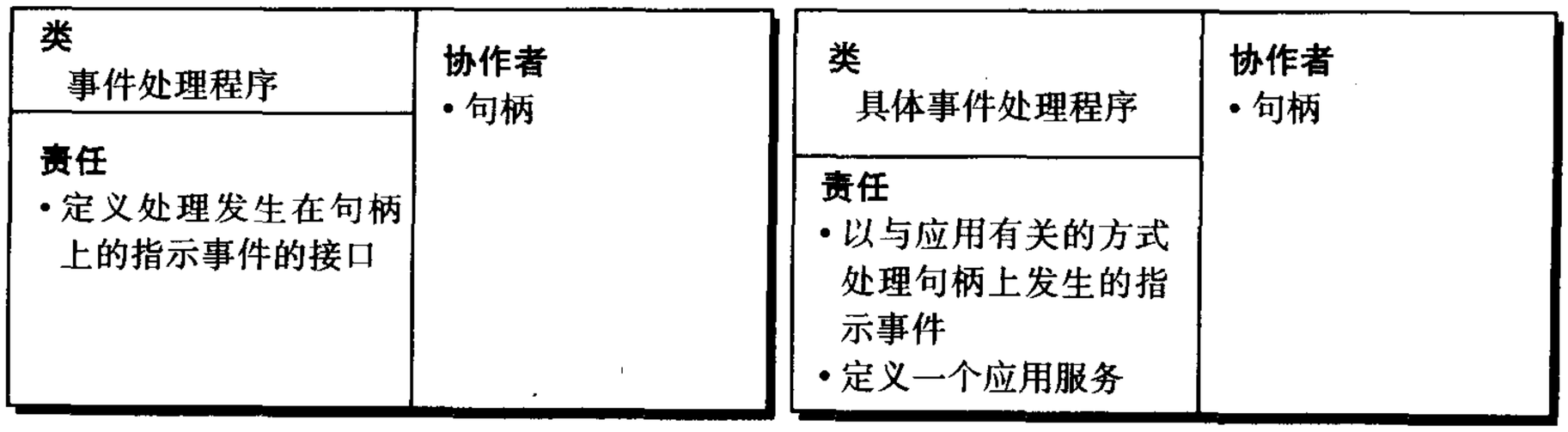


图 3-3

反应器(reactor)定义了一个接口，允许应用程序注册或删除事件处理程序及其相应的句柄，并运行应用程序的事件循环。反应器使用同步事件多路分解器等待在句柄集上发生指示事件。当有指示事件时，反应器首先从发生指示事件的句柄上将每个指示事件多路分解给相应的事件处理程序，然后在句柄上分配合适的钩子方法以处理这些事件（如图3-4）。

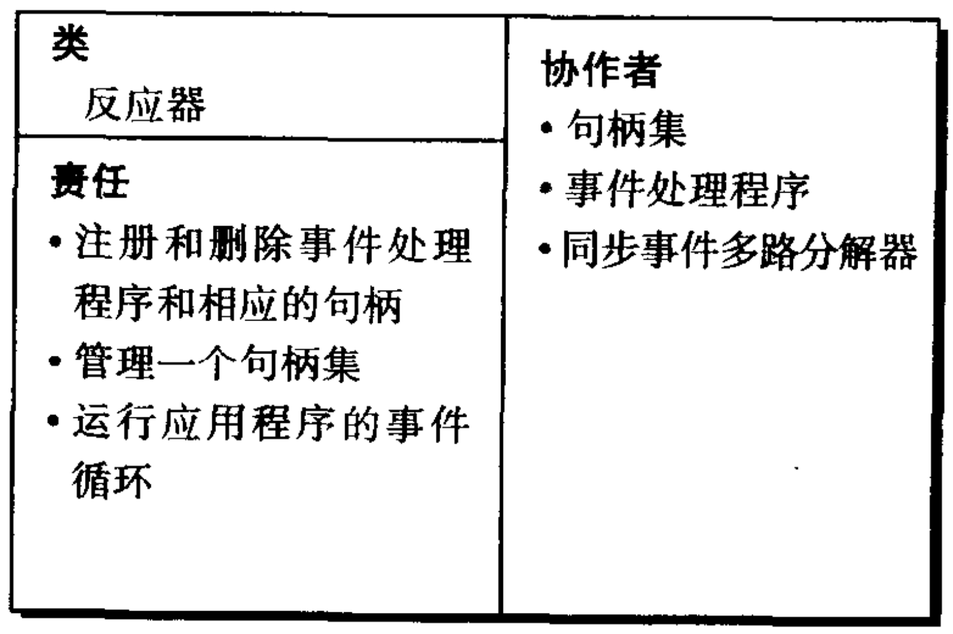


图 3-4

183

要留意反应器模式引入的结构是如何将应用中的控制流“倒置”的。等待指示事件，多路分解这些事件给它们的具体事件处理程序；向具体事件处理程序分配相应的钩子方法，所有这些是反应器而不是应用程序的责任。特别是，具体事件处理程序并不调用反应器，而是由反应器分配一个具体事件处理程序，具体事件处理程序对某个指定的事件的发生做出反应。这种“控制逆转”又称为“好莱坞原则”[Vlis98a]。

因此，应用程序开发者只要负责实现具体事件处理程序，并在反应器上注册它们，应用程序就可以简单地重用反应器的多路分解和分配机制了。

类图3-5说明了反应器模式中各组成部分之间的结构：

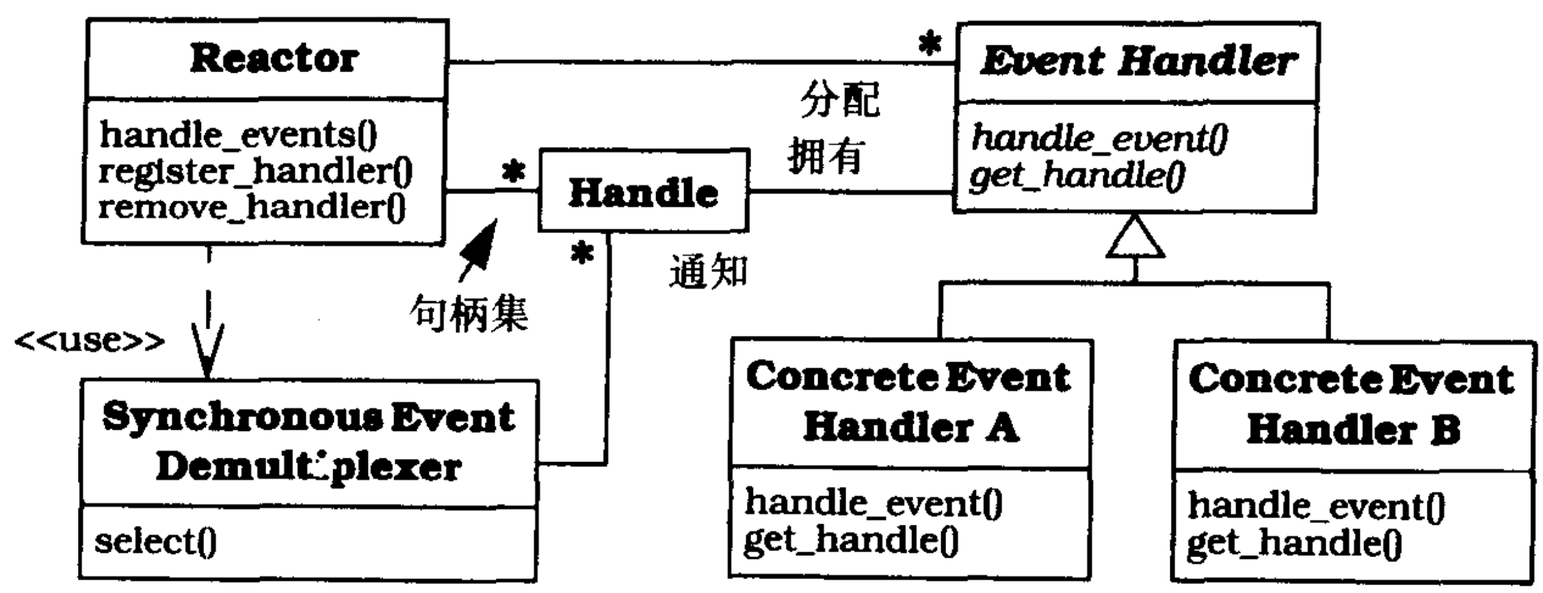


图 3-5

7. 动态特性

反应器模式中的协作过程说明了控制流如何在反应器和事件处理程序之间振荡（如图3-6）：

- 应用程序在反应器上注册一个具体事件处理程序。此时，应用程序还要指定指示事件的类型，事件处理程序希望当这种类型的事件在相应的句柄上发生时反应器能通知它。
- 反应器要求每个事件处理程序提供它们的内部句柄，在这里的例子中是通过调用get_handle()方法实现的，该句柄将同步事件多路分解器和操作系统都视为指示事件的来源。
- 注册了全部事件处理程序后，应用程序开始进入名为handle_events()的反应器事件循环。从此，反应器将每个注册的事件处理程序的句柄组合成一个句柄集，然后调用同步事件多路分解器等待发生在句柄集上的指示事件。
- 当相应事件源的一个或者几个句柄变成“就绪”（例如当一个socket准备好读）时，同步事件多路分解器函数返回到反应器。
- 然后反应器用“就绪”的句柄作为关键字找出合适的事件处理程序并分配相应的钩子方法。可以将发生的指示事件的类型作为参数传递给钩子方法。钩子方法使用该类型信息进行任何其他的与应用有关的多路分解和分配操作[⊖]。
- 分配了事件处理程序内相应的钩子方法后，该钩子方法处理所调用的服务。该服务可以将处理的结果（如果有的话）写到与事件处理程序相关的句柄上，以便能够将结果返回给原先请求服务的客户机。

184

185

8. 实现

反应器模式的各个部分分成两层：

- *多路分解/分配基础设施层的组件*。本层执行一个通用的、与应用无关的策略，用于将指示事件多路分解到事件处理程序，然后分配相应的事件处理程序钩子方法。
- *应用层组件*。本层定义具体事件处理程序，在具体事件处理程序的钩子方法中进行与应用有关的处理。

本节介绍的实现活动，首先涉及到通用多路分解/分配基础设施层的组件，然后涉及到应用组件。这里主要考虑反应器的实现，该反应器在单个控制线程内多路分解句柄集、向事件处理程序分配钩子方法。“变体”一节介绍与开发并发反应器有关的活动。

⊖ “实现”一节中介绍了另一种分配方法。

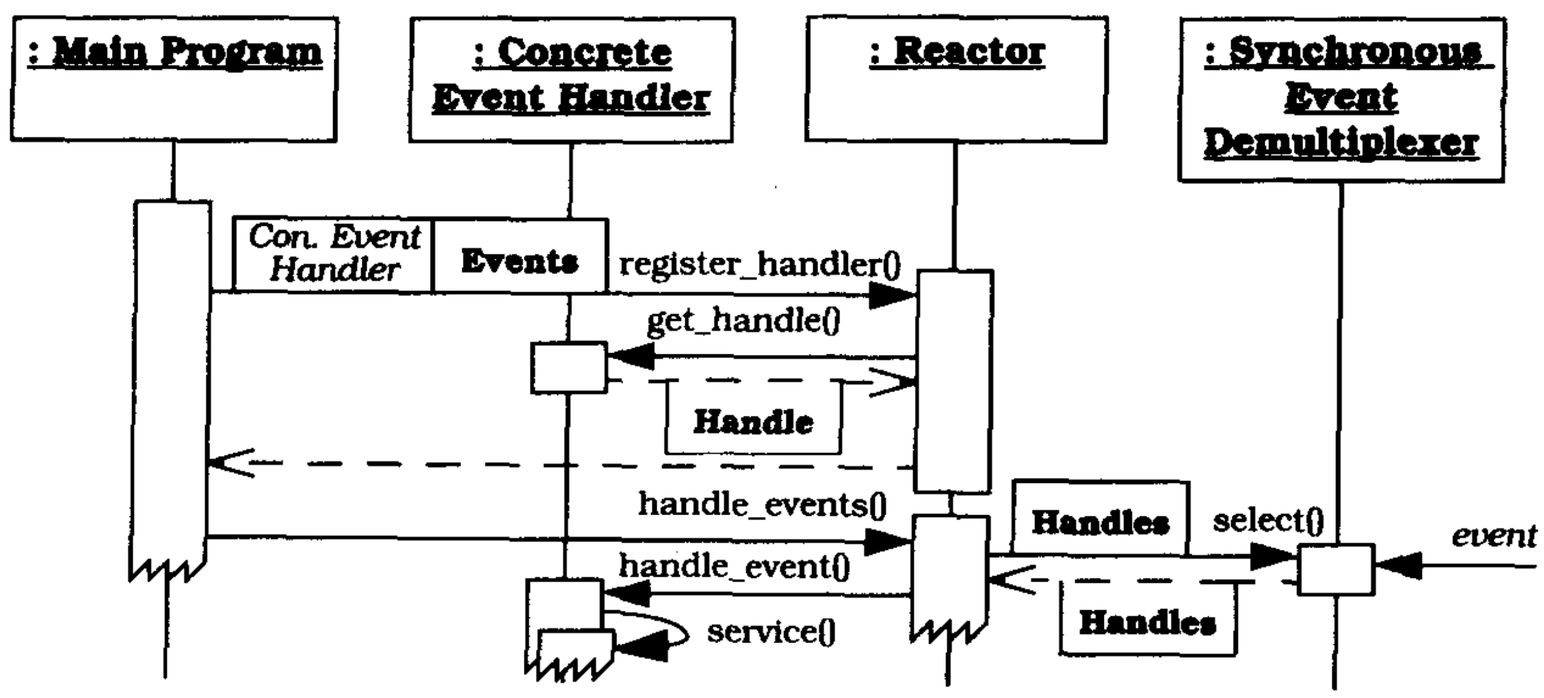


图 3-6

(1) 定义事件处理程序接口。事件处理程序具体指定了由一个或几个钩子方法组成的接口[Pree95]。这些钩子方法代表了处理由反应器接收并分配的指示事件时可用的服务集。正如实现活动（5）中所介绍的那样，应用程序开发者要创建具体事件处理程序，对某一指示事件执行相应的服务。定义一个事件处理程序接口由两个子活动构成：

(1.1) 确定分配目标的类型。为实现反应器的分配策略，一个句柄可以有两类事件处理程序。

- 事件处理程序对象（event handler object）。在面向对象的应用中，将事件处理程序与句柄结合起来的一个常用的方法是建立一个事件处理程序对象。例如，“结构”一节中实现的反应器模式分配具体事件处理程序对象。用一个对象作为分配目标，可以很方便地派生子类事件处理程序，以重用和扩展已有的组件。类似地，对象可以很容易地在一个组件中集成服务的状态和方法。
- 事件处理程序函数（event handler function）。将事件处理程序与句柄结合起来的另一个策略是向反应器而不是对象注册一个指向函数的指针。使用函数指针作为分配目标，不用定义继承事件处理程序基类的新的子类，就可以很方便地注册回调函数。 186

可以使用适配器模式[GoF95]同时支持对象和函数指针。例如，可以使用事件处理程序对象定义一个适配器，该事件处理程序对象中有指向事件处理程序函数的指针。当在事件处理程序适配器对象中调用钩子方法时，钩子方法可以向它封装的事件处理程序函数自动转发调用。

(1.2) 确定事件处理程序分配接口策略。下一步必须定义事件处理程序时支持的接口类型。假设使用事件处理程序对象而不用函数指针，这里有两种大致的策略：

- 单方法分配接口策略（single-method dispatch interface strategy）。图3-5说明了Event_Handler基类接口的实现方法，该接口只包括一个事件处理方法。反应器用此方法分配事件。在这种情况下，将所发生事件的类型作为参数传递给方法。

➡下面描述了一个C++抽象基类，以说明单方法接口的用法。首先有一个类型定义和枚举说明，在单方法和多方法分配接口策略中都要用到。

```
typedef unsigned int Event_Type;
enum {
    // Types of indication events.
```

```
    READ_EVENT = 01,    // ACCEPT_EVENT aliases READ_EVENT
    ACCEPT_EVENT = 01, // due to <select> semantics.
    WRITE_EVENT = 02, TIMEOUT_EVENT = 04,
    SIGNAL_EVENT = 010, CLOSE_EVENT = 020
    // These values are powers of two so
    // their bits can be "or'd" together efficiently.
};
```

接着给出Event_Handler类的说明：

```
class Event_Handler { // Single-method interface.
public:
    // Hook method dispatched by <Reactor> to handle
    // events of a particular type.
    virtual void handle_event (HANDLE handle,
                               Event_Type et) = 0;

    // Hook method that returns the I/O <HANDLE>.
    virtual HANDLE get_handle () const = 0;
protected:
    // Virtual destructor is protected to ensure
    // dynamic allocation.
    virtual ~Event_Handler ();
};
```

187

❑

使用单方法分配接口策略，就可以在不改变类接口的情况下，支持新的指示事件类型。但是该策略要求在具体事件处理程序的handle_event()方法中大量使用C++的switch和if语句来处理特定事件，从而降低了可扩展性。

- *多方法分配接口策略*。定义Event_Handler分配接口的另一个策略是为处理每一类事件（如输入事件、输出事件或超时事件）创建一个单独的钩子方法。这一策略比单方法分配接口策略更具有可扩展性，因为是由反应器实现而不是由具体事件处理程序的handle_event()方法来进行多路分解的。

➡下面的C++抽象基类是一个多方法接口：

```
class Event_Handler {
public:
    // Hook methods dispatched by a <Reactor> to handle
    // particular types of events.
    virtual void handle_input (HANDLE handle) = 0;
    virtual void handle_output (HANDLE handle) = 0;
    virtual void handle_timeout (const Time_Value &) = 0;
    virtual void handle_close (HANDLE handle,
                               Event_Type et) = 0;
    // Hook method that returns the I/O <HANDLE>.
    virtual HANDLE get_handle () const = 0;
};
```

❑

使用多方法分配接口策略，可以很容易地有选择地重载基类中的方法，从而避免了在钩子方法实现中通过switch和if语句进行的其他多路分解。但是这种策略要求模式的实现者提前预测事件处理程序方法。上面的Event_Handler分配接口中不同的handle_*()方法是专门针对用于I/O和超时的指示事件的，select()函数支持这些指示事件。该函数并不包括所有的指示事件类型，如在Win32平台中可以用WaitForMultipleObjects()函数处理的同步事件[SchSt95]。

188

单方法分配接口策略和多方法分配接口策略都是钩子方法（Hook Method）模式[Pree95]和模板方法（Template Method）模式[GoF95]的具体实现。它们的目的是提供定义良好的钩子，不同的应用程序可以对钩子进行具体化，并由低层分配代码回调。这样应用程序员可以使用继承和多态性来定义具体事件处理程序。

(2) *定义反应器接口*。应用程序使用反应器接口注册或者删除事件处理程序及相应的句柄，并调用应用程序的事件循环。通常用单件[GoF95]访问反应器接口，因为一个应用进程有一个反应器就足够了。

为了将应用程序和不同操作系统平台所具有的复杂的、不可移植的多路分解和分配机制分开，反应器模式可以使用桥（Bridge）模式[GoF95]。反应器接口对应于桥模式中的抽象参与者（abstraction participant），而按照桥模式中的实现层次，在内部通过指针可以访问与平台有关的反应器实例。

➡在登录服务器中的反应器接口定义了对注册和删除事件处理程序以及反应性运行应用程序的事件循环的抽象说明：

```
class Reactor {
public:
    // Methods that register and remove <Event_Handler>s
    // of particular <Event_Type>s on a <HANDLE>.
    virtual void register_handler
        (Event_Handler *eh, Event_Type et) = 0;
    virtual void register_handler
        (HANDLE h, Event_Handler *eh, Event_Type et) = 0;
    virtual void remove_handler
        (Event_Handler *eh, Event_Type et) = 0;
    virtual void remove_handler
        (HANDLE h, Event_Type et) = 0;

    // Entry point into the reactive event loop. The
    // <timeout> can bound time waiting for events.
    void handle_events (Time_Value *timeout = 0);
    // Define a singleton access point.
    static Reactor *instance ();
private:
    // Use the Bridge pattern to hold a pointer to
    // the <Reactor_Implementation>.
    Reactor_Implementation *reactor_impl_;
};
```

189

❑

通常一个典型的反应器接口还要定义两个重载的register_handler()方法，使应用程序运行时能在反应器的内部多路分解表（在实现活动3.3中介绍）中注册句柄和事件处理程序。一般来说，注册事件处理程序的方法具有下面的两个特征之一，或者两者都有：

- *两个参数*。在这种设计中，一个参数指明事件处理程序，而另一个参数指明指示事件的类型，该类型是注册的事件处理程序能处理的指示事件的类型。该方法的实现要用“双分配（double-dispatching）”方法[GoF95]，通过回调事件处理程序方法get_handle()得到句柄。这种设计方法的好处是不会偶尔地将“错误”的句柄和某一个事件处理程序关联起来。

➡下面的代码段说明在register_handler()的实现中如何使用“双分配”：

```
void Select_Reactor_Implementation::register_handler
        (Event_Handler *event_handler,
         Event_Type event_type) {
```

```
    // Double-dispatch to obtain the <HANDLE>.
    HANDLE handle = event_handler->get_handle ();
    // ...
}
```

❑

- 三个参数。在这个设计中用第三个参数显式地传递句柄。尽管这种方法比两参数方法更容易出错，但可以使应用程序能为多个句柄注册同一个事件处理程序，从而有助于节省内存空间。

两种注册方法都要将其参数存入相应的由句柄指示的多路分解表中。

反应器接口也定义了其他两个重载的remove_handler()方法，用于从反应器中删除事件处理程序。例如，可能一个应用程序不再需要在某一句柄上处理一个或者多个类型的指示事件。这些方法将事件处理程序从反应器内部的多路分解表中删除，使它不再为任何类型的指示事件服务。可以和事件处理程序注册方法一样向删除事件处理程序的方法传递一个句柄或者一个事件处理程序。

190

反应器接口还定义了主入口点方法handle_events()，应用程序利用它反应性地运行事件循环。该方法调用同步事件多路分解器等待句柄集上发生指示事件。应用程序可以使用timeout参数限制等待指示事件的时间，这样当没有事件发生时应用程序不会无限地阻塞。

一旦句柄集上发生一个或多个指示事件，同步事件多路分解器函数就返回。此时handle_events()方法做出“反应”，将指示事件多路分解到与每个句柄相关的已就绪的事件处理程序。然后分配处理程序的钩子方法来处理这些事件。

(3) *实现反应器接口*。要实现在实现活动(2)中定义的反应器接口，需要以下4个子活动：

(3.1) *开发反应器的实现层次*。在实现活动(2)中介绍的反应器接口抽象地将所有的多路分解和分配处理交给反应器实现，后者担当了桥模式[GoF95]中实现层次的角色。使用这种设计方法，有可能透明地实现和配置不同类型的反应器。例如，可以使用不同类型的同步事件多路分解器，如select()[Ste98]、poll()[Rago93]或WaitForMultipleObjects()[Sol98]，创建一个具体的反应器实现。每种同步事件多路分解器的特征和限制在实现活动(3.2)中有描述。

➡在登录服务器的例子中，将类Reactor_Implementation定义成反应器的层次结构的基类。在这里不给出对它的声明，因为该类和实现活动(2)中的Reactor接口基本上是一样的。二者主要的差别是：它的方法是纯虚的(pure virtual)，因为它在具体反应器实现的层次结构中是一个基类。❑

191

(3.2) *选择一种同步事件多路分解器机制*。反应器实现通过调用同步事件多路分解器等待反应器句柄集上发生的一个或者多个指示事件。当句柄集中的任何一个句柄“准备就绪”时调用返回，表明不用阻塞应用进程就可以调用句柄上的操作。同步事件多路分解器、句柄以及句柄集通常是操作系统已有的机制，不需要反应器的实现者另外开发。

➡在登录服务器的例子中，将select()函数作为同步事件多路分解器，事件驱动的反应性应用程序就可以使用这个函数在依应用而定的时间段内等待多个I/O句柄上发生不同类型的I/O事件：

```
int select (u_int max_handle_plus_1,
            fd_set *read_fds, fd_set *write_fds,
            fd_set *except_fds,timeval *timeout);
```

select()函数检查三个“文件描述集”（fd_set）参数，这三个参数的地址由read_fds、write_fds和except_fds传递，以判断它们的句柄是否“读就绪”、“写就绪”或者有一个“异常条件”。三个文件描述符集参数的句柄值合在一起，组成反应器模式中的句柄集成员。

在一次调用中，select()函数可以向调用者返回多个“就绪”句柄。然而，不能由多个控制线程并发地在同一句柄集上调用select()函数，因为这样的话，当句柄的同一子集上有I/O事件时，操作系统会错误地通知调用select()函数的多个线程[Ste98]。另外，当句柄集很大时，select()函数的性能会下降[BaMo98]。 ❑

另外两个被一些操作系统支持的同步事件多路分解器是poll()和WaitForMultipleObjects()函数。这两个函数和select()函数一样存在扩展性的问题。可移植性也不好，因为只在与Win32或者System V Release 4 UNIX系统兼容的系统中可用。“变体”一节介绍了WaitForMultipleObjects()独有的特性，由于有这种特性，多个控制线程可以在同一个句柄集上并发地调用该函数。 192

(3.3) *实现一个多路分解表*。除了调用同步事件多路分解器等待句柄集发生指示事件之外，反应器实现还要维护一个多路分解表。该表是一个管理者，包含一个格式为<句柄，事件处理程序，指示事件类型>的三元组表[Som97]。反应器实现使用句柄作为关键字和多路分解表中的事件处理程序建立关联。该表也存储了事件处理程序注册到句柄上的指示事件的类型（如CONNECT和READ）。

可以使用不同的搜索策略实现多路分解表，如直接索引、线性搜索或者动态散列查找。如果像UNIX一样用连续范围内的整数表示句柄，那么直接索引是最有效的，因为可以在常数时间$O(1)$内定位多路分解表组项的位置。

对于像Win32这样的平台，由于其中的句柄是非连续指针，直接索引就不可行了。因此必须使用某种线性搜索或者散列查找实现多路分解表。

➥UNIX中的I/O句柄是连续的整数值，这样可以用一个有固定大小的struct数组实现多路分解表。在这种设计中，句柄值本身就是多路分解表的数组的直接索引项，并可在常数时间内找到事件处理程序或事件注册类型。下面的类将HANDLE映射到Event_Handler和Event_Type，就是这种实现：

```
class Demux_Table {
public:
    // Convert <Tuple> array to <fd_set>s.
    void convert_to_fd_sets (fd_set &read_fds,
                             fd_set &write_fds,
                             fd_set &except_fds);

    struct Tuple {
        // Pointer to <Event_Handler> that processes
        // the indication events arriving on the handle.
        Event_Handler *event_handler_;

        // Bit-mask that tracks which types of indication
        // events <Event_Handler> is registered for.
        Event_Type event_type_;
    };
```

193

```
    // Table of <Tuple>s indexed by Handle values. The
    // macro FD_SETSIZE is typically defined in the
    // <sys/socket.h> system header file.
    Tuple table_[FD_SETSIZE];
};
```

在这个简单的实现中，用UNIX的I/O句柄值对Demux_Table中的table_数组进行索引，I/O句柄值的范围是0~FD_SETSIZE-1。自然地，一个更具有移植性的解决方案应该是用一个包装器外观封装与UNIX有关的实现细节。 ❑

(3.4) 定义反应器的具体实现。正如实现活动(2)中所介绍的，反应器接口中有一个指向反应器具体实现的指针，并将所有的方法调用传给该指针。

➡在这里的反应器具体实现使用select()作为同步事件多路分解器，用Demux_Table类作为多路分解表。它继承Reactor_Implementation类，并重载了其纯虚方法：

```
class Select_Reactor_Implementation :
    public Reactor_Implementation {
public:
```

handle_events()方法定义了进入Select_Reactor_Implementation反应性事件循环的入口：

```
void Select_Reactor_Implementation::handle_events
    (Time_Value *timeout = 0) {
```

该方法首先将Demux_Table元组转换为可以传递给select()的fd_set句柄集：

```
fd_set read_fds, write_fds, except_fds;

demux_table_.convert_to_fd_sets
    (read_fds,write_fds,except_fds);
```

下一步，调用select()函数等待发生在句柄集上的指示事件，timeout为最多等待时间。

```
HANDLE max_handle = // Max value in <fd_set>s.
int result = select
    (max_handle + 1,
     &read_fds, &write_fds, &except_fds,
     timeout);

if (result <= 0)
    throw /* handle error or timeout cases */;
```

194

最后，在句柄集上迭代，为因发生指示事件而句柄变为“就绪”的事件处理程序分配钩子方法：

```
for (HANDLE h = 0; h <= max_handle; ++h) {
    // This check covers READ_ + ACCEPT_EVENTs
    // because they have the same enum value.
    if (FD_ISSET (&read_fds, h))
        demux_table_.table_[h].event_handler_->
            handle_event (h, READ_EVENT);

    // ... perform the same dispatching logic for
    // WRITE_EVENTs and EXCEPT_EVENTs ...
}
```

为了简单起见，这里省略了反应器其他方法的实现细节，例如用于注册和注销事件处理程序的方法的实现细节。

反应器类的私有部分中有一个事件处理程序多路分解表：

```
private:
    // Demultiplexing table that maps <HANDLE>s to
    // <Event_Handler>s and <Event_Type>s.
    Demux_Table demux_table_;
};
```

注意，上述实现只能在那些用连续的无符号整数表示I/O句柄的操作系统平台上运行，如UNIX。在一些句柄是非邻接的指针的平台（如Win32）上，实现这一模式要求另外的数据结构以跟踪哪些句柄在使用。 ❑

(4) *确定应用程序所需要的反应器数量*。对于许多应用程序，可以使用反应器模式的单个实例进行构造。在这种情况下可以像实现活动（2）一样，使用单件模式[GoF95]实现反应器。该模式可用于在应用程序内部将事件的多路分解和分配集中于一个反应器实例中。

然而，有些操作系统对在单个控制线程内能等待的句柄数量有限制。在Win32中，在一个线程内`WaitForMultipleObjects()`最多等待64个句柄。在这种情况下，为了开发可扩展的应用程序，需要创建多个线程，每个线程运行自己的反应器模式实例。 195

对于某些实时应用程序，将一个反应器分配给一个线程也是很有用的[SMFG00]。例如，不同的反应器与具有不同优先级的线程相关联。这种设计为不同类型的同步操作处理指示事件提供了不同等级的服务质量。

注意，事件处理程序只在反应器模式的实例中是串行化的。因此多线程中的多个事件处理程序是并行工作的。如果不同线程中的事件处理程序并发地访问共享的状态，这种配置还需要使用额外的同步机制。“变体”一节介绍了为实现反应器和事件处理程序而增加并发控制的技术。

(5) *实现具体事件处理程序*。具体事件处理程序由实现活动（1）中描述的事件处理程序接口派生，用于定义与应用有关的功能。实现具体事件处理程序需要考虑三个活动。

(5.1) *确定维护具体事件处理程序状态的策略*。一个事件处理程序需要维护与特定请求有关的状态信息。在多路分解器例子中，当操作系统通知登录服务器由于有传输层的流控制而只从socket中读了部分登录信息时，需要维护状态信息。这样，具体事件处理程序就需要将登录数据片段保存在缓冲区中，返回到反应器事件循环，等待余下记录到达的通知。因此，具体事件处理程序必须记录已读的字节数，以便能正确地追加后续数据。

(5.2) *实现用一个句柄配置各具体事件处理程序的策略*。具体事件处理程序对句柄执行有关操作。用事件处理程序配置句柄的两个策略是：

- *硬编码*（hard-coded）。该策略将句柄或者句柄的包装器外观硬编码到具体事件处理程序中。这种方法很容易实现，但是如果对于不同的用况必须将不同类型的句柄或IPC机制配置到事件处理程序中时，重用性不好。 196

➡ “已解决的例子”一节中介绍了`SOCK_Acceptor`和`SOCK_Stream`类，它们被硬编码到了登录服务器组件中。这两个类是在包装器外观模式的“实现”一节中定义的包装器外观。它们在一个可移植的类型安全的面向对象接口中封装了socket句柄的流式Socket语义。在因特网领

域，用TCP实现流Socket。 ❑

- 类属（generic）方法。一个更通用的策略是在模板化事件处理程序类中以类型为参数来实例化包装器外观。这种方法能产生更灵活和可重用的事件处理程序，虽然在总使用一种句柄类型或者IPC机制的情况下，这种通用性是不必要的。

➡在接受器-连接器模式的“实现”一节中介绍的Acceptor、Connector和Service_Handler类是用包装器外观实例化的模板。 ❑

(5.3) 实现具体事件处理程序的功能。应用程序开发者必须确定当反应器实现调用一个服务相应的钩子方法时，为实现该服务所要进行的处理动作。为了将建立连接的功能和其后的服务处理分开，可以按照接受器-连接器模式将具体事件处理程序分成几个子类。特别是，服务处理程序实现了与应用有关的服务，而可重用的接受器和连接器分别根据这些服务处理程序主动和被动地建立连接。

9. 已解决的例子

登录服务器使用用select()同步事件多路分解器和两个具体事件处理程序（登录接受器、登录连接器）实现的单件反应器。登录接受器和登录连接器分别接受客户机的连接请求和处理登录请求。在讨论基于单方法分配接口策略的两个具体事件处理程序的实现之前，首先用两个场景说明登录服务器的一般特征。

197

第一个场景说明客户机连接到登录服务器时要采取的步骤（如图3-7）：

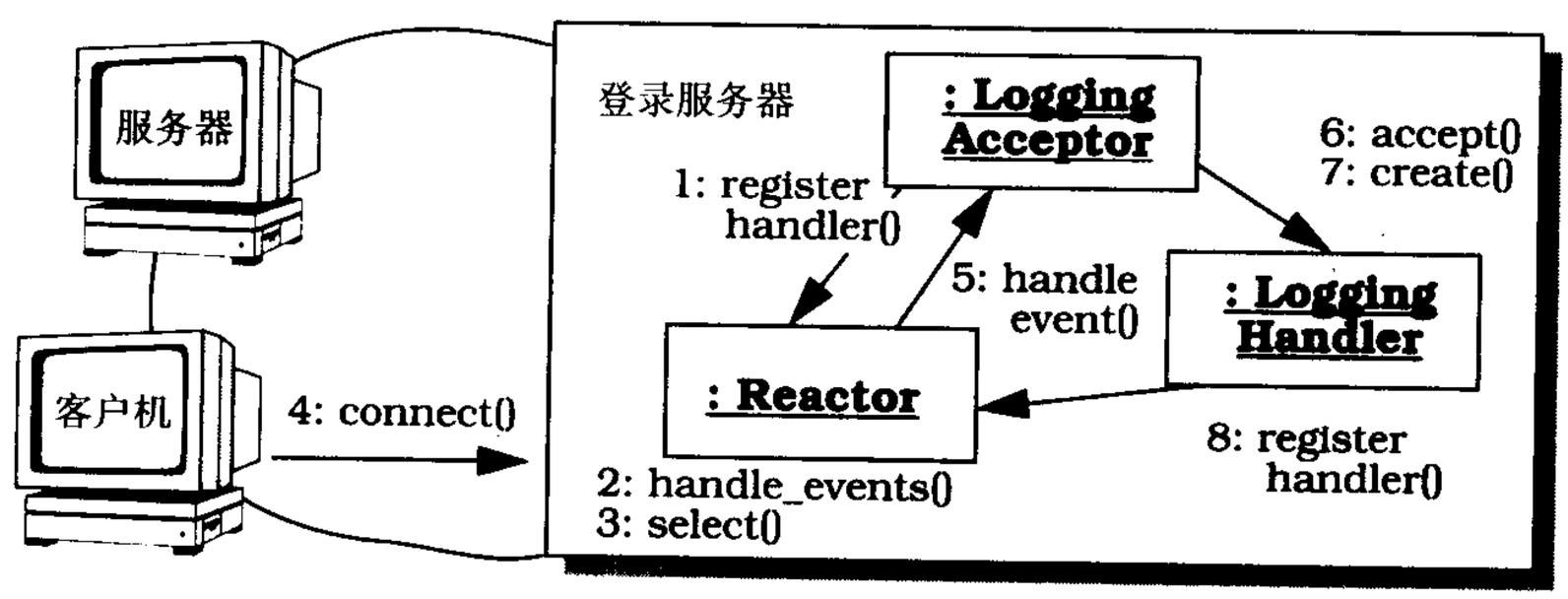


图 3-7

- 登录服务器首先向反应器注册登录接受器，以处理与客户机连接请求有关的指示事件（1）。然后登录服务器调用反应器单件的事件循环方法（2）。
- 反应器单件调用同步事件多路分解器select()操作，等待连接指示事件或登录数据指示事件的到达（3）。此后服务器上的所有的处理都由事件处理程序的反应性多路分解和分配所驱动。
- 客户机向登录服务器发送连接请求（4），使反应器单件分配登录接受器的handle_event()钩子方法（5），以通知它到达了一个新的连接指示事件。
- 登录接受器接受新的连接（6），产生一个登录处理程序为新客户机服务（7）。
- 登录处理程序向反应器单件注册socket句柄（8），并指示反应器在接收到一个表明“Socket已准备好读”的指示事件时通知它。

客户机连接好后，使用在第6步连接的socket句柄向服务器发送登录记录。 198

第二个场景说明反应性登录服务器为一条登录记录服务时要采取的步骤（如图3-8）：

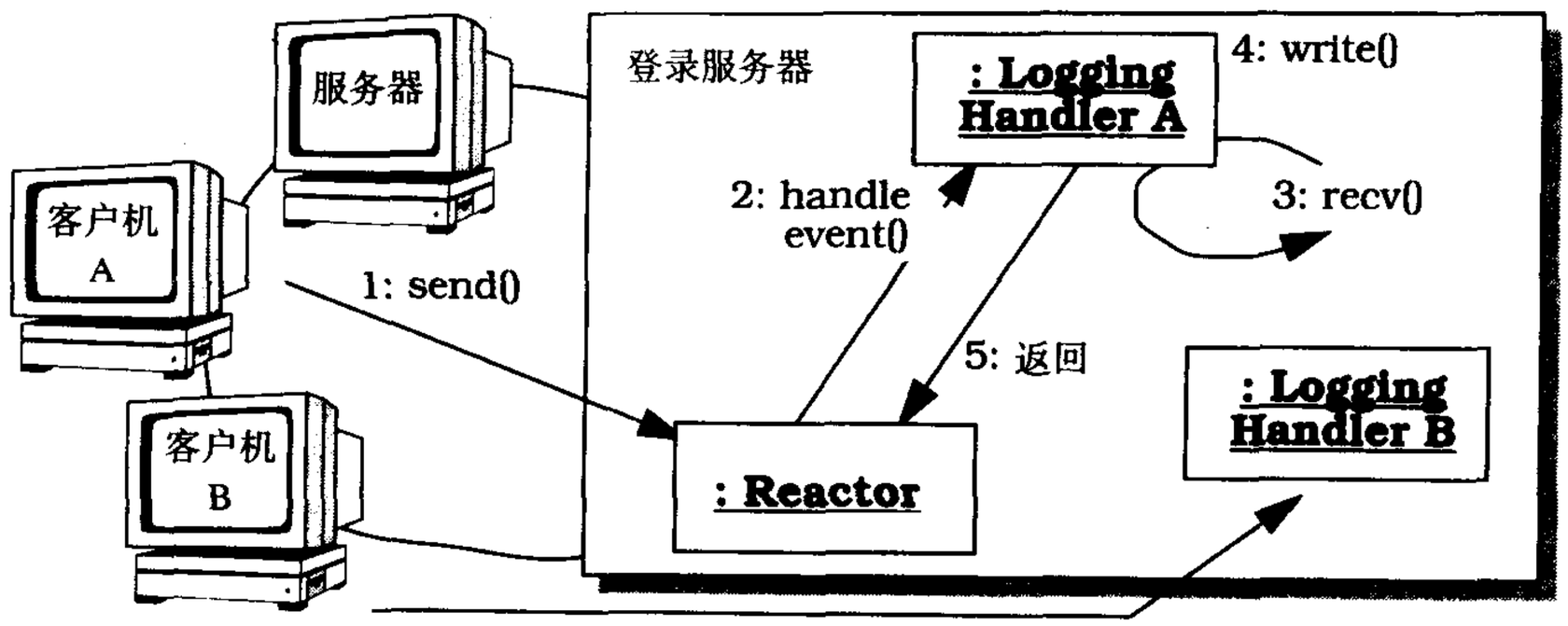


图 3-8

- 客户机发送登录记录请求（1），服务器的操作系统通知反应器单件在select()所选择的句柄上有一个未处理的指示事件。
- 反应器单件分配与该句柄有关的登录处理程序的handle_event()方法，通知它有一个新的指示事件（2）。
- 登录处理程序从Socket中以非阻塞方式读取记录（3）。重复第2、3步直到从socket句柄中全部接收了登录记录。
- 登录处理程序处理登录记录，并将记录写入到登录服务器的标准输出设备（4），也可以将登录记录重定向到任何合适的输出设备。
- 登录处理程序将控制返回给反应器的事件循环（5），在事件循环中等待后续的指示事件。

下面的代码实现了登录服务器例子中的具体事件处理程序。Logging_Acceptor类用于被动连接的建立，Logging_Handler类提供与应用有关的数据接收和处理服务。

Logging_Acceptor是接受器-连接器模式中一个接受器组件的例子。它将连接建立和服务初始化与其后进行的操作分开。该模式使得一个服务的与应用有关的部分（如Logging_Handler）多样化，这种多样化独立于所采用的建立连接和初始化句柄的机制。 199

Logging_Acceptor对象被动地从客户机应用程序接受连接请求，创建与客户机有关的Logging_Handler对象，该对象接收并处理客户机的登录记录。注意，Logging_Handler对象维持一个与所连接的客户机的会话。因此不需要对每一个登录记录建立一个新的连接。

Logging_Acceptor类继承了实现动作（1.2）中的Event_Handler基类的“单方法”分配接口变体。Logging_Acceptor构造函数向一个反应器注册自己，用于处理ACCEPT事件：

```
class Logging_Acceptor : public Event_Handler {
public:
    Logging_Acceptor (const INET_Addr &addr,
                      Reactor *reactor):
        acceptor_ (addr), reactor_ (reactor) {
        reactor_->register_handler (this, ACCEPT_EVENT);
    }
```

注意，register_handler()方法“双分配”到Logging_Acceptor的get_handle()方法以得到其被动模式的socket句柄。从此，一旦一个连接指示到达，反应器就分配Logging_Acceptor的handle_event()方法，此方法是一个工厂方法（factory method）［GoF95］：

```
virtual void handle_event
     (HANDLE, Event_Type event_type) {
     // Can only be called for an ACCEPT event.
     if (event_type == ACCEPT_EVENT) {
          SOCK_Stream client_connection;

          // Accept the connection.
          acceptor_.accept (client_connection);

          // Create a new <Logging_Handler>.
          Logging_Handler *handler = new
               Logging_Handler (client_connection,
                                reactor_);
     }
}
```

200

handle_event()钩子方法调用SOCK_Acceptor的accept()方法初始化SOCK_Stream。一旦SOCK_Stream被动地和一个新的客户机建立连接，登录服务器中就动态地分配一个Logging_Handler对象以处理登录请求。

该类的最后一个方法返回被动模式的底层socket的I/O句柄：

```
virtual HANDLE get_handle () const {
     return acceptor_.get_handle ();
}
```

注册了Logging_Acceptor后该方法就由反应器单件调用了。Logging_Acceptor类的私有部分是硬编码的，以包含一个SOCK_Acceptor包装器外观：

```
private:
     // Socket factory that accepts client connections.
     SOCK_Acceptor acceptor_;

     // Cached <Reactor>.
     Reactor *reactor_;
};
```

SOCK_Acceptor句柄工厂使Logging_Acceptor对象可以接受一个被动模式socket句柄上的连接指示，此句柄正在传输端点上侦听。当连接从客户机到达时，SOCK_Acceptor被动地接受连接并产生一个初始化的SOCK_Stream。然后SOCK_Stream使用TCP在客户机和登录服务器之间可靠地传输数据。

Logging_Handler类接收并处理客户机应用程序发送的登录记录。和上面介绍的Logging_Acceptor一样，Logging_Handler继承Event_Handler类，这样其构造函数能够在一个READ事件发生时分配的反应器上注册自己：

```
class Logging_Handler : public Event_Handler {
public:
     Logging_Handler (const SOCK_Stream &stream,
                      Reactor *reactor):
          peer_stream_ (stream) {
```

```
        reactor->register_handler (this, READ_EVENT);
}
```

结果是，当一条登录记录到达一个连接的Socket并且操作系统产生了一个相应的READ指示事件时，反应器就自动地分配与Logging_Handler相关的handle_event()方法：

201

```
virtual void handle_event (HANDLE,
                           Event_Type event_type) {
    if (event_type == READ_EVENT) {
        Log_Record log_record;

        // Code to handle "short-reads" omitted.
        peer_stream_.recv (&log_record,
                           sizeof log_record);

        // Write logging record to standard output.
        log_record.write (STDOUT);
    }
    else if (event_type == CLOSE_EVENT) {
        peer_stream_.close ();

        // Deallocate ourselves.
        delete this;
    }
}
```

handle_event()方法接收、处理登录记录[⊖]并将它们记录在标准输出设备（STDOUT）中。类似地，当客户机关闭连接时，反应器传递CLOSE_EVENT标志，通知Logging_Handler关闭它的SOCK_Stream并删除自己。如果CLOSE_EVENT传递给了handle_event()方法，在方法返回后，反应器就会自动地从其内部多路分解表中删除处理程序。该类的最后一个方法返回一个句柄，该句柄表示底层的数据模式的流socket：

```
virtual HANDLE get_handle () const {
    return peer_stream_.get_handle ();
}
```

当注册Logging_Handler后，反应器就调用该方法。Logging_Handler类的私有部分是硬编码的，以包含一个SOCK_Stream包装器外观：

```
private:
    // Receives logging records from a connected client.
    SOCK_Stream peer_stream_;
};
```

202

登录服务器的main()函数实现单线程的登录服务器，该登录服务器在反应器单件的handle_events()事件循环中等待：

```
// Logging server port number.
const u_short PORT = 10000;

int main () {
```

⊖ Log_Record的内存布局和传统C风格的struct一样。因此，其中没有虚函数、指针或引用，所有的值都是连续存放的。

```
    // Logging server address.
    INET_Addr addr (PORT);

    // Initialize logging server endpoint and register
    // with reactor singleton.
    Logging_Acceptor la (addr, Reactor::instance ());

    // Event loop that processes client connection
    // requests and log records reactively.
    for (;;)
        Reactor::instance ()->handle_events ();
    /* NOTREACHED */
}
```

当客户机的请求到达并由操作系统转换为指示事件时，反应器单件调用Logging_Acceptor和Logging_Handler具体事件处理程序的钩子方法分别进行接受连接、接收和处理登录记录等处理。

203

图3-9说明了登录服务器中的行为：

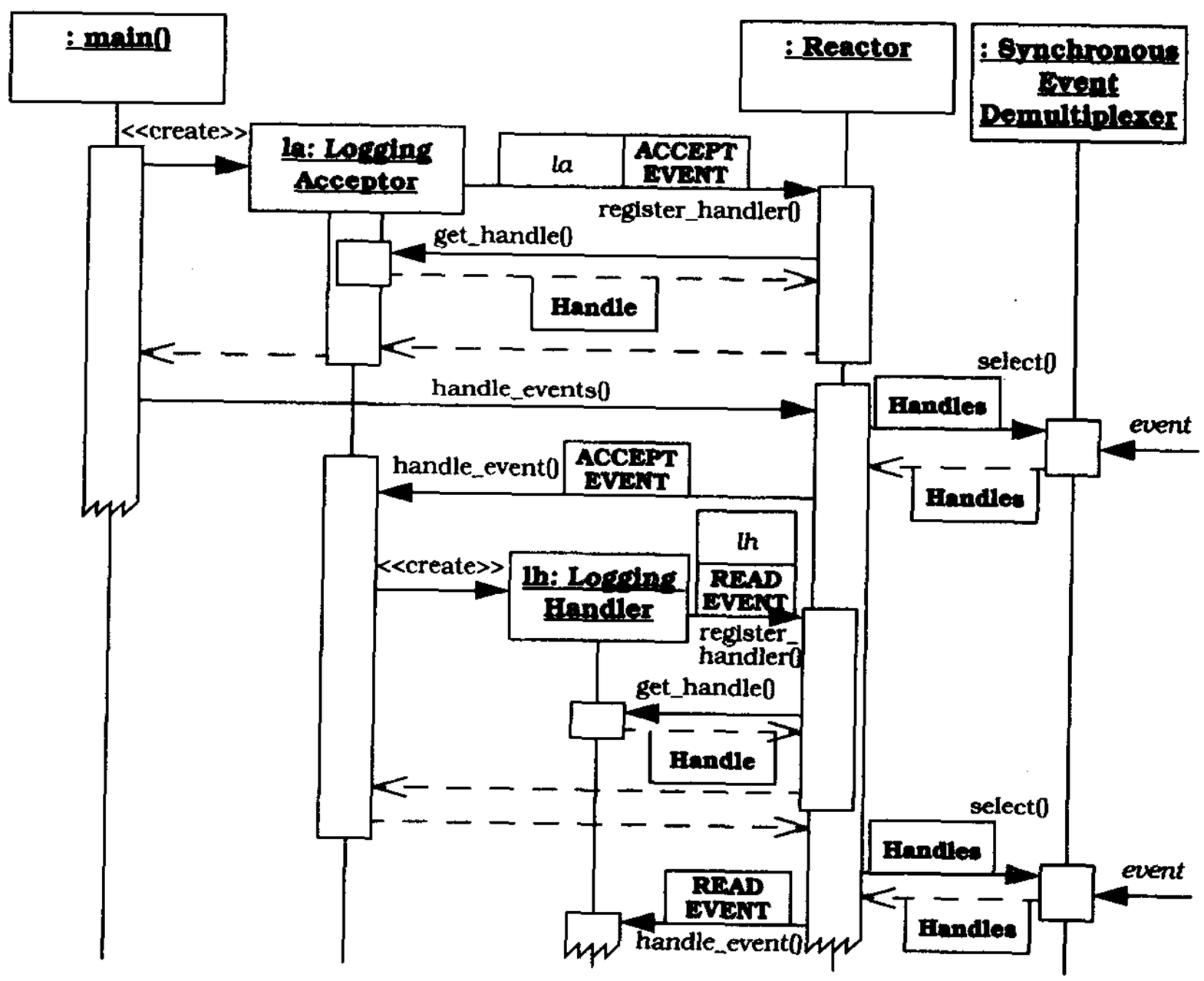


图 3-9

10. 变体

“实现”一节描述了与一个反应器的实现有关的动作，该反应器在一个控制单线程内从一个I/O句柄集中多路分解出指示事件。下面介绍反应器模式的变体，它们用于支持并发、重入或基于定时器的事件。

线程安全反应器（Thread-safe Reactor）。驱动单线程应用程序主事件循环的反应器不要求加锁，因为在它的应用程序进程中隐含地将事件处理程序handle_event()钩子方法的分配串

204

行化。

然而，在多线程应用中，反应器也可作为单线程的多路分解器/分配器。在这种情况下，虽然只有一个线程运行反应器的handle_events()事件循环方法，多个应用线程也可以向反应器注册或从反应器删除事件处理程序。另外，反应器调用的事件处理程序和其他线程状态共享，并和它们在同一状态下并发地工作。设计一个线程安全的反应器时必须考虑以下三个问题：

- *防止竞争条件*（race condition）。必须串行化反应器的临界区，以防止当多个应用线程修改反应器的内部共享状态时产生竞争条件。防止竞争条件的常用方法是使用互斥机制，如互斥或信号灯，以保护多个线程共享的内部状态。

  例如，可以为反应器的多路分解表增加一个互斥，在反应器的用于注册和删除事件处理程序的方法中使用定界加锁这一惯用法，以自动地施加和释放锁。这种改进有助于保证多线程不会同时注册和删除句柄和事件处理程序，否则会破坏反应器的多路分解表。

  为了确保在单线程应用程序中也可以很方便地实现反应器，可以应用策略化加锁模式将加锁机制参数化。

- *防止自死锁*（self-deadlock）。在多线程反应器中，在实现活动（3.4）中描述的反应器实现必须串行化，以防止注册、删除和多路分解事件处理程序时出现竞争条件。然而，如果对这种串行化考虑不周，当反应器的handle_events()方法回调与应用有关的具体的事件处理程序时，这些事件处理程序会通过处理程序注册和删除方法重新进入反应器，发生自死锁。

  为了防止自死锁，在互斥机制中可以使用*递归锁*（recursive lock）[Sch95]，拥有锁的线程可以重新获得该锁，而不会在线程上发生自死锁。当由反应器分配的多个事件处理程序钩子方法在同一个线程中拥有递归锁时，使用反应器模式有助于防止死锁。

205

- *显式地通知一个正在等待的反应器事件循环线程*。运行一个反应器事件循环的线程通常要在同步事件多路分解器花大量的时间等待句柄集中发生的指示事件。因此，当其他线程通过调用它的方法来注册和删除事件处理程序，从而改变了多路分解表的内容时，应该显式地通知反应器事件循环线程。否则，它将会在很晚时才知道这些变化，从而妨碍了它对重要事件的反应能力。

  应用程序线程通知反应器线程的一个有效的方式是在初始化反应器时预先建立一对“记录器/阅读器”IPC句柄，如UNIX管道或者一个回环TCP Socket连接。阅读器句柄以及一个特殊的“通知事件处理程序”注册在反应器上，当通过连接的记录器句柄得到一个字节时，由通知事件处理程序唤醒反应器。

  任何一个应用程序线程为注册和删除事件处理程序而调用反应器的方法时，它们先修改多路分解表，然后向记录器句柄发送一个字节。该字节唤醒反应器的事件循环线程，使它在再次等待同步事件多路分解器之前重构修改后的句柄集。

并发事件处理程序。在前面“实现”一节中介绍了一个单线程反应性分配的设计方法，其中事件处理程序借用了反应器的控制线程。事件处理程序也可以在自己的控制线程中运行。这样反应器可以与先前分配给事件处理程序的钩子方法并发地多路分解和分配新的指示事件。可以使用主动对象、领导者/追随者和半同步/半异步模式实现并发具体事件处理程序。

并发同步事件多路分解器。“实现”一节描述的同步事件多路分解器由反应器在一个控制线程中串行地调用。而其他类型的同步事件多路分解器，如WaitForMultipleObjects()函数，则由多个线程在同一个句柄集中并发地调用。

206

当可以在没有操作阻塞的情况下激活对句柄的操作时，并发的同步事件多路分解器向一个调用线程返回一个句柄，然后向相应的事件处理程序分配合适的钩子方法。

通过并发地调用同步事件多路分解器，多个线程可以同时向事件处理程序多路分解和分配事件，从而可以改善应用程序的吞吐率。不过，反应器的实现则因此变得很复杂，且可移植性差。

例如，可能需要对分配事件处理程序钩子方法进行引用记数，也可能因要注册和删除事件处理程序对反应器方法的调用进行排队，而它的实现需要通过使用命令（Command）模式[GoF95]，将变更推迟，直到没有一个线程将钩子方法分配给事件处理程序。如果具体事件处理程序必须是线程安全的，那么应用也可能变得更复杂。

重入反应器。一般来说，仅当被反应器调用时，具体事件处理程序才有反应，它并不调用反应器的事件循环本身。然而，有些具体情况则要求具体事件处理程序通过调用反应器的handle_events()方法，运行它的事件循环，从而获取特定的事件。例如，CORBA异步方法调用（AMI）的特性[ARSK00]要求ORB Core支持嵌套的work_pending()/perform_work() ORB事件循环。因此，如果ORB Core使用反应器模式[SC99]，那么反应器的实现应该是可重入的。

实现可重入反应器的一个通用方法是将多路分解表中的句柄集状态信息在调用同步事件多路分解器之前复制到运行时使用的堆栈中。这种方法能确保对句柄集中的任何改变相对于反应器的某一嵌套层来说都是局部的。

定时器和I/O事件的多路分解的集成。在“实现”一节描述的反应器主要考虑在支持登录服务例子中所必需的多路分解和分配特性。因此，它只将指示事件多路分解给句柄集。可以集成定时器事件和I/O事件的多路分解来实现更通用的反应器。

207

一个反应器的定时器机制应该允许应用程序注册基于时间的具体事件处理程序。使用这一机制，应用程序指定在将来的某一时刻调用事件处理程序handle_timeout()方法。可以使用不同的策略实现反应器的定时器机制，如堆[Balee98]、偏差表（delta-list）[CoSte91]、定时轮（timing wheel）[Vala97]。

- 堆是一个“偏序的、几乎完全的二叉树”，可以确保在平均和最坏的情况下插入或删除一个具体事件处理程序的时间复杂度是$O(\log n)$。
- 偏差表中用与前一个定时器值的偏差来表示时间。
- 定时轮使用环形缓冲区，这样在轮中启动、停止或者维护定时器的时间为常量$O(1)$。

➡要使应用程序能调度、取消和调用基于定时器的事件处理程序，需要对实现活动（2）中定义的反应器接口进行一些改进：

```
class Reactor {
public:
    // ... same as in implementation activity 2 ...

    // Schedule a <handler> to be dispatched at
    // the <future time>.  Returns a timer id that can
```

```
    // be used to cancel the timer.
    timer_id schedule (Event_Handler *handler,
                       const void *act,
                       const Time_Value &future_time);

    // Cancel the <Event_Handler> matching the <timer_id>
    // value returned from <schedule>.
    void cancel (timer_id id, const void **act = 0);

    // Expire all timers <= <expire_time>.  This
    // method must be called manually since it
    // is not invoked asynchronously.
    void expire (const Time_Value &expire_time);
private:
    // ...
};
```

208

应用程序通过schedule()方法来调度，使得一个具体事件处理程序在future_time之后就停止下来。可以传送一个异步完成标记（Asynchronous Completion Token, ACT）给schedule()方法。如果定时器时间到，将异步完成标记作为一个值传递给事件处理程序的handle_timeout()钩子方法。schedule()方法返回一个定时器ID值，它惟一标识在反应器定时器队列中注册的每个事件处理程序。可以将定时器ID传递给cancel()方法，以在定时器时间到之前删除事件处理程序。如果将一个非空的异步完成标记作为参数传递给cancel()，那么它将被赋值为在起初调度定时器时应用程序传递的异步完成标记，这样就可以删除动态分配的异步完成标记，从而避免内存泄漏。

为了对定时器和I/O事件多路分解加以集成，必须更好地实现反应器，使它能处理好定时器队列中安排的事件处理程序期限和传递给handle_events()方法的timeout参数。一般来说，handle_events()方法要等待最近的期限，该期限可以是timeout参数，或者是定时器队列中最近的期限。 ❑

11. 已知使用

InterViews[LC87]。在InterViews窗口系统中实现了被称为分配器的反应器模式。InterViews分配器用来定义应用程序的主事件循环，管理与一个或多个物理GUI显示设备的连接。因此，InterViews说明了如何使用反应器模式实现图形用户界面系统的反应性事件处理，这些图形用户界面系统可以是客户机，也可以是服务器。

X Windows中的Xt toolkit使用反应器模式实现其主事件循环。和在“实现”一节中介绍的反应器实现方式不一样，Xt toolkit中的回调使用C函数指针而不是事件处理程序对象。Xt toolkit同样说明了对于客户机和服务器的图形用户界面系统，如何将反应器模式用于实现反应性事件处理。

ACE反应器框架[Sch97]。ACE框架在其核心的多路分解器和分配器中使用了一个面向对象的框架来实现反应器模式。ACE中有一个称为ACE_Reactor的类，定义了反应器实现的一个公共接口。反应器实现包括ACE_Select_Reactor和ACE_WFMO_Reactor。可以分别使用不同的同步事件多路分解器（如WaitForMultipleObjects()和select()）创建这两个反应器实现。 209

有些CORBA实现中的ORB Core组件（OMG98a）（如TAO和ORBacus）使用反应器模式将客户机请求多路分解和分配给处理这些请求的服务程序。

呼叫中心管理系统。在呼叫中心管理系统中使用反应器模式管理在PBX和超级用户之间由事件服务器发出的事件。

Project Spectrum。Project Spectrum[PHS96]中的高速I/O传输子系统使用反应器模式多路分解和分配在电子医疗影像系统中的事件。

接收电话呼叫。在日常生活中经常有反应器模式，如电话呼叫。电话用户作为一个向反应器（通信网络）注册的事件处理程序，处理在某一电话号码（句柄）上接收的呼叫。另外有人拨了某一用户的电话号码时，网络通过响铃通知该用户有一个尚未处理的“呼叫请求”。该用户拿起电话时，他就是在对该请求做出响应，并通过与对方交谈来处理这些请求。

12. 结论

反应器模式有下列**优点**：

事务分离。反应器模式将与应用程序无关的多路分解和分配机制和与应用有关的钩子方法功能分开。与应用无关的机制被设计成可重用的组件，该组件知道如何多路分解指示事件并分配由事件处理程序定义的适当的钩子方法。相反，钩子方法中与应用有关的功能知道如何完成特定类型的服务。

模块化、可重用性和可配置性。该模式将事件驱动的应用功能分解成几个组件。例如，面向连接的服务可以分解成两个组件：一个负责建立连接，另一个负责接收和处理数据。

210

使用这种分解方法可以开发和配置通用的事件处理程序组件，如接受器、连接器、服务处理程序，它们被松散地集成在反应器中。这种模块化有助于更高级的软件组件重用，因为修改或者扩充服务处理程序的功能不影响接受器和连接器组件的实现。

➡在登录服务器例子中，可以很容易地将`Logging_Acceptor`类泛化，得到接受器-连接器模式中描绘的接受器组件。这个通用的接受器可以重用于很多不同的面向连接的服务中，如文件传输、远程登录、视频点播等。很显然可以在不影响可重用的接受器组件的情况下在`Logging_Handler`类中增加新的功能。❑

可移植性。UNIX平台提供了`select()`[Ste98]和`poll()`[Rago93]这两个同步事件多路分解函数，而在Win32平台中可以使用`WaitForMultipleObjects()`[Sol98]和`select()`同步地多路分解事件。尽管这些多路分解调用都要检测和报告可能在多个事件源同时发生的一个或多个指示事件，但这些API之间还是存在着细微的差别。只要把反应器接口从实现中使用的低层操作系统的同步事件多路分解器函数中分离出来，就可以很容易地跨平台移植使用反应器模式的应用程序。

粗粒度的并发控制。在应用程序的进程或线程内，反应器模式的实现在事件多路分解和分配层将事件处理程序的调用串行化。这种粗粒度的并发控制可以消除应用程序进程内更复杂的同步需求。

反应器模式有如下的**不足**：

应用范围受到限制。如果操作系统支持对句柄集的同步事件多路分解，使用反应器模式会很高效。然而，如果操作系统并不提供这类支持，可以在反应器实现中使用多线程模拟反应器模式的语义。例如，可以通过为每个句柄都分配一个线程来处理。

211

一旦句柄上有待处理的事件，与之相关的线程就读取该事件，并将其放在队列中，由反应

器顺序处理这个队列。不过这种设计效率不高，因为它串行化了所有的事件处理程序线程。因此，在没有增加应用层的并行性的情况下，反而增加了同步和语境切换的开销。

非抢先的方式（Non-pre-emptive）。在单线程的应用程序中，可以让反应器线程的具体事件处理程序运行直至终了，从而防止反应器分配其他的事件处理程序。因此，一般来说，事件处理程序不应该执行耗时长的操作，如对一个句柄的阻塞I/O操作，因为这样会阻塞整个进程，并妨碍反应器对连接到其他句柄上的客户机的响应。

要处理耗时长的操作，如传送几兆字节的图像[PHS96]，在独立的线程中处理事件处理程序会更有效。可以使用主动对象或者半同步/半异步模式的变体实现这种设计，在这种设计中，服务的执行与反应器的主事件循环并发地进行。

调试和测试的复杂性。因为使用了被逆转的控制流，所以难以对使用反应器模式的应用程序进行调试。在这种模式中，控制在框架底层结构和针对具体应用的事件处理程序的方法回调之间来回振荡。反应器的控制转换增加了在调试器中使用“单步法”考查反应框架运行特性的难度，因为应用程序开发者可能不理解或者根本不能接触到框架代码。

这些问题与调试用lex和yacc写的编译器的文法分析器和解析器时遇到的问题相似。在这些应用中，当控制线程在用户定义的语义动作例程内时，调试就很简单。但是，当控制线程返回给产生的确定的有限自动机（Deterministic Finite Automata，DFA）构架（skeleton）后，就难以理解程序的逻辑了。

参见 212

反应器模式与观察者（Observer）模式[GoF95]以及出版者-订阅者（Publisher-Subscriber）模式[POSA1]有关。在后两种模式中，当目标变化后，所有相关的组件都会得到通知。但是，在反应器模式中，在一个事件源上发生了与处理程序有关的事件时，单件处理程序会得到通知。一般来说，反应器模式用于将多个事件源的指示事件多路分解给相应的事件处理程序，而观察者或订阅者通常与单个事件源有关。

反应器模式与责任链（Chain of Responsibility）模式[GoF95]也有关，后者将请求交付给一个负责的服务处理程序。反应器模式与责任链模式的不同之处在于反应器将一个事件处理程序与特定的事件源关联起来。相反，责任链模式搜索整个链以找出第一个匹配的事件处理程序。

可以认为反应器模式是异步主动器模式的同步化变体。主动器支持多路分解和分配多个事件处理程序，这些事件处理程序由于异步操作的完成而激活。相反，反应器模式负责多路分解和分配多个事件处理程序，当指示事件表明可能在不阻塞的情况下同步地启动一个操作时激活这些事件处理程序。

主动对象模式将方法执行和方法调用分开，以简化不同线程中调用的方法对共享状态的同步访问。当不能使用线程或者当线程化的开销和复杂度很高时，通常使用反应器代替主动对象模式。

反应器模式可用于领导者/追随者以及半同步/半异步模式的实现中，作为底层的同步事件多路分解器。而且，如果反应器的事件处理程序处理的事件生存期很短(short-lived)，就可以用反应器模式代替这两种模式。这种简化还能明显地降低编写应用程序的工作量，并潜在地改善性能。

Java并不提供网络事件的同步多路分解器。特别是它并不封装select()，原因是很难以一种可移植的方式支持同步多路分解。因此难以用Java直接实现反应器模式。但是，AWT中的Java事件处理，尤其是侦听者模型和基于委托的模型，在下面几个方面和反应器模式很类似：

213

- 一般地，应用开发者总是要重用预制的图形组件，如各种类型的按钮。开发者一般编写事件处理程序实现应用特定的逻辑来处理具体的事件，如在按钮上的鼠标单击事件。在接收与按钮有关的事件之前，必须向该按钮注册一个事件处理程序，该事件处理程序处理所有这类称为ActionEvent的事件。
- 当Java虚拟机（Java Virtual Machine, JVM）调用底层的固有代码时，该代码通知按钮中的对等体，该对等体是位于固有代码之上的第一个Java层。按钮对等体是平台特定的，它发布一个新的ActionEvent事件，该事件将在一个由JVM创建的特殊目的线程事件处理程序线程内执行。
- 然后将事件放入一个队列中，由EventDispatchThread对象运行一个循环，将事件向上“吸取”到AWT配件层，AWT配件层最终将事件分配给所有注册的侦听者，这些侦听者存储在称为AWTEventMulticaster的递归数据结构中。

所有的提取、分配处理以及后续的事件处理都在同一线程中同步运行，这和反应器的同步事件处理类似。

致谢

John Vlissides是反应器的[PLoPD1]版本的领头人，他和Ralph Johnson、Doug Lea、Roger Whitney以及Uwe Zdun等人，为我们将原先的反应器概念以模式的形式文档化提出了很多有益的建议。

214

3.2 主动器

主动器（Proactor）体系结构模式使事件驱动的应用程序能有效地多路分解和分配由于异步操作的完成而激活的服务请求，这样在不存在不利条件时能获得并发所带来的好处。

1. 例子

考虑一个必须同时执行多个操作的网络应用程序，如需要处理从多个远程Web浏览器发来的HTTP请求的高性能Web服务器[HPS99]。当用户要从指定的URL下载内容时，有以下四个步骤（如图3-10）：

1) 浏览器与URL指定的Web服务器建立连接，然后向它发送HTTP GET请求。

2) Web服务器接收浏览器的CONNECT指示事件，接受连接，读取并解析请求。

3) 服务器打开并读取指定的文件。

4) 最后，服务器将文件的内容发回给Web浏览器，并关闭连接。

实现Web服务器的一个方法是使用与反应器模式一致的反应性事件多路分解模型。在这种设计中，一旦Web浏览器连接到一个Web服务器，就要创建一个新的事件处理程序，该事件处理程序读取、解析和处理请求，并将文件内容传给浏览器。在反应器上注册该处理程序，反应器将

协调每个指示事件向相应事件处理程序的同步多路分解与分配。

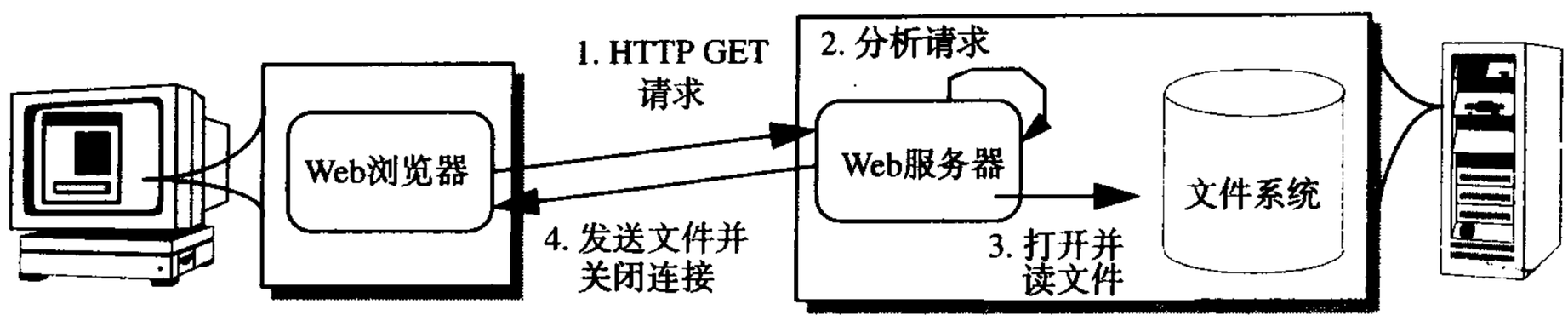


图 3-10

215

尽管可以根据反应性Web服务器的设计直接进行编程，但并不能扩展这种设计，使它能支持许多同时使用的用户和较长时间的用户请求，因为它在事件多路分解层串行化了所有的HTTP处理。结果是在任一时刻只能分配和处理一个GET请求。

实现Web服务器的一种更具扩展性的方法是使用某种形式的同步多线程方法。在这种模型中，有一个单独的服务器线程处理各浏览器的HTTP GET请求[HS98]。例如，为每个请求动态地建立一个新的线程，或者预先建立一个线程池，并用领导者/追随者或半同步/半异步模式进行管理。在这些情况下，每个线程负责建立连接、读取HTTP请求，解析请求和传送文件操作，这些操作都是同步的——也就是说，服务器处理操作一直阻塞，直到这些操作完成。

同步多线程是一个常用的并发模型。但正如在第3.1节的“例子”一节所讨论的，这种模式可能会有与效率、可扩展性、程序复杂性和可移植性等方面有关的问题。

在能有效地支持异步I/O的操作系统中，Web服务器可以异步地调用操作，以进一步改善可扩展性。例如在Windows NT中，可以使Web服务器异步调用Win32操作来处理外部产生的指示事件，如TCP CONNECT和HTTP GET请求，并异步地将被请求的文件传送给Web浏览器。

这些异步操作完成后，操作系统向Web服务器返回一个包含有结果的完成事件。Web服务器在返回到事件循环之前处理这些事件并执行相应的动作。由于异步调用和相应的完成事件在时间和空间上是分离的，所以难以构建这种软件，使之具有异步事件处理模型的好处。因此，异步程序设计要求有一种更成熟的、更易理解的事件多路分解和分配机制。

2. 语境

异步接收和处理多个服务请求的事件驱动的应用程序。

3. 问题

216

异步处理多个服务请求通常可以改善分布式系统中的事件驱动应用程序（特别是服务器）的性能。完成异步服务处理后，应用程序必须处理由操作系统发出的表示异步计算结束的相应的完成事件。

例如，一个应用程序必须向处理异步操作结果的内部组件多路分解和分配各完成事件。该组件可以响应外部客户机（如Web浏览器客户机）或内部客户机（如初始化异步操作的Web服务器组件）。要有效地支持这种异步计算模型，需解决以下四个强制条件：

- 为了改进可扩展性和延迟性能，一个应用程序必须在禁止耗时长的操作过分地延迟其他操作处理的情况下同时处理多个完成事件。

- 为了使吞吐率最大，和“例子”一节中介绍的一样，应避免CPU之间的任何不必要的语境切换、同步和数据移动。
- 将新的或改进的服务与已有的完成事件多路分解与分配机制集成所花的代价要最少。
- 应用程序代码应尽量不受多线程和同步机制复杂性的影响。

4. 解决方案

将应用服务分成两部分：异步执行的耗时长的操作和在这些操作完成后处理其结果的完成处理程序。将异步操作完成时对完成事件的多路分解和将它们分配到相应的处理它们的完成处理程序这两部分集成起来。将完成事件多路分解与分配机制与完成处理程序中与应用有关的对完成事件的处理分开。

具体地说就是：为应用程序所提供的每个服务引入一个异步操作。由这些异步操作通过一个句柄和完成处理程序来主动地初始化对服务请求的处理，完成处理程序处理包含这些异步操作结果的完成事件。例如，可以在应用程序中由启动程序调用异步操作来接受从远程应用程序发来的连接请求。异步操作由异步操作处理器执行。操作执行完毕后，异步操作处理器在完成事件队列中插入一个包含该操作结果的完成事件。

217

称为主动器的异步事件多路分解器等待该队列。异步事件多路分解器从队列中删除一个完成事件后，主动器向该异步操作相应的与应用有关的完成处理程序多路分解和分配该事件，然后该完成处理程序处理异步操作的结果，可能要调用其他的异步操作，这些异步操作遵循上述活动链。

5. 结构

主动器模式包括9个参与者：

操作系统提供句柄（Handle）以标识实体（如网络连接）或打开文件（能产生完成事件）。可以响应外部服务请求（如远程应用程序发来的连接或数据请求）而产生完成事件，也可以响应应用程序内部产生的操作（如超时或异步I/O系统调用）而产生完成事件。

➡Web服务器为每个Web浏览器连接产生一个独立的socket句柄。在Win32中，以“重叠I/O”模式产生各socket句柄，这意味着对句柄的操作是异步的。在异步执行的操作完成后，Windows NT I/O子系统也会产生完成事件。 ❑

异步操作（asynchronou operation）表示在实现如通过socket句柄异步读写数据这样的服务时要使用的耗时长的操作。调用一个异步操作后，执行该异步操作而不阻塞调用者的控制线程。这样调用者可以执行其他操作。如果一个操作必须等待另一事件的发生，如远程应用程序产生的一个连接请求，则它要被延迟到事件到达后才开始执行（如图3-11）。

218

➡主动Web服务器调用Win32 `AcceptEx()`操作来异步地接受来自Web浏览器的连接。接受连接后，Web服务器调用Win32的异步操作`ReadFile()`和`WriteFile()`，和与之连接的浏览器通信。 ❑

完成处理程序（Completion handler)定义了一个由一个或多个钩子方法组成的接口[Pree95][GHJV95]。可以使用这些方法处理与应用相关的完成事件中返回的信息。完成事件是在异步操作执行完后产生的。

类 句柄	协作者
责任 • 句柄标识一个操作系统资源，该资源可以是异步操作的目标，或者是操作系统中的完成事件源	

类 异步操作	协作者 • 句柄 • 异步操作处理器
责任 • 定义可以异步执行的操作 • 用于实现一个服务	

图 3-11

实现继承的钩子方法后，具体完成处理程序（concrete completion handler）特化了一个完成处理程序以定义一个特定的应用服务。当与完成处理程序相关的异步操作执行完后，钩子方法接收并处理包含在完成事件中的结果信息。一个具体完成处理程序对应于一个可以用于调用异步操作的句柄（如图3-12）。

例如，一个具体完成处理程序可以先调用一个异步读操作，然后接收该操作返回的数据。这样，具体完成处理程序可以先处理接收来的数据，然后调用一个异步写操作将结果返回给与之连接的远程同级应用程序。

➡在Web服务器中，有两个具体完成处理程序——HTTP接受器和HTTP 处理程序——对AcceptEX()、ReadFile()、WriteFile()异步操作的结果进行完成处理。HTTP 接受器是一个处理AcceptEX()异步操作的完成处理程序——它响应来自远程Web浏览器的连接请求事件，创建和连接HTTP 处理程序。然后HTTP 处理程序使用异步ReadFile()和WriteFile()操作处理来自远程Web浏览器的后续请求。 ❑ 219

类 完成处理程序	协作者
责任 • 定义一个用于处理异步操作结果的接口	

类 具体完成处理程序	协作者 • 句柄
责任 • 以一种与应用有关的方式处理异步操作的结果	

图 3-12

异步操作处理器（asynchronous operation processor）调用某一个句柄的异步操作并运行到结束，通常由操作系统内核实现异步操作处理器。异步操作执行完后，异步操作处理器产生相应的完成事件。根据操作所针对的句柄，异步操作处理器将该完成事件插入到与句柄相对应的完成事件队列(completion event queue)中，队列中保存有要被多路分解到相应的完成处理程序上的完成事件（如图3-13）。

➡在Web服务器中，Windows NT操作系统就是异步操作处理器。同样，完成事件队列就是

220 一个Win32完成端口［Sol98］，也就是由Windows NT内核代表应用程序维护的一个由完成事件组成的队列。一个异步操作完成后，Windows NT内核将完成事件在完成端口中进行排队，该完成端口与调用异步操作时使用的句柄对应。 ❑

类 异步操作处理器	协作者 • 异步操作 • 完成事件队列
责任 • 执行异步操作 • 异步操作完成时将完成事件在完成事件队列中排队	

类 完成事件队列	协作者 • 句柄
责任 • 当等待完成事件通过异步事件多路分解器删除时，保存完成事件	

图 3-13

异步事件多路分解器(asynchronous event demultiplexer)是一个函数，它等待在异步操作执行完后将被插入到完成事件队列中的完成事件。然后异步事件多路分解器函数将一个或多个完成事件从队列中删除，并返回给调用者。

➥Windows NT中的`GetQueuedCompletionStatus()`就是一个异步事件多路分解器。事件驱动的主动性应用程序可以使用该函数等待一段依应用而定的时间以得到下一个可用的完成事件。 ❑

主动器（proactor）为应用程序进程或线程提供事件循环。在该事件循环中，主动器调用一个异步事件多路分解器，等待完成事件的发生。当事件到达时，异步事件多路分解器返回。然后主动器将该事件多路分解给相应的完成处理程序，并为该处理程序分配合适的钩子方法，以处理完成事件的结果。如图3-14所示。

类 异步事件多路分解器	协作者 • 完成事件队列
责任 • 能够阻塞等待完成事件队列中出现完成事件 • 删除完成事件并将它返回给调用者	

类 主动器	协作者 • 完成处理程序 • 异步事件多路分解器 • 完成事件队列
责任 • 调用异步事件多路分解器将完成事件从队列中删除 • 向完成处理程序的钩子方法事件多路分解和分配完成事件	

图 3-14

➥Web服务器应用程序调用主动器的事件循环方法。该方法调用作为异步事件多路分解器的Win32函数`GetQueuedCompletionStatus()`。该函数等待主动器的完成端口中出现一个完成事件，并将它们从队列中删除。主动器的事件循环方法使用完成事件中的信息，将下一个

221

事件多路分解给相应的具体完成处理程序，并分配钩子方法。 ❑

启动程序（initiator）是应用程序内部的一个实体，它调用异步操作处理器的异步操作。通常启动程序处理所调用异步操作的结果，在这种情况下，它就担当了具体完成处理程序的角色（如图3-15）。

➡这个例子中，HTTP 接受器和HTTP 处理程序在Web服务器内部控制线程中既是启动程序又是具体完成处理程序。例如，HTTP 接受器调用`AcceptEx()`操作，异步接受来自远程Web浏览器的连接指示事件。有了一个连接指示事件后，HTTP 接受器创建一个HTTP处理程序，该处理程序调用异步的`ReadFile()`操作以获取和处理来自与之连接的Web浏览器的HTTP `GET`请求。 ❑

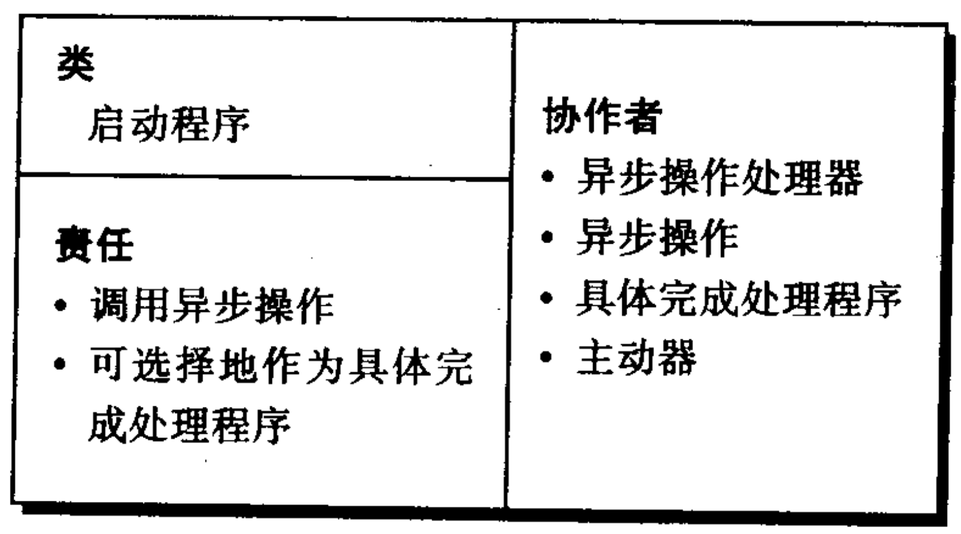


图 3-15

要留意在主动器模式中用启动程序和具体完成处理程序表示的应用程序组件为何是*主动性*实体。它们通过在一个异步操作处理器中*主动地*调用异步操作而促成了应用程序内部的控制流和数据流。

这些异步操作完成后，异步操作处理器和主动器通过完成事件队列进行协作。它们使用该队列将结果完成事件多路分解，回传给相应的具体完成处理程序，然后分配这些处理程序的钩子方法。处理完一个完成事件后，完成处理程序可以主动地调用新的异步操作。 222

主动器模式中的上述组成部分用类图3-16进行说明。

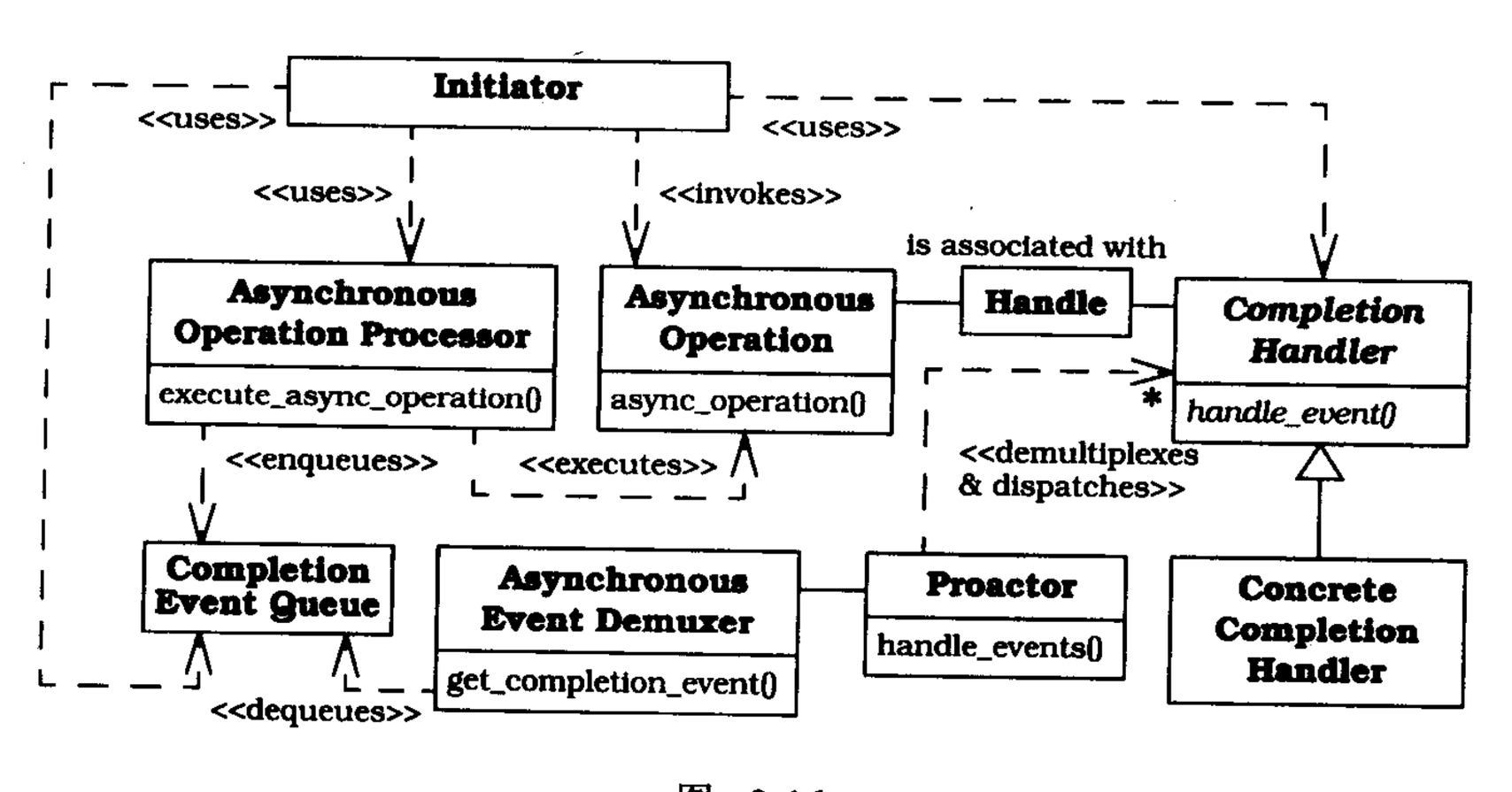


图 3-16

6. 动态特性

在主动器模式中有如下几种协作（如图3-17）：

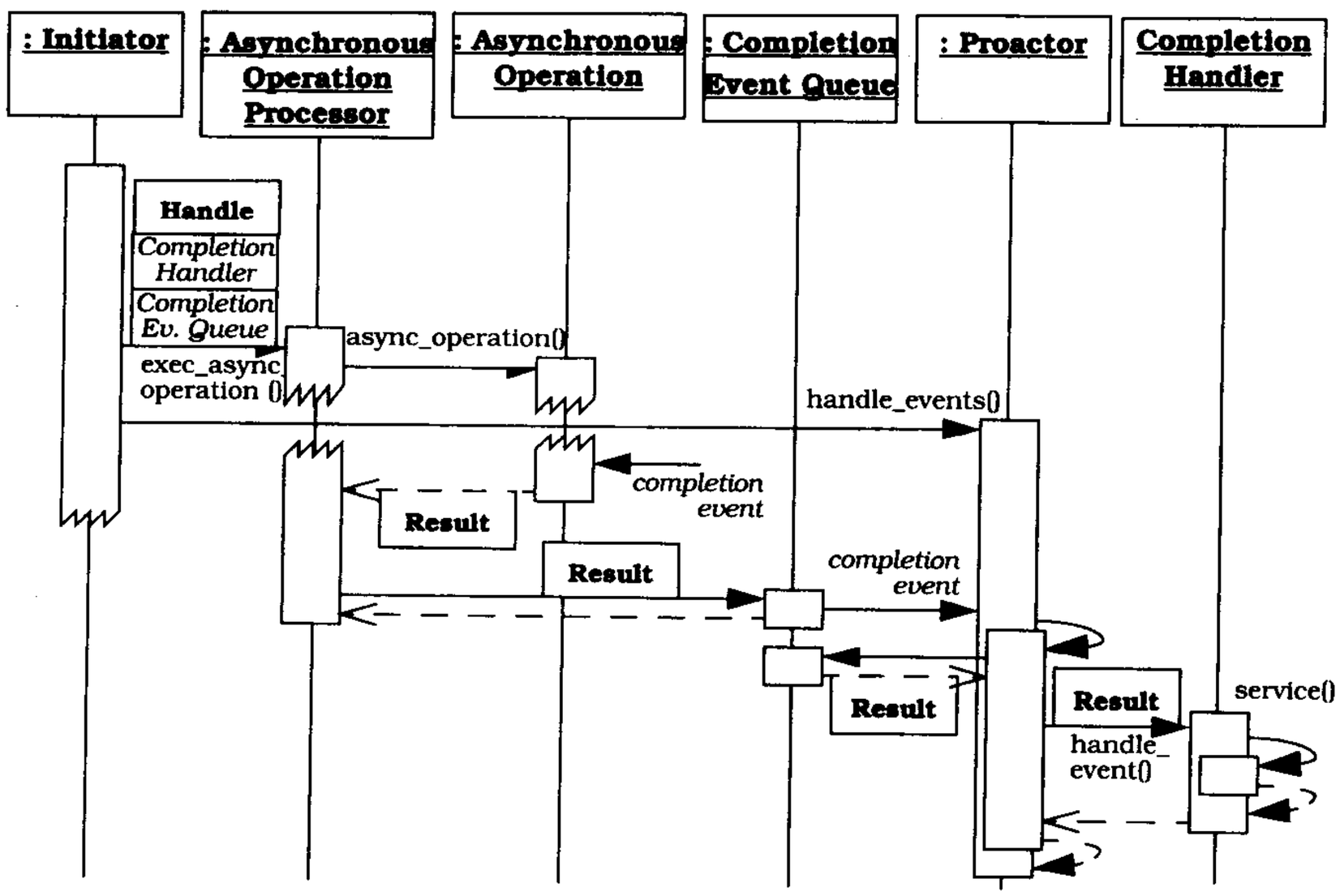


图 3-17

- 担当启动程序角色的应用程序组件通过某一句柄调用一个异步操作处理器的一个异步操作，除了将数据参数传递给异步操作之外，启动程序还将一些具体的完成处理参数，如完成处理程序或句柄传递给完成事件队列。异步操作处理器内部保存这些参数供以后使用。

➡在Web服务器中，HTTP处理程序通过在某套接字句柄上异步调用Read_File()操作，使操作系统读取新的HTTP GET请求。在激活该句柄上的这个操作后，将HTTP处理程序作为完成处理程序传递，以便它能处理异步操作的结果。❑

223

- 在启动程序调用异步操作处理器上的一个操作后，该操作和启动程序就可以独立地运行了。特别是启动程序可以调用新的异步操作，而其他操作可以继续并发地执行[⊖]。如果异步操作的目的是从远程应用程序接收一个服务请求，那么异步操作处理器会延迟该操作，直至服务请求到达。有了与请求相应的事件后，异步操作就结束执行了。

➡Windows NT操作系统延迟用于读取HTTP GET请求的ReadFile()异步操作，直到一个来自于远程Web浏览器的请求到达。❑

- 当一个异步操作执行完毕后，异步操作处理器产生一个完成事件。该事件中包括了异步操作的结果。然后，异步操作处理器将该事件插入到与调用异步操作的句柄相关的完成事件队列中。

➡如果HTTP处理程序调用异步操作ReadFile()来读取HTTP GET请求，Windows NT操作系统将会在完成事件中记录完成的状态，如表示成功还是失败的标志、读取的记录数等。❑

⊖ 为了简单起见，在图3-17中没有说明这种情况。

- 当应用程序可以处理由异步操作产生的完成事件时，它调用在这里称为handle_events()的主动器事件循环入口方法。该方法调用一个异步事件多路分解器[⊖]，等待异步操作处理器将完成事件插入到完成事件队列。从队列中删除完成事件后，主动器的handle_events()方法将该事件多路分解给对应的完成处理程序。然后向完成处理程序上分配合适的钩子方法，向钩子方法传递异步操作的结果。

  ➡在Web服务器例子中主动器使用一个Win32完成端口作为完成事件队列。它还使用Win32函数GetQueuedCompletionStatus()[Sol98]作为异步事件多路分解器，用于从完成端口中删除完成事件。❑ 224

- 然后，具体完成处理程序处理接收到的完成结果。如果完成处理程序向调用者返回一个结果，可能会有以下两种情况：第一，处理异步操作结果的完成处理程序也可以是起初调用该操作的启动程序。在这种情况下，完成处理程序并不需要将结果返回给调用者，因为它自己就是调用者。

  第二，一个远程应用程序或者应用程序内部的组件可能请求了异步操作。在这种情况下，完成处理程序可以调用传输句柄上的异步写操作，以将结果返回给远程应用。

  ➡为了响应远程Web浏览器的HTTP GET请求，HTTP处理程序可以指示Windows NT操作系统，通过异步调用WriteFile()函数，在网络中传输大文件。操作系统成功地完成该异步操作后，产生的完成事件中包含有传送到HTTP处理程序的字节数量。由于传输层有流量控制功能，所以可能在一次WriteFile()操作中不能传送完整个文件。在这种情况下，HTTP处理程序可以按照适当的文件偏移量调用另一个异步操作WriteFile()。❑

- 完成处理程序完成其处理后，可以调用另外的异步操作，在这种情况下本节所介绍的循环周期又开始了。

7. 实现

225

主动器模式中的参与者可以划分为两层：

- *多路分解/分配基础设施层组件*。该层按照通用的、与应用无关的策略执行异步操作。它也将异步操作的完成事件多路分解和分配给相应的完成处理程序。
- *应用程序层的组件*。本层定义了执行与应用有关的服务处理的异步操作和具体完成处理程序。

下面首先介绍通用的多路分解/分配基础设施层的组件，然后介绍应用程序层的组件。主要介绍使用单控制线程调用异步操作和向相应的完成处理程序分配钩子方法的主动器的实现方法。“变体”一节介绍与开发多线程主动器实现有关的活动。 226

(1) *将应用程序服务分成异步操作和完成处理程序*。为了实现主动器模式，必须将应用服务中通过句柄对异步操作的激活与对这些操作结果的处理区分开。异步操作通常耗时很长，或/和与I/O有关，如通过套接字句柄读写数据或与数据库通信。完成处理程序处理异步操作的结果。除此之外，完成处理程序还担当着启动程序的角色，也就是说，它们自己调用异步操作。

该活动的结果是一组异步操作，一组完成处理程序，以及一组异步操作与完成处理程序之间的关联关系。

⊖ 由于图3-17的空间限制，我们假定异步事件多路分解器与主动器组件集成在一块。

(2) 定义完成处理程序接口。完成处理程序中的接口由一个或多个钩子方法组成[Pree95]。这些钩子方法表示对与应用有关的完成事件的完成处理，这些完成事件是在异步操作执行完毕后产生的。实现完成处理程序的活动可分为三个子活动：

(2.1) 定义一种传递异步操作结果的类型。当异步操作完成或者被取消后，必须将完成事件结果传递给完成处理程序。这些结果表示异步操作是成功还是失败，以及成功传送的字节数。通常采用适配器模式[GoF95]将存储在完成事件中的信息转换成可以分配到相应的具体完成处理程序的形式。

➡下面的C++类将一个异步Win32操作的结果回传给具体完成处理程序：

```
class Async_Result : public OVERLAPPED {
    // The Win32 OVERLAPPED struct stores the file offset
    // returned when an asynchronous operation completes.
public:
    // Dispatch to completion handler hook method.
    virtual void complete () = 0;

    // Set/get number of bytes transferred by an
    // asynchronous operation.
    void bytes_transferred (u_long);
    u_long bytes_transferred () const;

    // Set/get the status of the asynchronous operation,
    // i.e., whether it succeeded or failed.
    void status (u_long);
    u_long status () const;

    // Set/get error value if the asynchronous operation
    // failed or was canceled by the initiator.
    void error (u_long);
    u_long error () const;
private:
    // ... data members omitted for brevity ...
};
```

227

Async_Result类从OVERLAPPED结构派生而来，这样应用程序可以在异步操作结果中增加一些特定的状态和方法。因为Win32 API并没有提供其他更直接的方式，用于当一个异步操作被调用时，向操作系统传递每个操作结果对象，所以在这里使用了C++的继承。 ❑

(2.2) 确定分配目标的类型。有两类完成处理程序可以与句柄关联，并作为主动器分配机制的目标：对象和函数指针。主动器模式的实现可以基于第3.1节中反应器模式的实现活动（1.1）中的准则，选择分配目标的类型。

(2.3) 定义完成处理程序分配接口策略。下一步定义由完成处理程序支持的用于处理完成事件的接口类型。和反应器模式一样，假设在此使用完成处理程序对象而不是使用函数指针。可以采用两种策略：

- 单方法分配接口策略。图3-16介绍了只包含一个事件处理方法（称之为Handle_Event()）的接口Completion_Handler的实现。主动器使用该方法向相应的完成处理程序分配完成事件。在这种情况下，将所发生的完成事件的类型作为参数传递给方法。第二个参数是所有异步操作结果的基类，可以根据完成事件，将该参数进一步地向下类型转换（downcast）为合适的类型。

228

➡下面的C++抽象基类说明了单方法分配接口策略。首先给出在单方法和多方法分配接口策略中都要用到的类型定义和一组枚举值。

```
typedef unsigned int Event_Type;
enum {
    // Types of indication events.
    READ_EVENT = 01,
    ACCEPT_EVENT = 01, // An "alias" for READ_EVENT.
    WRITE_EVENT = 02, TIMEOUT_EVENT = 04,
    SIGNAL_EVENT = 010, CLOSE_EVENT = 020
    // These values are powers of two so
    // their bits can be "or'd" together efficiently.
};
```

接着实现Completion_Handler类：

```
class Completion_Handler {
public:
    // Cache the <proactor> so that hook methods can
    // invoke asynchronous operations on <proactor>.
    Completion_Handler (Proactor *proactor):
        proactor_ (proactor) { }

    // Virtual destruction.
    virtual ~Completion_Handler ();

    // Hook method dispatched by cached <proactor_> to
    // handle completion events of a particular type that
    // occur on the <handle>. <Async_Result> reports the
    // results of the completed asynchronous operation.
    virtual void handle_event
        (HANDLE handle, Event_Type et,
         const Async_Result &result) = 0;

    // Returns underlying I/O <HANDLE>.
    virtual HANDLE get_handle () const = 0;
private:
    // Cached <Proactor>.
    Proactor *proactor_;
};
```

❑

使用单方法分配接口策略有可能在不改变类接口的情况下增加新的事件类型。然而，为了处理某一具体的事件，本策略需要在具体事件处理程序的handle_event()方法中使用C++的switch和if语句，但这又降低了其内在的可扩展性。 229

- *多方法分配接口策略*。实现Completion_Handler接口的另一个策略是为处理每类事件单独定义钩子方法，如handle_read()、handle_write()或handle_accept()。该策略比单方法分配接口策略更具可扩展性，因为多路分解是由主动器实现完成的，而不是由具体事件处理程序的handle_event()方法完成的。

➡下面的C++抽象基类说明了一个主动器采用的多方法分配接口，该接口用于处理基于Windows NT的Web服务器例子中的网络事件：

```
class Completion_Handler {
public:
    // The <proactor> is cached to allow hook methods to
    // invoke asynchronous operations on <proactor>.
```

```
Completion_Handler (Proactor *proactor):
    proactor_ (proactor) { }

// Virtual destruction.
virtual ~Completion_Handler ();

// The next 3 methods use <Async_Result> to report
// results of completed asynchronous operation.
// Dispatched by <proactor_> when an asynchronous
// read operation completes.
virtual void handle_read
    (HANDLE handle, const Async_Result &result) = 0;
// Dispatched by <proactor_> when an asynchronous
// write operation completes.
virtual void handle_write
    (HANDLE handle, const Async_Result &result) = 0;
// Dispached by <proactor_> when an asynchronous
// <accept> operation completes.
virtual void handle_accept
    (HANDLE handle, const Async_Result &result) = 0;

// Dispatched by <proactor_> when a timeout expires.
virtual void handle_timeout
    (const Time_Value &tv, const void *act) = 0;

// Returns underlying I/O <HANDLE>.
virtual HANDLE get_handle () const = 0;

private:
    // Cached <Proactor>.
    Proactor *proactor_;
};
```

230

使用多方法分配接口策略可以很容易地有选择地重载基类中的方法，从而避免在钩子方法中通过switch或if语句进行进一步的多路分解。然而，这种策略要求模式实现能预先知道钩子方法。上述例子中Completion_Handler接口的handle_*()钩子方法是专用于处理网络事件的。然而，这些方法并不能包括所有的通过Win32的WaitForMultipleObjects()机制处理的事件类型，如同步对象事件[SchSt95]。

单方法和多方法分配接口策略都是钩子方法[Pree95]和模板方法[GoF95]模式的实现。使用这些模式可以提供良好定义的钩子，应用程序可以对钩子进行具体化，并由低层分配代码回调。

通常完成处理程序既能作为主动器的完成分配的目标，又能作为调用异步操作的启动程序，正如“已解决的例子”一节中HTTP_Handler类所说明的那样。因此，Completion_Handler类的构造函数使一个Completion_Handler对象与一个指向主动器的指针关联。这种设计使得Completion_Handler的钩子方法可以调用新的异步操作，而对这些异步操作的完成处理将永远由同一个主动器进行分配。

(3) 实现异步操作处理器。异步操作处理器代表启动程序异步地执行操作。因此，其主要责任包括：

- 定义异步操作接口。
- 实现执行异步操作的机制，以及产生完成事件。
- 操作完成时将完成事件排队。

231

(3.1) 定义异步操作接口。可以向异步操作传递不同的参数，如句柄[⊖]、数据缓冲区、缓冲区长度以及操作完成后执行完成处理所需的信息。设计启动程序用以调用异步操作处理器的异步操作的程序接口时，要解决两个问题：

- 尽量增加可移植性和灵活性。异步操作可用于不同类型的I/O设备（如网络和文件）和不同的操作系统（Windows NT、VMS、Solaris和Linux等）上的数据读/写。可以使用包装器外观和桥模式使异步操作接口不依赖于底层的操作系统，从而确保接口能适应多种类型的I/O设备。
- 有效地、简洁地处理多个完成处理程序、主动器和完成事件队列。在一个应用中可以同时使用多个完成处理程序、主动器和完成事件队列。例如，不同的主动器对应于具有不同优先级的线程，这些不同的线程为不同完成处理程序的处理提供不同的服务质量。除了数据参数之外，异步操作还要指明当处理异步操作完成事件时要使用哪些句柄、具体完成处理程序、主动器和完成事件队列。

有效地整合所有这些完成处理信息的一个常用的方法是使用异步完成标记模式。当启动程序在一个句柄上调用一个异步操作时，可以向异步操作处理器传递一个异步完成标记，并由异步操作处理器保存起来供以后使用。每个异步完成标记包含有标识一个具体的操作以及指导其后续完成处理的信息。

232

当异步操作执行完后，异步操作处理器找出它预先保存的该操作的异步完成标记，并和它产生的完成事件关联。然后将这个修改过的完成事件插入到相应的完成事件队列。最终，运行应用程序事件循环的主动器会使用异步操作多路分解器从完成事件队列中删除完成事件的结果和异步完成标记。然后主动器使用该异步完成标记将完成事件结果多路分解和分配给异步完成标记指定的完成处理程序。

➡虽然Web服务器是使用Win32异步Socket操作实现的，但我们使用包装器外观模式将该类泛化（generalize），并使之与平台无关。因此，它可以用于异步操作处理器支持的其他类型的I/O设备。

Web服务器例子中HTTP处理程序使用下面的Async_Stream类接口来调用异步操作：

```
class Async_Stream {
public:
    // Constructor 'zeros out' the data members.
    Async_Stream ();

    // Initialization method.
    void open (Completion_Handler *handler,
               HANDLE handle, Proactor *proactor);

    // Invoke an asynchronous read operation.
    void async_read (void *buf, u_long n_bytes);

    // Invoke an asynchronous write operation.
    void async_write (const void *buf, u_long n_bytes);
private:
    // Cache parameters passed in <open>.
    Completion_Handler *completion_handler_;
    HANDLE handle_;
    Proactor *proactor_;
};
```

233

⊖ 句柄本身通常是由操作系统提供的，所以不需要再去实现。

一个具体的完成处理程序（如HTTP处理程序）可以将自己连同调用Async_Stream的async_read()和async_write()方法时所使用的句柄一起传递给open()：

```
void Async_Stream::open (Completion_Handler *handler,
                         HANDLE handle,
                         Proactor *proactor) {
    completion_handler_ = handler;
    handle_ = handle;
    proactor_ = proactor;

    // Associate handle with <proactor>'s completion
    // port, as shown in implementation activity 4.
    proactor->register_handle (handle);
}
```

下面的Async_Stream::async_read()方法实现说明了如何使用异步完成标记（ACT）。该方法使用Win32 ReadFile()函数异步地读取最多n_bytes个字节，并将它们保存在buf参数中。

```
void Async_Stream::async_read (void *buf, u_long n_bytes){
    u_long bytes_read;

    OVERLAPPED *act = new // Create the ACT.
        Async_Stream_Read_Result (completion_handler_);

    ReadFile (handle_, buf, n_bytes, &bytes_read, act);
}
```

作为指针传递给ReadFile()的异步完成标记是下面Async_Stream_Read_Result类的一个动态分配的实例：

```
class Async_Stream_Read_Result : public Async_Result {
public:
    // Constructor caches the completion handler.
    Async_Stream_Read_Result
        (Completion_Handler *completion_handler):
         completion_handler_ (completion_handler) { }

    // Adapter that dispatches the <handle_event>
    // hook method on cached completion handler.
    virtual void complete ();
private:
    // Cache a pointer to a completion handler.
    Completion_Handler *completion_handler_;
};
```

234

该类承担着异步完成标记和适配器的角色［GoF95］。它继承Async_Result，而Async_Result又继承Win32的OVERLAPPED结构，正如实现活动（2.1）中所示，可以将异步完成标记作为lpOverlapped参数传递给异步函数ReadFile()。ReadFile()将该异步完成标记传递给Windows NT操作系统，后者保存该异步完成标记备以后使用。

异步ReadFile()执行完毕后，它产生一个包含调用该操作时接收到的异步完成标记的完成事件。当主动器的handle_events()方法从完成事件队列中删除该事件时，它调用Async_Stream_Read_Result的complete()方法。然后和实现活动(5.4)中介绍的一样，该适配器方法分配完成处理程序的handle_event()钩子方法，以传递该事件。 ❑

(3.2) 选择异步操作处理机制。当启动程序调用一个异步操作时，异步操作处理器在不阻塞启动程序的控制线程的情况下执行该操作。一个异步操作处理器提供管理异步完成标记和异步执行操作的机制。当操作结束后它也产生完成事件，并将该事件插入到适当的完成事件队列中。

有些异步操作处理器允许启动程序取消异步操作，但完成事件还是要产生的。这样，完成处理程序就可以正确地重新利用异步完成标记和其他资源。

有些操作环境提供了这些执行异步操作和产生完成事件的机制，如实时POSIX[POSIX95]和Windows NT[Sol98]。在这种情况下，实现异步操作处理器只需要将已有的操作系统API映射为实现活动(3.1)中描述的异步操作包装器外观接口。“变体”一节描述了在本来不支持这些特性的操作系统平台上模仿异步操作处理器的技术。

(4) 定义主动器接口。应用程序使用主动器接口来调用一个事件循环，从完成事件队列中删除完成事件，将完成事件多路分解给相应的完成处理程序并分配相应的钩子方法。通常通过单件［GoF95］访问主动器接口，因为对每个应用进程来说一个主动器通常足够了。 235

主动器模式可以使用桥模式[GoF95]，使应用程序避免使用复杂的和不可移植的完成事件多路分解与分配机制。主动器接口对应于桥模式中的抽象组件，可以通过指针在内部访问与平台有关的主动器实例，这与桥模式中的实现层对应。

➡在Web服务器中主动器中定义了将句柄和完成端口关联起来以及主动地运行应用的事件循环的接口：

```
class Proactor {
public:
    // Associate <handle> with the <Proactor>'s
    // completion event queue.
    void register_handle (HANDLE handle);

    // Entry point into the proactive event loop. The
    // <timeout> can bound time waiting for events.
    void handle_events (Time_Value *wait_time = 0);

    // Define a singleton access point.
    static Proactor *instance ();
private:
    // Use the Bridge pattern to hold a pointer to
    // the <Proactor_Implementation>.
    Proactor_Implementation *proactor_impl_;
};
```

❑

主动器接口也定义了称为register_handle()的方法，如实现活动(5.5)中所描述的，它将句柄和主动器的完成事件队列关联起来，这种关联能够确保当异步操作执行完后，产生的完成事件能插入到某一个主动器的完成事件队列中。

主动器接口也定义了称为handle_events()的主入口方法，应用程序用它来运行主动性的事件循环[⊖]。正如在实现活动（3.1）中介绍的那样，这个方法调用异步事件多路分解器，后者等待完成事件队列中到达完成事件。应用程序也可使用超时参数来限定等待完成事件的时间。 236

⊖ 正如“变体”一节所介绍的，多个线程可以同时调用同一个主动器上的handle_events()。这种设计非常适合于I/O受限的应用程序[HPS99]。

这样，如果一直没有事件到达，应用程序不会无限期地阻塞。

异步操作处理器将一个完成事件插入到主动器的完成事件队列后，异步事件多路分解器函数就返回。此时，主动器的handle_events()方法删除队列中的完成事件，用相应的异步完成标记多路分解到异步操作处理器的完成处理程序，并分配处理程序的钩子方法。

(5) *实现主动器接口*。使用5个子活动实现主动器接口：

(5.1) *开发主动器实现层*。在实现活动（4）中介绍的主动器接口抽象将所有多路分解和分配处理委托给一个主动器实现。这和桥模式中的实现层的功能一样，这种设计方法可以透明地实现和配置多种类型的主动器，例如，可以使用不同类型的异步事件多路分解器，如POSIX aio_suspend()［POSIX95］、Win32 GetQueuedCompletionStatus()，或WaitForMultipleObjects()函数[Sol98]来创建一个具体的主动器实现。

➡在例子中Proactor_Implementation类是主动器实现层的基类。在这里不列出其类声明，因为该类的接口和实现活动（4）中的主动器接口基本上是一样的。主要的差别是它的方法是纯虚的，因为它是具体的主动器实现层的基类。 ❑

(5.2) *选择完成事件队列和异步事件多路分解器机制*。主动器实现的handle_events()方法调用一个异步事件多路分解器函数，后者等待异步操作处理器向完成事件队列插入一个完成事件。一旦队列中有了一个完成事件，该函数就返回。可以通过所支持的语义类型区分异步事件多路分解器。语义类型包括：

237

- *FIFO多路分解*。这一类异步事件多路分解器函数等待与完成事件队列相关联的任何异步操作所对应的完成事件，按照这些完成事件的插入顺序删除这些事件。

  ➡使用Win32的GetQueuedCompletionStatus()函数，事件驱动的主动性应用程序可以在一段依应用而定的时间内等待在一个完成端口上发生的完成事件。事件以FIFO顺序删除[Sol98]。❑
- *选择性的多路分解*。这种类型的异步事件多路分解器函数有选择性地等待完成事件的特定子集，当函数被调用时必须显式地传递这些完成事件。

  ➡要向POSIX aio_suspend()函数[POSIX95]和Win32的WaitForMultipleObjects()函数[Sol98]显式地传递一个代表异步操作的数组参数。它们将调用函数挂起一段依应用而定的时间，直到其中至少一个异步操作结束才结束挂起。 ❑

通常完成事件队列和异步事件多路分解器是操作系统已经提供的机制，不需要主动器模式的实现者开发。

GetQueuedCompletionStatus()、aio_suspend()和WaitForMultipleObjects()函数的主要区别在于，后两个函数可以有选择地等待由一个数组参数指定的完成事件，而，GetQueuedCompletionStatus()仅仅等待加入到它的完成端口的队列中的下一个完成事件。而且，POSIX aio_*()函数只能多路分解异步I/O操作，如aio_read()或aio_write()，而GetQueuedCompletionStatus()和WaitForMultipleObjects()可以多路分解其他的Win32异步操作，如定时器和同步对象。

238

➡在Web服务器例子中，使用一个Win32完成端口作为完成事件队列，用GetQueued-CompletionStatus()函数作为它的异步事件多路分解器：

```
BOOL GetQueuedCompletionStatus
```

```
(HANDLE CompletionPort,
LPDWORD lpNumberOfBytesTransferred,
LPDWORD lpCompletionKey,
LPOVERLAPPED *lpOverlapped,
DWORD dwMilliseconds);
```

如实现活动（5.5）中所示，主动器实现中，handle_events()方法使用这个函数从指定的CompletionPort中删除一个完成事件。将传送的字节数作为“输出”参数返回。lpOverlapped参数指向由先前的异步操作传递来的异步完成标记，在实现活动(3.1)中介绍的Stream::async_read()方法中调用的ReadFile()就是这样一种异步操作。

如果端口上没有完成事件结果，本函数阻塞调用线程，等待与完成端口对应的异步操作完成。一旦可以将一个完成事件结果从队列中删除，或者是当由dwMilliseconds指定的时间期满时，GetQueuedCompletionStatus()函数就返回。 ❑

(5.3) *确定如何向完成处理程序多路分解完成事件。*如实现活动（3.1）中介绍的那样，向完成处理程序多路分解完成事件的一个有效的、简洁的策略是使用异步完成标记模式。使用这种策略的话，一旦启动程序调用异步操作，异步操作处理器就会收到用于指导后续完成处理的信息。例如，可以传递句柄以标识一个特定的套接字端点和完成事件队列，传递一个异步完成标记以标识某一完成处理程序。

异步操作完成时，异步操作处理器产生相应的完成事件，将完成事件和异步完成标记关联起来，并将修改后的完成事件插入到相应的完成事件队列中。异步事件多路分解器从它的完成事件队列中删除完成事件后，主动器实现可以作为完成事件的异步完成标记在常数时间O（*1*）内多路分解给指定的完成处理程序。 239

➡正如实现活动（3.1）中介绍的，当在Async_Stream上调用async_read()或async_write()方法时，这些方法分别创建新的Async_Stream_Read_Result或Async_Stream_Write_Result 异步完成标记，并将它们传递给相应的Win32异步操作。当异步操作结束时，Windows NT内核将完成事件插入到由在异步操作调用期间传递的句柄所指定的完成端口。主动器使用异步完成标记将完成事件多路分解给在初始调用时指定的完成处理程序。 ❑

(5.4) *确定如何为指定的完成处理程序分配钩子方法。*主动器的handle_events()方法多路分解给完成处理程序后，它必须为完成处理程序分配相应的钩子方法。执行这种分配操作的一个有效策略是如实现活动（3.1）中所介绍的，将适配器模式［GoF95］和异步完成标记模式结合起来。

➡Async_Stream_Read_Result就是一个适配器，其中complete()方法能为保存在异步完成标记状态中的完成处理程序分配合适的钩子方法:

```
void Async_Stream_Read_Result::complete () {
    completion_handler_->handle_event
        (completion_handler_->get_handle (),
         READ_EVENT, *this);
}
```

要留意如何向handle_event()分配钩子方法传递一个调用它的Async_Stream_

Read_Result对象的引用。这种双分配的交互［GoF95］使完成处理程序可以访问异步操作结果，如传送的字节数或成功/失败的状态。 ❑

(5.5) 定义具体的主动器实现。主动器接口拥有指向一个具体的主动器实现的指针，并转发对它的所有方法调用，如实现活动（4）中介绍的一样。

240

➡这里的具体主动器实现重载了从类Proactor_Implementation中继承来的纯虚方法：

```
class Win32_Proactor_Implementation :
    public Proactor_Implementation {
public:
```

Win32_Proactor_Implementation的构造函数创建完成端口，并将其保存在completion_port_数据成员中。

```
Win32_Proactor_Implementation::
    Win32_Proactor_Implementation () {
        completion_port_ = CreateIoCompletionPort
            (INVALID_HANDLE, 0, 0, 0);
}
```

register_handle()方法将一个HANDLE与完成端口关联：

```
void Win32_Proactor_Implementation::register_handle
    (HANDLE h) {
    CreateIoCompletionPort (h, completion_port_,0,0);
}
```

通过HANDLE调用异步操作所产生的所有完成事件将会被Windows NT操作系统插入到主动器的完成端口中。

下一段代码说明了如何实现handle_events()方法：

```
void Win32_Proactor_Implementation::handle_events
    (Time_Value *wait_time = 0) {
    u_long num_bytes;
    OVERLAPPED *act;
```

该方法首先调用GetQueuedCompletionStatus()异步事件多路分解器函数从完成端口中删除下一个完成事件：

```
BOOL status = GetQueuedCompletionStatus
    (completion_port_, &num_bytes,
     0, &act,
     wait_time == 0 ? 0 : wait_time->msec ());
```

这个函数返回时，从Windows NT操作系统接收到的异步完成标记向下类型转换为一个Async_ Result*：

241

```
    Async_Result *async_result =
        static_cast <Async_Result *> (act);
```

GetQueuedCompletionStatus()返回的完成事件修改async_result中的完成结果数据：

```
    async_result->status (status);
```

```
if (!status)
    async_result->error (GetLastError ());
else
    async_result->bytes_transferred(num_bytes);
```

然后主动器实现中的handle_events()方法调用async_result适配器的complete()方法：

```
async_result->complete ();
```

实现活动(5.4)说明了Async_Stream_Read_Result适配器的complete()方法如何分配具体完成处理程序的handle_event()钩子方法。

最后，如实现活动(3.1)所示，主动器删除由一个异步操作接口方法动态分配的async_result指针。

```
    delete async_result;
}
```

主动器实现的私有部分保存有指向Windows NT完成端口的句柄：

```
private:
    // Store a HANDLE to a Windows NT completion port.
    HANDLE completion_port_;
};
```

❑

(6) *确定应用程序中主动器的数量*。很多应用程序可以只使用一个主动器模式实例。在这种情况下可以像实现活动（4）中所介绍的，使用单件模式[GoF95]实现主动器。这种设计有利于将完成事件的多路分解和分配集中在应用程序的一个地方。

然而，也可以在同一个应用进程中同时运行多个主动器。例如，不同的主动器可以对应于优先级不同的线程。这种设计为异步操作的完成处理提供不同级别的服务质量。

注意，完成处理程序只是在一个主动器实例的各线程中串行化，因此多个线程中的多个完成处理程序可以并行地运行。如果不同线程的完成处理程序需要并发地访问共享状态，这种配置方式还需要使用其他同步机制。可以使用像定界加锁这样的互斥和同步方法。 242

(7) *实现具体完成处理程序*。具体完成处理程序使实现活动（2.3）中介绍的完成处理程序接口具体化，从而定义了与应用有关的功能。实现具体完成处理程序需要关注以下三个子活动：

(7.1) *确定维护具体完成处理程序的状态的策略*。具体完成处理程序可能需要与特定请求有关的状态信息。例如，操作系统可能通知服务器，告知由于传输层的流控制的原因，只有部分文件被异步地写到一个套接字上了。具体完成处理程序则必须发送剩下的数据，直到文件传送完毕或者连接失效。因此它必须知道最初指定的是哪个文件，还要发送多少个字节以及上一个请求开始时文件的位置。

(7.2) *选择用句柄配置具体完成处理程序的机制*。具体完成处理程序调用句柄上的操作，可以使用反应器模式的实现活动（6.2）所描述的两种策略——*硬编码和类属*（generic）——在主动器模式中用完成处理程序配置句柄。在这两种策略中都可使用包装器外观封装由完成处理程序类使用的句柄。

(7.3) *实现完成处理程序的功能*。应用程序的开发人员必须确定当主动器调用服务所对应的钩子方法时，实现该服务的步骤。为了将建立连接的功能和随后的服务处理分开，可以按照接受器-连接器模式将具体完成处理程序分成几类。特别是，服务处理程序实现与应用有关的服务，而接受器和连接器分别代表这些服务处理程序，被动地和主动地建立连接。

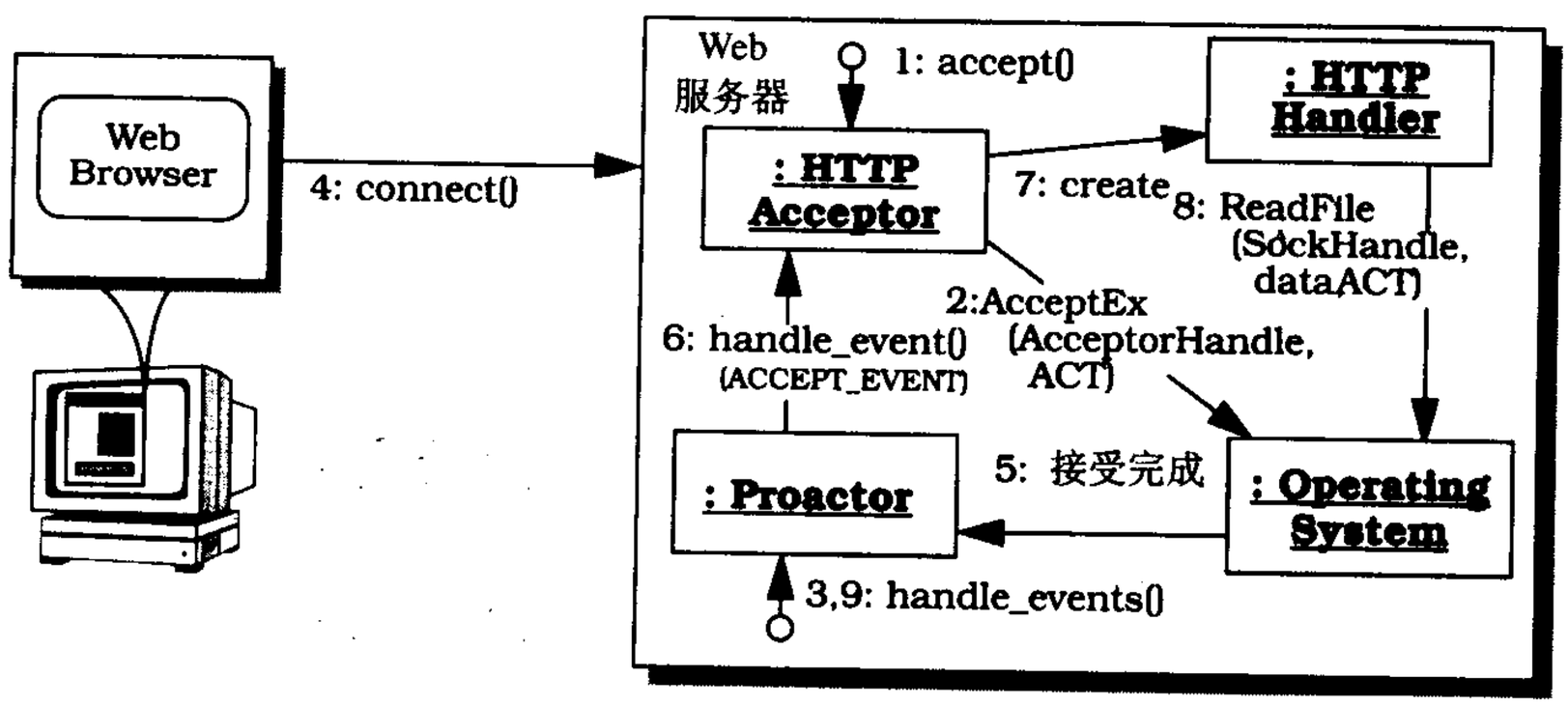
243

(8) *实现启动程序*。在很多主动性应用（如Web服务器例子）中，具体完成处理程序就是启动程序。在这种情况下可以跳过这一实现活动。然而，在应用程序启动阶段通常使用不是完成处理程序的启动程序启动异步服务处理。

8. 已解决的例子

Web服务器使用一些Windows NT的特性，如重叠I/O、完成端口和GetQueuedCompletion-Status()来主动地实现事件多路分解。它采用单方法完成处理程序分配接口策略，这样可以异步地处理多个Web浏览器服务请求。HTTP接受器使用接受器-连接器模式的变体异步地连接和建立HTTP处理程序。每个HTTP处理程序负责异步地接收、处理和回答Web浏览器GET请求，该请求通过完成事件分发到Web服务器的主动器。本例子使用单线程调用异步操作并处理完成事件。显然可以像“变体”一节描述的那样利用多线程来改进这个例子。

Web服务器的main()函数首先进行初始化处理，如建立一个主动器单件、一个Windows NT完成端口和一个HTTP接受器。接受器将被动模式的接受器句柄和主动器单件的完成接端口关联起来。然后Web服务器在其连接处理期间完成图3-18的场景：



244

图 3-18

- 调用HTTP接受器的accept()方法（1）。该方法创建一个异步完成标记，该异步完成标记是具体完成处理程序。
- HTTP接受器的accept()方法担当启动程序的角色，异步调用Win32 AcceptEx()操作（2）。它将异步完成标记和HANDLE参数传递给AcceptEx()，HANDLE既标识接受连接的被动模式的socket端点，也标识当AcceptEx()接受连接完成后Windows NT[⊖]用以将完成事件排队的完成端口。
- 然后Web服务器的main()函数调用反应器（3）的handle_events()方法。该方法运行

⊖ 为了简捷，在本节的余下部分使用术语“Windows NT”或“操作系统”，而不用“异步操作处理器”。

主动器的事件循环，GetQueuedCompletionStatus()异步事件多路分解器函数在该事件循环中被调用。这个函数等待操作系统在异步操作完成执行后将完成事件插入到完成端口中。

- 接着，一个远程Web浏览器连接到Web服务器（4），异步AcceptEx()操作接受连接并产生一个接受完成事件。然后操作系统找到这个操作的异步完成标记，并将它和完成事件关联，并将修改后的完成事件加入到适当的完成端口中（5）。
- 然后，运行在应用程序的事件循环线程中的GetQueuedCompletionStatus()函数删除完成端口中的完成事件。主动器使用与完成事件关联的异步完成标记，向HTTP接受器完成处理程序分配handle_event()钩子方法（6），并传递ACCEPT_EVENT事件类型。
- 为了处理完成事件，HTTP 接受器建立一个将其I/O句柄和主动器的完成端口关联的HTTP处理程序（7）。然后该HTTP处理程序立即调用一个异步操作ReadFile()(8)，以得到Web浏览器发送的GET请求数据。HTTP处理程序将自己作为异步完成标记中的完成处理程序，和I/O句柄一起传递给ReadFile()。异步ReadFile()操作结束时操作系统用完成端口和句柄通知主动器的handle_events()方法。 245
- Web服务器的控制返回到主动器的事件循环（9），在事件循环中调用GetQueued-CompletionStatus()函数继续等待完成事件。

图3-19说明了建立连接和创建HTTP处理程序后，主动性Web服务器为服务一个HTTP Get请求而要做的事情：

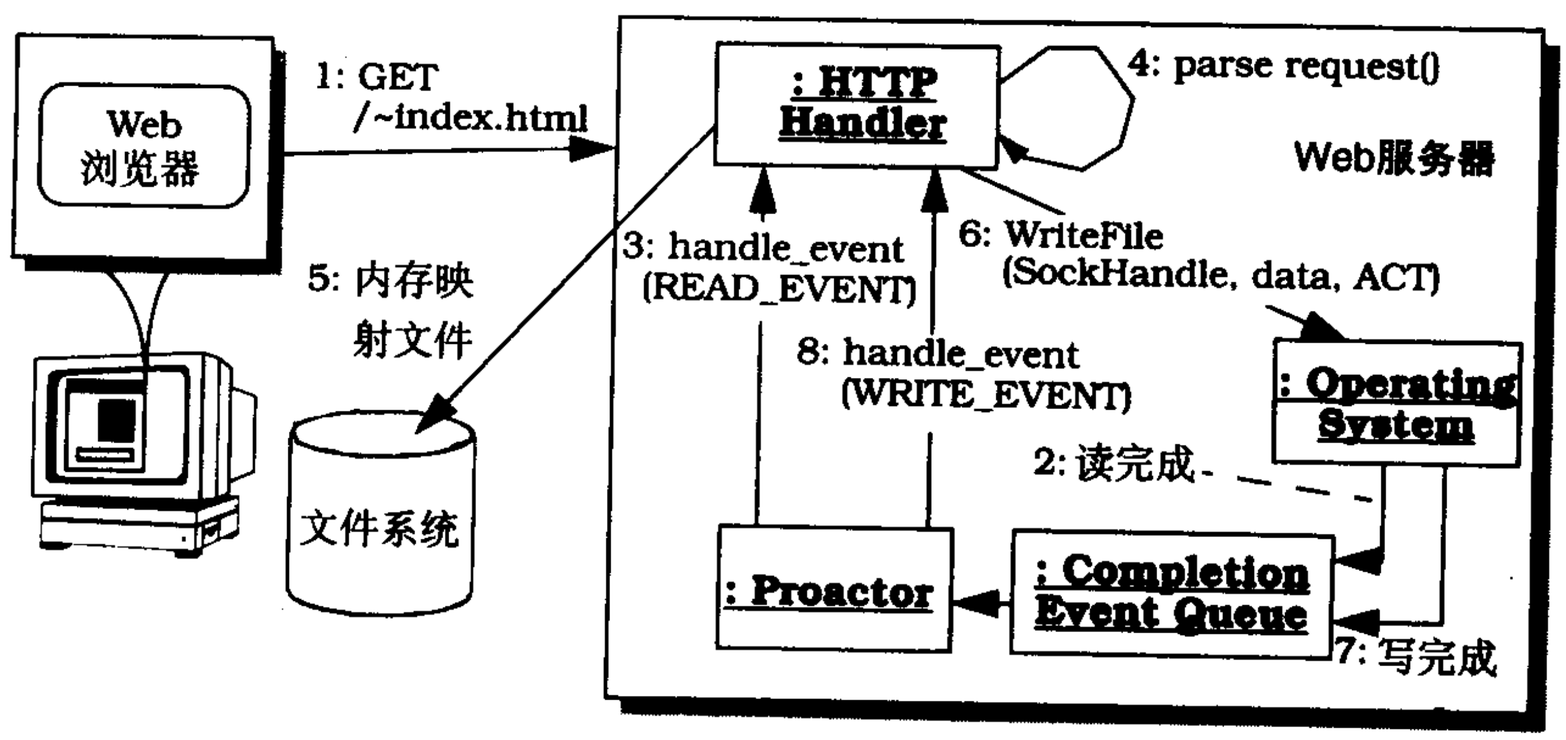


图 3-19

- Web浏览器发送一个HTTP GET请求（1）。
- 在前一个场景中调用的异步ReadFile()操作执行结束，操作系统将*read*完成事件插入到完成端口中（2）。该事件是被GetQueuedCompletionStatus()从队列中删除的，这个函数返回到主动器的handle_events()方法。该方法将完成事件的异步完成标记多路分解给指定的HTTP处理程序，并分配该处理程序的handle_event()钩子方法，向钩子方法传递READ_EVENT事件类型（3）。
- HTTP处理程序分析该请求（4）。在需要时重复步骤（2）~（4），直到异步地接收到了完整的Get请求。

246

- 接收和确认完GET请求后，HTTP处理程序将所请求的文件进行内存映射（5），并调用异步操作WriteFile()通过所建立的连接来传送文件数据（6）。HTTP处理程序将一个标识它自己为一个完成处理程序的异步完成标记传递给WriteFile()，这样在异步操作WriteFile()结束后，主动器可以通知它。
- 异步操作WriteFile()完成后，操作系统在完成端口中插入一个写完成事件。主动器再次使用GetQueuedCompletionStatus()删除完成事件（7）。它使用对应的异步完成标记多路分解给HTTP处理程序，然后分配其handle_event()钩子方法（8），以处理*write*完成事件结果。异步地连续执行（6）~（8），直到将整个文件发送给Web浏览器。

下面解释如何使用“实现”一节中介绍的Completion_Handler类编写Web服务器中的HTTP处理程序。

```
class HTTP_Handler : public Completion_Handler  {
    // Implements HTTP using asynchronous operations.
```

HTTP_Handler继承实现活动（2.3）中定义的Completion_Handler基类，该基类采用“单方法”分配接口策略。采用这种设计方法，可以使主动器单件在异步操作ReadFile()和WriteFile()完成时分配其handle_events()钩子方法。每一个HTTP_Handler对象包含下列数据成员：

```
private:
    // Cached <Proactor>.
    Proactor *proactor_;
    // Memory-mapped file_;
    Mem_Map file_;
    // Socket endpoint, initialized into "async-mode."
    SOCK_Stream *sock_;
    // Hold the HTTP Request while its being processed.
    HTTP_Request request_;
    // Read/write asynchronous socket I/O.
    Async_Stream stream_;
```

构造函数保存一个指向被HTTP_Handler使用的主动器的指针：

```
public:
    HTTP_Handler (Proactor *proactor):
        proactor_ (proactor) { }
```

247

当一个Web浏览器连接到Web服务器时，HTTP接受器要调用HTTP处理程序的open()方法：

```
virtual void open (SOCK_Stream *sock) {
    // Initialize state for request.
    request_.state_ = INCOMPLETE;

    // Store pointer to the socket.
    sock_ = sock;

    // Initialize <Async_Stream>.
    stream_.open
        (this, // This completion handler.
         sock_->handle (), proactor_);

    // Start asynchronous read operation on socket.
```

```
        stream_.async_read
         (request_.buffer (), request_.buffer_size ());
    }
```

在open()中，当异步操作ReadFile()、WriteFile()完成后用完成处理程序、句柄和主动器来初始化Async_Stream。然后，调用async_read()操作，并返回到分配它的主动器中。调用栈为空时Web服务器继续运行主动器单件的handle_events()事件循环方法。

异步操作ReadFile()结束后，主动器单件多路分解到HTTP_Handler完成处理程序，并分配其相应的handle_event()方法：

```
virtual void handle_event
     (HANDLE,
      Event_Type event_type,
      const Async_Result &async_result) {
     if (event_type == READ_EVENT) {
         if (!request_.done
              (async_result.bytes_transferred ()))
              // Didn't get entire request, so start a
              // new asynchronous read operation.
              stream_.async_read (request_.buffer (),
                                  request_.buffer_size ());
         else
              parse_request ();
     }
     // ...
}
```

如果没有完整的请求到达，会调用另一个ReadFile()异步操作，而且Web服务器再次返回到其事件循环。然而，收到Web浏览器的一个完整的GET请求后，随后使用的parse_request()方法会将所要的文件映射到内存中，并异步地向Web浏览器写数据： 248

```
void parse_request () {
     // Switch on the HTTP command type.
     switch (request_.command ()) {

     // Web browser is requesting a file.
     case HTTP_Request::GET:
          // Memory map the requested file.
          file_.map (request_.filename ());
          // Invoke asynchronous write operation.
          stream_.async_write (file_.buffer (),
                               file_.buffer_size ());
          break;
     // Web browser is storing file at the Web server.
     case HTTP_Request::PUT:
          // ...
     }
}
```

为了简单和清楚，上面的parse_request()使用了C++的switch语句。可以使用命令（Command）模式［GoF95］或命令处理器（Command Processor）模式[POSA1]使之更具可扩展性。

当异步操作WriteFile()完成时，主动器单件分配HTTP_Handler的handle_event()钩子方法：

```
virtual void handle_event
     (HANDLE, Event_Type event_type,
      const Async_Result &async_result) {
     // ... see READ_EVENT case above ...
     else if (event_type == WRITE_EVENT) {
          if (!file_.done
               (async_result.bytes_transferred ()))
               // Didn't send entire data, so start
               // another asynchronous write.
               stream_.async_write
                (file_.buffer (),file_.buffer_size ());
          else
               // Success, so free up resources...
     }
}
```

249

所有数据均被接收后，HTTP处理程序释放所有动态分配的资源。

Web服务器中的main()函数实现了一个单线程服务器。该服务器首先调用异步接受操作，然后在主动器单件的handle_events()事件循环中等待：

```
// HTTP server port number.
const u_short PORT = 80;

int main () {
     // HTTP server address.
     INET_Addr addr (PORT);

     // Initialize HTTP server endpoint, which associates
     // the <HTTP_Acceptor>'s passive-mode socket handle
     // with the <Proactor> singleton's completion port.
     HTTP_Acceptor acceptor (addr, Proactor::instance ());

     // Invoke an asynchronous <accept> operation to
     // Invoke the Web server processing.
     acceptor.accept ();

     // Event loop processes client connection requests
     // and HTTP requests proactively.
     for (;;)
          Proactor::instance ()->handle_events ();
     /* NOTREACHED */
}
```

当Web浏览器发来的服务请求被操作系统转换为指示事件时，主动器单件调用HTTP_Acceptor的事件处理钩子方法和HTTP_Handler具体事件处理程序，异步地接受连接，接收和处理日志记录。图3-20说明主动性Web服务器中的行为。

可将该图中的主动性处理模型加以扩展以适应多个HTTP处理程序和HTTP接受器处理远程Web浏览器同时发来的请求的情况。例如，每个处理程序/接受器可以调用并发执行的异步操作ReadFile()、WriteFile()和AcceptEx()。如果底层的异步操作处理器有效地支持异步I/O操作，Web服务器的整体性能可以相应地得到有效的提高。

250

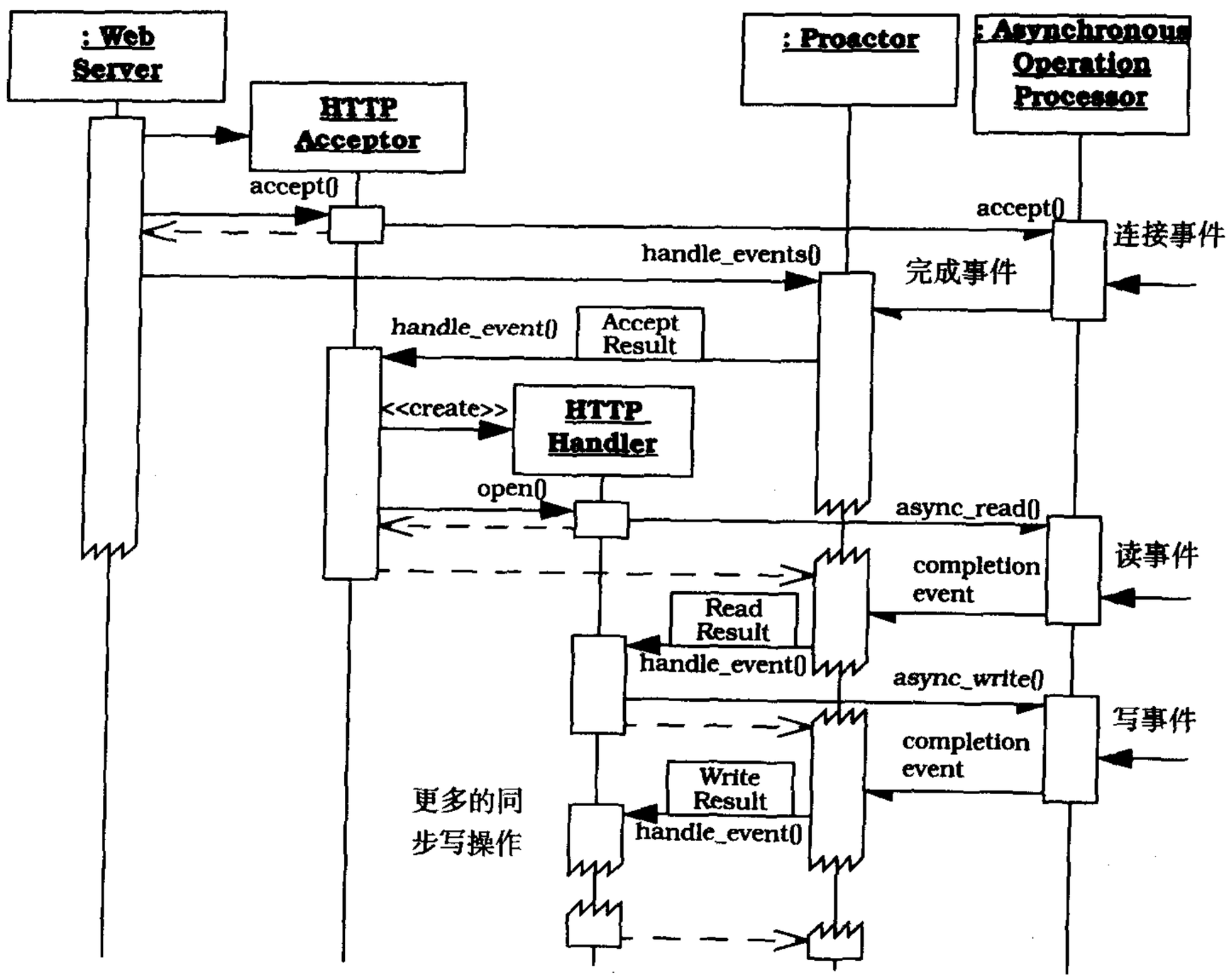


图 3-20

9. 变体

异步完成处理程序。“实现”一节介绍了实现一个在单主动器事件循环线程中向完成处理程序分配完成事件的主动器所需要的活动。一个具体完成处理程序分配后，它借用主动器的线程进行其完成处理。然而，这种设计要求具体完成处理程序只能执行耗时短的同步处理，以免明显地降低应用程序的整体响应能力。

为了解决这个问题，可以要求所有的完成处理程序都作为启动程序，直接调用耗时长的异步操作，而不是同步地进行完成处理。有些操作系统，如Windows NT，明确支持异步过程调用（Asynchronous Procedure Call, APC）。一个APC是一个在调用它的线程的语境中异步执行的函数。调用一个APC时，操作系统将它按顺序保存在线程语境中。下次当线程空闲时，如因一个I/O操作而阻塞时，线程可以运行排队APC。 251

并发的异步事件多路分解器。使用APC的一个缺点是不能有效地使用多CPU，这是因为每个APC运行在一个单线程的语境中。因此一个更具可扩展性的策略是创建一个共享异步事件多路分解器的线程池，这样，一个主动器可以并发地多路分解和分配完成处理程序。对于能有效实现异步I/O的操作系统平台，这种策略的可扩展性很好。

例如，WindowsNT对完成端口[Sol98]进行了优化，这样当来自多个线程的`GetQueuedCompletionStatus()`同时访问时，完成端口的运行效率依然很高[HPS99]。特别是Windows NT内核以后进先出（LIFO）的顺序调度在一个完成端口上等待的线程。这种LIFO协议能确保

等待最短时间的线程被优先调度，从而可以使CPU的缓存相似性（Cache Affinity）[Mog95]最大化。这种最先调度等待时间最短的线程的方法是失效前的刷新Fresh Work Before Stale模式[Mes96]的一个例子。

共享的完成处理程序。启动程序可以同时调用多个共享同一个具体完成处理程序的异步操作[ARSKOO]。然而，共享的处理程序需要准确判别哪个异步操作已经完成。在这种情况下，启动程序和主动器必须合作，在整个异步处理生命周期中跟踪与操作有关的状态信息。

和实现活动（3.1）一样，在这里也可以使用异步完成标记模式来区分每个异步操作——启动程序可以创建一个惟一识别异步操作的异步完成标记（ACT）。然后，当调用了异步操作处理器的一个异步操作时，它将该启动程序的异步完成标记放在传递的异步完成标记之上。当操作执行结束后，且将被主动器处理时，可以将这个“启动程序-异步完成标记”原封不动地传递给共享的具体完成处理程序的钩子方法。在收到一个异步操作的完成结果后，具体完成处理程序可以利用这个启动程序-异步完成标记控制后续的处理。

252

➡为了共享具体完成处理程序，首先在Async_Result类中增加一个启动程序-ACT数据成员和一对set/get方法：

```
class Async_Result : public OVERLAPPED {
private:
    const void *initiator_act_;
    // ....
public:
    // Set/get initiator's ACT.
    void initiator_act (const void *);
    const void *initiator_act ();
    // ...
```

下面修改Async_Stream I/O方法，将启动程序-异步完成标记放置在已有的异步完成标记上。

```
int Async_Stream::async_read (void *buf,
                              u_long n_bytes,
                              const void *initiator_act)
{
    u_long bytes_read;
    OVERLAPPED *act = new // Create the ACT.
        Async_Stream_Read_Result (completion_handler_);

    // Set <initiator_act> in existing ACT.
    act->initiator_act (initiator_act);

    ReadFile (handle_, buf, n_bytes, &bytes_read, act);
}
```

最后，可以在完成事件处理程序的handle_event()方法中通过Async_Result参数得到启动程序-异步完成标记。

```
virtual void handle_event
             (HANDLE, Event_Type event_type,
              const Async_Result &async_result) {
    const void *initiator_act =
        async_result.initiator_act ();
    // ...
}
```

handle_event()方法使用启动程序-异步完成标记来明确后续的处理。 ❑ 253

模拟异步操作处理器。大多数操作系统平台，包括传统版本的UNIX[MBKQ96]和Java虚拟机（JVM），并不支持应用程序的异步操作。不过，可以采用很多技术在这样的平台上模拟异步操作处理器。常用的办法是，和主动对象模式以及其他类型的线程模型一样，使用并发机制在不阻塞启动程序的情况下执行操作。实现多线程异步操作处理器要考虑以下三个活动：

- 操作调用。当调用一个操作时，异步操作处理器必须首先将相应的异步完成标记存储在一个内部表中。可以使用管理者模式实现这一目标［Som97］。
- 异步操作执行。接着在不同于调用者启动程序线程的控制线程中执行操作。一种方式是为每个操作创建一个线程。一个更具可扩展性的方法是使用主动对象（Active Object）模式的线程池变体由异步操作处理器维持一个线程池。这种方法要求启动程序线程在继续进行其他计算之前将操作请求放入队列中。

  每个操作会依次从队列中删除，并在异步操作处理器内部的线程中执行。例如，为了实现异步读操作，当从套接字或者文件句柄中读数据时会阻塞一个内部线程。从调用这些操作的启动程序看来，这些操作是异步执行的，即使在异步操作处理器内操作阻塞于它们的控制线程。
- 操作完成处理。当异步操作完成时，异步操作处理器产生一个完成事件，并将它和在早期调用时保存的相应的异步完成标记结合。然后将修改后的完成事件放入到合适的完成事件队列中。

其他的变体。主动器模式的一些变体和反应器模式的一些变体类似，如集成定时器和I/O事件的多路分解、支持并发的具体完成处理程序。 254

10. 已知使用

Windows NT中的完成端口。Windows NT操作系统提供了有效实现主动器模式的机制[Sol98]。Windows支持不同的异步操作，如超时、接受新的网络连接、读/写文件和Socket、通过Socket连接传送整个文件等等。这样操作系统本身就是异步操作处理器。操作的结果作为完成事件放入到Windows NT的完成端口中，然后由应用程序提供的主动器将完成事件删除和分配。

POSIX的AIO系列异步I/O操作。在一些实时POSIX平台中由aio_*()系列API实现主动器模式[POSIX95]。这些操作系统的功能类似于上述的Windows NT的功能。差别之一是可以用UNIX中的信号实现一个抢先式的异步主动器。在这样的主动器中，信号处理程序可以中断应用程序的控制线程。相反，在Windows NT 中，因为不能中断应用程序线程，所以API并不是抢先式异步的。因此，异步完成子程序要在定义好的Win32函数点处回调。

ACE主动器框架。ADAPTIVE Communication Environment (自适应通信环境ACE)[Sch97]提供了一个可移植的面向对象的主动器框架，其中封装了Windows NT的重叠I/O和完成端口以及POSIX平台上的aio_*() API系列异步I/O机制。ACE提供了一个抽象类ACE_Proactor，它为不同的主动器实现（如ACE_Win32_Proactor和ACE_POSIX_Proactor）定义了一个公共接口。可以分别使用不同的异步事件多路分解器，如GetQueuedCompletionStatus()和

aio_suspend()实现这些主动器。

操作系统设备驱动程序的中断处理机制。通常使用主动器模式完善操作系统内核的结构，操作系统内核调用由异步中断驱动的硬件设备的I/O操作。例如，应用程序可以将一个数据包写入到驻留内核的设备驱动程序，由设备驱动程序传递给硬件设备，后者再异步地发送数据。设备完成其传递后，产生一个硬件中断，通知设备驱动程序中相应的处理程序。然后设备驱动程序处理中断，如果应用程序还有其他数据，则可能还要启动另一次异步传送。

255

通过语音信箱的电话呼叫启动。下面介绍主动器模式的一个实际生活应用场景：你给一个朋友打电话，但她不在。因此你在她的语言信箱中留言，让她回来后给你回电话。套用主动器模式的话，你本人是一个启动程序，调用异步操作处理器（你朋友的语言信箱）的异步操作，通知你的朋友。在等待朋友回话时，你可以做其他事，如阅读本书。你的朋友听到语言信箱中的信息（相当于异步操作的完成）后，她担当主动器的角色，给你回话。你与她交谈时，你就是“处理”她的“回调”的完成处理程序。

11. 结论

主动器模式有下列**优点**：

事务分离。主动器模式将与应用程序无关的异步机制和与应用有关的功能分开。与应用无关的机制就成了可重用的组件，该组件知道如何多路分解与异步操作有关的完成事件、分配由具体完成处理程序定义的合适的回调函数。类似地，在具体完成处理程序中与应用有关的功能负责执行特定类型的服务，如HTTP处理。

可移植性。主动器模式使其接口被重用（这种重用独立于底层的执行事件多路分解的操作系统调用），从而改善了应用程序的可移植性。系统调用检测和报告可能同时发生在各事件源上的事件。事件源可以包括I/O端口、定时器、同步对象、信号等等。例如，在实时POSIX平台中，由aio_*()系列API提供异步I/O函数[POSIX95]。类似地，在Windows NT中，使用完成端口和重叠I/O来实现异步I/O[MDS96]。

256

并发机制的封装。将主动器和异步操作处理器分开的一个好处是应用程序可以用不同的并发策略配置主动器，而不会影响其他应用程序组件和服务。

将线程化与并发控制分开。异步操作处理器代表启动程序执行耗时长的操作。因此，应用程序并不需要创建很多线程以增加并发性。这样应用程序可以独立于线程策略而改变并发策略。例如，一个Web服务器可以只为每个CPU分配一个线程，但要通过异步I/O同时为更多的客户机服务。

性能。多线程操作系统在多个控制间循环进行语境切换。当执行语境切换的时间为常量时，如果操作系统切换到一个空闲的（idle）线程，则依次循环经过大量线程耗时很长，会明显地降低应用程序的性能[⊖]。例如，线程可能轮询操作系统的完成状态，这种做法效率不高。主动器模式可以只激活那些有待处理事件的逻辑控制线程，从而避免了相应的语境切换的开销。例如，如果没有要处理的GET请求，Web服务器没必要激活一个HTTP处理程序。

简化应用程序的同步。一旦具体完成处理程序不用创建别的控制线程，编写应用程序时就

⊖ 一些老的操作系统有这个特性，而大多数的现代操作系统则没有这个特性。

可以少考虑或者不考虑同步问题。可以和在常规的单线程环境中一样编写具体完成处理程序。例如，一个Web服务器的HTTP处理程序可以通过异步操作访问磁盘，如使用Windows NT的TransmitFile()函数[HPS99]，这样就不需要产生新的线程。 257

主动器模式存在如下**不足**：

应用范围受到限制。如果操作系统原本就支持异步操作，则可以很高效率地应用主动器模式。然而，如果操作系统并不提供这类支持，可以在实现中使用多线程模拟主动器模式的语义。例如，可以通过分配一个线程池来处理异步操作。但这种设计并不如操作系统的天然支持有效，因为它增加了同步和语境切换的开销，而没有增加应用级的并行性。

编程、调试和测试复杂性。由于操作的调用和完成在时间和空间上是分离的，所以难以编写使用异步机制的应用程序和更高层的系统服务。类似地，没必要使操作只按照某种处理顺序在某个预先规定好的地方开始运行，它们可以以非确定的顺序进行，从而使开发者难以理解。

用主动器模式写的应用程序同样难以调试和测试，因为被逆转的控制流在主动性框架基础设施和与应用有关的处理的方法回调之间振荡。这增加了在调试器中单步跟踪框架运行特性的难度，因为应用程序开发者可能不理解或者根本没有接触过主动性框架的代码。

异步地调度、控制和取消正在执行的操作。启动程序可能无法控制异步操作处理器执行的异步操作的调度顺序。因此，如果可能的话，一个异步操作处理器可以使用策略模式[GoF95]使启动程序能对异步操作赋予优先级或者取消执行异步操作。然而设计一种完全可靠和高效的取消所有异步操作的方案是很困难的，因为异步操作可能在被取消运行之前就完成了。 258

参见

主动器模式与观察者模式[GoF95]、出版者-订阅者模式[POSA1]有关。后两种模式中，目标变化后，所有相关的组件都会得到通知。在主动器模式中，来自于多个事件源的完成事件发生时，完成处理程序会自动得到通知。一般来说，主动器模式用于多路分解多个异步分发的完成事件源到相应的完成处理程序，而观察者或订阅者通常与单个事件源有关。

可以认为主动器模式是同步反应器模式的异步变体。反应器模式负责多路分解和分配多个事件处理程序，这些事件处理程序是在可能非阻塞地同步地调用操作时被激活的。相反，主动器模式支持对由于异步执行的操作的完成而激活的多个事件处理程序的多路分解和分配。

领导者/追随者以及半同步/半异步模式是另外两种可以同步地多路分解和处理不同类型事件的模式。在能有效支持异步I/O的平台上，主动器模式比其他模式的效率更高。但是，主动器模式难以实现，因为它的组成部分较多，也难以理解。由于主动器中同时有被逆转的控制流和异步性，所以要求应用程序开发者有更多的经验以高效地使用和调试。

主动对象模式将方法执行和方法调用分开，主动器模式也类似，因为异步操作处理器代表启动程序异步地执行操作。因此，这两种模式都可用于实现异步操作。在能有效地支持异步I/O的操作系统中通常使用主动器模式而不是主动对象模式。

责任链[GoF95]模式将事件处理程序和事件源分开。类似地，主动器将启动程序与完成处理程序分开。但在责任链模式中，事件源事先并不知道哪个处理程序将被执行。在主动器模式中，启动程序能完全控制目标完成处理程序。可以建立一个完成处理程序作为进入由外部工厂动态配置的责任链的入口点，这样就将这两种模式结合起来了。 259

目前，Java实现并不支持类主动器（Proactor-like）的事件处理方案，因为java.io并不支持异步I/O。在基本的java实现中，对I/O操作的阻塞甚至能导致整个Java虚拟机（JVM）的阻塞——I/O操作阻塞当前的线程，因为可以在用户空间实现多线程，所以操作系统会认为运行JVM的任务是阻塞的，故调度其他操作系统进程而不是其他JVM线程。

解决这一问题的更成熟的Java实现方法是在本机代码级实现异步I/O——进行阻塞调用的线程被阻塞，但其他的线程还能继续运行。随后回调被阻塞的线程，或者该线程等待阻塞调用返回。但应用程序并不能直接这样做，因为目前的JDK库中并没有异步I/O。这一点在目前正在开发的下一代Java I/O系统中将会有所改变，将会出现称为java.nio的包或其他类似的包[JSR51]。

一些程序设计语言（如Scheme）支持连续性（continuation）。可以在单线程程序中使用连续性，这样一个函数调用序列可以在阻塞时撤回其运行时的调用栈，而又不丢失调用栈的执行历史。在主动器模式的语境中，可将由异步操作调用到完成处理程序的后续处理的间接控制转移建模为一个连续性。

致谢

Tim Harrison、Thomas D.Jordan和Irfan Pyarali是主动器模式的最初版本的合作者。Irfan也对本版本提出了一些有用的建议。谢谢Ralph Johnson的建议，他的建议对改进这个模式很有帮助，他还指出了本模式与程序设计语言连续性特性的关系。

260

3.3 异步完成标记

异步完成标记（Asynchronous Completion Token, ACT）设计模式可以使应用程序能有效地多路分解和处理对调用服务的异步操作的响应。

1. 别名 主动多路分解（Active Demultiplexing）[PRS+99]，“Magic Cookie”。

2. 例子

考虑一个由一组Web服务器组成的大规模分布式电子商务系统。这些服务器响应来自Web浏览器的请求，存储和获取各种类型的数据。这样的电子商务系统的性能和可靠性是成功应用的关键。

例如，在基于Web的股票交易系统中，重要的是要确保当前的股票报价，以及后续的买卖定单能有效地和可靠地传送。因此，必须仔细监控电子商务系统中的Web服务器，以确保它们为用户提供必需的服务质量。自主管理工具可以将管理事件从电子商务系统的Web服务器回传给管理应用程序，从而解决这一问题。如图3-21所示。

261

系统管理员可以使用这些管理工具、应用程序和事件来监控、可视化和控制电子商务系统[PSK+97]中Web服务器的状态和性能。

一般地，一个管理应用程序使用出版者-订阅者模式[POSA1]订阅一个或多个管理工具来接收不同类型的事件，如报告Web浏览器与Web服务器建立新的连接的事件。当管理agent检测到与管理应用程序有关的活动时，它向管理应用程序发送完成事件，然后由管理应用程序处理这些事件。

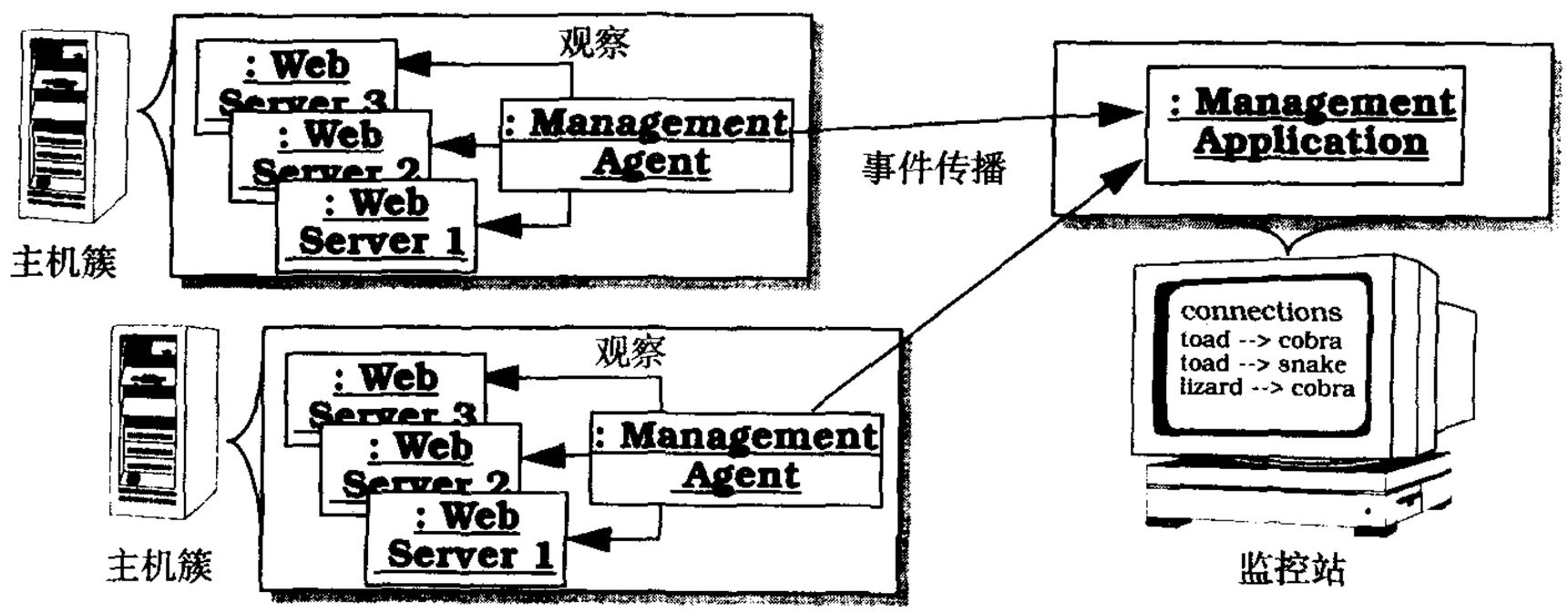


图 3-21

但是，在大规模的电子商务系统中，管理应用程序可以对很多管理代理调用订阅操作，请求对发生的很多不同类型的事件进行报告。而且，在管理应用程序中可以使用专门的完成处理程序以不同的方式处理不同类型的事件。应用程序用这些处理程序对事件进行响应，这些响应包括更新显示、在数据库中记录事件或自动检测性能瓶颈和系统失效。

管理应用程序匹配其订阅操作和相应的完成事件的一种方式就是为在管理代理中调用的每个事件订阅操作创建一个独立的线程。每个线程同步阻塞，等待来自于代理的完成事件，这些完成事件是对原先的订阅操作的响应。使用这种同步设计，处理管理代理事件响应的完成事件可以被显式地存储在每个线程的运行时栈中。

不过，这种同步多线程的设计方法同样存在着在第3.1节（反应器模式中的“例子”一节）中所描述的语境切换、同步和数据移动而使性能下降的缺点。因此，应该使管理应用程序能异步地激活订阅操作。在这种情况下，管理应用程序必须要有效地和可扩展地向相应的完成处理程序多路分解管理代理的完成事件，从而使管理应用程序在得到代理的通告时能做出正确的反应。 262

3. 语境

一个事件驱动的系统，其中应用程序异步地调用服务操作，并处理相应的服务完成事件响应。

4. 问题

当一个客户机应用程序异步地调用对一个或几个服务的操作请求时，每个服务通过一个完成事件将其响应返回给应用程序。然后应用程序必须向相应的处理程序多路分解该事件。处理程序可以是一个函数或对象，用于处理包含在完成事件中的异步操作响应。要有效地解决这个问题，需要考虑以下三个强制条件：

- 一个服务可能不知道客户机应用程序异步地调用其操作时所处的语境。例如，一个客户机可以为每个操作指派一个独立的线程，而另一个客户机可能在一个线程中处理所有操作。这种缺乏语境的情况使得服务难以知道客户机为了多路分解和处理完成事件而需要哪些信息。因此，客户机应用程序而不是服务器必须负责确定如何将完成事件多路分解给被指派去处理它们的处理程序。

  ➡在电子商务系统的例子中，管理代理服务不知道也不需要知道管理应用程序是如何多

路分解和处理不同的完成事件的，这些完成事件来自于管理代理，是对异步订阅操作的响应。 ❑

- 要确定在异步操作完成执行后客户机如何多路分解和处理完成事件，需要客户机应用程序和服务之间的通信，这种通信开销应尽可能地小。对于延迟受限的客户机应用程序以及与服务之间通过有限带宽连接的客户机应用程序来说，通信开销最小化非常重要。

  ➡在电子商务系统中，管理应用程序和代理服务应该有最少的交互，如调用异步订阅操作的交互和为每个完成事件响应的交互。而且，除了操作的输入参数和返回值之外，为向处理程序多路分解完成事件而要传送的数据应尽可能的少。 ❑

263

- 当服务响应到达了一个客户机应用程序时，应用程序应在尽可能少的时间内向处理异步操作响应的处理程序多路分解完成事件。

  ➡一个大规模的电子商务应用程序可能有几百个Web服务器和管理代理，几百个Web浏览器同时连接和有相应大量的异步订阅操作和完成事件。将一个完成事件响应与原先的异步操作请求联系起来要搜索一个很大的表，这种方式会显著降低管理应用程序的性能。 ❑

5. 解决方案

将表示*启动程序*（initiator）如何处理服务响应的信息和客户机启动程序对一个服务调用的每个异步操作一起传送。操作完成后将这些信息返回给启动程序，使用这些信息可以高效地多路分解该响应，使启动程序能相应地处理它。

具体地说，就是为每个由客户机启动程序对一个服务调用的每个异步操作建立一个*异步完成标记*。异步完成标记包括能惟一标识*完成处理程序*的信息，该完成处理程序是一个函数或对象，负责处理对该操作的响应。将异步完成标记和操作一起传递给服务。服务保存异步完成标记，但不修改异步完成标记。服务对启动程序做出响应时，响应中包括原先传送的异步完成标记。这样启动程序可以使用异步完成标记来识别用于处理来自原先的异步操作的响应的完成处理程序。

6. 结构

异步完成标记模式的结构中有4个组成部分：

服务提供一些能被异步访问的功能。

264

➡管理代理为电子商务系统提供一个分布式的管理和监控服务。 ❑

客户机*启动程序*异步调用服务的操作。它还将这些操作所返回的响应多路分解给指定的*完成处理程序*。完成处理程序是应用程序中的一个函数或对象，它负责处理服务响应。如图3-22所示。

➡在电子商务系统中，管理应用程序调用管理代理中的异步操作订阅不同类型的事件。然后当注册的事件发生时，管理代理异步地向管理应用程序发送完成事件响应。管理应用程序中的完成处理程序处理这些完成事件，修改其GUI显示，并执行其他的动作。 ❑

一个*异步完成标记*（ACT）中包含了标识某启动程序的完成处理程序的信息。当启动程序调用一个操作时，它将异步完成标记传递给服务；当异步操作完成时，服务将未改变的异步完成标记返回给启动程序。然后，启动程序将异步完成标记高效地多路分解给相应的完成处理程

序，该完成处理程序处理来自原来的异步操作的响应。服务中保存有一组异步完成标记，用于处理由启动程序同时调用的多个异步操作。如图3-23所示。

类 启动程序	协作者 • 服务 • 异步完成标记
责任 • 异步地调用服务的操作 • 定义完成处理程序	

类 服务	协作者
责任 • 提供应用程序的功能	

图 3-22

类 完成处理程序	协作者
责任 • 为处理异步操作的结果定义一个接口	

类 异步完成标记	协作者 • 完成处理程序
责任 • 标识完成处理程序，启动程序用该完成处理程序处理对服务器调用异步操作后的响应	

265

图 3-23

➡在电子商务系统中，管理应用程序的启动程序可以创建一组异步完成标记，它们作为完成处理程序表的索引，或者直接指向完成处理程序。但管理代理服务只是将异步完成标记不加改变地返回给管理应用程序的启动程序。 ❑

类图3-24说明异步完成标记模式的各组成部分及其他关系：

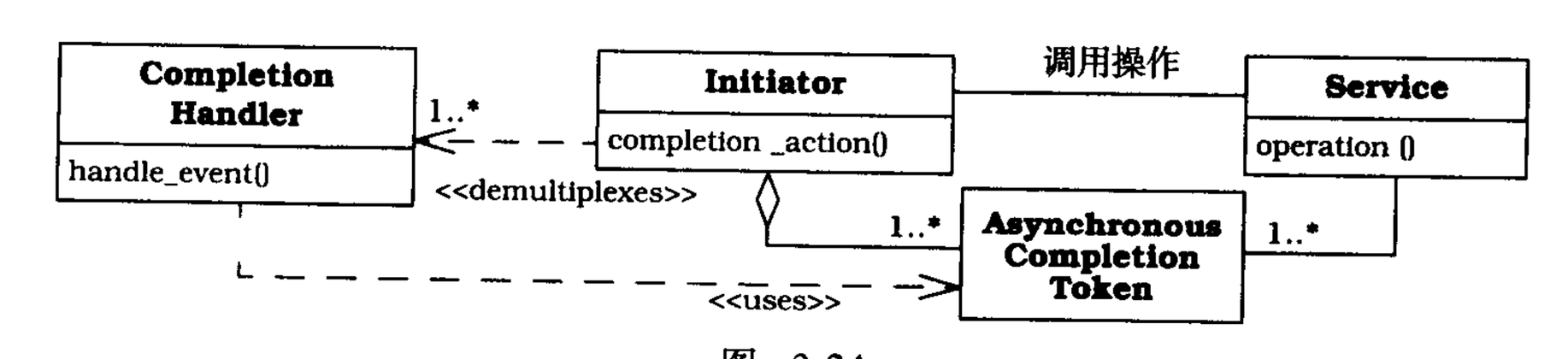


图 3-24

7. 动态特性

异步完成标记模式中有以下几种交互（如图3-25）：

- 调用服务的异步操作之前，启动程序创建一个异步完成标记，该异步完成标记标识与操作有关的完成处理程序。
- 调用对服务的操作时，启动程序将异步完成标记传递给服务。
- 在服务执行异步操作时，启动程序继续调用其他操作或处理响应。
- 异步操作完成时，服务向启动程序发送一个包含有初始异步完成标记的响应。启动程序将

266 异步完成标记多路分解给完成处理程序由它响应异步操作并执行与应用相关的处理。

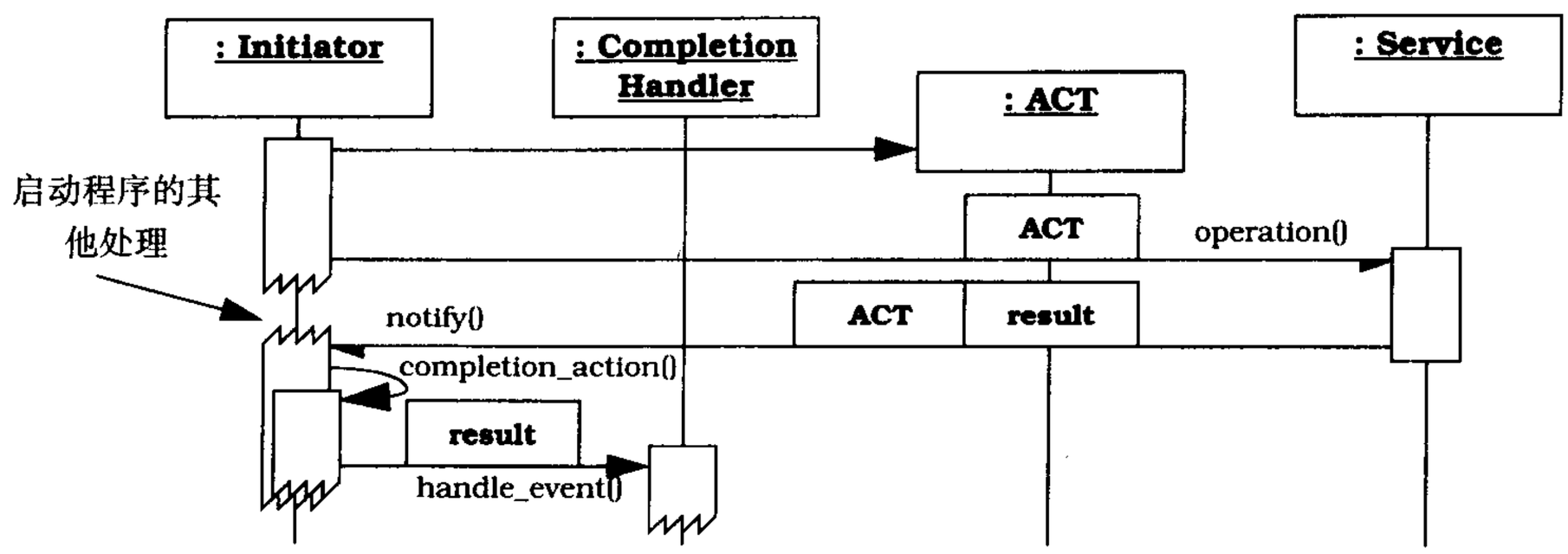


图 3-25

8. 实现

实现异步完成标记模式包括6个活动。这些活动与启动程序、服务和完成处理程序的实现都有关系，在其他模式如代理模式[GoF95][POSA1]、主动器模式或反应器模式中都有体现。因此本节主要讨论异步完成标记的实现、启动程序和服务之间交换异步完成标记所用的协议，以及有效地向相应的完成处理程序多路分解异步完成标记时采取的步骤。

(1) *定义异步完成标记的表示法*。异步完成标记的表示对启动程序和完成处理程序应该是有意义的，但对服务是不透明的。有三种常用的异步完成标记表示法。

- *指针异步完成标记*。通常用指向程序设计语言中的结构的指针表示异步完成标记，如C++中指向完成处理程序对象的指针，或者Java中对完成处理程序对象的引用。例如，在C++中，当启动程序启动了服务上的一个操作时，它可以创建一个完成处理程序异步完成标记对象，将其地址转换成void指针，通过异步操作调用将该指针传递给服务。指针异步完成标记主要用于在运行于相对同构的平台上的启动程序和服务之间传递异步完成标记，指针类型的字节数相同的平台就是一种相对同构的平台。因此，需要使用一些可移植性的特性，如typedef或者宏，以确保在异构分布式系统中统一表示指针。

267

- *对象引用异步完成标记*。为了简化在异构系统中对异步完成标记的使用，可以将异步完成标记表示为在分布式对象计算中间件（如CORBA[OMG98c]）中定义的对象引用。CORBA对象引用为分布式的启动程序和服务之间的异步完成标记通信提供了一个标准的、可移植的和可互操作的手段。当接收到来自服务的一个对象引用异步完成标记后，启动程序可以使用CORBA_narrow()操作将异步完成标记转换为对它来说有意义的完成处理程序类型。当然，如果不使用中间件，对象引用就不是一种表示异步完成标记的可行的方式。
- *索引异步完成标记*。也以将异步完成标记表示为对启动程序可以访问的完成处理程序表的索引。当接收到服务的一个响应后，启动程序使用异步完成标记来查找表，并访问相应的完成处理程序。对于像FORTRAN这样的不支持对象引用和指针的语言来说，这种表示法特别有用。使用数据库标识符或对内存映像文件的偏移量可以使将异步完成标记与其生命

期跨越多个进程的持久完成处理程序结合起来。和实现活动（2）中介绍的一样，索引异步完成标记对于改善启动程序的健壮性也是很有用的。

定义异步完成标记的表示法时要考虑的另一个特性是它是否支持启动程序向相应的完成处理程序多路分解。通过使用命令模式或适配器模式[GoF95]等常用的模式，异步完成标记可以为启动程序提供一个统一的接口。具体的异步完成标记实现就是将这些接口映射为某一完成处理程序所特有的接口，在实现活动（6）中有介绍。

➡下面的抽象类定义的C++接口可以作为很多指针异步完成标记的基类。它定义了一个纯虚方法handle_event()，启动程序用它来分配一个特定完成处理程序，以处理来自某一异步操作的响应：

```
class Completion_Handler_ACT {
public:
    virtual void handle_event
        (const Completion_Event &event) = 0;
};
```

268

应用程序开发者可以从Completion_Handler_ACT中派生一个子类，并重载其钩子方法handle_event()以调用特定的完成处理程序，从而可以定义新的完成处理程序类型。在“已解决实例”一节介绍的电子商务系统说明了这种实现策略。 ❑

(2) *选择在启动程序中保存异步完成标记的策略*。启动程序可以异步调用多个服务。要支持这种特性，就要求启动程序选择下面的策略之一，以保存异步完成标记：

- *隐含异步完成标记*。在这种策略中，将异步完成标记定义为启动程序的完成处理程序对象的地址，从而可以被“隐含地”保存。通常在指针或对象引用异步完成标记中使用这种策略。“已解决的例子”一节中解释了隐含异步完成标记的用法。隐含异步完成标记的时间和空间效率很高，因为它们只是指向相应的完成处理程序的内存地址。不过，和下面介绍的显式异步完成标记相比，隐含异步完成标记的健壮性和安全性不好。
- *显式异步完成标记*。在这种策略中，启动程序在一个明确的数据结构中保存异步完成标记，如使用管理者模式[Som97]或主动多路分解表[PRS+99]组织一个完成处理程序表来存放异步完成标记。这种策略适用于索引异步完成标记。在启动程序中使用显式数据结构保存异步完成标记的好处是能提高健壮性和可靠性。例如，如果因为崩溃或者挂起失败而使服务不返回响应，那么要在表中找到相应的异步完成标记并全部释放。

另外，如果启动程序不相信服务一定会不加修改地返回原先的异步完成标记，可以在显式数据结构中存储额外的信息来认证启动程序中是否真正存在返回的异步完成标记。然而，这种认证检查会增加查找异步完成标记的开销以及将其多路分解给启动程序的完成处理程序的开销。 269

无论采用哪种策略，当不再需要异步完成标记资源时，启动程序要负责释放与异步完成标记有关的任何资源。可以使用对象生命期管理模式[LGS99]管理对异步完成标记资源的可靠删除。

(3) *确定如何将异步完成标记从启动程序传递给服务*。启动程序可以使用两种策略传递异步完成标记和异步服务操作请求。

- *隐含参数通常被透明地存储在传递给服务的语境或环境中*。CORBA语境参数和GIOP服务

语境域[OMG98c]就是两个隐含参数的例子。

- 显式参数定义在异步操作中，“已解决的例子”一节说明了如何将异步完成标记作为显式参数传递给服务操作。

(4) 确定在服务中保存异步完成标记的策略。服务接收到一个异步完成标记后，在执行由启动程序调用的操作时，它必须保持异步完成标记。有两种策略用于在服务中保持异步完成标记：

- 如果服务是同步执行的，那么当服务处理操作时，可以简单地将异步完成标记驻留在服务的运行时栈中。如果服务和启动程序运行在不同的线程或进程中，可以同步执行服务操作，但仍向启动程序提供一种异步程序设计模型。
- 如果服务异步地处理启动程序操作，它可能需要同时处理多个请求。在这种情况下必须在数据结构中维持一个异步完成标记，该数据结构驻留于任何服务的运行时栈范围之外。可以使用管理者模式[Som97]来组织异步完成标记集。

(5) 确定异步完成标记可用的次数。启动程序和服务都可以多次使用同一个异步完成标记：

- 一般来说，启动程序为每个异步调用传递一个单独的异步完成标记。然而启动程序可以对一个服务多次调用某个异步操作。对于每次调用，它可以指派同一个完成处理程序来处理操作的响应，从而使创建和销毁异步完成标记的开销最少。类似地，启动程序可以对多个服务实例上的同一操作指派同一完成处理程序来处理来自所有实例的响应。
- 服务只返回一次异步完成标记和来自异步操作的响应。不过，一旦发生了一个特定事件，服务也可使用同一个异步完成标记发出一系列的响应。在这种情况下，来自一个异步操作调用的异步完成标记可以被多次返回给启动程序。

➡在电子商务系统中，管理应用启动程序可以订阅带有多个代理服务的同一个异步完成标记，并用它来多路分解和分配这些管理代理的响应（如“已建立的连接”完成事件），而且，每个agent可以多次返回该异步完成标记，如当它检测到一个新连接时建立。 ❑

(6) 确定启动程序将异步完成标记多路分解给完成处理程序钩子方法的策略。启动程序负责将异步操作完成事件高效地多路分解给它们的完成处理程序。当异步操作完成时，可以使用两种策略将一个异步完成标记多路分解给相应的完成处理程序：

- 排队的完成事件。在这种策略中，由一个服务，或者当服务在远程时由一个本地服务代理将异步完成标记置于一个完成事件队列中。启动程序从完成事件队列中删除该异步完成标记，并能用封装在异步完成标记中的信息控制其后续的处理。如“已知使用”一节所述［Sol98］，Windows NT中的一些特性，如重叠I/O、完成端口和`GetQueuedCompletionStatus()`函数，就使用了这种策略。

从完成事件队列中得到完成事件和相应的异步完成标记后，启动程序将异步完成标记多路分解给相应的完成处理程序。此时，它可以分配异步完成标记的钩子方法，如在实现活动（1）中所介绍的`handle_event()`方法。该钩子方法处理包含异步操作响应的完成事件。主动器模式中的实现活动（5.2）介绍了这种策略。

270

• 回调。在这种策略中，启动程序向服务传递一个回调函数或对象[⊖]。当异步操作完成时，服务或者本地服务代理可以调用回调函数。可以将异步完成标记作为参数返回给回调函数或对象，并进行向下类型转换，以找出进行后续高效处理的完成处理程序。例如，如上所述，可以调用异步完成标记的handle_event()方法，并传递完成事件。 271

一般来说，可以使用一个回调处理程序对象来有效地多路分解和处理不同类型的完成事件。因此，在启动程序中可以使用单件模式[GoF95]实现回调处理程序。

可以同步或异步地向启动程序发起回调。在异步策略中，通过中断或信号处理程序[POSIX95]来产生回调。因此，启动程序并不需要通过在事件循环中阻塞而显式地等待通知。在同步策略中，驻留启动程序的应用程序通常在反应性或主动性事件循环中等待。当服务的响应返回时，将它分配到相应的回调函数上。

➥电子商务系统说明了如何使用指针异步完成标记实现同步回调对象策略。这里定义了一个通用的回调处理程序类，它使用实现活动（1）中定义的Callback_Handler_ACT。

当管理代理返回完成事件时，该类的completion_event()方法启动合适的管理应用完成处理程序进行处理。

```
class Callback_Handler {
public:
    // Callback method.
    virtual void completion_event
        (const Completion_Event &event,
         Completion_Handler_ACT *act) {
        act->handle_event (event);
    };
};
```

调用一个服务的异步操作时，启动程序将回调处理程序实例的引用作为参数和异步完成标记一起传递给服务。异步操作完成时，服务或本地服务代理同步地分配回调处理程序的completion_event()方法，将完成事件和管理代理服务中的异步完成标记一起传递给管理应用程序。 272

然后completion_event()方法将接收的异步完成标记多路分解给handle_event()钩子方法。然后，这个方法对异步操作的响应进行完成处理程序处理。 ❑

请留意上述两种多路分解策略是如何让启动程序有效地处理许多不同类型的完成事件的。特别是不管用Completion_Handler_ACT的子类表示的完成处理程序有多少，对异步完成标记的多路分解只需要常量时间$O(1)$。

9. 已解决的例子

在电子商务系统中，系统管理员使用管理应用程序和管理代理来显示和记录所有在Web浏览器和Web服务器之间建立的连接。另外，管理应用程序显示和记录每次文件传送，因为HTTP1.1协议允许在一个连接上多路复用多个GET请求[Mog95]。

下面首先定义一个管理代理代理，管理应用程序用它来异步地订阅完成事件。然后说明如何根据电子商务系统中发生的完成事件的类型而专门定义具体的异步完成标记。最后，实现

⊖ 反应器模式中的实现活动（1.1）中介绍了回调函数与回调对象的正反两个方面的特性。

main()函数，将所有这些组件组合起来以创建管理应用程序。

管理代理代理[GoF95]。该类定义了管理应用程序可以订阅的事件的类型，以及让管理应用程序异步地订阅回调和管理代理的方法：

```
class Management_Agent_Proxy {
public:
    enum Event_Type { NEW_CONNECTIONS, FILE_TRANSFERS };

    void subscribe (Callback_Handler *handler,
                    Event_Type type,
                    Completion_Handler_ACT *act);
    // ...
};
```

273

代理类是用半对象加协议（Half-Object plus Protocol）[pLoPD1] 模式实现的，它定义了一个对象接口，该接口封装了代理与管理代理之间的协议。当一个特殊Event_Type事件发生时，管理代理向Management_Agent_Proxy()返回相应的完成事件。然后，该代理调用Callback_Handler的completion_event()方法。该方法返回一个指向由管理应用程序先前传递给subscribe()的指向Completion_Handler_ACT的指针，正如实现活动（6）中所介绍的。

具体的异步完成标记。电子商务系统中担当启动程序角色的管理应用程序和管理代理服务交换指向Completion_Handler_ACT子类对象的指针，下面的Management_Completion_Handler就是一个这样的Completion_Handler_ACT子类对象：

```
class Management_Completion_Handler :
    public Completion_Handler_ACT {
private:
    Window *window_; // Used to display and
    Logger *logger_; // to log completion events.
public:
    Management_Completion_Handler (Window *w, Logger *l):
        window_ (w), logger_ (l) { }

    virtual void handle_event
        (const Completion_Event &event) {
        window_->update (event);
        logger_->update (event);
    }
};
```

传递给Management_Completion_Handler构造函数的参数标识了具体完成处理程序的状态，管理应用程序利用这些状态来处理完成事件。这两个参数保存在类的内部数据成员中，它们分别指向数据库记录器和GUI窗口，当管理代理通过handle_event()钩子方法传递完成事件时GUI窗口将被更新。正如实现活动（6）中所介绍的一样，由Callback_Hander的completion_event()方法分配该钩子方法。

main()函数。下面的main()函数说明了管理应用程序如何调用管理agent代理上的异步订阅操作，以及如何处理随后的连接和文件传送完成事件响应。为了简化和优化对完成处理程序的多路分解和处理，当管理应用程序订阅一个管理agent代理时，它将一个指针异步完成标记传给

274

Management_Completion_Handler。可以很容易地推广这个例子，使之能处理多个管理agent和其他类型的完成事件。

```
int main () {
```

该应用程序一开始就建立在实现活动（6）中定义的Callback_Handler类的一个实例：

```
Callback_Handler callback_handler;
```

这个Callback_Handler由所有异步订阅操作共享，用于多路分解传来的各种类型的完成事件。

下一步应用程序创建上面描述的Management_Agent代理的一个实例：

```
Management_Agent_Proxy agent_proxy = // ...
```

当产生连接和文件传送完成事件时，该agent将回调callback_handler。

然后应用程序创建一些对象来进行登记和显示完成处理：

```
Logger database_logger (DATABASE);
Logger console_logger (CONSOLE);
Window main_window (200, 200);
Window topology_window (100, 20);
```

有些完成事件将记录在数据库中，其他的被写入控制台窗口。根据事件的不同类型，需要更新不同的图形显示。例如，一个布局窗口可能显示系统的一个图标视图。

main()函数创建两个Management_Completion_Handler对象，分别惟一标识处理连接建立和文件传送完成事件的具体完成处理程序：

```
Management_Completion_Handler connection_act
     (&topology_window, &database_logger);
Management_Completion_Handler file_transfer_act
     (&main_window, &console_logger);
```

275

用合适的指向Window和Logger对象的指针初始化Management_Completion_ Handler对象。当管理应用程序为每类事件异步地订阅Management_Agent_Proxy的callback_handler时，它显式地将指针异步完成标记传递给两个Management_Completion_Handler对象：

```
agent_proxy.subscribe
     (&callback_handler,
      Management_Agent_Proxy::NEW_CONNECTIONS,
      &connection_act);

agent_proxy.subscribe
     (&callback_handler,
      Management_Agent_Proxy::FILE_TRANSFERS,
      &file_transfer_act);
```

注意，如实现活动2中的介绍的一样，Management_Completion_Handler隐含地保存在启动程序的地址空间中。

一旦订阅完成，应用程序就进入事件循环，由来自完成事件的回调来驱动所有后续处理。

```
        run_event_loop ();
    }
```

当管理agent检测到新的连接或文件传送时，它向Management_Agent_Proxy发送相应的完成事件。该代理从完成事件中提取Management_Completion_Handler异步完成标记，使用Callback_Handler的completion_event()方法分配Management_Completion_Handler的handle_events()钩子方法，由该钩子方法处理每个完成事件。例如，可以在GUI窗口中显示文件传送事件，并记录在控制台上，而新的连接建立事件可以显示在系统布局窗口中，并记录在数据库中。

顺序图3-26说明了管理应用程序中各组件之间主要协作关系。为简单起见，这里没有表示出代理、回调处理程序、窗口和登录处理程序等的创建，并且只考虑一个窗口和一个登录处理程序。

276

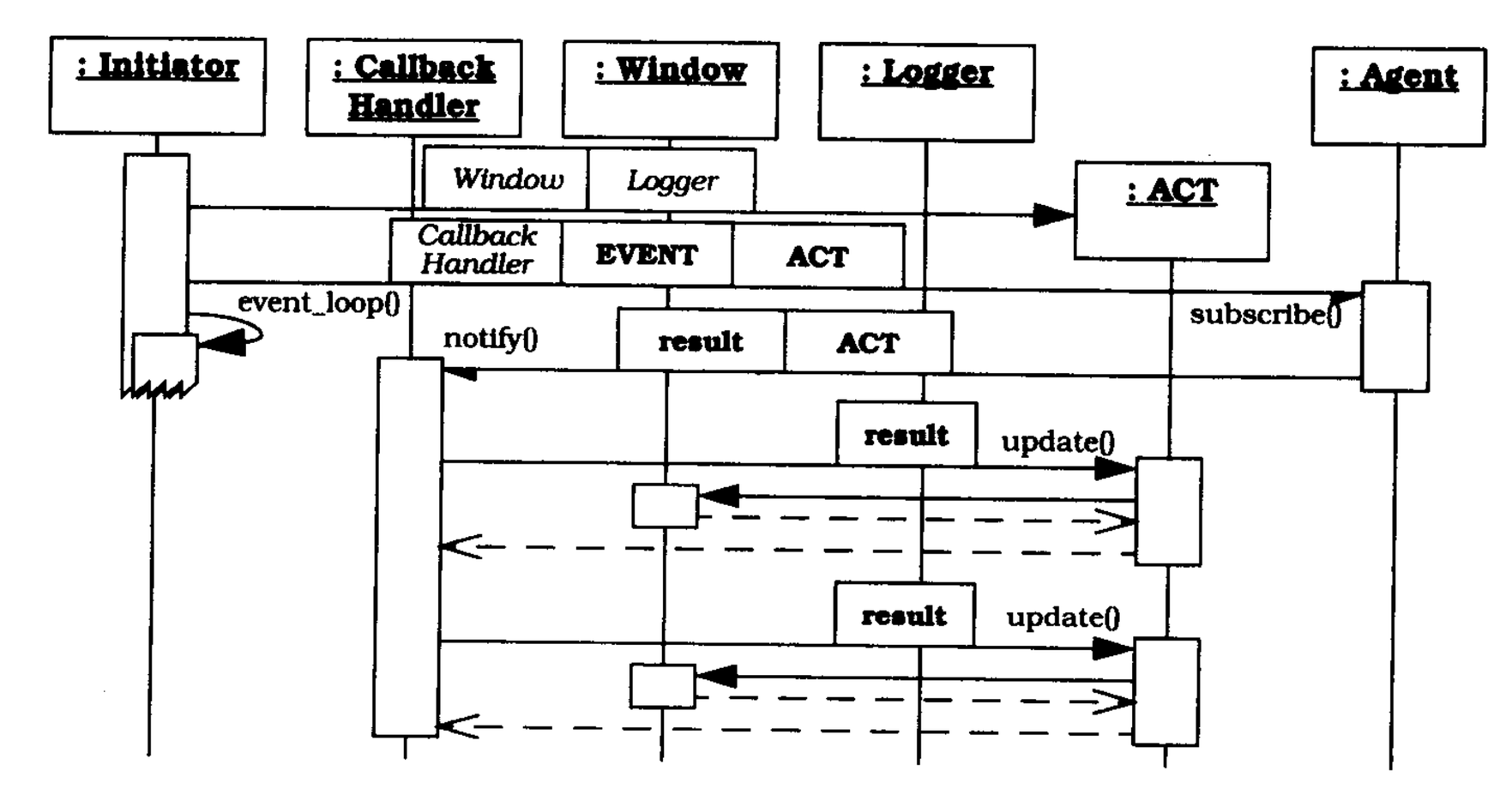


图 3-26

10. 变体

服务异步完成标记链。当中间服务也担当启动程序的角色，对其他服务调用异步操作来处理前一个启动程序的操作时，就产生一个服务链。

➡例如，一个管理应用程序调用agent的操作请求，该agent会因此调用定时器中的其他请求。在这种情况下，管理应用程序的启动程序使用了一个服务链，链中所有的中间服务（除了两个端点之外）既是启动程序又是服务，因为它们既接收异步操作，又启动异步操作。 ❑

一个服务链必须确定哪个服务最终响应启动程序。而且，如果链中的每个服务都使用异步完成标记模式，要考虑以下四个与异步完成标记的传递、存储和返回有关的问题：

277

- 如果链中没有服务产生新的异步完成标记，那么只需链中最后一个服务通知启动程序，这种设计方法可以优化异步完成标记的处理，因为这样不需要“打开”服务链（如图3-27）。
- 如果中间服务不需要对它启动的异步操作进行任何完成处理，它简单地将从前一个启动程序那里接收来的原先的异步完成标记转发出去（如图3-28）。

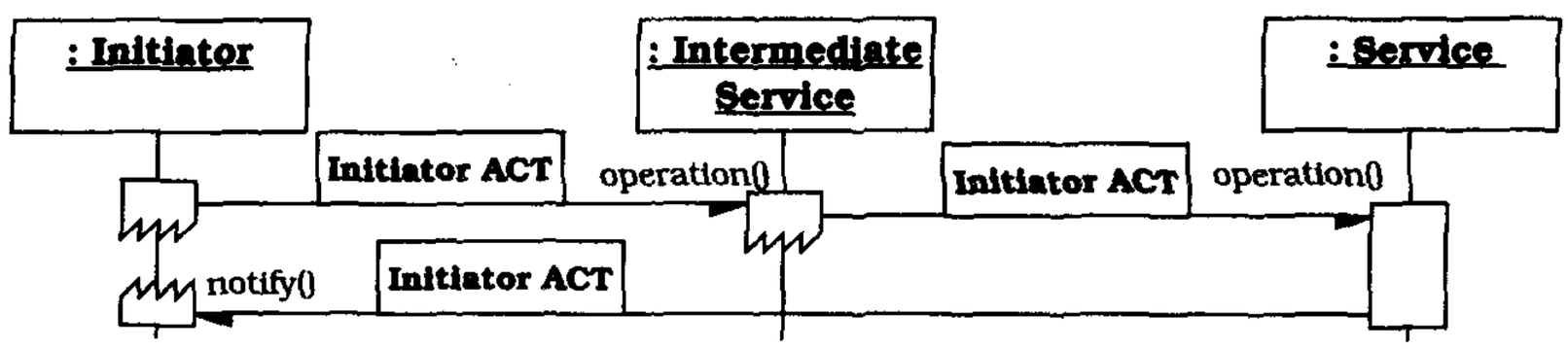


图 3-27

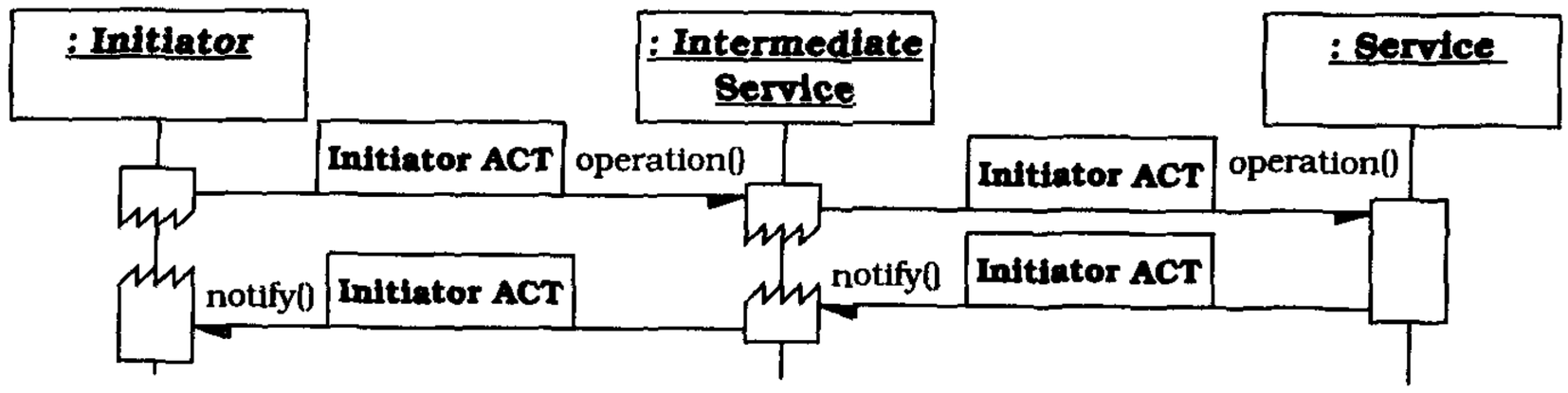


图 3-28

- 如果异步操作后必须有完成处理，而中间服务可以保证它的启动程序的异步完成标记值是惟一的，那么该服务可以使用启动程序的异步完成标记值对保存每个异步完成标记的数据结构进行索引查找，找出完成处理的动作和状态（如图3-29）。

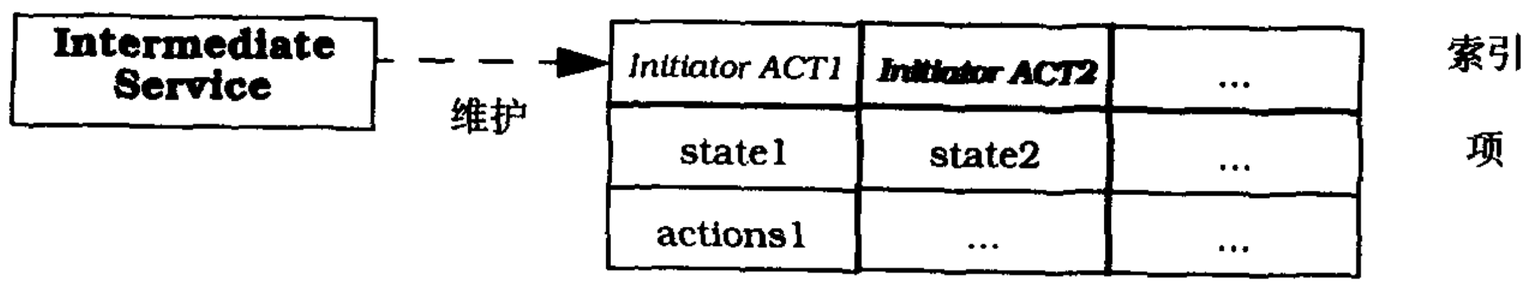


图 3-29

- 如果中间服务不能保证启动程序异步完成标记的惟一性，那么无法使用原来的异步完成标记来引用中间完成动作和状态。在这种情况下，中间服务必须建立一个新的异步完成标记，并将它们存储在一个表中，这样，当链“打开”时可以通过它们找到原来的异步完成标记（如图3-30）。

278

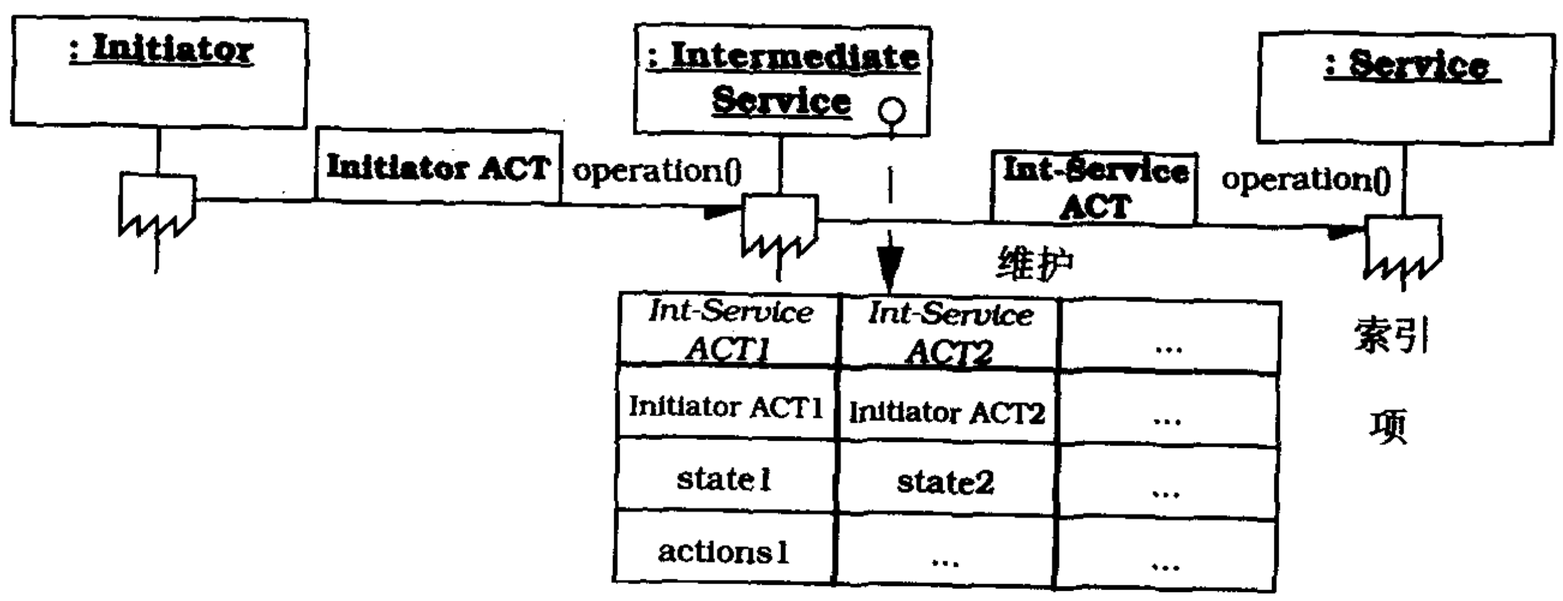


图 3-30

透明异步完成标记。在异步完成标记模式的一些实现中，服务并不将异步完成标记看成是

纯粹的不透明。例如，Win32的OVERLAPPED结构是一个透明的异步完成标记，因为操作系统内核可以修改某些字段。解决这个问题的办法之一就是如主动器模式的实现活动（1.1）中介绍的那样，传递一个OVERLAPPED结构的子类，该子类中包括有附加的异步完成标记状态。

同步异步完成标记。导致同步回调的操作也可以使用异步完成标记。在这种情况下，异步完成标记并不是真正的异步完成标记，而是同步的。用异步完成标记进行同步回调操作可以提供了一种很好的方法，将与操作有关的状态传递给服务。另外，这种方法与并发策略无关。这样，接收异步完成标记的代码既可用于同步操作，又可用于异步操作。

11. 已知使用

HTTP_Cookies。如果Web服务器要得到Web浏览器的响应，它可以使用异步完成标记模式。例如，用户使用Web浏览器先向Web服务器发送一个HTTP GET请求，下载一个表单，并在表单中填上数据。Web服务器希望用户传递表单中的数据。由于HTTP协议具有“无会话”特性，而且用户可以不立即填写该表单，所以作为启动程序的Web服务器向Web浏览器传递一个cookie（异步完成标记）和表单。服务器使用Cookie可以将用户的响应与先前的对表单的请求结合起来。Web浏览器不需要解释Cookie，而只需要将它不加修改地和完成的表单一起返回给Web服务器。

279

操作系统的异步I/O机制。支持异步I/O的操作系统使用了异步完成标记模式。例如Windows NT和POSIX使用了下面的技术：

- **Windows NT**。在Windows NT中异步完成标记与句柄、重叠I/O和Win32 I/O和完成端口一起使用[Sol98]。当创建Win32句柄[⊖]时，使用CreateIoCompletionPort()函数将它们和完成端口结合起来。完成端口为内核层的服务提供了一个将完成事件排队的场所。由起初调用异步操作的启动程序将这些事件从队列中删除并处理。

  例如，当启动程序通过ReadFile()和WriteFile()启动异步读、写操作时，它们可以指定一个OVERLAPPED结构的异步完成标记，当操作完成时，这个结构将从完成端口中删除。然后启动程序可以用GetQueuedCompletionStatus()函数删除完成事件，返回原先的OVERLAPPED结构作为异步完成标记。主动器模式中的实现活动（5.5）对这种设计方法有详细的介绍。
- **POSIX**。可以使用POSIX异步I/O API[POSIX95]将异步完成标记传递给异步I/O操作。实现方法是从aiocb结构中派生一个可传递给aio_read()或aio_write()的子类异步完成标记，然后用aio_suspend()异步事件多路分解函数获取这些异步完成标记，并向下类型转换为相应的完成处理程序。另外，启动程序可以规定用于异步I/O操作的完成事件，通过异步UNIX“实时”信号返回，或通过sigtimedwait()或sigwaitinfo()函数同步地返回。

CORBA多路分解。TAO CORBA对象请求代理[SC99]使用异步完成标记模式在客户机启动程序和服务器中有效地、可扩展地和可预见地多路分解不同类型的GIOP请求和响应。

280

例如，在多线程的客户机启动程序中，TAO使用异步完成标记将来自服务器的GIOP响应与一个客户机线程结合起来，该客户机线程在一个多路复用TCP/IP连接上启动对服务器进程的请

⊖ Win32句柄类似于UNIX的文件描述器。对于Win32重叠I/O，句柄用于标识网络传输端点或打开文件。

求。每个TAO客户机请求带有一个用32比特整数表示的惟一的非透明序列号，也就是异步完成标记。当调用一个操作时，客户机端的TAO ORB向该操作赋一个序列号，该序列号是对使用领导者/追随者模式管理的内部连接表的索引。

通过每个表项可以找到在多路复用的连接上等待其服务器响应的客户机线程。当服务器响应时，它返回由客户机发送过去的序列号异步完成标记。TAO的客户机端的ORB使用异步完成标记对连接表进行索引查找，确定唤醒哪些客户机线程并传递应答。

在服务器中，TAO使用异步完成标记模式提供低开销的、在一个对象适配器[SC99]的不同特性层上的多路分解机制。例如，当服务器创建了一个对象引用时，TAO 对象适配器在它的对象关键字中存储特定的对象ID和特定的POA ID值，最终该关键字作为包含在对象引用中的异步完成标记传给客户机。

当客户机传回了对象关键字和请求时，TAO的对象适配器从异步完成标记中提取特定的值，并利用它们直接在它所管理的表中进行索引查找。这种称为“主动多路分解”的图式[PRS+99]使用异步完成标记，以确保无论POA中的对象的个数或对象适配器嵌套的POA的个数是多少，查找都为常量时间O（*1*）。

电子医疗影像系统管理。这个模式中描述的管理实例来自于华盛顿大学为Project Spectrum [BBC94]开发的分布式电子医疗影像系统。在这个系统中，管理应用程序监控医疗影像系统中的各个分布式组件的性能和状态。这些组件包括影像服务器、模态、层次存储管理系统和放射专家诊断工作站。管理agent提供了一种异步服务，通知事件管理应用程序，如连接建立事件和影像传送事件。该系统使用异步完成标记模式，管理应用程序可以有效地将状态和早期由异步订阅操作接收的来自管理agent的事件结合起来。 281

Jini。Jini[Sun99a]分布式事件规范[Sun99b]中的归还（handback）对象是异步完成标记模式的基于Java的实例。当消费者注册一个事件源以接收通知时，它将一个归还对象传递给事件源。该对象是`java.rmi.MarshalledObject`，没有在事件源中整理它，而只是作为事件通知的一部分归还给消费者。

然后，消费者使用事件通知的`getRegistrationObject()`方法获取归还对象，该对象是在消费者向事件源注册时传递给事件源的。这样，消费者可以快速地恢复处理事件通知的语境。当第三方注册了该消费者以接收事件通知时，这种设计方法特别有用。

FedEx目录跟踪。异步完成标记模式的一个有趣的现实生活中的例子就是US联邦快递（Federal Express, FedEx）服务使用的目录跟踪机制。一张FedEx空运提单上包含这样一行标语：“你的内部单据证明信息（附言：前24个字符将会出现在发货单中）”

包裹的发货人将该标语作为异步完成标记。FedEx（服务）将该异步完成标记和发货单一起交给你（启动程序）。发货单用来通知发货人该事务已完成。FedEx故意采用这种不太严格的定义：最多24个字符，否则就是不合规范的。因此，发货人可以以不同的方式使用该域。例如，发货者可以在该域中填充内部数据库的记录的索引或包含在得知FedEx包已被接收后要进行的处理列表的文件名。

12. 结论

使用异步完成标记模式的**优点**：

简化启动程序的数据结构。启动程序不需要复杂的数据结构来将服务响应与完成处理程序结合起来。服务返回的异步完成标记可以被向下类型转换或重新解释，使之带有启动程序将其多路分解给相应的完成活动所需要的信息。

282

高效的状态查询。异步完成标记的时间性能很好，因为不需要对与服务响应一起返回的数据进行复杂的分析。将响应和先前的请求结合起来所需的相关信息可以存储在异步完成标记中或者在由异步完成标记所引用的对象中。另一方面，异步完成标记可作为对访问状态的指针或索引，用于高效访问，从而消除了昂贵的表搜索。

空间效率高。异步完成标记可以在消费最小的数据空间的情况下向应用程序提供足够的信息，将大量的状态结合起来，以处理异步操作完成动作。例如，在C和C++中，void指针异步完成标记可以引用启动程序应用程序中的任意大的对象。

灵活性。要利用服务的异步完成标记，用户自定义的异步完成标记不必非要继承一个接口。这样应用程序可以将不可预见其类型转换或者不可能进行类型转换的对象作为异步完成标记传递。异步完成标记的通用特性可以用于将任何类型的对象与一个异步操作联系起来。例如，当用CORBA对象引用实现异步完成标记时，异步完成标记可被裁剪为合适的具体接口。

非独裁的并发策略。因为可以从异步完成标记中有效地恢复耗时长的操作状态，所以可以异步地执行这样的操作。因此启动程序可以是单线程的，也可以是多线程的，这依赖于应用需求。相反，并不提供异步完成标记的服务可能要强迫对延迟很敏感的启动程序在线程中同步地执行操作以正确地处理操作完成。

使用异步完成标记模式要避免以下几种**不足**：

内存泄漏。如果启动程序使用指向动态分配的内存区域的指针作为异步完成标记，当服务不能返回异步完成标记时（如服务崩溃时），会产生内存泄漏。正如实现活动（2）中所介绍的那样，考虑这种可能性的启动程序应维持一个单独的异步完成标记存储池或表。这样可以在服务失败或服务破坏了异步完成标记时进行显式的垃圾收集。

283

认证。当异步事件完成时，异步完成标记返回给启动程序，启动程序在使用之前需要进行认证。如果服务器不能将异步完成标记看成是非透明的而有可能改变异步完成标记的值，这种认证是必需的。实现活动（2）介绍了一种解决这一问题的办法。

应用程序重映射。如果异步完成标记是直接指向内存的指针，将应用程序的某一部分直接映射到虚拟存储器中时就会发生错误。在崩溃后重新启动的持久应用程序中，以及对于从内存映射地址空间中分配的对象可能会出现这种情况。为了防止这些错误，和实现活动（1）中介绍的方法一样，可以将存储池的索引作为异步完成标记。由“索引异步完成标记”提供的这种间接方式可以防止重映射，因为在重映射过程中，索引能保持有效，而指向内存的指针则不行。

参见

异步完成标记和备忘录模式的组成部分是类似的。在备忘录模式[GoF95]中，原发器将备忘录传给看守者（caretaker），后者将备忘录看成是“不透明”的对象。在异步完成标记模式中，启动程序将异步完成标记传给服务，后者也将异步完成标记看成是“非透明的”对象。

不过这些模式在目标和应用范围上有差别。备忘录模式对对象状态进行“快照”，而异步完成标记模式将状态和异步操作的完成结合起来。另一个区别是动态特性。在异步完成标记模式中，启

动程序（相对于备忘录模式中的原发器）主动地创建一个异步完成标记，将它传给服务。在备忘录模式中，（相对应于异步完成标记模式中的启动程序）请求从反应性的原发器产生一个备忘录。

致谢

Irfan Pyarali和Timothy Harrison是异步完成标记模式的早期版本的合作者。感谢Paul Mckenney和Richard Toren，他们提出了一些深遂的建议，还有Michael Ogg提供了Jini的已知使用。 284

3.4 接受器-连接器

接受器-连接器(Acceptor-Connector)设计模式将网络化系统中合作的对等体服务的连接和初始化工作，与该服务在连接和初始化之后所执行的处理分开。

1. 例子

考虑一个监视和控制卫星群的大规模分布式系统管理应用系统的组成[Sch96]。这样的管理应用程序一般由多个服务、在连接远程对等体主机的传输端口之间传输数据的应用级网关组成。如图3-31所示。

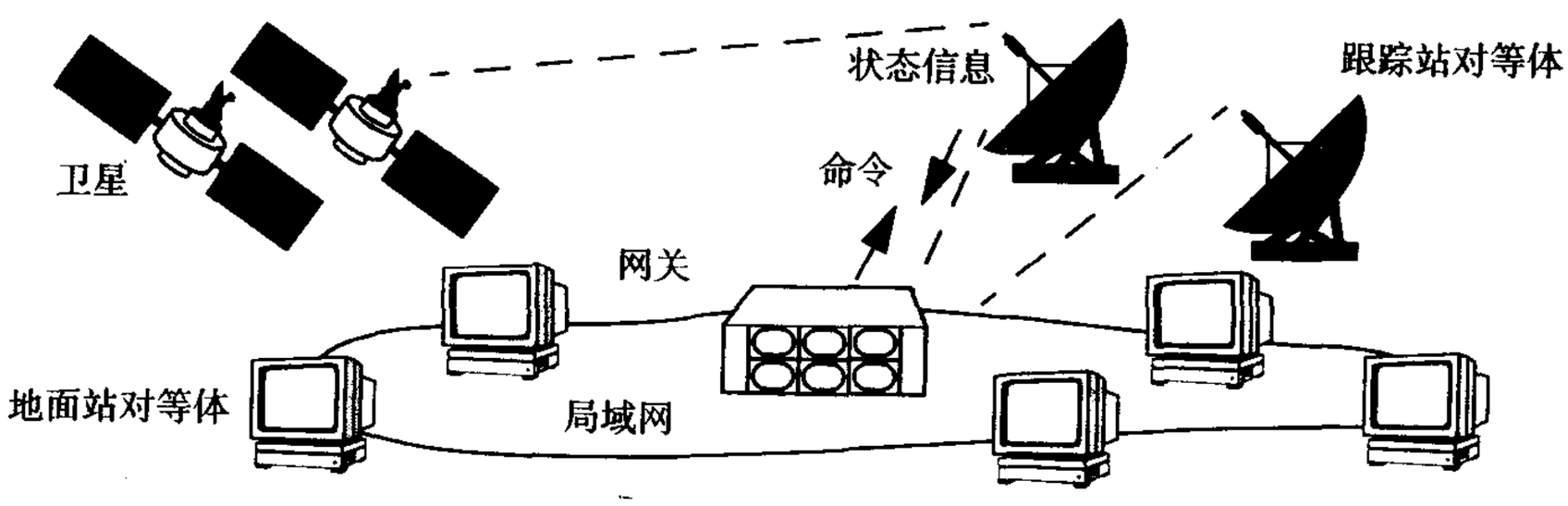


图 3-31

对等体主机上的每个服务使用网关来发送和接收不同类型的数据，包括状态信息、大量的数据和控制卫星的命令。对等体主机分布在局域和广域网中。

网关使用面向连接的TCP/IP协议[Ste93]在对等体主机之间传送数据，系统中的每个服务都有一个特定的传输地址，它是一个由IP主机地址和TCP端口号组成的元组。不同的端口号惟一地标识不同类型的服务。

和将服务绑定到一个特定的TCP/IP主机/端口元组的情况不同，先验地指定连接建立和服务初始化角色可能是不明智的，而这个元组可以在分布式系统的生命期的早期被选定。相反，网关和对等体主机中的服务应该能灵活地改变它们的连接角色，以支持如下的运行时特性： 285

- 网关的服务可以主动地向位于远程对等体主机中的服务启动连接请求，然后向它们发送数据。
- 网关中的服务可以被动地接收对等体主机中服务的连接请求，然后通过网关将该数据传送到另一个对等体主机的服务上。

- 驻留在对等体主机中的服务在一种情况下可以是主动的连接启动程序，在另一种情况下可以是被动的连接接受者。
- 可以在同一个网关或对等体主机中组合主动和被动连接行为而进行混合配置。

总的来说，支持这样一种运行时的特性所需要的灵活性要求负责连接建立、初始化和处理对等体服务的通信软件具有独立性。

2. 语境

一个网络系统或应用程序，它们使用面向连接的协议在通过传输端口连接的对等体服务之间进行通信。

3. 问题

在面向连接的网络化系统中，应用程序通常包含相当大量的用于建立连接和初始化服务的配置代码。这些配置代码大多独立于服务对在连接传输端点之间交换的数据所进行的处理。因此配置代码与服务处理代码之间的紧密耦合是不理想的，因为这样不能解决以下四个强制条件：

- 它应该很容易地改变连接角色，以支持不同的应用特征。和在“例子”一节讨论的一样，连接角色决定了应用程序是否主动地初始化或者被动地接受连接。相反，通信角色决定了应用程序是否担当客户机、服务器角色，或在对等体配置中，既是客户机又是服务器的角色。
- 应很容易地增加服务、服务实现和通信协议的种类，而不影响已有的连接建立和服务初始化配置代码。

286

➡例子中的网关要求与运行TP4或SPX传输协议，而不是TCP协议的目录服务集成，最好这种集成对实现网关服务本身没有或只有很小的影响。 ❑

- 总的来说，连接建立和服务初始化策略的改变要比应用程序实现的通信协议和服务的改变频率少。

➡FTP、TELNET、HTTP和CORBA IIOP服务都使用不同的应用层的通信协议。不过，可以用同样的连接和初始化机制来配置它们。 ❑

- 对大规模网络化系统，应能通过使用高级的操作系统特性，如异步连接机制来降低连接建立延迟。

➡具有大量对等体层的应用程序可能需要异步和并发地建立很多连接。对于在高延迟的广域网中通信的应用程序（如网关例子）来说，高效和可扩展的连接建立尤其重要。 ❑

4. 解决方案

将网络化应用中对等体服务的连接和初始化工作，与连接和初始化后这些服务所执行的处理分开。

具体地说：在对等体服务处理程序中封装应用服务。每个服务处理程序实现网络化应用中端到端的一半服务。使用两个工厂连接和初始化对等体服务：接受器和连接器。这两个工厂相互合作，创建在两个对等体服务处理程序与它们的两个连接的传输端点之间的全关联[Ste93]，每个传输端点封装在一个传输句柄中。

接受器工厂代表相关的对等体服务处理程序，在由远程对等体服务处理程序发出的连接请

求事件[⊖]到达后被动地建立连接。反过来，连接器工厂代表对等体服务处理程序主动地与指定的远程对等体服务处理程序建立连接。 287

连接建立后，接受器和连接器工厂初始化相应的对等体服务处理程序，向它们传送各自的传输句柄。然后对等体服务处理程序使用传输句柄通过连接的传输端点交换数据，进行与应用有关的处理。一般来说，在连接和初始化后，服务处理程序并不与接受器和发送器交互。

5. 结构

在接受器-连接器模式中有六个关键的参与者：

被动模式的传输端点是一个工厂，它侦听连接请求、接受连接请求和建立传输句柄，传输句柄中封装了新连接的传输端点。可以通过连接的传输端点读写相应的传输句柄来交换数据（如图3-32）。

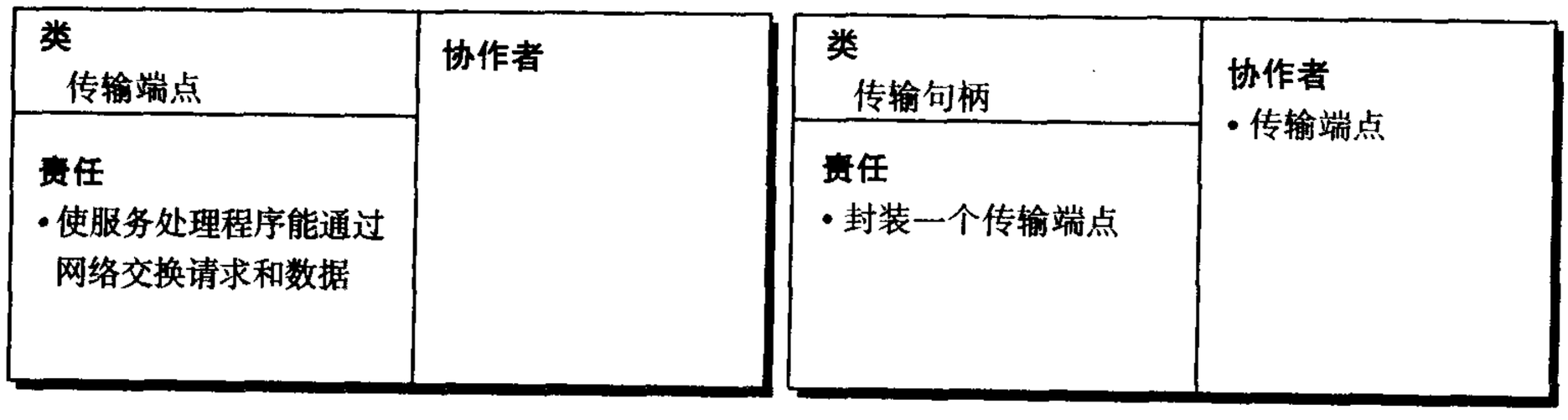


图 3-32

➡在网关例子中，使用套接字句柄来封装传输端点。在这种情况下，被动模式的传输端点是一个绑定到一对TCP端口号和IP地址的被动模式的套接字句柄[Ste98]。它创建一个由数据模式套接字句柄封装的连接的传输端点。标准的Socket API操作，如recv()和send()可以使用这些连接的数据模式套接字句柄读/写数据。 ❑

服务处理程序定义了网络系统中端到端的一半服务。在这个端到端的服务中一个具体的服务处理程序通常既具有客户机角色又具有服务器角色。在对等使用的情况下，它可以同时扮演这两个角色。一个服务处理程序提供了一个激活钩子方法，用于在连接到其对等体的服务处理程序后进行初始化。另外，服务处理程序还包括一个封装传输端点的传输句柄，如数据模式套接字句柄。一旦连接上，服务处理程序就使用传输句柄与对等体服务处理程序通过它们的连接传输端点交换数据。如图3-33所示。 288

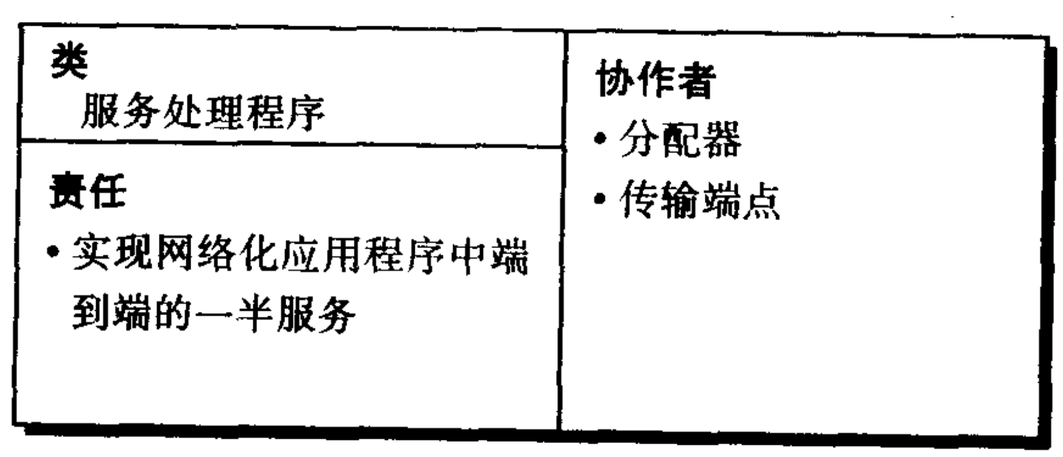


图 3-33

⊖ 以后，将连接请求事件和数据请求事件简单地表示为连接请求和数据请求。

➡在例子中，服务处理程序既是网关内的协作组件，又是通过连接的套接字句柄在TCP/IP上通信的对等体主机。服务处理程序负责处理状态信息、大量的数据和监控卫星群的命令。 ❑

接受器是一个工厂，它实现的功能包括被动地建立被连接的传输端点、创建和初始化相关的传输句柄和服务处理程序。一个接受器提供两个方法：连接初始化和连接完成，这两个方法在被动模式传输端点的帮助下实现上述功能。

调用初始化方法时，接受器将被动模式传输端点绑定到一个传输地址上，如一个TCP端口号和IP主机地址，它被动地侦听连接请求。

当连接请求到达时，接受器的连接完成方法执行下面三个步骤：

- 首先，它用被动模式传输端点创建一个连接的传输端点，并在传输句柄中封装该端点。
- 第二，它创建一个服务处理程序，用于处理对等体服务处理程序通过连接的传输端点发出的数据请求。
- 第三，它将传输句柄存储在相应的服务处理程序中，然后调用服务处理程序的激活钩子方法，该方法让服务处理程序自己完成初始化。

289

连接器[⊖]也是一个工厂，它实现的功能包括主动地建立连接的传输端点、初始化其相应的传输句柄和服务处理程序。它提供两个方法：连接启动和连接完成方法，这两个方法实现上述功能。

向连接启动方法传给一个已有的服务处理程序，该方法为它和接受器建立一个连接的传输端点。正如前面介绍的，该接受器一定正在侦听在某一传输地址上的连接请求（如图3-34）。

将连接器的连接启动方法和完成方法分开，这样可以使连接器支持透明的同步和异步的连接建立。

- 对于同步的情况，激活连接请求的连接器阻塞其调用者，直到传输端点被连接上。以后，连接器直接调用服务处理程序的激活钩子方法。
- 对于异步的情况，异步运行连接请求，连接器的激活方法立即返回。只有在连接器得到传输端点已异步地完成了连接的通知后，连接完成方法才激活服务处理程序。

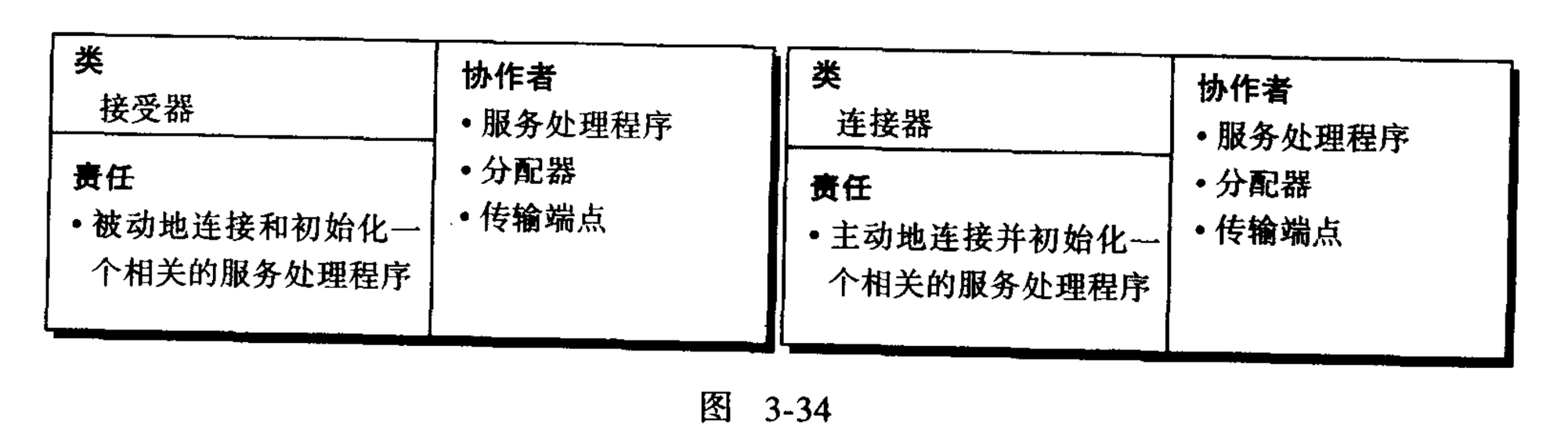


图 3-34

290

无论是同步还是异步地连接传输端点的，在连接上传输端点后，接受器和连接器都通过调用其激活钩子方法来初始化一个服务处理程序。以后，服务处理程序一般不需要与连接器和接受器工厂交互。

⊖ 注意，接受器-连接器模式中的连接器的概念与[SG96]中的连接器的概念不同。在[SG96]中，连接器是一个软件体系结构规约概念，表示系统中组件之间的协作。

分配器负责多路分解表示不同类型服务请求（如连接请求和数据请求）的指示事件（如图3-35）。

类 分配器	协作者 • 接受器 • 连接器 • 服务处理程序 • 传输端点
责任 • 注册接受器、连接器和服务处理程序 • 分配接受器、连接器和服务处理程序	

图 3-35

- 对于接受器，分配器多路分解从封装传输端点的一个或多个传输句柄中接收来的连接指示事件。多个接受器可以注册同一个分配器，分配器替它们侦听来自于对等体连接器的连接请求。
- 对于连接器，分配器多路分解因响应异步激活的连接而到达的完成事件。为了处理这种情况，连接器向一个分配器注册它自己，以接收这些连接完成事件，然后分配器运行它的事件循环。当完成事件到达时，它通知相应的连接器。然后连接器调用指定的服务处理程序的激活钩子方法，让服务处理程序来初始化自己。因此，一个分配器和连接器可以代表多个服务处理程序异步地激活和完成连接。

  注意，异步连接建立不需要使用分配器，因为启动连接的线程会因等待连接完成事件而阻塞。因此，该线程可以直接激活服务处理程序。

291

- 服务处理程序可以向分配器注册它们的传输句柄，在句柄上出现指示事件时该分配器将会通知服务处理程序。

可以从上述接受器-连接器模式的通用组件中派生子类并实例化来创建下面的具体组件，从而建立网络化的应用程序和服务。

具体的服务处理程序定义与应用有关的部分端到端服务，由具体接受器或具体连接器激活这部分服务。具体接受器用具体服务处理程序、传输端点和这些服务处理程序使用的传输句柄来实例化通用接受器。类似地，具体连接器也实例化通用连接器。

具体服务处理程序、接受器和连接器也用特定类型的进程间通信（Interprocess Communication, IPC）机制（如Socket[Ste98]或TLI[Rago93]）进行实例化，这些IPC机制用来建立连接到对等体服务处理程序的传输端点和传输句柄，以交换数据。

图3-36是接受器-连接器模式的示意图。

292

6. 动态特性

为了说明接受器-连接器模式中多组件之间的合作关系，我们考虑以下三种典型的场景：

- 场景I说明被动连接建立。
- 场景II说明同步主动连接建立。
- 场景III说明异步主动连接建立。

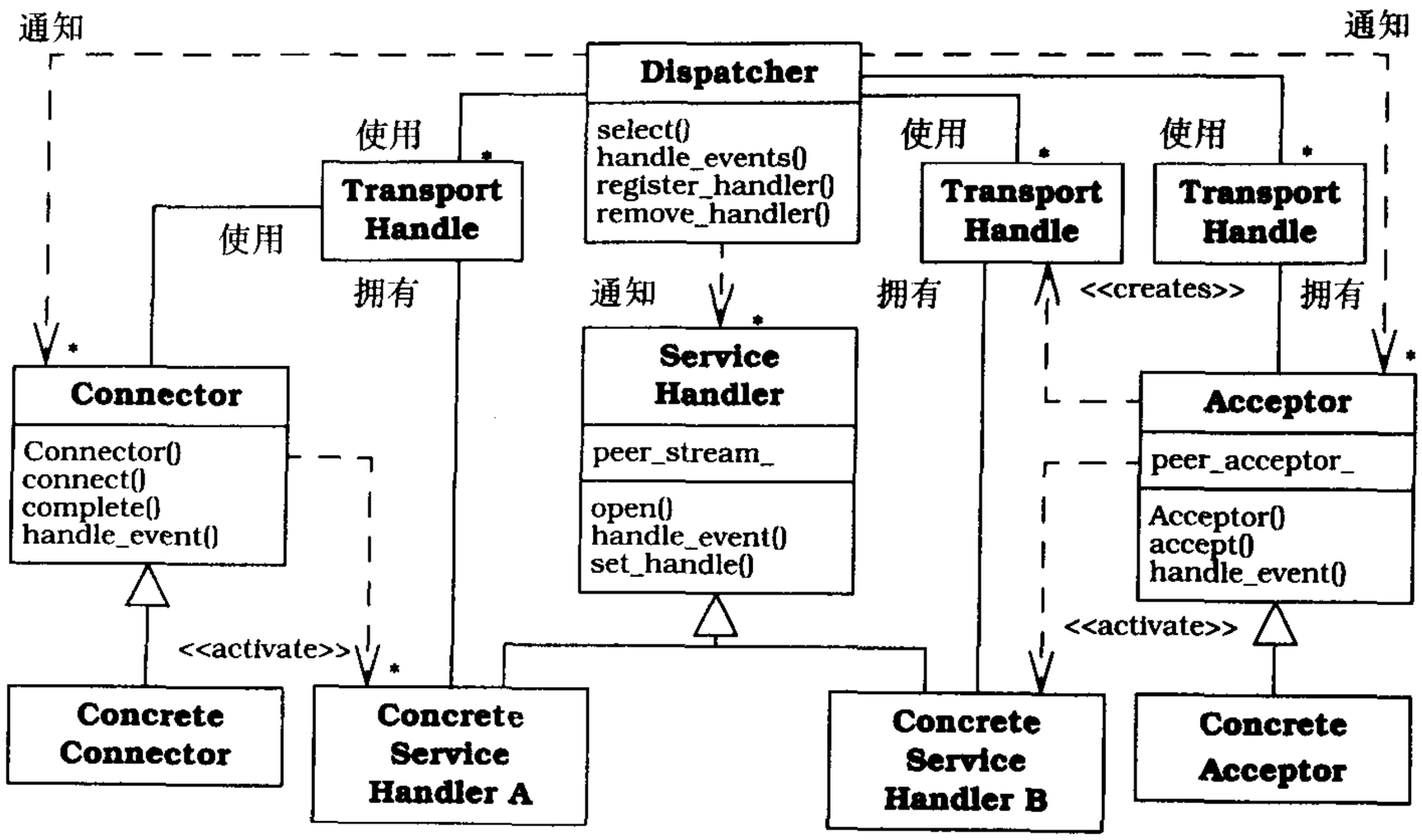


图 3-36

场景I：本场景用于解释接受器和服务处理程序组件之间的合作关系，该场景分成以下三个阶段（如图3-37）：

- 被动模式传输端点初始化阶段。担当被动连接角色的应用程序首先调用接受器的连接初始化方法。该方法初始化一个被动模式传输端点，并将其绑定到一个传输地址，如局部主机的IP地址和TCP端口号。然后接受器在该传输地址上侦听由对等体连接器启动的连接请求[⊖]。

  下一步，接受器的初始化方法注册一个分配器，当对等体连接器的连接指示事件到达时该分配器通知接受器。接受器初始化方法返回后，应用程序激活分配器的事件循环。该循环等待对等体连接器的连接请求和其他类型的指示事件。
- 服务处理程序初始化阶段。对某一传输地址的连接请求到达时，分配器通知相应的接受器，该接受器使用它的被动模式传输端点建立一个新连接的传输端点，将它封装在传输句柄中。然后它建立一个新的服务处理程序，将传输句柄存储在服务处理程序中，并调用服务处理程序的激活钩子方法。

  该钩子方法进行与服务处理程序有关的初始化，如分配锁、创建新线程或与登录服务建立会话。服务处理程序可以向一个分配器注册自己，在有包含数据请求的指示事件时该分配器自动地通知处理程序。

293

- 服务处理阶段。在连接到传输端点，且初始化了相应的服务处理程序后，进入服务处理阶段。这个阶段在对等体服务处理程序之间使用应用级通信协议（如TELNET、FTP、HTTP或CORBA IIOP）通过连接的传输端点交换数据。数据交换完成后，关闭传输端点、传输句柄、服务处理程序，释放它们的资源。

⊖ 为了简单起见，对这个场景，在顺序图中省略了被动模式传输端点的创建和使用。

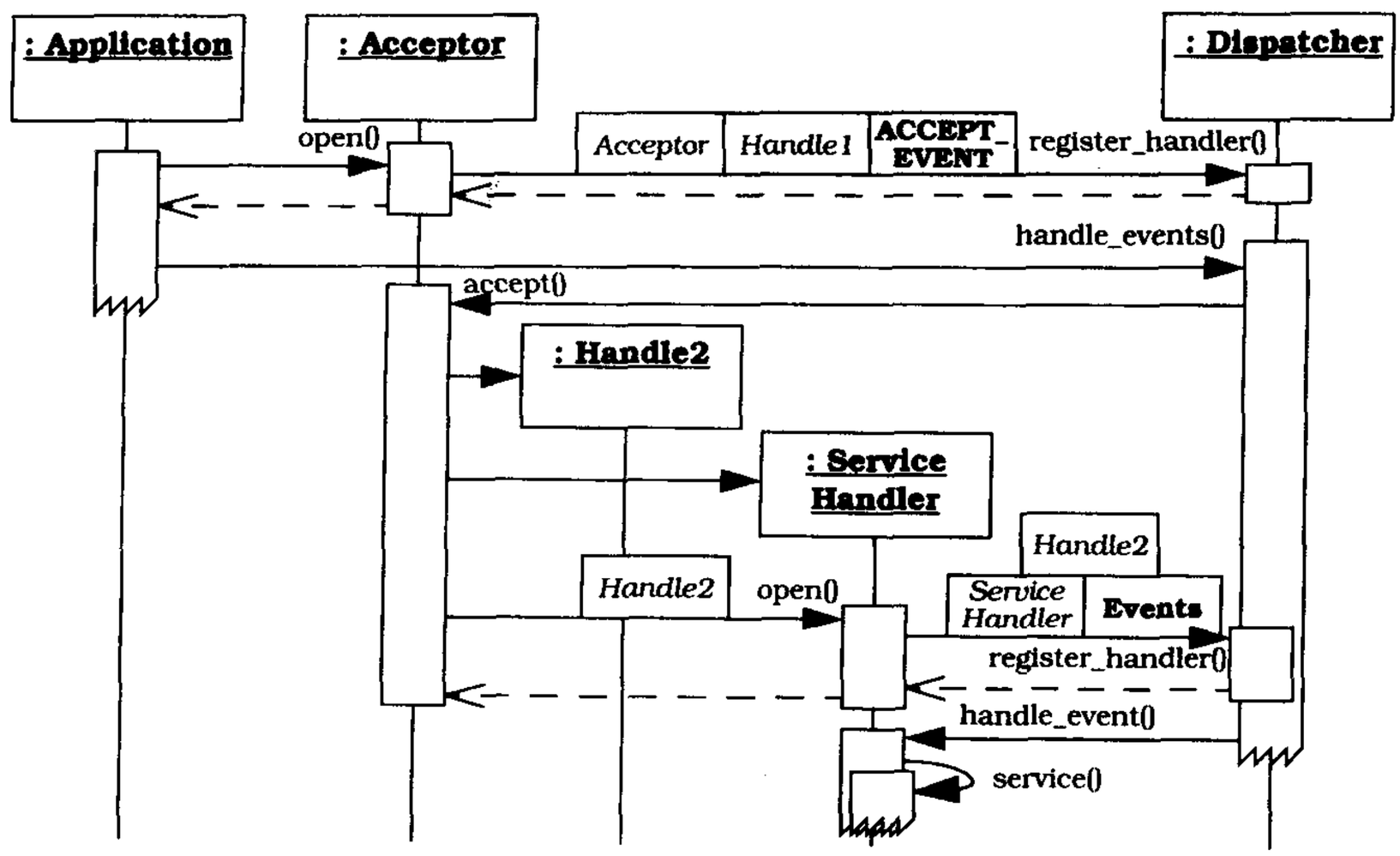


图 3-37

连接器可以使用同步和异步两种策略初始化其服务处理程序。在下列情况下可以使用同步服务初始化：

- 当连接建立有很短的延迟，如通过一个回环网络设备与同一个主机上的服务建立连接时。
- 当可以使用多控制线程且可以高效地使用“每个连接一个线程”模型[Sch97]同步地连接每个服务时。
- 如果以固定的顺序初始化这些服务，且只有在所有的连接建立后客户机端进行有用的处理。 294

在其他情况下可使用异步服务初始化，如在高延迟的链路上建立连接，使用单线程应用程序或者初始化大量的可以以任意顺序连接的对等体。

场景II：在同步连接场景中各组成部分之间的协作可分为三个阶段（如图3-38）：

- *连接启动阶段*。要在两个对等体服务处理程序之间同步地建立连接，应用程序可以调用连接器的连接启动方法。该方法使用与服务处理程序对应的传输句柄，通过阻塞应用程序线程，直到连接同步完成而主动地建立连接。
- *服务处理程序初始化阶段*。同步地完成连接后，连接启动方法直接调用相应的服务处理程序的激活钩子方法。该钩子方法执行与服务处理程序有关的初始化工作。
- *服务处理阶段*。初始化服务处理程序后，该服务处理程序使用与连接的对等体服务处理程序交换的数据进行与应用有关的服务处理。这个阶段类似于由接受器创建并初始化的服务处理程序在服务处理阶段所做的工作。 295

在同步场景中，连接器在一个阻塞操作中组合了连接启动和服务初始化两个阶段。线程只建立一个连接就可以多次调用连接器的连接启动方法。

场景III：异步连接器场景中多个组成部分之间的协作分为三个阶段（如图3-39）：

- *连接启动阶段*。要异步地连接两个对等体的服务处理程序，应用程序调用连接器的连接启动方法。和场景II一样，连接器主动地启动连接。

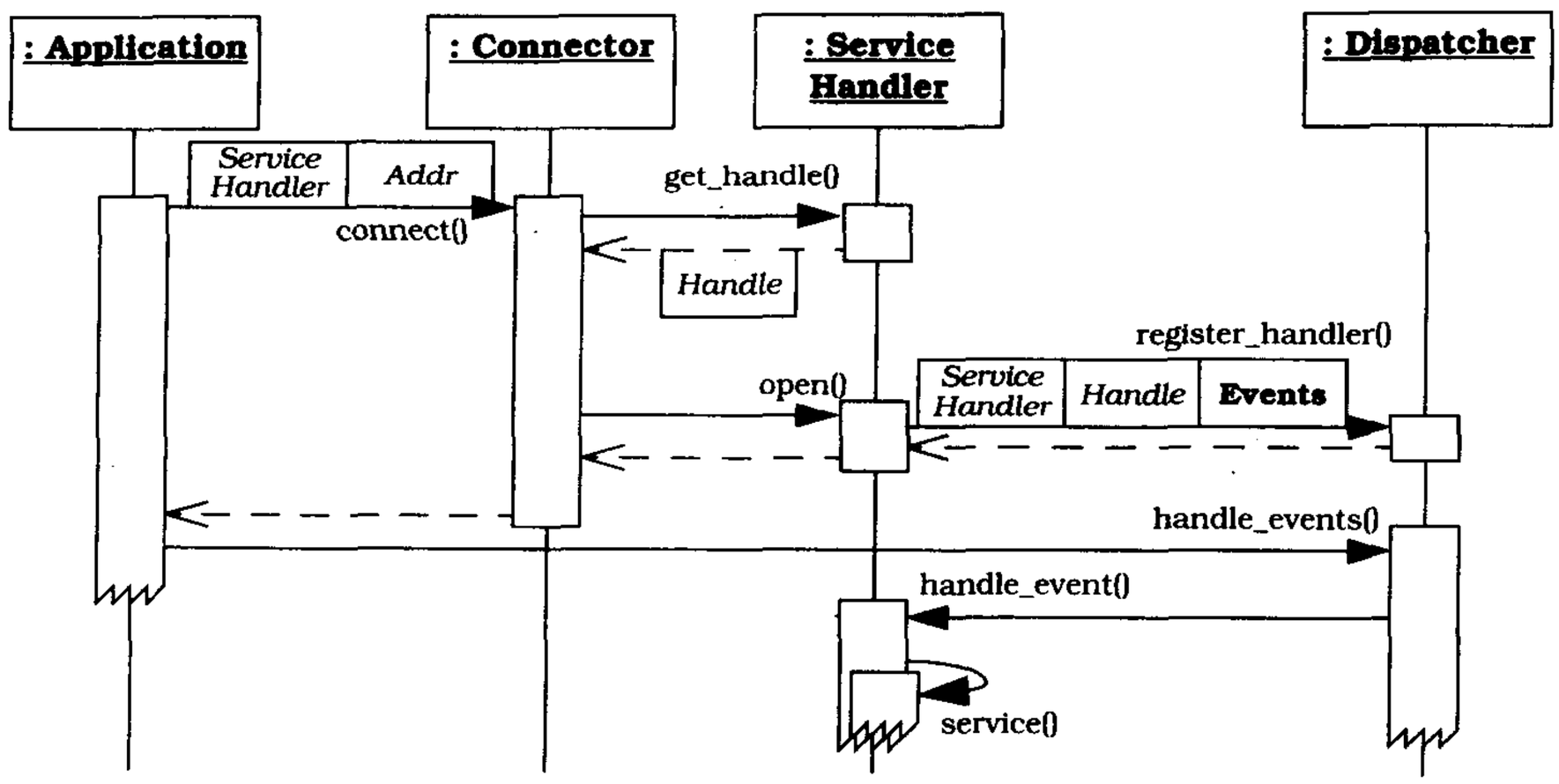


图 3-38

和场景II不一样的是，连接启动请求是异步执行的。因此应用程序线程在等待连接完成时并不阻塞。为了在连接完成时得到通知，连接器要向分配器注册自己和服务处理程序的传输句柄，并将控制返回给应用程序。

- *服务处理程序初始化阶段*。异步地完成连接后，分配器通知连接器的连接完成方法。该方法清除所有被分配的用于管理该连接的任何资源，然后调用服务处理程序的激活钩子方法进行与服务有关的初始化处理。
- *服务处理阶段*。激活服务处理程序后，该服务处理程序使用与连接的对等体服务处理程序交换的数据进行与应用有关的服务处理。该阶段类似于场景I和场景II中所介绍的服务处理阶段。

请留意图3-39中场景III的连接启动阶段是如何与服务处理程序初始化阶段临时分离开的，这种分离使得可以并发地进行多个连接启动和完成处理，从而能最大限度地利用网络和主机中的并行性。

296

7. 实现

接受器-连接器模式中的各组成部分可以分解成以下三层：

- *多路分解/分配基础设施层组件*。该层执行通用的与应用无关的事件分配策略。
- *连接管理层组件*。该层执行通用的与应用无关的连接和初始化服务。
- *应用层组件*。该层对上两层的通用策略进行定制，它使用了子类派生、对象组合和/或参数化类型的实例化等方法，以创建具体组件，这些具体组件用于建立连接、交换数据和执行与服务有关的处理。

297

下面依次介绍接受器-连接器实现中的多路分解/分配组件层、连接管理和应用组件层。

(1) *实现多路分解/分配基础设施组件层*。本层处理发生在传输端点上的事件，由传输机制和分配机制组成，实现活动可分成两个子活动：

(1.1) *选择传输机制*，这些机制包括：

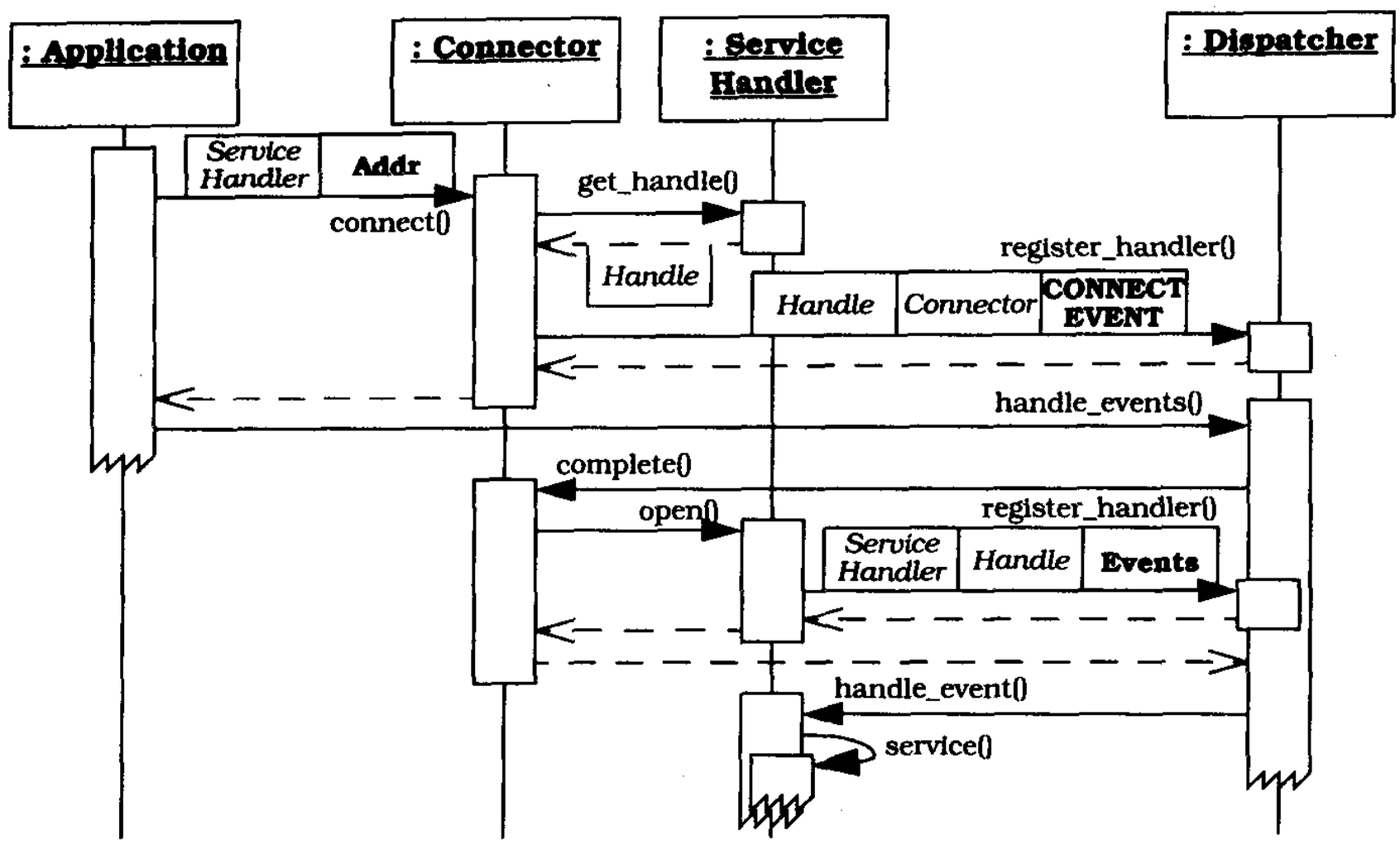


图 3-39

- 被动模式传输端点组件。
- 连接的传输端点组件。
- 传输地址组件。
- 传输句柄组件。

被动模式传输端点是一个工厂，该工厂在一个已知的传输地址上侦听连接请求的到达。当一个请求到达时，被动模式传输端点创建一个连接的传输端点，并在一个传输句柄中封装这个新端点。应用程序可以使用该传输句柄和与对等体连接的传输端点交换数据。通常由底层的操作系统平台提供传输机制组件。和实现活动（3）中介绍的一样，可以通过包装器外观访问这些传输机制组件。

➡在网关例子中，使用Socket API[Ste98]实现传输机制组件，分别用被动模式和数据模式套接字实现被动模式传输端点和连接的传输端点；用套接字句柄表示传输句柄；用IP主机地址和TCP端口号表示传输地址。 ❑

(1.2) 实现分配机制。这些机制由分配器和事件处理程序组件组成。分配器负责将请求与对应的接受器、连接器和服务处理程序联系起来。事件处理程序定义了由事件驱动的应用程序中服务所提供的事件处理接口。

实现分配机制要遵照反应器或主动器事件多路分解模式中介绍的指南进行。这些模式分别处理同步和异步的事件多路分解，也可以使用主动对象模式或领导者/追随者线程池将分配器实现为一个单独的线程或进程。 298

➡网关例子用反应器模式中的组件实现分配器和事件处理程序。这样可以在一个控制线程中高效地、同步地多路分解来自多个事件源的各种类型的事件。按照反应器模式中的术语称为“反应器”的分配器使用反应性模型多路分解和分配具体的事件处理程序。 ❑

因为在整个应用程序进程中只需要一个反应器单件[GoF95]实例，所以在这里我们使用反应

器单件。在网关例子中称为Event_Handler的事件处理程序类中有一些方法，当有注册的事件发生时，反应器通知服务处理程序、连接器和接受器时需要使用这些方法。因此，如实现活动（2）所介绍的一样，为了与反应器协作，这些组件必须是从类Event_Handler中派生出的子类。 ❑

(2) *实现连接管理组件层*。本层负责创建服务处理程序，主动或被动地将服务处理程序连接到远程服务处理程序上，以及在连接后激活服务处理程序。这一层的所有组件都是通用的，并依赖于具体的IPC机制、具体的服务处理程序、具体的接受器和具体的连接器。由实现活动（3）中介绍的应用层对它们进行实例化。在连接管理层中有三个主要的组件：*服务处理程序*，*接受器和连接器*。

(2.1) *定义通用的服务处理程序接口*。服务处理程序提供了一个通用的处理接口，它是由客户机、服务器或者在端到端的服务中两者（客户机和服务器）共同定义的服务。该接口包括初始化服务处理程序，执行它定义的服务以及维护用于通信的IPC机制等方法。要创建一个具体的服务处理程序，应用程序必须使用子类派生、对象组合或参数化类型的实例化等方法定制该组件。

应用程序还必须用封装了一个传输句柄和相应的传输端点的具体的IPC机制配置一个服务处理程序组件。通常将这些IPC机制实现为包装器外观。具体的服务处理程序可以利用这些IPC机制与远程对等体服务处理程序通信。

299

➡在网关例子中，定义了一个从Event_Handler类继承来的抽象基类Service_Handler。Event_Handler是在第3.1节反应器模式的实现活动（1）中定义的：

```
template <class IPC_STREAM>
    // <IPC_STREAM> is the type of concrete IPC data
    // transfer mechanism.
class Service_Handler : public Event_Handler {
public:
    typedef typename IPC_STREAM::PEER_ADDR Addr;

    // Pure virtual method (defined by a subclass).
    virtual void open () = 0;

    // Access method used by <Acceptor> and <Connector>.
    IPC_STREAM &peer () { return ipc_stream_; }

    // Return the address we are connected to.
    Addr &remote_addr () {
        return ipc_stream_.remote_addr ();
    }

    // Set the <handle> used by this <Service_Handler>.
    void set_handle (HANDLE handle){
        return ipc_stream_.set_handle (handle);
    }
private:
    // Template 'placeholder' for a concrete IPC
    // mechanism wrapper facade, which encapsulates a
    // data-mode transport endpoint and transport handle.
    IPC_STREAM ipc_stream_;
};
```

这种设计方式使一个Reactor可以分配它从类Event_Handler中继承来的服务处理程序

的事件处理方法。另外，Service_Handler类定义了访问其IPC机制的方法，在类中使用参数化类型配置IPC机制。最后，该类中包括一个激活钩子，一旦连接建立，接受器和连接器可以用它来初始化Service_Handler对象。这是一个称为open()的纯虚方法，具体的服务处理程序必须对它进行重载以进行与服务有关的初始化。 ❑

(2.2) 定义一个通用的接受器接口。接受器组件中实现一个通用的策略，该策略用于被动地建立连接、建立和初始化具体的服务处理程序，这些具体的服务处理程序利用这些连接与对等体服务处理程序交换数据。接受器中也定义一个初始化方法，供应用程序调用这个方法，用于向网络中其他应用程序通告它的被动模式传输端点。 300

和实现活动（3）中介绍的一样，应用层中的组件对通用的接受器进行定制，使该接受器可以代表某一个服务处理程序，使用指定的IPC机制被动地建立连接。接受器的实现使用两种策略支持这种定制：多态性和参数化类型。

- 多态性。“结构”一节中介绍了这种策略，它通过从一个通用的接受器中派生子类来定义具体的接受器。称为accept()的接受器方法是一个模板方法[GoF95]，它负责接收来自于远程客户机的连接请求。它进行被动连接建立和服务初始化的通用处理。

  每一步处理都委托给了同样在接受器接口中声明的钩子方法[Pree95]。具体接受器可重载这些钩子方法，进行与应用有关的处理，如使用某一具体的IPC机制被动地建立连接。一个工厂方法[GoF95]可以操纵由具体的连接器创建的具体服务处理程序，该方法是作为接受器的accept()模板方法中的一个步骤而被调用的。
- 参数化类型。这种策略将用于被动连接建立的IPC机制封装在包装器外观中，并通过参数化类型将它们配置在接受器中。和上述的多态性策略中一样，接受器的accept()是一个模板方法[GoF95]，其钩子方法代表了配置到接受器中的特定的IPC机制。实例化一个具体的接受器时使用的具体服务处理程序的类型也可以作为模板参数提供给通用接受器。

使用参数化类型的一个好处是可以很容易和有效地改变与接受器有关的IPC连接机制和服务处理程序。这种灵活性大大简化了将一个接受器的连接建立的代码移植到具有不同IPC机制的平台上所需做的工作，也使得可以在不同类型的具体服务处理程序中使用同样的连接建立和服务初始化代码。 301

继承和参数化类型的性能比较：

- 参数化类型会带来额外的编译和连接上的时间开销，但编译后的代码运行快[CarEl95]。
- 由于不直接的动态绑定[HLS97]，继承会产生运行时开销，但编译和连接速度快。

不同的应用程序和应用服务有最适合于某种策略的不同的需求。

➡网关例子中使用了参数化类型策略，用指定的服务处理程序和被动建立连接的具体IPC机制配置一个接受器。该接受器继承Event_Handler，接收来自于反应器单件的事件，该反应器单件担当着分配器的角色。

```
template <class SERVICE_HANDLER, class IPC_ACCEPTOR>
    // The <SERVICE_HANDLER> is the type of concrete
    // service handler created/accepted/activated when a
    // connection request arrives.
    // The <IPC_ACCEPTOR> provides the concrete IPC
    // passive connection mechanism.
```

```
class Acceptor : public Event_Handler {
public:
    typedef typename IPC_ACCEPTOR::PEER_ADDR Addr;

    // Constructor initializes <local_addr> transport
    // endpoint and register with the <Reactor>.
    Acceptor (const Addr &local_addr, Reactor *r);

    // Template method that creates, connects,
    // and activates <SERVICE_HANDLER>'s.
    virtual void accept ();
protected:
    // Factory method hook for creation strategy.
    virtual SERVICE_HANDLER *make_service_handler ();

    // Hook method for connection strategy.
    virtual void accept_service_handler
                (SERVICE_HANDLER *);

    // Hook method for activation strategy.
    virtual void activate_service_handler
                (SERVICE_HANDLER *);

    // Hook method that returns the I/O <HANDLE>.
    virtual HANDLE get_handle () const;

    // Hook method invoked by <Reactor> when a
    // connection request arrives.
    virtual void handle_event (HANDLE, Event_Type);

private:
    // Template 'placeholder' for a concrete IPC
    // mechanism that establishes connections passively.
    IPC_ACCEPTOR peer_acceptor_;
};
```

302

用具体的IPC_ACCEPTOR和SERVICE_HANDLER类型对接受器模板进行参数化。IPC_ACCEPTOR表示具体IPC机制，接受器使用该机制被动地建立由对等体连接器启动的连接。SERVICE_HANDLER表示具体的服务处理程序，它处理与对等体连接的服务处理程序之间的数据交换。由应用层中的组件提供了这两种具体的类型。 ❑

(2.3) *实现通用的接受器方法*。应用程序通过调用其初始化方法来初始化接受器。调用初始化方法时要有一个指明传输地址（如本地主机IP名和TCP端口号）的参数。接受器使用这个地址参数侦听对等体连接器启动的连接请求。它通过接受器的具体接受器子类或通过类型化参数将这个地址传递给配置到通用接受器中的具体IPC连接机制。然后IPC连接机制就初始化接受器的被动模式传输端点，将它的地址告知要与该接受器建立连接的远程应用程序。

用于定制通用接受器的具体IPC机制的类型决定了接受器的被动模式传输端点的性质。通常使用包装器外观，如Socket[Ste98]的ACE包装器外观[Sch92]、TLI[Rago93]、STREAM管道[PR90]，或者Win32的命名管道[Ric97]来访问该IPC机制。

接受器的初始化方法也会在分配器上注册它自己。然后该分配器向接受器执行一个“双分配”[GoF95]，以得到底层具体IPC机制的被动模式传输端点的句柄。对等体连接器的连接请求到达时分配器利用该句柄通知接受器。

303

➥网关例子中接受器进行初始化的构造函数方法如下：

```
template <class SERVICE_HANDLER, class IPC_ACCEPTOR>
void Acceptor<SERVICE_HANDLER, IPC_ACCEPTOR>::Acceptor
    (const Addr &local_addr, Reactor *reactor) {
    // Initialize the IPC_ACCEPTOR.
    peer_acceptor_.open (local_addr);

    // Register with <reactor>, which uses <get_handle>
    // to get handle via 'double-dispatching.'
    reactor->register_handler (this, ACCEPT_MASK);
}
```

❑

当从远程对等体来的连接请求到达后，分配器自动回调接受器的accept()模板方法[GoF95]。该模板方法实现了接受器的创建新的具体服务处理程序、接收连接和激活该处理程序的策略。接受器的具体实现由钩子方法完成。这些钩子方法代表实现个性化的服务处理程序连接和初始化策略时一组可用的操作。

如果使用多态性策略实现具体的接受器，在具体接受器子类中的钩子方法被分配给具体接受器子类中相应的实现。当使用参数化类型时，钩子方法对用于实例化通用接受器的模板参数调用相应的方法。在这两种情况中，具体接受器都可以在不改变accept()方法接口的情况下透明地改变通用的接受器策略。有了这种灵活性，才可以设计出将其行为与被动连接和初始化分开的具体的服务处理程序。

304

➡网关实例中的接受器类实现了下面的accept()方法：

```
template <class SERVICE_HANDLER, class IPC_ACCEPTOR>
void Acceptor<SERVICE_HANDLER, IPC_ACCEPTOR>::accept () {
    // The following methods comprise the core
    // strategies of the <accept> template method.

    // Factory method creates a new <SERVICE_HANDLER>.
    SERVICE_HANDLER *service_handler =
        make_service_handler ();

    // Hook method that accepts a connection passively.
    accept_service_handler (service_handler);

    // Hook method that activates the <SERVICE_HANDLER>
    // by invoking its <open> activation hook method.
    activate_service_handler (service_handler);
}
```

通用接受器模板方法使用make_service_handler()钩子工厂方法[GoF95]建立新的具体的服务处理程序。用accept_service_handler()钩子方法定义其连接接受策略。缺省时该方法将连接建立的处理交给IPC_ACCEPTOR的accept()方法，它定义了一个具体的被动IPC连接机制。由activate_service_handler()方法定义接受器的服务处理程序激活策略。一个具体的服务处理程序可以使用该方法进行初始化并选择并发策略。

在网关例子中，分配器是一个反应器，它通过接受器从Event_Handler类继承的handle_event()方法间接地通知接受器的accept()方法。handle_event()方法是一个适配器[GoF95]，它转换反应器的通用事件处理接口以通知接受器的accept()方法。 ❑

当由于出错或应用程序关闭而导致接受器中止时，分配器通知接受器释放它动态获得的所有资源。

➡在网关例子中，反应器调用接受器的handle_close()钩子方法来关闭它的被动模式的套接字。 ❑

305

(2.4) 定义通用的连接器接口。连接器组件实现了用于主动建立连接和初始化服务处理程序的通用策略，这些服务处理程序负责处理连接上的请求和响应事件。

连接器中有一个具体服务处理程序的映像，这些具体服务处理程序管理异步连接的完成处理。被异步启动连接的服务处理程序将被插入到该映像中。这样，分配器和连接器可以在连接完成后激活这些处理程序。

和实现活动（2.2）中介绍的通用接受器一样，实现活动（3）中介绍的应用层组件用某些具体服务处理程序和IPC机制定制通用连接器。因此我们必须选择多态性或参数化类型策略来定制具体连接器。实现活动（2.2）介绍了这两种策略并对它们进行了比较。

和接受器的accept()方法一样，连接器的连接启动方法connect()和完成方法complete()是模板方法[GoF95]。这些方法实现了主动地建立连接和初始化服务处理程序的通用策略。由钩子方法完成这些策略的具体步骤[Pree95]。

➡在网关例子中为连接器定义了下面的接口。它使用C++模板，用一个具体服务处理程序和具体IPC连接机制配置连接器。连接器从Event_Handler类派生，从反应器分配器中接收异步完成事件通知。

```
template <class SERVICE_HANDLER, class IPC_CONNECTOR>
    // The <SERVICE_HANDLER> is the type of concrete
    // service handler activated when a connection
    // request completes. The <IPC_CONNECTOR> provides
    // the concrete IPC active connection mechanism.
class Connector : public Event_Handler {
public:
    enum Connection_Mode {
        SYNC, // Initiate connection synchronously.
        ASYNC // Initiate connection asynchronously.
    };
    typedef typename IPC_CONNECTOR::PEER_ADDR Addr;

    // Initialization method that caches a <Reactor> to
    // use for asynchronous notification.
    Connector (Reactor *reactor): reactor_ (reactor) { }

    // Template method that actively connects a service to
    // a <remote_addr>.
    void connect (SERVICE_HANDLER *sh,
                  const Addr &remote_addr,
                  Connection_Mode mode);
protected:
    // Hook method for the active connection strategy.
    virtual void connect_service_handler
        (const Addr &addr, Connection_Mode mode);

    // Register the <SERVICE_HANDLER> so that it can be
    // activated when the connection completes.
    int register_handler (SERVICE_HANDLER *sh,
                          Connection_Mode mode);

    // Hook method for the activation strategy.
    virtual void activate_service_handler
        (SERVICE_HANDLER *sh);
```

306

```
    // Template method that activates a <SERVICE_HANDLER>
    // whose non-blocking connection completed. This
    // method is called by <connect> in the synchronous
    // case or by <handle_event> in the asynchronous case.
    virtual void complete (HANDLE handle);
private:
    // Template 'placeholder' for a concrete IPC
    // mechanism that establishes connections actively.
    IPC_CONNECTOR connector_;

    typedef map<HANDLE, SERVICE_HANDLER*> Connection_Map;

    // C++ standard library map that associates <HANDLE>s
    // with <SERVICE_HANDLER> *s for pending connections.
    Connection_Map connection_map_;

    // <Reactor> used for asynchronous connection
    // completion event notifications.
    Reactor *reactor_;

    // Inherited from <Event_Handler> to allow the
    // <Reactor> to notify the <Connector> when events
    // complete asynchronously.
    virtual void handle_event (HANDLE, Event_Type);
};
```

307

用具体的IPC_CONNECTOR和SERVICE_HANDLER作为Connector模板的参数。IPC_CONNECTOR是连接器用以主动地与远程接受器同步或异步地建立连接的具体IPC机制。SERVICE_HANDLER模板参数定义了处理与连接的对等体服务处理程序交换的数据的一半服务。由应用层中的组件提供这两种具体类型。 ❑

(2.5) *实现通用连接器方法*。应用程序用连接器的connect()方法来启动一个连接，具体连接器可以使用该模板方法[GoF95]，在不改变连接器接口和实现的情况下透明地修改主动连接策略。因此，connect()将其连接策略中的具体步骤交给钩子方法去完成，具体连接器可以重载这些钩子方法以进行个性化的操作。

当connect()代表服务处理程序异步地建立一个连接时，连接器将该处理程序插入到一个内部容器中（在例子中是一个C++标准模板库map[Aus98]），由该容器记录待处理的连接。当异步启动的连接完成时，分配器通知连接器。连接器使用待处理的连接映射来完成对与该连接有关的服务处理程序的激活处理。

如果connect()同步地建立连接，连接器可以直接调用具体服务处理程序的激活钩子，而不用调用complete()。这样减少了不必要的用于同步连接建立和服务处理程序初始化的动态资源管理和处理。

➡下面是Connector的connect()方法的代码段。如果该方法的Connection_Mode参数取值SYNC，在连接完成后这个具体服务处理程序会被同步地激活。相反，通过向connect()方法的Connection_Mode传递ASYNC值，可以异步地启动连接：

```
template <class SERVICE_HANDLER, class IPC_CONNECTOR>
void Connector<SERVICE_HANDLER,IPC_CONNECTOR>::connect
        (SERVICE_HANDLER *service_handler,
         const Addr &addr, Connection_Mode mode) {
    // Hook method delegates connection initiation.
    connect_service_handler (service_handler, addr,mode);
}
```

308

模板方法connect()将连接启动的任务交给connect_service_handler()钩子方法。该方法定义了一种连接器连接策略的缺省实现。该策略使用由IPC_CONNECTOR模板参数提供的具体IPC机制同步或异步地建立连接。

```
template <class SERVICE_HANDLER, class IPC_CONNECTOR>
void Connector<SERVICE_HANDLER,
               IPC_CONNECTOR>::connect_service_handler
          (SERVICE_HANDLER *svc_handler,
           const Addr &addr,
           Connection_Mode mode) {
     try {
          // Concrete IPC_CONNECTOR establishes connection.
          connector_.connect (*svc_handler, addr, mode);

          // Activate if we connect synchronously.
          activate_service_handler (svc_handler);
     } catch (System_Ex &ex) {
          if (ex.status () == EWOULDBLOCK && mode == ASYNC)
          {
               // Connection did not complete immediately,
               // so register with <reactor_>, which
               // notifies <Connector> when the connection
               // completes.
               reactor_ ()->register_handler (this,
                                              WRITE_MASK);

               // Store <SERVICE_HANDLER *> in map.
               connection_map_ [connector_.get_handle ()]
                    = svc_handler;
          }
     }
}
```

请留意，当连接正好同步地完成时，connect()是如何直接激活具体服务处理程序的。这是可能发生的，例如，当对等体接受器位于同一主机或进程中时。 ❑

在启动的连接完成后，complete()方法激活具体服务处理程序。对于被同步启动的连接，如果connect()方法直接激活服务处理程序，就不需要调用complete()。不过，对于被异步启动的连接，连接完成时由分配器调用complete()。complete()方法检查具体服务处理程序的映射，寻找刚刚完成连接的服务处理程序。它将该服务处理程序从映射中删除。然后调用它的激活钩子方法。另外，complete()从分配器中注销该连接器，以防止它偶尔地通知连接器。

309

注意，无论连接是同步还是异步建立的，甚至当它们主动或被动地连接上时，都要调用服务处理程序的open()激活钩子方法。有了这种统一性，就可以定义这样的具体服务处理程序，其处理可以完全与其被连接和初始化的时间和方式分开。

➡在网关例子中，如上所述，同步地启动一个连接时，由Connector的connect_service_handler()方法而不是它的complete()方法来启动与之相关的具体服务处理程序。相反，对于异步连接，反应器通知由Connector从Event_Handler类继承的handle_event()方法。

该方法被实现为一个适配器[GoF95]，它转换反应器的事件处理接口，以进一步调用连接器的complete()方法。然后complete()方法调用一个具体服务处理程序的激活钩子方法，该具体服务处理程序的异步激活的连接是最近成功完成的。

下面的complete()方法必须从待处理连接的内部映射中找出并删除连接的服务处理程序，并将套接字句柄传递给服务处理程序。

```
template <class SERVICE_HANDLER, class IPC_CONNECTOR>
void Connector<SERVICE_HANDLER,
               IPC_CONNECTOR>::complete(HANDLE handle) {
    // Find <service_handler> associated with <handle> in
    // the map of pending connections.
    Connection_Map::iterator i =
        connection_map_.find (handle);

    if (i == connection_map_.end ())
        throw /* ...some type of error... */;

    // We just want the value part of the <key, value>
    // pair in the map
    SERVICE_HANDLER *svc_handler = (*i).second;

    // Transfer I/O handle to <service_handler>.
    svc_handler->set_handle (handle);

    // Remove handle from <Reactor> and .
    // from the pending connection map.
    reactor_->remove_handler (handle, WRITE_MASK);
    connection_map_.erase (i);

    // Connection is complete, so activate handler.
    activate_service_handler (svc_handler);
}
```

310

请留意complete()方法是如何通过调用其activate_service_handler()方法而初始化服务处理程序的。该方法实现了在具体服务处理程序的open()激活钩子方法中指定的初始化策略。 ❑

(3) *实现应用层的组件*。本层定义了在接受器-连接器模式的“*结构*”一节中介绍的具体服务处理程序、具体接受器和具体连接器。应用层中的组件对实现活动（2）中介绍的通用服务处理程序、接受器和连接器组件进行实例化，以产生特殊的具体组件。

具体服务处理程序定义了应用程序的服务。在实现网络化系统中由多个对等体服务处理程序组成的一个端到端的服务时，使用半对象和协议模式[Mes95]有助于改进这些服务处理程序的实现结构。特别是，半对象和协议模式将端到端服务的责任分解成服务处理程序接口和它们之间进行协作所用的协议。

具体服务处理程序也可以定义一个服务的并发策略。例如，服务处理程序可以继承事件处理程序和使用反应器模式在一个单控制线程内处理从对等体传来的数据。反过来，服务处理程序也可以使用主动对象（Active Object）或监控对象（Monitor Object）模式处理来自于不同的控制线程而不是来自于由连接它的接受器所使用的控制线程的数据。

在“*已解决的例子*”一节中将介绍在不影响接受器-连接器模式的结构和特征的情况下，网关例子如何在具体服务处理程序中灵活地配置不同的并发策略。

311

具体连接器和具体接受器是创建具体服务处理程序的工厂。一般它们继承相应的通用类，以一种与应用程序有关的方式，特别是重载由accept()和connect()模板方法[GoF95]调用

的不同的钩子方法来创建具体服务处理程序。

实现具体接受器的另一种方式是和前一个实现活动所讨论的，并在后面的“已解决的例子”中要提到的一样，用一个具体服务处理程序和具体IPC被动连接机制来参数化通用接受器。类似地，可以用具体服务处理程序和具体IPC主动连接机制参数化通用连接器从而实现具体连接器。

应用层的组件还可以提供个性化的IPC机制，用于配置具体服务处理程序、具体连接器和具体接受器。可以按照包装器外观模式在一个单独的类中封装IPC机制。这些包装器外观创建和使用传输端点和传输句柄，传输句柄用来和连接的对等体服务处理程序透明地交换数据。

使用包装器外观模式可以简化编程、增加重用，并允许通过类属编程技术对IPC机制进行整体替换。例如，在“已解决的例子”一节中使用的SOCK_Connector、SOCK_Acceptor和SOCK_Stream类是由ACE C++ Socket包装器外观库[Sch92]提供的。

8. 已解决的例子

在卫星群管理例子中对等体主机和网关组件使用接受器-连接器模式来简化其连接建立和服务初始化过程（如图3-40）：

- 第一，说明如何实现对等体主机组件，它们在例子中是被动的角色。
- 第二，说明如何实现网关，它主动地与被动对等体主机建立连接。

通过使用接受器-连接器模式，可以在对实现对等体主机和网关服务的服务处理程序施加最少影响的情况下，转换或者组合这些角色。

312

实现对等体主机应用程序。每个对等体主机中包括三个组件：Status_Handler、Bulk_Data_Handler和Command_Handler，它们是处理与网关交换的路由消息的具体服务处理程序。

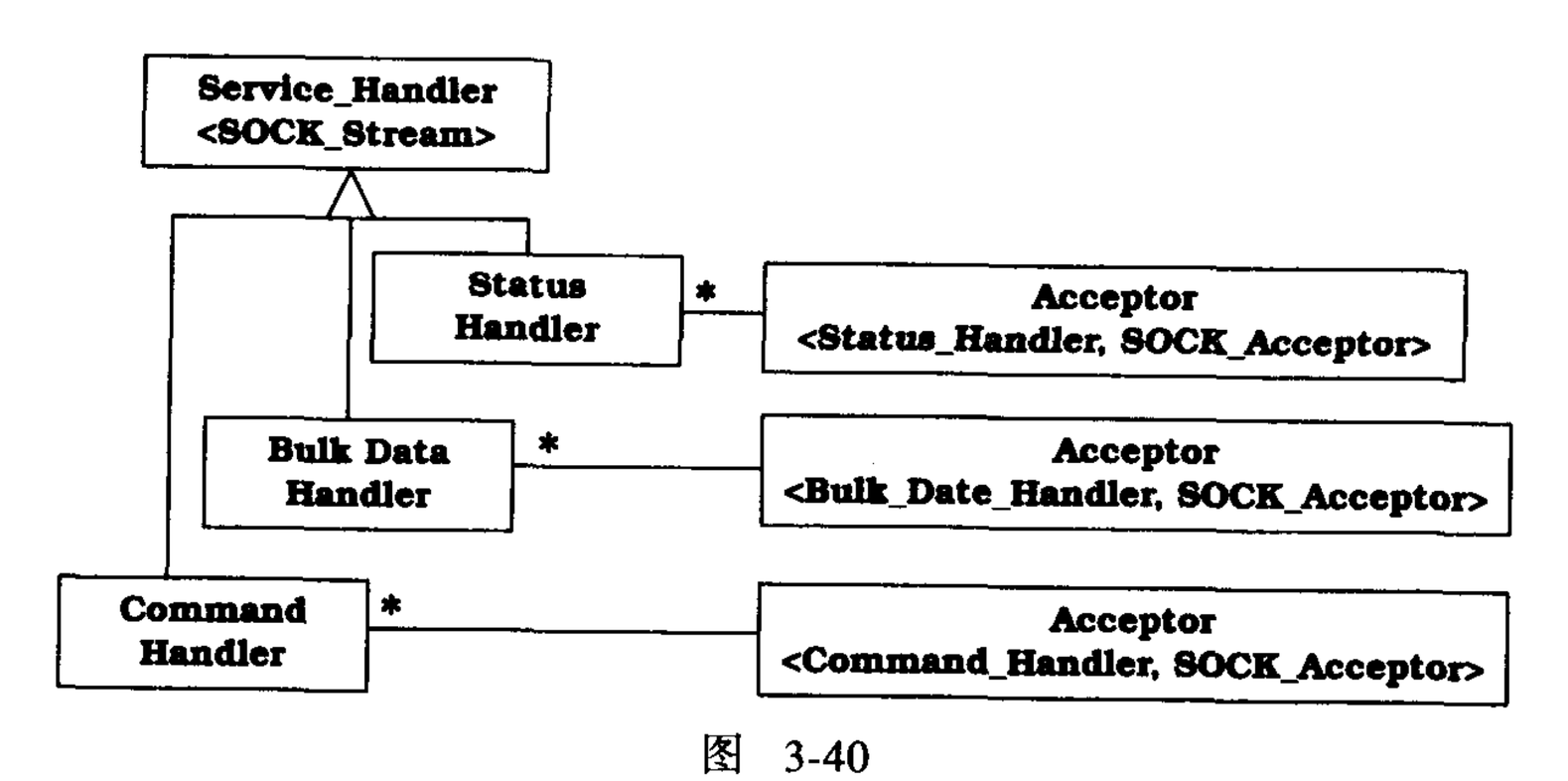


图 3-40

每个服务处理程序继承实现活动（2.4）中定义的Service_Handler类，这样接受器可以被动地初始化它们。每种类型的具体服务处理程序都有一个对应的具体接受器，该接受器创建、连接和初始化具体服务处理程序的实例。

为了说明接受器-连接器模式的灵活性，例子中每个具体服务处理程序的open()钩子方法实现了不同的并发策略。例如，当激活一个Status_Handler时，该服务处理程序运行在一个单独的线程中，Bulk_Data_Handler作为一个单独的进程运行，而Command_Handler和向

具体接受器多路分解连接请求的Reactor运行在同一线程中。注意，改变这种并发策略并不会影响Acceptor类的实现。

下面首先给出一个类型定义Peer_Handler：

```
typedef Service_Handler <SOCK_Stream> Peer_Handler;
```

该类型定义用SOCK_Stream包装器外观对Service_Handler通用模板类进行实例化。这个包装器外观[⊖]定义了一个具体的IPC机制，用于在连接的传输端点间使用TCP传输数据。这里的PEER_HANDLER类型定义是例子中后面使用的所有具体服务处理程序的基础。例如，下面的Status_Handler类从Peer_Handler派生，并处理与网关交换的状态数据，如遥测信息：

313

```
class Status_Handler : public Peer_Handler {
public:
    // Performs handler activation.
    virtual void open () {
        // Make this handler run in separate thread (note
        // that <Thread::spawn> requires a pointer to
        // a static method as the thread entry point).
        Thread_Manager::instance ()->spawn
            (&Status_Handler::svc_run, this);
    }

    // Static entry point into thread. This method can
    // block on the handle_event() call because it
    // runs in its own thread.
    static void *svc_run (Status_Handler *this_obj) {
        for (;;)
            this_obj->run ();
    }

    // Receive and process status data from Gateway.
    virtual void run () {
        char buf[BUFSIZ];
        peer ().recv (buf, sizeof buf);
        // ...
    }
    // ...
};
```

例子中的另外两个具体服务处理程序（Bulk_Data_Handler和Command_Handler）也是Peer_Handler的子类。它们的主要区别在于open()和handle_event()方法的不同实现。选择的并发机制不同，这些实现也有些差异。

下面显示的Status_Acceptor、Bulk_Data_Acceptor和Command_Acceptor类定义是分别创建、连接和激活Status_Handlers、Bulk_Data_Handlers和Command_Handlers具体服务处理程序的具体接受器工厂的模板实例。

314

```
// Accept connection requests from the gateway and
// activate a <Status_Handler> to process status data.
typedef Acceptor<Status_Handler, SOCK_Acceptor>
        Status_Acceptor;

// Accept connection requests from a gateway and activate
```

⊖ 在包装器外观模式的“实现”一节列出了SOCK_Stream的完整接口。

```
// a <Bulk_Data_Handler> to process bulk data requests.
typedef Acceptor<Bulk_Data_Handler, SOCK_Acceptor>
        Bulk_Data_Acceptor;

// Accept connection requests from a gateway and
// activate a <Command_Handler> to process commands.
typedef Acceptor<Command_Handler, SOCK_Acceptor>
        Command_Acceptor;
```

上面给出的类型定义都是通过用SOCK_Acceptor包装器外观对实现活动(2.2)中定义的通用Acceptor模板类进行实例化而得到的，该SOCK_Acceptor包装器外观是被动建立连接的具体IPC机制。[⊖]

请留意使用C++模板和动态绑定为何能灵活改变具体接受器和具体服务处理程序的某些细节，特别是改变Status_Handler、Bulk_Data_Handler和/或Command_Handler的并发策略时，不需要改变接受器组件。这种设计的灵活性来源于接受器-连接器模式所强调的事务分离。特别是将并发策略提取出来，在具体服务处理程序中实现，而不是与接受器紧密耦合。

对等体主机应用程序的主函数向具体接受器的构造函数传递用以向对等体连接器通告每个服务的TCP端口，从而初始化该具体接受器。如实现活动(2.3)所述，每个具体接受器自动地将它自己注册在作为参数传递给其构造函数的一个Reactor实例上。

```
// Main program run on a peer host.
int main () {
    // Initialize concrete acceptors to listen for
    // connections on their well-known ports.
    Status_Acceptor s_acceptor (STATUS_PORT,
                                Reactor::instance ());
    Bulk_Data_Acceptor bd_acceptor (BULK_DATA_PORT,
                                    Reactor::instance ());

    Command_Acceptor c_acceptor (COMMAND_PORT,
                                 Reactor::instance ());

    // Event loop that accepts connection request
    // events and processes data from a gateway.
    for (;;)
        Reactor::instance ()->handle_events ();
    /* NOTREACHED */
}
```

315

在初始化三个具体接受器后，主对等体主机的应用程序进入一个事件循环，使用反应器单件检测来自于网关的连接请求。出现这样的连接请求时，反应器通知相应的具体接受器，该接受器创建相应的具体服务处理程序，接受到该处理程序的连接，并激活该处理程序，以便能与网关交换信息。

实现网关应用程序。上面的main()函数说明了在卫星群管理例子中如何为对等体主机应用程序定义具体接受器和具体服务处理程序。图3-41说明如何实现网关应用中的具体连接器和相应的具体服务处理程序。

⊖ 在包装器外观模式的“实现”一节列出了SOCK_Acceptor的完整接口。

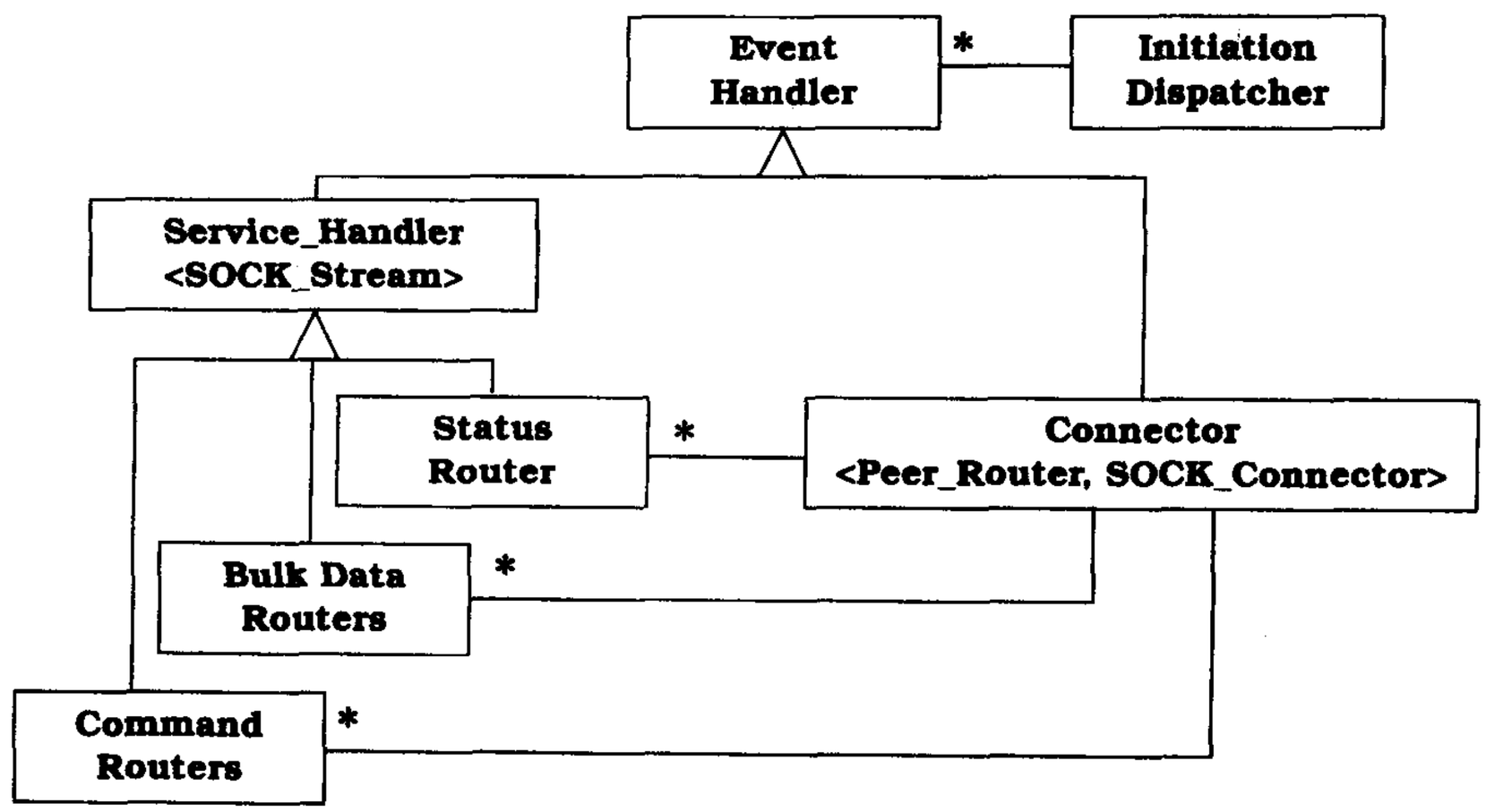


图 3-41

316

一个网关包括Status_Router、Bulk_Data_Router和Command_Router具体服务处理程序的多个实例，它们向一个或几个对等体目标主机发送从对等体源主机接收来的数据。这些具体服务处理程序继承Service_Handler类，这样连接器可以主动地连接它们，并自动地初始化。

具体服务处理程序中的每个open()钩子方法都实现不同的并发策略。这类似于对等体主机应用程序中定义的处理程序。和以前的一样，改变这些并发策略并不影响Connector的实现，从而使Connector具有很高的灵活性和可重用性。

下面首先定义一个用于TCP/IP数据传输的具体服务处理程序。用由SOCK_Stream包装器外观封装的适当的IPC机制对它进行实例化：

```
typedef Service_handler <SOCK_Stream> Peer_Router;
```

该类型定义是网关中所有其他服务的基类。例如，Bulk_Data_Router按如下方式与对等体主机交换数据：

```
class Bulk_Data_Router : public Peer_Router {
public:
    // Activates router in separate process.
    virtual void open () {
        if (fork () == 0) // In child process.
            // This method can block because it runs in
            // its own process.
            for (;;)
                run ();
        // ...
    }

    // Receive and route bulk data from/to Peers.
    virtual void run () {
        char buf[BUFSIZ];
        peer ().recv (buf, sizeof buf);
        // Routing takes place here...
    }
};
```

和对等体主机应用程序中定义的有自己的运行线程的Status_Handler不一样，Bulk_Data_Router运行在自己的进程中。同样可以为Status_Router和Command_Router定义类似的子类。和前面一样，所有的子类之间主要在open()和handle_event()方法实现上有差别，这些差别依赖于所选择的并发机制。

317

下面是用于Peer_Router的连接器的类型定义：

```
typedef Connector<Peer_Router, SOCK_Connector>
        Peer_Connector;
```

本类型定义用SOCK_Connector包装器外观对实现活动(2.4)中定义的Connector类进行实例化。该包装器外观是与远程对等体传输端点主动地建立连接的具体IPC机制。网关应用程序中只需要一个具体连接器，因为传递给connect()方法的具体服务处理程序在具体连接器之外创建和初始化。因此可以像对待Peer_Router一样对待网关的具体服务处理程序，如Bulk_Data_Router或Command_Router。

相反，对等体主机应用程序要求每种类型的具体服务处理程序（如Bulk_Data_Handler、Command_Handler）都有一个具体接受器，因为必须在具体Acceptor的模板实例的型构（Signature）中先验地指定服务处理程序的具体类型。

在网关应用程序的主函数中，由函数get_peer_addrs()创建具体服务处理程序Status_Router、Bulk_Data_Router和Command_Router。该函数从配置文件或命名服务中读取对等体地址列表。每个对等体地址由IP主机地址和端口号组成。一旦初始化了这些具体服务处理程序，就通过向具体连接器的connect()方法传递ASYNC标志异步地初始化所有的连接。

```
// Obtain a C++ standary library vector of
// <Status_Router>s, <Bulk_Data_Router>s, and
// <Command_Router>s from a configuration file.
void get_peer_addrs (vector<Peer_Router> &peers);

// The gateway application's main entry point.
int main () {
    // Concrete connector that serves as a factory
    // for <Peer_Router>s.
    Peer_Connector peer_connector (Reactor::instance ());
    // A vector of <Peer_Router>s that perform
    // the gateway's routing services.
    vector<Peer_Router> peers;

    // Get vector of peers to connect with.
    get_peer_addrs (peers);

    // Iterate through all the <Peer_Router>s and
    // initiate connections asynchronously.
    typedef vector<Peer_Router>::iterator Peer_Iterator;
    for (Peer_Iterator peer = peers.begin ();
         peer != peers.end ();
         ++peer) {
         peer_connector.connect (*peer,
                                 peer->remote_addr (),
                                 Peer_Connector::ASYNC);
    }
```

318

```
    // Event loop that handles connection indication
    // events and routes data from peer hosts.
    for (;;)
        Reactor::instance ()->handle_events ();
    /* NOTREACHED */
}
```

通过peer_connector的complete()方法异步调用所有的连接并同时完成这些连接。在反应器的事件循环语境中，反应器通过回调通知complete()方法。反应器的事件循环也为Command_Router对象多路分解和分配发送事件。这些和反应器在同一个控制线程中运行。相反，Status_Router和Bulk_Data_Router的实例分别在独立的线程和进程中执行。

9. 已知使用

UNIX网络超级服务器。像Inetd[Ste98]、Listen[Rago93]以及来自ACE框架的服务配置器[JS97b]等超级服务器实现都使用一个主接受器进程在一组通信端口上侦听连接。例如在Inetd中，每个端口关联一个服务，如标准Internet服务FTP、TELNET、DAYTIME以及ECHO。接受器进程将Inetd超级服务器的功能分解成两个独立的部分：一个用于建立连接，另一个接收和处理对等体的请求。在由Inetd监控的端口上出现一个服务请求后，它接收该请求，分配相应的预先注册的处理程序来完成该服务。

319

CORBA对象请求代理（ORB）[OMG98a]。在CORBA的许多实现中，当客户机请求对象请求代理（Object Request Broker, ORB）服务时，ORB核心层使用接受器-连接器模式被动或主动地初始化连接处理程序。[SC99]中描述了如何使用接受器-连接器模式在The ACE ORB（TAO）中实现ORB核心部分。TAO是CORBA的一种实时的实现。

Web浏览器。Web浏览器中的HTML分析组件（如Netscape和Internet资源管理器）使用异步形式的连接器组件与服务器建立连接，这些服务器是与嵌入在HTML网页中的图片有关的。使用这种模式，可以异步地激活多个HTTP连接。这样可避免浏览器的主事件循环阻塞的可能性。

爱立信EOS呼叫中心管理系统。该系统使用接受器-连接器模式使应用级的呼叫中心管理者事件服务器[SchSu94]主动地和网络化中心管理系统中被动的超级用户建立连接。

Project Spectrum。Project Spectrum[BBC94]的高速医疗影像传输子系统采用接受器-连接器模式被动地建立连接，初始化用于存储大量医疗影像的应用服务。一旦连接建立，应用程序与影像存储器之间就可以来回发送和接收几兆的医疗影像数据。

ACE[Sch97]。ACE的并发的面向对象网络编程环境以可重用的C++类的形式提供了“实现”一节介绍的通用Service_handler、Connector和Acceptor组件的实现。

Java ACE[JACE99]是ACE的Java实现版本，它提供了JACE. Connection. SvcHandler、JACE.Connection.Acceptor和JACE.Connection.Connector组件。它们分别对应于接受器-连接器模式的服务处理程序、接受器和连接器参与者。

经理和秘书。在为经理配备秘书的组织中，可以找到接受器-连接器模式在现实生活中的实

320 现例子。一个经理想给另一位经理打电话，他并不直接拨号，而是吩咐秘书为他呼叫。被呼叫经理并不直接接收呼叫，而是由秘书接收呼叫。在连接建立后，才由两位经理通话。接照接受器-连接器模式的术语，启动电话呼叫的秘书是连接器，接收呼叫的秘书是接受器，而两位经理则是对等体服务处理程序。

10. 结论

接受器-连接器模式有如下三点**优点**：

可重用性、可移植性和可扩展性。接受器-连接器模式将连接和初始化服务处理程序的机制与在此之后进行的服务处理分开。接受器和连接器中与应用无关的机制是可重用的组件，它们知道如何建立连接以及在连接建立时如何初始化相关的服务处理程序。类似地，服务处理程序知道如何进行应用特定的服务处理。

这种严格的事务分离是通过将连接和初始化策略与服务处理策略分开而得到的。因此可以独立地改进每一部分。连接和初始化策略可以只写一次，放在类库或类框架中，并通过继承、对象组合或模板实例化得到重用。因此不同的应用程序并不需要重新编写同样的连接和初始化代码。

相反，根据不同的应用需求，服务处理程序可以有很大的不同。通过将接受器和连接器组件用某种类型的服务处理程序参数化，这种差异的影响只局限在少量软件组件中。

健壮性。接受器-连接器模式将服务处理程序与接受器分开。这种分离能确保被动模式传输端点（在网关实例中为PEER_ACCEPTOR）不能用于读/写数据。这样增加了类型安全性，能消除使用弱类型的网络编程接口（如Socket或TLI[SHS95]）时经常会出现的一类错误。

高效性。接受器-连接器模式可以在延迟很大的广域网中异步地和高效地与大量主机主动地
321 建立连接。在这种情况下，异步是很重要的，因为大规模网络系统可能有成千上万的主机需要连接。

一种将所有主机连到网关的方式是使用“实现”一节描述的同步机制。但是在延时长的广域网（通过如地球同步卫星或跨大西洋的光缆）中3段TCP连接握手中的每次握手延时可能是几秒钟。在这种情况下，异步连接机制会更好，因为它们能够利用广域网中网络和主机的内在并行性。

接受器-连接器模式有如下**不足**：

增加了间接性。与直接使用底层的网络编程接口相比，接受器-连接器模式是不直接的。然而，当编译器将用于实现该模式的方法内联时，使用像C++、Ada、Eiffel这样的支持参数化类型的语言可以在不明显增加开销的情况下实现这些模式。

增加了复杂性。对于只连接到一个服务器，使用单一网络编程接口执行一个服务的简单客户机应用程序，接受器-连接器模式增加了不必要的复杂性。不过使用通用的接受器和连接器包装器外观可以使开发人员不用考虑令人厌烦的、易出错的和不可移植的低层网络编程机制，从而简化了应用程序的设计。

参见

在将主动连接建立与随后的服务处理分开这一点上，接受器-连接器模式和客户机-分配器-

服务器（Client-Dispatcher-Server）模式[POSA1]的目标是类似的。主要的区别在于接受器-连接器模式考虑同步和异步连接的被动和主动连接建立和初始化。相反，客户机-分配器-服务器模式局限于同步连接建立。

致谢

Eric Samuelsson对运行示例中的UML类图提出了好几处改进意见。

322

第4章 同步模式

"我将成为所有病人的模式"。
威廉·莎士比亚

本章介绍能简化并发系统加锁机制的一种惯用法和三种模式：定界加锁、策略化加锁、线程安全接口以及双检查加锁优化。 323

开发多线程应用程序比开发顺序应用程序要困难，因为在多线程应用程序中，多个线程可以并发地操作一个对象，从而可能破坏其内部状态。使用互斥和信号灯这样的同步机制可以确保正确地串行处理对象。本章介绍三种模式和一种惯用法，解决与同步并发对象有关的问题。

第一类惯用法和模式是关于获取/释放锁以及加锁策略的：

- C++惯用法定界加锁（Scoped Locking）能确保当控制进入到某一范围时，自动获得锁，而当控制离开该范围时，自动释放锁，不管从该范围返回的路径是什么。
- 策略化加锁(Strategized Locking)设计模式是策略模式[GoF95]的一种特例，它把同步机制参数化，这一机制保护临界区免受并发访问。

用C++实现策略化加锁模式时，通常使用定界加锁惯用法。另外两种模式有助于改善同步机制的健壮性和有效性。

- 线程安全接口(Thread-Safe Interface)设计模式使加锁的开销减至最少，确保组件内部的方法调用不会因为要再次获得一个组件已经拥有的锁而"自死锁"。
- 如果程序执行期间临界区代码只能以线程安全的方式获得一次锁，双检查加锁优化(Double-Checked Locking Optimization)设计模式能够降低争用和同步开销。

以上四种模式和惯用法可以用于改善第5章"并发模式"中介绍的各种模式的实现。

与同步有关的模式还包括：代码加锁和数据加锁[McK95]、阅读器/记录器加锁 [McK95] [Lea99a]、对象同步器[SPM99]，还有阻止与保护挂起[Lea99a]。 324

4.1 定界加锁

C++ 惯用法定界加锁（Scoped Locking）能确保当控制进入到某一范围时，自动获得锁，而当控制离开该范围时，自动释放锁，不管从该范围返回的路径是什么。

1. 别名 同步化阻塞（Synchronized Block）、资源获取初始化(Resource-Acquisition-is-

Initialization)[Str97]、[⊖] 哨兵（Guard）、围绕对象的执行（Execute Around Object）[Hen00]。

2. 例子

商业Web服务器中通常用“点击率”记录在一段时间内每个URL被客户机访问的次数。为了减少延迟，Web服务器进程将点击率计数器放在一个驻留内存的组件中而不是放在磁盘文件中。

为增加吞吐量，Web服务器进程通常是多线程的[HS98]。因此必须将点击率组件中的公有方法串行化，以防止并发地改变点击率时多线程破坏其内部状态。

串行化对点击率组件的访问的一种方法是在每个公有方法中显式地获得和释放一个锁。下面的C++例子使用在包装器外观模式中定义的Thread_Mutex来串行化对临界区的访问：

```
class Hit_Counter {
public:
    // Increment the hit count for a URL <path> name.
    bool increment (const string &path) {
        // Acquire lock to enter critical section.
        lock_.acquire ();
        Table_Entry *entry = lookup_or_create (path);
        if (entry == 0) {
            // Something's gone wrong, so bail out.
            lock_.release ();
            return false; // Return a 'failure' value.
        }
        else { // Increment hit count for <path> name.
            entry->increment_hit_count ();
            // Release lock to leave critical section.
            lock_.release ();
            return true;
        }
    }

    // Other public methods omitted...
private:
    // Lookup the table entry that maintains the hit count
    // associated with <path> name, creating the entry if
    // it doesn't exist.
    Table_Entry *lookup_or_create (const string &path);

    // Serialize access to the critical section.
    Thread_Mutex lock_;
};
```

325

尽管这段代码可以正常工作，但难以开发和维护Hit_Counter类。例如，进行维护的程序员可能在increment()方法之外的某些返回路径上忘记了释放lock_，如当修改其else分支以检查一些新的失败条件时：

```
else if (entry->increment_hit_count () == SOME_FAILURE)
    return false; // Return a 'failure' value.
```

另外，这种实现对异常也不是异常安全（exception-safe）的。因此，如果increment()方法的后续版本抛出一个异常或者调用一个抛出异常的方法，那么lock_就不会被释放。

⊖ 定界加锁惯用法是Stroustrup的“资源获取初始化”惯用法[Str97]的特化。这里介绍该惯用法是为了使本书内容完整，并说明Stroustrup的惯用法可以用于并发程序。

这两种修改方法都会导致increment()方法在没有释放lock_的情况下返回。然而，如果lock_不被释放、试图得到lock_的其他线程永远阻塞时，Web服务器进程也被挂起。而且，如果这种情况很少出现，在系统测试期间这些代码所带来的问题往往不能表现出来。

3. 语境

包含由多线程并发操纵的共享资源的并发应用程序。

4. 问题

必须保护不能被并发执行的代码，方法是使用某些类型的锁，当控制进入或离开一个临界区时分别获得或者释放这种锁。但是，如果程序员必须显式地获得或释放锁，就难以保证在经过该代码的所有路径上锁都被释放了。例如，在C++中，可以用return、break、continue或者goto语句以及由于一个传递到某区域之外的未处理异常而使控制离开该区域。 326

5. 解决方案

定义一个哨兵（guard）类，当控制进入一个区域时，哨兵类的构造函数自动获得一个锁；当控制离开这个区域时，哨兵类的析构函数自动释放该锁。将哨兵类实例化，以在定义临界区的方法和块区域中获得或释放锁。

6. 实现

定界加锁惯用法的实现很简明。

(1) 定义一个哨兵类，其构造函数和析构函数分别获得和释放某种类型的锁。哨兵类的构造函数中存放一个指向锁的指针或对锁的引用，据此获得该锁。析构函数使用由构造函数存储的指针或引用释放该锁。

➡下面是一个用于Thread_Mutex包装器外观的哨兵类：

```
class Thread_Mutex_Guard {
public:
    // Store a pointer to the lock and acquire the lock.
    Thread_Mutex_Guard (Thread_Mutex &lock)
        : lock_ (&lock), owner_ (false) {
        lock_->acquire ();

        // Only set to true if <acquire> succeeds.
        owner_ = true;
    }

    // Release the lock when the guard goes out of scope.
    ~Thread_Mutex_Guard () {
        // Only release the lock if it was acquired
        // successfully, i.e., <false> indicates that
        // <acquire> failed..
        if (owner_) lock_->release ();
    }
private:
    Thread_Mutex *lock_; // Pointer to our lock.
    bool owner_; // Is <lock_> held by this object?

    // Disallow copying or assignment.
    Thread_Mutex_Guard (const Thread_Mutex_Guard &);
    void operator= (const Thread_Mutex_Guard &);
};
```

327

在哨兵类的实现中，应使用指向锁的指针而不是使用锁对象，以防止对锁的复制或赋值。正如在包装器外观模式中所讨论的，对锁的复制或赋值会引起错误。

另外，增加一个标志是很有用的，如上述Thread_Mutex_Guard例子中的owner_标志，它表示哨兵是否成功地获得了锁。该标志也可以指示当错误地使用静态/全局锁[LGS99]时，由“初始化错误顺序”而导致的失败。通过在析构函数中检查这个标志，可以避免当哨兵释放它并不拥有的锁而产生的微小的运行时错误。

(2) *让临界区对应于哨兵对象的范围和生命期*。要避免并发访问临界区，应指定一个区域（如果还没有这样做的话），并由该区域的第一条语句在栈中创建一个哨兵对象。哨兵类的构造函数自动获得该锁。在离开临界区域时，自动调用释放该锁的析构函数。根据C++析构函数的语义，即便在临界区中抛出一个异常，这个保护锁也会被释放。

➡在多线程Web服务器中，定界加锁惯用法用Hit_Counter类解决了原有的问题：

```
class Hit_Counter {
public:
    // Increment the hit count for a URL <path> name.
    bool increment (const string &path) {
        // Use Scoped Locking to acquire and release
        // the <lock_> automatically.
        Thread_Mutex_Guard guard (lock_);

        Table_Entry *entry = lookup_or_create (path);
        if (entry == 0)
            return false; // Destructor releases <lock_>
        else {
            // Increment hit count for this <path> name.
            entry->increment_hit_count ();
            return true; // Destructor releases <lock_>
    }

    // Other public methods omitted.
private:
    // Serialize access to the critical section.
    Thread_Mutex lock_;
};
```

328

在这种解决方案中，哨兵能确保当控制进入和离开increment()方法时，能分别自动地获得或释放锁lock_。 ❑

7. 变体

显式访问器。“实现”一节中的Thread_Mutex_Guard接口有一个缺点，就是不可能在不离开一个方法或者块区域的情况下显式地释放锁。

➡例如在下面的代码段中，根据if语句的条件是否为真，可能两次释放锁：

```
{
    Thread_Mutex_Guard guard (lock);
    // Do some work ...
    if (/* a certain condition holds */)
        lock->release ()
    // Do some more work ...
    // Leave the scope, which releases the lock again.
}
```

❑

为了避免这种错误的用法，程序员不应该直接访问锁。相反可以在锁的哨兵类中定义对锁的显式访问器方法：

➡按如下方式修正Thread_Mutex_Guard类：

```
class Thread_Mutex_Guard {
public:
    // Store a pointer to the lock and acquire the lock.
    Thread_Mutex_Guard (Thread_Mutex &lock)
        : lock_ (&lock), owner_ (false) {
        acquire ();
    }

    void acquire () {
        lock_->acquire ();
        // Only set to <true> if <acquire> succeeds.
        owner_ = true;
    }

    void release () {
        // Only release <lock_> if it was acquired
        // successfully and we haven't released it yet!
        if (owner_) {
            owner_ = false;
            lock_->release ();
        }
    }

    // Release the lock when the guard goes out of scope.
    ~Thread_Mutex_Guard () { release (); }
private:
    Thread_Mutex *lock_; // Pointer to our lock.
    bool owner_; // Is <lock_> held by this object?

    // ... disallow copying and assignment ...
};
```

329

❑

acquire()和release()访问器方法跟踪锁是否已被释放。如果已被释放，在guard的析构函数中就不释放该锁。

➡使用改进过的Thread_Mutex_Guard，代码就能够正常工作了：

```
{
    Thread_Mutex_Guard guard (lock);
    // Do some work ...
    if (/* a certain condition holds */)
        guard.release ();
    // Do some more work ...
    // Leave the scope, lock is not released again.
}
```

❑

策略化定界加锁。为每种类型的锁定义一个哨兵很乏味，易出错，也是不必要的，因为它会增加应用程序或者组件占用的内存空间。因此，定界加锁惯用法的一个常用的变体就是使用策略化定界加锁模式的参数化类型或者多态性版本。

8. 已知使用

Booch组件。Booch组件[BV93]是用于多线程C++程序的第一个使用定界加锁惯用法的C++类库。

ACE[Sch97]。在自适应通信环境(ADAPTIVE Communication Environment, ACE)框架中广泛地使

330 用了定界加锁方法，其中定义了一个类似于在“实现”和“变体”两节中介绍的Thread_Mutex_Guard的ACE_Guard实现。

Threads.h++。按照ACE定界加锁的设计模型，Rogue Wave的Threads.h++库定义了一组哨兵类。

Java。Java定义了一个称为同步块的程序设计特性，它在该语言中实现了定界加锁惯用法。Java编译器产生了一个相应的字节代码指令块，该指令块以monitorenter开头，以monitorexit结尾。为了确保锁总能被释放，编译器还生成一个异常处理程序，以捕获同步块中抛出的所有异常[Eng99]。

9. 结论

定界加锁惯用法有如下**优点**：

增加了健壮性。通过使用这个方法，当控制进入或离开由C++方法和块区域定义的临界区时，会自动地获得和释放锁。这种方法消除了与同步和多线程有关的常见的编程错误，从而提高了并发应用程序的健壮性。

在并发应用程序和组件中使用定界加锁惯用法有两个**不足**：

递归使用时可能发生死锁。如果一个使用定界加锁惯用法的方法递归地调用自己，当该锁不是一个“递归”的互斥时，就会发生“自死锁”。线程安全接口模式中引入了一种避免该问题的技术。该模式只允许接口方法使用定界加锁惯用法，而实现方法中不能使用。

受与语言相关的语义的限制。定界加锁惯用法是基于C++语言特性的，因此不会和与操作系统有关的系统调用集成起来。这样当进程或线程在一个受保护的临界区内部失败或退出时，锁就不能自动地释放。同样，如果调用了标准C的 longjmp()函数，锁也不能被正确地释放，因为当运行栈清空时该函数不会调用C++对象的析构函数。

331

➡像下面这样修改increment()后无法实施定界加锁惯用法：

```
Thread_Mutex_Guard guard (&lock_);
Table_Entry *entry = lookup_or_create (path);
if (entry == 0)
    // Something's gone wrong, so exit the thread.
    thread_exit ();
    // Destructor will not be called so the
    // <lock_> will not be released!
```

❑

因此，一般来说，中止或退出一个线程或在一个组件内处理是不合适的。相反应使用异常处理机制或错误传递（error-propagation）模式[Mue96]。

过多的编译器警告。定界加锁惯用法定义了一个在一个区域中没有被显式使用的哨兵对象，因为其析构函数隐含地释放锁。不幸的是，当在某区域定义了哨兵而没有显式地使用到哨兵时，一些C++编译器会显示“statement has no effect”的警告信息。最好的情况下是这些警告信息仅仅分散注意力——而在最坏的情况下，这些警告信息会使开发人员关闭编译警告，从而屏蔽那些指示代码实际问题的警告。处理这个问题的一个有效的方法是使用宏，它能消除这些警告，又不会产生额外的代码。

➡下面是ACE[Sch97]中定义的宏：

```
#define UNUSED_ARG(arg) { if (&arg) /* null */; }
```

将该宏放在哨兵的后面，避免很多C++编译器产生假的警告：

```
{ // New scope.
    Thread_Mutex_Guard guard (lock_);
    UNUSED_ARG (guard);
    // ...
```

❑

参见

定界加锁惯用法是更一般的C++惯用法和围绕对象执行（Execute Around Object）惯用法[Hen00]的特例，后者要求当进入某个区域时构造函数获得一个资源，而当退出该区域时析构函数释放一个资源。当这些惯用法应用于并发应用程序时，获得和释放的资源就是某些类型的锁。

致谢

感谢Brad Appleton为定界加锁惯用法所提出的意见。 332

4.2 策略化加锁

策略化加锁(strategized locking)设计模式把同步机制参数化，这一机制保护组件的临界区免受并发访问。

1. 例子

文件缓存是实现高性能Web服务器必需的一个关键组件，它将URL路径名映射为内存映像文件或打开的文件句柄[HS98]。当客户机请求的URL在文件缓存中时，Web服务器可以立即将文件内容传送给客户机，而不需多次使用read()或write()操作访问速度更慢的辅助存储器。

可移植、高性能的文件缓存实现应该能有效地运行在多线程和单线程操作系统中。获得这种可移植性的方法之一是开发多个文件缓存文件类：

```
// A single-threaded file cache implementation.
class File_Cache_ST {
public:
    // Return a pointer to the memory-mapped file
    // associated with <path> name.
    const void *lookup (const string &path) const {
        // No locking required because we're
        // single-threaded.
        const void *file_pointer = 0;


        // ... look up the file in the cache, mapping it
        // into memory if it is not currently in the cache.
        return file_pointer;
    }
    // ...
private:
    //File cache implementation...

    // No lock required because we are
    // single-threaded.
};
```

```
// A multi-threaded file cache implementation.
class File_Cache_Thread_Mutex {
public:
    // Return a pointer to the memory-mapped file
    // associated with <path> name.
    const void *lookup (const string &path) const {
        // Use the Scoped Locking idiom to serialize
        // access to the file cache.
        Thread_Mutex_Guard guard (lock_);
        const void *file_pointer = 0;

        // ... look up the file in the cache, mapping it
        // into memory if it is not currently in the cache.
        return file_pointer;
    }
    // ...
private:
    //File cache implementation...

    // Synchronization strategy.
    mutable Thread_Mutex lock_;
};
```

这两种类的实现是组件系列的一部分，这些组件仅仅在其同步策略上存在差异。作为组件之一的File_Cache_ST类实现了一个不带锁的单线程文件缓存。另一个组件即类File_Cache_Thread_Mutex实现的文件缓存使用互斥对并发访问缓存的多个线程串行化。但是，同时用不同的方法实现这些文件缓存组件是令人生厌的。特别是，将来需要对所有组件实现进行一致性的 333

修改或改进。

2. 语境

其组件必须在不同的并发体系结构中有效运行的应用程序或服务。

3. 问题

运行在多线程环境中的组件必须保护其临界区不被客户机并发访问。同步机制与组件功能的集成需要解决以下两个**强制条件**：

- 不同的应用程序可能要求不同的同步策略，如互斥、阅读器/记录器锁或信号灯[McK95]。因此，应该可以按照具体应用的需求定制组件的同步机制。

  ➡在这里的例子中，同步策略是硬编码的。因此，为了改善大规模多处理器平台上的性能，必须定义一个使用阅读器/记录器锁而不是线程互斥，并支持文件缓存实现的新类。不过，定制一个已有的文件缓存类使之支持新的、更有效的同步策略是很费时的。 ❑

- 加入新的功能和隐错修正应很容易。特别为避免“版本混乱”，应始终将这些变化一致地自动地应用于组件系列的所有成员上。

  ➡如果存在同一基本文件缓存组件的多个副本，很可能出现版本混乱，因为对一个组件的改变可能会不一致地应用到其他组件实现中。手工地改变也是容易出错的，且不好扩展。❑

4. 解决方案

将组件的同步特性变成“可插”的类型，用这种方式将组件的同步特性参数化。每种类型将特定的同步化策略对象化，同步化策略包括互斥、阅读器/记录器锁、信号灯或“空”锁等。将这些可插的类型的实例定义为包含在组件中的对象，该组件可以使用这些对象有效地使其方法实现同步化。

334

5. 实现

可以用5个活动实现策略化加锁模式。

(1) 不考虑组件的同步特性，定义组件的接口和实现。

➡下面的类定义了File_Cache接口和实现：

```
class File_Cache {
public:
    const void *lookup (const string &path) const;
    // ...
private:
    // data members and private methods go here...
};
```

❑

(2) 将加锁机制策略化。许多组件有相对简单的同步特性，可以使用常用的加锁策略，如互斥和信号灯等实现这些同步特性。可以统一地使用多态性或参数化类型将同步特性策略化。一般来说，如果直到运行时才知道加锁策略，就应该使用多态性。相反，如果在编译时就知道加锁策略，那就应该使用参数化类型。参数化类型有运行效率高的特点，而多态性有运行时可扩展的潜力。

假设使用定界加锁惯用法，策略化锁包括两个子活动：

(2.1) 为加锁机制定义一个抽象接口。为了使组件能使用不同的加锁机制，这些机制的所有具体实现应该使用具有共同特征的抽象接口，用于在多态性或参数化类型的基础上获取和释放锁。

- 多态性。在这种策略中，定义一个包含动态绑定的acquire()和release()方法的多态锁对象。正如实现活动（5）中介绍的，从这个基类派生出所有具体的锁，并重载其方法，定义具体的加锁策略。

335

➡为文件缓存例子实现一个多态锁对象，首先要定义一个有虚方法acquire()和release()的抽象加锁类：

```
class Lock {
public:
    // Acquire and release the lock.
    virtual void acquire () = 0;
    virtual void release () = 0;

    // ...
};
```

❑

- 参数化类型。在这种策略中，必须确保所有的具体锁对于锁的获取和释放具有同样的特性。使用包装器外观模式实现具体锁可以达到这一目的。

(2.2) 使用定界加锁惯用法定义一个用同步特征将其策略化的哨兵类。这种设计遵循策略模式[GHJV95]，其中，哨兵类作为拥有某一特殊锁的语境，而具体锁提供策略。定界加锁惯用法可以用多态性或参数化类型实现。

- 多态性。在这种方法中，将多态锁对象传递给哨兵的构造函数，并定义该锁对象的一个实例作为私有数据成员。要获得锁和释放锁，哨兵类的实现中可以使用在子活动（2.1）中定义的多态Lock基类的接口。

➡控制多态锁的Guard类定义如下：

```
class Guard {
public:
    // Store a pointer to the lock and acquire the lock.
    Guard (Lock &lock)
        : lock_ (&lock), owner_ (false)
        { lock_->acquire (); owner_ = true; }

    // Release the lock when the guard goes out of scope.
    ~Guard () {
        // Only release lock if it <acquire> succeeded.
        if (owner_) lock_->release ();
    }

private:
    // Pointer to the lock we're managing.
    Lock *lock_;
    // Records if the lock was acquired successfully.
    bool owner_;
};
```

336

❑

- 参数化类型。在这种方法中，定义一个模板哨兵类，其模板参数是将被自动获取和释放的锁的类型。

➡下面是用LOCK模板参数策略化的一个Guard类：

```
template <class LOCK>
class Guard {
public:
    // Store a pointer to the lock and acquire the lock.
    Guard (LOCK &lock)
        : lock_ (&lock), owner_ (false)
        { lock_->acquire (); owner_ = true; }

    // Release the lock when the guard goes out of scope,
    // but only if <acquire> succeeded.
    ~Guard () { if (owner_) lock_->release (); }
private:
    // Pointer to the lock we're managing.
    LOCK *lock_;

    // Records if the lock is held by this object.
    bool owner_;

    // ... disallow copying and assignment ...
};
```

❑

(3) *更新组件的接口和实现*。在策略化同步机制后，组件可以使用这些机制，通过显式地获得或释放一个锁，或者使用在实现活动（2.2）中定义的哨兵类来保护临界区。后一种方法遵循定界加锁惯用法。根据采用的是多态性还是参数化策略，可以将锁作为构造函数的参数，或者在组件声明中增加一个锁模板参数而将锁传给组件。正如实现活动（2.2）中讨论的那样，在这两种情况下，传给组件的锁必须具有哨兵类所期望的特征。

337

➡向这一版本的文件缓存组件传递一个多态锁参数：

```
class File_Cache {
public:
    // Constructor.
    File_Cache (Lock &l): lock_ (&l) { }

    // A method.
    const void *lookup (const string &path) const {
        // Use the Scoped Locking idiom to acquire
        // and release the <lock_> automatically.
        Guard guard (*lock_);
        // Implement the <lookup> method.
    }

    // ...
private:
    // The polymorphic strategized locking object.
    mutable Lock *lock_;
    // Other data members and methods go here...
};
```

同样，也可以定义文件缓存的一个模板化的版本：

```
template <class LOCK>
class File_Cache {
public:
    // A method.
    const void *lookup (const string &path) const {
        // Use the Scoped Locking idiom to acquire
        // and release the <lock_> automatically.
        Guard<LOCK> guard (lock_);
        // Implement the <lookup> method.
```

```
    }

    // ...
private:
    // The parameterized type strategized locking object.
    mutable LOCK lock_;

    // Other data members and methods go here...
};
```

如果C++编译器支持缺省模板参数，可以为句柄增加带缺省值的LOCK，该缺省值满足绝大多数用况。例如，可以将定义缺省的LOCK为一个阅读器/记录器锁：

```
template <class LOCK = RW_Lock>
class File_Cache { /* ... */ }
```

❑ 338

(4) *修改组件实现以避免死锁和删除不必要的加锁开销*。如果组件间存在方法调用，那么开发人员必须仔细地设计其组件实现，以避免自死锁和不必要的同步开销。线程安全接口模式提供了一种简单的技术用于预防这些问题。

(5) *定义一组具有统一接口的加锁策略*，这些策略可以支持各种应用特定的并发设计。常用的加锁策略包括递归与非递归互斥、阅读器/记录器锁、信号灯和文件锁 。

使用多态性方法时，和实现活动(2)中介绍的一样，在抽象类Lock的子类中实现加锁策略。如果使用参数化类型，要确保具体锁的实现满足实现活动2中定义的锁的特征。

➡除了在包装器外观中定义的Thread_Mutex加锁策略之外，Null_Mutex也是很有用的。该类为单线程应用程序和组件定义了一个有效的加锁策略：

```
class Null_Mutex {
public:
    Null_Mutex () { }
    ~Null_Mutex () { }
    void acquire () { }
    void release () { }
};
```

Null_Mutex中的所有方法都是空的C++内联函数，可以被具有优化功能的编译器彻底删除。这个类是空对象（Null Object）模式[PLoPD3]的一个例子，它定义了一个能在组件的实现中删除条件语句的“no-op”占位符，从而能简化应用程序。“*已解决的例子*”一节中介绍了一些应用Null_Mutex和其他加锁策略的例子。 ❑

如果已有的加锁机制的接口不兼容，可以使用包装器外观或者适配器模式[GoF95]，以确保接口与组件的同步特性一致。

339

➡下面的类包装了包装器外观模式的“*实现*”一节中的Thread_Mutex，从而使之成为了这里的多态加锁层次结构的一部分：

```
class Thread_Mutex_Lock : public Lock {
public:
    // Acquire and release the lock.
    virtual void acquire () { lock_.acquire (); }
    virtual void release () { lock_.release (); }
private:
```

```
    // Concrete lock type.
    Thread_Mutex lock_;
};
```

❑

6. 已解决的例子

可以使用策略化加锁模式的参数化类型的形式实现用于保存Web服务器内容的文件缓存，该文件缓存可以用于不同的单线程和多线程并发模型中：

- 单线程文件缓存。

```
typedef File_Cache<Null_Mutex> Content_Cache;
```

- 使用线程互斥的多线程文件缓存。

```
typedef File_Cache<Thread_Mutex> Content_Cache;
```

- 使用阅读器/记录器锁的多线程文件缓存。

```
typedef File_Cache<RW_Lock> Content_Cache;
```

- 使用支持缺省模板参数的C++编译器的多线程文件缓存，如实现活动(3)中所介绍的一样，锁的缺省类型是阅读器/记录器锁。

```
typedef File_Cache<> Content_Cache;
```

注意，针对于不同的配置，Content_Cache接口和实现都不要求改变。这种灵活性来自于策略化加锁模式，它将同步特性抽象为“可插的”参数化类型。而且，通过C++的typedef将加锁的细节策略化，因此只需很简单地定义一个不将同步特性展示给应用程序的Content_Cache对象：

340

```
Content_Cache cache;
```

7. 变体

桥策略。令人遗憾的是，实现活动（3）中的配置多态性文件缓存与配置模板化的文件缓存有差别，因为用指针实现的多态锁不能作为参数传递给模板类File_Cache和Guard。相反，这里需要一个“真实的”对象，而不是指向对象的指针。幸好，桥（Bridge）模式［GoF95］可以实现一系列同时适用于多态性和参数化类型方法的加锁策略。只需定义一个封装了多态锁，并用多态锁进行配置的抽象类就可以使用桥策略。这样可以统一地向多态性组件和模板化组件传递该抽象类的一个实例。

➡实现活动(2)中定义的多态锁层次可以认为是一个桥实现类层次。下面的抽象类可用于封装该层次结构：

```
class Lock_Abstraction {
public:
    // Constructor stores a reference to the base class.
    Lock_Abstraction (Lock &l): lock_ (l) { };

    // Acquire the lock by forwarding to the
    // polymorphic acquire() method.
    void acquire () { lock_.acquire (); }

    // Release the lock by forwarding to the
```

```
    // polymorphic release() method.
    void release () { lock_.release (); }
private:
    // Maintain a reference to the polymorphic lock.
    Lock &lock_;
};
```

请留意，这种设计是如何使程序员用一个Lock_Abstraction类来初始化多态的和参数化的File_Cache和Guard类的。可以用加锁机制层次结构中的一个具体锁来配置要用于初始化的Lock_Abstration类。 ❑

使用这种改进的策略化加锁方法的一个结果是，加锁机制系列更易被重用，更易被跨应用程序应用。不过要注意，这种图式灵活性很好但难以实现，因此使用是要很小心。 341

8. 已知使用

ACE[Sch97]。ACE框架中广泛使用了策略化加锁模式。大多数ACE容器组件的同步特性，如ACE_Hash_Map_Manager都可以通过参数化类型而被策略化。

Booch组件。Booch组件[BV93]是最早的用模板来参数化加锁策略的C++类库之一。

Dynix/PTX操作系统在它的内核中广泛使用了策略化加锁模式。

ATL Wizard。Visual Studio中的Microsoft ATL Wizard，使用了策略化加锁的参数化类型实现，它带有缺省模板参数。另外，它还实现了一个类似于Null_Mutex的类。如果一个COM类实现为单线程的容器（apartment），那么就要用到no-op锁类，多线程容器中要使用“真实的”递归互斥。

9. 结论

使用策略化加锁模式有以下三点**优点**：

增加灵活性和个性化。为特定的并发模型配置和定制一个组件是很容易的，因为组件的同步特性被策略化了。如果对于一种新的并发模型没有合适的加锁策略可用，可以在不影响已有代码的情况下扩充新的加锁策略。

降低组件维护的代价。使用策略化加锁模式很容易改进组件和隐错修正，因为对于各种并发模型只有一个实现，而不是每种并发模型对应一个独立的实现。将问题集中的方法有助于减少版本混乱。

改善重用性。用这种模式实现的组件不太依赖于具体的同步机制。这样组件更具有可重用性，因为可以与其特性正交地配置加锁机制。

策略化加锁模式有以下两点**不足**：

强行（obtrusive）加锁。如果使用模板来参数化加锁特性，将在应用程序代码中直接体现加锁策略。尽管这种方法很灵活，但也是强行的，特别是对于不能有效地和正确地支持模板的编译器来说。避免这种问题的方法是使用多态性策略来使组件加锁特性多样化。 342

过分工程化（over-engineering）。通过将加锁机制置于组件的接口中来扩展加锁机制在有些情况下可能提供过多的灵活性。例如，经验不足的开发人员可能用一个错误的锁类型来参数化一个组件，从而产生不正确的编译时或运行时行为。同样，对于某一组件类型可能只需要一种类型的同步机制，在这种情况下，是不需要策略化加锁的灵活性的。总的来说，如果实践经验

表明一个组件的行为与其加锁策略正交，且加锁策略确实在语义含义和效率方面有很大的不同时，这种模式才最有效。

参见

Java中主要的同步机制是监视器（monitor）。Java语言并不为应用开发者提供“常规的”并发控制机制，如互斥和信号灯。因此Java中不需要直接使用策略化加锁模式。

不过有可能用Java实现不同的并发原语，如互斥、信号灯和阅读器/记录器锁。例如，[Lea99a]中的util.concurrent包定义了多种不同类型的锁，如阅读器/记录器锁。可以将这些原语的实现作为加锁策略使用，以支持各种与应用相关的并发使用的情况。因为到目前为止，Java规范中没有参数化类型，所以只能使用策略化加锁模式的多态性方法来配置不同的同步策略。在这种情况下，这种模式的Java实现类似于前面介绍的C++版本。

致谢

感谢Brad Appleton关于本模式的意见，以及Prashant Jain，他解释了如何将这种模式用于Java中的部分。

343

4.3 线程安全接口

线程安全接口（Thread-Safe Interface）模式将加锁开销减至最小，并能保证组件间的方法调用不会因为想再次获得一个已经被组件拥有的锁而导致“自死锁”。

1. 例子

在设计彼此间有方法调用的线程安全的组件时，开发人员必须小心谨慎，以避免自死锁和不必要的加锁开销。例如，考虑在策略化加锁模式中介绍的File_Cache组件的一个更完整的实现：

```
template <class LOCK>
class File_Cache {
public:
    // Return a pointer to the memory-mapped file
    // associated with <path> name, adding
    // it to the cache if it doesn't exist.
    const void *lookup (const string &path) const {
        // Use the Scoped Locking idiom to acquire
        // and release the <lock_> automatically.
        Guard<LOCK> guard (lock_);
        const void *file_pointer = check_cache (path);
        if (file_pointer == 0) {
            // Insert the <path> name into the cache.
            // Note the intra-class <insert> call.
            insert (path);
            file_pointer = check_cache (path);
        }
        return file_pointer;
    }
    // Add <path> name to the cache.
    void insert (const string &path) {
```

```
        // Use the Scoped Locking idiom to acquire
        // and release the <lock_> automatically.
        Guard<LOCK> guard (lock_);
        // ... insert <path> into the cache...
    }
private:
    mutable LOCK lock_;
    const void *check_cache (const string &) const;
    // ... other private methods and data omitted...
};
```

345

只有当用如策略化加锁模式中介绍的Null-Mutex这样的“空”锁来策略化时，File_Cache的这一实现才能有效地工作。然而，当用递归互斥策略化File_Cache实现时，如果insert()方法中要求再次拥有互斥，File_Cache会产生不必要的开销。更糟糕的是，如果用一个非递归互斥进行策略化，lookup()方法调用insert()方法时代码会出现“自死锁”，其原因是insert()试图获得已经由lookup()获得了的Lock。

可见将策略化加锁模式应用于上述File_Cache的实现中会有副作用，因为会出现许多的限制和一些微妙的问题。不过，File_Cache抽象仍可以受惠于策略化加锁所提供的灵活性和个性化。

2. 语境

在多线程应用程序中包含组件间方法调用的组件。

3. 问题

多线程组件通常包括多个可被公共访问的接口方法以及可以改变组件状态的私有实现方法。为了避免出现竞争条件，可以使用一个组件内部的锁对访问其状态的接口方法调用串行化。尽管当每个方法都自包容的时候，这种设计方法能很好地工作，但组件方法可能会相互调用以完成计算任务。如果是这样，在多线程组件中有以下几个强制条件尚未解决，它们使用了错误的组件间方法调用的设计方法：

- 应该使线程安全组件避免“自死锁”。当一个组件方法在该组件中获得一个非递归锁，然后调用试图获得同一锁的另一个组件方法时，会发生自死锁。
- 为了避免出现对组件状态的竞争条件等情况，应该使线程安全组件具有最小的加锁开销。如果为避免上面介绍的自死锁问题而选择一个递归组件锁，由于组件间方法调用多次获得和释放锁，而会产生不必要的开销。

346

4. 解决方案

按照以下两个约定将处理组件间方法调用的所有组件结构化：

- *接口方法检查*。所有接口方法（如C++公有方法）应该只获得/释放组件锁，在组件的“边缘”进行同步检查。获得一个锁后，接口方法立即转发给一个实现方法，后者执行实际的方法功能。实现方法返回后，接口在将控制返回给调用者之前释放该锁。
- *实现方法信任*。只有被接口方法调用，实现方法（如C++的私有或保护方法）才能完成它们的任务。因此它们相信只有在获得锁后才会调用它们，它们本身不应该获得或释放锁。实现方法也不应该“向上”调用接口方法，因为接口要获得锁。

5. 实现

可以使用以下两个活动实现线程安全接口模式：

(1) *确定接口和相应的实现方法*。接口方法定义组件的公共API。为每个接口方法对应地定义一个实现方法。

➡File_Cache的接口和实现方法定义如下：

```
template <class LOCK>
class File_Cache {
public:
    // The following two interface methods just
    // acquire/release the <LOCK> and forward to
    // their corresponding implementation methods.
    const void *lookup (const string &path) const;
    void insert (const string &path);
private:
    // The following two implementation methods do not
    // acquire/release the <LOCK> and perform the actual
    // work associated with managing the <File_Cache>.
    const void *lookup_i (const string &path) const;
    void insert_i (const string &path);
    // ... Other implementation methods omitted ...
};
```

347

❑

(2) *对接口和实现方法进行编程*。按照“解决方案”一节中的约定，对接口和实现方法的主体进行编程。

➡对File_Cache的实现，使用线程安全接口来使加锁开销减至最少，并防止类方法中的自死锁：

```
template <class LOCK>
class File_Cache {
public:
    // Return a pointer to the memory-mapped file
    // associated with <path> name, adding it to
    // the cache if it doesn't exist.
    const void *lookup (const string &path) const {
        // Use the Scoped Locking idiom to acquire
        // and release the <lock_> automatically.
        Guard<LOCK> guard (lock_);
        return lookup_i (path);
    }

    // Add <path> name to the file cache.
    void insert (const string &path) {
        // Use the Scoped Locking idiom to acquire
        // and release the <lock_> automatically.
        Guard<LOCK> guard (lock_);
        insert_i (path);
    }
private:
    mutable LOCK lock_; // The strategized locking object

    // The following implementation methods do not
    // acquire or release <lock_> and perform their
    // work without calling any interface methods.
    const void *lookup_i (const string &path) const {
        const void *file_pointer = check_cache_i (path);
        if (file_pointer == 0) {
            // If <path> name isn't in the cache then
            // insert it and look it up again.
```

```
            insert_i (path);
            file_pointer = check_cache_i (path);
            // The calls to implementation methods
            // <insert_i> and <check_cache_i> assume
            // that the lock is held and perform work.
        }
        return file_pointer;
    }

    const void *check_cache_i (const string &) const
        { /* */ }

    void insert_i (const string &) { /* ... */ }

    // ... other private methods and data omitted ...
};
```

348

❑

6. 变体

线程安全的外观。在必须串行化对整个子系统或者粗粒度组件的访问时，使用这种变体。可以引入一个外观[GoF95]作为所有客户机请求的入口点。外观的方法对应于接口方法。属于子系统或组件的类提供实现方法。如果这些类有自己的内部并发机制，可能需要进行再分解以避免嵌套的监视器阻塞（nested monitor lockout）[⊖] [JS97a]。

当一个线程获取对象X的监视器锁，而没有释放已经获得的在监视器Y上的锁时，会阻止另一个线程获得Y的监视器锁，也就发生了嵌套的监视器加锁（lockup）。这会导致死锁，因为获得监视器锁X后，第一个线程可能等待某个条件变成真，而只有当另一个进程获得监视器Y后这个条件才能得到改变。在不能修改子系统或组件的情况下（如它们是第三方产品或遗产系统时），对代码进行正确的重构（refactor）以避免嵌套监视器阻塞是不可能的。此时，不应使用线程安全外观。

线程安全的包装器外观。这种变体可用于对不可修改的非同步类或函数API的访问的同步化。包装器外观提供了接口方法，它们封装了对应的类或函数API的实现调用，这些类或函数API用于获得或释放锁。因此，包装器外观提供了一个同步代理[POSA1][GoF95]，该代理串行化对该类中的方法或函数API的访问。

7. 已知使用

ACE[Sch97]。在ACE框架中使用了线程安全接口模式，如在ACE_Message_Queue类中就用到了。

349

Dynix/PTX操作系统在其内核中使用了线程安全接口模式。

Java。在java.util.Hashtable中实现的散列表使用了线程安全接口设计模式。Hashtable的接口方法，如put(Object key, Object value)，在修改由一个链表数组组成的底层数据结构之前要获得一个锁。当装填因子超过某一阈值时，调用rehash()实现方法，并产生一个更大的散列表，将旧表中的元素全部转移到新表中，并由垃圾收集器将旧表回收。注意和可公共访问的方法（如put(Object key, Object value)）不同，rehash()方法并不受锁的保护。由于有Java的重入监视器，锁对rehash()的保护不会死锁程序，但会因为加锁开销而降低其性能。

⊖ 参见监视器对象模式的“结论”一节，那里有关于嵌套的监视器封锁问题的详细讨论。

在JDK1.2的Collection类中有一个更成熟的用况，其中使用线程安全的包装器外观变体使集合数据结构是线程安全的。Java.util.Collections将所有的类实现放在一个Map接口中，并返回另一个实现Map的类——SynchronizedMap。SynchronizedMap中的方法仅仅与内部监视器同步，并调用原先对象的方法。因此开发者可以在数据结构的快速和线程安全两个变体中进行选择，而不需要两次实现。

安全性检查点。当一个人进入一个国家或一个商务大楼时，会在边境或入口处遇到一个安全卫兵的检查，这就是线程安全接口模式现实生活中的一个例子。要想进入，他必须先签名。被允许进入后，与他打交道的其他人都会假设他是那儿的人。

8. 结论

应用线程安全接口模式有三个**优点**：

提高了健壮性：由于使用组件间方法调用，该模式可以避免自死锁。

350

改善了性能：该模式确保在必要时才获得或释放锁。

简化了软件：将加锁和功能特性问题分开，可以使加锁和功能特性得到简化。

不过，应用线程安全接口模式有四点**不足**：

增加间接的和额外的方法。每个接口方法至少要有一个实现方法，因此增加了组件在内存中的占用空间，也可能为每个调用增加了方法调用的层次。减少这一开销的一个方法是内联接口和/或实现方法。

潜在的死锁。线程安全接口模式本身没有完全解决自死锁问题。例如，一个客户机实际调用组件A的一个接口方法，该方法又委派给调用另一个组件B上接口方法的实现方法。如果组件B的方法回调组件A的接口方法，当试图获得一个锁，而该锁已经被该调用链中另一次调用获得时，就会出现死锁。

潜在的误用。像C++和Java这样的面向对象程序设计语言都支持类级而不是对象级的访问控制。因此一个对象可以不经过公共接口而调用同一类中另一对象的私有方法，从而也就没有经过那个对象的锁。因此程序员应该小心，避免调用其他对象的私有方法。

潜在的开销。线程安全接口模式能避免多个组件共享同一锁。这样会增加同步开销，也难以检测和避免死锁，因为必须获得多个锁。而且该模式不能在比组件更小的粒度上进行加锁，这增加了对锁的争用，从而降低了性能。

参见

线程安全接口模式与装饰器（Decorator）模式［GoF95］有关，后者通过动态地增加额外的责任而透明地扩展一个对象。线程安全接口模式的目标类似，它增加了健壮的、有效的加锁策略以使组件是线性安全的。二者的主要差别是装饰器模式动态地将责任加到对象上，而线程安全接口模式在组件类中静态地划分方法责任。

351

按照策略化加锁模式设计的组件应该使用线程安全接口模式以确保组件的健壮性和高效性，而不管选择何种加锁策略。

Java在方法层上通过由同步关键字表示的监视器对象实现加锁。Java中监视器是递归的。只要开发者重用同一监视器，也就是在同一对象上同步，死锁的问题就不会发生。不过，如果不

加考虑地使用多嵌套监视器，Java中可能发生嵌套监视器阻塞[JS97a][Lea99a]的问题。

加锁开销的问题依赖于使用何种Java虚拟机（JVM）。如果某一JVM低效地实现监视器，且监视器是递归获得的，那么线程安全接口模式可以改善组件运行时的性能。

致谢

感谢Brad Appleton有关本模式的意见。Prashant Jain写成了线程安全接口模式的变体，以及对Java嵌套监视器阻塞的讨论。

352

4.4 双检查加锁优化

当代码的临界区必须在程序执行期内以线程安全的方式获得一次锁时，双检查加锁优化（double-checked locking optimization）设计模式能减少争用和同步开销。

1. 别名 加锁线索（Lock Hint）。[Bir91]

2. 例子

单件模式能确保类只有一个实例，并为该实例提供一个全局的访问点。下面的C++代码是[GoF95]中的一个单件的典型实现:

```
class Singleton {
public:
    static Singleton *instance () {
        if (instance_ == 0) {
            // Enter critical section.
            instance_ = new Singleton ();
            // Leave critical section.
        }
        return instance_;
    }
    void method_1 (); // Other methods omitted.
private:
    static Singleton *instance_;
    // Initialized to 0 by the compiler/linker.
};
```

应用程序使用静态instance()方法获得一个指向单件的指针，然后调用公有方法:

```
Singleton::instance ()->method_1 ();
```

不幸的是，上述单件模式的典型实现在具有抢先多任务或真正的硬件并行性的平台上是有问题的。特别是在下列情况下Singleton的构造函数可能被多次调用:

- 多个抢先线程在初始化之前同时调用Singleton::instance()。
- 多个线程在临界区内执行Singleton构造函数进行动态初始化。

353

最好的情况是多次调用Singleton构造函数会导致内存泄露。最坏的情况是当Singleton的初始化非等幂时会引起灾难性后果。

为了使临界区不被并发访问，可以使用定界加锁惯用法自动地获得和释放一个互斥锁:

```
class Singleton {
public:
    static Singleton *instance () {
        // Scoped Locking acquires <singleton_lock_>.
        Guard<Thread_Mutex> guard (singleton_lock_);
        if (instance_ == 0)
            instance_ = new Singleton;
        return instance_;
        // Destructor releases lock automatically.
    }
private:
    static Singleton *instance_;
    static Thread_Mutex singleton_lock_;
};
```

在singleton_lock_被正确初始化的情况下，Singleton是线程安全的。不过加锁开销会很大。特别是每次对instance()的调用都要获得和释放锁，即使临界区只被执行一次。可以将哨兵放在条件检查里边，从而避免加锁开销：

```
static Singleton *instance () {
    if (instance_ == 0) {
        Guard<Thread_Mutex> guard (singleton_lock_);
        // Only come here if <instance_>
        // has not been initialized yet.
        instance_ = new Singleton;
    }
    return instance_;
}
```

可是，这种解决方案不能提供线程安全的初始化，因为在多线程应用中竞争条件可以使Singleton多次被初始化。例如，考虑两个同时检查instance_==0的线程，假设两个都成功，一个通过guard获得锁然后释放锁，而另一个将阻塞。在第一个线程初始化Singleton并释放锁之后，那个被阻塞的线程会得到该锁，并错误地再次初始化Singleton。

354

3. 语境

包含有被多个线程并发访问的共享资源的应用程序。

4. 问题

并发应用程序必须确保当访问或修改线程的共享资源时，某些代码必须顺序地执行，以避免竞争条件。避免竞争条件的一个常用的办法是通过锁(如互斥)来串行化对共享资源的临界区的访问。要进入临界区的每一个线程，必须首先获得一个锁。如果该锁已经被其他线程拥有，那么该线程将阻塞，直到该锁被其他线程释放并被本线程获得。

上述串行化方法对于只要求执行一次初始化的对象或组件是不合适的，例如，Singleton例子中的临界区代码在其初始化阶段必须执行一次。但是对Singleton的每个方法调用都要获取和释放互斥锁，从而过多地增加了开销[PLoPD3]。为避免这些开销，并发应用的程序员可能转而使用全局变量而不是单件模式。不过这种解决方案有两个缺点[LGS99]：

- 可移植性不好。因为通常没有指定在不同文件中定义的全局对象被构造的顺序。
- 资源的浪费。因为即使不使用全局变量，也要产生全局变量。

5. 解决方案

引入一个标志，表示在获得一个保护临界区的锁之前确定是否需要执行该临界区。如果不需要执行该代码，就跳过该临界区，从而避免了不必要的加锁开销。下面给出这段代码的一段伪码设计:

```
// Perform first-check to evaluate 'hint'.
if (first_time_in_flag is FALSE) {
    acquire the mutex
    // Perform double-check to avoid race condition.
    if (first_time_in_flag is FALSE) {
        execute the critical section
        set first_time_in_flag to TRUE
    }
    release the mutex
}
```

355

6. 实现

可以用以下三个活动实现双检查加锁优化模式:

(1) 确定要仅执行一次的临界区。该临界区进行的操作（如初始化逻辑）在程序中仅执行一次。

➡例如，Singleton在程序中只初始化一次。在临界区中对Singleton构造函数的调用也只执行一次，而不管Singleton:: instance()被调用多少次。 ❑

(2) 实现加锁逻辑。加锁逻辑对只执行一次的临界区代码的访问串行化。可以使用定界加锁方法实现这种加锁逻辑，以确保当进入某一区域时自动获得锁，而当离开该区域时自动释放该锁。

➡按照定界加锁惯用法，要使用Thread_Mutex Singleton_Lock_以确保单件的构造函数不会并发执行。 ❑

必须在第一次调用只执行一次的代码之前初始化该锁。在C++中，一种确保在第一次使用之前初始化锁的方法就是将锁定义成静态对象，如“例子”一节所示。不过，C++语言规范并不确保定义在不同编译单元中的静态对象的初始化顺序。因此，不同的C++编译器和连接程序的行为可能不一致，当第一次访问锁时它不一定被初始化过。

避免这一问题的一个更好的办法是使用对象生命期管理者模式[LGS99]。该模式定义一个可移植对象管理者组件，它以下列方式控制全局或局部对象的整个生命期:

- 对象管理者先创建这些对象，然后才能使用这些对象。
- 确保在程序结束时这些对象能被正确的销毁。

例如，可以将锁置于一个对象管理者的控制之下，使之在任何单件试图使用该锁来串行化该单件的初始化之前被初始化，对象管理者也可以在程序结束时删除单件，从而避免在单件模式中出现的内存和资源泄漏[Vlis98a]。 356

(3) 实现首次进入(first-time-in)标志。该标志表示临界区是否被执行过。

➡使用Singleton::instance_指针作为首次进入标志。如果该标志为真，则跳过临界区。如果和这里的Singleton::instance_一样，该标志还有与应用有关的特殊用途，那么

它必须是原子类型，即不能进行部分读/写。下面的Singleton例子代码是线程安全的，而且由于将对new的调用放在另一个条件测试之内，也避免了不必要的加锁开销。

```
class Singleton {
public:
    static Singleton *instance () {
        // First check
        if (instance_ == 0) {
            // Use Scoped Locking to acquire and
            // release <singleton_lock_> automatically.
            Guard<Thread_Mutex> guard (singleton_lock_);
            // Double check.
            if (instance_ == 0)
                instance_ = new Singleton;
        }
        return instance_;
    }
private:
    static Singleton *instance_;
    static Thread_Mutex singleton_lock_;
};
```

获得Singleton_Lock_的第一个线程创建Singleton对象，并且将对象指针赋给instance_，在例子中instance()是“首次进入”的标志。以后调用的所有线程会发现instance_不为0，就跳过初始化步骤。

第二个检查可以避免当多个线程试图同时初始化时产生Singleton竞争条件。它解决了多个线程并行执行的问题。在以上的代码中所有线程排队等待Singleton_lock_互斥。当排队线程获得Singleton_lock_后，它们会发现instance_非0，从而跳过Singleton初始化。

357

对于这种Singleton::instance()方法实现，只有那些第一次初始化Singleton时，在instance()内部活动的线程会产生加锁开销。在以后对instance()的调用中，instance_指针非零，所以Singleton_lock_不会被获得或释放。 ❑

7. 变体

易失（volatile）数据。如果编译器通过以某种方式存储“首次进入”标志，如将其存储在CPU寄存器中，而优化“首次进入”标志，在这种情况下就存在缓存一致性（cache coherency）的问题。例如，如果一个线程对值的改变没有反映到其他线程的副本中时，放置在CPU中的由多个线程同时拥有的首次进入标志的副本就会变得不一致。这种情况下需要修改双检查加锁优化模式的实现。

与此相关的一个问题是，一个高度优化的编译器可能会以为第二个对flag==0的检查是多余的，而将它优化掉。解决这两个问题的办法之一是将该标志声明为volatile数据，以确保编译器不会过分地优化而改变程序的语义。

➡对于Singleton例子，相应的代码是：

```
class Singleton {
    // ...
private:
    static Singleton *volatile instance_;
    // instance_ is volatile.
};
```

使用volatile可以使编译器不会将instance_指针放入寄存器中，也不会优化掉instance()

中的第二个检查。 ❑

使用volatile带来的问题是所有的访问都通过内存而不是通过寄存器，这样会降低性能。

模板适配器。当在C++实现双检查加锁优化模式时，可以使用另一个变种。其中创建一个模板适配器将类进行转换，使类具有类单件的特性，并能自动地执行双检查加锁优化模式。 358

➡下面的代码说明如何写C++中的这种模板：

```
template <class TYPE>
class Singleton {
public:
    static TYPE *instance () {
        // First check
        if (instance_ == 0) {
            // Scoped Locking acquires and release lock.
            Guard<Thread_Mutex> guard (singleton_lock_);
            // Double check instance_.
            if (instance_ == 0)
                instance_ = new TYPE;
        }
        return instance_;
    }
private:
    static TYPE *instance_;
    static Thread_Mutex singleton_lock_;
};
```

用作为一个单件被访问的TYPE来对Singleton模板参数化。在instance()方法中对singleton_lock_自动采用双检查加锁优化模式。

Singleton模板适配器也可以和对象生命期管理者模式[LGS99]集成，后者能确保当应用进程退出时，动态分配的单件会被自动释放。这种模式也能保证静态singleton_lock_在首次使用之前能正确地初始化。 ❑

预初始化单件。该变种在很多情况下可以代替双检查加锁优化。它在程序启动阶段（如程序的main()函数中）显式地初始化所有对象。这样就不会出现竞争条件，因为初始化被限制在单线程内进行。

不过，如果必须进行一些耗时的计算，而在有些情况下这些耗时的计算是非必需的，这种方法就不合适了。例如，如果在程序执行期间并不会实际地创建一个单件，那么在程序启动期间，对它的初始化只会浪费资源。使带有单件的应用程序组件暴露单件的信息使之能显式地初始化，这又破坏了封装性。同样，使用预初始化使得用由组件配置器模式动态配置的组件组成应用程序很困难。 359

8. 已知使用

ACE[Sch97]。在ACE框架中广泛地使用了双检查加锁优化模式。为了减少代码重复，ACE定义了一个称为ACE_Singleton的可重用适配器模板，它类似于“变体”一节中介绍的模板，用于将“正常”的类转换成单件。尽管单件并不是ACE框架中双检查加锁优化模式的惟一用法，但它是说明该模式用途的一个普通例子。

Sequent Dynix/PTX。在Sequent Dynix/PTX操作系统中也使用了双检查加锁优化模式。

POSIX和Linux。双检查加锁优化模式可以用于实现POSIX的“once”变量[IEEE96]。这种变量能保证在程序中函数仅被调用一次。该模式已用于LinuxThreads的pthread_once()中，

以确保函数指针参数init_routine()只被调用一次，即第一次调用后不会再被调用。

Andrew Birrell在[Bir91]中介绍了双检查加锁优化模式的用法。Birrell将对标志的第一次检查称为加锁线索（lock hint）。

在线程特定的存储器模式的“已知使用”一节给出了关于phtread_key_create(3T)的Solaris 2.x文档，该文档说明了如何使用双检查加锁优化来初始化线程特定的数据。

9. 结论

使用双检查加锁优化模式有两个**优点**：

使加锁开销最小。通过进行两次“首次进入”标志检查，在一般情况下，双检查加锁优化模式减少了开销。在设置标志后，第一次检查确保后续的访问不再需要加锁。

360

防止竞争条件。对“首次进入”标志进行第二次检查将确保只执行一次临界区。

不过，如果将双检查加锁优化模式用于某些类型的操作系统硬件或编译连接器平台上，会有以下三个**不足**。由于该模式适用于大部分平台，所以下面列出了克服这些局限的技术：

非原子指针或集成赋值语义。如果在单件实现中用instance_指针做实现标志，在一个单件操作中必须完整地读或写该单件的instance_指针的所有位。如果调用new之后的写内存操作不是原子的，其他线程可能会读到一个无效的指针。这样可能导致偶尔发生的非法内存访问。

在某些系统中这种场景是可能发生的，这些系统的内存地址跨越字定位边界，如将32位指针用于16位总线的计算机上，每次指针访问需要两次访问内存。在这种情况下，需要使用一个独立的、字对齐的集成标志——假设硬件支持原子的、基于字的读写——而不是使用一个instance_指针。

多处理器缓存的连贯性。一些多处理器平台（如COMPAQ Alpha和Intel Itanium）要进行积极的内存缓存优化，其中，可以在多CPU缓存间“无次序”地执行读写操作。在这些平台上，不可能不加修改地使用双检查加锁优化模式，因为如果在没有获得锁的情况下访问共享数据，CPU缓存线不会被正确地清除。

要在这一类硬件平台上正确地使用双检查加锁优化模式，必须在实现中插入与CPU有关的指令，如清除缓存线的内存障栏。注意使用双检查加锁优化模式的模板适配器变种有一个意外的副作用——它将与CPU有关的缓存指令集中放在了一起。

361

➡例如，可以在Singleton模板适配器类的instance()方法中放入内存障栏指令：

```
template <class TYPE>
TYPE *Singleton<TYPE>::instance () {
    TYPE *tmp = instance_;

#if defined (ALPHA_MP)
    // Insert the CPU-specific memory barrier instruction
    // to synchronize the cache lines on multi-processor.
    asm ("mb");
#endif /* ALPHA_MP */

    // First check
    if (tmp == 0) {
        // Scoped Locking acquires and releases lock.
```

```
            Guard<Thread_Mutex> guard (singleton_lock_);

            // Double check.
            if (tmp == 0) {
                tmp = new TYPE;

#if defined (ALPHA_MP)
                // Insert a second CPU-specific memory
                // barrier instruction.
                asm ("mb");
#endif /* ALPHA_MP */

                instance_ = tmp;
            }
        }
        return tmp;
    }
```

只要一致地使用Singleton模板适配器，就可以简单地定位与CPU有关的代码，而不影响应用程序。相反，如果在每个要使用的地方手工将双检查加锁优化模式插入到单件中，增加这个CPU特定的代码要付出更多的努力。❑

不过，双检查加锁优化模式的实现中对与CPU相关的代码的需求使得这一模式不适于Java应用。Java的字节代码是跨平台的，因此Java虚拟机缺少解决上述问题的内存障栏指令。

额外的互斥使用。不管是否按需分配了一个单件，在程序的生命期中总要分配和获得某些类型的锁，如例子中的Thread_Mutex。减少这一开销的技术是在一个对象管理器[LGS99]中预先分配一个锁，并使用该锁来串行化所有的对单件的初始化。虽然这可能增加对锁的争用，但不会影响程序的性能，因为每个单件被初始化后都只能获得和释放锁。 362

参见

双检查加锁优化模式是惰性评价（Lazy Evaluation）模式[Mey98][Beck97]的线程安全的变体。通常在像C语言这样的缺少构造函数的程序设计语言中使用，以确保先初始化组件后才能访问组件的状态。

➡例如下面的C代码初始化了一个栈：

```
static const int STACK_SIZE = 1000;
static Type *stack_;
static int top_;

void push (Type *item) {
    // First-time-in-check.
    if (stack_ == 0) {
        // Allocate the pointer, which implicitly
        // indicates that initialization was performed.
        stack_ = malloc (STACK_SIZE * sizeof Type);
        top_ = 0;
    }
    stack_[top_++] = item;
    // ...
}
```

第一次调用push()时，stack_为0，从而触发通过malloc()进行的隐含的初始化过程。❑

致谢

本模式的最初版本[PLoPD3]的合作者是Tim Harrison。感谢John Basrai、James Coplien、Ralph Johnson、Jaco van der Merwe、Duane Murphy、Paul McKenney、Bill Pugh以及Siva Vaddepuri对双检查加锁优化模式的建议和评述。

363

第5章
并发模式

“你目标不明，现状不清，茫然失措时，
不妨回顾曾走过的路，一个模式已隐约显现出来。
根据该模式推进，总会发现一些有价值的东西。”
Robert M.Pirsig

本章介绍了五种模式，用以解决组件、子系统和应用程序中各种类型的并发体系结构和设计问题：主动对象、监视器对象、半同步/半异步、领导者/追随者和线程特定的存储器。 365

并发体系结构的选择对多线程连网中间件和应用的设计及性能产生了巨大的影响。一种并发体系结构并不能适合所有的工作情况及软、硬件平台。因此，本章的模式为各种并发问题提供了总的解决方法。

本章首先定义了为在多线程或进程间共享资源而设计的2种模式：

- 主动对象设计模式将方法执行和方法调用分离开来。目的是加强并发和简化对驻留在自身控制线程中的对象的同步访问。
- 监视器对象设计模式同步化并发方法的执行，以确保同一时刻在对象内部只有一个方法运行。它也允许对象的方法协作调度方法的执行顺序。

两种模式都可以对并发调用对象的方法进行同步和调度。主要差别是主动对象是在不同于客户机线程的另一线程中执行它的方法，而监视器对象借用客户机的线程执行其方法。因此，主动对象虽然开销大，但可以进行更复杂的任务调度，确定自身方法的执行顺序。

下面两种模式定义了高层并发体系结构：

- 半同步/半异步体系结构模式将并发系统中的同步和异步进程分离，简化了编程而又不影响性能。该模式引入两个通信层，一个处理同步服务，另一个处理异步服务。还有一个排队层协调异步层和同步层服务之间的通信。
- 领导者/追随者体系结构模式提供了一种高效的并发模型，多个线程依次共享一组事件源，从而对事件源中产生的服务请求进行检测、多路分解、分配和处理。对由缓冲池中的线程进行处理的请求没有同步化或没有排序约束时，领导者/追随者模式可以代替半同步/半异步和主动对象模式，以改进性能。 366

半同步/半异步和领导者/追随者模式的实现可以使用主动对象和监视器对象模式有效地协调对共享对象的访问。

本章最后一种模式为解决某些并发的内在复杂性提供了另一种策略：

- 线程特定的存储器设计模式允许多个线程使用一个“逻辑上全局的”访问点检索某一本地对象，而不会导致访问对象时发生加锁开销。在某种程度上，这种模式可以看做其他模式的“对立面”，因为它通过防止进程间的资源共享来实现复杂的内部并发机制。

本章所有模式的实现可以使用第4章（同步模式）中的模式来保护并发访问中的临界区。

文献中提到的涉及并发问题的其他模式有：主从（Master-Slave）模式[POSA1]，生产者/消费者（Producer-Consumer）模式[Grand98]，调度程序（Scheduler）模式[Lea99a]和两阶段终止（Two-phase Termination）模式[Grand98]。

367

5.1 主动对象

主动对象（Active Object）设计模式将方法执行和方法调用分离，加强并发和简化对驻留在自身控制线程中对象的同步访问。

1. 别名 并发对象(concurrent object)。

2. 例子

考虑一个通信网关的设计[⊖]，该设计分离协作组件并允许它们在彼此没有直接依赖的情况下进行交互。如图5-1所示，在分布式系统中，网关可能从一个或多个供应者进程将信息发送到一个或多个消费者进程。

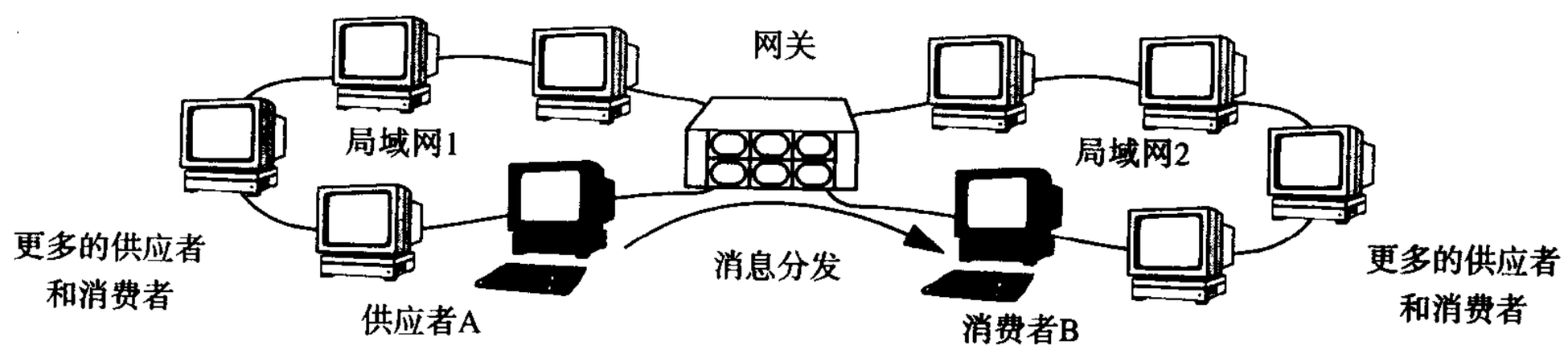


图 5-1

供应者，消费者和网关使用TCP[Ste93]通信，TCP是一个面向连接的协议。网关在向远程消费者发送数据时，可能遭遇TCP传输层的流控制。TCP使用流控制确保速度较快的供应者或网关产生数据的速度不会超过速度较慢的消费者或发生拥塞的网络能够缓冲和处理数据的速度。为了改进所有供应者和消费者的端到端服务质量（Quality of Service, QoS），整个网关进程在等待流控制终止对某一消费者的连接时不会发生阻塞。并且，当供应者和消费者增加时，网关必须相应地按比例扩大。

369

防止阻塞和改进性能的一种有效方法是将并发引入网关设计，例如为每个TCP连接关联一个不同的控制线程。这种设计可以阻塞其TCP连接被流控制的线程，而又不妨碍其连接未被流控制的线程的执行。因此需要确定怎样编写网关线程以及这些线程如何与供应者和消费者处理程序交互。

3. 语境

访问运行在独立控制线程中的对象的客户机。

⊖ 关于本例子更多的细节，参见接受器-连接器模式。

4. 问题

许多应用程序受益于使用并发对象而改进了服务质量，例如允许一个应用程序同时处理多个客户机请求。被动对象在客户机的调用对象方法的控制线程中执行这些方法，与使用单线程被动对象不同，并发对象处于自身的控制线程中。然而，对象并发运行时，如果这些对象被多个客户机线程共享和修改，则必须同步化对它们的方法和数据的访问，这时产生三个强制条件：

- 并发地调用对象的处理密集的方法不应无限期地阻塞整个进程，并因此降低其他并发对象的服务质量。

  ➡例如，在网关例子中，如果一个流出的TCP连接由于流控制而被阻塞，在等待流控制释放时，网关进程应该仍然能够运行其他能将新消息排队的线程。类似地，如果其他流出的TCP连接没有被流控制，那么网关内的其他线程也应该可以不管被阻塞的连接而向消费者发送消息。 ❑

- 应该很容易编程实现对共享对象的同步化访问。特别地，对共享对象的客户机方法调用的串行化和调度都应是透明的，该共享对象具有同步约束。

  ➡如果开发者使用低层同步机制，如显式地请求和释放互斥锁，则很难编程实现类似网关的应用。当对象被多个线程访问时，受到同步约束限制的方法，如在TCP连接中将消息入队列和出队列，应实现透明的串行化。 ❑ 370

- 应用程序的设计应透明地使用在硬件/软件平台提供的并行机制。

  ➡在网关例子中，发往不同消费者的消息应当被网关通过不同的TCP连接同时发送。然而，如果整个网关设计成只能在一个控制线程中运行，则不能通过在多处理器平台上运行网关而透明地消除性能瓶颈。 ❑

5. 解决方案

对受以上强制条件约束的对象而言，将其方法调用与执行分离。方法调用应该发生在客户机的控制线程中，而方法执行应发生在另一个线程中。此外，要将这种分离设计成使客户机线程看上去像调用普通的方法一样。

细节：代理（Proxy）[POSA1][GoF95]代表主动对象的接口，服务者（Servant）[OMG98a]提供了主动对象的实现。代理和服务者在不同的线程中运行，因此方法调用和方法执行可以并发执行。代理在客户机线程中运行，而服务者在另一个线程中运行。

在运行时代理将客户机的方法调用转换为方法请求，调度程序将方法请求存储在一个激活表（Activation List）中。调度程序的事件循环和服务者在同一线程中连续运行，将方法请求从激活表队列中取出并将它们分配给服务者。客户机可以通过代理返回的前景得到方法执行的结果。

6. 结构

一个主动对象由6部分组成：

代理[POSA1][GoF95]提供一个接口允许客户机调用主动对象的公共可访问的方法。代理的使用允许应用程序使用标准的强类型语言特性编程，而不是在线程间传递弱类型的消息。代理驻留在客户机线程中。

371

客户机调用一个由代理定义的方法时，它触发方法请求对象的构造函数。方法请求包括语境信息（如方法的参数），执行指定方法调用和向客户机返回结果时需要这些信息。方法请求类定义了一个执行主动对象方法的接口。接口也包括可以用来确定某一方法请求是否可执行的哨兵方法。对由主动对象中需要同步化访问的代理所提供的每个公共方法，从方法请求类中派生出具体方法请求类。如图5-2所示。

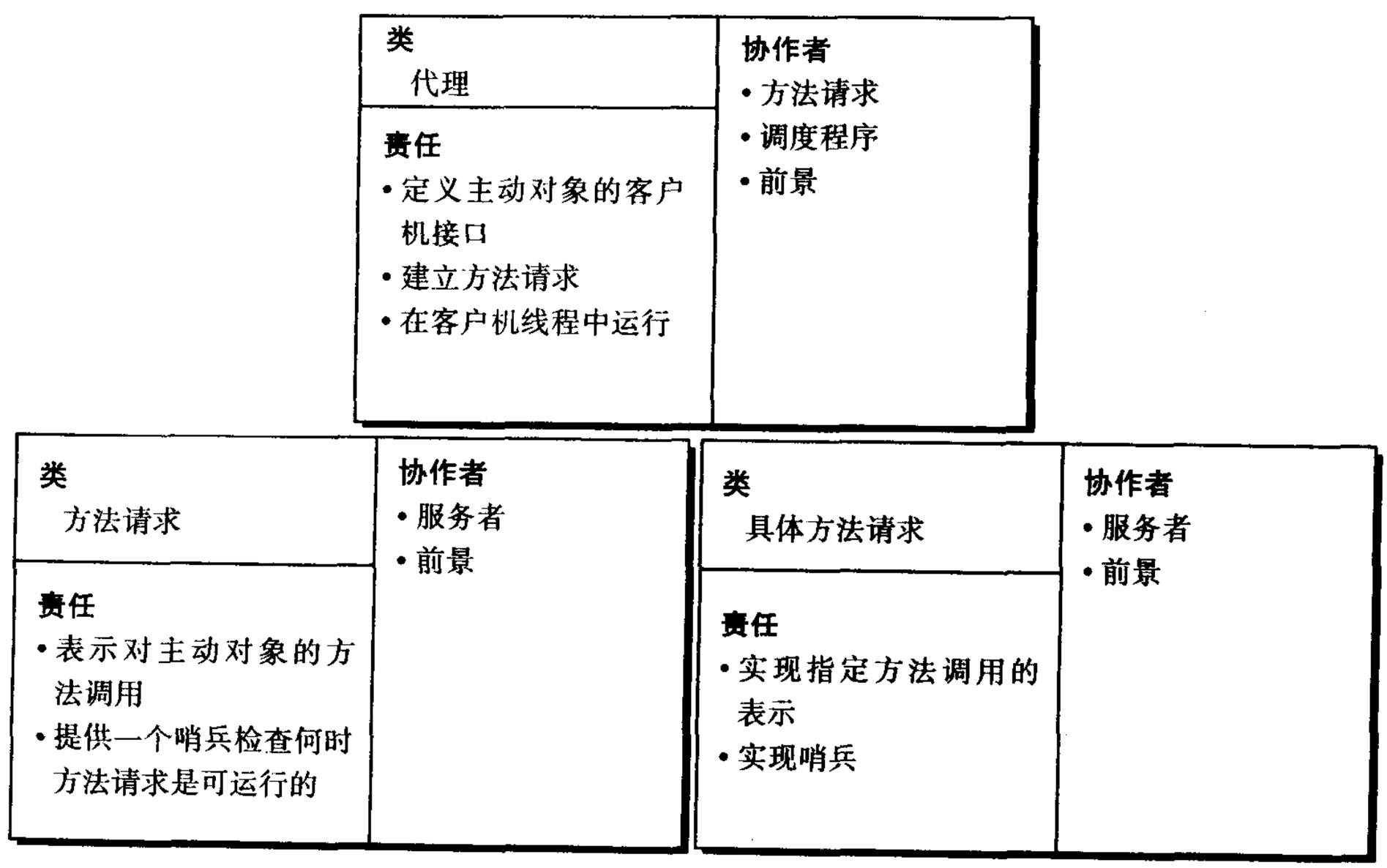


图 5-2

代理将它建立的具体方法请求插入一个激活表。该表包含一个待处理的由代理建立的方法请求组成的有限缓冲区，并跟踪哪些方法请求可执行。激活表将驻留代理的客户机线程和执行服务的线程分开，这样两个线程可以同时运行。因此，激活表的内部状态必须串行化，以防对它同时访问。

372

调度程序在其客户机代理之外的线程中运行，也就是主动对象的线程。它决定下一次对主动对象将执行哪一个方法请求。这一调度决策基于不同的准则（如排序——对主动对象调用方法的顺序），或者基于主动对象的某些属性（如它的状态）。调度程序可以使用方法请求哨兵评估这些属性，方法请求哨兵确定何时可以执行这些方法请求[Lea99a]。调度程序用激活表管理待处理的方法请求。当客户机调用其方法时，代理将方法请求插入激活表中。如图5-3所示。

服务者被建模为主动对象，其中定义了其行为和状态。服务者实现的方法对应于代理接口和由代理建立的方法请求。服务者可能还包含其他谓词方法，方法请求能够用这些谓词方法来实现哨兵。在调度程序执行与之关联的方法请求时调用服务者方法。因而，它在调度程序的线程中执行。

当客户机调用代理上的方法时，会接收到一个前景[Hal85][LS88]。该前景允许客户机在服务者完成方法的执行后获得方法调用的结果。每前景都为被调用的方法保留一定的空间，存放

方法调用的结果。当客户机想得到该结果时，它可以与前景汇合（rendezvous），阻塞或轮询直到结果被计算出来并存入前景。如图5-4所示。

类 激活表	协作者
责任 • 维护待执行的方法请求 • 方法请求由代理插入，由调度程序删除	

类 调度程序	协作者 • 激活表 • 方法请求
责任 • 将方法请求插入激活表 • 在主动对象线程中运行	

图 5-3

类 服务者	协作者
责任 • 实现主动对象 • 在主动线程中运行	

类 前景	协作者
责任 • 存储主动对象方法调用的结果 • 为客户机提供汇合点	

图 5-4

373

主动对象模式类图如图5-5所示。

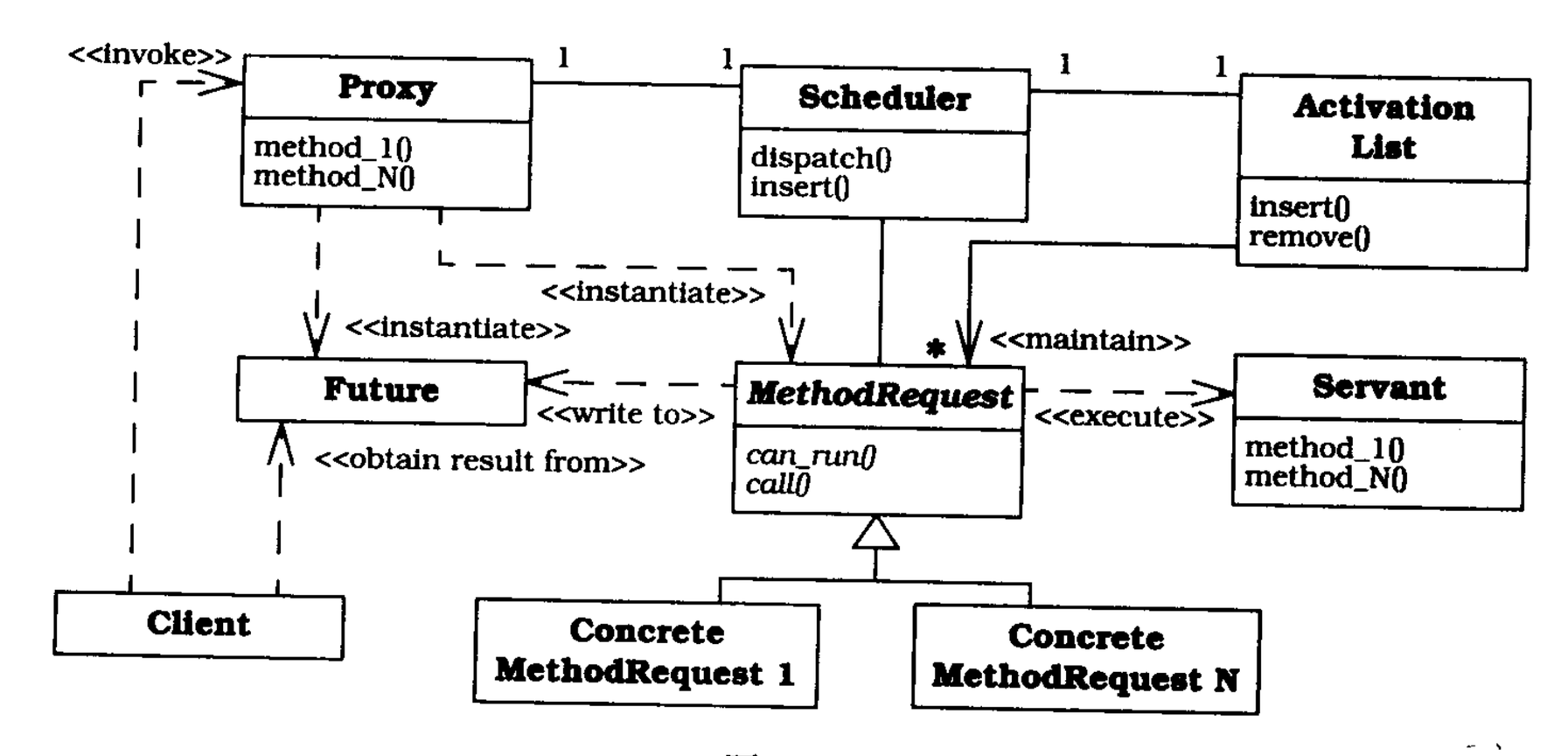


图 5-5

7. 动态特性

主动对象模式的行为可以分为三个阶段（如图5-6）：

- 方法请求的构造和调度。客户机调用代理的方法。该行为触发了方法请求的建立，方法请求维护对方法的参数绑定以及执行方法和返回结果所必须的其他任何绑定。然后代理将方法请求传递给它的调度程序，调度程序将方法请求放入激活表队列。如果方法被定义为双

向调用[OMG98a]，则向客户机返回一个前景。如果方法是单向的，意味着没有返回值，则不返回前景。

- *方法请求的执行*。主动对象的调度程序在与客户机不同的线程中持续运行。调度程序监视它的激活表，通过调用方法请求的哨兵方法确定哪一个方法请求可运行。当方法请求可运行时，调度程序将它从激活表中删除，将请求绑定到服务者，并分配合适的服务者方法。当该方法被调用时，该方法可以访问并更新它的服务者的状态，如果该方法请求是双向方法调用的话，还可以创建结果。

374

- *完成*。在这一阶段，调用的结果（如果有的话）存储在前景中，主动对象的调度程序返回，监视激活表的状态，以获得可运行的方法请求。当双向方法完成后，客户机可以通过前景获得方法的执行结果。一般说来，任何与前景汇合的客户机都可以获得它的结果。当方法请求和前景不再被引用时，可以直接删除或者回收它们。

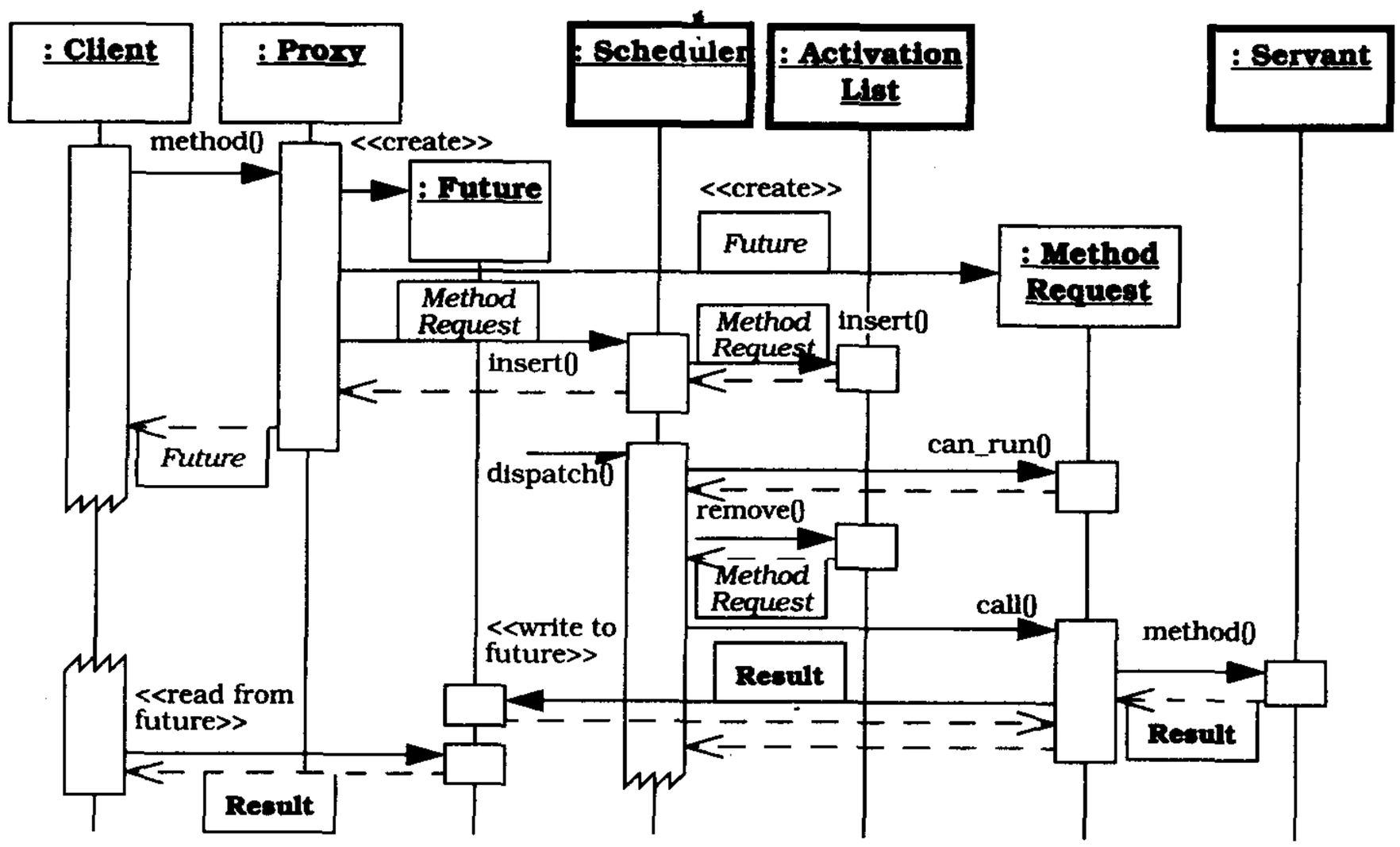


图 5-6

8. 实现

可以用五种活动说明如何实现主动对象模式。

(1) *实现服务者*。服务者被建模为主动对象，其中定义了其行为和状态。另外，服务者还可能包含用来确定何时执行方法请求的谓词方法。

➡对于网关例子中的每个远程消费者来说，都存在一个消费者处理程序，它包含一个到消费者进程的TCP连接，消费者进程运行在远程计算机上。每个消费者处理程序都包含一个被建模为主动对象的消息队列，并用MQ_Servant类实现。该主动对象存储着从供应者传递到网关，并等待传送到远程消费者的信息[⊖]。如下的C++类说明了MQ_Servant类：

375

⊖ 本例中的主动对象消息队列是一种对消息进行缓冲的实现机制，以避免当在TCP连接中发生流控制时阻塞网关。它与激活表没有关系，激活表是一个主动对象模式参与者，存储未执行的方法请求。对本例更多的讨论参见“已解决的例子”一节和监视器对象模式。

```
class MQ_Servant {
public:
    // Constructor and destructor.
    MQ_Servant (size_t mq_size);
    ~MQ_Servant ();

    // Message queue implementation operations.
    void put (const Message &msg);
    Message get ();

    // Predicates.
    bool empty () const;
    bool full () const;
private:
    // Internal queue representation, e.g., a circular
    // array or a linked list, that does not use any
    // internal synchronization mechanism.
};
```

put()和get()方法实现了队列中信息的插入和删除操作。服务者定义了两个谓词：empty()和full()，这两个谓词区分了三种内部状态：空、满和既不空也不满。这些谓词用来确定何时可以调用服务者的put()和get()方法。 ❑

一般来说，防止服务者的临界区被并发访问的同步机制与服务者之间的结合不应太紧密，而服务者应该仅仅实现应用程序的功能特性。相反，同步机制应与方法请求关联。这种设计避免了*继承异常的问题*[MWY91]，如果子类需要与基类不同的同步策略，该问题会妨碍对服务者实现的重用。因此，对主动对象同步约束的改变不应影响到服务者的实现。

➥设计的MQ_Servant类省略了服务者中的同步机制。因此，为了简明扼要，MQ_Servant类中的方法实现不需包含任何同步机制。 ❑ 376

(2) *实现调用的基础设施*。在这一活动中，描述了客户机调用主动对象的方法所必需的基础设施。这种基础设施由*建立方法请求的代理构成*，可通过两个子活动实现。

(2.1) *实现代理*。代理为客户机提供了访问服务者方法的接口。对于客户机的每次方法调用，代理都建立一个具体的方法请求。每一方法请求是方法语境的抽象，也称为方法闭包（closure）。一般情况下该语境包括方法参数，到被方法应用的服务者的绑定，结果的前景以及执行方法请求的代码。

➥在网关例子中，MQ_Proxy提供了如下在实现活动1中定义的到MQ_Servant的接口：

```
class MQ_Proxy {
public:
    // Bound the message queue size.
    enum { MQ_MAX_SIZE = /* ... */ };
    MQ_Proxy (size_t size = MQ_MAX_SIZE):
        scheduler_ (size), servant_ (size) { }

    // Schedule <put> to execute on the active object.
    void put (const Message &msg) {
        Method_Request *mr = new Put (servant_, msg);
        scheduler_.insert (mr);
    }

    // Return a <Message_Future> as the "future" result of
    // an asynchronous <get> method on the active object.
    Message_Future get () {
```

```
        Message_Future result;
        Method_Request *mr = new Get (servant_, result);
        scheduler_.insert (mr);
        return result;
    }

    // empty() and full() predicate implementations ...
private:
    // The servant that implements the active object
    // methods and a scheduler for the message queue.
    MQ_Servant servant_;
    MQ_Scheduler scheduler_;
};
```

377

MP_Proxy是一个构造方法请求实例并将它们传递给调度程序的工厂[GoF95]，调度程序将方法请求放入队列，等待随后在独立的线程中的执行。

进程中的多个客户机线程可以共享同一个代理。代理方法不需要被串行化，因为它在建立后不会改变状态。代理方法的调度程序和激活表负责所需的内部串行化处理。 ❑

➡网关例子包括许多供应者处理程序，它们通过多个消费者处理程序从对等端接收和发送消息。一些供应者处理程序可以使用属于一个客户机处理程序的代理来调用方法，而不需要任何显式的同步机制。 ❑

(2.2) *实现方法请求*。方法请求可以看做一个命令对象[GoF95]。方法请求类声明了一个被所有具体方法请求使用的接口。它为调度程序提供了一个统一的接口，而不用了解如何评估同步约束或者如何触发具体方法请求的执行。一般地，该接口声明了一个can_run()方法，该方法定义了一个钩子方法哨兵，用于检查何时可以执行方法请求。该接口也声明了一个call()方法，定义一个执行服务者的方法请求的钩子。

必须由子类定义方法请求类中的方法。每个代理中定义的方法必须都有一个具体的方法请求类。can_run()方法通常在服务者谓词的帮助下实现。

➡在网关例子中Method_Request基类定义了两个虚钩子方法，分别是can_run()和call()：

```
class Method_Request {
public:
    // Evaluate the synchronization constraint.
    virtual bool can_run () const = 0

    // Execute the method.
    virtual void call () = 0;
};
```

378

定义两个Method_Request类的子类：Put类对应于代理的put()方法调用，Get类对应get()方法调用。两个类都包含一个到MQ_Servant的指针。Get类可以实现如下：

```
class Get : public Method_Request {
public:
    Get (MQ_Servant *rep, const Message_Future &f)
        : servant_ (rep), result_ (f) { }

    virtual bool can_run () const {
```

```
        // Synchronization constraint: cannot call a
        // <get> method until queue is not empty.
        return !servant_->empty ();
    }

    virtual void call () {
        // Bind dequeued message to the future result.
        result_ = servant_->get ();
    }
private:
    MQ_Servant *servant_;
    Message_Future result_;
};
```

请留意，can_run()方法如何使用MQ_Servant的empty()谓词来允许调度程序确定何时可执行get方法请求。执行方法请求时，主动对象的调度程序调用它的call()钩子方法。该call()钩子使用运行时绑定到MQ_Servant的Get方法请求来调用服务者的get()方法，get()方法在服务者的语境中执行。然而，主动对象的调度程序通过方法请求的can_run()方法加强了所需的同步约束，因此它不需要任何显式的串行机制。 ❑

对于代理中的每个具有返回值的公共双向方法（如网关例子中的get()方法），代理将一个前景传递给相应的方法请求类的构造函数。该前景被返回到调用方法的客户机线程，这在实现活动(5)中有讨论。

(3) *实现激活表*。每个方法请求都插入到一个激活表中。该表可以实现为同步限界缓冲区，该缓冲区由客户机线程和运行主动对象调度程序及服务者的线程共享。激活表也可以提供一个*健壮的迭代器*（robust iterator）[Kof93][CarEl95]，调度程序可以遍历和删除它的元素。 379

通常用的并发控制模式（如监视器对象）设计激活表，而并发控制模式使用普通同步机制（如状态变量和互斥量[Ste98]）。当它们与定时器机制联合作用时，调度程序线程可以确定等待具体操作完成的时间。例如，可以用定时等待来限制从空激活表中删除一个方法请求或向一个满的激活表插入一个方法请求的时间[⊖]。如果超时，控制将返回到调用线程，而不会执行方法请求。

➡对于网关例子，可以定义如下的Activation_List类：

```
class Activation_List {
public:
    // Block for an "infinite" amount of time waiting
    // for <insert> and <remove> methods to complete.
    enum { INFINITE = -1 };

    // Define a "trait".
    typedef Activation_List_Iterator iterator;

    // Constructor creates the list with the specified
    // high water mark that determines its capacity.
    Activation_List (size_t high_water_mark);

    // Insert <method_request> into the list, waiting up
    // to <timeout> amount of time for space to become
    // available in the queue. Throws the <System_Ex>
    // exception if <timeout> expires.
    void insert (Method_Request *method_request,
```

⊖ 当表的当前方法请求数等于它的上限时，称该表“满”。

```
                    Time_Value *timeout = 0);

    // Remove <method_request> from the list, waiting up
    // to <timeout> amount of time for a <method_request>
    // to be inserted into the list. Throws the
    // <System_Ex> exception if <timeout> expires.
    void remove (Method_Request *&method_request,
                 Time_Value *timeout = 0);
private:
    // Synchronization mechanisms, e.g., condition
    // variables and mutexes, and the queue implemen-
    // tation, e.g., an array or a linked list, go here.
};
```

380

insert()和remove()方法提供了一个限界缓冲区生产者/消费者[Grand98]同步模型。该设计使调度程序线程和多个客户机线程可以同时删除和插入Method_Request而不会破坏Activation_List的内部状态。客户机线程扮演着生产者的角色并且通过代理插入Method_Request。调度程序线程扮演着消费者的角色。它在Method_Request的哨兵得到“真”值时，将它们从Activation_List中删除。然后调用它们的call()钩子方法执行服务者方法。 ❑

(4) *实现主动对象的调度程序*。调度程序是一个命令处理程序[POSA1]，用于管理激活表和执行满足同步约束的未处理的方法请求。调度程序的公共接口通常提供两个方法，一个被代理用于将方法请求插入到激活表中，另一个用于将方法请求分配给服务者。

➡针对网关例子，定义了如下的MQ_Scheduler类：

```
class MQ_Scheduler {
public:
    // Initialize the <Activation_List> to have
    // the specified capacity and make <MQ_Scheduler>
    // run in its own thread of control.
    MQ_Scheduler (size_t high_water_mark);

    // ... Other constructors/destructors, etc.

    // Put <Method_Request> into <Activation_List>. This
    // method runs in the thread of its client, i.e.
    // in the proxy's thread.
    void insert (Method_Request *mr) {
        act_list_.insert (mr);
    }

    // Dispatch the method requests on their servant
    // in its scheduler's thread of control.
    virtual void dispatch ();
private:
    // List of pending Method_Requests.
    Activation_List act_list_;

    // Entry point into the new thread.
    static void *svc_run (void *arg);
};
```

381

❑

调度程序在客户机线程之外的另一个控制线程中执行它的dispatch()方法。每个客户机线程使用一个代理将方法请求插入到主动对象调度程序的激活表中。该调度程序监视自身线程的激活表，选择哨兵求值为‘真’，也就是满足同步约束的方法请求。然后从激活表中删除该方

法请求，并通过调用它的call()钩子方法执行该方法请求。

➡在网关例子中，MQ_Scheduler的构造函数初始化Activation_List，并使用Thread_Manager包装器外观产生一个新的控制线程。

```
MQ_Scheduler::MQ_Scheduler (size_t high_water_mark):
    act_queue_ (high_water_mark) {
    // Spawn separate thread to dispatch method requests.
    Thread_Manager::instance ()->spawn (&svc_run, this);
}
```

向Thread_Manager::spawn()方法传递一个指向MQ_Scheduler::svc_run()静态方法的指针和一个指向MQ_Scheduler对象的指针。svc_run()静态方法是到新建立的控制线程的入口点，该线程运行svc_run()方法。该方法只是一个适配器[GoF95]，调用this参数所指对象的MQ_Scheduler::dispatch()方法：

```
void *MQ_Scheduler::svc_run (void *args) {
    MQ_Scheduler *this_obj =
        static_cast<MQ_Scheduler *> (args);

    this_obj->dispatch ();
}
```

dispatch()方法基于底层的MQ_Servant谓词empty()和full()，确定处理put和get方法请求的顺序。这些谓词反映了服务者的状态，如消息队列为空、满，还是不空不满。 382

通过方法请求的can_run()方法估计这些谓词约束的值后，调度程序可以确保对MQ_Servant的正确访问：

```
virtual void MQ_Scheduler::dispatch () {
    // Iterate continuously in a separate thread.
    for (;;) {
        Activation_List::iterator request;
        // The iterator's <begin> method blocks
        // when the <Activation_List> is empty.
        for (request = act_list_.begin ();
            request != act_list_.end ();
            ++request) {
            // Select a method request whose
            // guard evaluates to true.
            if ((*request).can_run ()) {
                // Take <request> off the list.
                act_list_.remove (*request);
                (*request).call ();
                delete *request;
            }
            // Other scheduling activities can go here,
            // e.g., to handle when no <Method_Request>s
            // in the <Activation_List> have <can_run>
            // methods that evaluate to true.
        }
    }
}
```

❑

在例子中，MQ_Scheduler::dispatch()的实现连续迭代，执行满足can_run()方法值为“真”的下一方法请求。不过，调度程序的实现可以更复杂，包含许多表示服务者的同步状态的变量。

例如，为了实现多阅读器/单记录器同步策略，预期的记录器会调用代理上的write，传递要写入的数据。类似地，阅读器将调用read，并获得一个前景作为返回值。主动对象的调度程序保持几个计数器变量以跟踪同步状态，如阅读和记录请求的次数。调度程序也保持预期记录器的标识信息。

主动对象的调度程序可以使用这些同步状态计数器来确定单记录器何时可以进行处理，也就是，什么时候当前记录器的数量是零并且当前没有从其他记录器发出的记录请求等待处理。当一个记录请求到达后，调度程序可以分配记录器以确保公平。反之，当阅读请求到达并且服务者因状态非空而可以满足这些请求时，调度程序可以阻塞所有记录活动并且先分配阅读请求。

383

上述的同步状态计数器变量值不依赖于服务者的状态，因为它们仅仅被调度程序用来代表服务者加强正确的同步策略。服务者的任务仅仅集中于临时存储客户机特定的应用程序数据。相反，调度程序关注多个阅读器和记录器的协调。这种设计增强了模块性和可重用性。

调度程序可以通过使用策略模式[GoF95]支持多种同步策略。每种同步策略被封装进独立的策略类中。用特定的同步策略来配置扮演着策略模式中语境角色的调度程序，调度程序根据该策略进行所有后续的调度决策。

(5) *确定汇合和返回值策略*。汇合策略确定客户机如何从对主动对象调用的方法中得到返回值。在一个线程中执行的主动对象服务者向在另一线程中运行的调用该方法的客户程序传递返回值时，汇合发生。实现主动对象模式通常选择如下汇合和返回值策略：

- *同步等待*。在代理中同步阻塞客户机线程，直到调度程序分配方法请求并且得到结果并存储到前景中。
- *同步定时等待*。阻塞有限的时间，并且如果主动对象的调度程序在规定时间内未能分配方法请求则失败。如果超时为零，客户机线程就“轮询”，也就是在调度程序不能将它立即分配时可返回到调用者而不需将方法请求放入队列。
- *异步*。将方法调用放入队列并且立即将控制返回给客户机。如果方法是一个产生结果的双向调用，则必须使用某些形式的前景以提供对值的同步化访问，或者在调用失败时对错误状态的同步化访问。

384

前景的构造允许双向异步调用[ARSK00]，将值返回给客户机。在服务者完成了方法执行后，它获取一个对前景的记录锁并用结果更新前景。因等待结果而被阻塞的所有客户机线程都被唤醒，并且可以并发地访问结果。前景可以在记录器和所有阅读器线程不再引用它后被回收。在类似C++的不支持碎片回收的语言中，当不再通过类似计数指针的惯用法[POSA1]使用前景时，可以重新声明前景。

➡在网关例子中，调用MQ_Proxy中的get()方法最后会导致Get::call()方法被MQ_Scheduler分配，如实现活动（2）所示。MQ_Proxy::get()方法返回一个值，因此一个Message_Future被返回到调用它的客户机：

```
class Message_Future {
public:
    // Binds <this> and <f> to the same <Msg._Future_Imp.>
    Message_Future (const Message_Future &f);
```

```
    // Initializes <Message_Future_Implementation> to
    // point to <message> m immediately.
    Message_Future (const Message &message);

    // Creates a <Msg._Future_Imp.>
    Message_Future ();

    // Binds <this> and <f> to the same
    // <Msg._Future_Imp.>, which is created if necessary.
    void operator= (const Message_Future &f);

    // Block upto <timeout> time waiting to obtain result
    // of an asynchronous method invocation. Throws
    // <System_Ex> exception if <timeout> expires.
    Message result (Time_Value *timeout = 0) const;
private:
    // <Message_Future_Implementation> uses the Counted
    // Pointer idiom.
    Message_Future_Implementation *future_impl_;
};
```

使用计数指针惯用法[POSA1]实现Message_Future。该惯用法通过使用一个带引用计数的Message_Future_Implementation结构（只能通过Message_Future句柄访问该结构）简化了动态分配C++对象的内存管理。 ❑ 385

通常，可以使客户机立即从前景中求出结果值，这种情况下客户机会一直阻塞，直到调度程序执行了方法请求。相反，对主动对象方法调用所返回结果的求值可以推迟。在这种情况下，客户机线程和执行方法的线程可以异步地进行处理。

➡针对网关例子，可以阻塞在单独的线程中运行的客户机处理程序，直到来自于供应者的新的消息到达：

```
MQ_Proxy message_queue;

// Obtain future and block thread until message arrives.
Message_Future future = message_queue.get ();
Message msg = future.result ();

// Transmit message to the consumer.
send (msg);
```

❑

9. 已解决的例子

在网关例子中，网关的供应者和消费者处理程序分别是远程供应者和消费者的本地代理

[POSA1][GoF95]。供应者处理程序接收远程供应者的消息并检查消息中的地址域。该地址用来作为路由表中的关键字区分哪个远程客户机将接收该消息。

386

路由表中维持着消费者处理程序的一个映射，每个处理程序负责通过独立的TCP连接向远程消费者发送消息。为了处理不同的TCP连接上的流控制，每个消费者处理程序包含一个使用主动对象模式实现的消息队列。这种设计将供应者和消费者处理程序分离，以便它们可以互不依赖地并发运行和阻塞。如图5-7所示。

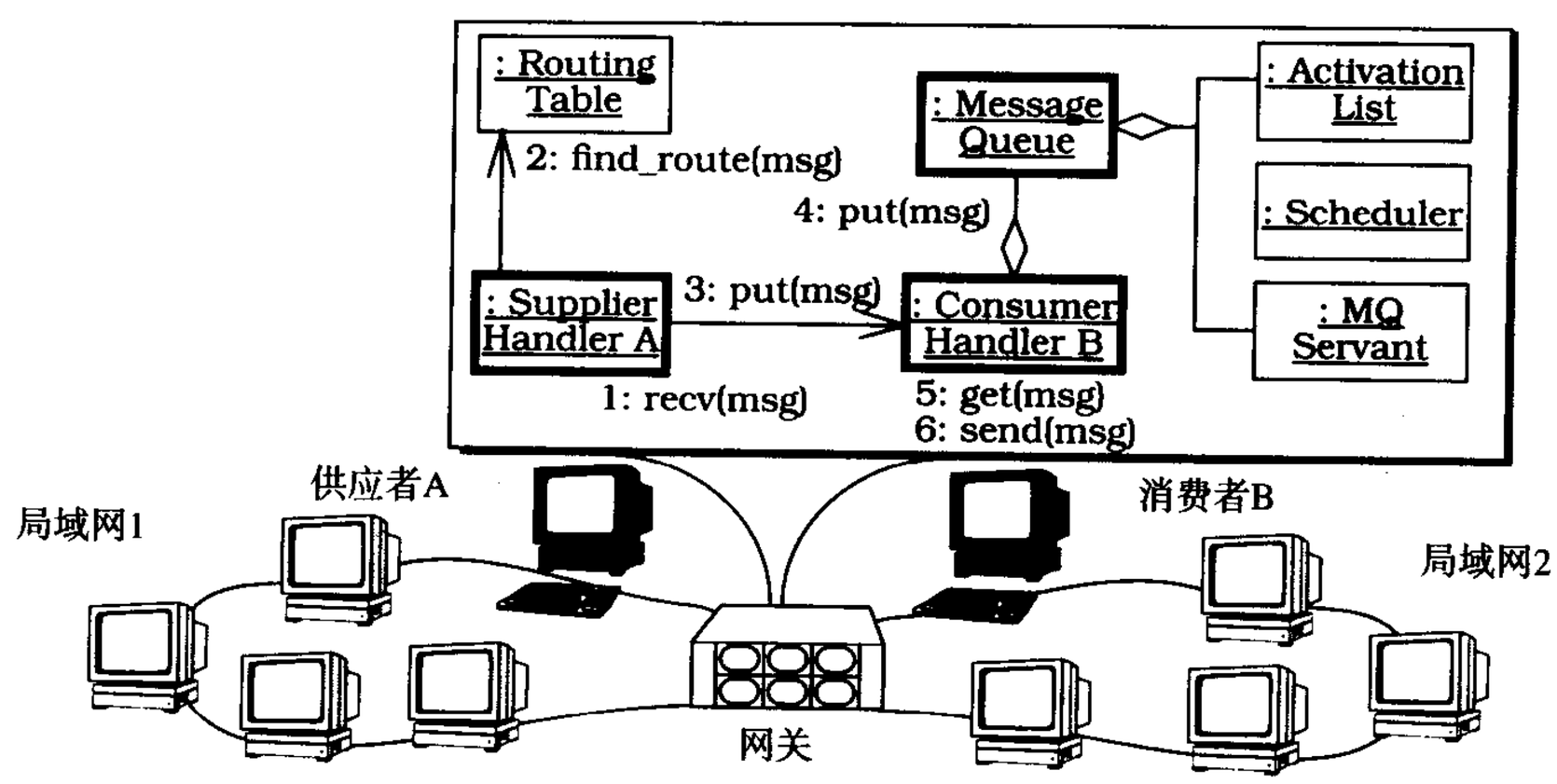


图 5-7

Consumer_Handler类定义如下：

```
class Consumer_Handler {
public:
    // Constructor spawns the active object's thread.
    Consumer_Handler ();

    // Put the message into the queue.
    void put (const Message &msg) { msg_q_.put (msg); }
private:
    MQ_Proxy msg_q_; // Proxy to the Active Object.
    SOCK_Stream connection_; // Connection to consumer.

    // Entry point into the new thread.
    static void *svc_run (void *arg);
};
```

387

运行在各自线程中的Supplier_Handlers可以将消息放入相应的Consumer_ Handler的消息队列主动对象中：

```
void Supplier_Handler::route_message (const Message &msg)
{
    // Locate the appropriate consumer based on the
    // address information in <Message>.
    Consumer_Handler *consumer_handler =
        routing_table_.find (msg.address ());

    // Put the Message into the Consumer Handler's queue.
```

```
    consumer_handler->put (msg);
}
```

为了处理插入队列中的消息，每个Consumer_Handler使用Thread_Manager包装器外观在它的构造函数中产生一个独立的控制线程：

```
Consumer_Handler::Consumer_Handler () {
    // Spawn a separate thread to get messages from the
    // message queue and send them to the consumer.
    Thread_Manager::instance ()->spawn (&svc_run, this);
    // ...
}
```

这个新的线程执行svc_run()的方法入口点，由svc_run()方法获得由供应者处理程序线程放入队列中的消息，并通过TCP连接将它们发送给消费者：

```
void *Consumer_Handler::svc_run (void *args) {
    Consumer_Handler *this_obj =
        static_cast<Consumer_Handler *> (args);
    for (;;) {
        // Block thread until a <Message> is available.
        Message msg = this_obj->msg_q_.get ().result ();
        // Transmit <Message> to the consumer over the
        // TCP connection.
        this_obj->connection_.send (msg, msg.length ());
    }
}
```

每个Consumer_Handler对象使用实现为主动对象的消息队列并在自己的线程中运行。因此，可以阻塞它的send( )操作，而不会影响到其他Consumer_Handler对象的服务质量。 388

10. 变体

多角色。如果一个主动对象实现的是多角色，每个角色被特定类型的客户机使用，那么可以为每个角色引入一个独立的代理。通过使用扩展接口模式，客户机可以获得它们需要的代理。这种设计有助于事务分离，因为客户机只能见到自身操作需要的主动对象的特定方法，这样进一步简化了主动对象的演化。例如，新服务可以通过提供新的扩展接口代理的方式加入主动对象而不用改变现存服务。不需要访问新服务的客户机不会受扩展的影响，甚至不需要重新进行编译。

集成化调度程序。为了减少实现主动对象模式所需要的组件数量，可以在调度程序组件中集成代理和服务者的角色。同样，从代理的方法调用到方法请求的转换也可以集成到调度程序中。不过，服务者仍然需要在代理线程之外的线程中运行。

➡下面给出了一个使用集成化调度程序的消息队列的实现：

```
class MQ_Scheduler {
public:
    MQ_Scheduler (size_t size)
        : servant_ (size), act_list_ (size) { }
    // ... other constructors/destructors, etc.

    void put (const Message m) {
        Method_Request *mr = new Put (&servant_, m);
        act_list_.insert (mr);
```

```
    }

    Message_Future get () {
        Message_Future result;
        Method_Request *mr = new Get (&servant_, result);
        act_list_.insert (mr);
        return result;
    }

    // Other methods ...
private:
    MQ_Servant servant_;
    Activation_List act_list_;
    // ...
};
```

389

通过将方法请求的生成集中在一个地方，可以简化主动对象模式的实现，因为这样的话组件更少。当然，存在的缺陷是调度程序必须知道服务者和代理的类型，这使不同类型的主动对象难以重用一个调度程序。

消息传递。对集成化调度程序变体的进一步精简是去掉代理和服务者，并在客户机线程和主动对象的调度程序线程间使用直接消息传递。

➡例如，考虑如下调度程序的实现：

```
class Scheduler {
public:
    Scheduler (size_t size): act_list_ (size) { }
    // ... other constructors/destructors, etc.

    void insert (Message_Request *message_request) {
        act_list_.insert (message_request);
    }

    virtual void dispatch () {
        for (;;) {
            Message_Request *mr;
            // Block waiting for next request to arrive.
            act_list_.remove (mr);
            // Process the message request <mr>...
        }
    }

    // ...
private:
    Activation_List act_list_;
    // ...
};
```

在这个变体中，没有代理，因此客户机自己直接创建了一个相应类型的消息请求并调用insert()，将请求放入激活表队列。程序中同样没有服务者，因此在调度程序线程中运行的dispatch()方法只需将下一个消息请求从队列中取出，并根据请求类型进行处理。 ❑

390

一般说来，开发消息传递机制比开发一个主动对象更容易，因为前者具有更少的组件。然而，消息传递可能更乏味且易出错，因为应用程序开发者必须编程实现代理和服务者逻辑，而主动对象开发者则不必。因此，消息传递实现比主动对象实现的类型安全性更少一些，因为它们的接口是隐式的而不是显式的。另外，对于应用程序开发者来说，通过消息传递分配客户机

和服务器更困难，因为没有代理对数据的列集和散集进行封装。

多态前景[LK95]。多态前景允许对用前景表示的最终结果类型的参数化，并增强了必要的同步。特别地，多态前景描述了一个类型化的前景，客户机线程可以用它来获取方法请求的结果。客户机是否阻塞于一个前景取决于是否计算出了结果。

➡如下类是C++的多态前景模板：

```
template <class TYPE>
class Future {
    // This class can be used to return results from
    // two-way asynchronous method invocations.
public:
    // Constructor and copy constructor that binds <this>
    // and <r> to the same <Future> representation.
    Future ();
    Future (const Future<TYPE> &r);

    // Destructor.
    ~Future ();

    // Assignment operator that binds <this> and <r> to
    // the same <Future> representation.
    void operator = (const Future<TYPE> &r);

    // Cancel a <Future> and reinitialize it.
    void cancel ();

    // Block upto <timeout> time waiting to obtain result
    // of an asynchronous method invocation. Throws
    // <System_Ex> exception if <timeout> expires.
    TYPE result (Time_Value *timeout = 0) const;
private:
    // ...
};
```

391

客户机可以如下使用多态前景：

```
try {
    // Obtain a future (does not block the client).
    Future<Message> future = message_queue.get ();
    // Do something else here...

    // Evaluate future and block for up to 1 second
    // waiting for the result to become available.
    Time_Value timeout (1);
    Message msg = future.result (&timeout);

    // Do something with the result ...
} catch (System_Ex &ex) {
    if (ex.status () == ETIMEDOUT) /* handle timeout */
}
```

❑

定时方法调用。在实现活动（3）中说明的激活表定义了一个可以限制调度程序等待插入或删除方法请求时间的机制。虽然前面的模式中的例子没有使用这一特性，但许多应用程序可以从定时方法调用中获益。只需要简单地通过调度程序和代理提供超时机制就可以来实现该特性。

➡在网关例子中，可以修改MQ_Proxy，以便客户机使用它的方法限定愿意等待执行的时间：

```
class MQ_Proxy {
public:
    // Schedule <put> to execute, but do not block longer
    // than <timeout> time. Throws <System_Ex>
    // exception if <timeout> expires.
    void put (const Message &msg,
              Time_Value *timeout = 0);

    // Return a <Message_Future> as the "future" result of
    // an asynchronous <get> method on the active object,
    // but do not block longer than <timeout> amount of
    // time. Throws the <System_Ex> exception if
    // <timeout> expires.
    Message_Future get (Time_Value *timeout = 0);
};
```

如果timeout的值是0，那么get()和put()都将一直阻塞，直到Message被删除或插入调度程序的激活表。如果timeout达到设定值，那么在包装器外观模式中定义的System_Ex异常，其status()的值为ETIMEDOUT，客户机必须要捕捉到它。

392

为了进一步完善对定时方法调用的支持，必须为MQ_Scheduler加入对超时的支持：

```
class MQ_Scheduler {
public:
    // Insert a method request into the <Activation_List>
    // This method runs in the thread of its client, i.e.
    // in the proxy's thread, but does not block longer
    // than <timeout> amount of time. Throws the
    // <System_Ex> exception if the <timeout> expires.
    void insert (Method_Request *method_request,
                 Time_Value *timeout) {
        act_list_.insert (method_request, timeout);
    }
}
```

❑

分布式主动对象。在这种变体中，代理和调度程序间存在一个分布边界，而不仅仅是线程边界。这种模式变体引入了两个新的参与者：

- 一个扮演桩（stub）角色的客户机代理，将方法参数列集为通过网络发送并由服务者在独立的服务器地址空间执行的方法请求。
- 一个服务器端构架，方法请求参数被传递到服务器的服务者方法之前散集这些方法请求参数。

该分布式主动对象模式变体类似于代理者模式[POSA1]。主要的区别是代理者模式通常协调许多对象的处理，而分布式主动对象仅仅处理单独的对象。

线程池主动对象。这种主动对象模式的泛化支持每个主动对象对应多个服务者线程，以增加吞吐量和反应能力。当没有处理请求时，每个在线程池主动对象块中的服务者线程阻塞于一个单独的激活表。一旦有一个方法请求可执行，主动对象的调度程序就分配一个新的方法请求给线程池中的一个可用的服务者线程。

线程池中所有服务者线程都共享同一个服务者实现。因此在服务者方法没有通过某种同步机制（如互斥机制）对它们的内部状态进行保护时不能使用这种设计。

393

可以在[Lea99a]中找到主动对象的更多变体。

11. 已知使用

ACE框架[Sch97]。ACE框架提供了主动对象模式中的方法请求、激活表以及特性组件的可重用的实现。ACE中相应的类称为ACE_Method_Request、ACE_Activation_Queue和ACE_Future。这些组件用来实现许多并发和网络化的系统[Sch96]。

西门子 Syngo。主动对象模式用于西门子Syngo框架，为电子医疗影像系统提供一个基于组件的黑盒框架。Syngo采用了主动对象模式与命令处理器模式[POSA1]结合的方式以简化访问各种医疗服务器上的患者信息的客户机窗口应用程序[JWS98]。

西门子 FlexRouting——自动呼叫分配[Flex98]。该呼叫中心管理系统使用主动对象模式的线程池变体。呼叫中心提供的服务作为他们自己的应用程序来实现。例如，根据所提供的服务类型的不同，可能有一个热线应用程序、一个订购应用程序和一个产品信息应用程序。这些应用程序帮助操作员为各种客户服务。这些应用程序的每个实例是一个独立的服务者组件。一个对应于调度程序的“FlexRouter”组件自动将到达的客户请求发送给可以处理这些请求的操作员应用程序。

Java JDK1.3在类java.util.Timer和java.util.TimerTask中引入了并发执行基于定时器任务的机制。一旦任务的计划执行时间到，它就会被执行。特别地，Timer提供了允许客户机指定任务应何时执行以及间隔多久执行的不同的调度函数。单步任务是一直向前的，循环的任务可以在周期间隔内调度。在客户机线程中执行调度调用，而任务自身在Timer对象拥有的线程中执行。因为上述两个线程是并发执行的，所以有锁保护Timer内的任务队列。 394

任务队列可以实现为优先级队列，因此可以有效地识别下一个要超时的TimerTask。定时器线程仅仅等待预定的时间。这里没有显式的哨兵方法和谓词，因为确定一个任务何时是“可执行的”仅仅依赖于调度时间是否到达。

任务可以作为TimerTask类的子类实现，子类重载原来的run()钩子方法。TimerTask子类提供了一个类和一个接口方法TimerTask.run()，统一了隐藏在方法请求和服务者后面的概念。

上述方案简化了主动对象的以定时执行为目标的机制。代理和客户机并不直接调用调度程序——Timer对象。客户机并不调用通常的方法，因此并发是不透明的。并且，也没有连接到run()方法的返回值或前景对象。应用程序可以通过构造若干个Timer对象来使用若干个主动对象，每个Timer对象都具有自己的线程和任务队列。

餐厅厨师。可以在餐厅发现一个主动对象模式的现实生活的例子。侍应生将顾客点的菜单送给厨师，并在厨师烹饪食物的同时为其他顾客服务。厨师根据某种工作表了解顾客的菜单。然而，厨师可能按照与点菜顺序不同的顺序进行烹饪，以便更有效地利用可用的资源，如炉子、壶或平底锅。烹饪好以后，厨师把做好的菜和菜单放在柜台上，侍应生可以在柜台取到食物上给顾客。

12. 结论

主动对象模式提供了如下**优点**：

增强了应用程序的并发性，简化了同步的复杂性。通过允许客户机线程和异步方法同时执

行而增强了应用程序的并发性。通过使用调度程序而简化了同步的复杂性，调度程序计算同步约束，以保证根据其状态对服务者进行串行访问。

395 *透明地应用可用的并行性*。如果硬件和软件平台能有效地支持多CPU，那么该模式可以允许多个主动对象并行地执行，是否并行执行仅仅取决于它们的同步约束。

方法的执行顺序可以和方法的调用顺序不同。异步调用的方法是根据由方法的哨兵和调度策略定义的同步机制而执行的。因此，方法执行的顺序可以与方法调用的顺序不同。这种分离有助于提高应用程序的性能和灵活性。

然而，主动对象模式也有一些**不足**之处：

性能开销。根据主动对象的调度程序的实现方式不同（例如在用户空间或是内核空间[SchSu95]中），当调度和执行主动对象方法调用时可能引起语境切换、同步化和数据移动开销。一般来说，主动对象模式最适用于相对粗粒度的对象。相反，如果对象是细粒度的，主动对象的性能开销可能会比相关的并发模式（如监视器对象）更多。

复杂的调试。因为各种主动对象的调度程序和底层的操作系统线程调度程序的并发性和不确定性，所以很难调试使用主动对象模式的程序。特别是，方法请求哨兵确定了执行的顺序，但可能很难理解和调试这些哨兵的行为。不正确定义的哨兵可能导致饿死，也就是某些方法调用可能永远不会被执行的情况。另外，程序调试器可能不会充分支持多线程应用程序。

参见

监视器对象模式确保了任一时刻在线程安全的被动对象中仅有一个方法执行，而不管并发调用对象方法的线程有多少。一般来说，监视器对象比主动对象更有效，因为它们需要的语境切换和数据移动开销要少。然而，很难使用监视器对象模式在客户机和服务器线程间增加一个分布边界。

396 比较“*已解决的例子*”一节中的主动对象模式的解决方案和监视器对象模式的解决方案是很有指导意义的。两种解决方案具有类似的总体应用体系结构。特别是，`Supplier_Handler`和`Consumer_Handler`实现几乎相同。

二者之间的主要区别是主动对象模式中的`Message_Queue`支持复杂的方法请求排队和调度策略。同时，因为主动对象在客户机线程之外的线程中执行，因此在某些情况下可以使用主动对象通过异步执行多个操作来改善应用程序的总体并发性。在这些操作完成后，客户机可以通过前景获得操作的结果[Hal85][LS88]。

另一方面，当使用监视器对象模式实现时，`Message_Queue`自身会比使用主动对象模式更容易编程实现，并且往往更高效。

反应器模式负责多路分解和分配多个事件句柄，当可以启动一个操作而不发生阻塞时，这些句柄被触发。反应器模式通常代替主动对象模式将回调操作调度给被动对象。主动对象也可以和反应器结合，形成半同步/半异步模式。

半同步/半异步模式将系统中的同步I/O同异步I/O分离，简化了并发编程的工作而又不降低执行效率。这种模式的变体用主动对象模式来实现同步任务层，用反应器模式实现异步任务层，用生产者-消费者模式[Lea99a]（如管道和过滤器模式[POSA1]的变体，或者监视器模式）来实现排队层。

命令处理器模式[POSA1]将发出请求和执行请求分离。命令处理器对应于主动对象模式的调度程序，管理作为命令[GoF95]实现的未执行的服务请求。命令由供应者执行，供应者与服务者对应。然而，命令处理器模式并不关注并发性。实际上，客户机、命令处理器和供应者常常处在同一控制线程中。类似地，没有代理将服务者提供给客户机。客户机可以创建命令，并将它们直接发送给命令处理器。

397

代理者模式[POSA1]定义了许多与主动对象模式相同的组件。特别是，客户机通过代理访问代理者，而服务器通过服务者实现远程对象。代理者和主动对象的一个不同点是：在代理者模式中，代理和服务者之间存在一个分布边界，而在主动对象模式中代理和服务者之间存在线程边界。另一个不同点是主动对象通常只有一个服务者，而代理者可以有多个服务者。

致谢

Greg Lavender[PLoPD2]首先提出将主动对象作为模式文档化。Ward Cunningham对主动对象模式的这一版本的形成提供了帮助。Bob Laferriere 和 Rainer Blome提供了有用的建议，使模式“实现”一节的内容更加清晰。感谢Doug Lea在[Lea99a]中提供的许多内幕。

398

5.2 监视器对象

监视器对象（Monitor Object）设计模式使并发方法的执行同步化，以确保任一时刻仅有一个方法在对象内运行。它也允许对象方法相互协调，调度方法的执行顺序。

1. 别名 线程安全被动对象（thread-safe passive object）。

2. 例子

重新考虑主动对象模式中描述的通信网关的设计。[⊖] 如图5-8所示。

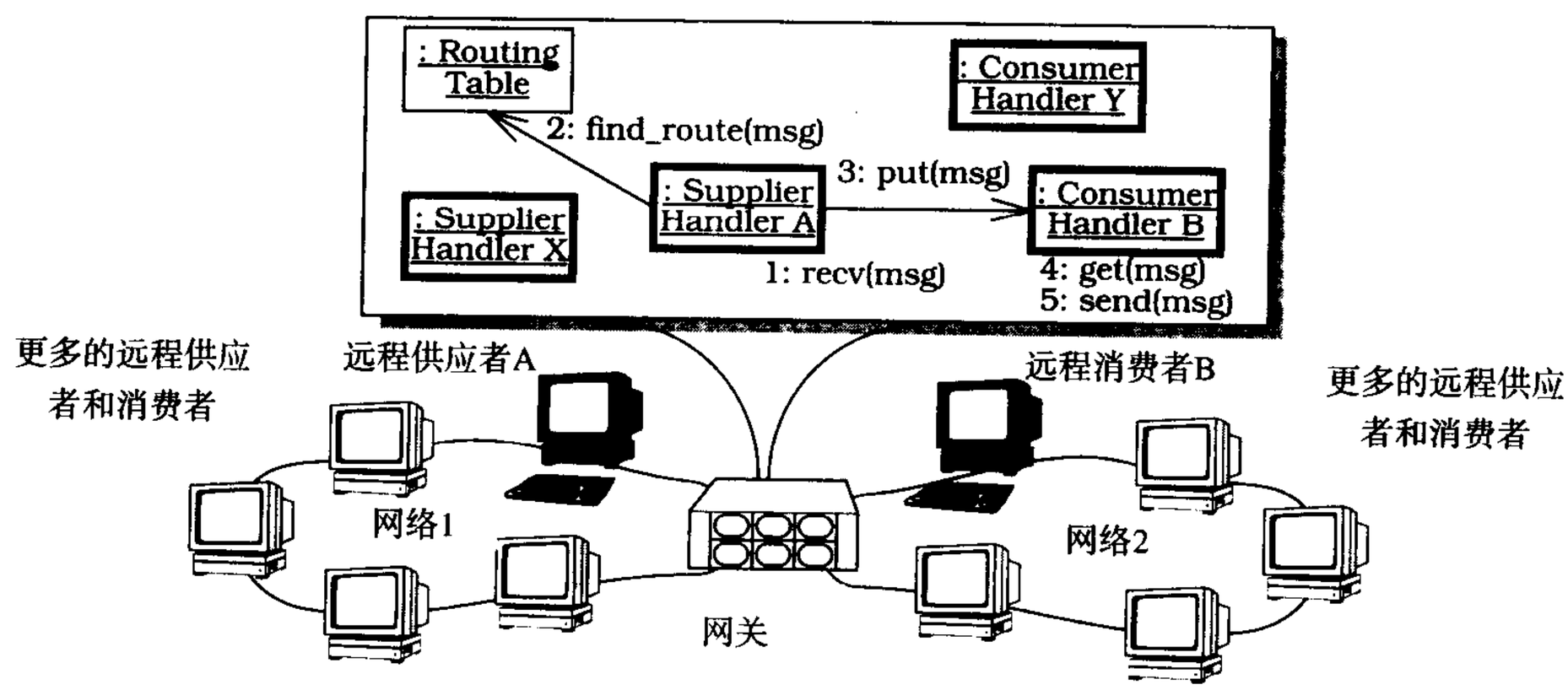


图 5-8

⊖ 为了对网关和它的相关组件进行深入讨论，建议在阅读本模式前先阅读主动对象模式。

网关进程是一个协调器（mediator）[GoF95]，包括多个供应者和消费者处理程序对象。这些对象在相互独立的线程中运行，从一个或多个远程供应者向一个或多个远程消费者发送消息。一个供应者处理程序线程接收从远程供应者发来的消息后，它使用消息中的地址域确定相应的消费者处理程序。然后处理程序线程将消息分发给它的远程消费者。

399

当供应者和消费者驻留在不同的主机中时，网关使用一个面向连接的协议（如TCP[Ste93]），提供可靠的消息发送和端对端流控制。流控制是一个协议机制，在发送者产生消息的速度比接收者能够处理的速度快时对发送者进行阻塞。但是，整个网关在等待流控制中止正在进行的TCP连接时，不应阻塞。特别地，流入的TCP连接应该继续得到处理并且消息应该继续通过任何非流控制的TCP连接来发送。

为了使阻塞最少，每个消费者处理程序可以包含一个线程安全的消息队列。每个队列对它从供应者处理程序线程接收的新路由消息进行缓冲。这种设计将网关进程中的供应者处理程序线程和消费者处理程序线程分离，这样所有的线程可以并发运行，并可以在不同的TCP连接中发生流控制时独立地阻塞。

实现线程安全消息队列的一种方法是应用主动对象模式将调用方法的线程和执行方法的线程分离。然而，如果模式引入的整个基础设施不是必要的，使用主动对象就可能不太合适。例如，一个消息队列的入队列和出队列方法可能不需要复杂的调度策略。在这种情况下，实现主动对象模式方法请求、调度程序和激活表等参与者可能引起不必要的性能开销和编程负担。

相反，线程安全的消息队列的实现必须能有效地避免不必要的性能降低。为了避免供应者和消费者处理程序实现的紧密耦合，实现机制对供应者处理程序的实现者也应当是透明的。否则，独立地改变任何一种实现都可能变得极其复杂。

3. 语境

并发访问同一对象的多个控制线程。

4. 问题

许多应用程序包含其方法被多个客户机线程并发调用的对象。这些方法通常修改其对象的状态。因此，为了使这些并发应用程序能正确地执行，有必要对对象的访问进行同步和调度。

400

对于这个问题，必须考虑四个强制条件：

- 为了事务分离并避免对象状态不受控制地修改，面向对象编程人员习惯于只通过对象的接口方法访问对象。扩展这种面向对象编程模型以避免对象的数据不受控制的并发修改（称为竞争条件）相对比较直接。因此对象的接口方法应该定义它的同步边界，在同一对象中某一时刻仅有一个方法能处于活动状态。
- 如果客户机必须显式地获取和释放低层同步机制，如信号灯、互斥或条件变量[IEEE96]，那么并发应用程序将更难进行编程。因此，对象应该负责确保它们需要同步的任何方法被透明地串行化，而不需要客户机的显式介入。
- 如果一个对象的方法在执行时必须阻塞，它们应该能够自愿放弃它们的控制线程，这样其他客户机线程调用的方法就可以访问这个对象。这种属性有助于防止死锁并能够利用硬、软件平台提供的并发机制。

- 一个方法自愿放弃它的控制线程后，它必须让对象处于稳定状态，即必须保持与对象有关的不变式。类似地，只有在对象处于稳定状态时才可以在对象内恢复一个方法的执行。

5. 解决方案

将对象方法的访问同步化，以便一次仅有一个方法可以执行。

细节：对于每个被多个客户机线程并发访问的对象，把它定义为一个监视器对象。客户机只能通过它的同步化方法访问监视器对象定义的函数。为了防止对象内部状态的竞争条件，一次只能有一个同步化方法能在监视器对象内运行。为了将对象状态的并发访问串行化，每个监视器对象都包含一个监视器锁。同步化方法可以基于一个或多个与监视器对象关联的监视器条件确定它们挂起和恢复执行的环境。 401

6. 结构

监视器对象模式中有四个参与者：

监视器对象提供一个或多个方法。为了保护监视器对象的内部状态不被非受控地修改和防止竞争条件，所有的客户机必须仅通过这些方法访问监视器对象。每个方法在调用它的客户机线程中执行，因为监视器对象没有自己的控制线程[⊖]。

同步化方法实现由监视器对象提供的线程安全的功能。为了防止竞争条件，一次只能有一个同步化方法可以在监视器对象中执行。不管并发调用对象的同步化方法的线程数量或者对象类中同步化方法的数量是多少，都可采用这种原则（如图5-9）。

➡网关应用程序中消费者处理程序的消息队列可以作为监视器对象实现，需要将它的put()和get()操作转化为同步化方法。这种设计确保了路由消息可以被多个线程并发地插入或删除，而不会破坏队列的内部状态。 ❑

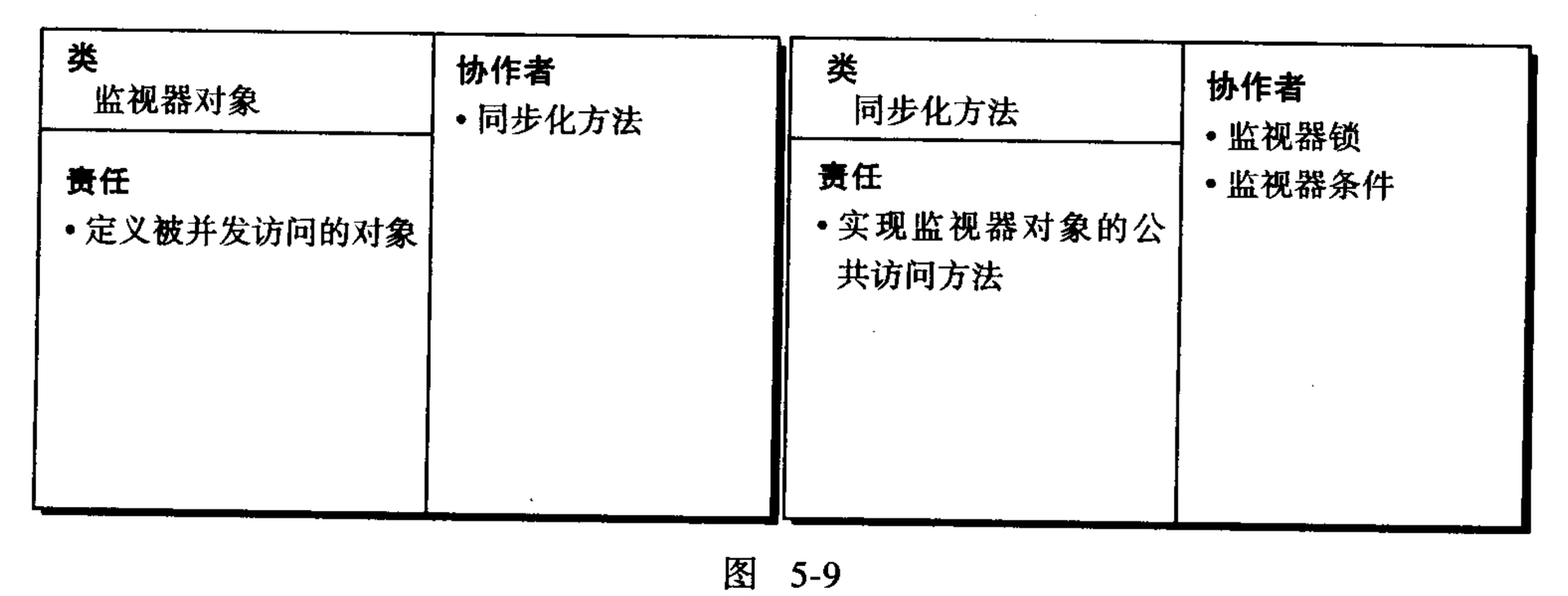


图 5-9

每个监视器对象都包含有自己的监视器锁。同步化方法基于完全对象原则使用该锁将方法调用串行化。每个同步化的方法在进入或退出对象时，必须请求和释放对象的监视器锁。这个协议确保了不论何时同步化方法执行操作访问或修改对象的状态，监视器锁都能被保持。 402

监视器条件。运行在不同线程中的多个同步化方法可以通过与监视器对象关联的监视器条件彼此等待和通知，从而协作调度它们的执行顺序。同步化方法将它们的监视器锁与监视器条

⊖ 相反地，主动对象具有自己的控制线程。

件结合使用，确定它们挂起或恢复处理时所处的环境（如图5-10）。

类 监视器锁	协作者
责任 • 确保在给定时间，监视器对象中仅有一个同步化方法可以执行	

类 监视器条件	协作者 • 监视器锁
责任 • 允许同步化方法确定其挂起或恢复处理的环境	

图 5-10

➥在网关应用程序中，可以用POSIX互斥[IEEE96]来实现消息队列的监视器锁。可以用一对POSIX条件变量来实现消息队列的非空和非满监视器条件：

- 当消费者处理程序线程试图从一个空消息队列中删除一个路由消息时，队列的get()方法必须能自动释放监视器锁，并将自己挂起，等待非空的监视器条件。get()方法保持挂起，直到队列不为空，即供应者处理程序线程向队列插入了一个消息。
- 当供应者处理程序线程试图向一个满队列插入数据时，队列的put()方法必须自动地释放监视器锁并将自己挂起，等待非满的监视器条件。put()方法保持挂起，直到队列不为满，即一个消费者处理程序从消息队列中删除一个消息。

403

注意，非空和非满的监视器条件都共享同一监视器锁。 ❑

用类图5-11说明监视器对象模式的结构：

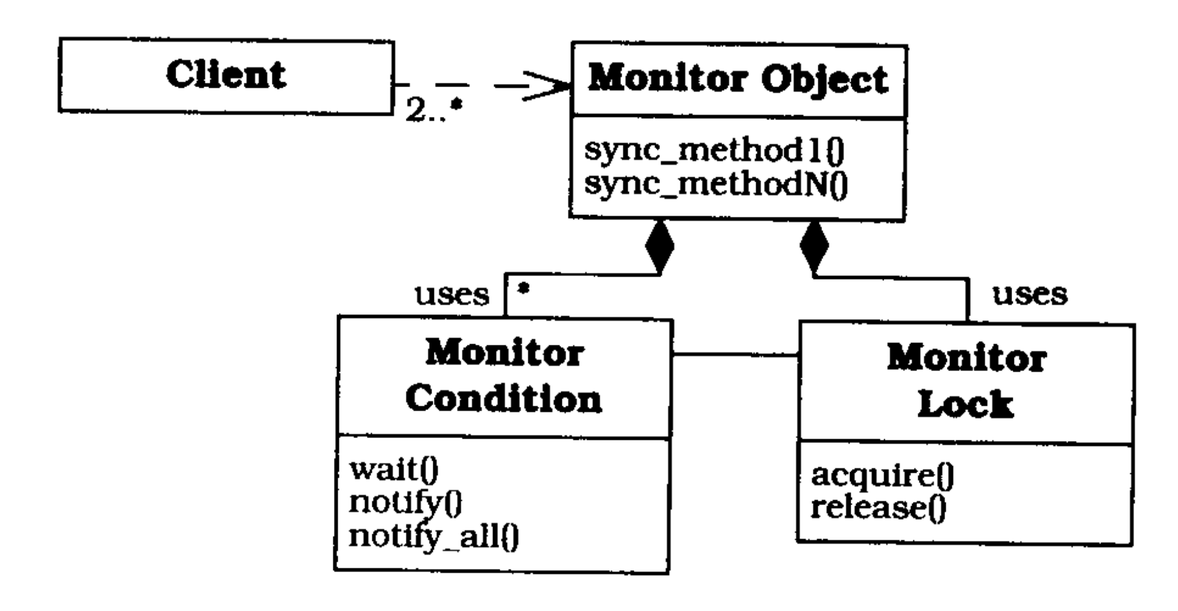


图 5-11

7. 动态特性

监视器对象模式中参与者之间的协作分为四个阶段（如图5-12）：

- 同步化方法的调用和串行化。当客户机线程T_1调用监视器对象上的同步化方法时，该方法必须首先获得对象的监视器锁。当线程T_2中的另一个同步化方法正在监视器对象中执行时，就不能获得监视器锁。在这种情况下，客户机线程T_1会一直阻塞，直到同步化方法获得锁。一旦T_1调用的同步化方法执行完毕，监视器锁就被释放，这样另一线程调用的其他同步化方法就可以访问监视器对象了。

- 同步化方法线程挂起。如果一个同步化方法必须阻塞或因其他原因不能立即处理，它可以等待监视器条件。这使同步化方法暂时“离开”监视器对象[Hoare74]。监视器对象的实现负责确保在切换到另一线程前的状态是稳定的。一个同步化方法离开监视器对象后，客户机线程挂起等待监视器条件，并且监视器锁被操作系统线程调度程序自动释放。另一个线程中的另一个同步化方法此时就可以在监视器对象中执行了。
- 监视器条件通知。同步化方法可以通知一个监视器条件。此操作唤醒一个同步化方法的线程，该同步化方法在此之前挂起等待该监视器条件。该操作也可以通知所有其他在此之前将线程挂起、等待该监视器条件的同步化方法。在这种情况下，所有线程都被唤醒，一次只有其中一个线程可以获得监视器锁并在监视器对象中运行。 404
- 同步化方法线程恢复。一旦挂起的同步化方法线程得到通知，它就可以在等待监视器条件的地方恢复执行。操作系统线程调度程序隐式地完成了这种恢复操作。在得到通知的线程“重新进入”监视器对象并在同步化方法中恢复其执行之前，可以自动地重新获得监视器锁。

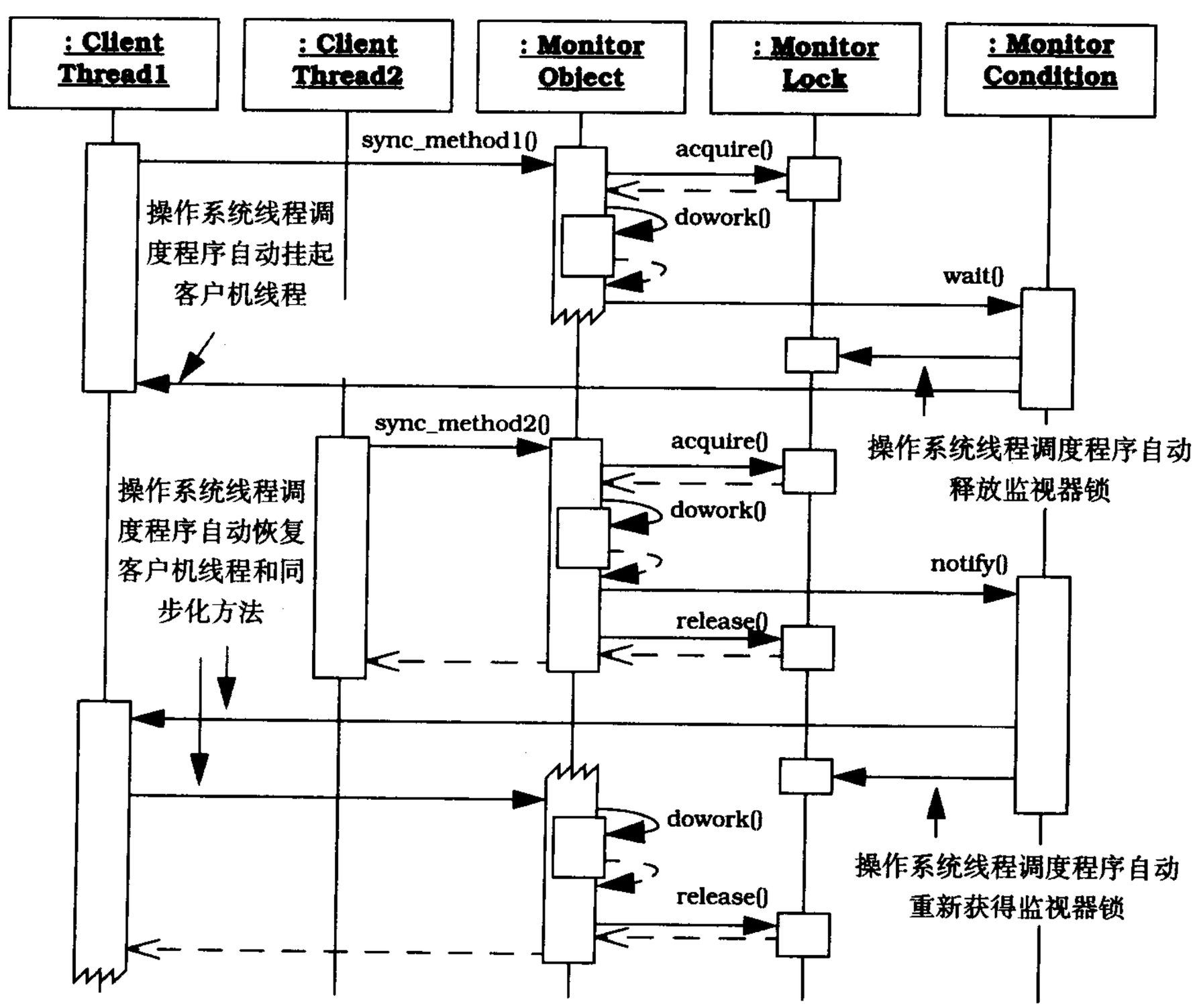


图 5-12

405

8. 实现

用四个活动说明如何实现监视器对象模式。

(1) 定义监视器对象的接口方法。监视器对象的接口向客户机提供一系列方法。接口方法通常是同步的。就是说，在一个监视器对象中一次只有一个方法可以被线程执行。

➡在网关例子中，每个消费者处理程序包含一个消息队列和一个TCP连接。消息队列可以作为一个监视器对象实现，用于对从供应者处理程序线程接收到的消息进行缓冲。在消费者处理程序线程到它们的远程消费者的TCP连接上遇到流控制时，这种缓冲操作有助于防止整个网关处理发生阻塞。如下的C++类为消息队列监视器对象定义了接口：

```
class Message_Queue {
public:
    enum { MAX_MESSAGES = /* ... */; };

    // The constructor defines the maximum number
    // of messages in the queue. This determines
    // when the queue is 'full.'
    Message_Queue (size_t max_messages = MAX_MESSAGES);

    // Put the <Message> at the tail of the queue.
    // If the queue is full, block until the queue
    // is not full.
    /* synchronized */ void put (const Message &msg);

    // Get the <Message> from the head of the queue
    // and remove it. If the queue is empty,
    // block until the queue is not empty.
    /* synchronized */ Message get ();

    // True if the queue is empty, else false.
    /* synchronized */ bool empty () const;

    // True if the queue is full, else false.
    /* synchronized */ bool full () const;
private:
    // ... described later ...
};
```

Message_Queue监视器对象接口提供四个同步化方法。empty()和full()方法是客户机可以用来区分如下三种内部队列状态的谓词：空、满和不空不满。put()和get()方法将消息入队列和出队列，当队列分别为满和空时会阻塞。 ❑

406

(2) 定义监视器对象的实现方法。监视器对象通常包含内部实现方法，用这种同步化的接口方法来实现对象的功能。这种设计有助于将核心的监视器对象功能特性与它的同步和调度逻辑分离开。这种设计也有助于避免对象内的死锁和不必要的加锁开销。

以下两个基于线程安全接口模式的约定可以用来指导监视器对象接口和实现方法的事务分离：

- 接口方法仅仅获得和释放监视器锁以及等待或通知具体的监视器条件。另外它们还将控制转发给实现监视器对象功能特性的实现方法。
- 仅仅当被接口方法调用时，实现方法才执行。它们不获得和释放监视器锁，也不明确地等待或通知监视器条件。

类似地，与线程安全接口模式一致，实现方法不应调用任何在类接口中定义的同步化方法。这种限制有助于避免对象内方法死锁和不必要的同步开销。

➡在网关中，Message_Queue类定义了四个实现方法：put_i()、get_i()、empty_i()和full_i()：

```
class Message_Queue {
public:
    // ... See above ...
private:
    // Put the <Message> at the tail of the queue, and
    // get the <Message> at its head, respectively.
    void put_i (const Message &msg);
    Message get_i ();

    // True if the queue is empty, else false.
    bool empty_i () const;

    // True if the queue is full, else false.
    bool full_i () const;
};
```

❑ 407

实现方法通常是非同步化的。当涉及到阻塞的调用时必须小心，因为调用实现方法的接口方法可能已经获得了监视器锁。因此一个被阻塞的拥有锁的线程可能会无限期拖延整个程序的处理。

(3) 定义监视器对象的内部状态和同步机制。监视器对象包含定义其内部状态的数据成员。这种状态必须受到保护，免受由非同步并发访问引起的竞争条件的破坏。因此，监视器对象包含一个监视器锁使它的同步化方法被串行执行，以及一个或多个监视器条件，用于调度监视器对象中同步化方法的执行。一般以下每种情况都对应一个独立的监视器条件：

- 第一种情况，同步化方法必须将它们的处理挂起，等待某些状态改变事件发生。
- 第二种情况，同步化方法必须恢复其他线程，这些线程的同步化方法将其自身挂起，等待监视器条件。

监视器对象的方法实现负责确保在释放它的锁之前对象处于稳定状态。可以用不变式描述稳定状态，就像在消息队列中需要通过有效的指针连接所有节点一样。只要监视器对象方法等待相应的状态变量，不变式都应成立。

类似地，当监视器对象被通知而且操作系统线程调度程序决定继续恢复其线程时，监视器对象的方法实现负责确保在处理前不变式确实成立。这种检查是需要的，因为在通知操作和恢复操作期间其他线程可能改变了对象的状态。因此，监视器对象必须确保在允许同步化方法恢复执行前，不变式成立。

监视器锁可以使用互斥来实现。当持有互斥的线程执行临界区的代码时，互斥使协作的线程能相互等待。可以使用条件变量[IEEE96]实现监视器条件。条件变量可以使线程等待，直到特定事件发生或者复杂的条件表达式获得一个特定的稳定状态。条件表达式通常访问对象或线程间共享的状态变量，它们可以用来实现保护挂起（Guarded Suspension）模式[Lea99a]。 408

➡在网关例子中，Message_Queue定义了它的内部状态，说明如下：

```
class Message_Queue {
    // ... See above ....
private:
    // ... See above ...

    // Internal Queue representation omitted, could be a
```

```
    // circular array or a linked list, etc.. ...

    // Current number of <Message>s in the queue.
    size_t message_count_;

    // The maximum number <Message>s that can be
    // in a queue before it's considered `full.'
    size_t max_messages_;

    // Mutex wrapper facade that protects the queue's
    // internal state from race conditions during
    // concurrent access.
    mutable Thread_Mutex monitor_lock_;

    // Condition variable wrapper facade used in
    // conjunction with <monitor_lock_> to make
    // synchronized method threads wait until the queue
    // is no longer empty.
    Thread_Condition not_empty_;

    // Condition variable wrapper facade used in
    // conjunction with <monitor_lock_> to make
    // synchronized method threads wait until the queue
    // is no longer full.
    Thread_Condition not_full_;
};
```

Message_Queue监视器对象定义了三种类型的内部状态：

- 队列表示数据成员。这些数据成员定义了内部的队列表示。该表示以循环队列或链表形式存储队列的内容，还有确定队列空、满或不空不满的内部状态所需的簿记信息。只有put_i()、get_i()、empty_i()和full_i()实现方法才能操纵内部队列表示。
- 监视器锁数据成员。Message_Queue的同步化方法用monitor_lock_来串行化对队列内部表示状态的访问。要改变监视器对象的状态，就必须持有监视器对象的锁，以确保满足不变式。要使用在包装器外观模式中定义的与平台无关的Thread_Mutex类实现这种监视器锁。
- 监视器条件数据成员。put()和get()同步化方法使用监视器条件is_full_和is_empty_，当Message_Queue离开它的满和空边界条件时分别挂起和恢复自己的同步化方法。这些监视器条件使用在包装器外观模式中定义的与平台无关的Thread_Condition类实现。 ❑

409

(4) 实现监视器对象的所有方法和数据成员。如下两个子活动可以用来实现上面定义的所有监视器对象的方法和内部状态。

(4.1) 初始化数据成员。该子活动将对象特定的数据成员，以及监视器锁和任何监视器条件初始化。

➡Message_Queue的构造函数建立一个空队列，并对监视器条件not_empty_和not_full_初始化：

```
Message_Queue::Message_Queue (size_t max_messages)
    :   not_full_ (monitor_lock_),
        not_empty_ (monitor_lock_),
        max_messages_ (max_messages),
        message_count_ (0) { /* ... */ }
```

在这个例子中，两个监视器条件共享同一monitor_lock_。这种设计确保Message_Queue状态（如message_count_）在多个线程试图对一个队列同时put()和get()消息时正确地被串行化以防止竞争条件破坏不变式。 ❑ 410

(4.2) 应用线程安全接口模式。在该子活动中，按照线程安全接口模式实现接口方法和实现方法。

➡在Message_Queue实现中，两对接口和实现方法检查队列是否为空（为空则意味着Message_Queue不包含消息），或满（为满则意味着包含max_messages_）。首先列出接口方法：

```
bool Message_Queue::empty () const {
    Guard<Thread_Mutex> guard (monitor_lock_);
    return empty_i ();
}

bool Message_Queue::full () const {
    Guard<Thread_Mutex> guard (monitor_lock_);
    return full_i ();
}
```

这些方法说明了一个简单的线程安全接口模式的例子。它们使用定界加锁惯用法获得和释放监视器锁，然后立即转发相应的实现方法：

```
bool Message_Queue::empty_i () const {
    return message_count_ == 0;
}

bool Message_Queue::full_i () const {
    return message_count_ == max_messages_;
}
```

按照线程安全接口模式，假定这些实现方法拥有monitor_lock_，因此它们仅仅检查队列的边界状态。

put()方法在队列尾插入一个新的Message，它是主动对象模式定义的一个类。它是一个同步化方法，说明了线程安全接口模式的一种更复杂的使用。

```
void Message_Queue::put (const Message &msg) {
    // Use the Scoped Locking idiom to
    // acquire/release the <monitor_lock_> upon
    // entry/exit to the synchronized method.
    Guard<Thread_Mutex> guard (monitor_lock_);

    // Wait while the queue is full.
    while (full_i ()) {
        // Release <monitor_lock_> and suspend the
        // calling thread waiting for space in the queue.
        // The <monitor_lock_> is reacquired
        // automatically when <wait> returns.
        not_full_.wait ();
    }

    // Enqueue the <Message> at the tail.
    put_i (msg);

    // Notify any thread waiting in <get> that
    // the queue has at least one <Message>.
```

411

```
    not_empty_.notify ();
} // Destructor of <guard> releases <monitor_lock_>.
```

注意，这个公共的同步化的put()方法是如何只执行同步和调度逻辑的，这种同步和调度逻辑的执行对于串行化对监视器对象的访问以及当队列满时进行等待是必需的。一旦队列中有空间，put()就转到put_i()实现方法。该方法将消息插入队列并更新队列的簿记信息。并且，put_i()方法不是同步化的，因为put()方法如果不首先获得monitor_lock_就不会调用它。同样，put_i()方法不需要检查队列是否满，因为只有full_i()返回真，才能调用它。

get()方法删除队首的消息并将它返回给调用者：

```
Message Message_Queue::get () {
    // Use the Scoped Locking idiom to
    // acquire/release the <monitor_lock_> upon
    // entry/exit to the synchronized method.
    Guard<Thread_Mutex> guard (monitor_lock_);

    // Wait while the queue is empty.
    while (empty_i ()) {
        // Release <monitor_lock_> and suspend the
        // calling thread waiting for a new <Message> to
        // be put into the queue. The <monitor_lock_> is
        // reacquired automatically when <wait> returns.
        not_empty_.wait ();
    }

    // Dequeue the first <Message> in the queue
    // and update the <message_count_>.
    Message m = get_i ();
    // Notify any thread waiting in <put> that the
    // queue has room for at least one <Message>.
    not_full_.notify ();
    return m;

    // Destructor of <guard> releases <monitor_lock_>.
}
```

412

同样要注意的是，将出队列的功能特性转移给get_i()实现方法时，同步化get()接口方法是如何实现同步和调度逻辑的。 ❑

9. 已解决的例子

网关本质上包含两个类的实例，这两个类是Supplier_Handler和Consumer_Handler。它们分别作为远程供应者和消费者的本地代理[GoF95][POSA1]。每个Consumer_Handler包含一个使用监视器对象模式实现的线程安全的Message_Queue对象。这种设计将供应者处理程序和消费者处理程序线程分开，这样它们就可以并发运行以及独立地阻塞。并且，通过向消息队列监视器对象中嵌入和自动化同步机制，可以保护消息队列的内部状态不被破坏，保持不变式并使客户机不受低层同步问题的困扰。

Consumer_Handler定义如下：

```
class Consumer_Handler {
public:
    // Constructor spawns a thread and calls <svc_run>.
```

```
    Consumer_Handler ();

    // Put <Message> into the queue monitor object,
    // blocking until there's room in the queue.
    void put (const Message &msg) {
        message_queue_.put (msg);
    }
private:
    // Message queue implemented as a monitor object.
    Message_Queue message_queue_;

    // Connection to the remote consumer.
    SOCK_Stream connection_;

    // Entry point to a distinct consumer handler thread.
    static void *svc_run (void *arg);
};
```

每个Supplier_Handler都在自己的线程中运行，从远程供应者接收消息，并将消息发送给指定的远程消费者。通过检查每条消息中的地址域来进行路由，该地址域充当路由表的关键字，路由表将关键字映射到Consumer_Handler，如图5-13所示。 413

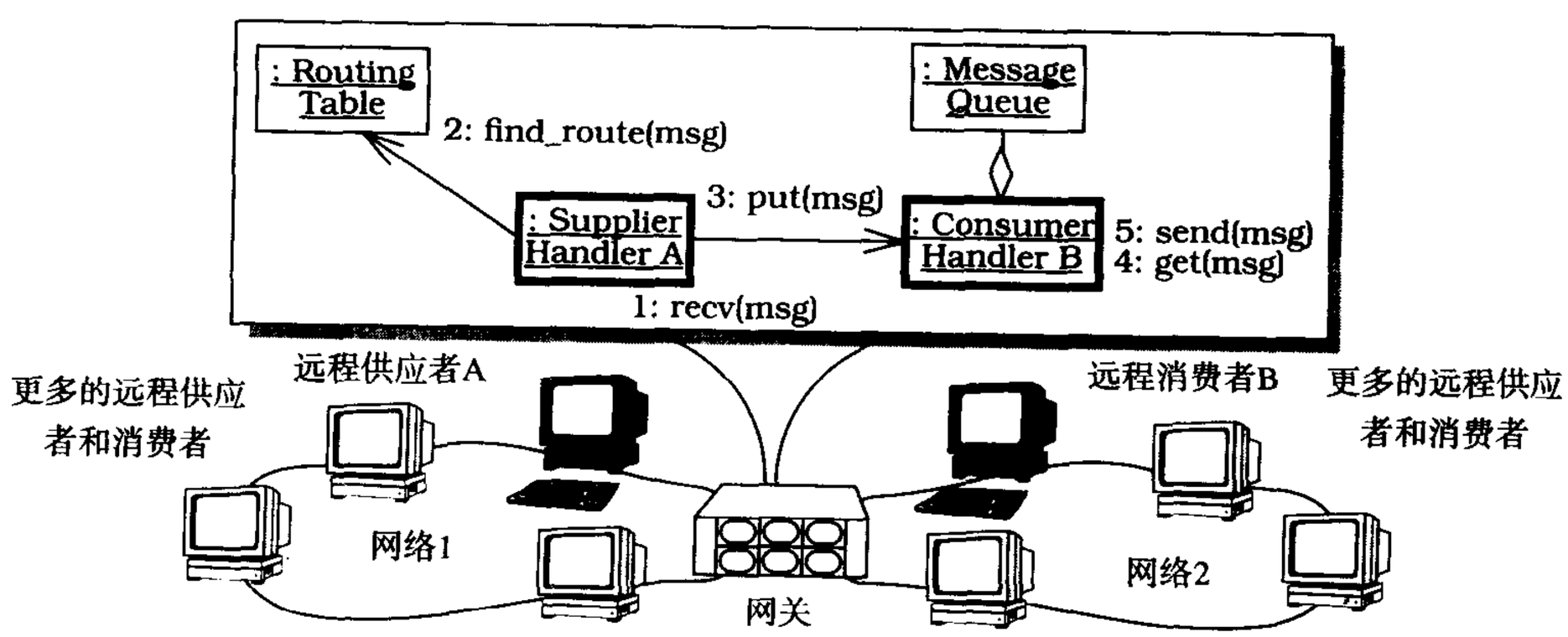


图 5-13

每个Consumer_Handler负责通过put()方法接收供应者的消息并将消息存储到它的Message_Queue监视器对象：

```
void Supplier_Handler::route_message (const Message &msg)
{
    // Locate the appropriate <Consumer_Handler> based
    // on address information in the <Message>.
    Consumer_Handler *consumer_handler =
        routing_table_.find (msg.address ());

    // Put <Message> into the <Consumer Handler>, which
    // stores it in its <Message Queue> monitor object.
    consumer_handler->put (msg);
}
```

为了处理被Supplier_Handler放入队列中的消息，每个Consumer_Handler使用在包装器外观模式中定义的Thread_Manager类，在它的构造函数中产生一个独立的控制线程：

```
Consumer_Handler::Consumer_Handler () {
    // Spawn a separate thread to get messages from the
    // message queue and send them to the remote consumer.
    Thread_Manager::instance ()->spawn (&svc_run, this);
}
```

414

这个新的Consumer_Handler线程执行svc_run()入口点。这是一个静态方法，它获取由Supplier_Handler线程放入消息队列的路由消息，并将它们通过TCP连接发送给远程消费者：

```
void *Consumer_Handler::svc_run (void *args) {
    Consumer_Handler *this_obj =
        static_cast<Consumer_Handler *> (args);
    for (;;) {
        // Blocks on <get> until next <Message> arrives.
        Message msg = this_obj->message_queue_.get ();
        // Transmit message to the consumer.
        this_obj->connection_.send (msg, msg.length ());
    }
}
```

SOCK_Stream的send()方法可以在Consumer_Handler线程中阻塞。它不会影响其他Consumer_Handler或Supplier_Handler线程的服务质量，因为它不与其他线程共享数据。与此类似，Message_Queue::get()的阻塞不会影响其他线程的服务质量，因为Message_Queue是一个监视器对象。因此，Supplier_Handler可以通过它的put()方法将新消息插入Consumer_Handler的Message_Queue而不会引起无限期阻塞。

10. 变体

定时同步化方法调用。某些应用程序需要“定时的”同步化方法调用。该特性允许这些应用程序设定时间区间，它们愿意在此时间内等待同步化方法进入其监视器对象的临界区。可以使用定时同步化方法调用实现[Lea99a]中描述的阻行（Balking）模式。

➡可以修改前面定义的Message_Queue监视器对象接口，以支持定时同步化方法调用：

```
class Message_Queue {
public:
    // Wait up to the <timeout> period to put <Message>
    // at the tail of the queue.
    void put
        (const Message &msg, Time_Value *timeout =0);

    // Wait up to the <timeout> period to get <Message>
    // from the head of the queue.
    Message get (Time_Value *timeout = 0);
};
```

415

如果timeout等于0，则get()和put()一直阻塞，直到一个消息被插入到Message_Queue监视器对象或从中删除。如果超时值非零并且达到时限，会抛出Timedout异常。客户机必须准备好处理该异常。

下面说明了如何使用实现活动（3）中指出的Thread_Condition条件变量包装器的定时等待特性实现put()方法：

```
void Message_Queue::put
     (const Message &msg, Time_Value *timeout)
          /* throw (Timedout) */ {
     // ... Same as before ...
     while (full_i ())
          not_full_.wait (timeout);
     // ... Same as before ...
}
```

当队列为满时，该“定时的”put()方法释放monitor_lock_并将调用线程挂起，等待队列中出现可用空间或等待由timeout指定的时间消逝。不管是否发生超时，一旦wait()返回，monitor_lock_会自动被重新获得。 ❑

策略化加锁。可以使用策略化加锁模式，使监视器对象的实现更灵活、高效、可重用和健壮。例如，可以使用策略化加锁，用各种类型的监视器锁和监视器条件来配置监视器对象。

➥如下的模板类使用了*类属编程技术*[Aus98]将Message_Queue的同步特征参数化：

```
template <class SYNCH_STRATEGY> class Message_Queue {
private:
     typename SYNCH_STRATEGY::Mutex monitor_lock_;
     typename SYNCH_STRATEGY::Condition not_empty_;
     typename SYNCH_STRATEGY::Condition not_full_;
     // ...
};
```

416

用如下empty()方法来修改同步化方法：

```
template <class SYNCH_STRATEGY>
bool Message_Queue<SYNCH_STRATEGY>::empty () const {
     Guard<SYNCH_STRATEGY::Mutex> guard (monitor_lock_);
     return empty_i ();
}
```

为将与Message_Queue关联的同步特征参数化，可以定义两个具有适当C++特征的类MT_Synch和NULL_SYNCH：

```
class MT_Synch {
public:
     // Synchronization traits.
     typedef Thread_Mutex Mutex;
     typedef Thread_Condition Condition;
};

class Null_Synch {
public:
     // Synchronization traits.
     typedef Null_Mutex Mutex;
     typedef Null_Thread_Condition Condition;
};
```

因此，为了定义线程安全的Message_Queue，可以简单地用MT_Synch策略对它进行参数化：

```
Message_Queue<MT_Synch> message_queue;
```

同时，为了建立一个非线程安全的Message_Queue，可以用如下的Null_Synch策略对它进行参数化：

```
Message_Queue<Null_Synch> message_queue;
```

注意，在C++中使用策略化加锁模式时，通用组件类不可能知道对特定的应用程序该配置哪种类型的同步策略。因此应用实现活动（4.2）中描述的线程安全接口模式就很重要了，这样可以确保对象内方法调用（如put()调用full_i()和put_i()）避免自死锁和使整体加锁开销最小化。

多角色。如果监视器对象实现多角色，每个角色被不同类型的客户机使用，则可以为每个角色引入一个接口。使用扩展接口模式可以使客户机获得他们需要的接口。这种设计有助于事务分离，因为客户机只能看见它自身操作需要的监视器对象的特定方法。该设计特性使监视器对象的演化变得简单了。例如，可以通过提供新的扩展接口而不改变现存的接口来将新服务加入主动对象。因此不需要访问新服务的客户机将不受扩展的影响。

417

11. 已知使用

Dijkstra和Hoare型监视器。Dijkstra[Dij68]和Hoare[Hoare74]定义的称为监视器的编程语言特性将函数和它们的内部变量封装进线程安全模块。为了防止竞争条件，监视器包含一个一次只允许一个函数在监视器中处于活动状态的锁。想暂时离开监视器的函数可以阻塞等待一个条件变量。程序设计语言的编译程序负责生成运行时代码，该代码实现并管理监视器锁和它的条件变量。

Java对象。Java中的主要同步机制是基于Dijkstra/Hoare型监视器的。每个Java对象可以是包含监视器锁和单独监视器条件的一个监视器对象。对于一般的用况，Java监视器的使用是很简单的，因为它们允许线程隐式地通过方法调用接口串行执行，以及通过调用所有对象中定义的wait()、notify()和notifyAll()方法协调它们的行动。

然而，对于更复杂的用况，Java语言简单的结构可能会误导开发者以为并发比实际情况更容易编程实现。特别地，大量使用内部互相依赖的Java线程会产生复杂的内部关系、饿死、死锁和过大的开销。[Lea99a]描述了许多用于处理Java中简单和复杂并发用况的模式。

可以在适当的Java虚拟机（JVM）中以几种方式实现上述的Java语言同步结构。JVM实现者必须在两个实现决定中进行选择：

- 在JVM内部实现Java线程。如果线程是在内部实现的，那么对于操作系统来说JVM看上去是一个单块任务。在这种情况下，当JVM符合Java语言的规范描述时，它可以自主决定何时挂起和继续线程，以及怎样实现线程的调度。
- 将Java线程映射到本地操作系统线程。在这种情况下，Java监视器可以利用底层平台的同步原语和调度特性。

418

内部线程实现的优势是它的平台独立性。然而，它的一个缺点是不能够利用硬件的并行性。因此，越来越多的JVM是用将Java线程映射到本地操作系统线程而实现的。

ACE网关。“已解决的例子”一节中的例子是基于包含在ACE框架[Sch96]中的一个通信网关应用程序的，它用监视器对象简化了并发编程，并改进了使用多处理器时的性能。和作为编程语言特性的Dijkstra/Hoare和Java监视器不同，在网关中被Consumer_Handler使用的Message_Queue是使用监视器对象模式实现的可重用ACE C++组件。虽然C++不提供将监视

器对象作为语言特性的直接支持，单ACE可以通过应用其他模式和惯用法（如在“实现”一节介绍的保护挂起模式（Guarded Suspension）[Lea99a]和定界加锁惯用法）实现监视器模式。

快餐厅。监视器对象模式的一个现实生活中的例子发生在繁忙的快餐厅点菜时。消费者像客户机，等着跟收银员订餐。一次只能有一个消费者与收银员接触。如果订单不能被马上服务，那么该消费者将临时站到一边，这样其他的消费者可以继续订餐。订单准备完毕时，该消费者重新进入队列的前面，从收银员的手里领取食物。

12. 结论

监视器对象模式提供了两个**优点**：

简化了并发控制。监视器对象模式为在协作线程间共享对象提供了简明的编程模型。例如，对象同步对应于方法调用。类似地，在调用监视器对象上的方法时客户机不需要考虑并发控制。这样可以从被动对象相对直接地建立一个监视器对象，被动对象从它的调用者那里借用控制线程执行它的方法。 419

简化了方法执行的调度。同步化方法使用它们的监视器条件确定它们可能挂起或恢复它们的执行以及协作监视器对象的环境。例如，在发生特别复杂的情况时，方法可以将它们自己挂起并等待通知，无需使用低效的轮询。该特性使监视器对象可以协作调度在相互独立的线程中的方法。

监视器对象模式有如下四个**不足**：

当串行化多个线程对监视器对象的访问时，由于争用增加，单监视器锁的使用会限制可扩展性。

监视器对象的功能特性和它的同步机制的紧密耦合了导致复杂的可扩展性语义。通过独立的调度程序的参与，将主动对象的功能特性与它的同步策略分离是相对直接的。但是，监视器对象的同步化和调度逻辑通常和其方法的功能特性紧密耦合。这种耦合常常使监视器对象比主动对象更高效。不过，同时也很难在不修改监视器对象的方法实现的情况下改变它们的同步策略或机制。

由于继承异常的问题[MWY91]，也很难透明地继承监视器对象。当子类需要不同的同步机制时，该问题抑制了对同步化方法实现的重用。降低监视器对象中同步和功能特性之间的耦合度的方法之一是使用面向特征的编程[KLM+97]或如“实现”和“变体”部分所示的，使用策略化加锁和线程安全接口模式。

嵌套的监视器锁定。该问题与前一缺点类似。当监视器对象嵌套在其他的监视器对象中时会产生这类问题。 420

考虑如下两个Java类：

```
class Inner {
    protected boolean cond_ = false;

    public synchronized void awaitCondition () {
        while (!cond)
            try { wait (); }
            catch (InterruptedException e) { }
            // Any other code.
    }
```

```
    public synchronized void notifyCondition (boolean c){
        cond_ = c;
        notifyAll ();
    }

class Outer {
    protected Inner inner_ = new Inner ();

    public synchronized void process () {
        inner_.awaitCondition ();
    }
    public synchronized void set (boolean c) {
        inner_.notifyCondition (c);
    }
}
```

这段代码说明了在Java中嵌套监视器锁定问题[JS97a]的一般形式。当一个Java线程在监视器的等待队列中阻塞时，除了队列中对象的锁，保留所有其他的锁。

假设线程T_1产生一个对Outer.process()的调用，并且在Inner.awaitCondition()中的wait()调用中被阻塞，会发生什么？在Java中，Inner和Outer类不共享它们的监视器锁。在waitCondition()调用中的wait()语句因此会释放Inner监视器，同时保持Outer监视器。而另一个线程T_2就不能获得Outer监视器，因为它被同步process()方法加锁了。结果Outer.set不能设定Inner.cond_为真，并且T_1将永远阻塞于wait()中。

可以通过在多监视器条件间共享一个监视器锁而避免嵌套的监视器阻塞。在基于POSIX条件变量[IEEE96]的监视器对象模式的实现中很容易实现这一点。而在Java中非常困难，因为它的简单并发和同步模型将监视器锁和其他监视器对象紧密联系在一起。关于Java中避免嵌套监视器阻塞的Java惯用法在[Lea99a][JS97a]中有描述。

421

参见

监视器对象模式类似于代码加锁（Code Locking）模式[McK95]的面向对象的版本，它确保了代码区域的串行性。在监视器对象模式中，代码区域是同步化方法实现。

监视器对象模式与主动对象模式有几个共同属性。例如，两个模式都可以同步以及调度并发调用对象的方法。然而，这里有两个关键的区别：

- 主动对象在它的客户机线程之外的线程执行它的方法，而监视器对象在它的客户机线程内执行它的方法。因此，主动对象可以进行更复杂的（虽然开销更大）调度，重新安排方法执行的顺序。
- 监视器对象常常将它们的同步逻辑与方法的功能特性更紧密地耦合。相反，因为有一个独立的调度程序，所以更容易将主动对象的功能特性与它的同步策略分离。

比较在“已解决的例子”一节的监视器对象模式解决方案和主动对象模式的解决方案具有指导意义。两种解决方案有相似的总体应用程序体系结构。特别是，Supplier_Handler和Consumer_Handler实现几乎一样。主要的区别是在使用监视器对象模式而不是主动对象模式实现时，Message_Queue更容易编程同时更有效。

然而，如果需要一个更成熟的排队策略，主动对象模式可能更适合。类似地，因为主动对

象在客户机线程之外的线程中执行，因此在有些情况下主动对象可以通过异步执行多个操作来改善总体应用程序的并发性。这些操作完成后，客户机可以通过前景[Hal85][LS88]获得它们的结果。 422

5.3 半同步/半异步

半同步/半异步（Half-Sync/Half-Async）体系结构模式将并发系统中的异步和同步服务处理分离，简化了编程，同时又没有降低性能。该模式介绍了两个通信层，一个用于异步服务处理，另一个用于同步服务处理。

1. 例子

性能敏感的并发应用程序（如电信交换系统和航空电子设备任务计算机）会执行混合的同步和异步处理，以协调不同类型的应用程序、系统服务以及硬件。系统级的软件（如操作系统）也有类似的特征。

BSD UNIX操作系统[MBKQ96][Ste98]是一个协调标准因特网应用程序服务（如FTP、INETD、DNS、TELNET、SMTP和HTTPD）和硬件I/O设备（如网络接口、磁盘控制器、用户终端和打印机）之间通信的并发系统的例子。如图5-14所示。

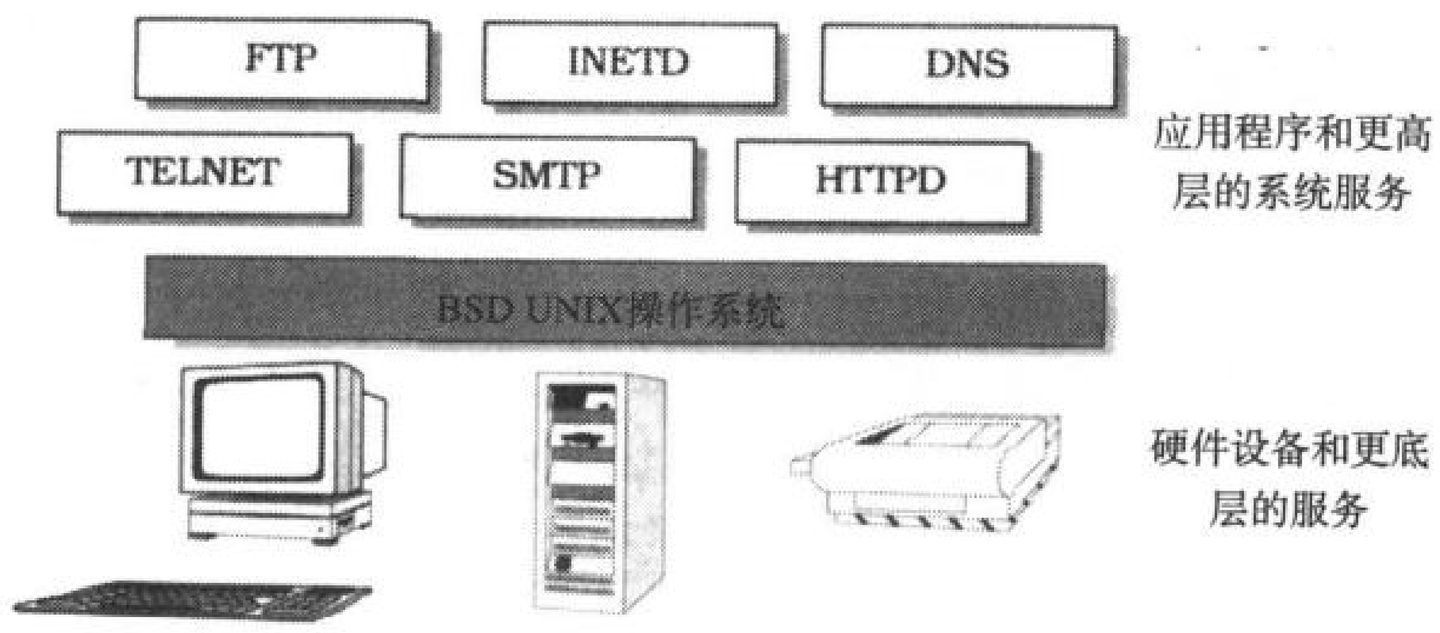


图 5-14

BSD UNIX操作系统异步地处理一些服务，以使性能最大化。例如，BSD UNIX 内核中的协议处理是异步运行的，因为I/O设备是由网络接口硬件触发的中断驱动的。如果内核不立即处理这些异步中断，硬件设备就可能功能失效并丢失数据包或破坏内存缓冲区。 423

虽然BSD操作系统内核是由异步中断驱动的，但是很难使用异步机制（如中断或信号）开发应用程序和高层系统服务。特别是，编程、验证、调试和维护异步程序所需的努力让人望而却步。例如，当一个中断突然抢先占有了一个正在运行的计算时，异步可能产生微妙的时序问题以及竞争条件。

为了避免异步编程的复杂性，BSD UNIX中的高层服务在多个进程内同步运行。例如，使用

同步read()和write()系统调用的FTP或TELNET因特网服务可以阻塞等待I/O操作的完成。反过来，阻塞I/O使开发者能够隐式地在线程的运行堆栈中，而不是在由开发者显式地管理的独立数据结构中维护状态信息和执行历史。

然而，在一个操作系统的语境中，同步和异步处理并不是完全独立的。特别地，在BSD UNIX内同步执行的应用层因特网服务必须与异步运行的内核层协议处理协作。例如，被HTTP服务器调用的同步read()系统调用间接地与异步数据接收和协议处理协作，数据是从以太网接口到达的。

在BSD UNIX的开发中一个关键的挑战是对异步和同步处理的结构化，以增强编程的简易性和改善系统性能。特别地，必须避免同步应用程序的开发者考虑复杂的异步编程细节。但要注意，不能因为在BSD UNIX内核中使用了低效的同步处理机制而降低系统的整体性能。

2. 语境

424

执行必须相互通信的异步和同步处理服务的一个并发系统。

3. 问题

并发系统通常既包含异步处理服务，又包含同步处理服务。系统程序员有充分的理由使用异步特性改善性能。异步程序一般更高效，因为可以直接将服务映射为异步机制，如硬件中断处理程序或者软件信号处理程序。

相反，应用程序员也有充分的理由使用同步处理简化他们的编程强度。通常同步程序更简单，因为可以限制某些服务在处理序列中的良好定义的点上运行。

因此，在定义一个既有同步又有异步执行服务的软件体系结构时，必须解决两个强制条件：

- 该体系结构应该这样设计：需要同步处理的简易性的应用程序开发者无需考虑异步的复杂性。同时，必须将性能最大化的系统开发者不需要考虑同步处理的低效性。
- 该体系结构应该让同步和异步处理服务能够相互通信，而不会使它们的编程模型复杂化或者过度地降低它们的性能。

虽然对编程的简易性和高性能二者的需求可能看起来相互矛盾，但是在一些具体并发系统（特别是大型或复杂的并发系统）中需要解决这两个强制条件。

4. 解决方案

将系统中的服务分解成两层[POSA1]，同步和异步，并且在它们之间增加一个排队层协调异步和同步层中服务之间的通信。

细节：在独立线程或进程中同步地处理高层服务（如耗时长的数据库查询或文件传输），从而简化并发编程。相反，异步地处理底层服务（如由网络接口硬件发出的中断所驱动的短期协议处理程序），以增强性能。如果驻留在相互独立的同步和异步层中的服务必须相互通信或同步它们的处理，则应允许它们通过一个排队层向对方传递消息。

425

5. 结构

半同步/半异步模式的结构遵循分层（Layer）模式[POSA1]并包括四个参与者：

同步服务层（Synchronous service layer）完成高层处理服务。同步层中的服务在独立的在操作时可以阻塞的线程或进程中运行。

➡操作系统例子中的因特网服务在独立的应用程序进程中运行。这些进程根据因特网服务的要求调用read()和write()操作进行同步I/O处理。 ❑

异步服务层（asynchronous service layer）进行低层处理服务，这些低层处理服务通常由一个或多个外部事件源发出。异步层中的服务在进行操作时不能阻塞，而不会过度降低其他服务的性能。如图5-15所示。

➡BSD UNIX操作系统内核中对I/O设备和协议的处理是在中断处理程序中异步进行的。这些处理程序一直运行到完毕，就是说，它们不会被阻塞或与其他线程同步它们的执行，直到结束。 ❑

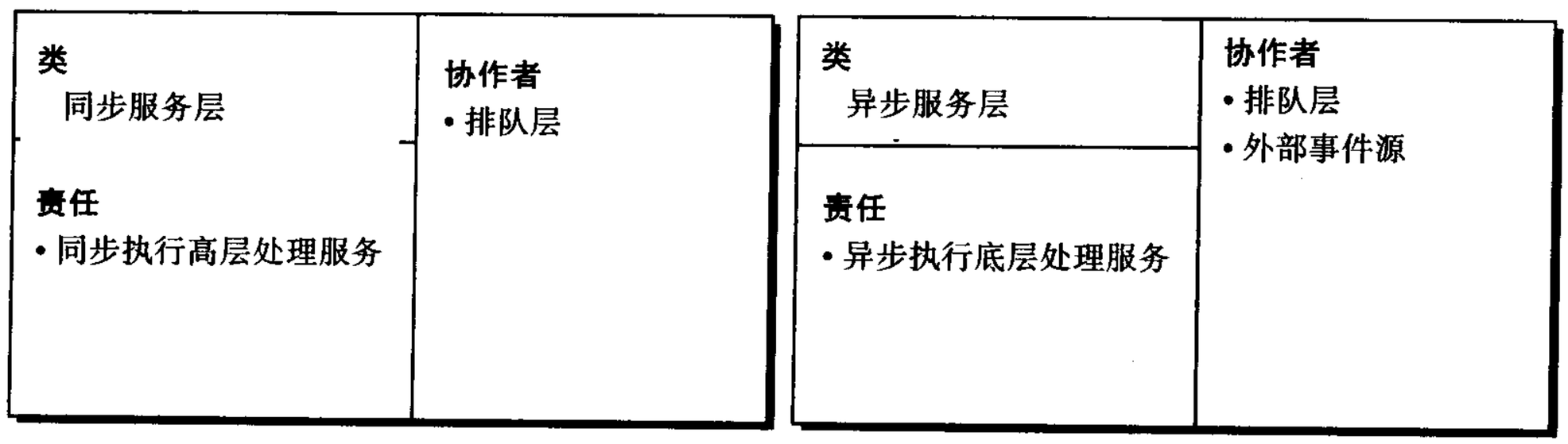


图 5-15

排队层（queueing layer）为同步和异步层服务之间提供通信机制。例如，异步服务产生包含数据和控制信息的消息，然后将这些消息缓冲保存在排队层中，以后同步服务可以获取它们，反之亦然。当消息从另外的层传递给某一个层时，排队层负责通知这一层中的服务。因此排队层使异步和同步层可以以“生产者/消费者”方式交互，类似于管道和过滤器模式中定义的结构[POSA1]。 426

➡BSD UNIX操作系统提供一个Socket层[Ste98]。该层作为同步因特网服务应用程序进程和BSD UNIX内核中异步的中断驱动的I/O硬件服务之间的缓冲和通知点。 ❑

外部事件源（external event source）产生被异步服务层接收和处理的事件。常见的操作系统的外部事件源包括网络接口、磁盘控制器和用户终端。如图5-16所示。

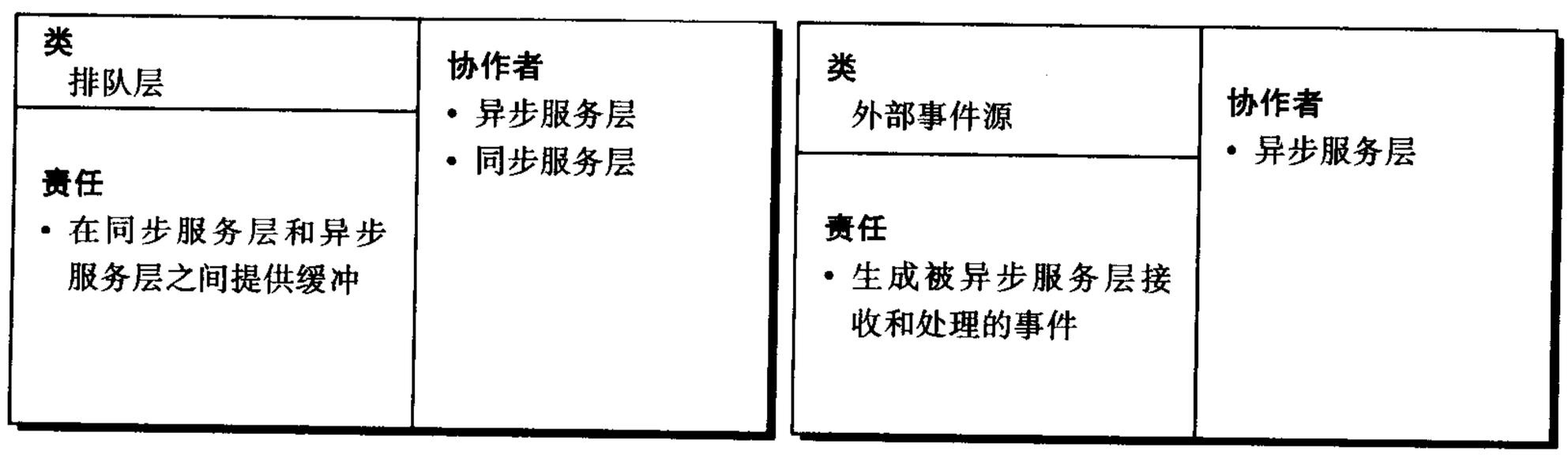
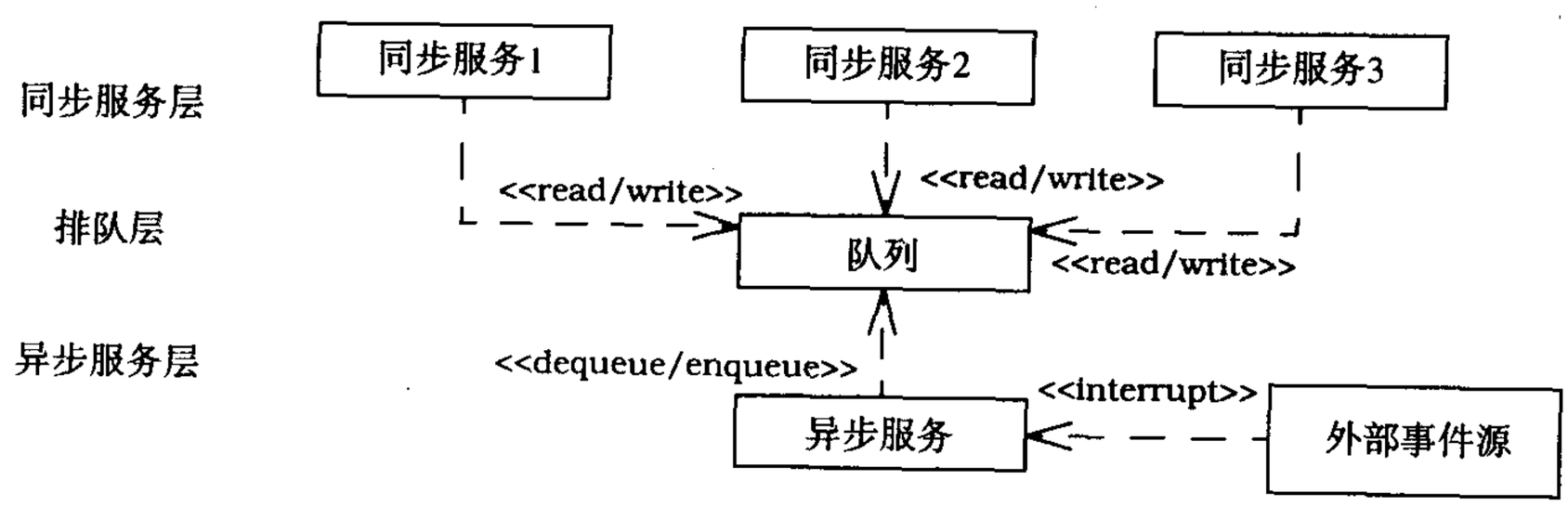


图 5-16

类图5-17说明了这些参与者之间的结构和关系：



427

图 5-17

6. 动态特性

半同步/半异步模式中的异步和同步层通过排队层传递消息进行交互。下面描述从外部事件源到来的输入“自底向上”到达时所发生交互的三个阶段（如图5-18）：

- 异步阶段。在这一阶段，外部输入源通过异步事件通知（比如中断或信号）与异步服务层交互。异步服务完成了输入处理后，它们可以通过排队层将它们的结果传递给同步层中指定的服务。
- 排队阶段。在该阶段，排队层对从异步层传递到同步层的输入进行缓冲，并通知同步层输入可用。
- 同步阶段。在这一阶段，同步层中相应的服务获得并处理被异步层的服务放入排队层中的输入。

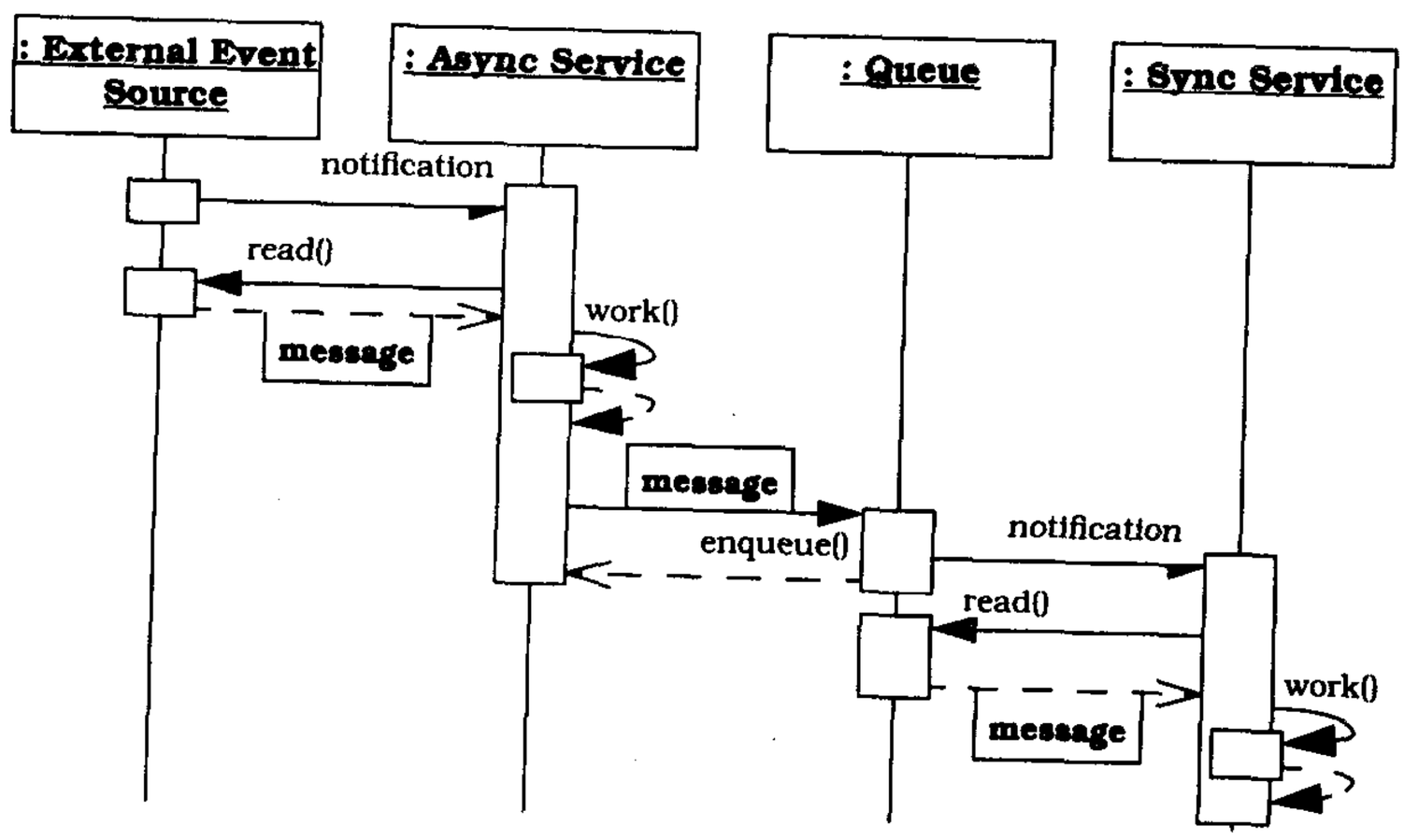


图 5-18

当运行在同步层中的服务发出的输出到达时，各层和模式参与者之间的交互被颠倒而形成一个“自顶向下”的顺序。

428

7. 实现

本节描述了用来实现半同步/半异步模式并和将其应用于构造高层应用程序（如Web服务器[Sch97]和数据库服务器）以及低层系统（如BSD UNIX操作系统）的体系结构活动。这里提供

了几个出自不同领域的例子。

(1) *将整体系统分解为三层*：同步层、异步层和排队层。可以用三个子活动来说明如何分离一个按照半同步/半异步模式设计的系统体系结构。

(1.1) *标识高层和/或耗时长的服务并将它们装配到同步层中*。并发系统中的许多服务在使用同步处理编程的时候更容易实现。这些服务通常进行相对高层或耗时长的应用处理，如传输Web服务器中的大容量数据流或者进行数据库的复杂查询。因此同步层的服务应该运行在独立的进程或线程中。如果没有可用的数据，那么服务可以在端到端应用程序通信协议的控制下在排队层中阻塞，等待响应。

➡BSD UNIX操作系统例子中的每个因特网服务都在独立的应用程序进程中运行。每个应用程序进程使用与它实现的因特网服务有关的协议与它的客户机通信。可以通过在TCP Socket上的同步阻塞和等待BSD UNIX内核异步完成I/O操作来实现这些进程中的I/O操作。 ❑

(1.2) *标识低层和/或耗时短的服务并将它们配置到异步层中*。系统中的某些服务由于延时不能阻塞。这种服务一般实现与外部事件源交互的低层或耗时短的系统处理，如用户终端或由中断驱动的硬件网络接口。为了将反应能力和效率最大化，这些事件源必须尽快得到处理并且不能阻塞为它们服务的线程。它们的服务应该被从外部事件源发出的异步通知或中断触发并一直执行到结束，这时它们可以将包含结果的消息插入排队层。 429

➡在操作系统例子中，为响应异步硬件中断BSD UNIX 内核进行对I/O设备驱动程序和通信协议的处理。内核中每个异步操作一直执行到结束，在必须与同步层中运行因特网服务应用程序进程交互时，将包含数据和/或控制信息的消息插入到Socket层。 ❑

(1.3) *标识层间通信策略并将它们装配到排队层中*。排队层是一个中介者[GoF95]，将异步层和同步层服务之间的通信分离开。这些服务就不会彼此进行直接访问，仅通过排队层互相访问。排队层实现的与通信相关的策略包括多路复用（分解）、缓冲、通知和流控制。异步层和同步层中的服务使用这些排队策略实现在同步层和异步层之间传递消息的协议[SC96]。

➡在BSD UNIX操作系统例子中，Socket机制[Ste98]定义了同步因特网服务应用程序进程和异步操作系统内核间的排队层。每个因特网服务使用一个或多个Socket，这些Socket是由BSD UNIX管理的队列，用于对应用程序进程、TCP/IP协议栈和内核中的网络硬件设备间交换的消息进行缓冲。 ❑

(2) *实现同步层中的服务*。同步层中的高层和/或耗时长的服务通常使用多线程和多进程技术实现。与线程相比，进程包含更多的状态信息并需要更多的开销来生成、同步化、调度和相互间通信。因此，在独立的线程中比在独立的进程中实现同步服务可以使应用程序更简单和更高效。

然而，多线程会降低应用程序的健壮性，因为进程中各独立的线程会相互干扰。例如，一个错误线程可能破坏由进程中的其他线程共享的数据，并因此产生不正确的结果，使进程崩溃，或导致进程永久性挂起。因此，为了增加健壮性，可以在独立的进程中实现应用程序服务。 430

➡BSD UNIX例子中的因特网服务是在独立的进程中实现的。这种设计增加了它们的健壮性并防止对某些资源（如其他用户拥有的文件）的未授权访问。 ❑

(3) *实现异步层中的服务*。异步层中的低层和/或耗时短的服务通常没有它们自己专有的控制线程。它们必须从其他地方借用一个线程，如操作系统内核的“空闲线程”或一个独立的中断

堆栈。为了确保对其他系统服务（如高优先级硬件中断）有足够的反应时间，这些服务必须异步地运行并且不能被长时间阻塞。

下面是两个可以用来触发异步服务执行的策略：

- 异步中断。该策略通常在开发被外部事件源（如网络接口或磁盘控制器）发出的硬件中断触发的异步服务时使用。在这种策略中，当事件在一个外部事件源上发生时，中断通知与事件关联的处理程序，然后该处理程序处理事件直到结束。

  在复杂的并发系统中，可能有必要定义中断层次（hierarchy）来允许不太关键的处理程序被拥有更高优先级的处理程序占先。为了防止中断处理程序在被访问时破坏共享状态，必须保护被异步层使用的数据结构，例如可以采用提高中断优先级[WS95]的方式。

  ➡BSD UNIX内核使用一个两级中断图式管理网络数据包处理[MBKQ96]。时间要求严格的处理在高优先级上完成，而不太关键的软件处理在低优先级上完成。这种两级中断图式防止了因软件协议处理的开销而延误高优先级的硬件中断的服务。 ❑

431

- 主动I/O。这种策略常用于开发基于高层操作系统API，如Windows NT重叠I/O和I/O完成端口[Sol98]或者异步I/O系统调用的POSIX aio_*系列的异步服务。在这种策略中，I/O操作由异步操作处理器执行。一个异步操作完成后，异步操作处理器产生一个完成事件。然后该事件被分配给与事件关联的处理程序，该处理程序处理事件直到完成。

  ➡例如，主动器模式中的Web服务器介绍了一个使用由Windows NT系统调用API定义的主动I/O机制的应用程序。该例子强调了这样一个事实：异步处理和半同步/半异步模式可以用于不直接访问硬件设备的高层应用程序中。 ❑

这两种异步处理策略有个相同的约束：处理程序不可能长时间阻塞而不破坏对来自其他外部事件源的事件的处理。

(4) 实现排队层。异步层中的服务完成了对来自于外部事件源的输入的处理后，它们一般将结果消息插入排队层。同步层中相应的服务会接着将从排队层中删除这些消息，并对它们进行处理。输出处理时这些角色正好倒过来。在实现排队层时必须定义两个与通信有关的策略：

(4.1) 实现缓冲策略。异步层和同步层中的服务不能直接访问对方的内存，而是通过一个排队层交换消息。排队层对消息进行缓冲，这样同步和异步服务可以并发运行，而不是死板地采用“停止并等待”的流控制协议运行。因此缓冲策略必须实现一个排序、串行化、通知和流控制策略。注意，策略模式[GoF95]可以用来简化选择的策略的配置。

- 实现排序策略。简单的排队层按到达的顺序存储消息，也就是“先进先出”（FIFO）。被某个层的服务放入队列的第一个消息就是第一个被另一个层中的服务移出的消息。FIFO排序易于实现，但是如果高优先级的消息在低优先级消息之后入队列，可能会导致优先级倒置[SMFGOO]。因此，可以使用更复杂的队列策略，按“优先级”顺序存储和获得消息。

432

- 实现串行化策略。异步层和同步层中的服务可以并发地执行。因此队列必须被串行化以避免消息被并发地插入和删除时出现的竞争条件。这种串行化通常使用简单同步机制（如互斥机制[Lew95]）实现。这种机制确保消息可以被插入到排队层的消息缓冲区和从中删除，而不会破坏它的内部数据结构。
- 实现通知策略。在从一层向另一层发送的消息到达时，有必要通知该层的服务。通常使用

更高级和更复杂的同步机制（如信号灯或条件变量[Lew95]）实现由排队层提供的通知策略。这些同步机制可以在发给同步层或异步层的数据到达排队层时通知同步层或异步层中的相应服务。“变体”一节概述了其他的几个基于异步信号和中断的通知策略。

- 实现流控制策略。系统不可能提供无限制的资源量对排队层中的消息进行缓冲。因此，需要控制在同步层和异步层之间传递的数据量。流控制是一种技术，能防止同步服务向异步层发出消息的速度高于消息能在网络接口传输和排队的速度而“淹没（flood）”异步层。

同步层中的服务可以阻塞。普通的流控制策略在一个同步服务产生和排队超过某一数量的消息时，简单地将它置于睡眠。在异步服务层将队列置空到低于某个水平时，排队层可以唤醒处于睡眠的同步服务，继续它的处理。

433

相反，异步层中的服务不能阻塞。如果它们产生过多的消息，那么普通的流控制策略允许排队层取消消息，直到同步服务层完成了对队列中消息的处理。如果消息与一个可靠的面向连接的传输协议（如TCP[Ste93]）关联，发送者将最终超时并重新发送被取消的消息。

(4.2) 实现多路复用（分用）机制。在半同步/半异步模式的简单实现中，如在领导者/追随者模式的“例子”一节描述的OLTP服务器，排队层里只有一个队列。该队列被所有异步层和同步层中的服务共享，任何服务可以处理任何请求。这种配置减轻了对复杂多路复用（分用）机制的需要。在这种情况下，一般的实现是定义一个单件队列[GoF95]，所有服务使用它插入和删除消息。

在更复杂的半同步/半异步模式的实现中，某个层中的服务可能需要对另一个层中的一些服务发送和接收特定的消息。因此一个排队层可能需要多个队列（如每个服务一个队列）。对于多个队列，需要更复杂的多路分解机制，以确保在不同层的服务间交换的消息放置在相应的队列中。一般的实现是使用某些类型的多路复用（分用）机制（如哈希表[HMPT89][MD91]）将消息放进相应的队列。

➡监视器对象和主动对象模式中定义的Message_Queue组件说明了各种实现排队层的策略：

- 监视器对象模式使用互斥机制和条件变量，这样能确保不管并发调用队列方法的线程数是多少，某一时刻队列中仅有一个方法执行。队列在客户机线程中，也就是运行同步和异步服务的线程中，执行它的方法。

434

- 主动对象模式将队列上的方法调用和方法执行分离。这样多个同步和异步服务可以并发地调用队列的方法。方法在运行同步和异步服务的线程之外的线程中执行。

主动对象和监视器对象模式的“参见”一节讨论了使用这些模式实现排队层的利弊。 ❑

8. 已解决的例子

第1章“并发和基于网络化的对象”，以及本书中其他模式（如主动器、定界加锁、策略化加锁和线程安全接口）说明了Web 服务器应用程序设计的各种特性。在这一节，我们通过介绍BSD UNIX操作系统[MBKQ96][Ste93]如何应用半同步/半异步模式来通过以太网上的TCP/IP协议栈接收一个HTTP GET请求，研究执行Web服务器的更广泛的系统语境。

BSD UNIX是一个不能有效支持异步I/O的操作系统。因此它不适合使用主动器模式实现Web服务器。因此下面概述BSD UNIX如何协调同步应用程序进程和异步操作系统内核之间的服

务和通信。

特别是，要描述[⊖]：

- 被一个Web服务器应用程序（HTTPD进程）同步调用的read()系统调用。
- 到达以太网接口的数据的异步接收和协议处理。
- read()调用的同步完成，该函数调用将控制权和GET请求数据返回到HTTPD进程。

435

以上步骤如图5-19所示。

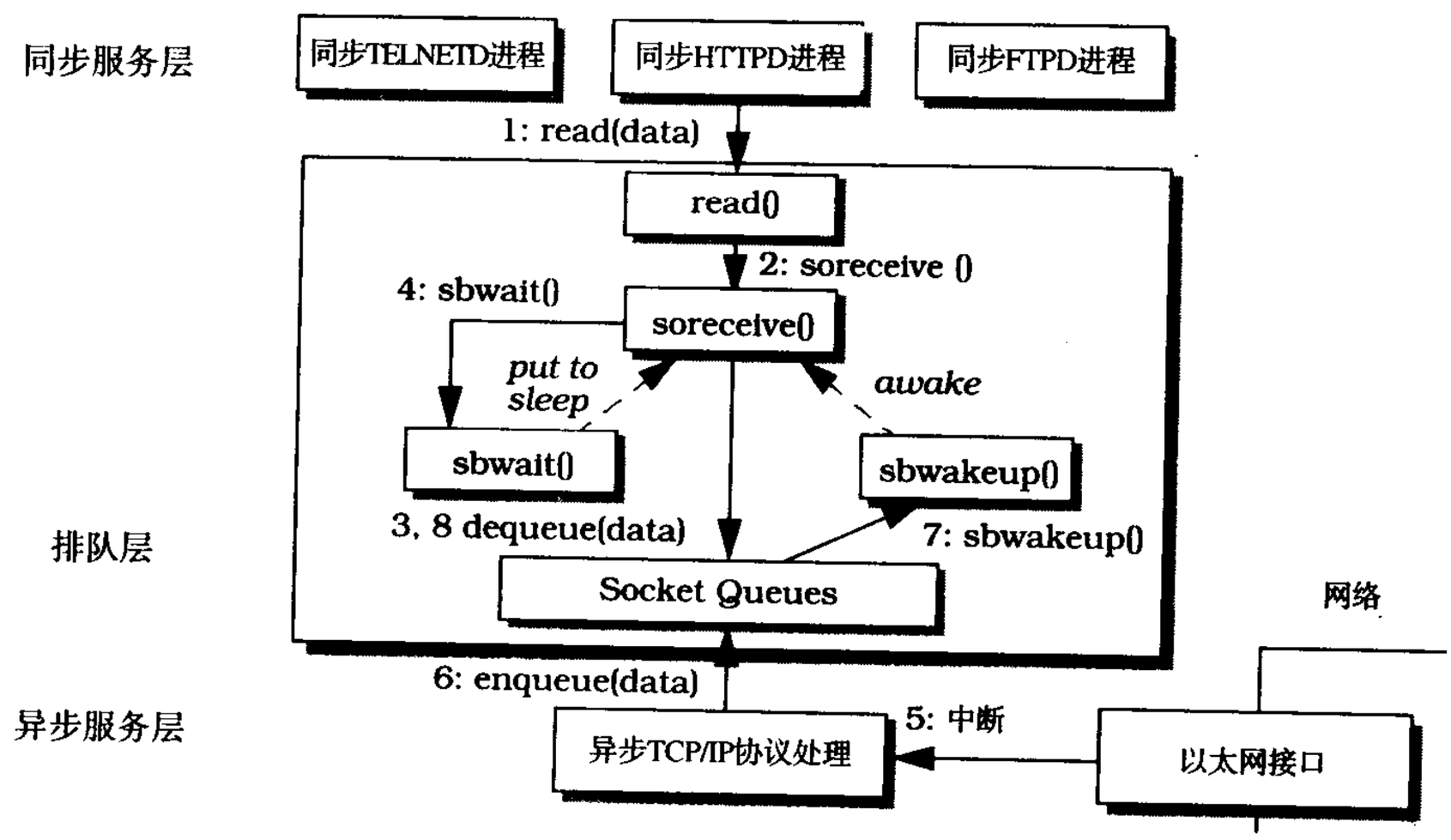


图 5-19

如图5-19所示，HTTPD进程在一个连接的套接字句柄上调用read()系统调用，接收一个封装在TCP包中的HTTP GET请求。从HTTPD进程的角度看，read()系统调用是同步的，因为进程调用read()并阻塞，直到GET请求数据被返回。然而，如果数据不是立即可得的，那么BSD UNIX内核将HTTPD进程置于睡眠状态直到数据从网络到达。

然而，实现同步read()系统调用时会发生许多异步活动。虽然HTTPD进程在等待数据时可以睡眠，但是BSD UNIX内核不能睡眠，因为其他应用程序进程（如内核中的FTP和TELNET服务以及I/O设备）需要内核的服务并发和高效地运行。

在read()系统调用开始后，应用程序进程切换到“内核模式”并开始运行特权指令（privileged instruction），这些指令将它同步指向BSD UNIX连网子系统。最后，应用程序进程的控制线程结束于内核的soreceive()函数。该函数通过将数据从套接字队列传递到应用程序进程，为各种套接字（如数据报套接字和流套接字）处理输入。soreceive()函数为向外发出的数据包定义了同步应用程序进程层和异步内核层之间的边界。

436

HTTPD进程的read()系统调用可以用两种方式被soreceive()处理，这取决于Socket的特征和套接字队列中的数据量：

⊖ 一个按深度进行的代码遍历显示半同步/半异步模式如何应用在BSD UNIX网络和文件系统中，在[PLoPD2]中有描述。

- 完全同步。如果由HTTPD进程请求的数据在套接字队列中，那么soreceive()函数可以立即对它进行拷贝并且read()系统调用将同步完成。
- 半同步和半异步。如果HTTPD进程请求的数据还不可用，内核就调用sbwait()函数将进程置于睡眠，直到请求的数据到达。

在sbwait()将进程置于睡眠后，BSD UNIX调度程序将切换到另一个准备运行的进程语境。然而，从HTTPD进程的角度看，read()系统调用看上去是被同步执行的。当含有被请求数据的包到达时，内核将对它们进行异步处理，正如以下所述。当足够的数据放进套接字队列以满足HTTPD进程的请求时，内核将唤醒该进程并完成它的read()系统调用。然后该调用被同步地返回，这样HTTPD进程可以分析并执行GET请求。

为了使BSD UNIX内核的性能最大化，所有协议处理被异步地执行，因为I/O设备由硬件中断驱动。例如，到达以太网接口的包通过被以太网硬件异步初始化的终端处理程序分发到内核。这些处理程序接收设备发出的包并触发随后的异步高层协议处理，如IP和TCP。最后，包含应用程序数据的合法包在Socket层被排队，BSD UNIX内核调度并分发在等待的HTTPD进程同步使用该数据。

例如，当包到达一个以太网接口时开始进行与HTTPD进程的read()系统调用有关的“半异步”处理，并触发一个异步硬件中断。在中断处理程序的语境中实现对输入包的处理。在中断过程中，由于没有应用进程语境和相应的线程控制，所以BSD UNIX内核不能进行睡眠或阻塞。以太网中断处理程序因此“借用”内核的控制线程。类似地，BSD UNIX内核在应用进程进行系统调用时，借用应用进程的控制线程。 437

如果包用于一个应用进程，那么它将被传递给传输层，由传输层实现附加的协议处理（如TCP指定段重组和确认）。最后，传输层将数据添加到接收套接字队列中并调用sbwakeup()，该函数是输入包的异步和同步层的边界。该调用唤醒在soreceive()中正休眠等待套接字队列中的数据的HTTPD进程。如果HTTPD进程请求的所有数据都到达，那么soreceive()将把数据拷贝到HTTPD提供的缓冲区，允许系统调用将控制返回给Web服务器。因此，即使当该进程处于睡眠时进行了异步处理和语境切换，从HTTPD进程的角度来说，read()调用看起来还是同步的。

9. 变体

具有同步数据I/O的异步控制。“实现”一节描述的HTTPD Web服务器根据它的判断同步地从排队层“拉出”消息，因此将控制和数据活动结合起来。然而，在某些操作系统平台上可以将控制和数据分离，这样将消息插入排队层时可以异步地通知同步层中的服务。该变体的主要优点是更高层的“同步”服务可以更加灵敏，因为可以异步地通知它们。

➡UNIX的信号驱动I/O机制[Ste98]实现了半同步/半异步模式变体。在数据到达其中一个Socket时，UNIX内核使用SIGIO信号将控制“推”到更高层的应用进程。当一个进程异步地接收到该控制通知时，它可以同步地通过read()从套接字排队层“拉出”数据。 ❑ 438

当然，使用异步控制的缺点是高层服务的开发者现在必须面对“问题”一节指出的许多异步复杂性问题。

半异步/半异步。这种变体是对前一种变体的扩充，其扩充手段是传播异步控制通知和数据

操作一直到“同步”层中的更高层服务。因此这些高层服务可以利用低层异步机制的效率优势。

➡例如，在POSIX实时编程规范[POSIX95]中定义的实时信号接口就支持该变体。特别地，当一个实时信号出现时，可以把缓冲区指针传递给由操作系统分配的信号处理程序函数。Windows NT支持一个使用重叠I/O和I/O完成端口[Sol98]的类似机制。在这种情况下，当一个异步操作完成时，与其相关的重叠I/O结构指明哪个操作已经完成，并且将数据单独传递。主动器模式和异步完成标记模式描述了如何利用异步操作和重叠I/O来构造应用程序的方法。❑

该变体的缺点与前面的变体类似。如果大多数或全部服务可以由异步操作驱动，那么设计可以通过应用主动器模式而不是半同步/半异步模式更好地进行建模。

半同步/半同步。该变体为低层服务提供同步处理。如果异步层是多线程的，那么它的服务可以自动运行，并使用排队层向同步服务层传递消息。该变体的优势是异步层中的服务可以简化，因为它们可以阻塞而不影响该层中的其他服务。

➡微内核操作系统（如Mach[Bl90]或Amoeba[Tan95]），有代表性地使用了这种变体。微内核作为独立的多线程“进程”运行，该“进程”与应用进程交换消息。类似地，多线程操作系统微内核（如Solaris[EKBF+92]）可以在内核中支持多个同步I/O操作。❑

439 使内核多线程化可以用于实现的中断轮询，通过专门提供一个内核线程在共享内存中以固定时间间隔对一个域进行轮询，为高性能连续介质系统减少了语境切换量[CP95]。相反，单线程操作系统内核（如BSD UNIX）限制了低层内核服务使用异步I/O，而且仅支持高层应用进程的同步多编程。

当然，为低层服务提供同步处理的缺点是它可能增加开销，并因此极大地降低整体系统性能。

半同步/半反应。在面向对象应用程序中，半同步/半异步模式可以作为一种结合了主动器模式和主动对象模式的线程池变体的混合体系结构模式来实现。在这种公共变体中，反应器的事件处理程序构成了“异步”层[⊖]服务，且排队层可以由主动对象的激活表实现。被主动对象线程池中的调度程序分配的服务者构成同步层中的服务。这种变体的主要优点是它提供了简化功能。这种简化通过在一个单线程的反应器中进行事件多路分解和分配而获得的，这个反应器与在一个主动对象的线程池中事件的并发处理分离开。

➡在领导者/追随者模式的“例子”一节中描述的OLTP服务器应用了该变体。“异步”服务层使用反应器模式对多客户端发出的事务请求进行多路分解，并分配事件处理程序。该处理程序将请求插入排队层，该排队层是一个使用监视器对象模式实现的激活表。类似地，同步服务层使用主动对象模式的线程池变体将请求从激活表传播到工作者线程池，该线程是来自客户机的服务事务请求。每个主动对象的线程池中的线程可以同步地阻塞，因为它们具有
440 自己的运行时栈。❑

这种变体的缺点是排队层会引入对某些应用程序来说不必要的附加语境切换、同步、数据分配和数据拷贝开销。在这种情况下，在构建并发应用程序方面，领导者/追随者模式可能比半同步/半异步模式更高效、更可预测、更可扩展。

⊖ 虽然该反应层不是真正异步的，但它具有异步服务的主要特征。特别地，由反应器分配的事件处理程序不能在不耗尽其他事件资源的情况下阻塞很长时间。

10. 已知使用

UNIX连网子系统。BSD UNIX连网络子系统[MBKQ96]和UNIX STREAMS通信框架[Ris98]使用半同步/半异步模式构造应用进程和操作系统内核的并发I/O体系结构。这些内核中的I/O是异步的并且由中断触发。排队层由BSD UNIX[Ste98]中的Socket层和UNIX STREAMS[Rago93]中的Stream Heads实现。应用进程的I/O是同步的。

大多数UNIX网络驻留进程（如TELNETD和FTPD），是作为同步调用`read()`和`write()`系统调用的应用进程开发的[Ste98]。这种设计能保证应用程序开发者不被内核处理的异步I/O复杂性困扰。然而，还有一些混合的机制，如UNIX SIGIO信号，可以用来通过异步控制通知触发同步I/O处理。

CORBA ORB。MT-Orbix[Bak97]使用半同步/半异步模式的变体在并发服务器中分配CORBA远程操作。在MT-Orbix的ORB Core中，一个独立线程与每个连接到客户机的套接字句柄关联。每个线程同步地阻塞，从客户机读取CORBA请求。接收到一个请求后，将该请求多路分解并插入排队层。接着同步层中的一个主动对象线程唤醒，从队列中取出请求，并通过进行对CORBA服务者的上调（upcall），将请求处理完。

ACE。ACE框架[Sch97]将半同步/半异步模式的“半同步/半反应”变体应用到应用层网关中，这种网关在分布式系统的各对等体间传递消息[Sch96]。`ACE_Reactor`是反应器模式的ACE实现，对“异步”层中与它们关联的事件处理程序多路分用指示事件。ACE `Message_Queue`类实现排队层，同时ACE `Task`类在同步服务层中实现主动对象模式的线程池变体。 441

管道（Conduit）。选择（Choices）操作系统项目[CIRM93]中的Conduit通信框架[Zweig90]实现一个半同步/半异步模式的面向对象版本。应用进程是同步主动对象，适配器管道用作排队层，而且和管道微内核异步，并通过中断与硬件设备通信。

餐厅。许多餐厅使用半同步/半异步模式的变体。例如，餐厅常常雇用一个领班负责迎接顾客，并在餐厅繁忙时留意给顾客安排桌位，为等待就餐的顾客按序排队是必要的。领班由所有顾客“共享”，不能被任何特定顾客占用太多时间。当顾客在一张桌子入座后，有一个侍应生专门为这张桌子服务。

11. 结论

半同步/半异步模式有如下**优点**：

简化和性能。简化高层同步处理服务的编程，又没有降低低层系统服务的性能。并发系统常常有比低层服务更多并且种类更丰富的高层处理服务。将高层同步服务和低层异步处理服务分离能简化应用程序编程，因为复杂的并发控制、中断处理和时序服务可以局限在异步服务层中。异步层也可以处理应用程序开发者难于健壮地编程的低层细节，如中断处理。此外，异步层可以管理与硬件相关组件的交互，如DMA、内存管理和I/O设备寄存器。

使用同步I/O也可以简化编程，可以在多处理器平台上改善性能。例如，长时间数据传输（如从一个分层存储管理系统中下载一个大的医疗图像[PHS96]）可以使用同步I/O进行简化并有效地实现。特别地，可以对某个线程指定一个处理器用于传输数据。这使该CPU的指令和数据高速缓存与整个图像传输操作关联。 442

事务分离。分离每一层的同步策略，因此各层不需要使用相同的并发控制策略。例如，在单线程BSD UNIX内核中，异步服务层通过低层机制（如提升和降低CPU中断级别）实现同步。相反，同步服务层中的应用程序进程通过高层机制（如监视器对象和同步消息队列）实现同步。

➡遗产库，如X Windows和较旧的RPC工具包，通常不是可重入的。因此不能使用多个控制线程在导致的竞争条件中并发调用这些库函数。然而，为了改善性能或利用多CPU，有必要在单独的线程中执行整体数据传输或数据库查询。在这种情况下，半同步/半异步模式的半同步/半反应变体可以用来将应用程序的单线程部分与多线程部分分离。

例如，可以在反应器的控制下运行应用程序的X Windows GUI处理。同样，长数据传输可以在主动对象线程池的控制下运行。通过使用半同步/半异步模式分离应用程序各层的同步策略，可以继续使用非可重入函数而不需要改变现有代码。 ❑

集中的层间通信。层间通信集中在一个访问点进行，因为所有交互都由排队层协调。排队层对另外两层间传递的消息进行缓冲。这消除了加锁和串行化的复杂性，否则，如果同步和异步服务层直接访问对方内存中的对象，这就是必要的。

半同步/半异步模式也有如下**不足**：

443

当数据在同步和异步服务层之间通过排队层传输时，语境切换、同步化和数据拷贝开销可能会引起越界带来的开销。例如，大多数操作系统通过将排队层放置在用户层和内核层保护域之间的边界上实现半同步/半异步模式。越过这个边界会导致极大的性能损失[HP91]。

一种降低这种开销的方法是在同步服务层和异步服务层之间共享一个内存区域[DP93]。该“零拷贝”设计允许两个层直接交换数据，而不用对排队层拷入或拷出数据。

➡文献[CP95]为BSD UNIX I/O子系统提供了一系列扩展，通过使用轮询中断改善连续介质I/O流来将越界损失控制到最小。这种方法定义了一个缓冲区管理系统，可以在应用进程、内核和它的设备之间使用高效的页面重映射和共享内存机制。 ❑

高层应用程序服务可能不会从异步I/O的效率中获益。对于更高层服务来说，有效地使用低层异步I/O设备也许是不可能的，这取决于操作系统或应用程序框架接口的设计。例如，即使外部I/O源支持计算和通信的异步交替，BSD UNIX操作系统还是阻碍了应用程序高效地使用某些类型的硬件。

调试和测试的复杂性。使用半同步/半异步模式编写的应用程序在调试和测试方面所遇到的困难与主动器和反应器模式的“结论”一节中描述的困难相同。

参见

主动器模式可以视为半同步/半异步模式的一种扩展，将异步控制和数据操作通过各种途径传递到高层服务。通常，如果操作系统平台能够有效地支持异步I/O，并且应用程序开发者能够适应异步I/O编程模型，就应该使用主动器模式。

反应器模式可以与主动对象模式结合使用，以实现半同步/半异步模式的半同步/半反应变体。

444

类似地，如果在异步层和同步层之间没有使用排队层的必要，可以在半同步/半异步模式中使用领导者/追随者模式。

管道和过滤器模式[POSA1]为在软件系统的组件间实现生产者-消费者通信给出了一般原则。因此，半同步/半异步模式的某些配置可以视为管道和过滤器模式的实例，在这里过滤器包含多个细粒度服务的完整层。更进一步说，过滤器可以包含主动对象，衍生出半同步/半反应或半同步/半同步变体。

层模式[POSA1]描述了将服务分散到各独立层中的一般原理。半同步/半异步模式可以看做层模式的特化，层模式的目的是通过在并发系统中为每种服务类型引入两个指定的层将同步处理与异步处理分开。

致谢

Chuck Cranor是本模式原始版的合作者[PLoPD2]。同样感谢Lorrie Cranor和Paul McKenney针对该模式提出的改进建议。

445

5.4 领导者/追随者

领导者/追随者（Leader/Followers）体系结构模式提供一个高效的并发模型。在该模型中，为了检测、多路分解、分配和处理事件源上引发的服务请求，多线程轮流共享一个事件源集合。

1. 例子

考虑一个多层、高容量的在线事务处理（On-line transaction processing, OLTP）系统的设计[GR93]。在该设计中，前端通信服务器将事务请求从远程客户机（如游历agent、断言处理中心或销售点终端）传送到处理请求的后端数据库服务器。在事务提交后，数据库服务器将事务的结果返回到相关的通信服务器，然后通信服务器将结果传递给最初的远程客户机。这种多层体系结构通常使用负载平衡和冗余机制来改善整体系统吞吐量和可靠性。这种机制也减轻了后端服务器对远程客户机的不同通信协议的管理负担。如图5-20所示。

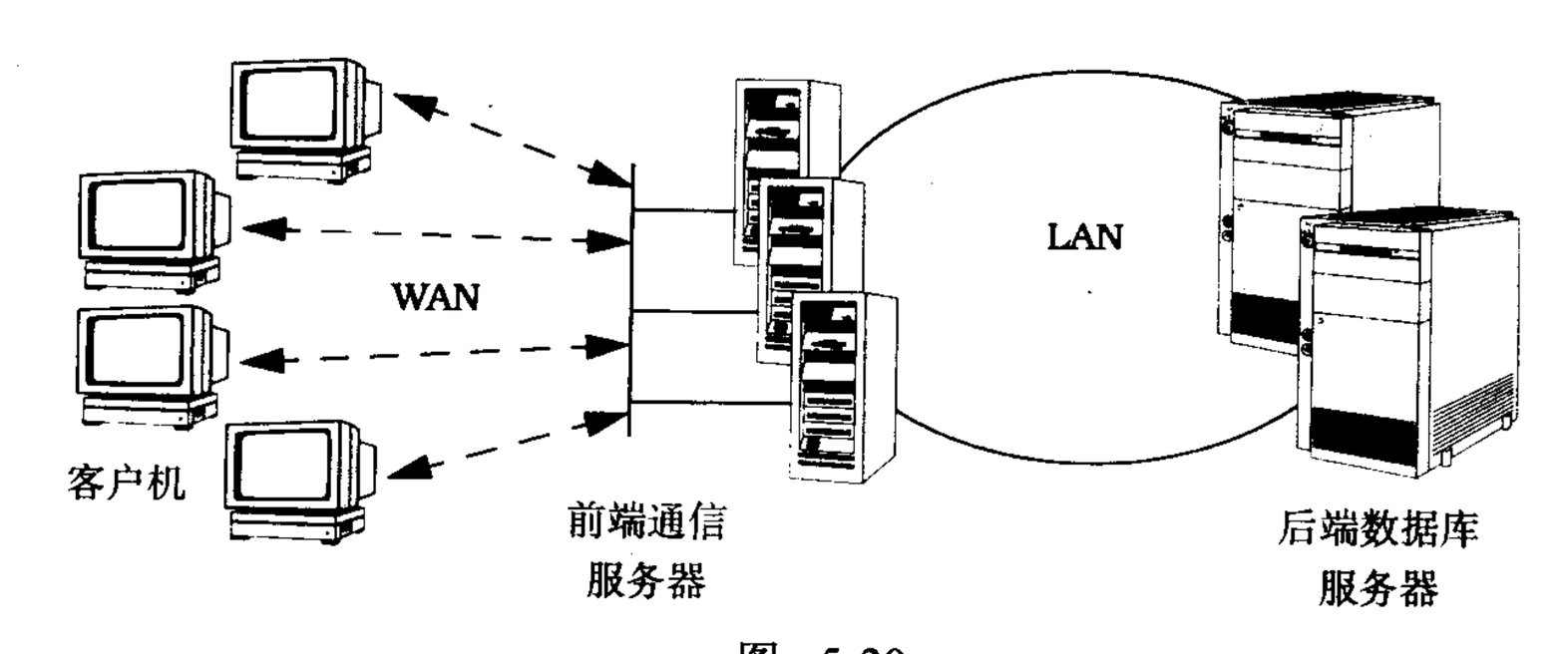


图 5-20

一种实现OLTP服务器的方法是使用基于反应器模式的单线程事件处理模型。然而，这种模型对事件处理进行串行化，在处理耗时长的客户请求事件或阻塞客户请求事件时降低了整体服务器的性能。同样地，单线程服务器不能明显地受益于多处理器平台。

447

改善OLTP服务器性能的一种通用策略是使用多线程并发模型，以同时处理不同客户机发出

的请求以及相应的结果[HPS99]。例如，可以通过建立一个基于半同步/半异步模式的半同步/半反应变体的线程池，将一个OLTP后端服务器多线程化。在这种设计中，OLTP后端服务器包含专门的网络I/O线程，该线程使用select()[Ste98]事件多路分解器等待连接到前端通信服务器上的套接字句柄集上的事件发生。如图5-21所示。

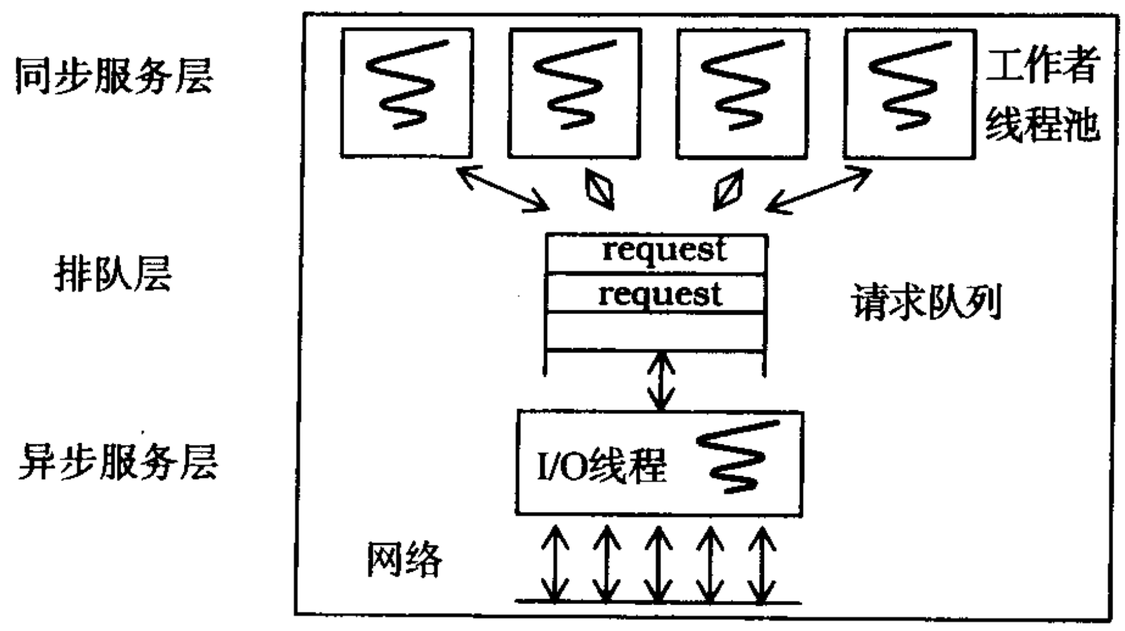


图 5-21

当句柄集中的句柄上有事件发生时，select()将控制返回给网络I/O线程，并指出集合中哪个套接字句柄上有事件等待处理。然后I/O线程从套接字句柄读取事务请求，将它们存储到动态分配的请求中，并将这些请求插入一个用监视器对象模式实现的同步化的消息队列。有一个工作者线程池为该消息队列服务。当线程池中有一个工作者线程时，它将请求从队列中删除，执行指定的事务，并将响应返回给前端通信服务器。

虽然上述线程模型用于许多并发应用程序中，但是正如OLTP例子中表现的一样，它在高容量服务器中使用时可能引起额外的开销。例如，即使是在小工作量下，半同步/半反应线程池设计也将引起动态的内存分配、多同步操作以及在网络I/O线程和工作者线程间传递请求消息的语境切换。这些开销造成了即使在最好的情况下延时也会达到不可接受的界限[PRS+99]。并且，如果OLTP后端服务器运行在一个多处理器系统上，在线程间传递请求所需要的处理器缓存一致性（coherency）协议会产生很大的开销[SKT96]。

448

如果OLTP后端服务器运行在能有效支持异步I/O的操作系统平台上，那么半同步/半反应线程池可以被基于主动器模式的纯异步线程池取代。这种做法通过消除网络I/O线程降低了上述大部分的同步、语境切换和高速缓存一致性开销。可是，许多操作系统不支持或者常常不能有效地支持异步I/O。[⊖] 不过，高容量OLTP服务器将有效地请求多路分解到线程，使之能并发处理结果，这一点很重要。

2. 语境

一个事件驱动的应用程序，其中必须由共享事件源的多个线程高效地处理在一组事件源上到达的多个服务请求。

3. 问题

多线程是实现并发处理多事件的应用程序的一种常用技术。然而，很难实现高性能的多线

⊖ 例如，某些操作系统通过为每个异步操作生成一个线程来支持异步I/O，这样就失去了异步的潜在性能优势。

程服务器应用程序。这些应用程序通常处理大量同时到达的多类型事件，如在OLTP例子中的CONNECT、READ和WRITE事件。为了有效地处理这种问题，有三个强制条件必须解决：

- 服务请求可以来自为每个已连接的客户机分配的多事件源（如多TCP/IP套接字句柄[Ste98]）。因此，一个关键的设计强制条件是在线程和事件源间确定有效的多路分解关联。特别地，由于应用程序或底层操作系统及网络平台的可扩充性限制，为每个事件源关联一个线程也许不可行。

  449

  ➡对于OLTP服务器应用程序来说，为每个套接字句柄关联一个独立的线程可能不现实。特别是，当连接数大大增加时，这种设计在许多操作系统平台上不可能有效地扩展。 ❑
- 为了将性能最大化，必须尽量减少引起与并发有关的开销的原因与（如语境切换、同步化和缓存一致性管理）。特别地，为在多个线程间传递的请求动态分配内存的同步模型会在传统的多处理器操作系统中产生巨大的开销[SchSu95]。

  ➡使用“例子”一节给出的半同步/半反应线程池变体实现OLTP服务器需要在网络I/O线程中动态分配内存，将即将到来的事务请求存储到消息队列中。这种设计在将请求插入或移出消息队列时导致大量的同步和语境切换（如在监视器对象模式中说明的那样）。 ❑
- 多路分解共享的事件源集合上事件的多个线程必须相互协作，以防止竞争条件。竞争条件可能出现在多个线程试图同时访问或修改某些类型的事件源的时刻。

  ➡例如，一个线程池不能并发地使用select()对一个套接字句柄集进行多路分解，因为当在同一个套接字句柄集上有未处理的I/O事件时，操作系统将错误地通知多于一个调用select()的线程[Ste98]。并且，对于面向字节流的协议（如TCP），当多个线程在同一个套接字句柄上调用read()或write()时将破坏或丢失数据。 ❑

4. 解决方案

构造一个线程池，通过对到达事件源的事件多路分解，并向处理事件的应用服务同步地分配事件，依此轮流进行来共享事件源集合。

详细细节：设计一个线程池机制，允许其中的多个线程相互协作并在检测、多路分解、分配和处理事件时保护临界区。在这种机制中，每次允许一个线程——领导者——等待在事件源集合上出现一个事件。同时，其他线程——追随者——排队等待它们成为领导者的机会。当前的领导者线程从事件源集合检测到一个事件后，它首先将一个追随者线程提升为新的领导者，然后扮演处理线程的角色，对事件多路分解并分配给指定的事件处理程序，在处理线程中实现与应用有关的事件处理。在当前领导者线程在由所有线程共享的事件源集合上等待新的事件时，多个处理线程可以并发地处理事件。在处理完事件后，处理线程恢复到追随者角色，并等待再次成为领导者线程。 450

5. 结构

在领导者/追随者模式中有四个关键的参与者：

由操作系统提供句柄，用来区分可以生成事件并将之排队的事件源（如网络连接或打开文件）。事件可以由外部事件源（如从客户机发送一个服务的CONNECT事件或READ事件），或者内部事件源（如超时）引发。句柄集是句柄的集合，可以用来等待一个或多个事件在该句柄集中

的句柄上发生。当可以激活句柄集中句柄上的一个操作而不发生操作阻塞时，句柄集返回到它的调用者。

➡OLTP服务器关注两类事件——CONNECT事件和READ事件——它们分别代表连接和事务请求。前端和后端服务器为每个客户机都建立了一个独立的连接，前端服务器的客户机称为“远程”客户机，前端服务器自身是后端服务器的客户机。每个连接是一个事件源，在服务器中由一个独立的套接字句柄表示。OLTP服务器使用select()事件多路分解器，以此标识事件源中有等待执行事件的句柄，这样应用程序可以在这些句柄上调用I/O操作而不会阻塞调用线程。❑

451

类 句柄和句柄集	协作者
责任 • 句柄在操作系统中标识事件源 • 句柄可以将事件排队 • 句柄集是句柄的汇集	

图 5-22

事件处理程序明确了由一个或多个钩子方法[Pree95][GoF95]组成的接口。这些方法表示可以对发生在句柄上的与应用有关的事件进行处理的操作集。

具体事件处理程序是事件处理程序的特化，并实现应用程序提供的特定服务。特别地，具体事件处理程序实现负责处理从句柄接收的事件的钩子方法。如图5-23所示。

类 事件处理程序	协作者 • 句柄
责任 • 为处理在句柄上发生的事件定义接口	

类 具体事件处理程序	协作者 • 句柄
责任 • 定义应用程序服务 • 以与应用有关的方式处理句柄上接收到的事件 • 在处理线程中运行	

图 5-23

➡例如，OLTP前端通信服务器中的具体事件处理程序接收和确认远程客户机的请求，然后将请求转发给后端数据库服务器。同样，后端数据库服务器中的具体事件处理程序从前端服务

452

器接收事务请求，读/写相应的数据库记录以执行事务，并将结果返回给前端服务器。所有的网络I/O操作都通过套接字句柄来实现，套接字句柄标识各种事件源。 ❑

领导者/追随者模式的核心是线程池，它是一组线程，这组线程共享一个同步器（如信号灯或条件变量），并实现协调角色转换的协议。一个或多个线程扮演追随者角色并在线程池同步器上排队等待扮演领导者角色。其中一个线程被选择成为领导者，等待事件在句柄集中的句柄上发生。当有一个事件发生时，当前领导者线程将一个追随者线程提升为新的领导者。原来的领导者接着扮演处理线程的角色，对从句柄集到相应事件处理程序的事件进行多路分解，并分配处理程序的钩子方法进行事件处理。在处理线程完成了事件的处理后，它再次返回扮演追随者线程的角色，在线程池同步器上等待再次成为领导者线程。如图5-24所示。

类 线程池	协作者 • 句柄集 • 句柄 • 事件处理程序
责任 • 轮流扮演三个角色的线程：在领导者角色中等待事件，在处理角色中处理事件，在追随者角色中排队等待成为领导者 • 包含一个同步器	

图 5-24

➡用领导者/追随者模式设计的每个OLTP服务器有一个等待处理事务请求的线程池，请求到达由句柄集标识的事件源。在任何时刻，池中可以有多线程正在处理事务请求并将结果发回客户机。池中有一个线程是当前领导者，等待新的CONNECT或READ事件到达由所有线程共享的句柄集。上述情况发生时，领导者线程变成处理线程并对事件进行处理，同时池中的一个追随者线程被提升为新的领导者。 ❑ 453

类图5-25说明了领导者/追随者模式中参与者的结构。在该结构中，多个线程共享线程池、事件处理程序和句柄集参与者的相同实例。线程池确保了正确、高效的线程协作：

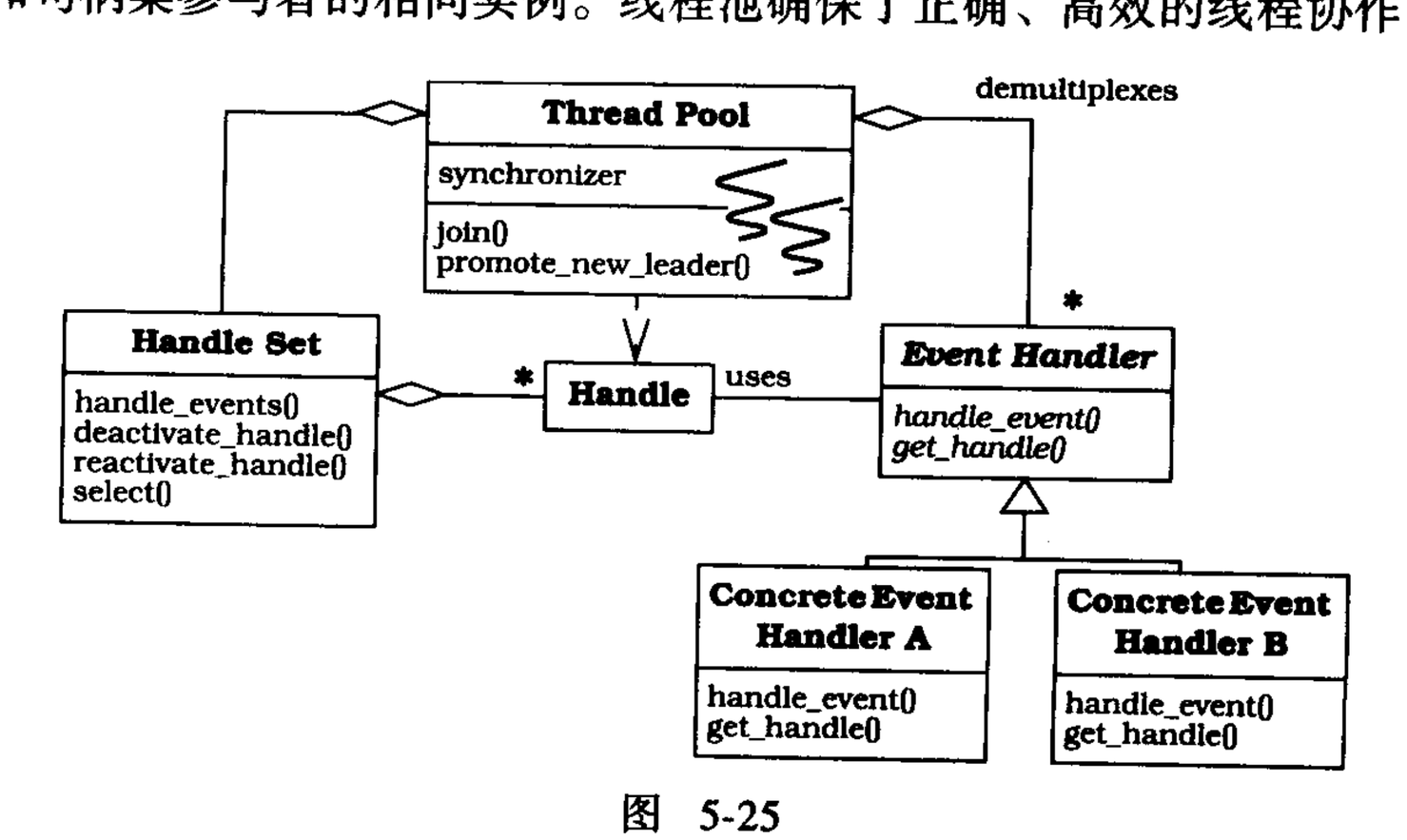


图 5-25

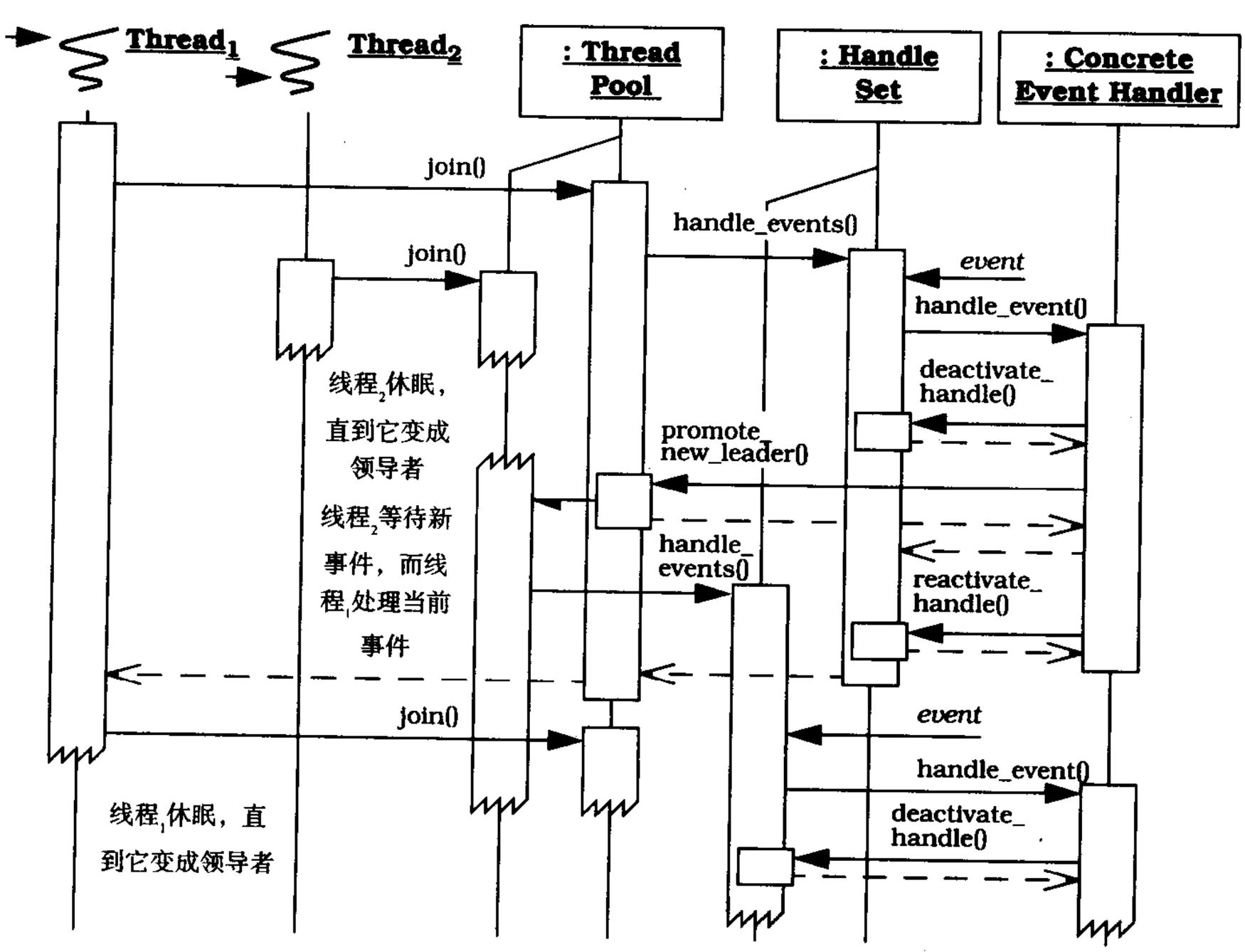
6. 动态特性

领导者/追随者模式中的协作分为四个阶段（如图5-26）：

- *领导者线程多路分解*。领导者线程等待在句柄集中句柄上事件的发生。例如，如果由于事件到达的速度比线程为它们提供服务的速度还快，使得当前没有领导者线程，那么底层操作系统可以将事件在内部排队直到出现一个可用的领导者线程。
- *追随者线程的提升*。在领导者线程检测到新的事件后，它使用线程池选择一个追随者线程，使之成为新的领导者。

454

- *事件处理程序多路分解和事件处理*。在帮助一个追随者线程提升为新的领导者后，以前的领导者线程就扮演处理线程的角色。该线程将检测到的事件并发地多路分解给与事件关联的处理程序并且分配处理程序的钩子方法处理事件。处理线程可以与领导者线程以及处于处理状态的其他线程并发地执行。
- *重新加入线程池*。处理线程的事件处理运行完毕后，它可以重新加入线程池等待处理另一个事件。如果当前没有领导者线程，处理线程可以立即变成领导者。否则，处理线程将又扮演追随者线程的角色，并在线程池同步器上等待，直到它被领导者线程提升。



455

图 5-26

线程在不同状态之间的转换可以用图5-27表示：

7. 实现

实现领导者/追随者模式包含如下六项活动：

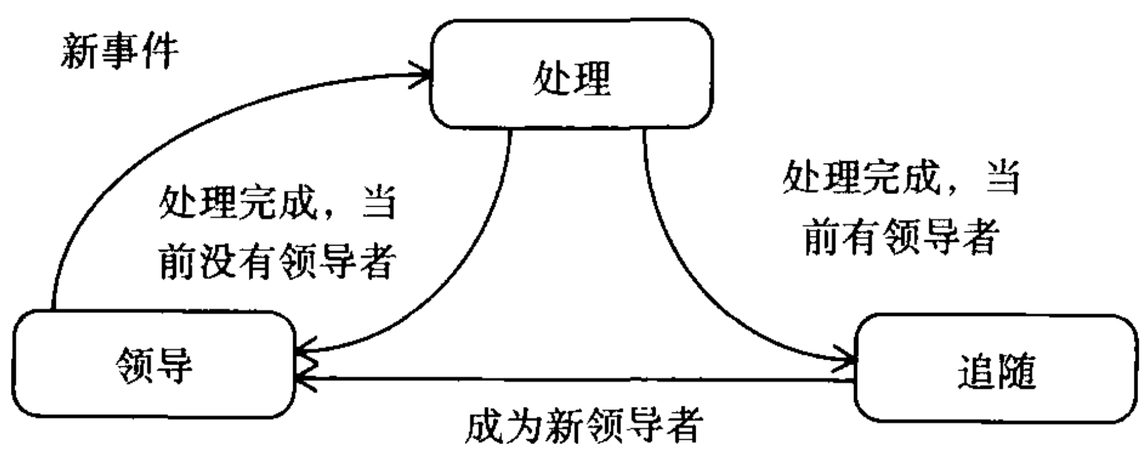


图 5-27

(1) *选择句柄和句柄集机制*。句柄集是句柄的集合，领导者线程可以用它来等待在事件源集上发生事件。开发者通常选择底层操作系统提供的句柄和句柄集机制，而不是随便地实现它们。下面四项子活动有助于选择句柄和句柄集机制：

(1.1) *确定句柄类型*。有两种通用句柄类型：

- *并发句柄*。这种类型的句柄允许多个线程并发访问事件源的句柄而不会引发可能破坏、丢失或扰乱数据的竞争条件[Ste98]。例如，面向记录的Socket API协议（如UDP）允许多线程并发地调用同一个句柄上的`read()`或`write()`操作。
- *迭代句柄*。这种类型的句柄需要多线程迭代访问事件源上的句柄，因为并发访问将导致竞争条件。例如，面向字节流的Socket API协议（如TCP）不保证`read()`或`write()`操作遵守应用层的消息边界。这样，如果Socket上的I/O操作没有正确地串行化，则将导致数据的破坏或丢失。

456

(1.2) *确定句柄集的类型*。有两种通用句柄集类型：

- *并发句柄集*。这种类型的句柄集上可以有并发的动作，例如，线程池对它的并发访问。每次当可能启动句柄集中一个句柄的操作而不阻塞操作时，一个并发句柄集都将该句柄返回给它的调用线程。例如，Win32 `WaitForMultipleObjects()`函数[Sol98]通过允许线程池在同一个句柄集上同时等待来支持并发句柄集。
- *迭代句柄集*。这种类型的句柄集在它可以启动句柄集中一个或多个句柄上的一个操作而不阻塞操作时返回到它的调用者。虽然迭代句柄集可以在单个调用中返回多个句柄，但是它一次只能由一个线程调用。例如，`select()`[Ste98]和`poll()`[Rago93]函数支持迭代句柄集。因此线程池不能使用`select()`或`poll()`在同一句柄集上并发多路分解事件，因为可能会通知多个线程有相同的I/O事件待处理，这将引发错误的行为。

下表概括了并发句柄、迭代句柄和句柄集的每个组合的代表性例子：

句柄 / 句柄集	并发句柄	迭代句柄
并发句柄集	UDP Sockets + `WaitForMultipleObjects()`	TCP Sockets + `WaitForMultipleObjects()`
迭代句柄集	UDP Sockets + `select()`,`poll()`	TCP Sockets + `select()`,`poll()`

(1.3) 确定选择某个句柄和句柄集机制的效果。一般来说，领导者/追随者模式用于防止多线程错误地破坏或丢失数据（如在共享的TCP字节流套接字句柄上并发地调用读操作或在共享句柄集上并发地调用select()）。然而，某些应用程序不需要预防这些问题。特别是，如果句柄和句柄集机制都是并发的，那么许多后续的实现活动都可以跳过。

457

正如在实现活动（1.1）和（1.2）中讨论的，某些协议组合和网络编程API的语义支持在共享句柄上的并发多I/O操作。例如，Socket API支持的UDP确保一条完整的消息总是由一个线程读，另一个线程写，没有部分read()的危险也没有数据因交错write()而被破坏的危险。同样，某些句柄集机制（如Win32 WaitForMultipleObjects()函数[Sol98]）每次调用返回一个句柄，这允许它们能由线程池并发调用。[⊖]

在这些情况下，可以通过简单地使用操作系统的线程调度程序对线程、句柄集和句柄多路复用（分用），健壮地实现领导者/追随者模式，也因此可以跳过实现活动（2）到（6）。

(1.4) 实现事件处理程序的多路分解机制。除了调用事件多路分解器等待一个或多个事件（如select()）出现在它的句柄集上之外，领导者/追随者模式的实现必须将事件多路分解到事件处理程序，并且分配它们的钩子方法以对事件进行处理。一般说来，有两种策略可用于实现这种机制：

- 对低层操作系统事件多路分解机制编程。在这种策略中，直接使用操作系统提供的句柄集多路分解机制。因此，领导者/追随者实现必须包含一个多路分解表，该表是一个包含一个<句柄，事件处理程序，事件类型>三元组集合的管理器[Som97]。在多路分解表中每个句柄作为将事件处理程序与句柄关联的“关键字”，多路分解表也存储每个事件处理程序即将处理的事件类型，如CONNECT和READ。将该表的内容转换到句柄集中再传递到原来的事件多路分解机制，如select()[Ste98]或WaitForMultipleObjects()[Sol98].

  ➡反应器模式的实现活动（3.3）说明了如何实现多路分解表。 ❑

458

- 对高层事件多路分解模式编程。在这种策略中，开发者使用诸如反应器、主动器和包装器外观等高层模式。这些模式有助于简化领导者/追随者的实现，并将解决编程的偶然复杂性所需的工作量减少为直接使用原来的操作系统句柄集多路分解机制。此外，应用高层模式更容易将系统的I/O和多路分解特征与它的并发模型分离，因此减少了代码重复和维护的工作量。

  ➡在OLTP服务器例子中，事件必须被多路分解到与套接字句柄关联的接收事件的具体事件处理程序。反应器模式支持这种行为，因此它可以用于简化领导者/追随者模式的实现。然而，在领导者/追随者模式的语境中，反应器一次仅多路分解一个句柄到与它关联的具体事件处理程序，而不管有多少句柄有待处理的事件。一次仅多路分解一个句柄可以使线程池中的并发最大化，并通过降低对管理独立的待处理事件队列的需求来简化领导者/追随者模式的实现。 ❑

(2) 在句柄集中实现临时激活（停用）句柄的协议。当事件到达时，领导者线程执行以下三步：

- 在句柄集中暂时停用该句柄。

⊖ 然而，当有一个事件时，WaitForMultipleObjects()自身并不解决通知某一特定线程的问题，而这对于支持“变体”一节讨论的受限线程/句柄关联是必要的。

• 将一个追随者线程提升为新的领导者。

• 继续处理事件。

在句柄集中将该句柄停用避免了从选择新领导者到处理事件的时间内出现竞争条件。如果新的领导者在这一段时间内等待句柄集中的同一个句柄，那么它可能再次将该事件多路分解，这样做是错误的，因为这时已经在进行分配了。在事件被处理后，句柄在句柄集中被再次激活，这样领导者线程等待事件在该句柄或句柄集中其他激活的句柄上出现。 459

➡在OLTP例子中，通过扩展反应器模式的实现活动（2）中定义的Reactor接口可以提供句柄停用和重激活协议：

```
class Reactor {
public:
    // Temporarily deactivate the <HANDLE>
    // from the internal handle set.
    void deactivate_handle (HANDLE, Event_Type);

    // Reactivate a previously deactivated
    // <Event_Handler> to the internal handle set.
    void reactivate_handle (HANDLE, Event_Type);
    // ...
};
```

❑

(3) 实现线程池。为了将一个追随者线程提升为领导者角色，以及确定哪个线程是当前的领导者，领导者/追随者模式的工具必须管理一个线程池。一种直接的实现方法是简单地将所有追随者线程放入集合，等待单独的同步器，如信号灯和条件变量。在这种设计中，不管是哪个线程来处理事件，只要共享句柄集的池中的所有线程都是串行化的。

➡例如，如下的LF_Thread_Pool类可以用于OLTP例子中的后端数据库服务器：

```
class LF_Thread_Pool {
public:
    // Constructor.
    LF_Thread_Pool (Reactor *r):
        reactor_ (r), followers_condition_ (mutex_) { };

    // Threads call <join> to wait on a handle set and
    // demultiplex events to their event handlers.
    void join (Time_Value *timeout = 0);

    // Promote a follower thread to become the
    // leader thread.
    void promote_new_leader ();

    // Support the <HANDLE> (de)activation protocol.
    void deactivate_handle (HANDLE, Event_Type et);
    void reactivate_handle (HANDLE, Event_Type et);
private:
    // Pointer to the event demultiplexer/dispatcher.
    Reactor *reactor_;

    // The thread id of the leader thread, which is
    // set to NO_CURRENT_LEADER if there is no leader.
    Thread_Id leader_thread_;

    // Follower threads wait on this condition
    // variable until they are promoted to leader.
    Thread_Condition followers_condition_;
```

460

```
    // Serialize access to our internal state.
    Thread_Mutex mutex_;
};
```

LF_Thread_Pool的构造函数对传递给它的反应器进行高速缓存。默认情况下，该反应器实现使用select()方法，select()支持迭代句柄集。因此，LF_Thread_Pool负责将在反应器的句柄集上轮流调用select()的多线程串行化。

应用程序线程调用join()在句柄集上等待，并且将新的事件多路分解到与它们关联的事件处理程序。正如在实现活动（4）中显示的，该方法直到应用程序结束或join()发生超时的时候才返回到它的调用者。Promote_new_leader()方法将集合中一个追随者线程提升为新的领导者，如实现活动（5.2）所示。

deactivate_handle()方法和reactivate_handle()方法分别在反应器句柄集中停用和再激活句柄。这些方法的实现与在实现活动（2）中显示的Reactor接口中的方法定义相同。

注意，该线程池中的所有线程共享一个条件变量同步器followers_condition_。如实现活动（4）及（5）所示，LF_Thread_Pool的实现使用了监视器对象模式。 ❑

461

(4) *实现允许线程初次加入（以及以后再次加入）线程池的协议。*

该协议用于以下两种情况：

- 在首创了能获得和处理事件的线程池后；以及
- 在处理线程已完成并且可以处理另一个事件时。

如果没有领导者线程可用，则处理线程可以立即变为领导者。如果领导者线程已经获得，一个处理线程可以在线程池同步器上等待变成追随者。

➡后端数据库服务器可以实现如下LF_Thread_Pool的join()方法，在句柄集上等待、并多路分解新事件到与它们相关联的事件处理程序：

```
void LF_Thread_Pool::join (Time_Value *timeout) {
    // Use Scoped Locking idiom to acquire mutex
    // automatically in the constructor.
    Guard<Thread_Mutex> guard (mutex_);

    for (;;) {
        while (leader_thread_ != NO_CURRENT_LEADER)
            // Sleep and release <mutex> atomically.
            followers_condition_.wait (timeout);

        // Assume the leader role.
        leader_thread_ = Thread::self ();

        // Leave monitor temporarily to allow other
        // follower threads to join the pool.
        guard.release ();

        // After becoming the leader, the thread uses
        // the reactor to wait for an event.
        reactor_->handle_events ();

        // Reenter monitor to serialize the test
        // for <leader_thread_> in the while loop.
        guard.acquire ();
    }
}
```

在for循环中，调用线程在leader（领导者）线程、processing（处理）线程和follower（追随者）线程之间切换自己的角色。在该循环的第一部分，线程一直等待，直到可以成为领导者，在这里它使用反应器在共享句柄集上等待事件。当反应器在句柄上检测到一个事件时，它将事件多路分解到与它关联的事件处理程序，并分配给它的handle_event()方法以达到提升新领导者和对事件进行处理的目的。在反应器多路分解了一个事件后，该线程再次恢复追随者角色。这些步骤持续循环，直到应用程序结束或发生超时。❑ 462

(5) *实现追随者提升协议*。紧接着领导者线程检测到一个事件之后，同时也在它多路分解事件到事件处理程序并处理事件之前，领导者线程必须将一个追随者线程提升为新的领导者。如下两个子活动可以用来实现这一协议：

(5.1) *实现句柄集同步协议*。如果句柄集是迭代的，并且我们盲目地提升了一个新的领导者线程，那么可能新的领导者线程会试图处理与前一个领导者线程检测到的并正在进行处理的同一事件。为了避免这种竞争条件，必须在将一个追随者句柄提升为领导者句柄并将事件分配给具体事件处理程序之前，将该句柄从句柄集的候选句柄中删除。分配并处理事件后，必须从句柄集中再次激活该句柄。

➡应用程序可以实现具体事件处理程序，该处理程序是由反应器模式的实现活动（1.2）中定义的Event_Handler类派生的子类。同样，领导者/追随者实现可以使用装饰器模式[GoF95]创建修饰Event_Handler的LF_Event_Handler类。该装饰器提升一个新的领导者线程，并以与具体事件处理程序无关的方式激活/停用反应器句柄集中的处理程序。

```
class LF_Event_Handler : public Event_Handler {
public:
    LF_Event_Handler (Event_Handler *eh,
                      LF_Thread_Pool *tp)
        : concrete_event_handler_ (eh),
          thread_pool_ (tp) { }

    virtual void handle_event (HANDLE h, Event_Type et) {
        // Temporarily deactivate the handler in the
        // reactor to prevent race conditions.
        thread_pool_->deactivate_handle (h, et);

        // Promote a follower thread to become leader.
        thread_pool_->promote_new_leader ();

        // Dispatch application-specific event
        // processing code.
        concrete_event_handler_->handle_event (h, et);

        // Reactivate the handle in the reactor.
        thread_pool_->reactivate_handle (h, et);
    }
private:
    // This use of <Event_Handler> plays the
    // <ConcreteComponent> role in the Decorator
    // pattern, which is used to implement
    // the application-specific functionality.
    Event_Handler *concrete_event_handler_;

    // Instance of an <LF_Thread_Pool>.
    LF_Thread_Pool *thread_pool_;
};
```

463 ❑

(5.2) *确定提升协议排序*。有几个排序策略可以用于确定提升哪个追随者线程：

- LIFO顺序。在许多应用程序中，下一次提升哪个追随者线程不重要，因为所有线程都是对等的。在这种情况下，领导者线程可以按照后进先出（LIFO）的顺序提升追随者线程。LIFO协议通过确保等待时间最短的线程首先被提升[Sol98]将CPU高速缓存的相似性最大化[SKT96][MB91]，这是一个失效前刷新工作模式的例子[Mes96]。

  如果最近被阻塞的线程在调度程序再次运行的时候执行几乎相同的代码和数据，高速缓存的变相似性可以改善系统性能。然而，实现LIFO提升协议需要附加的数据结构（如等待线程栈），而不是仅使用诸如信号灯这样的本地操作系统同步对象。
- 优先级顺序。在某些应用程序中，特别是实时应用程序中，线程可能运行在不同的优先级上。在这种情况下，需要根据追随者线程的优先级对它们进行提升。这种协议可以使用某些类型的优先级队列实现，如堆[BaLee98]。虽然该协议比LIFO协议更复杂，但是为了使优先级反序[SMFG00]的情况最少，就需要按照追随者线程的优先级对它们进行提升。
- 由实现定义的顺序。在使用操作系统同步器（如信号灯或条件变量）来实现句柄集时常用这种顺序，通常按照由实现定义的顺序分配等待线程。这种协议的优点是它能高效地映射到本地操作系统同步器上。

464

➡OLTP后端数据库服务器可以使用如下的简单协议对追随者线程进行提升，所采用的顺序是原来的操作系统条件变量将它们放入队列的顺序：

```
void LF_Thread_Pool::promote_new_leader () {
    // Use Scoped Locking idiom to acquire mutex
    // automatically in the constructor.
    Guard<Thread_Mutex> guard (mutex_);

    if (leader_thread_ != Thread::self ())
        throw /* ...only leader thread can promote... */;

    // Indicate that we are no longer the leader
    // and notify a <join> method to promote
    // the next follower.
    leader_thread_ = NO_CURRENT_LEADER;
    followers_condition_.notify ();

    // Release mutex automatically in destructor.
}
```

正如实现活动(5.1)所示，promote_new_leader()方法在它转发对事件进行处理的具体事件处理程序前被LF_Event_Handler装饰器调用。 ❑

(6) *实现事件处理程序*。应用程序开发者必须决定当具体事件处理程序的钩子方法被领导者/追随者模式实现中的处理线程调用时，实现什么动作。反应器模式的实现活动（5）描述了各种与实现具体事件处理程序关联的问题。

465

8. 已解决的例子

“例子”一节描述的OLTP后端数据库服务器可以使用领导者/追随者模式实现从套接字句柄高效地多路分解I/O事件给事件处理程序的线程池。在这种设计中，没有指定的网络I/O线程。相反，线程池在数据库服务器初始化过程中进行预分配：

```
const int MAX_THREADS = /* ... */;

// Forward declaration.
void *worker_thread (void *);

int main () {
    LF_Thread_Pool thread_pool (Reactor::instance ());
    // Code to set up a passive-mode Acceptor omitted.
    for (int i = 0; i < MAX_THREADS - 1; ++i)
        Thread_Manager::instance ()->spawn
            (worker_thread, &thread_pool);

    // The main thread participates in the thread pool.
    thread_pool.join ();
};
```

这些线程没有绑定到任何特殊套接字句柄。因此，线程池中的所有线程通过调用LF_Thread_Pool::join()方法轮流扮演网络I/O线程的角色：

```
void *worker_thread (void *arg) {
    LF_Thread_Pool *thread_pool =
        static_cast <LF_Thread_Pool *> (arg);

    // Each worker thread participates in the thread pool.
    thread_pool->join ();
};
```

如实现活动（4）所示，join()方法仅允许领导者线程使用Reactor单件对连接到OLTP前端通信服务器的共享的Socket句柄集调用select()。如果请求到达时所有线程忙，那么请求将在套接字句柄中排队，直到线程池中的线程可以执行请求。

一个请求事件到达时，领导者线程将select()的句柄集中的套接字句柄临时停用，然后将一个追随者线程提升为新的领导者，并作为处理线程继续处理请求事件。接着，该处理线程将请求读入驻留在运行时栈中的缓冲区，或者读入使用线程特定的存储器模式分配的缓冲区。[⊖] 所有OLTP活动都在处理线程中发生。因此，不需要进一步的语境切换、同步化或数据移动，直到处理结束。当请求处理结束后，处理线程重新扮演追随者的角色，在线程池中的同步器上等待。此外，处理线程处理的套接字句柄和其他句柄集中的Socket一起，在Reactor单件的句柄集中被重新激活，这样select()可以等待在该套接字句柄上发生I/O事件。 466

9. 变体

绑定句柄/线程关联。本模式前面几节描述了非绑定句柄/线程关联，其中在线程和句柄之间没有固定关联。因此，任何线程都可以处理在句柄集中的任何句柄上的任何事件。在工作者线程池轮流多路分解共享句柄集时通常使用非绑定关系。

领导者/追随者模式的一个变体使用绑定句柄/线程关联。在该变体中，每个线程绑定到它自身的句柄，此句柄用来处理特殊事件。当线程在套接字句柄上等待它向服务器发送的双向请求的响应时，应用程序的客户机通常使用绑定关联。在这种情况下，客户机应用线程希望在同一线程的发送最初请求的这个句柄上处理响应事件。

⊖ 相反，在“例子”一节描述的半同步/半反应线程池必须从共享堆中动态分配每个请求，因为请求是在线程间传递的。

因此，在绑定句柄/线程关联变体中，如果领导者线程没有处理事件所必需的语境，那么线程池中的领导者线程可能需要将事件传递给追随者线程。在领导者检测到一个新的事件后，它检查与事件关联的句柄，确定哪个线程负责处理该事件。如果领导者线程发现它自己负责该事件，则它将一个追随者线程提升为新的领导者。反之，如果事件应由别的线程处理，则领导者必须将事件传递给指定的追随者线程。然后该追随者线程就可以暂时禁用句柄，并处理该事件。同时，当前领导者线程则继续等待其他事件在句柄集上发生。

467

图5-28说明了后续状态和处理状态之间的附加转换：

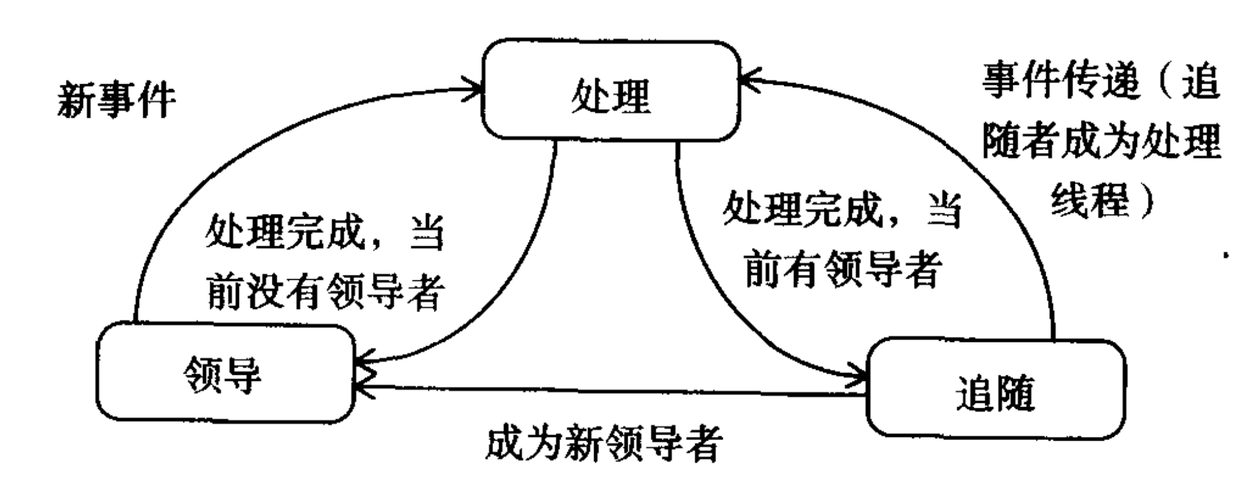


图 5-28

可以隐式地维护领导者/追随者线程池，例如，使用一个同步器（如信号灯或条件变量）；也可以显式地维护，如使用容器和管理者模式[Som97]。维护方式的选择很大程度上取决于领导者线程是否必须显式地通知一个特定追随者线程进行事件传递。

对绑定的句柄/线程关联变体和它的实现的详细讨论可以参考[SRPKBOO]。

放松串行化约束。有些操作系统允许多个领导者线程在一个句柄集上同时等待。例如，Win32函数`WaitForMultipleObjects()`[Sol98]支持允许线程池在同一句柄集上并发等待的并发句柄集。因此，使用该函数设计的线程池可以在其他线程等待事件时利用多处理器硬件并发处理多个事件。

领导者/追随者模式的两种变体可以用于多领导者线程同时活动：

- *用于多句柄集的领导者/追随者*。该变体将传统的领导者/追随者实现分别应用于多句柄集。例如，每个线程被分配一个指定的句柄集。该变体在可使用多句柄集的应用程序中特别有用。然而，该变体限制线程只能使用特定的句柄集。
- *多领导者和多追随者*。在该变体中，扩展模式以支持多个同步领导者线程，其中任何领导者线程都可以在任何句柄集上等待。当一个线程再次加入线程池中时，它检查领导者是否已与每个句柄集关联。如果存在没有领导者的句柄集，再次加入的线程就可以立即成为该句柄集的领导者。

468

混合线程关联。某些应用程序使用混合设计以同时实现绑定和非绑定句柄/线程关联。同样，一个应用程序中的某些句柄可能用专用线程处理某些特定事件，而其他句柄则可以被任何线程处理。因此，领导者/追随者模式变体按照事件活动发生的句柄的情况，使用它的事件移交机制通知某些线程子集。

➡例如，OLTP前端通信服务器可能有多个线程使用领导者/追随者模式等待从客户机发出的新请求事件。同样，可能也有线程等待对它们在后端服务器上调用请求产生的响应。实际上，

线程在它们的生存期扮演着两种角色，由作为分配新输入请求的线程开始，然后向后端服务器发出请求以满足客户机应用程序的需求，最后等待从后端服务器到来的响应。 ❑

混合客户机/服务器。在对等应用程序同时扮演客户机和服务器角色的复杂系统中，通信基础设施在等待一个或多个应答时能处理到达的请求是很重要的。否则，系统会因为一个客户机将它的所有线程阻塞以等待响应而发生死锁。

在此变体中，线程和句柄的绑定动态地发生改变。例如，一个线程可能最初未绑定，然而当处理一个应用程序发现的到达的请求时，它需要分布式系统中另一个对等体提供的服务。在这种情况下，未绑定线程在执行应用程序代码时分配一个新的请求，有效地将自己绑定到发送请求的句柄。然后，在响应到达并且线程完成最初的请求后，它将再次变成未绑定状态。 469

选择性事件源和汇。考虑不仅通过句柄而且通过其他事件源所获得的事件的系统（如共享内存或消息队列）。例如，在UNIX中没有可以同时等待I/O事件、信号灯事件和/或消息队列事件的事件多路分解功能。然而，在同一时刻线程也可以阻塞以等待某一类型的事件。因此，可以扩展领导者/追随者模式来同时等待多种类型的事件：

- 与整个系统中只有一个领导者线程相比，每个事件源都被分配一个领导者线程。
- 在事件被接收后，还未处理前，领导者线程可以选择任何追随者线程在事件源上等待。

然而，这种变体的缺点是参与的线程数必须要大于事件源的数量。因此，当事件源数增加的时候，这种方法也许不能很好地扩展。

10. 已知使用

ACE 线程池反应器框架[Sch97]。ACE框架提供了一个称为“线程池反应器”（ACE_TP_Reactor）的使用领导者/追随者模式的面向对象框架实现，在线程池中反应器将事件多路分解给事件处理程序。使用线程池反应器时，应用程序预先生成固定数量的线程。当这些线程调用ACE_TP_Reactor的handle_events()方法时，一个线程将变成领导者并等待事件。线程被ACE线程池反应器框架看做非绑定的。因此，在领导者线程检测到事件后，它会将任意一个线程提升为下一个领导者，然后将事件多路分解给与事件关联的事件处理程序。

CORBA ORB和Web服务器。许多CORBA实现（包括Chorus COOL ORB[SMFG00]和TAO[SC99]）在客户机端连接模型和服务器端并发模型中都使用领导者/追随者模式。此外，JAWS Web服务器[HPS99]为操作系统平台使用领导者/追随者线程池模型，该平台不允许多线程在被动模式套接字句柄上同时调用accept()。 470

事务监视器。流行的事务监视器（如Tuxedo）一般以每进程为基础进行操作，例如，事务总是与一个进程关联。然而，如今的OLTP系统要求高性能和可扩展性，并且以每个进程为基础的事务可能不满足这些要求。因此，下一代事务服务，如CORBA事务服务[OMG97b]的实现，在线程和事务间使用绑定领导者/追随者关联。

出租车候车台。在日常生活中领导者/追随者模式用于管理许多飞机场出租车候车台。在该用例中，出租车扮演“线程”的角色，排在第一辆的出租车成为领导者，剩下的出租车成为追随者。同样，到达出租车候车台的乘客构成了必须被多路分解给出租车的事件，一般以先进先出顺序。一般说来，如果任何出租车可以为任何顾客服务，该场景就主要相当于在“实现”一节描述的非绑定句柄/线程关联。然而，如果仅仅是某些出租车可以为某些乘客服务，该场景就

相当于在“变体”一节描述的绑定句柄/线程关联。

11. 结论

领导者/追随者模式提供了几个**优点**：

性能增强。与“例子”一节描述的半同步/半反应线程池方法相比，领导者/追随者模式以如下方式改善了性能：

- 增强了CPU高速缓存相似性并消除了动态内存分配和线程间共享的数据缓冲区要求。例如，处理线程可以将请求读入在其运行时栈上分配的缓冲区中或者读入通过使用线程特定的存储器模式分配的内存中。
- 通过在线程间不交换数据的方法来使加锁开销达到最小，因此降低了线程同步化。在绑定句柄/线程关联中，领导者线程基于句柄值将事件多路分解给它的事件处理程序。然后由处理事件的追随者线程从句柄中读出请求事件。在非绑定关联中，领导者线程自身将请求事件从句柄中读出，并对它进行处理。
- 可以将优先级逆序的数量减少到最小，因为在服务器中没有进行其他排队。当与实时I/O子系统[KSL99]结合时，领导者/追随者线程池模型可以大大减少服务器请求处理过程中的不确定性根源。
- 不需要语境切换以处理每个事件，减少了事件分配延时。注意，提升追随者线程以履行领导者职能一定需要语境切换。如果两个事件同时到达，则增加了第二个事件的分配延时，但是性能不会比半同步/半反应线程池实现更差。

471

编程简单性。领导者/追随者模式简化了并发模型的编程，其中多线程可以使用共享句柄集接收请求、处理响应并多路分解连接。

然而，领导者/追随者模式有如下**不足**：

实现复杂性。领导者/追随者模式的高层变体比半同步/半反应线程池更难实现。特别地，用做多线程连接多路分解器时，领导者/追随者模式必须维护一个追随者线程池等待处理请求。在一个追随者线程被提升为领导者并且在有线程重新加入追随者线程池时必须更新该集合。所有这些操作可能会以不可预知的顺序并发地发生。因此，领导者/追随者模式实现在确保操作原子性的同时必须高效。

缺乏灵活性。基于半同步/半异步模式的半同步/半反应变体的线程池模型允许排队层中的事件被丢弃或重新排列优先级。同样，系统可以维护多个由优先级不同的线程服务的独立队列，以减少不同优先级事件间的争用和优先级倒置。然而，因为没有显式的队列，所以在领导者/追随者模型中很难丢弃或重排序事件。一种支持该功能的方法是在应用程序中使用多领导者/追随者组提供不同层次的服务，每个服务层由具有不同优先级的线程服务。

472

网络I/O瓶颈。如“实现”一节描述的，领导者/追随者模式通过每次仅允许一个线程在句柄集上等待来实现串行处理。在某些环境中，由于每次仅有一个线程可以多路分解I/O事件，所以该设计可能成为瓶颈。然而，实际上并没有关系，因为绝大多数I/O密集的处理都由操作系统内核完成。因此，应用层I/O操作可以快速地执行。

参见

反应器模式通常构成领导者/追随者模式实现的核心。然而，反应器模式可以在每个事件仅

需要短时间进行处理时用来代替领导者/追随者模式。在这种情况下，可以避免领导者/追随者模式的复杂性的调度。

主动器模式为并发多路分解异步事件完成定义了另一个模型。在下列条件满足时，它可以用来替代领导者/追随者模式：

- 当操作系统能有效支持异步I/O时，并且
- 当程序员适应与主动器模式有关的控制的异步逆转。

半同步/半异步和主动对象模式是领导者/追随者模式的另外两个替代模式。如下情形出现时，这些模式可能是比领导者/追随者模式更合适的选择：

- 在由线程池中的线程处理它们之前，有附加的同步或排序约束必须通过对队列中的请求重新排序来解决时，和/或
- 当单个事件多路分解器不能有效地等待事件源时。

受控的反应器（Controlled Reactor）模式[DeFe99]包括一个遵守用户规范的为事件处理程序控制线程使用的性能管理器，当受控性能是一个重要的目标时，它是一种可能的选择。 473

致谢

Michael Kircher、Carlos O'Ryan和Irfan Pyarali是领导者/追随者模式初版的作者。感谢Ed Fernandez为帮助改善本版本提出的建议。 474

5.5 线程特定的存储器

线程特定的存储器（Thread-Specific Storage）设计模式允许多线程使用一个“逻辑上全局的”访问点获得一个局限于某一线程的对象，而不会导致对象访问中的加锁开销。

1. 别名 线程限制存储器（Thread-Local Storage）。

2. 例子

考虑对一个多线程网络登录服务器的设计，远程客户机应用程序用该服务器来在分布式系统中集中记录其状态信息。反应器模式例子中给出的登录服务器在单个线程中反复地多路分解所有客户机连接，而这里的登录服务器使用每个连接一个线程（thread-per-connection）[Sch97]的并发模型并发地处理请求。如图5-29所示。

在每个连接一个线程的模型中，为每个客户机连接建立一个单独线程。各线程从与之关联的TCP Socket中读取登录记录，处理这些记录并将它们写入到相应的输出设备，如登录文件或打印机。 475

每个登录服务器线程也负责检测和报告各种低层网络条件或在进行I/O处理时发生的系统错误。许多操作系统（如UNIX和Windows NT）通过一个称为errno的全局访问点将这些低层信息报告给应用程序。在进行系统调用（如read()或write()）时，如果出现错误或异常状态，操作系统将设置errno以指出发生了什么，并返回一个特定状态值（如-1）。应用程序必须判断这些返回值，然后检查errno以确定发生了何种类型的错误或异常状态。

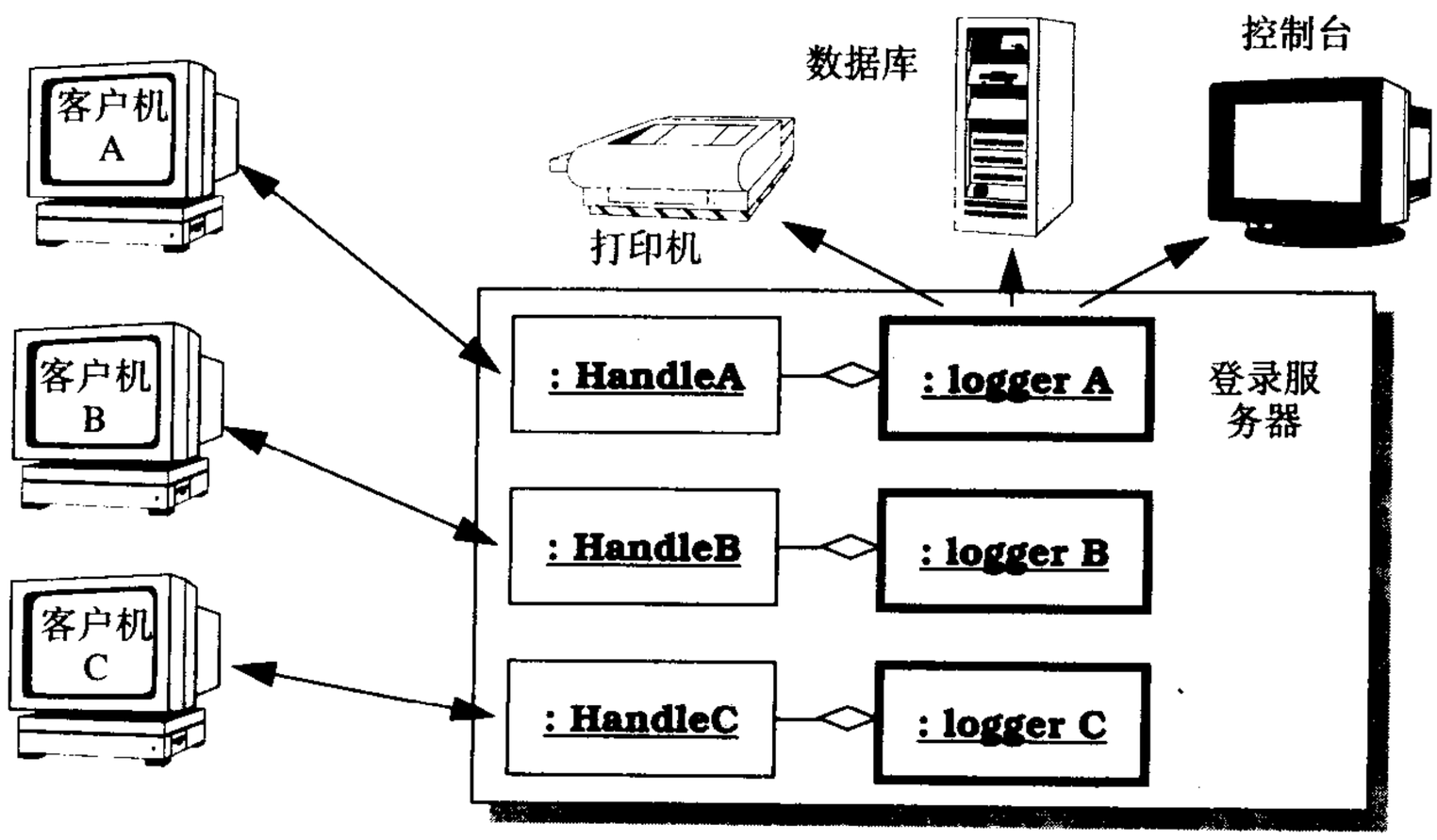


图 5-29

考虑如下从TCP 套接字句柄集接收客户机登录记录到非阻塞模式[Ste98]的C代码段：

```
// One global <errno> per-process.
extern int errno;

void *logger (HANDLE socket) {
    // Read logging records until connection is closed.
    for (;;) {
        char log_record[MAXREC];
        if (recv (socket, log_record, MAXREC, 0) == -1) {
            // Check to see why <recv> failed.
            if (errno == EWOULDBLOCK)
                sleep (1); // Try getting data later.
            else // Display error result.
                cerr << "recv failed, errno=" << errno;
        } else // Normal case ...
    }
}
```

如果recv()返回-1，那么登录服务器logger代码检查errno以确定发生了什么，并决定如何处理。

虽然将errno实现为全局量的方式在单线程的应用程序中工作得很好，但是它可能在多线程应用程序中出现难于捉摸的问题。特别地，抢先多线程系统中的竞争条件会导致一个线程中的errno值集在别的线程中被错误地解释。如果多线程同时执行了logger()函数，则将发生错误的交互。

例如，假定线程T_1调用一个返回-1的非阻塞recv()，并将errno设定为EWOULDBLOCK表示当前没有数据在Socket上排队。然而，在T_1可以检查这一情况之前，它被抢先了，且线程T_2开始运行。假设T_2被一个异步信号（如SIGALRM）中断，T_2将errno置为EINTR。如果T_2由于它的时间片用完而立即被抢先，则T_1会错误地假定它的recv()调用被中断，并执行错误的动作（如图5-30）。

476

对这个问题一个显而易见的解决方案是应用包装器外观模式，使用一个包含锁的对象包装器封装errno。定界加锁惯用法可以用来在设置或检查errno之前获得锁以及后来释放它。可是，该方案不能解决竞争条件问题，因为对全局errno值的设置和检查操作不是原子的。相反，

它包括如下两项活动：

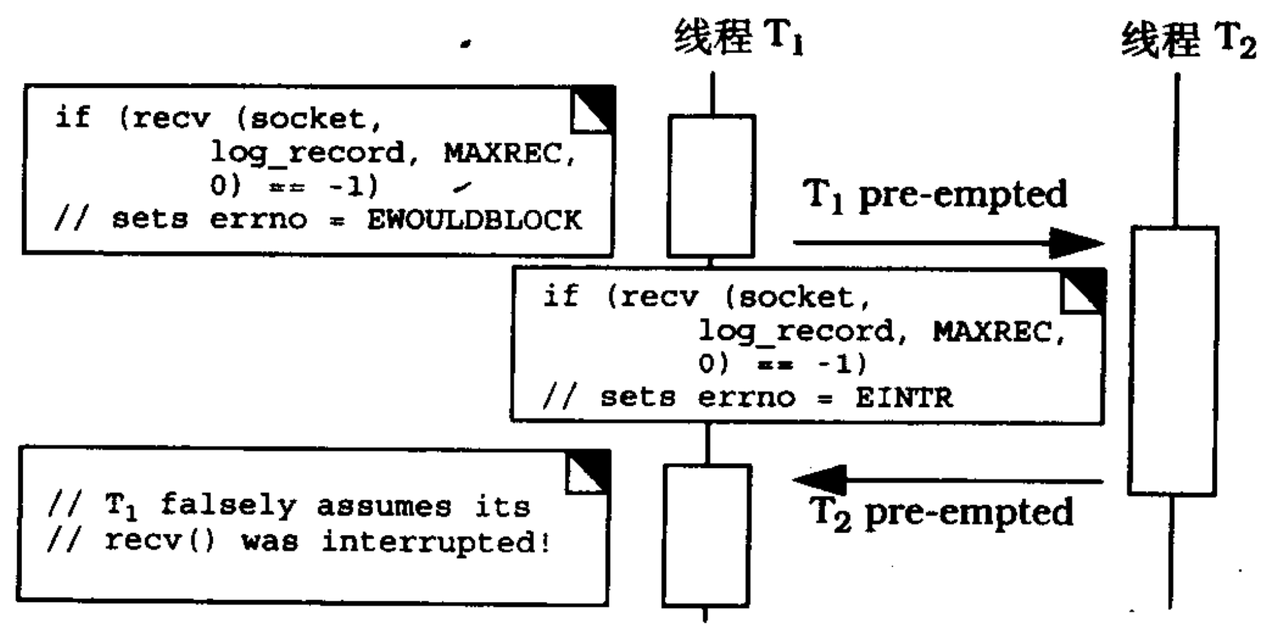


图 5-30

1) recv()调用设置errno。

2) 应用程序检查errno，确定采取什么行为。

防止竞争条件的一种更健壮的方法是改进errno加锁协议。例如，recv()系统调用可以在设置errno前获得一个全局errno_lock。然后，当recv()返回后，应用程序在判断了errno的值后释放errno_lock。然而，这种解决方案容易出错，因为应用程序可能忘记释放errno_lock，从而导致资源耗尽和死锁。同样，因为应用程序可能需要经常检查errno的状态，所以会产生额外的加锁开销，从而极大地降低性能，特别是当应用程序在单线程配置下运行的时候。

因此我们需要的是透明地给每个线程自身一个“逻辑上全局的”对象（如errno）本地拷贝的机制。

477

3. 语境

经常访问逻辑上全局的数据或对象的多线程应用程序，但对每个线程来讲，这些数据或对象的状态物理上应该是本地的。

4. 问题

为了避免竞争条件、资源耗尽和死锁[Lea99a]多线程应用程序需要复杂的并发控制协议，从而难以编程。由于存在加锁开销，所以多线程应用程序的性能往往比不上单线程应用程序，事实上它们的性能可能更糟，特别是在多处理器平台上[SchSu95]。在并发程序中有两个强制条件：

- 多线程应用程序应既容易编程又高效。特别地，对逻辑上全局的但物理上局限于一个线程的数据的访问应是原子的，且不会导致对每次访问的加锁开销。[⊖]

  ➡如在“例子”一节描述的，操作系统常常将errno作为“逻辑上全局的”变量来实现，开发者把它真的当做一个实际的全局变量进行编程。然而，为了避免竞争条件，在本地分配用来存储errno的内存，每个线程一次。 ❑

⊖ 注意该用况与多线程在使用全局或共享数据的单任务上协作的情况相反。在这种情况下，数据不是针对具体线程的，并且每个线程对数据的访问必须通过同步机制（如互斥或信号灯）来控制。

- 许多遗产库和应用程序最初是在单一控制线程的假定下编写的。因此它们常常在方法间通过诸如errno的全局对象隐式地传递数据，而不是显式地传递参数。将这些代码转移到多线程中运行时，通常不太可能改变遗产应用程序中现存的接口和代码。

  ➡隐式地由errno返回错误状态代码的操作系统不可能轻易地被改为显式地返回这些错误代码而不分隔应用程序和库组件。 ❑

5. 解决方案

478 为每个针对具体线程的对象引入一个全局访问点，但是在每个线程的存储器中保持“真实”的对象。应用程序仅通过它们的全局访问点管理这些线程特定对象。

6. 结构

线程特定的存储器模式由六个参与者组成。

线程特定对象是一个只能由特定线程访问的对象实例（如图5-31）。

➡例如，在支持多线程处理的操作系统中，errno是一个int，在每个线程中都有不同的实例。 ❑

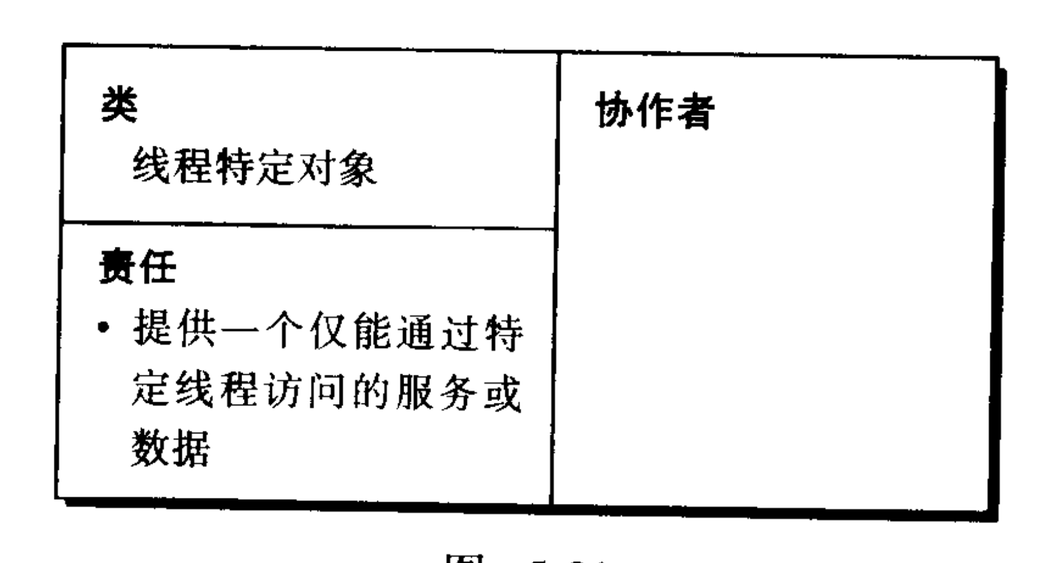


图 5-31

线程使用一个由关键字工厂分配的关键字来标识一个线程特定对象。由关键字工厂生成的关键字有一个取值范围，以确保每个线程特定对象在“逻辑上”是全局的（如图5-32）。

➡例如，多线程操作系统通过建立一个全局惟一的关键字实现errno。每个线程使用该关键字隐式地访问其自身的本地errno实例。 ❑

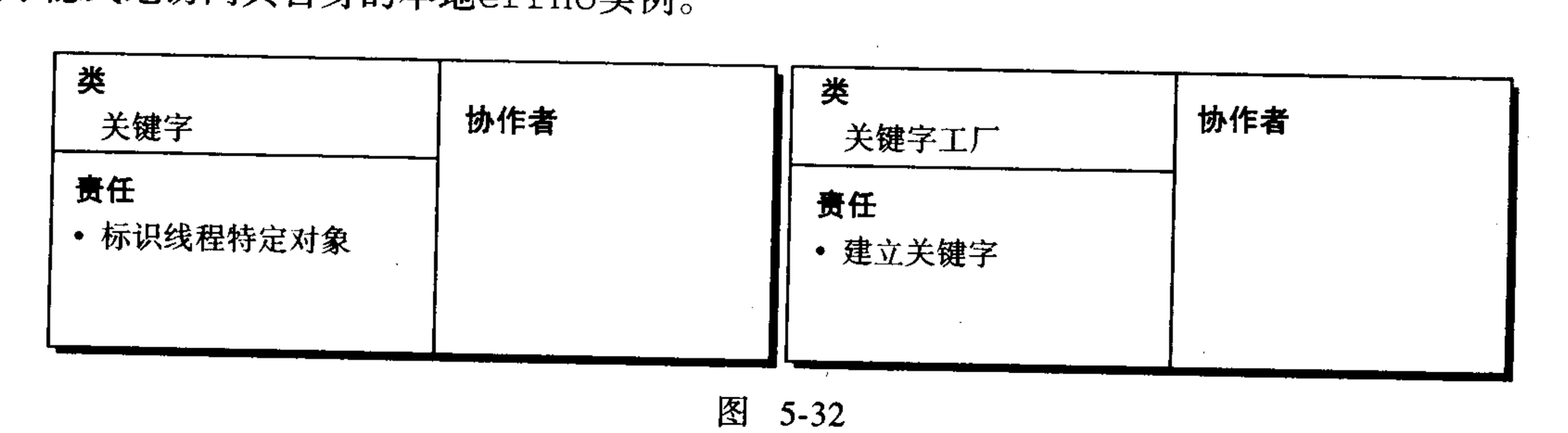


图 5-32

一个线程特定对象集包含与特定线程关联的线程特定对象的集合。每个线程有自己的线程特定对象集。在线程特定对象集内部定义了一对方法，set()和get()，将全局管理的关键
479 字集合映射到集合中存储的线程特定对象。通过向get()传递一个标识对象的关键字作为参数，线程特定对象集的客户端可以获得一个指向某一线程特定对象的指针。客户机可以通过

由get()方法返回的指针检查或修改对象。类似地，客户机可以通过将对象指针以及相关的关键字以参数形式传递给set()来向对象集添加一个指向线程特定对象的指针。如图5-33所示。

➡通常由操作系统的线程库实现线程特定对象集。该集合包含errno数据以及其他线程特定对象。 ❑

类 线程特定对象集	协作者 • 线程特定对象
责任 • 维护线程特定对象	

图 5-33

可以定义一个线程特定对象代理[GoF95][POSA1]，使客户机像访问普通对象一样访问一个特定类型的线程特定对象。如果不使用代理，那么客户机必须直接访问线程特定对象集并显式地使用关键字，这样做单调乏味并且易出错。每个代理实例存储一个惟一区分线程特定对象的关键字，这样，每个关键字和每个线程都对应一个线程特定对象。

线程特定对象代理与相关的线程特定对象的接口一致。在内部，代理的接口方法首先使用由线程特定对象集提供的set()和get()方法，获得一个由被存储在代理中的关键字指定的指向线程特定对象的指针。在指针被获得后，代理将最初的方法调用委托给它。

➡例如，errno实现为预处理宏，它扮演代理的角色并保护应用程序免受线程特定的操作的影响。 ❑

应用程序线程是客户机，它使用线程特定对象代理访问驻留在线程特定的存储器中的线程特定对象。对于应用程序线程而言，当对一个线程特定对象进行实际方法调用时，该方法看上去好像是在对一个普通对象进行调用。多应用线程可以使用相同的线程特定对象代理访问它们惟一的线程特定对象。代理使用调用它的接口方法的应用线程的标识符，来区分它所封装的线程特定对象。如图5-34所示。 480

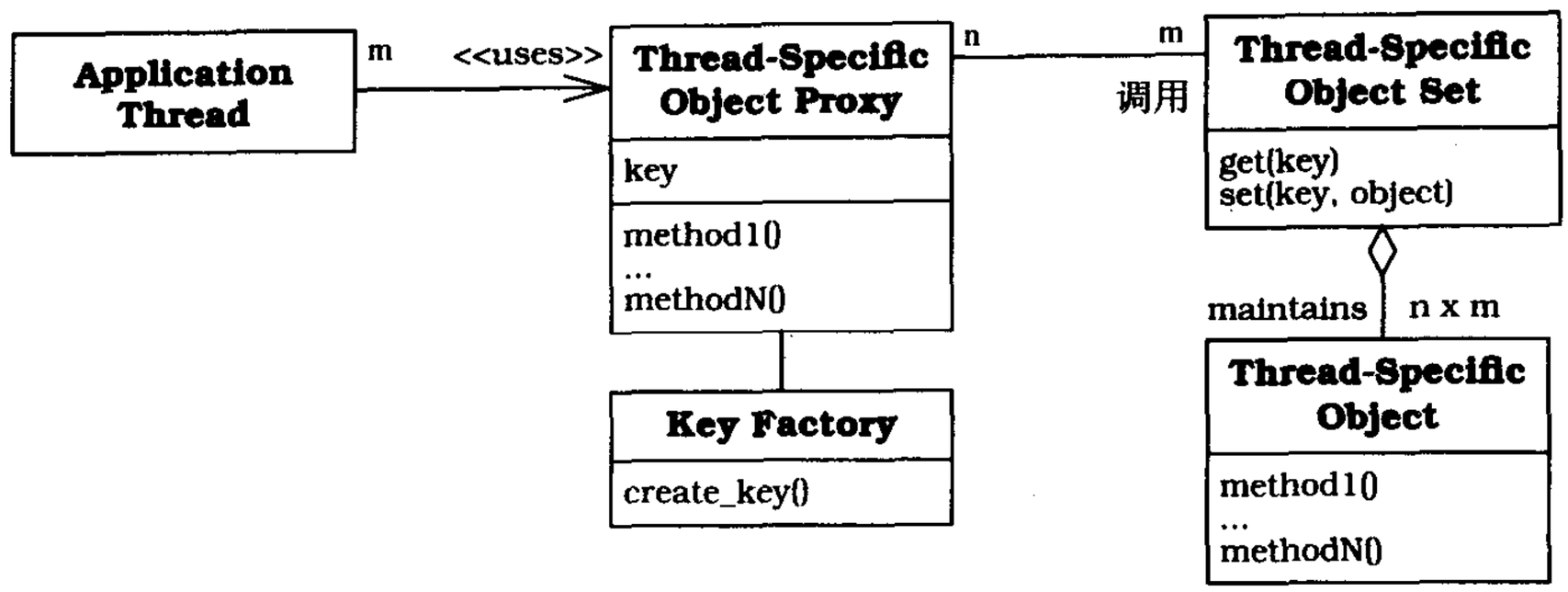
➡例如，“例子”一节中运行logger函数的线程是应用程序线程。 ❑

类 线程特定对象代理	协作者 • 线程特定对象集 • 线程特定对象
责任 • 简化对线程特定对象集和关键字的应用程序线程编程，以访问特定类型的线程特定对象	

类 应用程序线程	协作者 • 线程特定对象代理
责任 • 通过它们的线程特定的代理访问线程特定对象	

图 5-34

类图5-35说明了线程特定的存储器模式的一般结构：



481

图 5-35

线程特定的存储器模式的参与者可以在概念上用二维矩阵进行建模，该矩阵的行对应关键字，列对应线程。矩阵k行t列处的元素是指向相应线程特定对象的指针。创建一个关键字等价于向矩阵加入一行，创建一个新线程等价于加入一列。

线程特定对象代理与线程特定对象集一起为应用程序线程提供一个类型安全机制，以访问处于k行和t列上的特定对象。关键字工厂保存了已用关键字的个数。一个线程特定对象集包含某一列的条目。如图5-36所示。

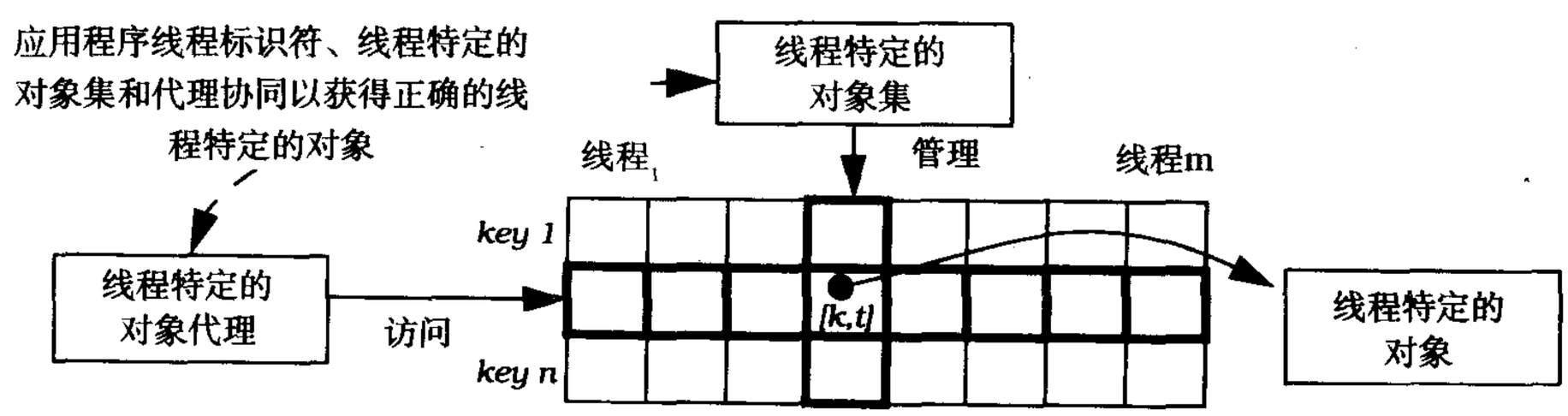


图 5-36

注意上述模型只是个类比。实际上，实现线程特定的存储器模式并不使用二维矩阵，因为关键字未必是连续整数。线程特定对象集的条目也可以驻留在它们相应的线程中，而不是使用一个全局二维矩阵。然而，将线程特定的存储器模式的结构形象化为一个二维矩阵是有用的。因此我们在下面小节引用该比喻。

7. 动态特性

在线程特定的存储器模式中有两个普通场景：*建立*和*访问*线程特定对象。**场景 1** 描述了线程特定对象的建立（如图5-37）：

- 应用程序线程调用线程特定对象代理接口中定义的方法。
- 如果代理还没有相关的关键字，它将要求关键字工厂建立一个新关键字。该关键字在每个线程的对象集中惟一标识相关线程特定对象。然后代理存储该关键字，以优化随后的由应用程序线程完成的方法调用。

482

- 线程特定对象代理动态地建立一个新对象。然后它使用线程特定对象集的set()方法将指向该对象的指针存储到关键字指定的位置。
- 接着执行由应用程序线程调用的方法，如场景Ⅱ所示。

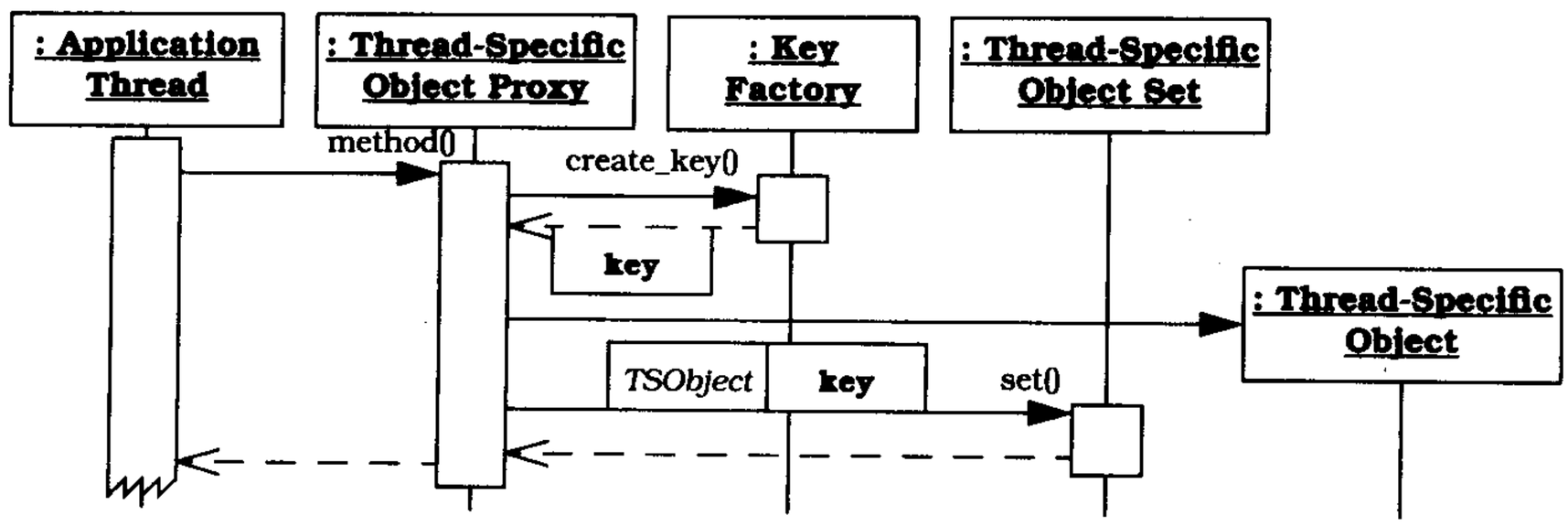


图 5-37

场景II描述了应用程序线程如何访问已有的线程特定对象（如图5-38）：

- 应用程序线程在线程特定对象代理上调用一个方法。
- 线程特定对象代理将它存储的关键字传递给应用程序线程的线程特定对象集的get()方法。然后获得指向相应线程特定对象的指针。
- 代理使用该指针将初始方法调用委托给线程特定对象。注意，这里不需要加锁，因为对象是通过一个只在客户机应用程序线程内部访问的指针引用的。

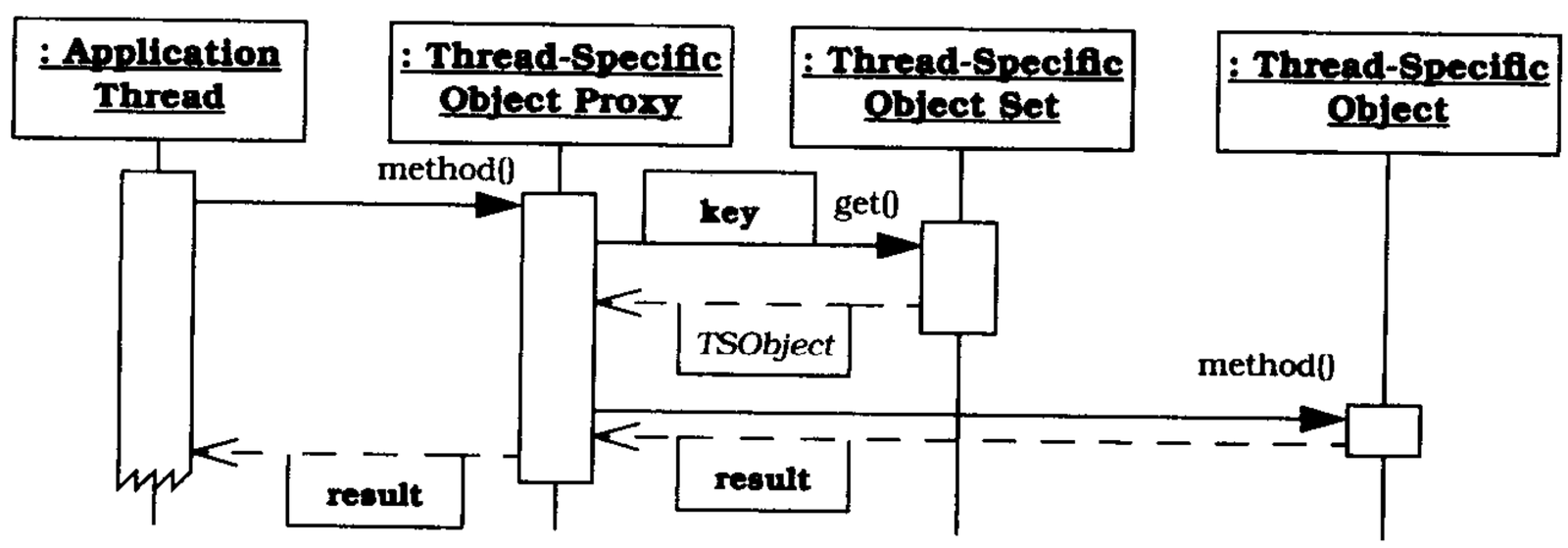


图 5-38

483

8. 实现

实现线程特定的存储器模式的焦点是线程特定对象集和线程特定对象代理的实现。这两个组件建立了管理和访问驻留在线程特定的存储器中的对象的机制。因此将它们的实现（包括潜在替代品）描述为两个独立的活动，从线程特定对象集开始，然后是线程特定对象代理。

和访问它们的应用程序代码一样，线程特定对象自身，是由应用程序开发者定义的，因此这里不提供这些模式参与者的一般实现活动。不过，在“已解决的例子”一节，我们使用多线程登录服务器例子说明在应用程序中如何高效地编程实现线程特定的存储器模式。

(1) 实现线程特定对象集。该活动分为六项子活动。

(1.1) 确定线程特定对象的类型。按照二维矩阵类比，线程特定对象是矩阵中的一个条目，

它具有如下属性：

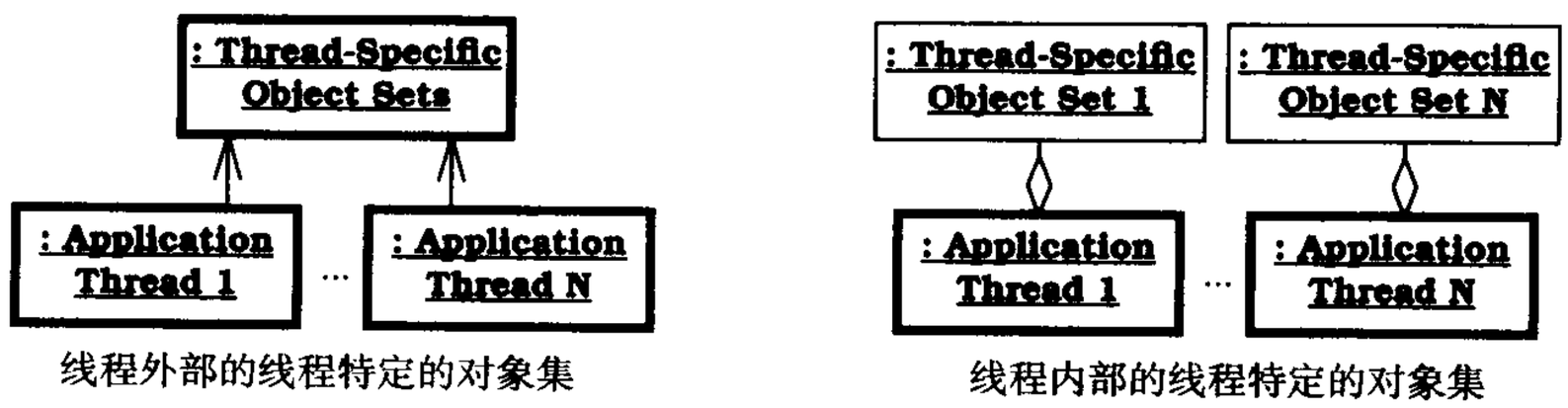
- 行编号对应于惟一标识“逻辑上全局的”对象的关键字。
- 列编号对应于特殊应用程序线程标识符。

为了实现更通用的线程特定的存储器模式，要存储指向线程特定对象的指针而不是对象自身。这些指针常常是“松散类型化的”，如C/C++ void*，因此它们可以指向任何类型的对象。虽然松散类型定义的void *非常灵活，但是很难对它们进行正确地编程。因此实现活动(2)描述了几种策略，使用更不易出错的、强类型化的代理类封装void *。

(1.2) 确定将线程特定对象集存储在何处。按照二维矩阵类比，线程特定对象集对应矩阵的列，对每个应用程序线程都分配一个线程特定对象集。因此每个应用程序线程标识符表示这个概念性的二维矩阵中的一列。

484

每个线程特定对象集既可以存储在所有线程外也可以存储在线程内（如图5-39）：



线程外部的线程特定的对象集　　线程内部的线程特定的对象集

图 5-39

每种策略各有利与弊：

- 在所有线程外部。该策略将每个应用程序线程的标识符映射为一个存储在所有线程外部的全局线程特定对象集表。注意，应用程序线程可以通过调用线程库中的API函数获得它自身的线程标识符。因此，外部线程特定对象集的实现可以容易地确定哪个线程特定对象集与指定应用程序线程相关联。

取决于外部表策略的不同实现，线程可以访问其他线程中的线程特定对象集。首先，该设计可能看上去使整个线程特定的存储器模式观点落空，因为对象和指针自身并不驻留在线程特定的存储器中。但是，如果线程特定的存储器实现可以在不再需要关键字时回收它们，这也许是有用的，例如，应用程序由于某些原因不再需要访问一个全局对象，如errno。

全局表有助于由“清除”线程访问所有线程特定对象集以删除对应于被回收关键字的条目。关键字的回收对于仅支持有限数量关键字的线程特定的存储器模式实现特别有用。例如，Windows NT将每个进程的关键字限制在64个。实时操作系统通常只支持更少的关键字。

485

将线程特定对象集存储在线程外部的全局表中的缺点是，增加了访问每个线程特定对象的开销。该开销是由每次修改包含所有线程特定对象集的全局表时，为避免竞争条件而使用的同步机制所引起。特别地，当关键字工厂建立一个新关键字时，需要进行串行化，因为其他应用程序线程可能在并发地建立关键字。然而，在相应线程特定对象集被确定后，应用程序线程不需要进行任何加锁操作就可以访问集合中的线程特定对象。

- 在每个线程内部。这种策略要求每个线程连同它的其他内部状态，如它的运行时线程栈、程序计数器、通用寄存器和线程标识符，一同存储在线程特定对象集中。当线程访问一个

线程特定对象时，使用相应的关键字作为到线程内部的线程特定对象集的索引，从而获取该对象。和上述外部策略不同，当线程特定对象集存储在每个线程内部时，不需要进行串行化。在这种情况下，所有对线程内部状态的访问都发生在线程内部。

然而，虽然不用消耗更多的总内存，但在每个线程本地存储线程特定对象集时每个线程需要更多状态。只要大小的增长不会导致线程建立、语境切换或销毁的开销的显著增长，线程特定对象的内部策略就比外部策略更有效。

如果操作系统提供适当的线程特定的存储器机制，可以通过本地操作系统机制在每个线程内部实现线程特定对象集。否则，线程特定对象集可以使用两级映射策略在外部实现。在该策略中，本地线程特定的存储器机制中有一个关键字专门用于指向在各线程外部实现的线程特定对象集。

(1.3) *定义一个将应用程序线程标识符映射到线程特定对象集的数据结构*。按照二维矩阵类比，应用程序线程标识符映射到矩阵中代表线程特定对象集的列。

应用程序线程标识符的取值可以很小也可以很大。大的取值范围表明可以处理驻留在每个线程内部的对象集。在这种情况下，线程标识符隐式地与相应包含在线程状态中的线程特定对象集相关联。因此，不需要实现一个独立的数据结构来将应用程序线程标识符映射到线程特定对象集。 486

然而，对于驻留在所有线程外部的线程特定对象集，为每个可能的线程标识符值都使用一个固定大小的数组是不切实际的。这就是为什么二维矩阵类比仅仅是一个概念模型而不是一个现实实现策略的一个原因。在这种情况下，使用一个动态数据结构将线程标识符映射到线程特定对象集更具有空间效率。

例如，一个通常的策略是使用线程标识符计算一个散列函数，获得散列表上的偏移量。在该偏移量上的入口包含一个元组链，这些元组将线程标识符映射到与它们相对应的线程特定对象集。

(1.4) *定义在线程特定对象集中将关键字映射到线程特定对象的数据结构*。按照二维矩阵类比，这种映射通过它的行（关键字）上某一列（与特定应用程序线程标识符关联的线程特定对象集）区分一个特定矩阵条目（线程特定对象）。对于外部和内部线程特定对象集实现，必须为这种映射同时选择定长或变长数据结构。

如果线程特定的关键字值的范围相对较小并且连续，线程特定对象集可以存储在定长数组中。例如，POSIX Pthreads标准[IEEE96]定义了一个标准宏`_POSIX_THREAD_KEYS_MAX`设置Pthreads实现能支持的关键字数量的最大值。如果该宏定义的值很小并且固定，例如64个关键字，那么通过直接使用区分线程特定对象的关键字索引线程特定对象集数组，搜寻时间将是$O(1)$。

然而，某些线程特定的存储器实现提供的线程特定的关键字的范围很大并且几乎无界。例如，Solaris线程没有对应用进程中线程特定的存储器的关键字数量预先定义一个范围。Solaris因此使用一个变长数据结构（如散列表）将关键字映射到线程特定对象。虽然这种数据结构比定长数组更灵活，但是在分配大量关键字时会增加管理线程特定对象集的开销。 487

➡如下代码说明如何使用存储为`void *`的定长线程特定对象数组在线程内部实现线程特定对象集。使用这种内部设计意味着不需要将应用程序线程标识符映射到线程特定对象集中。相反，只需要提供一个在线程特定对象集中将关键字映射到线程特定对象的数据结构。

“实现”一节的所有C代码例子取自POSIX Pthreads [IEEE96] 中一个公用用户层库实现[Mue93]。例如，对应实现活动（1.4）的object_set_数据结构包含在如下的thread_state struct结构中，该struct被Pthreads库实现用来存储每个线程的状态：

```
struct thread_state {
    // Thread-specific 'object' set implemented via
    // void *'s.
    void *object_set_[_POSIX_THREAD_KEYS_MAX];
    // ... Other thread state.
};
```

除了跟踪指向线程特定的存储器对象的指针数组之外，thread_state的实例也包含其他线程状态。该状态包含一个指向线程栈和一段存储空间的指针，这段存储空间用于在语境切换时保存和恢复存储线程特定的寄存器。Pthreads实现还定义了几个宏以简化它的内部编程：

```
// Note that <errno>'s key number is 0, i.e.,
// it is in first slot of array object_set_.
#define ERRNO_KEY 0
// Define a macro that's used internally to the Pthreads
// implementation to set and get <errno> values.
#define INTERNAL_ERRNO \
        (pthread_self ()->object_set_[ERRNO_KEY])
```

INTERNAL_ERRNO宏使用的pthread_self()函数是一个内部实现子例程，返回一个指向当前活动线程语境的指针。 ❑

488

(1.5) 定义关键字工厂。在二维矩阵类比中，关键字对应于矩阵中的行。关键字工厂生成一个新关键字以惟一标识“逻辑上全局的”对象（行）。该对象的状态将物理地驻留在局限于每个线程的存储器中。

对于每个线程中逻辑上全局的但物理上局部的特定对象，所有线程使用同一关键字值*k*访问相应的线程特定对象。因此，存储的当前使用的关键字个数可以对所有线程是全局的。

➡下面的代码说明了pthread_key_create()关键字工厂的Pthreads库实现。用整数值表示关键字：

```
typedef int pthread_key_t;
```

一个静态变量跟踪当前线程特定的存储器实现中关键字的个数：

```
// All threads share the same key counter.
static pthread_key_t total_keys_ = 0;
```

total_keys_变量在每次有新的线程特定的关键字被请求时会自动增加，这等价于向概念性的二维矩阵加入新的一行。下一步，我们定义关键字工厂自身：

```
int pthread_key_create (pthread_key_t *key,
                        void (*thread_exit_hook)(void *)) {
    if (total_keys_ >= _POSIX_THREAD_KEYS_MAX) {
        // Use our internal <errno> macro.
        INTERNAL_ERRNO = ENOMEM;
        return -1;
    }
```

```
    thread_exit_hook_[total_keys_] = thread_exit_hook;
    *key = total_keys_++;
    return 0;
}
```

pthread_key_create()函数是一个关键字工厂，它分配一个新关键字惟一标识线程特定的数据对象。该函数不需要内部同步化，因为它必须在拥有一个外部锁的情况下被调用，如实现活动(2.1)中的TS_Proxy所示。

当建立一个关键字后，pthread_key_create()函数允许调用线程将thread_exit_hook与新关键字关联。该钩子是一个指向函数的指针，所指向的函数可以用来删除任何动态分配的与关键字关联的线程特定对象。当一个线程退出时，Pthread库自动为每个注册了退出钩子的关键字调用该函数指针。 489

为了实现该特性，可以将一个指向“线程退出钩子”的函数指针数组作为静态全局变量存储在Pthreads库中：

```
// Array of exit hook function pointers that can be used
// to deallocate thread-specific data objects.
static void
(*thread_exit_hook_[_POSIX_THREAD_KEYS_MAX]) (void *);
```

pthread_exit()函数说明在线程退出前如何回调退出钩子函数：

```
// Terminate the thread and call thread exit hooks.
void pthread_exit (void *status) {
    // ...
    for (i = 0; i < total_keys_; ++i)
        if (pthread_self ()->object_set_[i]
            && thread_exit_hook_[i])
            // Indirect pointer to function call.
            (*thread_exit_hook_[i])
                (pthread_self ()->object_set_[i]);
    // Terminate the thread and clean up internal
    // resources...
}
```

对于每个关键字，应用程序可以注册相同的函数指针、不同的函数指针或函数指针的任意组合。当每个线程退出时，Pthreads实现调用在每个关键字创建时注册的函数。应用程序通常按如下方法实现线程退出钩子，因为删除动态分配的线程特定对象的操作是公共的：

```
void cleanup_tss_object (void *ptr) {
  // This cast is necessary to invoke the
  // appropriate destructor (if one exists).
  delete (Object_Foo *) ptr;
}
```

❑

(1.6) 定义从线程特定对象集存储和获得线程特定对象的方法。按照矩阵类比，这两个方法设置和获得矩阵条目的值。set()方法在矩阵条目[k,t]上存储一个void *，而get()方法在矩阵条目[k,t]上获得一个void *。在线程特定的存储器实现中，k作为一个关键字参数进行传递，t是调用pthread_self()返回的隐式线程标识符。 490

➥pthread_setspecific()函数是一个set()方法，使用调用它的客户机应用线程传

递的关键字存储一个void*：

```
int pthread_setspecific (pthread_key_t key, void *value) {
    if (key < 0 || key >= total_keys) {
        // Use our internal <errno> macro.
        INTERNAL_ERRNO = EINVAL;
        return -1;
    }
    // Store value into appropriate slot in the thread-
    // specific object set.
    pthread_self ()->object_set_[key] = value;
    return 0;
}
```

同时，pthread_getspecific()函数使用客户机应用线程传递的关键字来获得一个void*：

```
int pthread_getspecific (pthread_key_t key, void **value){
    if (key < 0 || key >= total_keys) {
        // Use our internal <errno> macro.
        INTERNAL_ERRNO = EINVAL;
        return -1;
    }
    *value = pthread_self ()->object_set_[key];
    return 0;
}
```

在该实现中，上述两个函数都不需要锁来访问其线程特定对象集，因为该集合驻留在每个线程状态的内部。 ❑

(2) 实现线程特定对象代理。在理论上，上述用C编写的线程特定对象集足够用来实现线程特定的存储器模式。然而，由于如下两个原因，实践中依赖于低层的C函数API不是我们所期望的：

- 虽然流行的线程库（如POSIX Pthreads、Solaris线程和Win32线程）中的线程特定的存储器API都是相似的，但它们在语义上有微妙的差异。例如，Win32线程，与POSIX Pthreads及Solaris线程不同，没有提供一个可靠的途径在线程退出时将线程特定的存储器中分配的对象释放。相反，在Solaris线程中，没有删除关键字的API。这些不同的语义使编写能在所有这些平台上正确运行的代码非常困难。

491

- POSIX Pthreads、Solaris和Win32线程特定的存储器API将指向线程特定对象的指针作为void*存储。虽然该方法提供了极大的灵活性，但是它是易出错的，因为void*略去了类型安全性检查。

为了克服这些限制，线程特定的存储器模式定义了一个线程特定对象代理。每个代理应用代理模式[GoF95][POSA1]定义一个对象作为线程特定对象的“代理”。在代理上调用方法的应用线程看起来，就像是访问一个普通对象，而实际上代理将方法传递给线程特定对象。该设计防止应用程序了解线程特定的存储器的使用时间和方式。它也允许应用程序使用高层的、类型安全的以及平台无关的包装器外观访问由低层C函数API管理的线程特定对象。

线程特定对象代理的实现可以分为三个子步骤：

(2.1) 定义线程特定对象代理接口。对于线程特定对象，有两个设计代理接口的策略，多态

和参数化类型：

- 多态。在该策略中，一个抽象代理类声明并实现每个代理支持的数据结构和方法。例子包含线程特定对象集与某一特殊“逻辑上全局的”对象关联的关键字，或建立关键字时为避免竞争条件所必需的锁。

  对由线程特定对象提供的具体方法的访问功能是由通用代理的子类所提供，每类线程特定对象对应一个类。在将客户机应用程序请求传递给线程特定对象中的相应方法之前，代理首先通过存储在代理中的关键字从线程特定对象集获取一个对象指针。

  使用多态机制实现代理是一个一般策略[POSA1]。然而，动态绑定导致的额外间接寻址使它可能产生开销。

- 参数化类型。在该策略中，可以用将要驻留在线程特定的存储器中的对象类型将代理参数化。正如上述多态策略描述的，代理机制仅仅声明和实现代理所支持的数据结构和方法。在线程特定对象上调用指定的方法前，代理机制也在线程特定对象集上进行所有必需的操作。参数化可以删除与多态有关的间接性，以改善代理的性能。 492

  使用参数化类型策略引发的一个关键的设计问题，是如何选择一种方便的机制访问被代理封装的线程特定对象的方法。特别地，不同的线程特定对象类型具有不同的接口。访问这些对象的机制因此不能定义任何具体的服务方法。这不同于上述多态策略。

  解决这个问题的一种方法是使用智能指针[Mey95]（如`operator->`），在C++里也称为箭头操作符[Str97]。该操作符允许客户机应用线程访问代理时，感觉就好像它们在直接访问线程特定对象一样。C++编译器对`operator->`方法进行特殊的处理。它首先获得一个指向相应类型的线程特定对象的指针，然后将最初调用的方法委托给它。

  另一个访问线程特定对象方法的一般途径是应用扩展接口模式。这种解决方案为代理引入一种通用方法，该方法允许客户机检索由已配置的线程特定对象支持的具体接口。 493

➡在例子中，使用了C++参数化类型来定义一个类型安全模板，应用程序可以用具体线程特定对象来实例化线程特定对象代理：

```
template <class TYPE>
class TS_Proxy {
public:
    // Constructor and destructor.
    TS_Proxy ();
    ~TS_Proxy ();

    // Define the C++ '->' and '*' operators to access
    // the thread-specific <TYPE> object.
    TYPE *operator-> () const;
    TYPE &operator* () const;
private:
    // Key that uniquely identifies the 'logically
    // global' object that 'physically' resides locally
    // in thread-specific storage.
    mutable pthread_key_t key_;

    // "First time in" flag.
    mutable bool once_;

    // Avoid race conditions during initialization.
    mutable Thread_Mutex keylock_;
```

```
    // A static cleanup hook method that deletes
    // dynamically allocated memory.
    static void cleanup_hook (void *ptr);
};
```

该线程特定的代理模板由对象的类型来参数化，而通过线程特定的存储器来访问该对象。此外，它定义了C++智能指针operator->以访问类型为TYPE的线程特定对象。 ❑

(2.2) *实现线程特定对象代理的建立和析构*。不论是应用多态还是参数化类型策略定义线程特定对象代理，都必须对线程特定对象代理的建立和析构进行管理。

494

➡线程特定对象代理模板类的构造函数是最小化的，它仅仅将对象的数据成员初始化：

```
template <class TYPE>
TS_Proxy<TYPE>::TS_Proxy (): once_ (false), key_ (0) { }
```

❑

一般来说，代理的构造函数内部并不分配关键字或新的线程特定对象实例，有两个原因：

- *线程特定的存储器语义*。不同于使用代理的线程，线程特定对象代理通常由一个线程（例如应用程序的主线程）来创建。因此，在构造函数中，预初始化一个新的线程特定对象并不能获得什么好处，因为该实例只能由创建它的线程访问。
- *延迟建立*。在某些操作系统中，关键字资源是有限的，而且应该直到绝对需要时才被分配。关键字的建立因此应当延迟，直到代理方法第一次调用。在实现例子中，这一时间点发生在operator->方法中。

线程特定对象代理的析构函数提供了几个富有技巧的设计。“显而易见的”解决方案是释放由关键字工厂分配的关键字。然而，该方法有几个问题：

- *不可移植性*。很难编写一个可移植的释放关键字的代理析构函数。例如，Solaris线程（与Win32和POSIX Pthreads不同）没有释放不需要的关键字的API。
- *竞争条件*。Solaris线程不提供释放关键字的API的原因是它很难有效和正确地实现。问题在于每个线程都保持关键字所引用对象的独立的拷贝。只有所有线程都退出，并且内存被回收后才能安全地释放一个关键字。

这些问题导致了代理的析构函数一般是“空操作（no-op）”。这意味着在关键字的实现中不对它进行回收。因此，我们实现了一个线程退出钩子函数取代析构函数，这在实现活动（1.5）中进行了讨论。该钩子在线程退出时被线程特定的存储器实现自动分配。它将线程特定对象删除，以确保对象的析构函数被调用。

495

➡TS_Proxy类的析构函数是一个“空操作”：

```
template <class TYPE>
TS_Proxy<TYPE>::~TS_Proxy () { }
```

为了确保调用正确的析构函数，线程退出钩子在删除它之前将其ptr参数传递给一个指向适当TYPE的指针：

```
template <class TYPE>
void TS_Proxy<TYPE>::cleanup_hook (void *ptr) {
    // This cast invokes the destructor (if one exists).
    delete (TYPE *) ptr;
}
```

注意，TS_Proxy类中的cleanup_hook()定义为static方法。这样，可以将它作为一个从指针到函数的线程退出钩子，传递给pthread_key_ create()。 ❑

(2.3) 实现对线程特定对象的访问。前面讨论过，有两种通用策略——多态和参数化类型——来访问由代理表示的线程特定对象的方法。

使用多态策略时，具体代理的接口必须包括所有该类中线程特定对象提供的方法。具体代理的方法实现通常执行如下四个步骤：

- 如果还没有创建这种线程特定对象的话，创建一个新关键字。必须防止多线程为同一TYPE的线程特定对象同时创建新关键字，并因此可以避免竞争条件。可以通过应用双检查加锁优化模式解决该问题。
- 下一步，方法必须使用由代理存储的关键字，以通过线程特定对象集获得线程特定对象。
- 如果对象还不存在，则“按需”建立该对象。
- 所请求的操作被转发到线程特定对象。所有操作结果都被返回给客户机应用线程。

为了避免在每个代理方法中重复这些代码，建议在实现这些一般步骤的线程特定对象代理基类中引入一个帮助程序（helper）方法。 496

使用参数化类型实例化一个一般代理时，可以使用实现活动（2.1）中描述的智能指针和扩展接口模式策略，实现访问任何线程特定对象方法的通用机制。和多态策略一样，通用访问机制必须遵循上述实现步骤。

➡通过使用参数化类型策略和重载C++矢量操作符operator->，应用程序可以调用TS_Proxy实例上的方法，就好像在调用一个TYPE参数实例上的方法一样。C++箭头操作符控制所有对TYPE类的线程特定对象的访问。它使用下面的代码完成大部分工作：

```
template <class TYPE>
TYPE *TS_Proxy<TYPE>::operator->() const {
    TYPE *tss_data = 0;
    // Use the Double-Checked Locking Optimization
    // pattern to avoid excessive locking.
    if (!once_) {
        // Use Scoped Locking idiom to ensure <keylock_>
        // is acquired to serialize critical section.
        Guard <Thread_Mutex> guard (keylock_);
        if (!once_) {
            pthread_key_create (&key_,
                                &cleanup_hook);
            // Must come last so that other threads
            // don't use the key until it's created.
            once_ = true;
        }
        // <Guard> destructor releases the lock.
    }
    // Get data from thread-specific storage. Note that no
    // locks are required, because this thread's own copy
    // of the thread-specific object will be accessed.
    pthread_getspecific (key_, (void **) &tss_data);

    // Check if it's the first time in for this thread.
    if (tss_data == 0) {
        // Allocate memory dynamically off the heap,
        tss_data = new TYPE;
```

```
        // Store pointer in thread-specific storage.
        pthread_setspecific (key_, (void *) tss_data);
    }
    return tss_data;
}
```

497

TS_Proxy模板是一个代理，它将普通C++类转换为其实例驻留在线程特定的存储器中的类型安全类。它将operator->方法与C++特性（如模板、内嵌和重载）结合。此外，它使用通用并发控制模式和惯用法，如双检查加锁优化、定界加锁和策略化加锁。

operator->两次使用双检查加锁优化模式以测试once_标志位。虽然多线程可以同时访问TS_Proxy的同一实例，但是仅有一个线程可以正确地通过pthread_key_create()方法建立关键字。接着所有线程使用该关键字访问参数化TYPE类的与之相关的线程特定对象。因此operator->方法使用一个Tread_Mutex类型的keylock_确保一次仅有一个线程执行pthread_key_create()。

key_建立后，访问线程特定对象再不需要其他加锁，因为pthread_getspecific()和pthread_setspecific()函数都从客户机应用线程的状态中获得TYPE类的线程特定对象，客户机应用线程独立于其他线程。除了降低加锁开销外，上述TS_Proxy类的实现保护应用程序代码不受对象位于调用线程内这一事实的影响。 ❑

扩展接口和多态代理的实现与前述一般智能指针方法类似。多态代理方法简单地调用线程特定对象的方法并返回结果。类似地，扩展接口方法从线程特定对象返回扩展接口并将它传回客户机。

9. 已解决的例子

如下应用程序类似于“例子”一节中最初的登录服务器。下面给出的logger()函数是到每一线程的入口点，此处线程具有到远程客户机应用程序的惟一连接。主要区别是logger()函数使用实现活动（2.3）中定义的TS_Proxy模板类访问errno值。

498

该模板由如下Error_Logger类实例化：

```
class Error_Logger { // Define a simple logging API.
public:
    // Return the most recent error residing in thread-
    // specific storage.
    int last_error ();

    // Format and display a logging message.
    void log (const char *format, ...);
    // ...
};
```

Error_Logger类定义了“逻辑上”全局的、但是“物理上”线程特定的登录对象的类型，通过如下TS_Proxy线程特定对象代理模板创建登录对象：

```
static TS_Proxy<Error_Logger> my_logger;
```

logger()函数被日志服务器中的每个连接处理线程调用。使用包装器外观模式中描述的SOCK_Stream类从网络连接读取数据，而不是直接访问低层C Socket API：

```
static void *logger (void *arg) {
```

```
    // Network connection stream.
    SOCK_Stream *stream =
        static_cast <SOCK_Stream *> arg;

    // Read a logging record from the network connection
    // until the connection is closed.
    for (;;) {
        char log_record[MAXREC];

        // Check to see if the <recv> call failed, which
        // is signified by a return value of -1.
        if (stream->recv (log_record, MAXREC) == -1) {
            if (my_logger->last_error () == EWOULDBLOCK)
                // Sleep a bit and try again.
                sleep (1);
            else // Record error result.
                my_logger->log
                    ("recv failed, errno = %d",
                     my_logger->last_error ());
        } else // Other processing.
    }
}
```

考虑上面对my_logger->last_error()方法的调用。C++编译器生成的代码用两个方法调用替代这一调用。第一个是对TS_Proxy::operator->的调用，返回相应驻留在线程特定的存储器中的Error_Logger实例。然后编译器生成对由前一调用返回的Error_Logger对象中的last_error()方法的第二个方法调用。在这种情况下，TS_Proxy作为代理允许应用程序和普通C++对象一样访问和管理线程特定的错误值。 499

10. 变体

从JDK 1.2开始，Java通过java.lang.ThreadLocal类支持线程特定的存储器模式。java.lang.ThreadLocal类中的对象是一个线程特定对象代理，对应二维矩阵类比中的一行。ThreadLocal对象通常作为静态变量在内部建立，因此它们在大范围内是可见的。ThreadLocal内部散列表保存了线程特定对象的入口，每个入口对应一个线程。这些入口的类型是Object，这意味着散列表不知道它保存对象的具体类型。因此应用程序必须维护相关信息并进行必要的向下类型转换，其利弊分析在实现活动（1.1）和（2）中有讨论。

Java应用程序线程可以通过调用foo.set(newValue)设置ThreadLocal对象foo的值。ThreadLocal类型的foo对象使用线程标识符从散列表返回线程的当前对象入口。散列表是一个普通数据结构，但是可以通过调用Collections.synchronizedMap(hashtable)将一个线程安全层包装在散列表外层。该特性将装饰器[GoF95]和线程安全接口模式结合以确保现存的Java集合能被正确地串行化。

java.lang.InheritableThreadLocal类是ThreadLocal类的扩展。该子类允许一个子线程从它的父线程继承所有线程特定对象，它们的值预置为当前父类的值。

11. 已知使用

广泛使用线程特定的存储器模式的例子是支持errno机制的操作系统平台，如Win32和Solaris。如下是Solaris在<errno.h>中对errno的定义：

```
#define errno (*(___errno()))
```

500

基于在实现活动（1）中描述的低层C语言的线程特定的存储器函数，由该宏调用的__errno()函数可以实现如下：

```
int *___errno () {
    // Solaris ensures that static synchronization
    // objects are always initialized properly.
    static pthread_mutex_t keylock;
    static pthread_key_t key;
    static int once;
    int *error_number = 0;

    if (once) {
        // Apply Double-Checked Locking Optimization.
        pthread_mutex_lock (&keylock);
        if (once) {
            // Note that we pass in the <free> function
            // so the <error_number> memory will be
            // deallocated when this thread exits!
            pthread_key_create (&key, free);
            once = 1;
        }
        pthread_mutex_unlock (&keylock);
    }
    // Use <key> to retrieve <error_number> from the
    // thread-specific object set.
    pthread_getspecific (key, &error_number);
    if (error_number == 0) {
        // If we get here, then <error_number> has not
        // been created in this thread yet. Thus, we'll
        // create it and store it into the appropriate
        // location in the thread-specific object set.
        error_number = (int *) malloc (sizeof (int));
        pthread_setspecific (key, error_number);
    }
    return error_number;
}
```

Win32 GetLastError()和SetLastError()函数实现了类似的线程特定的存储器模式。

在Win32操作系统API中，窗口由线程拥有[Pet95]。拥有窗口的每个线程有自己的消息队列，操作系统将用户接口事件放入该队列。然后事件处理API调用将调用线程的消息队列中的下一个消息移出队列，消息队列驻留在线程特定的存储器中。

501

COM 中的**主动模板库**使用扩展接口方法实现线程特定的存储器模式。

OpenGL[NDW93]是一个用来绘制三维图形的C API。程序借助多边形来绘制图形，而多边形是通过反复调用glVertex()函数，将多边形每个顶点传递给函数库来描述的。在顶点被传递给函数库之前状态变量集可以准确地确定OpenGL接收到顶点时绘出什么内容。该状态作为一个封装的全局变量存储在OpenGL函数库或图形卡中。在Win32平台上，OpenGL函数库在线程特定的存储器中为每个使用函数库的线程保存了惟一的状态变量集。

线程特定的存储器在ACE框架[Sch97]中用来实现它的错误处理方案，该方案类似于在“已解决的例子”一节描述的方法。此外，ACE使用C++模板实现了类型安全的线程特定对象代理，如实现活动（2）所述。ACE线程特定的存储器模板类称为ACE_TSS。

本地电话查号服务。线程特定的存储器模式一个现实生活中的例子是电话查号服务。例如，在美国，“逻辑上全局的”电话号码411可以接通到本地查号台接线员，查询某个地区的区号。

12. 结论

使用线程特定的存储器模式有四个**优点**：

高效。可以实现线程特定的存储器模式，使得访问线程特定的数据时不需要加锁。例如，通过将errno放入线程特定的存储器，每个线程可以可靠和高效地设置和测试线程中所调用的方法的完成状态，而无需使用复杂的同步协议。该设计消除了在线程中共享数据时的加锁开销，比获得以及释放一个互斥锁更快[EKBF+92]。

可重用性。通过应用包装器外观模式，并将可重用的线程特定的存储器模式代码与应用程序特定的类相分离，可以使开发者避免考虑复杂并且不可移植的线程特定的关键字的建立和分配逻辑。例如，可以与双检查加锁优化模式集成，形成一个可重用的线程特定对象代理组件以自动防止竞争条件。 502

易用性。使用包装器外观封装时，线程特定的存储器相对来说易于被应用程序员使用。例如，通过抽象（如线程特定对象代理模板或像errno这样的宏）线程特定的存储器可以完全隐藏在源代码层。将一个类改为线程特定的类或者相反仅仅需要改变类中对象定义的方式。

可移植性。线程特定的存储器在大多数多线程操作系统平台上都能找到。它可以方便地在需要它的平台上实现，如VxWorks或pSoS。进一步讲，线程特定对象代理可以将依赖于平台的操作封装在统一的和可移植的接口内。因此将应用程序移植到另一个线程库（如Win32的TLS接口）只需要改变TS_Proxy类，而不改变使用类的应用程序代码。

然而，使用线程特定的存储器模式也有如下**不足**：

它鼓励对（线程特定的）全局对象的使用。许多应用程序不需要多线程通过公共访问点访问线程特定的数据。在这种情况下，应该存储数据以便只有拥有数据的线程才能访问它。

➡考虑使用线程池处理来自客户机的登录记录的登录服务器。除了将登录记录写入永久存储器之外，每个线程都可以记录它完成的服务数量和类型。该登录机制可以作为一个全局Error_Logger对象通过线程特定的存储器访问。然而有一个更简单的方法（虽然可能效率较低并且更加冒失），即用主动对象代表每个登录线程，将Error_Logger实例作为数据成员而不是线程特定的存储器进行存储。在这种情况下，Error_Logger可以作为一个数据成员被主动对象方法访问或作为参数传递给主动对象调用的所有外部方法或函数。 ❑

它使系统结构变得模糊。线程特定的存储器的使用由于模糊了组件之间的关系，因此可能使应用程序更难理解。例如，从登录服务器的源代码中不能明显看出每个线程具有自己的Error_Logger实例，因为my_logger和一个普通全局对象很像。在某些情况下，可以通过显式地表示组件之间的包含或聚合关系来消除对线程特定的存储器的需求。 503

它限制了实现方式的选择。不是所有语言都支持参数化类型或智能指针，并且不是所有应用程序类都提供扩展接口。线程特定对象代理的“优雅的”实现解决方案并不能应用于所有的系统。这种情况发生时，必须来用不太完善和较低效的解决方案（如多态或低层函数，来实现线程特定的存储器模式）。

参见

如errno的线程特定对象通常作为完全线程（per-thread）的单件使用[GoF95]。然而线程特定

的存储器并不都是作为单件使用的，因为线程可以具有从线程特定的存储器分配的某种类型的多个实例。例如，通过ACE_Task[Sch97]实现的每个主动对象实例都存储一个线程特定的清除钩子。

线程特定的存储器模式与数据所有权模式[McK95]有关，后一模式中线程协调客户机对某一对象的访问。

致谢

Tim Harrison和Nat Pryce是线程特定的存储器模式初始版本的共同作者。感谢Tom Cargill对本模式初始版本的建议。

504

第6章 将模式组织在一起

“语言上的局限就是对我们视野的限制。”
路德维希·维特根斯坦

“没有哪个模式是孤岛，自成一体；
每个模式都是整个大陆的一块，是整体的一部分。”
约翰·多恩《祈祷》(Devotions)释义

本书中的模式可以独立应用，每个模式都有助于解决一个与并发和网络化有关的某些强制条件。然而，单独使用这些模式限制了模式的能力，因为采用孤立地解决问题的方式并不能有效地开发现实世界中的软件系统。

为发挥本书的更大作用，本章要说明第2章到第5章中出现的模式是如何彼此连接、互补以及支持的，以形成模式语言的基础来建立高质量的分布式对象计算中间件，以及并发和网络化应用程序。并且，我们将指出这些模式中有多少可以应用于并发和网络化语境之外。

6.1 从独立模式到模式语言

如[POSA1]中的模式一样，第2章出现的模式是以一种自包含方式描述的。例如，模式的表示尽可能通用，以避免将其应用范围限制在对其他问题、模式或设计的特定配置上。因此，一旦出现模式能解决的问题时，就可以应用这些模式。并且，模式的解决方案描述和它们的实现指南均不关注由其他模式描述的类似问题的解决方案。每个模式仅仅引用有助于实现它自身解决方案结构和动态特性的模式。

6.1.1 没有模式是孤立的

只考虑个别模式并不能有效地支持现实世界中软件系统的构造。例如，正如第1章“并发和网络化对象”的Web服务器例子中所描述的，许多内部相互联系的设计问题和强制条件必须在开发并发和网络化系统时得到解决。在解决关键设计和实现问题时必须考虑这些关系。因此，不管它们的作用是什么，独立的模式只能解决孤立的小问题，因为它们并不考虑应用这些模式的更大语境。

即使独立模式在某种程度上连接起来，使它们具有解决更大问题的潜力，也很难从它们的描述中提取它们的关系。例如，如果两个模式描述相互引用，则哪个模式应在另一个模式之前应用可能不很明显。然而在使用模式开发软件应用和系统时，模式应用的顺序可能对它们的成功集成至关重要[Bus00a]。

正确排序对体系结构模式特别重要，这些模式引入一个结构，为整个软件系统定义基线体系结构。该结构中每个组件本身是“复杂的”，常常可以使用其他设计模式实现。因此准确地表示这些模式之间的关系以确定哪个模式先应用，哪个模式后应用至关重要。

506

例如，反应器体系结构模式引入一个参与者——事件处理程序，它的具体实现可以定义自己的并发模型。因此在反应器模式的“变体”一节中，对事件处理程序实现的讨论引用了可用的并发模式，包括主动对象和监视器对象。

同时，为了说明哪个特定类型的事件处理程序对网络化应用程序有用，反应器的实现指南引用了接受器-连接器模式。接着，该模式引入了一个参与者——服务处理程序——它的具体实现也定义了自己的并发模型。因此服务处理程序的实现指南再次引用主动对象和监视器对象模式。这种在各种模式参与者之间错综复杂的内部关系集合可以用图6-1说明：

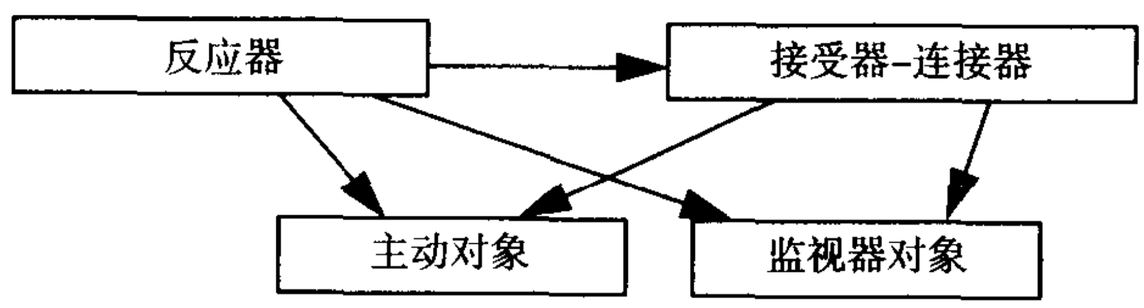


图 6-1

然而，当单独阅读反应器模式描述时，难以弄清楚在有可能用到主动对象和监视器对象的接受器-连接器的情况下如何有效地应用这些模式。例如，反应器模式不指定反应器是否应该应用主动对象来实现特定类型的事件处理程序，它也不指定使用反应器的接受器-连接器是否应该使用监视器对象来实现它的接受器、连接器和服务处理程序。

一般来说，并不是这四种模式的所有可能的组合都有用。然而，因为每个模式描述都是自足的，并不依赖于其他模式，因此很难从单独的模式描述中提取有用的组合。

507

6.1.2 走向模式语言

为了支持特定软件系统和应用程序框架的开发[POSA1]，应以更广的视点看待可用的模式集。特别是，不能把模式看成是孤立的。它们应该被组织成相互关联的模式网络，以便定义一个系统地解决软件开发问题的过程[Cope96][Gab96][Cope97]。我们称之为模式语言。

虽然模式语言提供了用于讨论特定问题的词汇[SFJ96]，但它不是形式语言。模式和模式语言合在一起可以帮助开发者交流体系结构知识，学习新的设计范型或体系结构风格，避免在传统程序设计中只有通过丰富的经验才能避免的陷阱和缺陷[PLoPD1]。

一个或多个模式定义模式语言的“入口点”并解决必须在开发特定类型应用程序或部分应用程序时解决的粒度最粗的问题。在阅读本书时，读者可以把体系结构模式看做在软件体系结构中解决粗粒度问题的模式。每个入口点模式指定应使用其他哪些模式来解决初始问题的子问题，以及应用这些模式的顺序。

因此，被引用的模式使这些“较大的”模式更完整，从而对于特定的问题指定“较小的”模式的组合。在软件体系结构中，这些较小的模式通常对应于设计模式。然后这些较小的模式可以定义如何在解决其他子问题的方案中应用其他模式。将较大的模式连续地迭代分解成较小

的模式，直到给定领域中的任何一个问题都能被某一设计模式解决。

因此，随着模式作者对其领域的熟悉，他们应该努力将能够互补和相互“完善”的模式连接起来[AIS77]。通过一次应用一个模式[Ale79][Bus00b]并遵循模式之间的关系，可以生成高质量的软件体系结构和设计。最后得到的模式语言是“有增效作用的”，也就是说，不仅仅是组成模式的简单相加。例如，连接的模式可以分解软件开发时产生的问题，这有助于产生更好的系统体系结构。每个解决方案建立在相关问题的解决方案之上，这些问题可以使用模式语言中的模式加以解决。 508

为了说明该迭代分解过程，可以集成本章前面介绍的反应器例子中的模式，形成一个袖珍模式语言。接受器-连接器模式可以不作为实现反应器事件处理程序的一个选择，而是要求应用它。因此在反应器体系结构中，有三类事件处理程序——接受器、连接器和服务处理程序——后者可以用并发模型实现，如主动对象和监视器对象（如图6-2）：

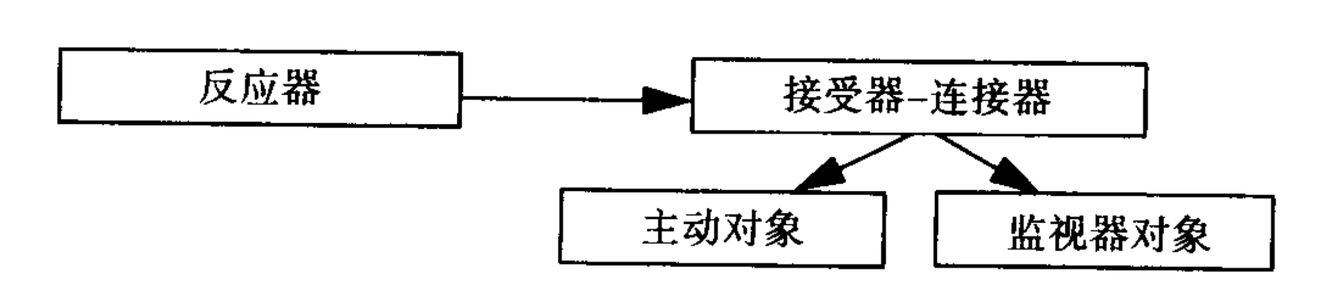


图 6-2

以这种方式重分解这四个模式间的关系对反应器的实现有两个影响：

- 确保了反应器模式的实现能指定事件处理程序的“正确的”类型——接受器、连接器和服务处理程序，它们与接受器-连接器模式有关。将主动对象、监视器对象模式和接受器-连接器联系起来也可以说明对于引入并发，哪类事件处理程序（服务处理程序）是最有用的。
- 可以从反应器模式中删除对并发模式的引用，只需要它出现在接受器-连接器模式中。这简化了四种模式之间的关系并强调了其中重要的模式。

使用反应器、接受器-连接器、主动对象和监视器对象模式建立的袖珍模式语言有助于生成好的软件体系结构，因为可以避免反应器或其他模式单独应用时可能发生的常见实现错误。重分解模式的关系也有助于我们进一步理解四种模式如何连接成可以生成更大的体系结构的范围更广的模式语言。 509

6.2 用于中间件和应用程序的模式语言

本节研究本书介绍的模式之间存在的关系。目标是定义模式语言的基础，这种模式语言比仅仅独立应用模式要能更有效地支持并发和网络化软件系统的开发。

应用两个处理步骤将本书中的模式连接成模式语言：

- *识别模式关系*。首先检查每个模式的自包含描述，确定模式中出现的关系哪些应该保留，哪些应该忽略。特别地，仅仅考虑模式间的“使用”关系而忽略其他所有关系，如“被……使用”和传递关系。关系集还包括其他模式所有的可选使用情况。
- *定义模式排序*。其次，以剩下的关系为基础，定义语言中模式的顺序，也就是哪些模式

是入口点，哪些模式紧随其后，哪些模式在末端。在模式语言中，将具有最广范围的模式——体系结构模式——定义为入口点。接着用“使用”关系定义模式间的排序。

6.2.1 模式语言细节

遵循上述策略，可以将本书中描述的模式连接成模式语言。该语言可使用图6-3进行概括。建议读者在阅读模式语言条目时参阅该图。

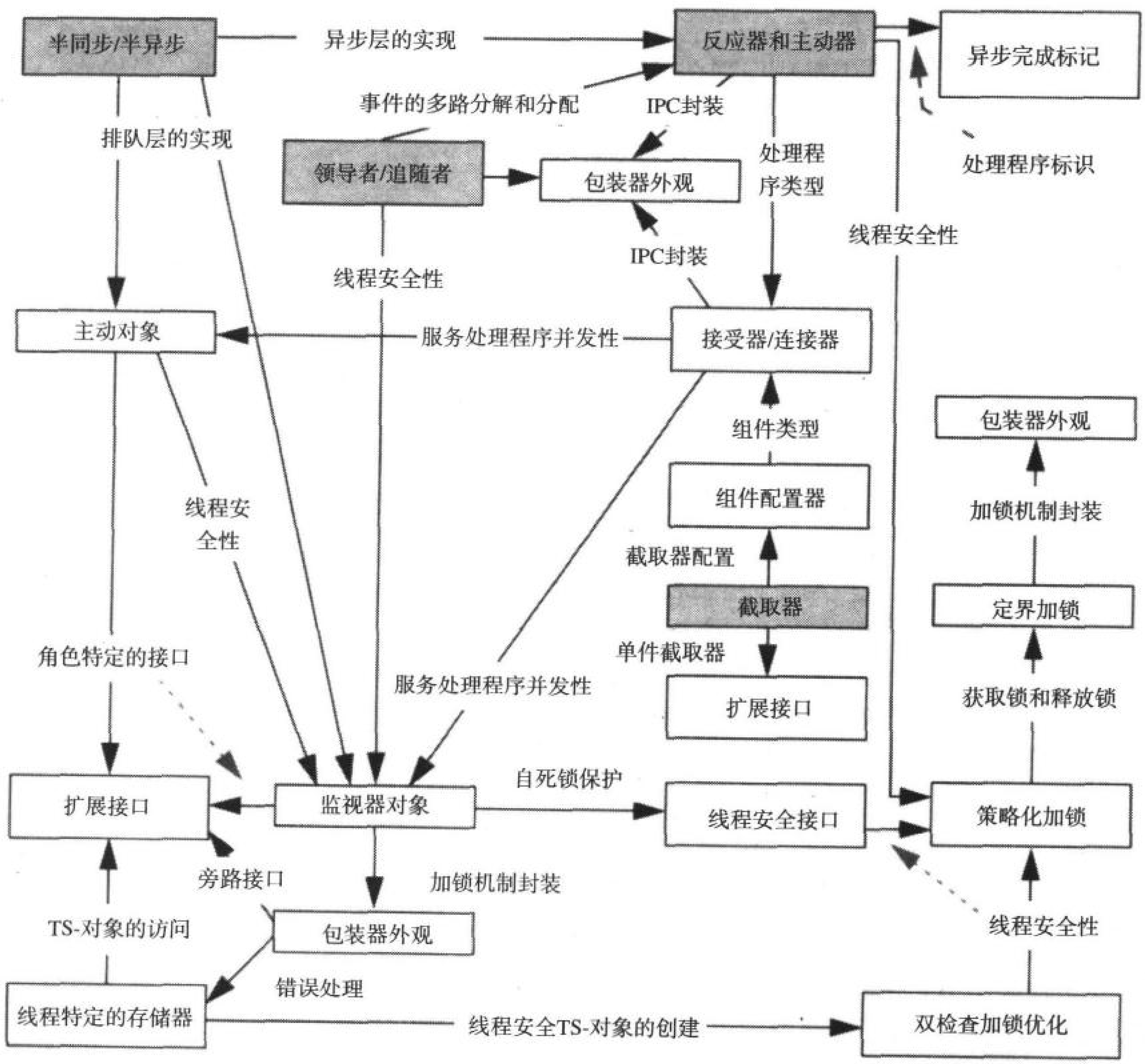


图 6-3

如果一个模式在实现时用到另一个模式，则使用一个箭头指向该模式。对于被其他模式频繁引用的模式，这里复制了多个条目，以避免太多交叉关系。用阴影表示体系结构模式，表明是语言的“开始处”。

510

1. 半同步/半异步

半同步/半异步体系结构模式将可使用异步和同步服务处理混合实现的并发系统结构化。

该模式为异步和同步服务处理引入两个指定的层，以及一个允许服务在其他两层间交换消息和数据的排队层。如果操作系统支持复杂的异步I/O操作，可以使用主动器模式实现异步服务处理层。主动对象模式和监视器对象模式可用于实现排队层。可以通过结合反应器模式和主动对象模式实现半同步/半异步模式的半同步/半反应性变体。 511

2. 领导者/追随者

领导者/追随者体系结构模式提供了一种并发模型，允许多线程依次共享事件源集，以检测、多路分解、分配和处理在事件源上发生的服务请求。

该模式的核心是线程池机制。它允许多线程相互协作，并保护检测、多路分解、分配和处理事件时涉及到的临界区。一次允许一个线程——领导者——等待事件在事件集上发生。同时其他的线程——追随者——可以在队列中等待成为领导者。

当前的领导者线程从事件源集检测到一个事件后，它首先将一个追随者线程提升为新的领导者，接着扮演处理线程的角色，将事件多路分解并分配给指定的事件处理程序。然后事件处理程序依次进行所要求的面向应用的事件处理。在领导者线程等待线程共享的事件源集上的新事件发生时，多个处理线程可以并发运行。在处理了事件后，处理线程又变回追随者角色并等待再次成为领导者线程。

监视器对象模式可以用来实现线程池机制，允许多线程相互协作。反应器或主动器模式可以用来将来自事件源集的事件多路分解和分配给指定的事件处理程序。

3. 反应器

反应器体系结构模式将事件驱动的应用程序结构化，特别是并发地从多客户机接收请求但迭代地处理它们的服务器。

该模式引入两个协作组件，反应器和同步事件多路分解器。它们将到达的请求多路分解并分配给事件处理程序集，事件处理程序集定义了应用程序处理这些请求的服务。通过句柄接收来自客户机的请求，并发送响应。句柄封装了网络化系统中的传输端点。 512

接受器-连接器模式通常指定了三种事件处理程序——接受器、连接器和服务处理程序。通常通过包装器外观设计模式实现封装IPC机制的句柄。可以使用策略化加锁模式实现线程安全反应器。反应器模式的定时器队列机制可以使用异步完成标记模式识别哪个事件处理程序已超时。

4. 主动器

主动器体系结构模式对事件驱动的应用程序结构化，特别是从多个客户机并发接收和处理请求的服务器。

应用程序服务分为两部分：

- 异步执行的操作，如接收客户机请求的操作。
- 处理这些异步操作（例如某个特定客户机请求）结果的相应的完成处理程序。

异步操作由异步操作处理器执行，它将异步操作的结果——完成事件——插入到一个完成事件队列。然后主动器使用异步事件多路分解器将完成事件从完成事件队列中删除，接着将它们

分配到相应的完成处理程序以完成服务处理。通过句柄接收客户机发出的请求，并发送响应，句柄封装了在网络化系统中的传输端点。

为了支持将完成事件从异步操作中有效地多路分解和分配给指定的完成处理程序，异步操作处理器和主动器都可以使用异步完成标记模式来区分哪些异步操作已经完成，以及哪些完成处理程序应该处理它的结果。和反应器模式一样，如同接受器-连接器模式中所指出的，有三种通用类型的完成处理程序——接受器、连接器和服务处理程序。可以用包装器外观模式实现异步操作处理器以及封装IPC机制的句柄。可以用策略化加锁模式实现线程安全主动器。

513

5. 截取器

截取器体系结构模式允许透明地将功能特性添加到应用程序框架中。这些功能特性在框架内部事件发生时自动调用。

截取器模式为框架内部选定的事件类型指定并揭示一个截取器接口回调。应用程序可以从该接口派生出具体的截取器以实现附加功能，附加功能特性以面向应用的方式处理发生的相应事件类型的事件。每个截取器都有一个分配程序，这样应用程序可以向框架注册具体的截取器。一旦指定的事件发生，框架通过关联的截取器回调具体截取器。必须在事件处理过程中修改框架行为的具体截取器可以利用语境对象，这些对象提供对框架内部状态的受控访问。语境对象在被框架分配后被传递给具体截取器。

扩展接口模式有助于避免实现多个具体截取器。它用单个截取器实现多个接口，每个接口对应于一个特定的具体截取器。类似地，组件配置器模式可以用来在运行时动态地将具体截取器链接到具体框架中。

6. 接受器-连接器

接受器-连接器设计模式将网络化系统中连接建立和服务初始化的处理与服务处理分离开。该模式引入三种类型的组件：

- 服务处理程序定义网络化系统中一半的端到端服务，并处理从它们连接的远程对等体发来的请求。
- 接受器进行被动连接建立，接受远程对等体的连接请求，并将服务处理程序初始化以处理随后从这些对等体发出的服务请求。
- 连接器进行主动的连接建立，并作为服务处理程序启动一个到远程组件的连接。一旦连接被建立，服务处理程序就与远程组件进行通信。

514

接受器、连接器和服务处理程序通过句柄发送请求和从对等体接受请求，该句柄在网络化系统中封装传输端点。

为了同时处理多个服务请求，可以使用主动对象和监视器对象模式定义的并发模型实现服务处理程序。可以通过包装器外观模式实现用来访问底层的操作系统IPC机制的句柄。

7. 组件配置器

组件配置器设计模式允许系统在运行时将它的组件实现链接和解除链接，而不用对应用程序进行修改、重编译或静态重链接。它也支持将组件重配置到不同的进程中，而不用关闭或重新启动运行进程。

在该模式中，组件定义了一个统一的接口，用于配置和控制特定类型的应用服务或应用服务提供的功能。具体组件以面向应用的方式实现接口。应用程序或管理员可以使用组件接口动态地初始化、挂起、继续和中止具体组件，也可以获得关于每个配置过的具体组件的运行时信息。将具体组件打包进适当的配置单元，如动态链接库（DLL）或共享库，它们可以在组件配置器的控制下动态链接到应用程序中或者从应用程序中解除链接。这里使用了组件仓库来跟踪所有配置到应用程序中的具体组件。

用组建配置器配置到网络化系统中的组件可以是接受器-连接器模式定义的接受器、连接器和服务处理程序，或者是截取器模式中定义的截取器。 515

8. 主动对象

主动对象设计模式将方法执行和方法调用分离，以增强并发性，简化对驻留在自身控制线程中的对象的同步访问。

代理表示主动对象的接口，服务者提供对象的实现。代理和服务者都在单独的线程中运行，因此方法调用和方法执行可以并发运行。在运行时，代理将客户机的方法调用转换为方法请求，并被调度程序存储在激活表中。调度程序的事件循环和服务者在同一个线程中连续运行，不断地将方法请求从激活表队列中取出并将它们分配给服务者。调用方法时，客户机可以通过一个由代理返回的前景(future)获得方法执行的结果。

扩展接口模式有助于提供与角色有关的代理，使客户机仅访问它们需要的主动对象服务。监视器对象模式可以用来实现线程安全的激活表。

9. 监视器对象

监视器对象设计模式对并发方法的执行同步化，以确保一次只有一个方法在对象中运行。它也允许对象的方法相互协调，调度方法的执行顺序。

客户只能通过监视器对象的同步化方法访问监视器对象定义的功能。为了防止涉及监视器对象状态的竞争条件，一次只能有一个同步化的方法运行在监视器对象中。每个被监视的对象包含一个监视器锁，同步化的方法用它串行化对对象行为和状态的访问。另外，同步化的方法可以基于一个或多个与监视器对象有关的监视器条件，确定它们挂起和继续执行时所处的环境。

扩展接口模式有助于导出对监视器对象的与角色有关的视图，以使客户机仅仅访问它们需要的服务。线程安全接口模式有助于防止在监视器对象的同步化的方法调用同一监视器对象上的另一个同步化的方法时发生自死锁。 516

10. 包装器外观

包装器外观设计模式将现存的非面向对象API提供的函数和数据封装在更简明、更健壮、可移植、可维护和内聚的面向对象的类接口中。

扩展接口模式可以用来让客户机访问包装器外观的与实现有关的一些特征。在不能有效或可移植地支持异常处理的平台上实现包装器外观的出错处理机制时，可以使用线程特定的存储器模式。

11. 扩展接口

扩展接口设计模式防止开发者扩展或修改组件功能特性时发生接口膨胀和客户机代码破坏的现象。多扩展接口可以附属于同一个组件，每个接口在组件和它的客户机间定义一个契约。

使用扩展接口模式时，组件的功能特性仅通过扩展接口输出，它实现的每个角色有一个扩展接口。因此客户可以访问接口但是不能直接访问组件实现。一个关联的工厂负责建立组件实例并向客户返回一个对客户机初始接口的引用。客户可以使用该接口获得其他扩展接口。

组件内部扩展接口功能特性的实现可以使用主动对象和监视器对象模式，使它们运行在自己的控制线程中。

12. 异步完成标记

异步完成标记设计模式允许客户机异步调用服务上的操作，在操作完成时高效地分配其后继的处理行为并返回它们的结果。

对于客户机调用服务的每个异步操作来说，客户机建立一个异步完成标记，识别为处理操作完成而必需的动作和状态。客户机将异步完成标记与操作一起传递给服务。当服务回应客户机时，它的响应必须包括最初发送的异步完成标记。然后客户机用异步完成标记识别处理异步操作结果的完成处理程序。

517

13. 线程特定的存储器

线程特定的存储器设计模式允许多个线程使用一个“逻辑上全局”的访问点获得线程内部的一个对象——称为“线程特定对象”——而不会在每次访问对象时产生加锁开销。

使用一个线程特定对象集来维护某一线程的线程特定对象。对某一线程特定对象的全局访问点可以由线程特定对象代理实现。代理隐藏线程特定对象的创建细节，在访问线程特定对象的方法时从线程特定对象集获得它。

通过代理访问线程特定对象方法的另一个选择是使用扩展接口模式，在扩展接口模式中代理返回一个线程特定对象实现的接口。双检查加锁优化模式通常用来正确地、透明地在多线程应用程序中建立线程特定对象。

14. 线程安全接口

线程安全接口设计模式将加锁开销最小化，并确保组件内部的方法调用不会因为试图获得一个组件已经获得的锁而导致“自死锁”。通过使用该模式，组件的方法被划分为两类，实现方法和接口方法:

- 实现方法在组件内部并且不能被它的客户机调用，它实现组件的功能特性，如果需要可以调用其他实现方法。实现方法“确信”它们是被正确地调用的，因此不需要获得/释放锁。
- 相反，接口方法将组件的功能特性展示给客户。这些方法首先通过获得一个锁进行“检查”，将方法的执行指派给相应的实现方法，最后在实现方法完成执行后将锁释放。接口方法从不在同一组件上调用其他接口方法。

518

策略化加锁模式可以用来实现锁的获得和释放，也可以将被使用的锁类型参数化。

15. 双检查加锁优化

在程序执行期间，代码的临界区必须以线程安全方式只获得一次锁时，双检查加锁优化设

计模式可以减少竞争和同步开销。

该模式使用一个标志表示是否需要在获得保护一个临界区的锁之前执行它。如果该关键代码部分已经初始化，就不需要再次执行，从而避免了不必要的加锁开销。

策略化加锁模式用于实现锁的获得和释放，也用于将使用的锁的类型参数化。

16. 策略化加锁

策略化加锁设计模式将组件中的同步机制参数化，该同步机制防止它的临界区受到并发访问。它允许组件的同步机制作为“可插的”类型来实现。每个类型将特定的同步策略具体化，如互斥、阅读器/记录器或信号灯。这些可插类型的实例可以定义为包含在组件中的对象，该组件可以用这些对象有效地同步它的方法实现。

定界加锁惯用法可以用来获得和释放某一类型的锁，通过策略化加锁模式将锁参数化为组件。并且，策略化加锁可以对应用定界加锁惯用法的哨兵类模板化，以便可以透明地对同步机制参数化。

17. 定界加锁

定界加锁C++惯用法确保控制进入某个范围时获得锁，当控制离开这个范围时自动释放锁，而不论该范围的返回路径是什么。

该模式定义了一个哨兵类，该类的构造函数在控制进入某个范围时自动获得一个锁，当控制离开这个范围时该类的析构函数将锁自动释放。在定义临界区的方法或块区域中，建立哨兵类的实例以获得和释放锁。 519

6.2.2 对模式语言的讨论

以上对模式的浓缩描述和模式关系图揭示了大多数模式是如何以多种方式相互补充和相互完善，并形成模式语言的：

- 虽然每个模式单独看来都是有用的，但是模式语言更强大，因为它集成了对重要问题领域中特定问题的解决方案，如事件处理、连接管理和服务访问、并发模型以及同步策略。在开发并发和网络化系统时，这些领域中每个问题必须要统一协调地加以解决。
- 模式语言也揭示了这些一般问题领域的独立性。例如，为一个网络化应用程序选择一个特定事件处理模式时，不是所有潜在可用的并发模式都能有效地应用。

这两点只有在将模式连接成为模式语言时才变得清晰，因为每个单独模式仅仅关注其自身。这使得很难认清模式相互之间的关系并有效地解决更复杂的系统体系结构问题。相反，基于模式语言的设计可以一致地、协作地将模式集成，从而成功地实现对软件系统的需求。

模式语言已经应用于现实世界的许多应用程序中，包括——但不仅限于——那些使用ACE框架建立的系统[Sch97]。因此该语言是一种指定和实现中间件和应用程序的重要工具。但要注意，模式语言是不完善的，它为开发分布式对象计算中间件以及并发和网络化应用程序的更大规模语言提供了基础。

幸运的是，可以通过应用在模式文献中定义的其他模式完善模式语言。例如，截取器模式与本书中其他大多数模式是正交的。然而，代理者体系结构模式[POSA1]为分布式软件系统定义 520

了一个基本结构，这些分布式系统通常使用截取器模式以支持额外扩展[NMM99][HS99a]，使用半同步/半异步体系结构模式或它的半同步/半反应性变体[Sch98b]以对它的代理者组件结构化。因此代理者模式将截取器模式和模式语言中的其他模式连接起来，它们的内部关系就很清楚了。

文献中的其他模式有助于精炼模式语言中的模式。例如，由接受器-连接器模式定义的对等体服务处理程序可以使用半对象加协议模式[Mes95]和/或抽象会话模式[Pry99]实现。类似地，代理模式的远程代理变体[GoF95][POSA1]可以用于实现一个特定截取器变体。还有其他例子：转发器-接收器模式[POSA1]有助于实现半同步/半异步模式的半同步/半反应变体。类似地，代理者[POSA1]和对象同步器[SPM99]模式可以用于实现主动对象的变体。

模式语言也可以结合最初由并发和网络化之外的领域中的例子产生的模式。例如，本书引用的许多熟知的通用模式，包括[GoF95]中的抽象工厂、适配器、桥、命令、装饰器、外观、工厂方法、迭代器、中介者、备忘录、观察者、单件、策略和模板方法，[POSA1]中的命令处理器、层、管道和过滤器，以及映像，[PLoPD3]中的管理者和空对象，以及[Pree95]中的钩子方法。

所有这些互连模式的集成为开发分布式对象计算中间件以及并发和网络化应用程序形成了一个更广范围的模式语言。毫无疑问，可以用其他已发布的模式或还未发现和未记入文档的模式扩展该语言。每次扩展都使模式语言变得更强大、完善和富有表现力。以这种方式，可以改善并发和网络化软件系统的模式的集成。

521

6.3 并发和网络化之外

前一小节说明了本书描述的模式以及其他来源的模式如何定义模式语言的基础，以开发分布式对象计算中间件以及并发和网络化应用程序。尽管模式语言强调在本特定领域使用这些模式，但许多模式也可应用于这些领域外。

对模式的“问题”部分和“已知使用”部分进行分析，可以发现它们的应用范围比本书关注的更广泛。某些模式，例如包装器外观，一般可应用于需要将已有的独立函数和数据用面向对象类接口进行封装的任何领域。其他模式用于解决许多系统中出现的特定类型的问题。例如扩展接口，解决如何设计对由多角色组件提供的功能特性的可扩展访问。

因此，本节指出本书中的模式可以应用的除模式并发和网络化之外的领域。

6.3.1 图形用户界面

我们描述的有些模式已经被用于设计和实现各种图形用户接口：

- 包装器外观模式通常用于将特定GUI库的细节进行封装以对应用程序开发者隐藏它的实现细节。该模式在GUI库语境中的已知两个突出的使用是微软基本类库（MFC）[Pro99]和Java Swing library[RBV99]。
- 反应器模式的变体已用于组织具有图形用户界面的系统中的事件处理。例如，反应器模式由内部视图分配器（Interviews Dispatcher）框架实现，内部视图分配器框架用来定义应用程序的主事件循环以及管理到一个或多个物理GUI显示的连接[LC87]。X Windows发行

版本中的Xttoolkit也使用了反应器模式。 522

6.3.2 组件

有几个模式在组件和基于组件开发的语境中应用：

- 包装器外观模式指定如何实现内聚的低层组件集合以及将它们应用于各种语境，如线程调度和进程间通信组件[Sch97]。
- 组件配置器模式支持对组件实现的动态配置和重配置。除了作为动态安装操作系统设备驱动程序的基础[Rago93]之外，该模式也支持动态下载和配置Java小应用程序[JS97b]。
- 截取器模式引入了一个建立可扩展组件和应用程序的机制。它的“已知使用”部分列出了应用该模式的当前组件模型，如微软的组件对象模型 (COM)[Box97]、Enterprise JavaBeans(EJB)[MaHa99]和CORBA组件模型(CCM)[OMG99a]。
- 扩展接口模式定义了一个通用机制，用于设计组件，并允许客户访问它们的服务。当前所有的组件标准，如COM、EJB和CCM都实现了该模式的变体。

6.3.3 常规编程

总的来说，本书中的某些模式或惯用法可以用于编程：

- 定界加锁是一种通用C++编程技术在安全资源获取和释放上的特例，在[Str97]中，由Bjarne Stroustrup 在一个更通用的语境中对该技术描述，称做“对象的建立是资源的获取”。
- 双检查加锁优化可以用于保护应该只执行一次的代码，特别是初始化代码。 523

总而言之，以上讨论的七种不同的模式和惯用法——包装器外观、反应器、组件配置器、截取器、扩展接口、定界加锁和双检查加锁优化——显然可应用于并发和连网领域之外。如果对具有良好设计的软件系统进行分析，可能会发现别的应用了这些模式或其他POSA2模式的领域。虽然本书主要介绍开发并发和网络化系统语境下的模式，但是应该认识到这些模式可能有助于解决其他领域中的问题。

6.4 模式语言与模式系统

前一节研究了本书引入模式的模式语言特征。除了用于定义建立分布式对象计算中间件以及并发和网络化应用程序的模式语言基础之外，也可以将本书中的模式组织成模式系统[POSA1]。例如，可以使用本书中的模式覆盖的问题领域：服务访问和配置、事件处理、同步和并发对[POSA1]中定义的模式系统进行扩展。然后可以相应地对模式进行重新分类。

这种分类图式提供了一种划分模式空间的有趣概念运用。可将每个模式归类并分派到多维矩阵的一个单元，每个矩阵的维数代表特定的模式属性。如果必须对孤立问题进行解决，使用这种分类法可以快速访问潜在的基于模式的可行解决方案。下表显示了一种将本书中的模式以及从[GoF95][POSA1][PLoPD1][PLoPD2][PLoPD3][PLoPD4]中选择出来的模式组织成为一个适

524 用于并发和连网模式系统的方法。[⊖]

	体系结构模式	设计模式	惯用法
基线体系结构	代理者(Broker)		
	层(Layers)		
	微内核(Microkernel)		
通信	管道和过滤器	抽象会话(Abstract Session)[Pry99]	
	(Pipes and Filters)	命令处理器(Command Processor)	
		转发器-接收器(Forwarder-Receiver)	
		观察者(Observer)[GoF95]	
		远程操作(Remote Operation)[KTB98]	
		串行化器(Serializer)[RSB+97]	
初始化		激活器（Activator）[Stal00]	
		客户机-分配器-服务器(Client-Dispatcher-Server)	
		驱逐器（Evictor）[HV99]	
		定位器(Locator)[JK00]	
		对象生命期管理器(Object Lifetime Manager)[LGS99]	
服务访问和配置	**截取器**(Interceptor)	**组件配置器**(Component Configurator)	
		扩展接口(Extension Interface)	
		半对象加协议(Half Object Plus Protocol)[Mes95]	
		管理者-Agent(Manager-Agent)[KTB98]	
		代理(Proxy)	
		包装器外观(Wrapper Facade)	
事件处理	**主动器**(Proactor)	**接受器-连接器**(Acceptor-Connector)	
	反应器(Reactor)	**异步完成标记**(Asynchronous Completion Token)	
		事件通知（Event Notification）[Rie96]	
		观察者(Observer)[GoF95]	
		出版者-订阅者(Publisher-Subscriber)	
同步	对象同步器(Object	阻拦(Balking)[Lea99a]	定界加锁
	Synchronizer)[SPM99]	代码加锁Code Locking[Mck95]	(Scoped Locking)
		数据加锁(Data Locking)[Mck95]	
		保护挂起(Guarded Suspension)[Lea99a]	
		双检查加锁优化(Double-Checked Locking Optimization)	
		阅读器/记录器加锁(Reader/Writer Locking)[McK95]	
		特定通知(Specific Notification)[Lea99a]	
		策略化加锁(Strategized Locking)	
		线程安全接口(Thread-Safe Interface)	
并发	**半同步/半异步**	**主动对象**(Active Object)	
	(Half-Sync/Half-Async)	主控-从属(Master-Slave)	
	领导者/追随者	**监视器对象**(Monitor Object)	
	(Leader/Follower)	生产者-消费者(Producer-Consumer)[Grand98]	
		调度程序(Scheduler)[Lea99a]	
		两阶段终止(Two-phase Termination)[Grand98]	
		线程特定的存储器(Thread-Specfic Storage)	

⊖ 本书中描述的模式都**加下划线**表示。

然而，根据具体特定领域或属性对模式进行分类，某种程度上不能体现某个特定模式集合中存在的关系和相互依赖性。这些关系和相互依赖性影响了模式在和其他模式一起使用的情况下的适用性，因为不是每个模式都可以与任何其他模式进行有意义的组合。 525

例如，线程特定的存储器不适合与领导者/追随者线程池设计一起使用，因为线程池中的线程和超限时间进行处理的事件之间可能没有固定的关联。因此一般来说，在建立真实世界的系统时识别“正确的”模式组合是很重要的，因为这些系统展示了许多问题间的必须解决的依赖性。

大的模式系统也趋于复杂，因为模式系统包括的问题领域越多，越有可能为模式指定更多类别。包装器外观模式是这种情况的一个很好的例子，因为如第6.3节所述，它可以用于建立并发和网络化系统、图形用户界面接口和组件。因此它在覆盖所有这些系统的模式系统中至少会出现三次。随着模式数量的增加，这些重复的模式入口可能使模式系统膨胀起来，使模式难以学习和使用。

解决这一问题的一个方法是为特定领域指定一定数量的更小的模式系统，而不是指定一个通用的模式系统。例如前面描述的适用于并发和网络化模式系统，或用于组件构造的模式系统。然而，如果将该方法应用于本书中出现的模式，读者会注意到模式系统在结构上类似于模式语言，第6.2节对模式语言的基础进行了描述。

并且，最后得到的模式系统可能不强调模式以及这里描述的模式语言之间的关系。模式可能保持孤立，而对用于并发和网络化中间件和应用程序的完整软件系统的基于模式的开发而言，模式系统可能没有很大用处。

根据我们的经验，将本书中出现的模式组织成模式语言比将它们归类成模式系统更有效。然而，还需要进行更多的研究，以对完成模式语言必需的所有模式进行识别、形成文档和集成。 526

第7章
模式的过去、现在和未来

"预言很难，尤其是预测未来。"
Yogi Berra, 棒球名人堂中的哲学家

科学贤哲詹姆斯·伯克喜欢指出历史很少按适当的顺序或在适当的时间发生，但是历史学家的任务是使它（历史）看上去确实如此。《面向模式的软件体系结构》（POSA）丛书使我们有机会重新回顾过去的预言，总结实际上出现了什么，并且分析按照那样的方式发生的原因。

本章回顾了我们1996年对"模式将向何处去"的预言，该预言出现在POSA丛书的第1卷《模式系统》[POSA1]。我们对在过去4年中模式的实际发展方向进行了探讨，分析了模式的现状，并且事后聪明地回顾了我们关于其未来的观点。 527

7.1 在过去的4年中发生了什么

《模式系统》[POSA1]预测了模式演化的方式。4年后，在本书中相应章节我们重新总结了实际发生的情况。在这一部分引用了很多相关工作，但罗列并不完整。更多的引用可以在http://hillside.net/patterns 中找到。

7.1.1 模式

在1996年，我们预测在以后几年中许多软件开发领域的许多模式会被（重新）发现并文档化，期望这些模式进一步丰富模式文献。部分预言成为了现实，但仍有多个领域文献不多，特别是在分布式、容错、事务性、实时以及嵌入式系统、中间件和应用程序等领域。

模式文献的主体内容在过去4年中有了明显的增长。例如，美国PLoP和欧洲EuroPLoP模式会议非常兴旺并新增几个会议，如美国的ChiliPLoP和澳大利亚的KoalaPLoP。这些会议基于一种"作者专题讨论会"的形式[POSA1][CW99]。在作者专题讨论会中，作者相互传阅他们的模式和模式语言，并讨论它们的利弊以改进其内容和风格。

过去的4年中，很多学科出现了关于模式的书籍，如：

- 许多软件开发领域的模式汇编 [PLoPD3] [PLoPD4] [Ris98] [Ris00a] [Ris00b] 。
- Martin Fowler的用于健康卫生和公司金融领域的模式分析的启蒙书籍。
- 本书集中于并发和网络化中间件框架以及应用软件的体系结构和设计模式。 528
- 软件开发过程模式[Amb99a][Amb99b]和配置管理模式[BMT99]。
- 用于特定的编程语言（如Smalltalk[Beck97]和Java[Grand98][Grand99]）的模式，或者与特

定中间件（如CORBA[MM97]）相关的模式。

- 所谓“反模式”[BMBMM98][BMT99]提供软件开发问题的通用的“解决方案”，这些解决方案看起来是合理的，但是由于各种原因并不可行。

对于我们来说，指出这些书籍内容和质量的差异很重要。然而，批评和评论这些出版物则超出了本章的范围——我们把它留给读者自己去判断和进行在线书评（如amazon.com上的书评）。无论如何，过去四年中的出版量说明了软件研究和开发团体对模式不断增长的兴趣。

许多广泛应用的模式源于或用于文档框架[GoF95][POSA1][Sch96][RKSOP00]。可以将这些模式看做是帮助开发者重用软件体系结构的框架抽象描述[JML92]。框架可以从另一个角度看做允许直接设计和代码重用模式的具体化[HJE95][HS98]。经验表明成熟的框架中有很高的模式密度[RBGM00]。

7.1.2 模式系统和模式语言

随着软件团体对模式关注的增长，以模式语言为中心的工作也正在增长[Cope96][Gab96][Cope97]。早在1995年，模式团体的开创者就开始为特殊软件开发领域（特别是电信领域[DeBr95][Mes96][ACGH+96][HS99b]）中的模式集[PLoPD1]进行文档化。这些模式比早先出版的独立模式相互间有更紧密的联系。实际上，某些模式相互联系紧密，以至于它们不会单独存在。模式被组织进模式语言中，在模式语言中，每个模式建立在别的模式之上或组合成其他模式。

529

许多已出版的模式语言出现在[Beck97][PloPD2][PLoPD3][PLoPD4]中。正如在第6.2节“*适用于中间件和应用程序的模式语言*”中说明的，本书中的模式也形成了一种模式语言，或者更准确地说是一种模式语言的核心，它适用于开发分布式对象计算中间件、应用程序和服务。我们曾做出的在模式语言方面将会得到扩展的预言也成为了现实。然而，在文献中仍然存在明显的欠缺，部分原因是还需要进行大量的工作编写综合性的模式语言。

有些工作将模式分类并将模式组织成类目[Tichy98]、模式系统[POSA1]和[Ris00a]的综合模式年鉴（Pattern Almanac）。但是，在过去的4年中，模式语言相比模式系统和模式目录逐渐变得更受欢迎。模式团体发现，模式语言是记录紧密关联的模式集的最有前途的方式。第6.4节“*模式语言与模式系统*”为模式目录和系统逐渐向模式语言的过渡提供了更多的理由和动力。

7.1.3 方法和工具

尽管受到了某些有经验的开发者的怀疑[POSA1]，1996年我们仍然预测，为了更有效地使用模式所采用的方法和工具而进行的工作会得到扩展。软件研究团体在该主题上一直矢志不渝地工作着，试图找出了一些在现有系统中识别模式的方法[SMB96]，以及将模式应用于正向软件工程[BFVY96][FMW97][SRZ99]和逆向软件工程的工具[Ban98][PK98][Mic00][PC00] [PSJ00]。

虽然最近取得了一定的进展，但对用于模式的自动软件工具的研究还没有对软件工具团体产生很大的影响。这些进展包括：用于模式文档化的符号已经集成到UML中了[BRJ98]。类似地，出现了从通过元模型描述的模式中自动生成代码的工具[AONIX99][MTOOL99]。

530

7.1.4 算法与数据结构

我们预测，人们对将模式应用于算法、数据结构和模式之间关系的分析的兴趣会不断增长。我们期望这些工作取得成果之后，还要使用某种具体的模式形式，将算法和数据结构作为模式而文档化。

至今为止，有几篇论文[Nuy96]和教材[FF97][Preiss98]将 [GoF95]中的模式加入本科生的算法与数据结构课程中。然而，一般来说，这些工作的重点是本科生教育，并没有对模式研究团体的文献主体内容做出直接贡献。

7.1.5 形式化模式

在过去四年中，有很多将模式形式化的研究工作[LBG97][MK97][LS98][Mik98][BGJ99][EGHY99]。然而，这方面的工作还没有直接影响到模式团体或软件从业者。

缺乏影响的一个原因是模式是一种解决相关问题集的方案(schema)，因此它可以被重复实现而不必每次都完全相同。相反，形式化方法的目标是尽可能详细地获取某一个概念。因此，为了把握模式的内在变化情况，用于描述模式的形式化方法通常都很大而且复杂，使得对它们难以理解和使用。可是，这种缺陷使模式的很多优势都发挥不出来，模式的这些优势在增强（而不是减弱）软件开发团队成员之间的交流的时候是最有效的。 531

7.2 模式现状

现在，模式团体和4年前相比取得了很大的进步。特别是文档化的具体模式和模式语言的主体内容比4年前更大、更多样化——并且一直在增长。

然而，团体中不同的小组喜欢不同的模式范型，这些模式范型包括：

- 受Christopher Alexander对模式 [Ale79] 和模式语言 [AIS77] [Cope96] [Cope97]的观点的启发而提出的自顶向下面向理论的方法。
- 由四人帮[Gang-of-Four, GoF95]和POSA工作组[POSA1]提出的自底向上面向工程的方法。
- 由“专用”模式（反模式[BMBMM98]）的作者提出的方法。

除了在理论上对模式到底是什么有了更好的理解之外，现在模式团体对模式在实践中如何工作也有了更好的理解[Vlis98a][Bus00b]。到1996年为止，模式只是偶尔被独立研究者和开发者应用[GR96][HJE95][John92][Cope95][McK95][SchSt95][Maf96][Sch96] [KCC+96]。在其间的几年中，在软件工业的广泛主题和领域中，更系统地将模式应用于生产系统，这些领域包括：

- 应用程序框架[FS97][BGHS98][FJS99a][FJS99b][FJ99]。
- 实时CORBA中间件[SC99][PRS00]、高性能Web服务器[Sch97]、网络管理[KTB98]、并发控制和同步[SPM99]。
- 输入和输出处理[Har97][HS99b][Tow99]。
- 性能度量[NLN+99][GP97]和优化[PRS+99]。
- 安全[YB99]和可靠性[Maf96][SPM98][ACGH+96]。 532

- 评估应用模式利弊的经验研究[PUPT97]。
- Java编程语言[AG98][GJS96]、Java标准库[Sun00a][FHA99]和第三方框架[IBM98]、Java虚拟机规范[LY99]和实现[Sun00b]，以及它的开发团体[GG99]。

模式的影响不断增长的另外一个证据是，现在的开发者不仅在设计中隐式地使用它们——文献中还没有引入模式概念很久之前，他们就已经这么做了——而且他们还显式地将模式应用在日常工作中。模式提供了通用的词汇，研究者和开发者现在可以用来简练地交换在设计评审、问题分解、针对系统相关的特性的训练和软件体系结构文档化等方面的想法。

实际上，防止开发者过分热衷于模式，盲目地将模式应用于所有可能的地方，是很重要的。虽然模式可以改善设计复杂性和可（重）用性，但是它们也可能带来附加的费用。例如，实现某些模式可能需要更多的初始化编程或设计工作，或者可能需要维护更多的代码。某些模式实现涉及到额外的间接性，这种间接性可能会在某些平台上产生更多的性能开销。因此开发者和管理者必须在使用模式之前仔细地评估它们的好处和坏处，特别是在一个简单的类或函数就可以满足需求的时候！

7.3 模式向何处去

我们希望，在未来，随着研究者和主流软件项目不断地应用面向模式的范型、方法和过程，人们对模式的兴趣有本质性的增加。本节描述了我们对前面回顾的每个主要专题领域的观点，并且概述了对未来研究中主要问题的预测。我们也加入了几个新的类目——软件开发过程和教育——以反映在[POSA1]中未涉及到的模式团体中的有前途的研究方向。

533

7.3.1 模式

在过去的4年中，有重要影响的模式文献都集中讨论具体模式和模式语言，它们通常来源于面向对象的应用框架[John97]。展望未来，我们预期，模式团体的未来工作将超出它的传统领域。例如，下一代面向对象的应用程序和框架将明确地包含模式。模式也将继续用于描述框架的形式和内容[SFJ96]。其他将受益于具体模式挖掘的关键主题和领域包括：

- *分布式对象*。在过去10年中，许多用于并发和网络化对象与中间件和应用程序相关的模式（包括本书中提到过的）已经被文档化[SC99][Lea99a]。下一步的关键是将用于分布式对象的模式文档化，扩展前期的工作，重点关注分布式主题，如远程服务定位[JK00]和划分、命名和目录服务、负载平衡、可靠性和安全性。

  例如，越来越多的分布式对象计算系统必须为客户程序和最终用户提供高级别的可靠性。通过应用CORBA容错规范[OMG99g]和实施该规范的ORB，和过去相比，开发者将有更多的机会从其经验中获得用于容错分布式对象计算的模式[IM96][ACGH+96][NMM00]。
- *实时和嵌入式系统*。越来越多的计算系统是嵌入式的，包括自动控制系统和车载应用程序、工厂自动设备控制软件[BGHS98]、航空任务计算[HLS97]和手持计算设备。许多这些系统有很严格的计算资源限制，特别是内存优化和时间约束。

开发高质量的实时和嵌入式系统很困难并且还有一些“怪招”（black art）。因此在该领域中发展的模式相当少[NW98][Bus98][lan98][Lea94]。我们希望用于实时和嵌入式系统的模式主体在将来能得到发展。其中动力之一就是对象技术的成熟，以及使之（包括模式）成功的开发工具、方法和技术，它将越来越多地应用于这些领域。 534

- *移动系统*。无线网络正在普及，并且嵌入式计算设备也越来越小、越来越轻且易于使用。因此，移动系统将很快能支持许多消费者的通信和计算需求。移动系统的应用领域包括无所不在的计算、移动agent、个人助理、位置相关的信息保障、远程医疗诊断和放射治疗以及家庭和办公室自动化。另外，可以通过移动系统访问从Web浏览器漫游到在线银行的Internet服务。

  移动系统提出了许多挑战，如对低的而且多变的带宽和能耗的管理，对频繁的连接中断和降低的服务质量适应，协议分歧以及保持断开的网络节点之间缓存的一致性等等。我们希望有经验的移动系统开发者以模式的形式将他们的经验文档化，这样有助于满足该领域不断增长的软件开发实践需求。

- *商业事务和电子商务系统*。许多商业信息系统，如会计、工资管理、存货清单和账务系统，都是基于事务的。处理事务的规则十分复杂，而且必须很灵活以反映新的商业活动和企业兼并。业务系统还必须同时处理不断增加的大宗在线事务，实例可参见领导者/追随者模式的“例子”一节。

  Web上电子商务的出现将许多企业到企业（B2B）系统直接暴露在消费者面前。虽然这些系统很重要，但是关于对它们的分析、体系结构或设计模式的文章却相当少。我们希望关于事务和电子商务的模式主体内容在[John96][Fow97a][KC97]的前期工作基础上能有所增长。

- *基于商用现成产品（COTS）分布式系统的服务质量*。分布式系统，如视频流、网络电话和大规模交互仿真系统[BHG00][FJS99b]，对服务质量(QoS)的要求越来越迫切。关键QoS需求包括网络带宽和延时、CPU速度、存储访问时间和能耗水平。为了缩短开发周期和降低费用，越来越多地使用多层COTS硬件、操作系统和中间件组件开发这种分布式系统。 535

  然而，从过去的情况来看，很难配置可以同时满足多个QoS属性（如安全性、及时性和容错性）的基于COTS的系统。由于开发者和系统集成者一直控制提供端到端QoS的质量保证的复杂性，所以他们将有成效的模式记录归档是很重要的，这样有助于其他人配置、监控和处理控制具有相互依赖的QoS属性[ZBS97]的基于COTS的分布式系统。

- *反身(reflective)中间件*。该术语描述了一系列用于在自主或半自主分布式应用程序和系统中管理和控制系统资源的技术。反身中间件技术可以在有或者没有关于应用程序最终用户知识的情况下，通过适应核心软件和硬件协议、策略和机制而使应用程序行为得以动态改变[KRL+00]。和在分布式系统QoS中一样，在将最佳实践成文归档当中，模式起非常关键的作用，这有助于确保有效地使用基于反身中间件的应用程序。

- *优化原则模式*。许多现存模式文献的焦点都集中在软件质量因素上而不是性能上。虽然这在非功能性需求（如可用性和可扩展性）的领域中是可以接受的，但在其他领域（特别是分布式和嵌入式实时系统）更看重有效性、可扩展性、可预测性和可靠性，而不是其他的

软件质量。

因此，我们希望能在这些领域看到人们对优化原则模式的更多关注，并将用于优化复杂软件系统的规则文档化[NLN+99][PRS+99]。然而在模式团体中，对于优化原则模式是真正的模式还是仅仅只是原则，还存在争论[Bus00b]。

536

除了上述模式挖掘活动，我们希望看到对与特定应用领域相关的具体模式的编辑整理的出版物有所增加。例如，Linda Rising 编辑了一本关于*IEEE Communication Magazine*（IEEE 通信杂志）[IEEE99]的专辑，并且编写了一本关于电信领域模式和模式语言的资料汇编[Ris00b]。

7.3.2 模式语言

除了将独立的模式成文归档之外，我们希望看到，能持续地将模式编撰进日益成熟的模式语言中。我们还希望看到，现有的独立模式[GoF95][POSA1]和复合模式[Vlis98c]能被集成到现有或新的模式语言中，正如在第6.2节“用于中间件和应用程序的模式语言”中讨论的那样。

反过来，当在实现更大的体系结构模式（如反应器和主动器）时，取得了更多经验之后，我们希望将出现更小的模式，如“事件多路分解器”或“持续传递事件驱动系统”。将大的体系结构模式“分解”为小的模式，这样这些小模式可以有助于改进大的体系结构模式的具体实现。该分解过程是为一类系统定义模式语言的另一种方法，这类系统的结构由正被分解的体系结构模式定义。

朝模式语言方向发展的趋势来源于两种因素的影响：

- 随着对更多的模式的成文归档，特定领域中涉及的模式将趋于完整。
- 随着某一领域中的模式变得完整，该领域中的其他模式可以建立于已有模式的基础上或揭示与已有模式的关系[Ris00a]。

在一个领域中，新的模式成文归档后，它们通常解决与现存模式问题有关的问题和子问题，因此才会有这种增效作用。当这些关系变得明显时，领域中的许多模式可以连接起来形成模式语言。

537

从模式到模式语言的转变需要付出很多努力，也很重要[Vlis98b]。模式语言发展的一个关键障碍是需要将它们文档化。模式语言易于阅读和应用，但是很难编写，因为作者必须：

- 在某一特定领域成为技术专题的专家，并且
- 花费很多时间将他们的经验文档化，整理成可理解的文档，该文档应能区分语言中所有的模式，并将每个独立的模式集成到整体模式语言中。

幸运的是，独立模式和复合模式[Vlis98c]正越来越多地被模式团体文档化，其中大多数是基于对普通框架的抽象[Rie97]。所以，有更多的材料可以组织成模式语言，而不必从头开始编写所有内容。

从各个会场（如PLoP系列会议）举办的作者专题研讨会中，正在产生新一代的作家。随着时间的推移，我们希望这些作者能够填补现有模式语言文献中的空白。

7.3.3 经验报告、方法和工具

正如第7.2节“模式现在何处”提到的，现在已经出版了一些将模式用于工业软件开发的经

验报告。我们希望这种报告的数量继续增加，特别是在现在还未出现这种报告的领域中，包括电子医疗影像处理、实时飞行控制系统和全球因特网电子商务系统。

这种文档化经验的积累使研究者和开发者能识别通用的、与领域无关的核心原则，以有效地使用模式。许多原则已经被隐式地理解和应用于早期项目中。有了足够的实验报告后，我们希望模式团体能够显式地提取和文档化这些通用的核心原则，帮助开发者提高成功应用模式的能力。

我们还希望，将这些核心原则组织在一起，形成一种“方法”或原则集，用于基于模式的软件开发[Bus00a]。这种方法或原则集将补充和完善现有的软件开发方法（如统一的软件开发过程[JBR99]）和文档化“方法”（如UML[BRJ98]）。现在的UML模型主要指定软件设计的对象（What）和时间（When）。未来面向模式的方法和原则将有助于证明为什么特定模型适合解决特定领域中的主要问题。 538

我们相信最成功的面向模式方法和原则将是通过归纳从专家模式用户和软件设计师收集到的大量经验，采用“自底向上”的方法建立的。这种归纳过程比试图“自顶向下”地定义面向模式方法和原则的方法有更多的机会获得成功。同时，有了用于基于模式的软件开发的基于经验的方法和原则，我们希望将它们编进软件开发工具中。这种工具与现有的相比有更广泛的用途，因为工具将来自于经验的好的实践自动化了。

7.3.4 模式文档

我们希望书籍继续成为发布有关模式具体信息的主要媒体。然而，我们预计Web的使用将会增多，基于Web的协议和工具（如HTML和XML）会使模式文档化和将它们连接为模式语言变得更容易。前期的工作在一些孤立的语境中取得了一定的成功，如Ward Cunningham 的WikiWikiWeb中的Portland 模式仓库（Portland Pattern Repository）[PPR]。在这一领域中实践状况（State-of-the-practice）可能被三个趋势推动：

- 正在增长的Web访问无所不在。
- 低成本高质量的电子书阅读器的出现。
- 更有效地向作者付酬的电子商务手段。

7.3.5 形式化模式和模式语言

对形式化模式概念的研究肯定会继续，并且形式化会扩展这种研究，以获得更多可理解的模式语言。然而，还不能确定这项工作在将来是否会产生比现在更巨大的影响。我们希望会出现形式化特定系统或语境中的模式特定实例的工作，以将特定模式的实现文档化。 539

某些形式化技术（如根据约定的设计[Mey97]）可以有助于防止在特定应用中不适当地使用模式。因此关注模式实例的形式化对生产软件的开发有实际的意义，这样有助于改善系统设计和实施质量。

7.3.6 软件开发过程和组织

在模式语言上进行的许多工作集中在改进软件开发过程和组织上[Cope95][Cock97][Ris98][Har96]。一些软件开发团队已经应用了这些模式，以帮助他们改进工作环境的质量[Gab96]和生产

效率。这项工作也开始影响学术性的软件过程研究团体。我们希望在以后几年中看到模式和软件开发过程及组织的更多的集成。

模式看起来对“无足轻重的”软件开发过程的文档化特别有用，如开放源码[OS99]和极限编程(eXtreme Programming, XP)[Beck99][KiLe00]过程。另外，基于再分解的过程[Opd92][FBBOR99]已经并且正在成功地应用模式，以重构现有软件，增强它的模快性、可维护性和可重用性。

7.3.7 教育

在1996年，我们未能预见到对所谓“教学法模式”[PPP00][Bir00]的文档化的巨大成就。教学法模式的目标是获得高效教育的原则。并且，在本科课程中模式已广泛地用于软件设计和编程教学[Lea99b][Sch99][LG00][TM00][Wal00]。它们也用于对软件从业人员进行培训和再培训。

540

我们希望近期，许多本科课程将应用和讲授模式，传播关于软件体系结构和设计的良好实践和合理的原则的关键知识问题[Ken98]。现在在许多专业软件开发组织中，模式已经普遍使用。成功应用模式的基本经验也有了积累。学术机构将有希望收录这些经验，帮助教育从事模式和对模式高效地文档化和应用的下一代软件开发从业人员。当这些学生毕业并开始他们的职业生涯时，模式将比现在更加普及。

7.3.8 长远预测

预测遥远的将来，我们相信模式知识、模式语言和框架组件将获得极大的发展，使软件开发者拥有类似于生物学家所拥有的方法和工具。在经过数个世纪的试验和建模后，生物学家已经完全或部分破译了某些生物体的基因代码。在这些知识的指引下，生物学家现在开始理解DNA序列中不同生物间共同的以及使它们各不相同的基本要素。

生物学家们正在应用这些信息开发新的技术来控制DNA，以医治遗传疾病，如膀胱纤维化（Cystic Fibrosis）症。在这个领域，科学家们正在：

- 探索DNA序列的核心属性，并且
- 发明能安全且符合伦理地控制这些序列的新技术。

生物系统和软件系统都是高复杂度的。然而当把它们分解为基本组件时（如DNA序列和基因、模式和模式语言），就能够更容易地理解和控制它们。我们相信，随着核心软件模式和模式语言被理解和以可重用组件和框架的形式被具体化，人们可以以更可预期的方式开发更大规模的复杂软件系统。

我们预期，当前对模式和模式语言的研究和开发将最终导致对核心软件属性的突破性发现。

541

然后，这些“软件DNA”的发现将刺激新过程、方法、工具、组件、框架、体系结构和语言的发明。这些制品将使我们能够比在现在的技术状态下更有效地控制大型软件的复杂性。

7.4 对预测未来的临别思考

棒球名人堂里的哲学家Yogi Berra[⊖]说过一句非常意味深长的话：“预言很难，尤其是预测未

⊖ Yogi Berra是美国著名的棒球运动员和教练，1972年入选美国国家棒球名人堂。他还以言语多警句而闻名，有人称他为“言语被引用次数最多的人”。——编者注

来”。我们在1996年做出的预言并没有全部成为现实，特别是对面向模式的软件工具的使用将变得很普及这一预测。在本章，我们对未来的预测做了一些修订，其中包含有无关大雅的不确定性。特别是，我们现在的预测基于：

- 我们对模式团体的了解。
- 正在进行和处于计划中的事业和研究活动——在我们可以察觉到的范围内。
- 我们指导的或我们认为大有前途的研究方向。

希望我们大多数的预测将会成为现实。然而，模式和模式语言的未来可能会产生我们未能预期到的新方向，如基于模式的模型检查。请将本章的预测看做许多正在进行的关于模式和它们的未来的对话中的一个可能的想像，而不是灵验的神喻。

当几年后下一本POSA系列丛书出版的时候，我们将回顾这些预测，正如我们回顾1996年在《模式系统》[POSA1]中所做的预测一样。到那时我们将知道什么真正地发生了，并且有希望事后聪明地解释为什么！

致谢

感谢对本章给我们提出宝贵意见的审稿人，包括Joe Bergin、Bob Hanmer、Neil Harrison、Doug Lea、Ossama Othman、Carlos O'Ryan、Dirk Riehle、Linda Rising、John Vlissides和Roger Whitney。

第8章
结 束 语

"写作是一种冒险。
开始，它是一种自娱和消遣。
然后它变成情人，接着变成主人，
然后它成为暴君。最后，你就成了快要顺从于它的奴仆，
于是你杀死这个怪物并将它公诸于世。"
温斯顿·丘吉尔

本章邀请读者从我们的手中接过指挥棒，接下来为并发和网络化的面向对象应用程序和中间件建立一种更复杂的模式语言。

本书记录在几十年开发和应用这些软件系统类型的过程中得到的精髓和模式。这本书不仅代表了作者的实践经验，也代表了在过去20年间与作者共事的几百名研究者和开发人员的实践经验。而且，这里介绍的很多模式已经被成功地应用在很多系统中，而作者并没有直接参与这些系统。在全书的"已知使用"中列出了这些系统。

即便如此，正如在第6.2节"用于中间件和应用程序的模式语言"所讨论的，本书中的模式只涉及到最终将包括并发和网络化的所有特性的模式语言的基础部分。与所有关于模式的文献一样，这里介绍的模式也总是在不断地进步，我们只能根据我们共同拥有的知识撰写本书，而且还受到"线性"的文档形式、最大页数、出版期限和以及职业责任和家庭责任等的各种强制条件的限制。无疑，在一些模式、实现活动、结论和已知的使用中还有很多内容没有涉及到。

我们认为，本书的编著是我们在终身探索和描绘软件开发的迷人世界、获取"软件DNA"的本质的历程中向前迈进的一步。在今后的十年间，我们还将继续完善、完成和实现POSA2中包括的模式和模式语言，以进一步形成和充实POSA系列丛书。

最后，我们恳请你加入到我们的活动中，一起设计并推敲这里记录的模式。只有在大家的帮助下，我们才能够保证这些模式描述可以对现实世界中的最好的实践和经验系统化。而且还将帮助我们更广泛和更深入地理解在并发和网络化软件中发现的内在的和偶然的复杂性的解决方案。

附录A
词 汇 表

词汇表包含了许多在本书中经常用到的术语。所有这些术语都与并发和网络化软件的各种特性有关。如果一个术语的定义中包含词汇表中的其他术语，将用楷体表示这些术语。

这里已经省略了许多仅用于某一方面的术语，例如扩展接口和异步完成标记模式中的网络管理术语。这些术语在语境中定义了，而没有将它们包含在词汇表里。为了简洁，这里也省略了像“模式”、“软件体系结构”和“惯用法”这样的术语，在[POSA1]的不同章节对它们有深入的解释。

Abstract Class（抽象类） 没有实现接口中定义的所有方法的类。抽象类为其子类定义了公共抽象。

Abstract Component（抽象组件） 为其他组件定义一个或多个接口的组件。抽象组件不但可以像抽象类那样能被明确地给出，而且可以被隐含地给出，如C++模板函数的类参数。抽象组件是开发多态性和实现灵活系统的基础。为了避免将模式应用仅限于面向对象语言，这个术语的用法与抽象类相同。

Abstract Method（抽象方法） 必须由子类定义的类操作声明。

Active Connection Establishment（主动连接建立） 由启动到远程对等体的连接的对等体的应用程序扮演的连接角色（与被动连接建立进行对比）。

Active Object（主动对象） 在除了调用其方法的客户机线程之外的其他线程中执行其方法的对象（与被动对象进行对比）。

Application Programming Interface（应用编程接口, API） 应用编程接口。是软件平台的外部接口，如操作系统，它被建造在它之上的系统或应用程序所使用。

Application（应用程序） 完成客户或用户需求的程序或程序集。

Application Framework（应用框架） 为向相关的应用程序族提供可重用的软件体系结构而相互合作的组件的集成集合。在面向对象的环境中，应用框架由抽象类和具体类组成。这种框架的实例化包括从已有类的组装和从已有的类再分类。

Architectural Pattern（体系结构模式） 体系结构模式表示了软件系统的基本的结构化组织图式。它提供了一组预定义的子系统，规定它们的责任，并包含了用于组织它们之间关系的规则和指南。

Associative Array（相关数组） 通过任意关键值（如字符串，而不仅仅是整数）进行索引的数组。散列表是实现相关数组的一种常用的方式。

Asynchronous I/O（异步I/O） 一种用于发送和接收数据的机制，其中，启动I/O操作后，调用者并不封锁等待操作的完成。

Backus Naur Form（巴科斯范式，BNF） 用于描述语言的语法的标准技术。

Bandwidth（带宽） 通信媒介（如网络或者总线）的一种能力。

Bus（总线） 一种高速的通信通道，它将计算设备（如CPU、磁盘和网络接口）连接起来。

Busy Wait（忙等待） 为了获得一个锁，线程执行一个紧密循环并在每次循环中不断轮询该锁是否可用的一种技术，这种技术与通过休眠并允许其他线程运行来等待锁被释放的技术对应。

Cache Affinity（高速缓存相似性） 一种线程调度优化方法，它将一个线程分派给最近运行过的CPU，从而能保证在该CPU的指令和数据缓存中这些状态仍然存在的概率最大。

Class（类） 面向对象语言中的一个基本构造块。类定义了接口并且封装它的内部数据结构及其实例或对象的功能特性。通过继承，类可以扩展一个或多个类。

Client（客户机） 在我们的描述中，客户机是指调用或使用其他组件提供的功能特性的角色、组件或子系统。

Closure（闭包） 参见方法闭包。

Collaborator（协作者） 与另一个组件合作的组件。CRC卡的组成元素。

Collocation（组合） 与将对象与访问它的客户机放在同一进程或者主机中有关的活动。组合通常可以改变引用的局部性。（与分布进行对比）

Completion Event（完成事件） 包含有响应信息的事件，响应信息与由客户机启动的请求事件有关。

Component（组件） 完成特定服务或一组服务的软件系统的一个封装好的部分。组件有一个或多个提供对其服务进行访问的接口。它是系统结构的构造块。在程序设计语言层，组件可以是模块、类、对象或相关的函数集。没有实现其接口所有元素的组件被称为抽象组件。

Concrete Class（具体类） 其对象实例化了的类。与抽象类相反，在具体类中实现了所有的方法。这个术语用来区分从抽象超类得到的具体子类。

Concrete Component（具体组件） 实现了定义在接口中的全部元素的组件。与区分具体类与抽象类一样，它用于区分组件和仅定义接口的抽象组件。

Concurrency（并发） 对象、组件或系统执行“逻辑上同时的”操作的能力（与并行性进行对比）。

Condition Variable（条件变量） 条件变量[IEEE96]是一种同步机制，协作的线程用它来临时挂起自己，直到与线程间共享的数据有关的条件表达式达到预期的状态。条件变量常和互斥一起使用，线程对条件表达式求值时必须先获得一个互斥。如果条件表达式为假，则线程自动在该条件变量上挂起自己，并释放该互斥量，使其他线程可以改变共享数据。如果一个操作线程改变了共享数据，该线程通知条件变量，由条件变量自动恢复原先挂起的线程，并再次试图获得互斥。

Connection（连接） 对等体用来在网络化应用的端点间交换数据的全关联。

Container（容器） 包含有许多元素的数据结构的通称。例如，列表、集合和相关数组都是容器。另外，组件模型（如EJB[MaHa99]）和ActiveX控件，定义了提供运行时环境的容器，这种环境能使组件掩盖底层基础设施（如操作系统）的细节。

Continuation（连续性） 一种允许程序的运行系统将方法闭包的控制从程序的一部分转移

到另一部分的语言特性[DBRD91]。

Common Object Request Broker Architecture（公共对象请求代理体系结构，CORBA） 公共对象请求代理体系结构，是由对象管理组（Object Management Group, OMG）定义的分布式对象计算中间件标准。

Central Processing unit（中央处理单元，CPU） 中央处理单元。执行二进制程序指令的硬件组件。

CRC Card（CRC卡） 类-责任-协作者（Class-Responsibility-Collaborator）卡。一种用于描述软件体系结构中类的责任和协作者的设计工具和符号。在本书中，我们也使用CRC卡描述不是类的组件。

Critical Section（临界区） 可以使用临界区将对象或者子系统中不能被并发执行的代码同步化。临界区是遵循如下不变式的指令序列：当一个线程或进程在临界区中执行时，再不能有其他线程或进程在临界区中执行[Tan95]（与读方临界区和写方临界区进行对比）。

Deadlock（死锁） 死锁是一种并发危害，当多个线程试图获得多个锁时，它们在某一个循环等待状态下无限封锁，这时候就发生了死锁。

Daemon（守护进程） 一个在后台连续运行的服务器进程，为客户机执行多种服务。

Data-Mode Socket（数据模式套接字） 参见套接字。

Demarshaling（散集） 将列集的消息从与主机无关的格式转换为主机特定的格式。

Demultiplexing（多路分解） 将输入端口的接收事件分配到它对应的接收者的机制。在输入端口和接收者之间有一种1：N的关系。多路分解通常适用于输入事件和数据流。逆操作被称为“多路复用”。

Design（设计） 由软件开发者执行的活动，其结果是制定系统的软件体系结构。“设计”这个术语也常常可作为这些活动的结果的名称。

Design Pattern（设计模式） 设计模式提供了一种细化软件系统中的组件以及它们之间关系的图式。它描述了常用的通信组件的结构，这一结构解决了特定语境中的一般设计问题[GoF95]。

Device（设备） 在计算和/或通信系统中提供服务的硬件组件。

Device Driver（设备驱动程序） 操作系统内核的一个软件组件，负责控制连接在计算机上的硬件设备或者软件设备，如随机存取存储磁盘。

Distribution（分布） 与将对象放在与访问它的客户机不同的进程或主机中有关的活动。分布用于改善容错性或访问远程资源（与组合进行对比）。

Domain（领域，域） 表示与一个特定问题领域有关的概念、知识和其他术语。经常用“应用领域”来表示应用针对的问题领域。在因特网中，域是一个逻辑定位实体，如uci.edu或者siemens.de。

Dynamic Binding（动态绑定） 将操作名（消息）与对应的代码（方法）的关联延迟到运行时的机制。用于在面向对象语言中实现多态性。

Dynamically Linked Library（动态链接库，DLL） 可以被多个进程共享、能被动态地连接到进程的地址空间中或者地址空间之外的库，使用动态链接库可以改善应用程序运行时的灵

活性和可扩展性。

Endpoint（端点） 连接的一个终点。

Exception-Safe（异常安全） 如果在组件中产生的异常或者从由该组件调用的其他组件传播来的异常不会导致资源泄漏或者不稳定状态，那么该组件是异常安全的。

Event（事件） 传递某一活动的发生信息以及与此活动有关数据的消息。

Factory（工厂） 创建并收集用于实例化和初始化对象或组件实例所需资源的方法或函数。

Flow Control（流控制） 一种通信协议机制，用于防止高速的发送者超越低速的接收者的缓冲和计算资源。

Framework（框架） 参见应用框架。

Full Association（全关联） 在因特网协议域中标识一个TCP连接的五元组。由协议类型、本机地址和端口号、远程地址和端口号组成。

Function（函数） 一个完整的子例程，不带参数或者带有若干个参数，可以向其调用者返回一个值。一般函数是孤立的，而方法则不是，方法与类有关。

Functional Property（功能属性） 系统功能特性的特殊方面，通常与指定的功能需求有关。功能属性既可通过特殊的函数使应用程序的用户直接可见，也可表示它的实现方面，如用于计算函数的算法。

Future（前景） 客户机通过前景[Hal85][LS88]可以在调用一个方法后的任何时刻获得方法的结果。前景为调用的方法保存其结果预留了空间。当客户机要得到结果时，它可以封锁或者进行轮询，直到结果被计算出来了并存储在前景中，从而和前景交互。

Gateway（网关） 网关将网络中的合作组件分开，使它们在彼此没有直接依赖关系的情况下进行交互。

Graphical User Interface（GUI） 图形用户接口，图形用户界面。

Handle（句柄） 句柄标识由操作系统内核管理的资源。通常这些资源包括网络连接、打开文件、定时器和同步对象等。

Hardwiring（硬布线） 编写不可改变的程序，例如使用常数数字或字符串，而不是变量。因为数字本身不能解释它来自哪儿和它的目标，因此这样常数数字也被称为“魔术数字”。

Host（主机） 连接到网络上的可寻址的计算机。

HyperText Transfer Protocol（超文本传输协议，HTTP） 超文本传输协议，是位于TCP之上的一个简单协议层，客户机使用该协议从Web服务器中通过GET请求下载内容。

Idempotent Initialization（等幂初始化） 如果一个对象可以被多次初始化，而不会产生有害的副作用，那么可以说对象初始化是等幂的。

Idiom（惯用法） 惯用法是一种与程序设计语言有关的低层的模式。惯用语描述了如何使用给定语言的各种特性来实现组件的某些特性或者它们之间的关系。

Indication Event（指示事件） 包含有从客户机发送给服务提供者的请求信息的事件。

Inheritance（继承） 一种面向对象语言的特性：允许现有类派生新类。继承定义了实现

重用、子类型关系，或两者都定义。单继承或者多继承都是可能的，这依赖于编程语言。

Inlining（内联） 在编译时进行的代码扩展，用函数或方法体的代码替换对函数/方法调用的代码。对长函数方法体的内联可能导致代码“臃肿”，并带来存储消耗和页面调度的负影响。

Instance（实例） 来自于具体类的一个对象。经常作为面向对象环境中对象的同义词使用。这个术语也可用在其他语境中（与实例化进行对比）。

Instantiation（实例化） 通过模板创建新实例的机制。该术语用在几种语境中。对象是类的实例。对C++模板实例化可以产生新的类或新的函数。对应用框架的实例化可以产生一个应用。术语“实例化一个模式”有时是指取一个已描述过的模式并且在特定的应用语境中填写必要的实现细节。

Instruction and Data Cache（指令和数据缓存） 和CPU放在一起的特殊的高速存储器，可以改善系统的整体性能。

Interface（接口/界面） 类、组件或子系统的公共可访问的部分。

Internet（因特网） 一个基于因特网协议（Internet Protocol, IP）的分布全球的“网络的网络”。对人类来说，和取火及MTV的发明一样重要。

Internet Protocol（IP） 执行包的分段、收集和路由的网络层协议。

Intranet（内联网） 公司内部或其他组织的计算机网络。这样的网络可以防范外来访问，并使用因特网通信技术为公司范围内的信息交流、协同工作和工作流提供平台。

Introspection（自省） 由系统自己检查系统的结构、行为和状态的选择的特性。

Interprocess Communication（进程间通信，IPC） 驻留在不同的地址空间中的进程之间的通信。IPC机制的例子有共享内存、UNIX管道、消息队列和套接字通信等。

Invariant（不变式） 对象、组件或者模块的状态的一种特性，它在时间或空间的某一点上总是成立的。例如，“当控制经过方法`foo()`的第50行时，不变式`a<b`总成立。”

Jitter（抖动） 一系列操作的延迟的标准差。

Late Binding（后期绑定） 是动态绑定的同义词。

Latency（延迟） 操作的延时。

Layer（层） 在层次结构中定义一个特定的服务集的抽象级别。$layer_n$是$layer_{n-1}$服务的消费者，也是$layer_{n+1}$层服务提供者。

Load Balancing（负载平衡） 一种用于在网络中的不同进程和主机之间分配客户机工作负载的技术。

Lock（锁） 用于实现某些类型的临界区的机制。可以被依次获得和释放的锁（如静态互斥量），可以添加到类中。如果多个线程试图同时获得锁，那么只有一个线程能够成功，而其他的线程将会被封锁，直到那个锁可用[Tan92]。其他的加锁机制，如信号灯或阅读器/记录器锁，则定义了不同的同步语义。

Loopback Device（回环设备） 一种类似于应用程序的网络接口的软件设备，不过它简单地将传递给它的消息重定向到同一主机上的另一进程或线程。

Marshaling（列集） 将散集的消息从与主机特定的格式到与主机无关的格式的转换。

Message（消息） 消息用于对象、线程或进程间的通信。在面向对象的系统中，术语消

息用来表示对对象的操作或方法的选择和激活。这类消息是同步的，也就是说，发送者一直等到接收者完成被激活的操作。线程和进程间的通信通常是异步的，也就是发送者继续执行，不必等待接收者的回答。远程过程调用（Remote Procedure Call, RPC）是在网络上实现同步IPC的一种方法。IPC通信协议中的消息由协议所定义的结构组成，而且基本上对高层来说是不可见的，而在像IBM MQSeries或者Microsoft MSMQ消息这样的消息队列系统中，用户可以自定义消息体，高层可以隐式地传递用户数据。

Message Passing（消息传递） 一种用于线程或进程间交换消息的IPC机制（与共享内存进行对比）。

Method（方法） 对象执行的操作。方法在类中定义。该术语也用在“软件开发方法”中，它由一组在开发软件过程中工程师使用的规则、指南和符号组成。

Method Closure（方法闭包） 包含方法的语境的对象，其中可以有方法的参数、对处理该方法的服务者或者完成处理程序的绑定，或者是用于方法结果的远景。

Mix-In（混入） 是指定义了一些额外的接口和功能的“小”类，这些接口和功能将通过多继承加到类中。混入也表示通过继承类来添加这种功能特性的机制。

Middleware（中间件） 一组提供可重用的公用服务和网络编程机制的层或者组件。中间件位于操作系统及其协议栈的上面，但位于任何特殊应用程序的结构和功能特性之下。

Module（模块） 软件系统的语法或概念实体，经常用做组件或子系统的同义词。有时，“模块”也表示编译单元或文件。我们使用前一种含义。在谈到有自己的名字空间的代码体时，其他作者将这个术语作为“包”的等价词。

Monitor（监视器） 监视器将它的函数和其内部变量封装在线程安全的模块中。为了防止出现竞争条件，监视器中还要有一个锁，以保证一次监视器中只有一个线程是激活的。要临时离开监视器的线程可以封锁等待一个条件变量。

Moore's Law（摩尔定律） 一个相当准确的关于微芯片技术的发展速度的推断，具体内容是微芯片的数据存储量每一年翻一番，或者至少每十八个月翻一番。Gordon Moore在1965年为一场演讲做准备时，他发现到那个时候为止，微芯片的容量每年翻一番。在过去几年这一速度慢了一点点，定义也发生了改变——得到了Gordon Moore的同意——以反映这样一个事实，就是每十八个月翻一番。

Multiple Inheritance（多继承） 一个类可以有很多超类的继承。

Multicast（多播） 一种通信协议，允许客户机向多个服务器发送消息。广播是一种特殊形式的多播，其消息发送给一个特定域中的所有服务器。

Mutex（互斥） 互斥是一个“互相排斥”的加锁机制，能确保为了防止出现竞争条件，临界区内一次只有一个线程是活动的。

Network（网络） 允许主机和其他设备交换消息的通信媒介。

Network Interface（网络接口） 与网络中的主机连接的硬件设备。

Non-functional Property（非功能属性） 没有被其功能性描述所覆盖的系统特性。非功能属性经常针对有关可靠性、兼容性、效率、开销、易用性、系统维护或开发等方面的问题。

Object（对象） 在面向对象系统中可识别的实体。对象通过执行方法（操作）响应消息。

对象可以包含数据值以及对其他对象的引用，它们一起确定对象的状态。因此对象包括状态、行为和身份标识。

Object Request Broker（对象请求代理） 中间件层，允许客户机调用分布式对象上的方法而无须考虑对象位置、编程语言、操作系统平台、通信协议或硬件。

On-the-wire Protocol（在线协议） “在线协议”定义高层通信中间件（例如DCE、CORBA或Java RMI），或其他的通信协议（例如HTTP、将消息或对象转换成可通过电线（网络）传输的缓冲区）。术语“电线”也包括诸如微波、光纤和无线电波等传输媒介。

One-way Method Invocation（单向方法调用） 对方法的调用，这种调用只向服务器对象传递参数而不接收任何结果、错误值或其他来自服务器的信息（与双向方法调用进行对比）。

Operating System（操作系统） 根据应用程序和终端用户的要求管理硬件和软件资源的服务和API的集合。

Operating System Kernel（操作系统内核） 核心操作系统服务的集合，这些服务包括进程和线程管理、虚拟存储器和进程间通信（IPC）。

Out-of-Band（带外） 发生在正常的“带内”处理序列之外的协议或机制。

Packet（包） 在TCP/IP协议中用于传递报文头和数据信息的消息。

Parallelism（并行性） 对象、组件或系统执行“在物理上同时”的操作的能力（与并发性进行对比）。

Parameter（参数） 传递给函数、方法或者是参数化类型的一个数据类型的实例或者对象。

Parameterized Type（参数化类型） 允许其他的类型参数化一个类的一种程序设计语言的特性（与模板进行对比）。

Passive-Mode Socket（被动模式套接字） 参见套接字。

Passive Connection Establishment（被动连接建立） 由接受来自远程对等体的连接的对等体应用程序扮演的连接角色（与主动连接建立进行对比）。

Passive Object（被动对象） 借用调用者的线程执行其方法的对象（与主动对象进行对比）。

Pattern（模式） 模式描述了一种在某些设计语境中的特殊的重复设计问题，并提出了解决这一问题的被证明了的解决方法。通过描述各组成部分、它们各自的责任和关系，以及它们的协作方式，可以定义这种解决方法。

Pattern Language（模式语言） 系统地定义解决软件开发问题的过程的相关的模式集。

Peer-to-peer（对等） 在分布式系统中，对等体是相互通信的进程。与客户机-服务器体系结构中的组件相反，对等体既可以是客户机，又可以是服务器，还可以二者兼备，并且其角色可以动态改变。

Platform（平台） 用于实现系统的硬件和/或软件的组合。软件平台包括操作系统、库和框架。平台实现了可在其上运行应用程序的虚拟机。

Polymorphism（多态性） 一个名字可以表示不同的事情的概念。例如，一个函数名在不同的阶段可以绑定到不同的操作，或一个变量名可以绑定不同类型的对象。这个概念有助于实

现基于抽象的灵活系统。在面向对象语言中，多态性通过操作的动态绑定机制来实现。这暗示着同一段代码可能根据与它协作的对象的不同而有不同的行为。

Port（**端口**） 通信的端点。

Port Number（**端口号**） 在TCP协议中用于标识通信端点的16位整数值。

Priority Inversion（**优先级的颠倒**） 当低优先级的线程或者请求封锁了高优先级的线程或者请求的执行时发生的调度难题。

Process（**进程**） 进程提供了一些资源（如虚拟内存）和保护功能（如用户/组标识符），以及硬件保护的地址空间，这些资源可被进程内一个或多个线程使用。不过，和线程相比，进程维护了更多的状态信息，创建、同步和调度进程需要更多的开销，进程常常通过消息传递或共享内存和其他进程通信。

Protocol（**协议**） 一组描述如何在通信对等体之间交换消息，以及这些消息的语法和语义的规则。

Protocol Stack（**协议栈**） 一组结构上分层的协议。

Quality of Service（**服务质量**） 一组用于控制和改进通信质量（如带宽、延迟和抖动）的策略和机制的集合。

Race Condition（**竞争条件**） 竞争条件是当多个线程在一个没有被正确地串行化的临界区中同时执行时出现的一个并发机会。

Read-side Critical Section（**读方临界区**） 一组遵循如下不变式的指令序列集：当读方临界区一个或多个线程或进程执行时，对应的写方临界区中不能有线程或进程执行（与写方临界区进行对比）。

Reader/WriterLock（**阅读器/记录器锁**） 允许多个线程并发地访问一个资源，但一次只能有一个线程修改资源，并能防止以后的并发访问和修改的锁。

Reification（**具体化**） 为一个抽象创建具体实例的行为。例如，具体反应器实现是反应器模式的具体化，对象具体化了一个类。

Request Event（**请求事件**） 由客户机发送给服务提供者的事件，客户机请求服务提供者按照客户机的要求进行一些处理。

Response Event（**响应事件**） 服务提供者发送的包含有对客户请求事件的响应的事件。

Recursive Mutex（**递归互斥**） 可以被拥有互斥的线程重新获得，又不会引起线程自死锁的锁。

Refactoring（**再分解**） 改进组件或框架的内部结构的递增的活动。

Relationship（**关系**） 组件间的连接。关系可以是静态的也可以是动态的。静态关系直接用源码表示。它们在体系结构内处理组件布局。动态关系处理在组件间的相互作用。它们不会简单地从源码或图中看出。

Responsibility（**责任**） 在特定的语境中对象或组件的功能特性。责任一般被定义为一组语义相关的操作。责任是CRC卡的元素。

Role（**角色**） 在相互关联的组件语境中，一个组件的职责。例如，一个面向对象的类定义了一个它的所有实例都支持的角色。另一个例子就是定义一个所有实现都支持的角色的接口。

如果一个组件支持一个给定的角色，则它必须提供定义该角色的接口的实现。组件通过实现不同的接口而呈现不同的角色。不同的组件通过实现同一接口而呈现同一角色，这样客户机可以针对特定的角色而多态性地对待它们。既便是在同一个模式中，一个已实现的组件也可以扮演不同的角色。

Scheduler（调度程序） 决定线程或者请求事件被执行的顺序的机制。

Semaphore（信号灯） 具有一个计数器的加锁机制。只要计数器的值大于零，就有一个线程可以获得信号灯而不用封锁。不过，当计数器为零时，线程就会封锁信号灯，直到信号灯的计数器变为大于零的数。信号灯的计数器变化的原因是另一个线程释放了信号灯，而增加了计数。

SEP 可表示如下3个概念之一。其他人问题（Somebody Else's Problem）、软件工程过程（Software Engineering Process）或使用模式的软件工程（Software Engineering with Pattern）。

Serialization（串行化） 为防止出现竞争条件，保证一次只有一个线程在临界区中执行的一种机制。

Servant（服务程序） 由客户机请求触发的组件。当客户机请求到达时，服务者试图自己完成该请求，或将子任务委托给其他组件完成该请求。

Server（服务器） 服务器表示能向客户机提供服务（如中间件功能特性、数据库访问或者Web网页的访问）应用程序。在分布式对象计算中间件中，通过表示分布式对象的服务者来典型地实现这些服务。

Service（服务） 由服务提供者或者服务器向客户机提供的一组功能特性。

Shared Memory（共享内存） 允许一台计算机上的多个进程共享公共的存储段的操作系统机制（与消息传递进行对比）。

Single Inheritance（单继承） 最多只有一个直接超类的类的继承。

Socket（套接字） 一组与网络编程有关的术语。套接字是一个通信端点，标识了网络地址和端口号。套接字API是一组大多数操作系统支持的函数调用，网络应用程序用它来建立连接并通过套接字端点进行通信。数据模式的套接字用于在连接的对等体之间交换数据。被动模式的套接字是一个向连接的数据模式套接字返回一个句柄的工厂。

Software Architecture（软件体系结构） 软件体系结构是对软件系统的子系统、组件以及它们之间的关系的描述。通常通过不同的视图来定义子系统和组件，以说明软件系统的相关的功能属性和非功能属性。一个系统的软件体系结构是从软件设计活动导出的人工制品。

Starvation（饿死） 当一个或者多个线程不断被高优先级线程抢先而永远得不到执行时出现的调度问题。

Subclass(子类) 从超类继承而来的一个类。

Subsystem（子系统） 一组协作组件完成一项给定服务。在软件体系结构内子系统被视为单独实体。通过与其他子系统和组件的交互完成指派给它的任务。

Superclass（超类） 其他类可以继承的类。

Synchronization（同步化） 协调线程执行顺序的加锁机制。

Synchronous I/O（同步I/O） 用于发送和接收数据的一种机制，其中，I/O操作启动后，

调用者封锁等待操作完成。

System（系统） 软件和/或硬件的集合，执行一个或多个服务。系统可以是平台、应用程序或者两者兼而有之。

System Family（系统族） 一组解决相似服务的相关系统。系统族中的系统共享它们的软件体系结构和实现的大部分内容，这是因为每个系统都由相同的框架导出。当单个系统逐步演化时，发布的不同版本也建立了系统族。

Template（模板） C++程序设计语言的一种特性，允许用不同的类型、常数或者指向函数的指针来参数化类和函数。模板通常称为类属的或参数化类型。

Thread（线程） 在可被其他线程共享的地址空间中执行的独立指令序列。每个线程运行时有自己的栈和寄存器，这样可以在不封锁正在并发执行的其他线程的情况下进行同步I/O操作。和进程相比，线程维护了最少的状态信息，线程创建、同步和调度的开销相对小一些，而且通常通过全局内存的对象而不是通过共享内存来和其他线程通信。

Thread-per-Connection（每个连接一个线程） 为每个网络连接关联一个线程的并发模型。该模型在整个连接期间用一个单独的线程处理连接到服务器的每个客户机。对于必须支持与多个客户机的长时间会话的服务器来说，这种模型是有用的。但对于像*HTTP* 1.0 Web浏览器这样的客户机则用处不大，这样的客户机将每个连接与一个请求关联，对它有效的模型是每个请求一个线程的模型。

Thread-per-Request（每个请求一个线程） 为每个请求产生一个新线程的并发模型。对于必须处理来自多个客户机的长时间请求事件（如数据库查询）的服务器来说，这种模型是有用的。而对于短时间的请求则用处不大，因为为每个请求产生一个新线程会带来额外开销。如果多客户机同时发送请求，则这个模型还可能消耗大量的操作系统资源。

Thread Pool（线程池） 分配一个可以同时执行请求事件的线程池的并发模型。这种模型是每个请求一个线程模型的变体，这种模型预先创建一个线程池，从而分摊了线程创建的开销。对于想限制自身消耗的操作系统资源数量的服务器来说，这种模型是有用的。可以并发地执行客户机请求，直到同时请求的数量超过了池中线程的数量。此时，将其他的请求放在队列中，直到有一个线程可用。

Transmission Control Protocol（传输控制协议，TCP） 一种面向连接能保证本地和远程进程间可靠地按顺序和非重复地交换数据的字节流的传输协议。

Transport Endpoint（传输端点） 在传输层连接对等体应用程序的端点。

Transport Layer（传输层） 协议栈中负责端到端数据传输和连接管理的层。

Transport Layer Interface（传输层接口，TLI） TLI是由System V UNIX提供的一组函数调用，网络应用程序用它通过连接的传输端点建立连接和通信。

Two-way Method Invocation（双向方法调用） 对一个方法的调用，将参数传给服务器对象，并接收来自于服务器的结果（与单向方法调用进行对比）。

Unicode（统一的字符编码标准） 使用16位编码的字符表示标准。统一的字符编码标准包括适合所有可书写的语言、标点符号、数学符号以及其他符号。

Upcall（上调） 软件体系结构中的较低层向较高层发出的回调[Cla85]。

User Datagram Protocol（用户数据报协议，UDP） 一种不可靠的、无连接的本地和远程进程间交换数据报消息的传输协议。

View（视图） 视图表示软件体系系统中强调软件体系结构的特定属性的一部分特征。

Virtual Machine（虚拟机） 一个向高层应用程序或者其他的虚拟机提供一组服务的抽象层。

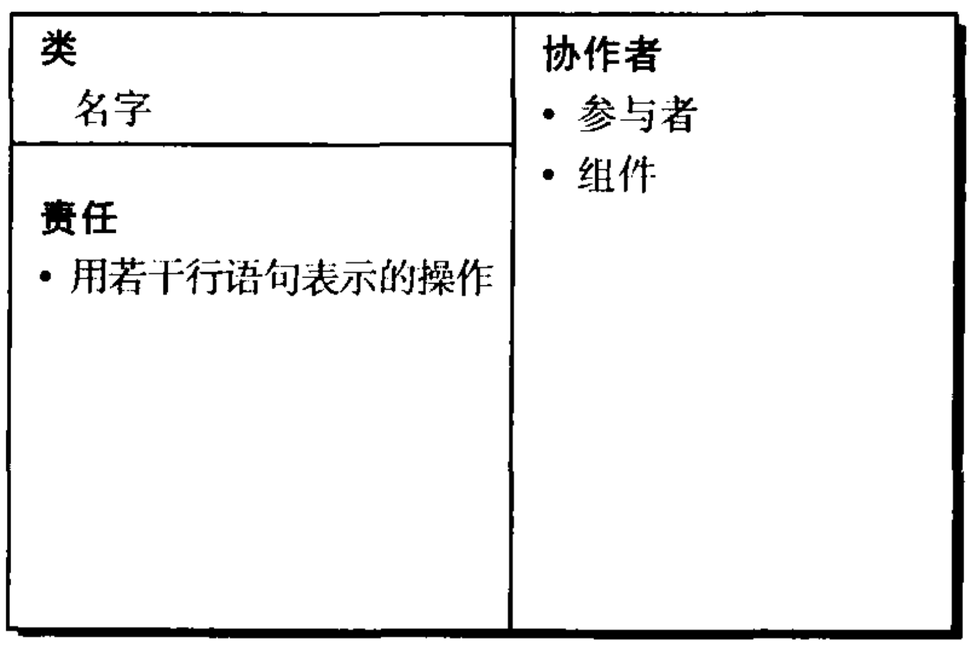
Virtual Memory（虚拟存储器） 允许开发人员编写其地址空间大于计算机的物理存储量的应用程序的操作系统机制。

Write-side Critical Section（写方临界区） 一组遵循如下不变式的指令序列：在写方临界区最多有一个线程或进程被执行，当在写方临界区一个线程或进程执行时，对应的读方临界区中不能有线程或进程被执行（与读方临界区进行对比）。

附录B
符　　号

类-责任-协作者卡

类-责任-协作者卡片 (Class-Responsibility-Collaborator, CRC卡片) [BeCu89]用于以一种非形式化的方式，特别是在早期的软件开发阶段识别和定义应用的对象或组件。

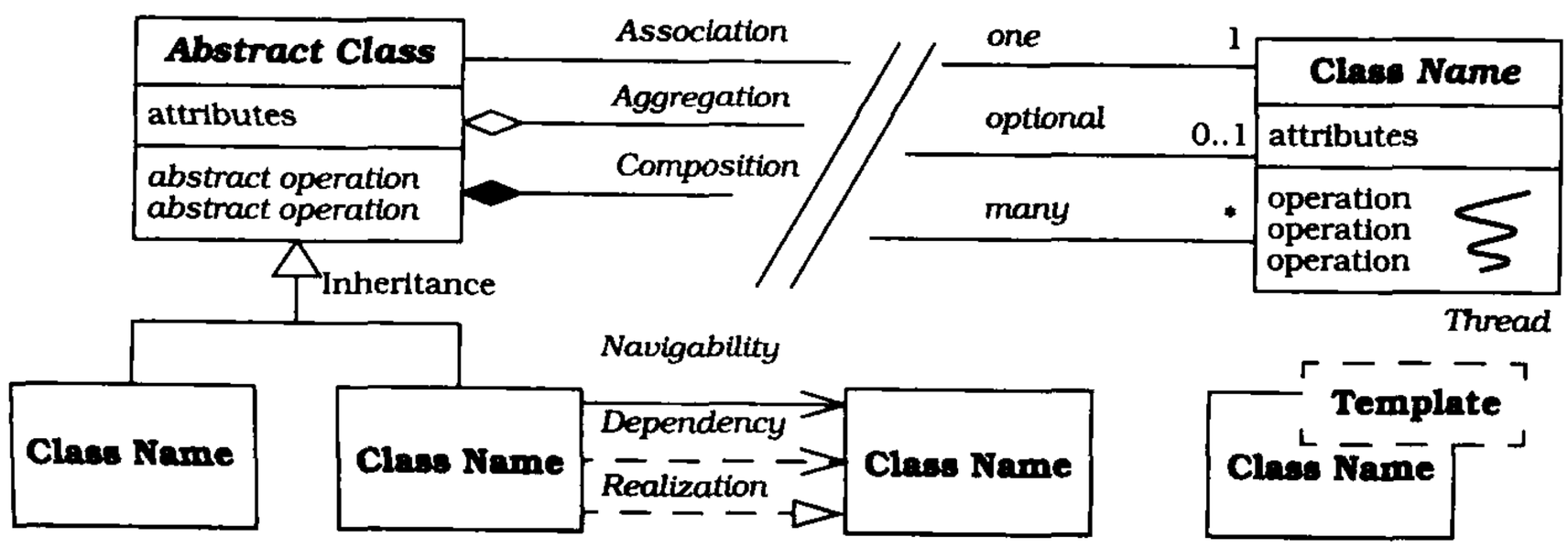


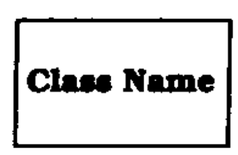
CRC卡片描述了组件、对象或者对象类。该卡由三个域组成，分别描述组件的名字、组件的责任以及其他协作组件的名字。术语“类”是历史用法[Ree92]——我们也将CRC卡片用于其他组件类型甚至单个对象。

UML类图

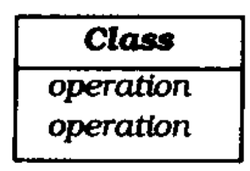
统一建模语言（Unified Modeling Language, UML）[BRJ98]是一种广泛使用的面向对象分析和设计的方法。在UML中一类最重要的图就是类图。类图描述了系统中对象的类型，以及它们之间存在的各种不同的静态关系。

作为对标准UML符号的扩展，我们引入了一个符号，表示运行在独立线程中的一个类的实例。

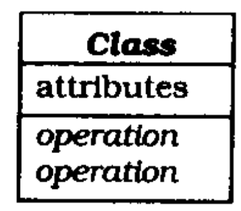




类(Class)　用一个矩形框表示类，其中有类的名字，还可以包括类的属性和操作。用*斜体*表示抽象类的名字以及相应的抽象方法。



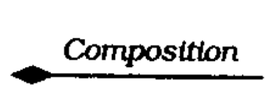
操作(Operation)　在类方框中列出的操作名。操作名表示类的方法。用*斜体*表示提供多态性接口的抽象操作。



属性(Attribute)　在类方框中列出的属性名。它们表示类的实例变量。

关联(Association)　是一条连接类的线。关联是可选的，也可以有多条。在关联的一端有一个数字表示它的基数。类的关联表示类之间除了聚合、组合和继承之外的任何关系。

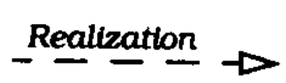
聚合(Aggregation)　在关联线的一端用中空的菱形表示一个类使用了在关联的另一端的类。



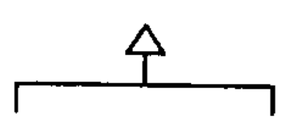
组合(Composition)　在关联线的一端用实心的菱形表示在关联的另一端上的类是类的一部分，它们有着相同的生命期。

导航(Navigability)　表示一个有向的关联。

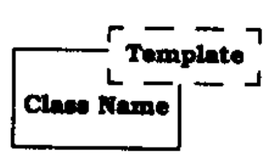
依赖(Dependency)　依赖表示为了某些目的，一个类使用另一个类的接口。



实现(Realization)　实现表示规范和其实现之间的关系。



继承(Inheritance)　用关联线顶端的三角形表示继承。三角形的顶点指向超类。



模板(Template)　在类盒的上面附加一个带虚线的矩形框，表示参数化的类。带虚线的矩形框中列出模板参数。

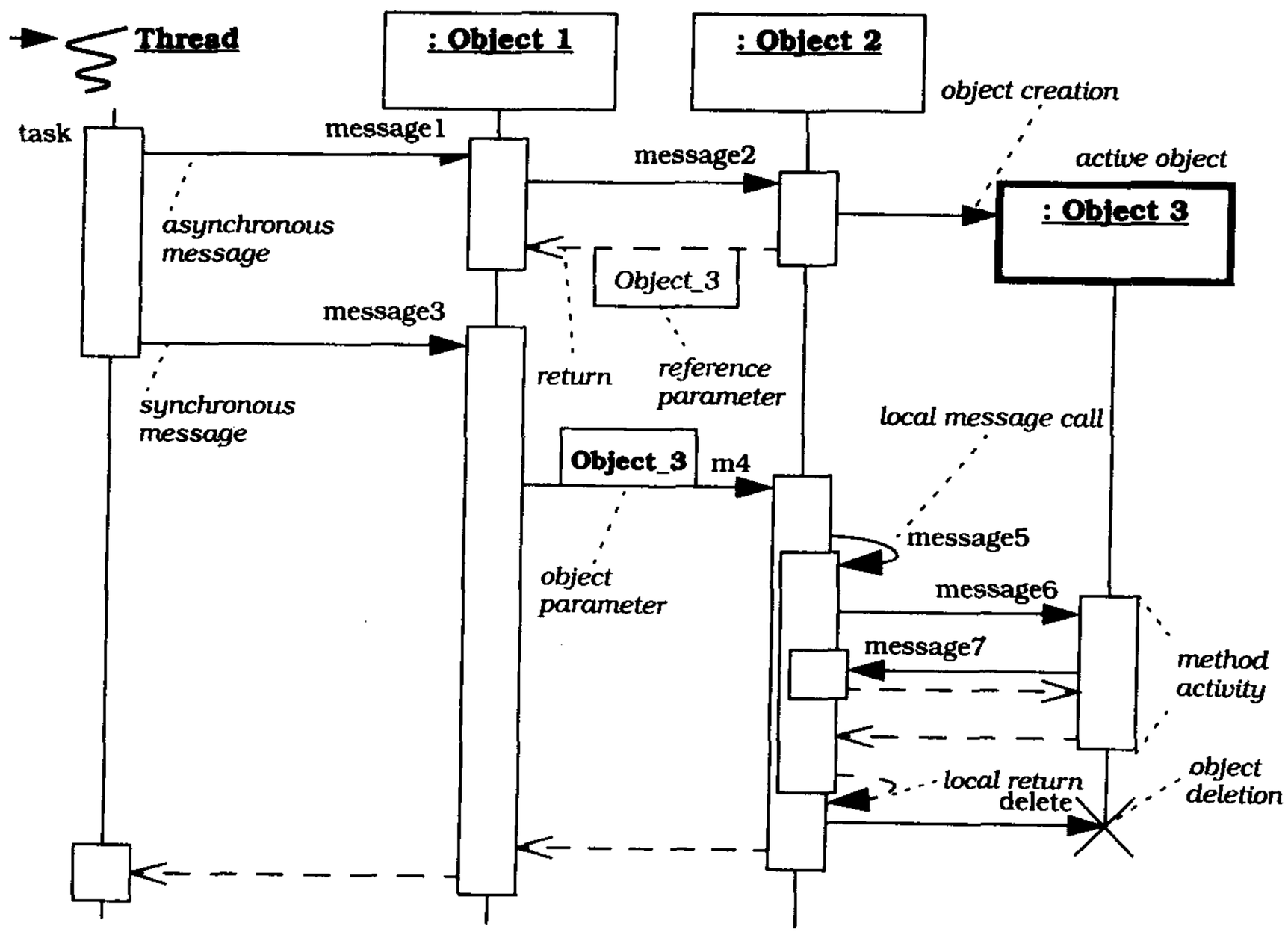
线程(Thread)　用一条波浪线附在类上，表示该类的实例运行在一个独立的线程中。这个线程符号是对UML类图符号的扩展，原来的UML类图缺乏表示多线程的有效方式。

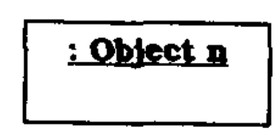
UML顺序图

UML顺序图(UML Sequence Diagram)描述了对象组如何按照某种行为进行协作[BRJ98]。这种符号基于在[POSA1]中定义的对象消息顺序图（Object Message Sequencing Chart, OMSC）。

不过，除了标准的UML符号之外，我们还使用了在原OMSC定义中规定的符号，来说明参数传递过程。这样可以指明对象的责任何时从一个对象传给另一个对象，或者一个对象何时包

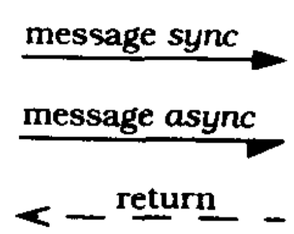
含对另一个对象的引用。除此之外，我们引入了一个特殊的符号，用来标记由控制线程触发的活动，而不是一特殊UML活动对象。





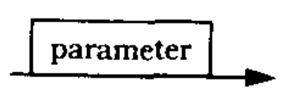
对象(Object)　在顺序图中用矩形框表示对象或者组件。用带有下划线的模式中组件的名字作为矩形框的标记。在顺序图中，发送或接收消息的对象是一个与方框底端连接的垂直条。表示主动对象的方框是粗线条的。

时间(Time)　时间顺序是自顶向下的，时间轴是没有刻度的。

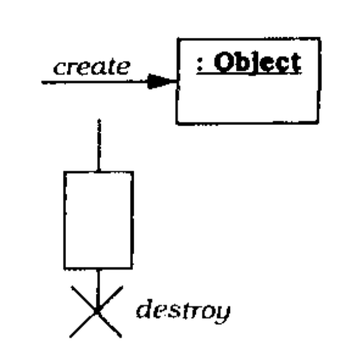


消息(Message)　用箭头表示对象之间的消息。如果可行的话，用方法名标识这些箭头。用带有实心全箭头的箭线表示同步消息，而用带有实心半箭头的箭线表示异步消息。用一条带空心箭头的虚箭线表示返回消息。

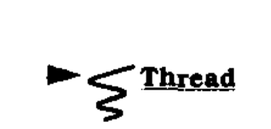
对象活动(Object Activity)　将矩形框放在与对象连接的垂直条上，来表示执行某一功能、过程或者方法的对象活动。对象也可以向自己发送消息，以激活其他的方法。用向右稍稍偏离的嵌套盒表示这种情形。



参数(Parameter)　只有在参数对于顺序图的理解很重要时才会显式地表示参数。将消息的参数表示成位于箭头上方的方框，而表示返回参数的方框位于返回箭头线的下方。如果将参数对象的责任沿着箭头传递，用**黑体**显示对象的名字。如果仅仅将对象的引用作为参数传递，用*斜体*显示对象的名字。这种符号是对UML顺序图的扩展。



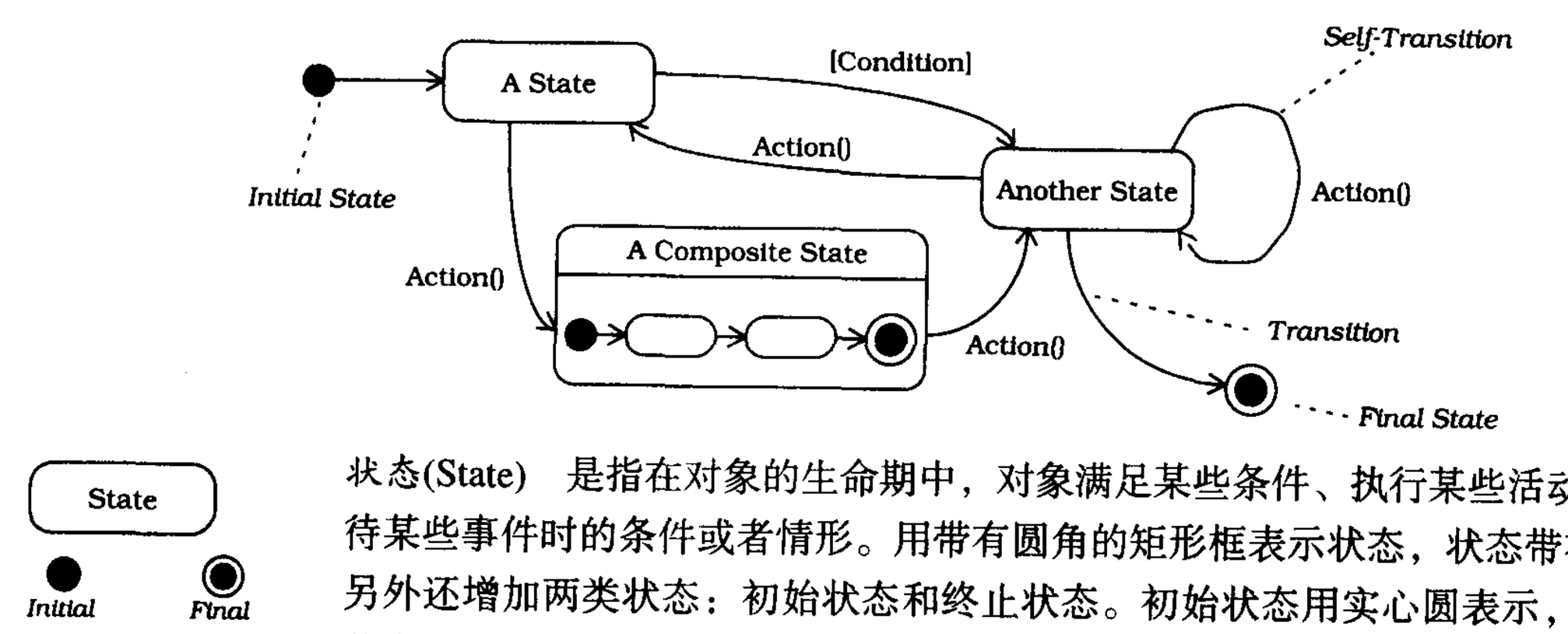
对象生命期(Object Lifecycle) 在大多数情况下，我们假定所有相关的对象都已经存在，所以在顺序图的顶端画出相应的方框。如果顺序图中还要表示对象的建立，可以使用一个指到顺序图内一方框的箭头来表示。用位于垂直条一端的十字架表示对象的销毁。这种符号对应于C++中的构造函数和析构函数的调用。

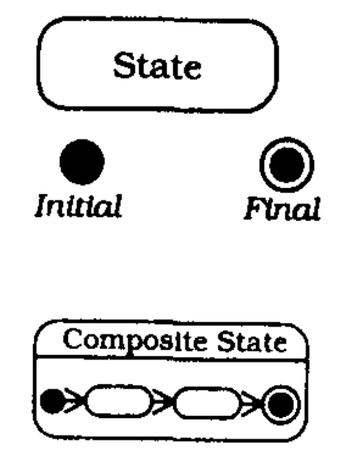


线程(Thread) 用附加在时间线上的波浪线表示发送或接收消息的控制线程。线程的名字带有下划线。线程符号是对UML顺序图符号的扩展，用它可以表示由运行在某一控制线程内的任何对象激活的活动，而不是由特定主动对象激活的活动。

UML状态图

UML状态图(UML Statechart Diagram)定义了对象或者交互在其生命期中为响应某些事件所经历的状态序列，以及相应的动作[BRJ98]。本书只使用基本的状态图符号。





状态(State) 是指在对象的生命期中，对象满足某些条件、执行某些活动或者等待某些事件时的条件或者情形。用带有圆角的矩形框表示状态，状态带有名字。另外还增加两类状态：初始状态和终止状态。初始状态用实心圆表示，指进入状态机的入口点；终止状态用嵌有一个小实心圆的圆表示，指状态机的完成。

组合状态(Composite State) 由不相交的子状态序列组成的状态。

转移(Transition) 指状态机内两个状态之间的关系。转移表示当一个特定的事件发生（如动作发生或者条件满足）时，对象从第一个状态进入第二个状态。如果源状态和目标状态相同，转移就被称为“自转移”。

动作(Action) 指可执行的计算过程（如一个方法），对于当前状态的变化，从概念上讲这种计算是原子的。

条件(Condition) 条件是一个哨兵（guard），当它为真时，触发一次转换。

附录C
参考文献

[ABLL92] T.E. Anderson, B.N. Bershad, E.D. Lazowska, H.M. Levy: *Scheduler Activation: Effective Kernel Support for the User-Level Management of Parallelism*, ACM Transactions on Computer Systems, pp. 53–79, February 1992

[ACGH+96] M. Adams, J. Coplien, R. Gamoke, R. Hanmer, F. Keeve, K. Nicodemus: *Fault-Tolerant Telecommunication System Patterns*, in [PLoPD2], 1996

[Ada79] D. Adams: *The Hitchhiker's Guide to the Galaxy*, Pan Books Ltd., London, 1979

[AG98] K. Arnold, J. Gosling: *The Java Programming Language (Java Series)*, 2nd edition, Addison-Wesley, 1998

[Agha86] G. Agha: *A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986

[AIS77] C. Alexander, S. Ishikawa, M. Silverstein with M. Jacobson, I. Fiksdahl-King, S. Angel: *A Pattern Language – Towns·Buildings·Construction*, Oxford University Press, 1977

[Ale79] C. Alexander: *The Timeless Way of Building*, Oxford University Press, 1979

[Amb99a] S.W. Ambler: *Process Patterns: Building Large-Scale Systems Using Object Technology*, Cambridge University Press, 1999

[Amb99b] S.W. Ambler: *More Process Patterns: Delivering Large-Scale Systems Using Object Technology*, Cambridge University Press, 1999

[Ant96] D.L.G. Anthony: *Patterns for Classroom Education*, in [PLoPD2], 1996

[AONIX99] Aonix: Software Through Pictures, Aonix, `http://www.aonix.com/`, 1999

[ARSK00] A.B. Arulanthu, C. O'Ryan, D.C. Schmidt, M. Kircher: *The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging*, in J. Sventek, G. Coulson (eds.): Middleware 2000, Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms, Springer, 2000, ACM/IFIP, Lecture Notes in Computer Science, Springer, 2000

[Aus98] M.H. Austern: *Generic Programming and the STL : Using and Extending the*

C++ Standard, Addison-Wesley, 1998

[Bak97] S. Baker: *CORBA Distributed Objects using Orbix*, Addison-Wesley, 1997

[BaLee98] R.E. Barkley, T.P.Lee: *A Heap-Based Callout Implementation to Meet Real-Time Needs*, Proceedings of the USENIX Summer Conference, USENIX Association, pp. 213–222, June 1998

[BaMo98] G. Banga, J.C. Mogul: *Scalable Kernel Performance for Internet Servers Under Realistic Loads*, Proceedings of the USENIX 1998 Annual Technical Conference, USENIX, New Orleans, Louisiana, June 1998

[Ban98] J. Bansiya: *Automating Design-Pattern Identification*, Dr. Dobb's Journal, June 1998

[Bat79] G. Bateson: *Mind and Nature: A Necessary Unity*, Bantam Books, 1979

[BBC94] G. Blaine, M. Boyd, S. Crider: *Project Spectrum: Scalable Bandwidth for the BJC Health System*, HIMSS, Health Care Communications, pp. 71–81, 1994

[Beck97] K. Beck: *Smalltalk Best Practice Patterns*, Prentice Hall, 1997

[Beck99] K. Beck: *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999

[BeCu89] K. Beck, W. Cunningham: *A Laboratory For Teaching Object-Oriented Thinking*, Proceedings of OOPSLA '89, N. Meyrowitz (ed.), special issue of SIGPLAN Notices, vol. 24, no. 10, pp. 1–6, October1989

[Ben90] M. Ben-Ari: *Principles of Concurrent and Distributed Programming*, Prentice Hall, 1990

[Ber95] S. Berczuk: *A Pattern for Separating Assembly and Processing*, in [PLoPD1], 1995

[BGHS98] F. Buschmann, A. Geisler, T. Heimke, C. Schuderer: *Framework-Based Software Architectures for Process Automation Systems*, Proceedings of the 9th IFAC Symposium on Automation in Mining, Mineral and Metal Processing (MMM '98), Cologne, Germany, 1998

[BGJ99] A. Berner, M. Glinz, S. Joos: *A Classification of Stereotypes for Object-Oriented Modeling Languages*, Proceedings of <<UML>>'99 – The Unified Modeling Language, (eds.) R. France, B. Rumpe, Lecture Notes in Computer Science, 1723, Springer, 1999

[BFF96] T. Berners-Lee, R. T. Fielding, H. Frystyk: *Hypertext Transfer Protocol – HTTP/1.0*, Network Working Group, RFC 1945, May 1996

[BFVY96] F.J. Budinsky, M.A. Finnie, J.M. Vlissides, P.S. Yu: *Automatic code generation from design patterns*, IBM Systems Journal, vol. 35, no. 2, 1996

[BHG00] H. Berndt, T. Hamada, P. Graubmann: *TINA: Its Achievements and Future*

Directions, IEEE Communication Surveys, on-line publication, http://www.comsoc.org/pubs/surveys, 2000

[Bir91] A.D. Birrell: *An Introduction to Programming with Threads*, Systems Programming with Modula-3, Greg Nelson (ed.), Prentice Hall, 1991

[Bir00] J. Birgin: *Fourteen Pedagogical Patterns*, http://csis.pace.edu/~bergin/PedPat1.3.html, 2000

[Bl90] D.L. Black: *Scheduling Support for Concurrency Parallelism in the Mach Operating System*, IEEE Computer, vol. 23, pp. 23–33, May 1990

[Bl91] U.D. Black: *OSI: A Model for Computer Communications Standards*, Prentice Hall, 1991

[BMBMM98] W.J. Brown, R.C. Malveau, W.H. Brown, H.W. McCormick, T.J. Mowbray: *Antipatterns: Refactoring Software, Architecture, and Projects in Crisis*, John Wiley & Sons, 1998

[BMT99] W.J. Brown, H.W. McCormick, S.W. Thomas: *Anti-Patterns and Patterns in Configuration Management*, John Wiley & Sons, 1999

[Boe81] B.W. Boehm: *Software Engineering Economics*, Prentice Hall, 1991

[Boo94] G. Booch: *Object-Oriented Analysis and Design With Applications*, 2nd edition, Benjamin/Cummings, Redwood City, California, 1994

[Box97] D. Box: *Essential COM*, Addison-Wesley, 1997

[BR94] K. Birman, R. van Renesse: *Reliable Distributed Computing with the Isis Toolkit*, IEEE Computer Society Press, 1994

[BRJ98] G. Booch, J. Rumbaugh, I. Jacobsen: *The Unified Modeling Language User Guide*, Addison-Wesley, 1998

[Broo87] F.P. Brooks: *No Silver Bullet: Essence and Accidents of Software Engineering*, IEEE Computer, vol. 20, no. 4, pp. 10–19, April 1987

[Bus98] F. Buschmann: *Real-Time Constraints as Strategies*, http://www.posa.uci.edu/, 1998

[Bus00a] F. Buschmann: *Applying Patterns*, http://www.posa.uci.edu/, 2000

[Bus00b] F. Buschmann: *Inside Patterns*, http://www.posa.uci.edu/, 2000

[BV93] G. Booch, M. Vilot: *Simplifying the Booch Components*, C++ Report, vol. 5, June 1993

[CarEl95] M.D. Carroll, M.A. Ellis (Contributor): *Designing and Coding Reusable C++*, Addison Wesley ,1995

[CFFT97] T. Chaney, A. Fingerhut, M. Flucke, J. S. Turner: *Design of a Gigabit ATM*

Switch, IEEE INFOCOM '97, IEEE Computer Society Press, Kobe, Japan, April 1997

[CIRM93] R. Campbell, N. Islam, D. Raila, P. Madany: *Designing and Implementing Choices: an Object-Oriented System in C++*, Communications of the ACM, vol. 36, pp. 117–126, September 1993

[Cla85] D.D. Clark: *The Structuring of Systems Using Upcalls*, Proceedings of the 10th Symposium on Operating System Principles, 1985

[CMP95] S. Crane, J. Magee, N. Pryce: *Design Patterns for Binding in Distributed Systems*, OOPSLA '95 workshop on Design Patterns for Concurrent, Parallel, and Distributed Object-Oriented Systems, ACM, October 1995

[Cock96] A. Cockburn: *Prioritizing Forces in Software Design*, in [PLoPD2], 1996

[Cock97] A. Cockburn: *Surviving Object-Oriented Projects: A Manager's Guide*, Addison Wesley Object Technology Series, Addison-Wesley, 1997

[Cope92] J.O. Coplien: *Advanced C++ – Programming Styles and Idioms*, Addison-Wesley, 1992

[Cope95] J.O. Coplien: *A Development Process Generative Pattern Language*, in [PLoPD1], 1995

[Cope96] J.O. Coplien: *Software Patterns*, SIGS Books, New York, New York, 1996

[Cope97] J.O. Coplien: *The Column Without a Name: Pattern Languages*, C++ Report, vol. 9, no. 1, pp. 15–21, January 1997

[Cope98] J.O. Coplien: *Multi-Paradigm Design with C++*, Addison-Wesley, 1998

[CoSte91] D.E. Comer, D.L. Stevens: *Internetworking with TCP/IP Volume II: Design, Implementation, and Internals*, Prentice Hall, 1991

[CoSte92] D.E. Comer, D.L. Stevens: *Internetworking with TCP/IP Volume III: Client-Server Programming and Applications*, Prentice Hall, 1992

[CP95] C. Cranor, G. Parulkar: *Design of Universal Continuous Media I/O*, Proceedings of the 5th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '95), pp. 83–86, April 1995

[Cris89] F. Cristian: *Probabilistic Clock Synchronization*, Distributed Computing, vol. 3, pp 146–158, 1989

[CRSS+98] M. Cukier, J. Ren, C. Sabnis, W.H. Sanders, D.E. Bakken, M.E. Berman, D.A. Karr, R.E. Schantz: *AQuA: An Adaptive Architecture that provides Dependable Distributed Objects*, IEEE Symposium on Reliable and Distributed Systems (SRDS), West Lafayette, IN, October 1998

[CW99] J.O. Coplien with B. Woolf: *A Pattern Language for Writers' Workshops*, in

[PLoPD4], 1999

[CY91] P. Coad, E. Yourdon: *Object-Oriented Analysis*, Prentice Hall, 2nd edition, 1991

[DBRD91] R.P. Draves, B.N. Bershad, R.F. Rashid, R.W. Dean: *Using Continuations to Implement Thread Management and Communication in Operating Systems*, Proceedings of the 13th Symposium on Operating System Principles (SOSP), October 1991

[DeBr95] D. DeBruler: *A Generative Pattern Language for Distributed Processing*, in [PLoPD1], 1995

[DeFe99] A.Delarue and E.B. Fernandez: *Extension and Java implementation of the Reactor-Acceptor-Connector pattern combination*, Proceedings of the 1999 Conference on Pattern Languages of Programming, 1999

[Dij68] E.W. Dijkstra: *Cooperating Sequential Processes*, in Programming Languages (F. Genuys, ed.), Academic Press, 1968

[DLF93] D. de Champeaux, D. Lea, P. Faure: *Object-Oriented System Development*, Addison-Wesley, 1993

[Doble96] J. Doble: *Shopper*, in [PLoPD2], 1996

[DP93] P. Druschel, and L.L. Peterson: *Fbufs: A High-Bandwidth Cross-Domain Transfer Facility*, Proceedings of the 14th Symposium on Operating System Principles (SOSP), December 1993

[ECOOP95] W. Olthoff (ed.): *ECOOP '95 – Object-Oriented Programming*, Proceedings of 9th European Conference on Object-Oriented Programming, Åarhus, Denmark, August 1995, Lecture Notes in Computer Science 952, Springer-Verlag, Berlin Heidelberg New York, 1995

[ECOOP97] M. Aksit, S. Matsuoka (eds.): *ECOOP '97 – Object-Oriented Programming*, Proceedings of 11th European Conference on Object-Oriented Programming, Jyväskylä, Finland, June 1997, Lecture Notes in Computer Science 1241, Springer-Verlag, Berlin Heidelberg New York, 1997

[ECOOP98] E. Jul (ed.): *ECOOP '98 – Object-Oriented Programming*, Proceedings of the 12th European Conference on Object-Oriented Programming, Brussels, Belgium, July 1998, Lecture Notes in Computer Science 1445, Springer-Verlag, Berlin Heidelberg New York, 1998

[EGHY99] A. H. Eden, J. Gil, Y. Hirshfeld, A. Yehudai: *Motifs in Object Oriented Architecture*, IEEE Transactions on Software Engineering, 1999

[EKBF+92] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Wecks, D. Williams: *Beyond Multiprocessing... Multithreading the SunOS Kernel*, Proceedings of the Summer USENIX Conference, San Antonio, TX, June 1992

[Eng99] J. Engel: *Programming for the Java Virtual Machine*, Addison-Wesley, 1999

[ENTERA00] Entera: *Internet Content Delivery Techniques*, Entera Inc., `http://www.entera.com/`, 2000

[FoYo99] B. Foote, J. Yoder: *Big Ball of Mud*, in [PLoPD4], 1999

[FBBOR99] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999

[FF97] M. Felleisen, D.P. Friedman: *A Little Java, A Few Patterns*, MIT Press, 1997

[FGMFB97] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee: *Hypertext Transfer Protocol – HTTP/1.1*, Network Working Group, RFC 2068, January 1997

[FGS98] P. Felber, R. Guerraoui, A. Schiper: *The Implementation of a CORBA Object Group Service*, Theory and Practice of Object Systems (TAPOS), vol. 4, no. 2, pp. 93–105, February 1998

[FHA99] E. Freeman, S. Hupfer, K. Arnold: *JavaSpaces Principles, Patterns and Practice*, Addison-Wesley, 1999

[FJ99] M. Fayad, R. Johnson (eds.): *Domain-Specific Application Frameworks: Frameworks Experience by Industry*, John Wiley & Sons, New York, NY, 1999

[FJS99a] M. Fayad, R. Johnson, D.C. Schmidt (eds.): *Object-Oriented Application Frameworks: Problems & Perspectives*, John Wiley & Sons, New York, NY, 1999

[FJS99b] M. Fayad, R. Johnson, D.C. Schmidt (eds.): *Object-Oriented Application Frameworks: Applications & Experiences*, John Wiley & Sons, New York, NY, 1999

[Flex98] Siemens AG, Geschäftsbereich Private Netze: *FlexRouting und BusinessView Produkte*, Folien zur Akquisitionsunterstützung, Siemens AG, 1998

[FMW97] G. Florijn, M. Meijers, P. van Winsen: *Tool support for object-oriented patterns*, in [ECOOP97], 1997

[Fow97a] M. Fowler: *Analysis Patterns*, Addison-Wesley, 1997

[Fow97b] M. Fowler: *UML Distilled*, Addison-Wesley, 1997

[FS97] M. Fayad, R. Johnson, D.C. Schmidt (eds.): *Special Issue on Object-Oriented Application Frameworks*, Communications of the ACM, vol. 40, no. 10, October 1997

[Gab96] R.P. Gabriel: *Patterns of Software*, Oxford University Press, 1996

[GG99] R.P. Gabriel, R. Goldman: *Jini Community Pattern Language*, Proceedings of the 1999 conference on Pattern Languages of Programming, 1999,

http://jerry.cs.uiuc.edu/~plop/plop99/proceedings/

[GJS96] J. Gosling, B. Joy, G. Steele: *The Java Language Specification*, Addison-Wesley, 1996

[GLDW87] R. Gingell, M. Lee, X. Dang, M. Weeks: *Shared Libraries in SunOS*, Proceedings of the Summer 19987 USENIX Technical Conference, 1987

[GoF95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[GP97] G. Somadder, D.C. Petri: *A Pattern Language for Improving the Capacity of Layered Client/Server Systems with Multi-Threaded Servers*, Proceedings of the 2nd European Conference on Pattern Languages of Programming (EuroPLoP'97), pp. 179–189, Irsee, Germany, 1997

[GR93] J. Gray, A. Reuter: *Transaction Processing: Concepts and Techniques*, Morgan Kaufman, 1993

[Grand98] M. Grand: *Patterns in Java*, Volume 1, John Wiley & Sons, 1998

[Grand99] M. Grand: *Patterns in Java*, Volume 2, John Wiley & Sons, 1999

[GR96] B. Goldfedder, L. Rising: *A Training Experience with Patterns*, Communications of the ACM, vol. 39, no. 10, 1996

[GS96] A. Gokhale, D.C. Schmidt: *Measuring the Performance of Communication Middleware on High-Speed Networks*, Proceedings of SIGCOMM '96, Stanford, CA, ACM, pp. 306–317, August 1996

[GS97] R. Guerraoui, A. Schiper: *Software-Based Replication for Fault Tolerance*, IEEE Computer, vol 30. no. 4, pp. 68–74, April 1997

[GS98] A. Gokhale, D.C. Schmidt: *Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems*, Journal on Selected Areas in Communications, special issue on Service Enabling Platforms for Networked Multimedia Systems, vol. 17, no. 9, September 1999

[GZ89] R. Gusella, S. Zatti: *The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3 BSD*, IEEE Transactions on Software Engineering, vol. 15, pp. 847–853, July 1989

[Hal85] R.H. Halstead, Jr.: *Multilisp: A Language for Concurrent Symbolic Computation*, ACM Transactions on Programming Languages and Systems, vol. 7, pp. 501–538, October 1985

[Har96] N.B. Harrison: *Organizational Patterns for Teams*, in [PLoPD2], 1996

[Har97] N.B. Harrison: *Patterns for Logging Diagnostic Messages*, in [PLoPD3], 1997

[Hen98] M. Henning: *Binding, Migration and Scalability in CORBA*, in Communica-

tions of the ACM, special issue on CORBA, ed. K. Seetharaman, vol. 41, no. 10, ACM, October 1998

[Hen00] K. Henney: *Executing Around Sequences*, Proceedings of the 5th European conference on Pattern Languages of Programming, EuroPLoP 2000, Irsee, July 2000

[HJE95] H. Hüni, R. Johnson, R. Engel: *A Framework for Network Protocol Software*, Proceedings of OOPSLA '95, ACM, Austin, TX, October 15–19, 1995

[HK93] Y. Huang, C. Kintala: *Software Implemented Fault Tolerance: Technologies and Experience*, Proceedings of the 23rd International Symposium on Fault-tolerance Computing (FTCS), pp. 2–10, Toulouse, France, June 1993

[HLS97] T. Harrison, D. Levine, D.C. Schmidt: *The Design and Performance of a Real-Time CORBA Event Service*, Proceedings of OOPSLA '97, ACM, Atlanta, GA, October 6–7, 1997.

[HMPT89] N.C. Hutchinson, S. Mishra, L.L. Peterson, V.T. Thomas: *Tools for Implementing Network Protocols*, Software Practice and Experience, vol. 19, no. 9, 1989

[HMS98] J. Hu, S. Mungee, Douglas C. Schmidt: *Principles for Developing and Measuring High-performance Web Servers over ATM*, Proceedings of INFOCOM '98, 1998

[Hoare74] C.A.R. Hoare: *Monitors: An Operating System Structuring Mechanism*, Communications of the ACM, vol. 17, October 1974

[HP91] N.C. Hutchinson, L.L. Peterson: *The x-kernel: An Architecture for Implementing Network Protocols*, IEEE Transactions on Software Engineering, vol. 17, pp. 64–76, January 1991

[HPS99] J.C. Hu, I. Pyarali, D.C. Schmidt: *The Object-Oriented Design and Performance of JAWS: A High-performance Web Server Optimized for High-speed Networks*, Parallel and Distributed Computing Practices Journal, special issue on Distributed Object-Oriented Systems, 1999

[HS98] J.C. Hu, D.C. Schmidt: JAWS: *A Framework for High-performance Web Servers*, in [FJ99], 1999

[HS99a] G.C. Hunt, M.L. Scott: *Intercepting and Instrumenting COM Application*, Proceedings of the 5th Conference on Object-Oriented Technologies and Systems, USENIX, San Diego, CA, May 1999

[HS99b] R. Hanmer, G. Stymfal: *An Input and Output Pattern Language: Lessons from Telecommunications*, in [PLoPD4], 1999

[HSGH99] T. Howes, M.C. Smith, G.S. Good, T.A. Howes: *Understanding and Deploying Ldap Directory Services*, Macmillan Network Architecture and Development Series), Macmillan Technical Publishing, 1999

[HV99] M. Henning, S. Vinoski: *Advanced CORBA Programming with C++*, Addison-Wesley, 1999

[IBM98] IBM Corporation: *San Francisco Patterns*, `http://www-4.ibm.com/software/ad/sanfrancisco/concepts/ibmsf.sf.SFConceptsAndFacilitiesPatterns.html`, 1998

[IEEE95] IEEE: *Scheduling and Load Balancing in Parallel and Distributed Systems*, (eds.) B.A. Shirazi, A.R. Hurson, K.M. Kavi, IEEE Computer Society, 1995

[IEEE96] IEEE: *Threads Extension for Portable Operating Systems*, (Draft 10), February 1996

[IEEE99] IEEE: *Design Patterns in Communications Software Architecture*, L. Rising (ed.), special issue of IEEE Communications Magazine, vol. 37, no. 4, April 1999

[IM96] N. Islam, M.V. Devarakonda: *An Essential Design Pattern for Fault-Tolerant Distributed State Sharing*, Communications of the ACM (CACM), vol. 39, no. 10, pp. 55–74, 1996

[ITUT92] ITU-T: *Principles for a Telecommunications Management Network*, (Recommendation M.3010), 1992

[JACE99] DOC Group at Washington University in St. Louis: *Java ACE, An Object-Oriented Network Programming Toolkit in Java*, Washington University, St. Louis, USA, 1999, `http://www.cs.wustl.edu/~schmidt/JACE.html`

[JBR99] I. Jacobson, G. Booch, J. Rumbaugh: *Unified Software Development Process*, Addison-Wesley, 1999

[JH98] G. James, D. Hillard: *The TAO Of Programming, Book 3 – Design*, `http://www.gnomehome.demon.nl/uddf/pages/zzztao.htm`, 1998

[JK00] P. Jain, M. Kircher: *The Lookup Design Pattern*, Proceedings of the 5th European Conference on Pattern Languages of Programming, EuroPLoP 2000, July 2000

[JML92] R. Johnson, C. McConnell, J.M. Lake: *The RTL System: A Framework for Code Optimization*, in R. Giegerich, S. Graham (eds.): Code Generation – Concepts, Tools, Techniques, pp. 255–274, Springer-Verlag, London, 1991

[John92] R. Johnson: *Documenting Frameworks Using Patterns*, in Proceedings of the 1992 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '92), pp. 63–76, ACM Press, 1992

[John96] R. Johnson: *Transactions and Accounts*, in [PLoPD2], 1996

[John97] R. Johnson: *Frameworks = Patterns + Components*, Communications of the ACM, M. Fayad and D. C. Schmidt (eds.), vol. 40, no. 10, October 1997

[JS97a] P. Jain, D.C. Schmidt: *Experiences Converting a C++ Communication Software Framework to Java*, C++ Report, vol. 9, Issue 1, January 1997

[JS97b] P. Jain, D.C. Schmidt: *Service Configurator: A Pattern for Dynamic Configuration of Services*, Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems, USENIX, June 1997

[JSR51] *New I/O APIs for the Java Platform*, `http://java.sun.com/aboutJava/communityprocess/jsr/jsr_051_ioapis.html`

[JWS98] P. Jain, S. Widoff, D.C. Schmidt: *The Design and Performance of MedJava – Experience Developing Performance-Sensitive Distributed Applications with Java*, IEE/BCS Distributed Systems Engineering Journal, 1998

[Kan92] S. Khanna: *Real-Time Scheduling in SunOS 5.0*, in Proceedings of the 1992 USENIX Winter Conference, USENIX Association, pp. 375–390, 1992

[Kar95] E-A. Karlsson (Ed.): *Software Reuse – A Holistic Approach*, John Wiley & Sons, 1995

[KC97] W. Keller, J. Coldewey: *Accessing Relational Databases*, in [PLoPD3], 1997

[KCC+96] K. Beck, R. Crocker, J.O. Coplien, L. Dominick, G. Meszaros, F. Paulisch, J. Vlissides: *Industrial Experience with Design Patterns*, Proceedings of the International Conference on Software Engineering 1996 (ICSE '96), Berlin, March 1996, also available in [Ris98]

[Ken98] E. Kendall: *Utilizing Patterns and Pattern Languages in Software Engineering Education*, Annals of Software Engineering, vol. 6, Software Engineering Education, 1998

[Kic92] G. Kiczales: *Towards a New Model of Abstraction in Software Engineering*, in A. Yonezawa, B.C. Smith (eds.): Proceedings of the International Workshop on New Models for Software Architecture '92 – Reflection and Meta-Level Architecture, Tokyo, Japan, 1992

[KiLe00] M. Kircher and D.L. Levine: *The XP of TAO – eXtreme Programming of Large, Open-source Frameworks*, in Proceedings of the 1st International Conference on eXtreme Programming and Flexible Processes in Software Engineering, Cagliari, Italy, June 2000

[KLM+97] G. Kiczales. J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, J. Irwin: *Aspect-Oriented Programming*, in [ECOOP97], 1997

[KML92] D. Kafura, M. Mukherji, G. Lavender: *ACT++: A Class Library for Concurrent Programming in C++ using Actors*, Journal of Object-Oriented Programming, pp. 47–56, October 1992

[Kof93] T. Kofler: *Robust Iterators for {ET++}*, Structured Programming, vol. 14, no. 2, pp. 62–85, 1993

[KRL+00] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L.C. Magalhães, R.H. Campbell: *Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB*, in J. Sventek, G. Coulson (ed.): Middleware 2000, Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms, 2000, ACM/IFIP, Lecture Notes in Computer Science, Springer, 2000

[KSL99] F. Kuhns, D.C. Schmidt, D.L. Levine: *The Design and Performance of a Real-Time I/O Subsystem*, Proceedings of the IEEE Real-Time Technology and Applications Symposium, IEEE, Vancouver, British Columbia, Canada, pp. 154–163, June 1999

[KTB98] R.K. Keller, J. Tessier, G. von Bochmann: *A Pattern System for Network Management Interfaces*, Communications of the ACM, vol. 41, no. 9. pp. 86–93, September 1998

[Lak95] J. Lakos: *Large-Scale Software Development with C++*, Addison-Wesley, 1995

[Lan98] M. Lange: *Time Patterns*, Proceedings of the 1998 European Conference on Pattern Languages of Programming, July 1998

[LBG97] K. Lano, J. Bicarregui, and S Goldsack: *Formalising Design Patterns*, in 1st BCS-FACS Northern Formal Methods Workshop, Electronic Workshops in Computer Science, Springer-Verlag, 1997

[LC87] M.A. Linton, P.R. Calder: *The Design and Implementation of InterViews*, Proceedings of the USENIX C++ Workshop, November 1987

[Lea94] Doug Lea: *Design Patterns for Avionics Control Systems*, DSSA ADAGE Technical Report ADAGE-OSW-94-01, Loral Federal Systems, 1994

[Lea99a] D. Lea: *Concurrent Programming in Java, Design Principles and Patterns*, 2nd edition, Addison-Wesley, 1999

[Lea99b] D. Lea: *Software Design*, State University Of New York at Oswego, class no. CSC 480, State University Of New York at Oswego, Oswego, 1999

[Lew95] B. Lewis: *Threads Primer: A Guide to Multithreaded Programming*, Prentice Hall, 1995

[LGS99] D.L. Levine, C.D. Gill, D.C. Schmidt: *Object Lifetime Manager – A Complementary Pattern for Controlling Object Creation and Destruction*, C++ Report, vol. 12, Number 1, January 2000,
`http://www.cs.wustl.edu/~levine/research/ObjMan.ps.gz`

[LG00] D.L. Levine, C.D. Gill: *Object-Oriented Software Development Laboratory*, Washington University in St. Louis, Class no. CS E61 342S, Washington University, St. Louis, 2000,
`http://www.cs.wustl.edu/~levine/courses/cs342`

[LK95] R.G. Lavender, D.G. Kafura: *A Polymorphic Future and First-Class Function*

Type for Concurrent Object-Oriented Programming in C++, `http://www.cs.utexas.edu/users/lavender/papers/future.ps`, 1995

[LS88] B. Liskov, L. Shrira: *Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems*, Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, pp. 260–267, June 1988

[LS98] A. Lauder, S. Kent: *Precise Visual Specification of Design Patterns*, in [ECOOP98], 1998

[LT93] D.L. Levine, R.N. Taylor: *Metric-Driven Re-engineering for Static Concurrency Analysis*, Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '93), ACM press, 1993

[LY99] T. Lindholm, F. Yellin: *The Java Virtual Machine Specification*, 2nd edition, Addison-Wesley, 1999

[Maf96] S. Maffeis: *The Object Group Design Pattern*, Proceedings of the 1996 USENIX Conference on Object-Oriented Technologies, USENIX, Toronto, Canada, June 1996

[MaHa99] V. Matena, M Hapner: *Enterprise JavaBeans*, Version 1.1, Sun Microsystems, 1999

[Mar95] R.C. Martin: *Designing Object-Oriented C++ Applications Using the Booch Method*, Prentice Hall, 1995

[MB91] J.C. Mogul, A. Borg: *The Effects of Context Switches on Cache Performance*, Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), ACM, Santa Clara, CA, April 1991

[MBKQ96] M.K. McKusick, K. Bostic, M.J. Karels, J.S. Quarterman: *The Design and Implementation of the 4.4 BSD Operating System*, Addison-Wesley 1996

[MD91] P.E. McKenney, K.F. Dove: *Efficient Demultiplexing of Incoming TCP Packets*, Sequent Computer Systems, Inc., SQN TR92-01, December 1991

[McK95] P.E. McKenney: *Selecting Locking Designs for Parallel Programs*, in [PLoPD2], 1996

[MDS96] *Microsoft Developers Studio, Version 4.2 – Software Development Kit*, 1996

[Mey95] S. Meyers: *More Effective C++ – 35 New Ways to Improve Your Programs and Designs*, Addison-Wesley, 1995

[Mey97] B. Meyer: *Object-Oriented Software Construction*, 2nd edition, Prentice Hall, Englewood Cliffs, NJ, 1997

[Mey98] S. Meyers: *Effective C++*, 2nd edition, Addison-Wesley, 1998

[Mes95] G. Meszaros: *Half Object plus Protocol*, in [PLoPD1], 1995

[Mes96] G. Meszaros: *A Pattern Language for Improving the Capacity of Reactive Systems*, in [PLoPD2], 1996

[MGG00] P. Merle, C. Gransart, J.M. Geib: *Using and Implementing CORBA Objects with CorbaScript*, in J.P. Bashoun, T. Baba, J.P. Briot, A. Yonezawa (eds.): Object-Oriented Parallel and Distributed Programming, Editions Hermes, Paris, France, January 2000

[Mic00] A. Michail: *Data Mining Library Reuse Patterns using Generalized Association Rules*, Proceedings of the 22nd International Conference on Software Engineering, 2000

[Mik98] T. Mikkonen: *Formalizing Design Patterns*, Proceedings of the 1998 International Conference on Software Engineering, pp. 115–124, IEEE Computer Society Press, 1998

[Mil88] D. Mills: *The Network Time Protocol*, University of Delaware, Network, 1988

[MK97] A. Mester, H. Krumm: *Formal behavioural patterns for the tool-assisted design of distributed applications*, IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS 97), Cottbus, Germany, Sep/Oct 1997, Chapman & Hall, 1997

[MM97] R.C. Malveau, T.J. Mowbray: *CORBA Design Patterns*, John Wiley & Sons, 1997

[MMN99] L.E. Moser, P.M. Melliar-Smith, P. Narasimhan: *A Fault Tolerance Framework for CORBA*, International Symposium on Fault Tolerant Computing, Madison, WI, June 1999

[Mog95] J.C. Mogul: *The Case for Persistent-connection HTTP*, Proceedings of the ACM SIGCOMM'95 Conference in Computer Communication Review, ACM press, pp. 299–314, 1995

[MWY91] S. Matsuoka, K. Wakita, A. Yonezawa: *Analysis of Inheritance Anomaly in Concurrent Object-Oriented Languages*, OOPS Messenger, 1991

[Mue93] F. Mueller: *A Library Implementation of POSIX Threads Under UNIX*, Proceedings of the Winter USENIX Conference, San Diego, CA, USA, 1993

[Mue96] H. Mueller: *Patterns for Handling Exception Handling Successfully*, C++ Report vol. 8, January 1996

[Naka00] T. Nakajima: *Dynamic Transport Protocol Selection in a CORBA System*, The 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Newport Beach, California, USA, March 15–17 2000

[NDW93] J. Neider, T. Davis, M. Woo: *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1*, Addison-Wesley, 1993

[NLN+99] S. Nimmagadda, C. Liyanaarchchi, D. Niehaus, A. Gopinath, A. Kaushal: *Performance Patterns: Automated Scenario Based ORB Performance Evaluation*, Proceedings of the 5th Conference on Object-Oriented Technologies and Systems, USENIX, San Diego, CA, May 1999

[NMM99] P. Narasimhan, L.E. Moser, P.M. Melliar-Smith: *Using Interceptors to Enhance CORBA*, IEEE Computer, vol. 32, no. 7, pp. 64–68, July 1999

[NMM00] P. Narasimhan, L.E. Moser, P.M. Melliar-Smith: *Patterns for Building Transparent Reliable Middleware*, Theory and Practice of Object Systems, vol. 6, 2000

[Nuy96] D. Nguyen: *Design Patterns for Data Structures*, `http://csis.pace.edu/~bergin/papers/nguyen/DesPat4DatStruct.html`, 1996

[NW98] J. Noble, C. Weir: *Proceedings of the Memory Preservation Society*, Proceedings of the 1998 European Conference on Pattern Languages of Programming, July 1998

[MTOOL99] microTOOL: *objectiF – For the Development of Object-Oriented and Component-Based Software Systems*, microTOOL GmbH, `http://www.microtool.de`, 1999

[OHE96] R. Orfali, D. Harkey, J. Edwards: *The Essential Distributed Objects Survival Guide*, Wiley & Sons, 1996

[OMG97a] Object Management Group: *CORBA Services – Naming Service*, TC Document formal/97-07-12, July 1997

[OMG97b] Object Management Group: *CORBA Services – Transactions Service*, TC Document formal/97-12-17, 1997

[OMG97c] Object Management Group: *CORBA Time Service Specification*, OMG TC Document orbos/97-12-21, December 1997

[OMG98a] Object Management Group: *CORBA Messaging Specification*, OMG TC Document orbos/98-05-05, May 1998

[OMG98b] Object Management Group: *CORBA Services: Common Object Services Specification Updated Version: December 1998, Chapter 16: The Trading Object Service Specification*, formal/98-07-05, 1998

[OMG98c] Object Management Group: *The Common Object Request Broker: Architecture and Specification*, 2.3 ed., December 1998

[OMG98d] Object Management Group: *CORBA Security Service v1.2 Specification*, OMG TC Document formal/98-12-17, December 1998

[OMG99a] Object Management Group: *CORBA Components Final Submission*, OMG TC Document orbos/99-02-05, February 1999

[OMG99b] Object Management Group: *Real-Time CORBA Joint Revised Submission*, OMG TC Document orbos/99-02-12, February 1999

[OMG99c] Object Management Group: *Notification Service Specification*, OMG TC Document telecom/99-07-01, July 1999

[OMG99d] Object Management Group: *minimumCORBA Joint Revised Submission*, OMG TC Document orbos/98-08-04, August1999

[OMG99e] Object Management Group: *Persistent State Service 2.0 Joint Revised Submission*, orbos/99-07-07, August 1999

[OMG99f] Object Management Group: *Portable Interceptors*, OMG TC Document orbos/99-12-02, December 1999

[OMG99g] Object Management Group: *Fault Tolerant CORBA Joint Revised Submission*, OMG TC Document orbos/99-12-08, December 1999

[Opd92] W.F. Opdyke: *Refactoring Object-Oriented Frameworks*, PhD Thesis, University of Illinois at Urbana Champaign, 1992

[OS99] O'Reilly On-line Book: *Open Sources: Voices from the Open Source Revolution*, `http://www.oreilly.com/catalog/opensources/book/toc.html`, O'Reilly, 1999

[OSSL00] OpenSSL: *An Open Source toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1)*, `http://www.openssl.org/`, 2000

[PC00] M.P. Plezbert, R.K. Cytron: *Recognition and Verification of Design Patterns*, Technical Report WUCS-00-01, Washington University in St. Louis, January 2000

[Pet95] C. Petzold: *Programming Windows 95*, Microsoft Press, 1995

[PHS96] I. Pyarali, T.H. Harrison, D.C. Schmidt: *Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging*, USENIX Computing Systems, vol. 9, November/December 1996

[PK98] L. Prechelt, C. Krämer: *Functionality versus Practicality: Employing Existing Tools for Recovering Structural Design Patterns*, Journal of Universal Computer Science, vol. 4, no. 12, pp. 866–882, December 1998

[PLoPD1] J.O. Coplien, D.C. Schmidt (eds.): *Pattern Languages of Program Design*, Addison-Wesley, 1995 (a book publishing the reviewed Proceedings of the First International Conference on Pattern Languages of Programming, Monticello, Illinois, 1994)

[PLoPD2] J.O. Coplien, N. Kerth, J. Vlissides (eds.): *Pattern Languages of Program Design 2*, Addison-Wesley, 1996 (a book publishing the reviewed Proceedings of the Second International Conference on Pattern Languages of

Programming, Monticello, Illinois, 1995)

[PLoPD3] R.C. Martin, D. Riehle, F. Buschmann (eds.): *Pattern Languages of Program Design 3*, Addison-Wesley, 1997 (a book publishing selected papers from the Third International Conference on Pattern Languages of Programming, Monticello, Illinois, USA, 1996, the First European Conference on Pattern Languages of Programming, Irsee, Bavaria, Germany, 1996, and the Telecommunication Pattern Workshop at OOPSLA '96, San Jose, California, USA, 1996)

[PLoPD4] N. Harrison, B. Foote, H. Rohnert (eds.): *Pattern Languages of Program Design 4*, Addison-Wesley, 1999 (a book publishing selected papers from the Fourth and Fifth International Conference on Pattern Languages of Programming, Monticello, Illinois, USA, 1997 and 1998, and the Second and Third European Conference on Pattern Languages of Programming, Irsee, Bavaria, Germany, 1997 and 1998)

[POSA1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: *Pattern-Oriented Software Architecture – A System of Patterns*, John Wiley & Sons, 1996

[POSIX95] *Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application: Program Interface (API) [C Language]*, 1995

[PPP00] *The Pedagogical Patterns Project*, `http://www-lifia.info.unlp.edu.ar/ppp/`, 2000

[PPR] *The Portland Pattern Repository*, `http://www.c2.com`

[PR90] D.L. Presotto, D.M. Ritchie: *Interprocess Communication in the Ninth Edition UNIX System*, UNIX Research System Papers, Tenth Edition, vol. 2, no. 8, pp. 523–530, 1990

[PQ00] V. N. Padmanabhan, L. Qiu: *The Content and Access Dynamics of A Busy Web Site: Findings and Interpretations*, Proceedings of the ACM SIGCOMM Conference, Stockholm, Sweden, August/September, 2000

[Pree95] W. Pree: *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995

[Preiss98] B.R. Preiss: *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*, John Wiley & Sons, 1998

[Pro99] J. Prosise: *Programming Windows With MFC*, 2nd edition, Microsoft Press, 1999

[PRS+99] I. Pyarali, C. O'Ryan, D.C. Schmidt, N. Wang, V. Kachroo, A. Gokhale: *Applying Optimization Patterns to the Design of Real-Time ORBs*, Proceedings of the 5th conference on Object-Oriented Technologies and Systems, San Diego, CA, USENIX, 1999

[PRS00] I. Pyarali, C. O'Ryan, D.C. Schmidt: *A Pattern Language for Efficient, Predictable, Scalable, and Flexible Dispatching Mechanisms for Distributed Object Computing Middleware*, Proceedings of the International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), IEEE/IFIP Newport Beach, CA, March 2000

[Pry99] N. Pryce: *Abstract Session*, in [PLoPD4], 1999

[PSJ00] D.C. Petriu, C. Shousha, A. Jalnapurkar: *Architecture-Based Performance Analysis Applied to a Telecommunication System*, accepted for publication in IEEE Transactions on Software Engineering, special issue on Software and Performance, 2000

[PSK+97] G. Parulkar, D.C. Schmidt, E. Kraemer, J. Turner, A. Kantawala: *An Architecture for Monitoring, Visualizing and Control of Gigabit Networks*, IEEE Network vol. 11, Sept. 1997

[PUPT97] L. Prechelt, B. Unger, M. Philippsen, W.F. Tichy: *Two Controlled Experiments Assessing the Usefulness of Design Pattern Information During Program Maintenance*, Empirical Software Engineering – An International Journal, December 1997

[Rago93] S. Rago: *UNIX System V Network Programming*, Addison-Wesley, 1993

[RBGM00] D. Riehle, R. Brudermann, T. Gross, K.U. Mätzel: *Pattern Density and Role Modeling of an Object Transport Service*, ACM Computing Surveys, March 2000.

[RBPEL91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: *Object-Oriented Modeling and Design*, Prentice Hall, 1991

[RBV99] M. Robinson, K. Brown, P.A. Vorobiev: *Swing*, Manning Publications Company, 1999

[Ree92] T. Reenskaug: *Intermediate Smalltalk, Practical Design and Implementation*, Tutorial, TOOLS Europe '92, Dortmund, 1992

[RG98] D. Riehle, T. Gross: *Role Model Based Framework Design and Integration*, Proceedings of the 1998 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98), pp. 117–133, ACM Press, 1998

[Ric97] J. Richter: *Advanced Windows*, 3rd edition, Microsoft Press, 1997

[Rie96] D. Riehle: *The Event Notification Pattern – Integrating Implicit Invocation with Object-Orientation*, Theory and Practice of Object Systems vol. 2, no. 1, pp. 43–52, John Wiley & Sons, 1996

[Rie97] D. Riehle: *Composite Design Patterns*, Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications

(OOPSLA '97). ACM Press, 1997

[Ris98] L. Rising (ed.): *The Patterns Handbook*, SIGS Publications, Cambridge University Press, 1998

[Ris00a] L. Rising: *The Pattern Almanac 2000*, Addison-Wesley, 2000

[Ris00b] L. Rising: *Design Patterns in Communications Software*, Cambridge University Press, 2000

[Rit84] D. Ritchie: *A Stream Input-Output System*, AT&T Bell Labs Technical Journal, vol 63 pp. 311–324, October 1984

[RKF92] W. Rosenberry, D. Kenney, G. Fischer: *Understanding DCE*, O'Reilly and Associates, Inc., 1992

[RKSOP00] C. O'Ryan, F. Kuhns, Douglas C. Schmidt, O. Othman, J. Parsons: *The Design and Performance of a Pluggable Protocols Framework for Real-Time Distributed Object Computing Middleware*, J. Sventek, G. Coulson (eds.): Middleware 2000, Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms, 2000, ACM/IFIP, Lecture Notes in Computer Science, Springer, 2000

[RSB+97] D. Riehle, W. Siberski, B. Bäumer, D. Megert, B. Fletcher: *Serializer*, in [PLoPD3], 1997

[San98] R.E. Sanders: *ODBC 3.5 Developer's Guide*, McGraw-Hill Series on Data Warehousing and Data, 1998

[SC96] A. Sane, R. Campbell: *Resource Exchanger: A Behavioral Pattern for Low-Overhead Concurrent Resource Management*, in [PLoPD2], 1996

[SC99] D.C. Schmidt, C. Cleeland: *Applying Patterns to Develop Extensible ORB Middleware*, IEEE Communications Magazine, special issue on Design Patterns, April 1999

[Sch92] D.C. Schmidt: *IPC_SAP: An Object-Oriented Interface to Interprocess Communication Services*, C++ Report vol. 4, November/December 1992

[Sch94] D.C. Schmidt: *A Domain Analysis of Network Daemon Design Dimensions*, C++ Report, SIGS, vol. 6, no. 3, pp. 1–12, March/April 1994

[Sch95] D.C. Schmidt: *An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit*, Technical Report WUCS-95-31, Washington University, St. Louis, September 1995,
`http://www.cs.wustl.edu/~schmidt/Concurrency.ps.gz`

[Sch96] D.C. Schmidt: *A Family of Design Patterns for Application-level Gateways*, The Theory and Practice of Object Systems, special issue on Patterns and Pattern Languages, vol. 2, no. 1, 1996

[Sch97] D.C. Schmidt: *Applying Design Patterns and Frameworks to Develop Object-Oriented Communication Software*, Handbook of Programming Languages, Peter Salus (ed.), MacMillan Computer Publishing, 1997

[Sch98a] D.C. Schmidt: *GPERF: A Perfect Hash Function Generator*, C++ Report, vol. 10, no. 10, November/December 1998

[Sch98b] D.C. Schmidt: *Evaluating Architectures for Multi-threaded CORBA Object Request Brokers*, Communications of the ACM, special issue on CORBA, Krishnan Seetharaman (ed.), vol 41., no. 10, ACM, October 1998

[Sch99] D.C. Schmidt: *Developing Object-Oriented Software with Patterns and Frameworks*, Washington University in St. Louis, Class no. CS 242, Washington University, St. Louis, 1999, `http://www.cs.wustl.edu/~schmidt/courses.html`

[SchSt95] D.C. Schmidt, P. Stephenson: *Experiences Using Design Patterns to Evolve System Software Across Diverse OS Platforms*, in [ECOOP95], 1995

[SchSu93] D.C. Schmidt, T. Suda: *Transport System Architecture Services for High-Performance Communications Systems*, IEEE Journal on Selected Areas in Communication, vol. 11, no. 4, pp. 489–506, IEEE, May 1993

[SchSu94] D.C. Schmidt, T. Suda: *An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems*, IEE/BCS Distributed Systems Engineering Journal, vol. 2, pp. 280–293, December 1994

[SchSu95] D.C. Schmidt, T. Suda: *Measuring the Performance of Parallel Message-Based Process Architectures*, Proceedings of the Conference on Computer Communications (INFOCOM), pp. 624–633, IEEE, April 1995

[SeSch70] A. Seidelmann, A. Schwarzenegger: *Hercules in New York*, RAF Industries, 1970

[SFJ96] D.C. Schmidt, M. Fayad. R.E. Johnson: *Software Patterns*, Communications of the ACM (CACM), vol. 39, no. 10, pp. 36–39, 1996

[SG96] M. Shaw, D. Garlan: *Software Architecture – Perspectives of an Emerging Discipline*, Prentice Hall, 1996

[SGWSM94] B. Selic, G.Gullekson, P.T. Ward, B. Selic, J. McGee: *Real-Time Object-Oriented Modeling*, John Wiley & Sons, 1994

[Shap93] M. Shapiro: *Flexible Bindings for Fine-Grain, Distributed Objects*, INRIA, Rapport de Recherche INRIA 2007, August 1993

[SHS95] D.C. Schmidt, T.H. Harrison, E. Al-Shaer, *Object-Oriented Components for High-Speed Networking Programming*, Proceedings of the 1st Conference on Object Technologies and Systems, USENIX, Monterey, CA, June 1995

[SKT96] J.D. Salehi, J.F. Kurose, D. Towsley: *The Effectiveness of Affinity-Based*

Scheduling in Multiprocessor Networks, IEEE INFOCOM, IEEE Computer Society Press, March, 1996

[SM88] S. Shlaer, S.J. Mellor: *Object-Oriented Systems Analysis – Modeling the World In Data*, Yourdon Press, Prentice Hall, 1988

[SMB96] F. Shull, W. Melo, V. Basili: *An Inductive Method for Discovering Design Patterns from Object-Oriented Software Systems*, University of Maryland CS-TR-3597, 1996

[SLM98] D.C. Schmidt, D.L. Levine, S. Mungee: *The Design and Performance of Real-Time Object Request Brokers*, Computer Communications, vol. 21, no. 4, pp. 294–324, Elsevier, April 1998

[SMFG00] D.C. Schmidt, S. Mungee, S. Flores-Gaitan, A. Gokhale: *Software Architectures for Reducing Priority Inversion and Non-Determinism in Real-Time Object Request Brokers*, Journal of Real-Time Systems, special issue on Real-Time Computing in the Age of the Web and the Internet, Ed.: A. Stoyen, Kluwer, 2000

[Sol98] D.A. Solomon: *Inside Windows NT*, 2nd edition, Microsoft Press, 1998

[Som97] P. Sommerlad: *Manager*, in [PLoPD3], 1997

[SPM98] A.R Silva, J.D. Pereira, J.A. Marques: *Object Recovery*, in [PLoPD3], 1998

[SPM99] A.R Silva, J.D. Pereira, J.A. Marques: *Object Synchronizer*, in [PLoPD4], 1998

[SRPKB00] D.C. Schmidt, C. O'Ryan, I. Pyarali, M. Kircher, and F. Buschmann, *Leader/Followers: A Design Pattern for Efficient Multi-Threaded Event Demultiplexing and Dispatching*, Proceedings of the 7th Pattern Language of Programming Conference, PLoP, Allerton Park, Illinois, USA, August 2000

[SRZ99] M. Schütze, J.P. Riegel, G. Zimmermann: *PSiGene – A Pattern-Based Component Generator for Building Simulation*, Theory and Practice of Object Systems (TAPOS), vol. 5, no. 2, 1999

[SS99] D. Sheresh, B. Sheresh: *Understanding Directory Services*, New Riders Publishing, 1999

[Stal00] M. Stal: *The Activator Design Pattern*, `http://www.posa.uci.edu/`, 2000

[Ste93] W.R. Stevens: *TCP/IP Illustrated, Volume 1*, Addison-Wesley, 1993

[Ste96] W.R. Stevens: *TCP/IP Illustrated, Volume 3*, Addison-Wesley, 1996

[Ste98] W.R. Stevens: *Unix Network Programming, Volume1: Networking APIs: Sockets and XTI*, 2nd edition, Prentice Hall, 1998

[Ste99] W.R. Stevens: *Unix Network Programming, Volume 2: Interprocess Communications*, 2nd edition, Prentice Hall, 1999

[Str97] B. Stroustrup: *The C++ Programming Language*, 3rd edition, Addison-Wesley 1997

[Sun88] Sun Microsystems: *Remote Procedure Call Protocol Specification*, Sun Microsystems, Inc., RFC-1057, June 1988

[Sun99a] Sun Microsystems: *The Jini Architecture Specification (AR)*, Version 1.0.1, `http://www.sun.com/jini/specs/jini101.html`, November 1999

[Sun99b] Sun Microsystems: *The Jini Distributed Event Specification (EV)*, Version 1.0.1, `http://www.sun.com/jini/specs/event101.html`, November 1999

[Sun00a] Sun Microsystems: *Java 2 Platform, Standard Edition Documentation*, `http://java.sun.com/products/jdk/1.3/docs/index.html`, May 2000

[Sun00b] Sun Microsystems: *Java 2 Platform, Sun Community Source Licensing*, `http://www.sun.com/software/communitysource/java2/`, May 2000

[SV96a] D.C. Schmidt, S. Vinoski: *Comparing Alternative Programming Techniques for Multi-Threaded CORBA Servers: Thread-per-Request*, C++ Report, vol. 8, no. 2, February 1996

[SV96b] D.C. Schmidt, S. Vinoski: *Comparing Alternative Programming Techniques for Multi-Threaded CORBA Servers: Thread Pool*, C++ Report, vol. 8, no. 4, February 1996

[SV96c] D.C. Schmidt, S. Vinoski: *Comparing Alternative Programming Techniques for Multi-Threaded CORBA Servers: Thread-per-Object*, C++ Report, vol. 8, no. 7, July 1996

[SV98a] D.C. Schmidt, S. Vinoski: *Using the Portable Object Adapter for Transient and Persistent CORBA Objects*, C++ Report, vol. 10, no. 4, April 1998

[SV98b] D.C. Schmidt, S. Vinoski: *Introduction to CORBA Messaging*, C++ Report, vol. 10, November/December 1998

[SW94] W.R. Stevens, G Wright: *TCP/IP Illustrated, Volume 2*, Addison-Wesley, 1994

[Tan92] A.S. Tanenbaum: *Modern Operating Systems*, Prentice Hall, 1992

[Tan95] A.S. Tanenbaum: *Distributed Operating Systems*, Prentice Hall, 1995

[Tichy98] W.F. Tichy: *A Catalogue of General-Purpose Design Patterns*, Proceedings of the 23rd conference on the Technology of Object-Oriented Languages and Systems (TOOLS 23), IEEE Computer Society, 1998

[TM00] W.F Tichy, M. Müller: *Ausgewählte Kapitel der Softwaretechnik*, Universität Karlsruhe, LV no. 24646, Universität Karlsruhe, 2000

[ToSi89] C. Tomlinson, V. Singh: *Inheritance and Synchronization with Enabled-Sets*, OOPSLA '89 Conference Proceedings, pp. 103–112, October 1998

[Tow99] D. Towell: *Display Maintenance: A Pattern Language*, in [PLoPD4], 1999

[U2] U2: *even BETTER than the REAL THING*, Island Records Ltd., 1991

[VaLa97] G. Varghese, T. Lauck: *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility*, IEEE Transactions on Networking, December 1997

[Vin98] S. Vinoski: *New Features for CORBA 3.0*, Communications of the ACM, vol. 41, no. 10, pp. 44–52, October 1998

[Vlis98a] J. Vlissides: *Pattern Hatching – Design Patterns Applied*, Addison-Wesley, 1998

[Vlis98b] J. Vlissides: *Composite Design Patterns (They Aren't What You Think)*, C++ Report, June 1998

[Vlis98c] J. Vlissides: *Pluggable Factory, Pt. 1*, C++ Report, November/December 1998

[Wal00] E. Wallingford: *The Elementary Patterns Home Page*, `http://www.cs.uni.edu/~wallingf/patterns/elementary/`, 2000

[WHO91] W. Wilson Ho, R.A. Olsson: *An Approach to Genuine Dynamic Linking*, Software: Practice and Experience, vol. 21, no. 4, pp. 375–390, April 1991

[WRW96] A. Wollrath, R. Riggs, and J. Waldo: *A Distributed Object Model for the Java System*, in Proceedings of the 1996 USENIX Conference on Object-Oriented Technologies (COOTS), Toronto, Canada, 1996

[WS95] G.W Wright, W.R. Stevens: *TCP/IP Illustrated, Volume 2*, Addison-Wesley, 1995

[WSV99] N. Wang, D.C. Schmidt, S, Vinoski: *Collocation Optimizations for CORBA*, C++ Report, vol. 11, no. 9, October 1999

[W3C98] W3C HTTP-NG Working Group: *W3C HTTP-NG Protocol*, www.w3.org/Protocols/HTTP-NG, 1998

[YB99] J. Yoder, J. Barcalow: *Architectural Patterns for Enabling Application Security*, in [PLoPD4], 1999

[ZBS97] J. A. Zinky, D.E. Bakken, R. Schantz: *Architectural Support for Quality of Service for CORBA Objects*, Theory and Practice of Object Systems, vol. 3, no. 1, John Wiley & Sons, 1997

[Zweig90] J.M. Zweig: *The Conduit: a Communication Abstraction in C++*, Proceedings of the 2nd USENIC C++ Conference, pp. 191–203, USENIX Association, April 1990

索　引

A

Abstract Component（抽象组件），545
Abstraction（抽象），52, 57, 64, 70, 73, 107, 117, 503
  Bloating of（膨胀），72
  Cohesion of（内聚），53
  Level of（层次），138
  Overdose of（过度），72
Acceptor（接受器），63, 97, 197, 211, 243, 287, 290, 298, 321, 507, 509, 513-515
  Behavior of（的行为），293-296
  Customization of（的定制），301, 303
    using Parameterized Types（用参数化类型），301, 304, 312
    using Polymorphism（用多态），301, 304, 312
  Deletion of（的删除），305
  Factory（工厂），289
  Implementation of（的实现），303
  Initialization of（的初始化），303
  Instantiation of（的实例化），292, 311
  Interface of（的接口），289, 300, 304
    Connection Completion Method（连接完成方法），289
    Connection Initialization Method（连接初始化方法），289, 293, 300
    Hook Method（钩子方法），301, 304, 312
    Template Method（模板方法），301, 304, 312
  Notification by Dispatcher（分配器的通知），291, 293
  Registration with Dispatcher（向分配器注册），291, 293, 303
  Structure of（的结构），289
ACE。参见ADAPTIVE Communication Environment
ACT。参见Asynchronous Completion Token
Activation List（激活表），371, 373, 378, 381, 392, 393, 400, 440, 516
  Behavior of（的行为），374
  Implementation of（的实现），379
  Iterator for（迭代器），379
  Monitoring of（监视），375
  State of（的状态），372
  Structure of（的结构），372
Active Demultiplexing（主动多路分解），269
Active Object（主动对象），371, 422, 545
  Distributed（分布式的），393
  Thread Pool（线程池），254, 393, 394, 440, 443
Ada，57, 73, 83, 322
Adaptation（适应、适配），73
Adapter（适配器），187
ADAPTIVE Communication Environment（自适应通信环境），xvii, 18, 28, 70, 86, 89, 104, 209, 255, 303, 312, 320, 330, 342, 349, 360, 394, 419, 441, 470, 502, 520
Affordability（可承受的），5, 6
Agility（灵活），5
Algorithm（算法），41
  Family（系列），41
  Skeleton（框架），138
  Variation（变化），138
Algorithm Pattern（算法模式），531
Amoeba（阿米巴/变形虫/又指一类计算机病毒），439
Analysis Pattern（分析模式），528
Anti-Pattern（反模式），529
API。参见Application Programming Interface
Application。参见Software System
Application Domain（应用领域），150, 538, 549
  Accounting（会计），535
  Aerospace Control（宇航控制），xiii, 538
  Avionics Mission Computing（航空任务计算），423, 534
  Billing（编制账单），535
  Call Center Management（呼叫中心管理），210, 394
  Corporate Finance（企业财政），528
  Data Communication（数据通信），xiii
  Database Search Engine（数据库搜索引擎），24
  e-Commerce（电子商务），261, 273, 538
  Financial Services（金融服务），xiii
  Health Care（卫生健康），528
  Home Automation（家庭自动化），535
  Industrial Process Automation（工业过程自动化），xiii, 44, 70, 534
  Information Provision（信息提供），535
  Inventory（财产目录编制），535
  Medical Diagnostics（医疗诊断），xiii, 535
  Medical Imaging（医疗影像），24, 210, 281, 394, 538
  Mobile Agent（移动代理），535

Network Management（网络管理），532
Office Automation（办公自动化），535
On-line Banking（在线银行），535
Payroll（工资支付），535
Personal Assistant（个人助理），535
Scientific Computing（科学计算），xiii
Simulation（模拟），535
Telecommunication（电信），xiii, xiv, 141, 168, 423, 529, 535
Teleradiology（电子放射医学），535
Transaction Processing（事务处理），44, 447, 466
Ubiquitous Computing（无所不在的计算），535
Video（视频），535
Application Framework（应用程序框架），xiv, 19, 45, 70, 107, 109, 110, 112, 119, 123, 133, 136, 141, 209, 394, 444, 508, 514, 528, 532, 534, 546, 550
Architecture of（的体系结构），138
Behavior of（的行为），111, 115, 117, 121, 514
Black-box（黑盒），110, 394
Blocking of（的模块化），137
Component-based（基于组件的），394
Control of Behavior（行为控制），118, 121, 128, 136, 138, 140
Evolution of（演化），45, 110
Functionality of（的功能特性），110
Interface of（的接口），111
Introspection of（自省），118, 121, 128, 140
Lightweight（轻量级），138
Service Integration with（与……的服务集成），110, 111, 139
Service Registration with（向……的服务注册），111
State of（的状态），111, 121, 514
State Transition of（的状态转移），118, 138
Application Programming Interface（应用程序编程接口，应用编程接口），xiv, 3, 8, 9, 16, 18, 47, 57, 72, 73, 232, 349, 432, 456, 491, 517, 546
Data of（的数据），28, 51, 517
Encapsulation of（的封装），27, 28
For Network Programming（用于网络编程的），458
Function of（的功能），28, 51, 517
Function-based（基于功能的），50
Low-level（低层），47, 53, 71, 74
Non-object-oriented（非面向对象的），28, 44, 47, 49, 51, 74
Platform-Specific（针对具体平台的/与平台有关的/平台特定的），49, 55, 69
Relationships between（之间的关系），53
Application Server（应用服务器），110, 112, 132
Application Service。参见Service
Architectural Pattern（体系结构模式），xxvi, 5, 440, 506, 508, 510, 525, 537, 546
Architectural Style（体系结构风格），508
Aspect-Oriented Programming（面向特征的编程），136, 420
Associative Array（相关数组），546
Asynchronous Completion Token（异步完成标记），232, 237, 252, 264, 282, 283, 517
Behavior of（的行为），266
Chain of（的链），277
Collection of（的集合），265
Completion Handler Identification Information within（内的完成处理程序标识信息），265
Creation of（的建立），245, 264, 266, 270
Demultiplexing of（的多路分解），267, 268
Implementation of（的实现），267-273
Interface of（的接口），268
Hook Method（钩子方法），268, 271
Location of（的位置），269
Meaningful（……的含义），267
Non-Opaque（透明的），279
Opaque（不透明的），267
Passing of（的传递），266, 267, 270
Explicit（显式传递），270
Implicit（隐式传递），270
Piggy-backing of（的捎带确认(法)），252
Processing Information within（在……内的处理信息），232, 265
Release of（的释放），269, 270, 283
Representation of（的表示），267
Index-based（基于索引），268, 269
Object Reference Based（基于对象引用），268
Pointer-based（基于指针），267
Result of（的结果），271
Structure of（的结构），265
Usage of（的用法），270
Asynchronous Event Demultiplexer（异步事件多路分解器），218, 221, 224, 233, 236, 239, 245, 513
Concurrent（并发的），252
Demultiplexing Strategy of（多路分解的策略），238
FIFO Demultiplexng（FIFO多路分解），238
Selective Demultiplexing（选择型多路分解），238

Selection of（的选择），237
Shared（共享的），252
Type of（的类型），237
aio_suspend()（(库函数名)），237
GetQueuedCompletionStatus()（库函数名），38, 221, 237
WaitForMultipleObjects()（库函数名），237
Asynchronous Operation（异步操作），36, 37, 177, 216, 217, 221, 226, 232, 251, 259, 261, 264, 513, 517
Behavior of（的行为），223, 266
Cancellation of（的取消），227, 235, 258
Completion of（的完成），36, 176, 213, 215, 218, 219, 224, 227, 233, 235, 236, 239, 245, 252, 254, 259, 263, 264, 265, 266, 271, 284, 517
Context of（的语境），263
Control of（的控制），258
Identification of（的标识），227, 252
Interface of（的接口），232-235
Invocation of（的调用），36, 217, 218, 222, 223, 225, 227, 231, 232, 235, 239, 245, 254, 263, 264, 270, 277
Parameter of（的参数），232
Priorization of（的优先级排列），258
Processing Information within（内处理信息），232
Processing of（处理），222, 224, 235, 254, 266
Result of（结果），37, 217, 219, 224, 227, 228, 264, 265, 517
Result Type of（的结果类型），227
Scheduling of（的调度），258
Structure of（的结构），218
Asynchronous Operation Processor（异步操作处理器），177, 218, 220, 223, 232, 237, 239, 252, 257, 258, 259, 432, 513
ACT Management within（内的异步完成标记管理），235
Emulation of（的仿真），254
Implementation of（的实现），231-235
Processing Information within（内处理信息），239
Asynchronous Procedure Call（异步过程调用），252
Asynchronous Transfer Mode（异步传输模式），9
Asynchrony（异步），38
Auditing（审核），109
Authentication（认证），12, 269, 283
Authorization（授权），12
Availability（可用性），76

B

Backus Naur Form（巴科斯范式），546
Bandwidth（带宽），8, 263, 546
Base-line Architecture（基线体系结构），506, 525
BNF。参见Backus Naur Form
Booch Components（Booch组件），330, 342
Broadcast（广播），11
Broker（代理者），398
Bus（总线），546
Busy Wait（忙等待），33, 546

C

C，xiii, 10, 11, 28, 47, 50, 53, 56, 57, 66, 70, 72, 83, 151, 171, 173, 209, 331, 363, 476, 484, 491
C++，xiii, xvii, xxix, 50, 55, 57, 65, 72, 84, 151, 156, 169, 171, 173, 322, 324, 325, 327, 331, 351, 353, 385, 417, 484, 519, 523
Assignment Operator（赋值运算符），57
break Statement（break语句），327
Compiler（编译器），332, 356, 493
Constructor（构造函数），62, 327, 519
continue Statement（continue语句），327
Copy Constructor（拷贝构造函数），57
Destructor（析构函数），62, 69, 327, 328, 519
Exception（异常），58, 327, 328
goto Statement（goto语句），327
Inlining（内联），72
Linker（链接器），356
operator->（运算符->），493
Pointer（指针），267
Private Method（私有方法），347
Protected Method（保护方法），347
Public Method（公有方法），347
return Statement（return语句），327
Standard Template Library（标准模板库），84, 97, 308
Static Object（静态对象），356
Template（模板），358, 416, 503, 561
Volatile Data（易变数据），358
Cache（高速缓存），6, 546
Affinity（相似性），252, 464, 471, 546
Coherency（相关性），358, 361, 449, 450
Flushing of（的刷新），361
Type of（的类型），552
Data Cache（数据缓存），552
Instruction Cache（指令缓存），552
Update of（的更新），34

Caching（缓存），7, 25, 358
Optimization of（的优化），361
Strategy（策略），25
Least-frequently Used（最不经常使用），25, 27, 39
Least-recently Used（最近最少使用），25, 27, 39
Call Center Management（呼叫中心管理），320
Callback（回调），14, 111, 272, 397, 514
Delivery（发出），272
Asynchronous（异步的），272
Synchronous（同步的），272
via Interrupt（通过中断），272
via Signal Handler（通过信号处理程序），272
CCM。参见CORBA Component Model
Centralization（集中），342, 443
Change Propagation（更改传播），139
Cheshire Cat（又称为句柄类（handle classes），是一种将实现与接口分开的技术），xxv
Choices（选择），442
Chorus COOL（一种用于面向对象应用程序的分布式编程环境），470
Class（类），xxix, 40, 51, 52, 53, 71, 173, 349, 517, 547
Abstract Class（抽象类），339, 545
Concrete Class（具体类），547
Constructor of（的构造函数），49
Destructor of（的析构函数），49
Interface of（的接口），28, 44, 47, 57, 73, 407, 517
Nested Class（嵌套类），147, 155
Private Method（私有方法），57
Relationships between（之间的关系），73
Subclass（子类），71, 299, 339, 376, 378, 420, 560
Superclass（超类），71, 187, 335, 376, 560
Class-Responsibility-Collaborator Card（类-责任-协作者卡），548, 563
Client（客户机），15, 75, 110, 139, 141, 146, 148, 165, 171, 173, 176, 179, 263, 264, 286, 288, 299, 366, 370, 374, 384, 397, 447, 469, 547
Asynchronous（异步的），16
Role of（的角色），16
Synchronous（同步的），16
Thin（瘦客户机），6
Closure（闭包），547
Cohesion（内聚性），28, 44, 47, 51, 54, 71, 72, 73, 517
Collaborator（协作者），547
Collocation（配置，组合），76, 94, 547
COM。参见Microsoft Component Object Model
Command Line（命令行），86
Command Processor（命令处理器），397
Commercial-Off-The-Shelf（商品），535
Component（组件），2
Middleware（中间件），18
Software System（软件系统），535
Common Object Request Broker Architecture。参见CORBA
Commonality（公共性/通用性），54
Communication（通信），525
Overhead（开销），76, 263
Peer-to-peer（对等），112
Protocol（协议），9, 286, 287
Role（角色），286
Service（服务），109
COMPAQ Alpha（一种硬件品牌），361
Compatibility（兼容性），50
Compiler（编译器），50, 52, 58, 72, 107
Conditional Compilation Directive for（条件编译指令），48
Defect of（缺点），47
Feature（的特点），47
Lexical Analyzer within（的词汇分析器），212
Limitation of（的限制），72
Optimizing（优化），358
Parser within（内的分析器、分析程序），212
Completion Event Queue（完成事件队列），218, 220, 221-224, 233, 235, 239, 271, 513
Event Buffering within（内的事件缓冲），220
Event Insertion into（事件插入），224, 245, 271
Event Removal from（事件删除），224, 245, 271
Management of（的管理），232
Selection of（的选择），237
Type of（的类型），239
Win32 Completion Port（Win32完成端口），239
Completion Handler（完成处理程序），37, 176, 177, 217, 220, 226, 235, 239, 259, 262, 264, 265, 267, 271, 282, 432, 513, 517
Behavior of（的行为），223, 266
Dispatching Target of（的目标分配），228
Function（函数），228
Object（对象），228
Event Processing within（在……内的事件处理），218, 219, 225, 227, 228, 231, 242, 245, 251, 253, 264, 265, 266, 272
Asynchronous（异步的），251
Synchronous（同步的），251
Identification of（的标识），227, 264, 265

Interface of（的接口），219, 227-231
Hook Method（钩子方法），177, 219, 221, 224, 227, 230, 236, 237, 240, 271
Multi-method（多方法），230
Single-method（单方法），228, 244
Lifecycle of（生命期），268
Management of（的管理），232
Result of（的结果），225
Serialization of（的串行化），242
Shared（共享的），252
Structure of（的结构），219
Subclassing of（从……派生子类），243
Table of（的表），268, 269
Complexity（复杂性），2, 11, 18, 38, 44, 107, 117, 137, 181, 207, 212, 217, 258, 322, 395, 425, 443, 444, 473
Accidental（偶发的），2, 8, 10, 18, 459
Implementation（实现），472
Inherent（固有的），2, 8, 19, 367
Physical（物理的），50
Structural（结构的），503
Component（组件），4, 9, 19, 39, 44, 70, 75, 78, 132, 141, 150, 160, 172, 173, 210, 217, 222, 256, 334, 346, 514, 515, 517, 518, 523, 526, 547
Access of（的访问），45, 93, 143, 144, 146, 172, 334, 349
by IPC（通过IPC），94
by Name（通过名字），94, 95
by Pointer（通过指针），94, 95
Concurrent（并发的），179
Local（本地），143
Remote（远程），143
Administration of（的管理），78, 79, 105
Architecture of（的体系结构），143
as a Unit of Configuration（作为配置单元），40, 78
Behavior of（的行为），79, 148
Cohesion of（的内聚性），51
Composition of（的成分），165, 172
Concurrency Strategy for（的并发策略），92, 106, 158
Configuration Mechanism for（的配置机制），91
Dynamic（动态的），8, 12, 27, 39, 77, 85, 106
Robust（健壮），91
Static（静态），8, 27, 75, 85, 106
Configuration of（的配置），8, 12, 40, 44, 75, 77, 78, 79, 80, 81, 85, 91, 95, 106, 335, 515, 523
Configuration Point of（的配置点），90
Configuration Policy for（的配置策略），13
Collocated（配置，组合），77, 94, 99
Distributed（分布式的），77, 94, 101, 143
In-band（带内），90
On-demand（按需），13
Out-of-band（带外），90
Configuration Protocol for（的配置协议），91
Connectivity of（的连接性），12
Control Interface of（的控制接口），82-83, 105, 106, 107
Inheritance-based（基于继承），83, 92, 104
Message Passing Based（基于消息传递），83, 92, 103
Control of（的控制），40, 78, 79, 84
Activation（激活），133
Finalization（完成），82
Identification（标识），165
Initialization（初始化），82
Lifecycle（生命期），82, 145, 161
Passivation（钝化），133
Resumption（恢复），77, 78, 79, 81, 82, 86, 91, 515
Suspension（挂起），77, 78, 79, 81, 82, 86, 91, 515
Termination（中止），77, 78, 81, 86, 91, 515
Control Status of（的控制状态），83, 84, 152
Cooperation between（的合作），139, 369
Corruption of（的破坏），93
Coupling of（的耦合），77
Customization of（的定制），342
Declaration of（的声明），337
Error Handling within（内的出错处理），151
Evolution of（的演化），44, 45, 51, 76, 77, 106, 142, 143, 144, 173, 342
Exchange of（的交换），76, 101
Factory（工厂），517
Family（系列/族），333, 334
Functionality of（功能性），45, 141, 144, 150, 165, 334, 517
Implementation of（实现），44, 75, 76, 78, 79, 106, 142, 143, 144, 148, 155-160, 170, 173, 335, 337, 339, 515
Implementation Strategy for（的实现策略），155
Multiple Inheritance（多继承），155
Nested Class（嵌套类），155
Separate Interface Class（分离接口类），156
Incompatibility of（的不兼容性），10
Initialization of（的初始化），77, 78, 79, 80, 86, 91, 107, 146, 355, 515
Instantiation of（的实例化），133, 144, 146, 148, 160, 517

Interface of（的接口），12, 44, 78, 79, 107, 141, 142, 143, 144, 173, 174, 335, 337, 515, 517, 518
Bloating（膨胀），45, 141, 143, 517
Constructor（构造函数），337
Implementation Method（实现方法），346, 347, 348, 518
Interface Method（接口方法），346, 347, 348, 518
Retrieval（获取/检索），174
Location of（的位置），86
Processing within（内的处理），81, 107, 146
Relationships between（关系），147, 504
Aggregation（聚合），504
Containment（包容），504
Establishment（建立），95
Logical（逻辑的），147
Physical（物理的），147
Role of（的角色），44, 143, 144, 145, 153, 172, 517
Self-deadlock of（的自死锁），339, 345
State of（的状态），95, 346
Structure of（的结构），79, 144
Thread-safety of（的线程安全），139, 334, 335, 337, 345, 346, 351
Usage Policy of（的使用策略），161
Version-skew of（的版本混乱），334
Component Configurator（组件配置器），78, 97, 98, 132, 135, 515
Behavior of（的行为），80
Component Configuration Directive for（组件配置命令），86-88
Parsing（分析、语法分析），88
Processing（处理），88
Specification（规范，规格说明书），86
Component Configuration Mechanism within（内的组件配置机制），88-92
Component Configuration Script for（的组件配置脚本），86
Directive Interpreter within（内的命令解释器），88
Implementation of（的实现），85-92
Interface of（的接口），85
Structure of（的结构），80
Component Factory（组件工厂），144, 162, 163, 169, 173
Behavior of（的行为），148
Finder（发现者），162, 173
Functionality of（的功能特性），161
Component Finding（组件发现），161
Component Instantiation（组件实例化），161
Component Lifecycle Management（组件生命期管理），161
Component Usage Policy（组件使用策略），161
Implementation of（的实现），160-164
Interface of（的接口），160, 170
per Component Type（每种组件类型），160
Singleton（单件），161
Structure of（的结构），146
Component Model（组件模型），149, 150, 523
Domain-specific（与领域相关的），150
Standard（标准的），150
Component Repository（组件仓库），78, 97, 515
Behavior of（的行为），80
Component Management within（内的组件管理），79
Implementation of（的实现），84
Structure of（的结构），79
Composite Pattern（组合模式），440
Compound Pattern（复合模式），538
Conciseness（简明性），28, 44, 47, 49, 51, 71, 73, 232, 517
Concrete Acceptor（具体接受器），292, 299, 301, 311
Concrete Completion Handler（具体完成处理程序），222, 245, 257
Behavior of（的行为），225
Functionality of（的功能特性），243
Handle Configuration of（的句柄配置），243
Generic（通用的，类属的），243
Hard-coded（硬编码的），243
Implementation of（的实现），243
State of（的状态），243
Structure of（的结构），219
Type of（的类型），243
Acceptor（接受器），243
Connector（连接器），243
Service Handler（服务处理程序），243
Concrete Component（具体组件），78, 515, 547
Behavior of（的行为），80
Concurrency Strategy for（的并发策略），92
Multi-processing（多进程），93
Multi-threading（多线程），93
Proactive（主动的），93
Reactive（反应的），93
Connection Establishment to（与……的连接建立），92
Implementation of（的实现），92-99
Processing within（内的处理），92
Structure of（的结构），79

Concrete Connector（具体连接器），292, 299, 306, 308, 311
Concrete Event Handler（具体事件处理程序），182, 184, 189, 205, 207, 229, 230, 452, 463, 465
Behavior of（的行为），184, 454
Functionality of（的功能特性），197
Handle Configuration of（的句柄配置），196
Generic（通用的，类属的），197
Hard-coded（硬编码的），196
Implementation of（的实现），196
State of（的状态），196
Structure of（的结构），182, 452
Type of（的类型），197
Acceptor（接受器），197, 211
Connector（连接器），197, 211
Service Handler（服务处理程序），197, 211
Time-based（基于时间的），208
Concrete Interceptor（具体截取器），111, 123, 124, 125, 127, 138, 514
Behavior of（的行为），115
Callback to（回调），113, 126
Dispatching of（分配），124
Implementation of（的实现），128
Instantiation of（的实例化），115
Interface of（的接口），115
Registration with Application Framework（向应用框架的注册），113, 115, 127
Structure of（的结构），112
Triggering by Framework（被框架激活），113
Concrete Method Request（具体方法请求），372, 377, 378
Concrete Service Handler（具体服务处理程序），292, 299, 300, 301, 306, 311
Concurrency（并发），xiv, xvi, xxvi, 5, 8, 10, 16-19, 25, 31, 176, 215, 254, 257, 283, 311, 326, 365, 369, 371, 395, 397, 459, 505, 516, 520, 524, 525, 532, 547
Control（控制），180, 211, 419, 442, 478
Mechanism（机制），16, 401
Strategey，216
Strategy（策略），16, 19, 26, 34, 41, 106, 158, 334, 342, 366, 447, 450, 472, 507, 509, 512, 515
Concurrency Pattern（并发模式），507, 509
Concurrent System（并发系统），2, 6-23, 326, 355, 366, 394, 401, 423, 424, 425, 429, 441, 442, 448, 506, 510, 511, 520, 521, 524, 526, 528
Condition Variable（条件变量），33, 44, 58, 380, 401, 409, 418, 433, 434, 453, 460, 465, 468, 547
Condition Expression within（条件表达式），58, 409
Destruction of（的销毁），60
Initialization of（的初始化），60
Notification of（的通知），59
Conditional Compilation（条件编译），48, 50, 55, 58, 69
Conduit（导管），442
Configurability（可配置性），28, 39, 72, 210
Configuration Management Pattern（配置管理模式），529
Connection（连接），514, 548
Acceptor（接受器），286
Establishment of（的建立），7, 92, 133, 177, 197, 199, 210, 243, 285, 286, 287, 294, 321, 322, 514
Active（主动的），63, 286, 288, 290, 295, 296, 299, 306, 322, 515, 545
Asynchronous（异步的），287, 290, 291, 294, 296, 306, 308, 309, 321, 322
Completion（完成），290, 295, 296, 308
Initiation（启动），295, 296, 306, 515
Passive（被动的），62, 286, 287, 289, 293, 299, 300, 301, 304, 322, 515, 556
Pending（待解决的），11, 296, 306, 308
Synchronous（同步的），290, 291, 294, 295, 308, 309, 322
Handshake（握手），322
Initiator（发起方/启动程序），286
Management of（的管理），27, 29, 30, 109, 520, 522
Role of（的角色），286
Active（主动的），294
Passive（被动的），293
Termination of（中止），133
Topology of（的拓扑），177
Connector（连接器），63, 197, 211, 243, 287, 290, 298, 303, 321, 507, 509, 513, 514, 515
Behavior of（的行为），293-296
Blocking of（的阻塞），290
Customization of（的定制），306
using Parameterized Types（使用参数化类型），306, 312
using Polymorphism（使用多态性），306, 312
Factory（工厂），290
Implementation of（的实现），308
Instantiation of（的实例化），292, 311
Interface of（的接口），306, 308
Connection Completion Method（连接完成方法），290, 296, 306, 308, 309
Connection Initiation Method（连接启动方法），290,

295, 296, 306, 308
Hook Method（钩子方法），306, 308,. 312
Template Method（模板方法），306, 308,. 312
Notification by Dispatcher（分配器通知），291, 296
Registration with Dispatcher（向分配器注册），291, 296
Service Handler Management within（内的服务处理程序管理），306, 308, 309
Structure of（的结构），290
Consumer（消费者），32, 59, 369, 386, 399, 413, 445
Container（容器），84, 116, 124, 133, 165, 468, 548
Context Object（语境对象），111, 126, 128, 136, 138, 139, 140, 514
Behavior of（的行为），116
Creation of（的建立），123
Deletion of（的删除），123
Event-specific（针对具体事件的/与事件有关的/事件特定的），116
Implementation of（的实现），121-123
Interface of（的接口），113, 122, 140
Accessor Method（访问器方法），113, 121
Mutator Method（变异器方法），114, 121
Passing of（的传递），123, 126, 127
Per-event（每个事件），123, 124
Per-registration（每次注册），123
Semantics of（的语义），121
Structure of（的结构），113
Context Switch（语境切换），19, 34, 180, 212, 217, 257, 262, 396, 441, 443, 449, 467, 472
Continuation（连续性），260, 548
Contract（合同），145, 517
Controller（控制器），141
CORBA（公共对象请求代理体系结构），xiii, 9, 23, 90, 94, 133. 136, 171, 210, 268, 280, 320, 441, 470, 529, 532,548
Asynchronous Method Invocation（异步方法调用），207
CORBA Component Mode（CORBA组件模式），12, 132, 149, 150, 171, 523
CORBA Fault Tolerant Specification(CORBA容错规范），134, 534
CORBA Naming Service（CORBA命名服务），23
CORBA Object Reference（CORBA对象引用），268
CORBA Trading Service（CORBA交易服务），23, 163
Event Notification Service（事件通知服务），12
Fault Tolerant CORBA Specification(容错CORBA规范），22
General Inter-ORB Protocol（通用的ORB间协议），270
IDL Mapping（IDL映射），156
Internet Inter-ORB Protocol（因特网ORB间协议），287, 294
Portable Interceptor Specification（可移植的截取器规范），134
Tie Adapter（联系适配器），156
Correctness（正确性），19
Cost-effectiveness（成本效率），6, 7
COTS。参见Commercial-Off-The-Shelf
Coupling（耦合），27, 54, 77
CPU（中央处理器），548
Starvation（饿死），14
CRC Card。参见Class-Responsibility-Collaborator Card
Creativity（创造性），5
Critical Section（临界区），20, 158, 205, 324, 226, 331, 333, 334, 337, 353-356, 367, 376, 408, 450, 512, 519, 548
Entering of（的进入），326, 328, 331
Execution of（的执行），355, 357, 360
Identification of（的标识），356
Leaving of（的离开），326, 328, 331
Read-side（读方），558
Return Path from（返回路径），20
Scope of（的范围），327, 328, 329, 331, 332, 356
Write-side（写方），562
Customization（定制/个性化），5, 342

D

Daemon（守护进程），549
Data（数据），31, 54, 73, 426
Access of（访问），370, 478
Allocation of（分配），441
Buffering of（缓冲），31, 232, 426, 430, 432, 443, 471
Flow Control（流控制），432, 433
Notification（通知），432, 433
Ordering（排序），432
Serialization（串行化），432, 433
Copying of（备份），19, 441, 443
Corruption of（的毁损），456, 457
Decryption of（解密），119
Encapsulation of（的封装），52
Encryption of（加密），12, 119
Exchange of（交换），286, 288, 294, 295, 296, 298, 300
Manipulation of（操纵），52
Movement of（移动），19, 180, 181, 217, 262, 396, 467, 471

Passing of（的传递），478
Processing of（处理），31
Pulling of（拉），438
Reception of（接收），285
Routing of（路由），285
Sending of（发送），285
Shared（共享的），58, 502
Stream of（流），139
Transfer of（传送），442
Type of（的类型），52
Data Structure（数据结构），54, 73
Creation of（的建立），55
Destruction of（销毁），55
Untyped（无类型的），71
Data Structure Pattern（数据结构模式），531
Database（数据库），6, 168, 268, 425, 429, 443, 447
Programming Library（编程库），49
DCE Cell Directory Service（DCE单元目录服务），23
DCOM。参见Microsoft Distributed Component Object Model
Deadlock（死锁），3, 8, 18, 19, 21, 138, 205, 326, 331, 339, 351, 407, 469, 478, 549
Debugger（调试器），3, 8, 136, 212, 396
Decoupling（去耦合，分解），73, 78, 106, 136, 172, 257
Demarshaling（散集），393, 549
Dependability（依赖性），21, 532, 534, 536
Design（设计），xiv, 4, 5, 18, 19, 149, 366, 506, 508, 509, 539, 549
Aspect（方面），138
By Contract（按合同），540
Complexity（复杂性），172
Force（强制条件），72
Logical（逻辑的），72
Method（方法），530
Object-oriented（面向对象的），xxviii
Pattern-based（基于模式的），520, 526
Physical（物理的），72
Principle（原则），56
Problem（问题），4
Stability（稳定性），149
Strategic（战略的），5
Tactical（战术的），5
Technique（技术），3, 56
Design Pattern（设计模式），xxvi, 5, 18, 506, 508, 525, 549
Design-by-contract（按合同设计），540
Desktop Computer（台式机），2
Deterministic Finite Automata（确定性有限自动机），212
Device（设备），549
Driver（驱动程序），66, 103, 549
Diligence（勤奋），5
Directory Service（目录服务），6
Discipline（学科），5
Dispatcher（分配器），111, 114, 120, 123, 131, 138, 140, 291, 298, 306, 514
Behavior of（的行为），115, 293-296
Event Demultiplexing within（事件多路分解），291
Event Loop of（事件循环），291, 293
Handler Notification by（处理程序通知），291, 292, 293, 296, 304, 308
Handler Registration with（处理程序注册），291, 293, 303
Interceptor Callback Policy for（的截取器回调策略），124, 127
First-in, First-out（先进先出），127
Interceptor Chain（截取器链），127
Last-in, First-out（后进先出），127
Priority-based（基于优先级的），124, 127
Interface of（的接口），113, 124, 136
Event Notification Method（事件通知方法），113, 125
Interceptor Registration Method（截取器注册方法），113, 124
Notification by Application Framework（应用程序框架的通知），126, 127
Specification of（的规范），124-126
Structure of（的结构），113, 291
Distributed Object Computing（分布对象计算），xiii, xv, 22, 23, 170, 505, 521, 524, 530, 534, 548
Directory Service（目录服务），534
Naming Service（命名服务），534
Remote Service Location（远程服务定位），534
Service Partitioning（服务划分），534
Distribution（分布），549
DLL。参见Dynamically Linked Library
DNS（域名系统），423
Domain Analysis（领域分析），150
Double-Dispatching（双分配），190, 200, 240, 303
Dynamic Binding（动态绑定），550
Dynamic Linking（动态连接），13, 29, 107
Mechanism（机制），13, 88
Policy（策略），13
Dynamically Linked Library（动态链接库），13, 40, 78,

84, 88, 132, 515, 550
DynamicTAO，104, 107, 135, 140
Dynix/PTM，342, 350, 360

E

Education（教育），531, 540
Efficiency（效率），5, 6, 18, 19, 21, 38, 49, 93, 137, 173, 232, 262, 269, 283, 302, 321, 324, 334, 367, 397, 400, 416, 420, 425, 429, 441, 444, 472, 502, 536
Eiffel，322
EJB。参见Java Enterprise JavaBeans
E-mail（电子邮件），9
Encapsulation（封装），xxviii, 27, 51, 52, 55, 57, 63, 71, 73, 74, 257, 359
Endpoint（端点），34, 550
Environment Variable（环境变量），86
Error Handling（出错处理），58, 65, 127, 170, 476, 517
  errno（错误号），66, 476, 498, 500, 503
  Error Code（错误代码），65
  Error Value（错误值），65
  Low-level（低层），10
Eternal（永久的），134
Ethernet（以太网），12, 435, 438
Event（事件），30, 109, 116, 121, 123, 138, 181, 213, 366, 447, 454, 470, 512, 526, 550
  Demultiplexing of（的多路分解），7, 16, 25-38, 83, 140, 176, 177, 179-260, 261, 263, 282, 298, 366, 397, 430, 434, 440, 447-474, 512, 513, 549
    Active（主动的），269
    Asynchronous（异步的），213
    Concurrent（并发的），455, 457
    Synchronous（同步的），30, 176, 211
  Detection of（的检测），16, 34, 366, 447, 450, 454, 463, 467, 512
  Dispatching of（的分配），7, 16, 30-38, 83, 112, 140, 176, 177, 179-260, 298, 366, 397, 440, 447-474, 512, 513
  Handling of（的处理），xvi, xxvi, 5, 14-16, 140, 175, 520, 522, 524, 525
  Hand-off of（的转交），467, 469
  Occurrence of（的发生），111, 183, 191, 192, 206, 213, 218, 221, 224, 236, 271
  Priority of（的优先级），472
  Processing Information about（处理信息），264, 467
  Processing of（的处理），14, 16, 29, 34, 36, 120, 176, 177, 179-284, 298, 366, 447-474, 512, 513
    Asynchronous（异步的），37, 216
    Concurrent（并发的），206, 217, 449, 451, 468, 513
    Iterative（迭代），512
    Proactive（主动的），176, 222
    Reactive（反应的），176, 184, 191
    Synchronous（同步的），30
  Propagation of（的传播），126
  Queueing of（的排队），181, 454
  Reception of（事件接收），16, 186, 207, 291, 470, 512
    Asynchronous（异步的），37, 216
    Concurrent（并发的），512, 513
    Synchronous（同步的），181
  Type of（的类型），15, 123, 125, 181, 187, 190, 193, 230, 259, 262, 449, 458, 470, 514
    Application-specific Event（针对具体应用的事件/应用特定的事件），14, 140, 182, 452, 514
    Completion Event（完成事件），16, 36-38, 177, 216-239, 245, 259, 262-264, 271, 273, 291, 432, 513, 547
    Control Event（控制事件），16
    Data Event（数据事件），16
    External Event（外部事件），14
    I/O Event（I/O事件），182, 192, 207, 457
    Indication Event（指示事件），15, 29, 36, 37, 176, 179-214, 215, 216, 291, 292, 293, 551
    Lifecycle Event（生命期事件），133
    Request Event（请求事件），15, 366, 447, 471, 512, 558
    Response Event（响应事件），15, 558
    Timer Event（定时器事件），207
Event Demultiplexer（事件多路分解器），14, 458, 473
  GetQueuedCompletionStatus()，38, 221, 237
  poll()，16, 191, 192, 457
  select()，14, 16, 29, 30, 182, 191, 192, 448, 457, 458, 459
  WaitForMultipleObjects()，16, 30, 191, 192, 237, 457, 459
Event Handler（事件处理程序），14, 30, 138, 176, 181, 190, 193, 205, 207, 211, 213, 215, 259, 263, 264, 298, 311, 440, 451, 453, 454, 458, 463, 465, 466, 471, 473, 507, 509, 512, 513
  Activation of（的激活），31
  Behavior of（的行为），184, 454
  Callback to（的回调），187
  Connection to（连接），31

Creation of（的建立），31
Dispatching Logic within（内的分配逻辑），37
Dispatching Target of（的分配对象），186
Function（函数），186
Object（对象），186
Event Processing within（内的事件处理），31, 182, 183, 185, 186, 197, 199, 206, 212, 452, 453
Implementation of（的实现），187
Interface of（的接口），182, 186-189, 196, 452
Hook Method（钩子方法），176, 182, 183, 184, 185, 186, 188, 189, 191, 197, 204, 207, 212, 452, 453, 455, 458, 465
Multi-method（多方法），188
Single-method（单方法），187, 197
Serialization of（的串行化），196, 207
Structure of（的结构），182, 452
Subclassing of（的子类派生），186
Templatized（模板化），197
Event Sink（事件汇），470
Event Source（事件源），14, 30, 34, 181, 184, 211, 213, 259, 366, 426-432, 447, 449, 456, 470, 473, 512
Blocking on（阻塞于），30
External（外部），181, 451
Internal（内部），181, 451
Set（集），450, 451
Shared（共享的），450
Evolution（进化, 演化），49, 51
Exception Handling（异常处理），49, 65, 151, 517
Benefits of（的好处），65
Drawbacks of（的缺点），66
Exception Hierarchy（异常层次），65
Platform-specific Variation of（针对具体平台的变种），66
Structured（结构化），66
Type-Safety of（的类型安全），65
Exception-safety（异常安全性），326, 550
Experience（经验），5
Experience Report（试验报告），538
Extended Markup Language（可扩展标记语言），539
Extensibility（可扩展性），6, 49, 54, 65, 112, 136, 137, 172, 186, 188, 229, 230, 321, 335, 420, 523, 536
Extension Interface（扩展接口），144, 146, 152, 157, 158, 163, 169, 170, 172, 173, 517
Access Control of（的访问控制），170
Behavior of（的行为），148
Error Handling within（错误处理），151
Implementation of（的实现），148, 155
Interface of（的接口），147, 148, 149
Asymmetric（非对称的），170
Bloating（膨胀），154
Domain-specific Interface（针对具体领域的接口），152
General-purpose Interface（通用接口），153, 170
Root Interface（根接口），144, 145, 148, 149, 150-153, 170, 171
Linking between（的连接），155
Naming of（命名），151
Property of（的属性），163
Reference Counting within（内的引用计数），151, 157
per Component（每个组件），157
per Extension Interface（每个扩展接口），157
Reflection about（对……的映像），152
Retrieval of（获取、检索），144, 145, 149, 151, 156
Reflexivity（自反性），156
Symmetry（对称性），156
Transitivity（传递性），156
Role of（的角色），146
Sharing of（共享），151
State of（的状态），158
Structure of（的结构），145
Extension Object（扩展对象），169
Extreme Programming（极限编程），540

F

Factory（工厂），550
Fault Tolerance（容错），7, 109, 111, 114, 136, 536
File System（文件系统），72, 84
File Transfer（文件传输），9, 425
File Transfer Protocol（文件传输协议），287, 294, 423, 436
Finite State Machine（有限状态机），14
Flexibility（灵活性），21, 40, 110, 136, 137, 197, 232, 283, 301, 302, 342, 343, 396, 472
Flow Control（流控制），31, 32, 196, 369, 387, 430, 432, 550
Flow Of Control（控制流），14, 176, 177, 222, 326
Inversion Of Control（控制逆转/控制转换），14, 38, 82, 176, 184, 212, 259, 473
Oscillation of（的振荡），184, 212, 258
Self directed（自指向的），14

FORTRAN，57, 268
FTP（文件传输协议），9, 10
Full Association（全关联），550
Function（函数/功能），44, 47, 52, 54, 73, 550
  Clustering of（簇），54
  Coalescing of（合并），54
  Cohesive Group of（ 内聚组），54, 72
  Encapsulation of（的封装），52
  Low-level（低层），73
  Parameter of（的参数），52
  Syntax of（的语法），54
Functional Property（函数属性），550
Future（前景），371, 377, 379, 516, 550
  Behavior of（的行为），374
  Blocking on（阻塞于），373
  Deletion of（的删除），375, 385
  Polling of（的轮询），373
  Rendezvous with（与……的汇合），373, 375, 384
    Asynchronous（异步的），384
    Synchronous Timed Wait（同步定时等待），384
    Synchronous Waiting（同步等待），384
  Result of（的结果），375, 391
    Evaluation（求值），386
    Space Consumption（空间消耗），373
  Structure of（的结构），373
  Type of（的类型），391
    Polymorphic（多态性），391
    Typed（类型的），391

G

Gang-of-Four（四人帮），532
Garbage Collection（无用单元收集），49, 66, 151, 283, 375, 385
Gateway（网关），285, 312, 369, 386, 399, 413, 419, 551
Generic Programming（类属编程），312, 416
Genetics（基因学），541
Global Variable（全局变量），52
Graphical User Interface（图形用户界面），70, 73, 86, 522, 526, 551
  Display（显示），209, 522
  Event Handling within（内的事件处理），209
  Library（库），522
Guard（哨兵），327, 332, 337, 354, 355, 519
  Creation of（的建立），327
  Implementation of（的实现），327
  Interface of（的接口），327
    Constructor（构造函数），327, 336
    Destructor（析构函数），327, 332
    Explicit Accessor Method（显式的访问器方法），329
  Owner Flag within（内的所有者标志），328
  Strategized（策略化的），336
    Bridge-based（基于桥的），341
    Parameterized Type based（基于参数化类型的），336, 337
    Polymorphism-based（基于多态性的），336, 337
  Templatized（模板化的），337
GUI。参见Graphical User Interface

H

Handle（句柄；处理），30, 179-260, 447-474, 513, 515, 551
  Access of（访问），456
  Activation of（激活），459
  Deactivation of（使……无效，停用），459, 463, 467
  Event Occurrence on（上的事件发生），451, 454, 456, 458, 467
  Handle/Thread Association（句柄/线程关联系）
    Bound（受限的），467, 471
    Hybrid（混合的），469
    Unbound（不受限的），467, 471
  Operation Initiation on（上的操作启动），451, 457
  Reactivation of（重新激活），463
  Type of（的类型），192, 456
    Concurrent（并发的），192, 207, 456, 458
    I/O Handle（I/O句柄），192
    Iterative（迭代），456
    Shared（共享的），458
    Socket Handle（套接字句柄），181
    Transport Handle（传输句柄），287
    Writer/Reader（记录器/阅读器），206
  Value of（的值），471
Handle Set（句柄集），179-214, 447-474
  Multiple（多），468
  Synchronization of（的同步），463
  Type of（的类型），457
    Concurrent（并发的），457, 458, 468
    Iterative（迭代），457, 463
    Shared（共享的），457, 467, 472

Hardware（硬件），366, 371, 395, 401, 425
Diversity of（的多样性），28
I/O Hardware（输入输出硬件设备），427
Interrupt Handler（中断处理程序），425, 431
Multi-processor Platform（多处理器平台），19, 21, 353, 361, 443, 447, 468, 478
Technology（技术），2
Hardwiring（硬布线），551
Hash Function（散列函数），487
Hash Table（散列表），434, 487
Heap（堆），464
Herculean Task（巨大的任务），4
Hollywood Principle（好莱坞法则），176, 184
Hook Method（钩子方法），83, 112, 116, 123, 125, 126, 138, 182, 219, 231, 301, 306, 378, 452
Host（主机），75, 77, 285, 312, 551
HTML。参见Hyper Text Markup Language
HTTP。参见Hyper Text Transport Protocol
Hyper Text Markup Language（超文本标记语言），539
Hyper Text Transport Protocol（超文本传输协议），9, 287, 294, 423, 551, 555
Client（客户机），24, 30
Cookie，279
Daemon Process（守护进程），435
Processing（处理），216
Request（请求），31, 215, 279, 435
Server（服务器），6, 24, 30

I

I/O（输入/输出），16, 227, 438, 448, 459, 476, 512
Architecture（体系结构），441
Bottleneck（瓶颈），473
Completion Port（完成端口），14, 16, 38, 432
Device（设备），232, 233, 423, 426, 436, 437, 444
Event（事件），182, 192, 207, 457, 466, 473
Handle（句柄），17, 192, 201
Hardware（硬件），427
Operation（操作），31, 35, 61, 232, 252, 424, 439, 458, 473
Protocol（协议），426
Socket（套接字），430
Stream（流），444
Thread of Control（控制线程），448
Transfer Subsystem（传输子系统），134, 210, 472
Type of（的类型），11
Asynchronous（异步的），11, 28, 36, 41, 93, 176, 216, 218, 252, 259, 397, 432, 435, 440, 441, 444, 449, 473, 512, 546
Blocking（阻塞），424
Overlapped（重叠的），218, 432, 439
Proactive（主动的），431
Signal-driven（信号驱动的），17, 438
Synchronous（同步的），41, 397, 426, 442, 560
Idiom（惯用法），xxvi, 57, 525, 529, 551
IDL。参见Interface Definition Language
Indirection（间接寻址/间接/间接方法），49, 72, 107, 173, 351, 492
Overhead（开销），49, 73, 322
Inetd，104, 319, 423
Inheritance（继承），xxix, 72, 83, 142, 172, 173, 189, 299, 302, 303, 551
Multiple（多继承），147, 155, 555
Single（单继承），559
Inheritance Anomaly（继承异常），376, 420
Initialization（初始化），525
Initiator（启动程序），218, 222, 227, 231, 232, 235, 239, 245, 251, 252, 259, 264, 267, 270, 277
ACT Dispatching within（内的异步完成标记分配），271
ACT Holding within（对异步完成标记的持有），269
Explicit（显式），269
Implicit（隐式），269
Behavior of（的行为），223, 266
Event Demultiplexing within（内的事件多路分解），265, 266, 271, 273, 282
Callback（回调），271
Queued Completion Event（放入队列的完成事件），271
Implementation of（的实现），244
Robustness of（的健壮性），268
Structure of（的结构），222, 265, 282
Type of（的类型），283
Multi-threaded（多线程的），283
Single-threaded（单线程的），283
Inlining（内联），55, 72, 551
Instance（实例），552
Instruction Pointer（指令指针），17
Intel Itanium，361
Interception（截取），116, 117, 118, 131, 138

Cascade（层叠），138
Interception Group（截取组），120, 123, 126
Interception Point（截取点），118-121, 122, 123, 124, 126, 127
Partitioning of（的划分），118
Reader Set（阅读器集），118, 121
Semantically-related（语义相关的），120
Writer Set（记录器集），118, 121
Interceptor（截取器），111, 119, 120, 121, 136, 137, 138, 139, 140, 514, 515
Behavior of（的行为），115, 117
Callback Policy（回调策略），124-128
Callback to（对……的回调），114, 115, 123, 127, 514
Dynamic Loading of（的动态装载），132
Factory for（工厂），132, 134
Implementation of（的实现），514
Instantiation of（的实例化），133
Interface of（的接口），123, 125, 128, 136, 514
Library of（库），132
Registration with Framework（向框架注册），114, 131, 132, 136, 138, 514
Automated（自动的），125
Explicit（显式的），132
Implicit（隐式的），132
Stack of（栈），127
Structure of（的结构），112
Triggering by Framework（由框架激活），114
Type of（的类型），128
Client-side（客户机端），136
Erroneous（错误的），137
Logical（逻辑上的），128
Malicious（恶意的），137
Physical（物理上的），128
Server-side（服务器端），137
Singleton（单件），131
Symmetrical（对称的），136
Interface（接口），552
Bloating of（膨胀），141, 142, 154, 172
Conversion of（转换），73
Negotiation（协商），171
Uniform（统一），54
Interface Definition Language（接口定义语言），170
Internet（因特网），7, 24, 424, 429, 430, 535, 552
Backbone（主干），2
Chat Room（聊天室），7
Domain Address（域地址），61
Service（服务），319
Internet Protocol（互联网协议），xxix, 9, 15, 47, 61, 94, 179, 285, 288, 289, 430, 435, 437, 449, 552
Interprocess Communication（进程间通信），xxix, 7, 9, 11, 90, 94, 196, 206, 513, 514, 515, 523, 552
Mechanism（机制），63, 292, 299, 301, 303, 306, 312, 513, 514, 515
Interrupt（中断），423, 429, 431, 433, 441
Handler（处理程序），431, 437
Handling of（处理），442
Hierarchy of（的层次），431
Priority of（优先级），431
InterViews（内部视图），209
Dispatcher（分配器），209
Interviews（内部视图），522
Intranet（内联网），23, 24, 552
Introspection（反省），111, 552
Invariant（不变式），401, 408, 552
IP。参见Internet Protocol
IPC。参见Inter Process Communication
Iterator（迭代器），379

J

Java，xiii, xvii, xxix, 13, 55, 65, 73, 84, 151, 156, 169, 171, 173, 213, 260, 267, 320, 331, 343, 350, 351, 352, 362, 394, 418, 500, 522, 523, 529, 533
Applet（小应用程序），103, 523
Bytecode（字节代码），362
Class Library（类库），70, 533
AWT，70
Swing（摇摆），70, 522
Development Community（开发团体），533
Enterprise JavaBeans（企业Jave Beans），12, 132, 149, 150, 171, 523
Event Handling（事件处理），214
Java Naming and Directory Interface（Java命名和目录接口），171
Java Virtual Machine（Java虚拟机），18, 70, 214, 254, 260, 352, 418, 533
Listener（监听者），214
Monitor（监视器），343, 352, 418
Remote Method Invocation（远程方法调用），xiii, 23, 555

Synchronized Block（同步阻塞），331
Synchronized Method（同步方法），352
Third-party Framework（第三方框架），533
Java RMI。参见Java Remote Method Invocation
Jini，xiii, 282
Handback Object（轮回对象），282
Jitter（抖动），8, 22, 553
Judgement（判断），5

K

Kerberos，110
Key（关键字/密钥），479, 480, 482, 484, 486, 487, 492, 495
Creation of（的建立），486, 489, 492, 495, 496, 502
Deletion of（的删除），491, 495
Globally-unique（全局惟一的），479
Range of（的范围），487
Recycling of（的回收），485, 495
Key Factory（关键字工厂），479, 486, 495
Behavior of（的行为），482
Implementation of（的实现），489
Key Creation within（内的关键字建立），489

L

Late Binding（后期绑定），553
Latency（延迟），8, 22, 34, 181, 217, 263, 287, 294, 321, 325, 449, 553
Layer（层），136, 138, 139, 140, 366, 423, 443, 445, 511, 553
Application Layer（应用层），14, 31, 82, 186, 226, 297, 299, 301, 306, 311
Asynchronous Processing Layer（异步处理层），31, 366, 423-445, 511
Base-level Layer（基级层），140
Component Layer（组件层），74
Configuration Management Layer（配置管理层），82
Connection Management Layer（连接管理层），297, 299
Event Demultiplexing Layer（事件多路分解层），14, 176, 186, 216, 226, 297, 298
Event Dispatching Layer（事件分配层），186, 226, 297, 298
Event Handler Layer（事件处理程序层），14
Event Source Layer（事件源层），14
Meta-level Layer（元级层），140
Queueing Layer（排队层），31, 32, 366, 423-445, 472, 511
Socket Layer（套接字层），427
Synchronous Processing Layer（同步处理层），31, 366, 423-445, 511
Transport Layer（传输层），34, 369, 562
LDAP 9, 23
Legacy（遗留），443, 478
lex，88, 212
Library（库），443, 478
Linker（连接器），72
Linux，103, 232, 360
LinuxThreads，360
Liskov Substitution principle（Liskov替换原则），142, 149
Listen（侦听），104, 319
Load Balancing（负载平衡），8, 109, 111, 114, 129, 136, 447, 534, 553
Local Area Network（局域网）。参见Network
Location Broker（定位代理者），23
Locator Service（位置服务），169
Lock（锁），19, 20, 33, 40, 44, 204, 323-363, 385, 418, 477, 492, 518, 519, 553
Abstract（抽象），335, 339
Acquisition of（获取），20, 40, 60, 205, 293, 323-363, 385, 477, 502, 518, 519
Automatic（自动的），327, 328, 331, 356
Explicit（显式的），326
Assignment of（的赋值），328
Contention of（对……的争夺），351, 363
Copying of（的复制），328
Hint（提示），353, 355, 357, 358, 360, 361
Initialization of（的初始化），356
Interface of（的接口），336, 339
Accessor Method（访问器方法），329
Release of（的释放），20, 40, 60, 69, 205, 323-363, 477, 502, 518, 519
Automatic（自动的），327, 328, 331, 332, 356
Explicit（显式的），326, 329
Sharing of（对……的共享），351

Type of（的类型），21, 327, 336, 518, 519
File Lock（文件锁），339
Mutex（互斥），20, 49, 57, 59, 66, 67, 70, 73, 205, 323-363, 380, 393, 401, 408, 433, 434, 502, 519, 555
Non-recursive Mutex（非递归互斥），339, 346
Null Lock（空锁），334, 346
Polymorphic Lock（多态锁），335, 336, 341
Readers/Writer Lock（阅读器/记录器锁），20, 334, 339, 343, 519, 558
Recursive Mutex（递归互斥），205, 339, 346, 558
Semaphore（信号灯），20, 205, 323-363, 401, 433, 453, 460, 464, 465, 468, 519, 559
Strategized Lock（策略化加锁），346, 518, 519
Locking（加锁），40, 324, 333, 443, 483, 486, 502, 519
Efficiency of（的效率），351
Logic（逻辑），356
Mechanism（机制），324, 333, 335, 342, 519
Family of（系列），339, 341
Parameterized Type based（基于参数化类型的），335, 336
Polymorphism-based（基于多态性的），335
Overhead of（的开销），324, 339, 345, 346, 354, 355, 360, 367, 407, 471, 475, 477, 478, 502, 518, 519
Protocol for（协议），477
Robustness of（健壮性），351
Logging（记录），47, 67, 109, 179, 197, 475, 498
Loopback Device（回环设备），553
LynxOS，18

M

Mach，439
Maintainability（可维护性），16, 19, 28, 44, 47, 49, 50, 51, 69, 71, 73, 326, 342, 424, 517, 540
Marshaling（列集），170, 393, 553
Memento（备忘录），95, 284
Hierarchy of（的层次），95
Maintenance of（的维护），95
Memory（存储器/内存），
Allocation of（的分配），34, 448, 450, 471
Barrier（障栅），361
Consumption of（消耗），486
Leak（泄漏），283, 354, 357
Management（管理），11, 54
Re-mapping of（重映射），284
Message（消息），31, 426, 553
Buffering of（的缓冲），426, 430, 432, 443
Flow Control（流控制），432, 433
Notification（通知），432, 433
Ordering（排序），432
Serialization（串行化），432, 433
Passing（传递），22, 554
Pulling of（拉），438
Message Queue（消息队列），470
Synchronized（同步的），59
Method (Development)（方法（开发）），18, 530, 538
Analysis（分析），3
Limitation of（的局限性），18
Pattern-based（基于模式的），530, 538
Method（方法）。参见Service
Method Closure（方法闭包），554
Method Request（方法请求），371, 376, 379, 384, 389, 392, 393, 400, 516
Behavior of（的行为），374
Creation of（的建立），371, 373, 374, 377, 390
Deletion of（的删除），375
Dequeueing of（从队列中删除），516
Dispatching of（分配），381, 516
Execution of（的执行），372, 373, 374, 375, 377, 378
Implementation of（的实现），378
Interface of（的接口），371, 378
Constructor（构造函数），379
Guard（哨兵），372, 373, 374, 378, 382, 396
Management of（的管理），373, 381
Pending of（待处理的），372, 373, 381
Scheduling of（的调度），374
Structure of（的结构），371
MFC。参见Microsoft Foundation Classes
Micro-architecture（微体系结构），4
Microkernel（微内核），439
Microsoft，522
ActiveX，548
Component Object Model（组件对象模型），xiii, 9, 12, 132, 135, 149, 150, 170, 342, 523
Active Template Library（主动模板库），502
Custom Marshaling（自定义列集），135
Globally Unique Identifier（全局惟一标识符），151, 163
Run-time System（运行时系统），135
Distributed Component Object Model（分布式组件对象

模型），23
Foundation Classes（基础类库），70, 522
Internet Explorer（因特网浏览器），135, 320
Visual Studio，342
ATL Wizard（ATL向导），342
Win32，xiv, 13, 18, 28, 51, 56, 60, 88, 182, 192, 193, 195, 216, 218, 237, 457, 501, 503
Asynchronous Operation（异步操作），36, 38
Asynchronous Procedure Call（异步过程调用），252
C API，70
CreateIoCompletionPort()，280
errno（错误号），500
Exception Handling（异常处理），65
GetQueuedCompletionStatus()，38, 221, 238, 244, 252, 271
I/O Completion Port（I/O完成端口），16, 221, 239, 244, 252, 271, 280, 432, 439
Mutex（互斥），58
Named Pipe（命名管道），303
Overlapped I/O（重叠输入输出），218, 244, 271, 279, 280, 432, 439
select()，211
Socket（套接字），233
Synchronization（同步），53
Threads（线程），10, 18, 29, 50, 53, 63, 491, 495
WaitForMultipleObjects()，16, 30, 206, 211, 238, 458, 468
Windows NT，28, 34, 36, 48, 66, 103, 216, 220, 232, 235, 255, 257, 432, 439, 476, 485
I/O Subsystem（输入输出子系统），218
Kernel（内核），38, 221, 252
Service Control Manager（服务控制管理器），103
Thread Scheduler（线程调度程序），252
WinSock，11
Middleware（中间件），xiii, xv, xxvii, 2, 9, 23, 109, 170, 366, 505, 510, 520, 521, 524, 528, 530, 532, 534, 554
Reflective（自反的，反身的），104, 135, 536
Mix-In（混入），554
Model（模型），141
Modularity（模块化），xxviii, 40, 72, 105, 210, 540
Module（模块），554
Monitor（监控/监视器），418, 554
Dijkstra-style（Dijkstra风格），418
Hoare-style（Hoare风格），418
Monitor Condition（监视器条件），408, 410, 416, 420, 516
Behavior of（的行为），404
Notification by（由…通知），420
Notification of（的通知），404, 407
Structure of（的结构），403
Waiting on（等待），404, 407
Monitor Daemon（监视器守护进程），22
Monitor Lock（监视器锁），401, 403, 408, 410, 416, 516
Acquisition of（获得），404, 407
Behavior of（的行为），404
Release of（的释放），404, 407
Structure of（的结构），402
Monitor Object（监视器对象），401, 402, 420, 516
Access of（访问），401, 402
Behavior of（的行为），404, 516
Configuration of（的配置），416
Critical Section within（内的临界区），415
Data Member of（的数据成员），408, 410
Execution of（的执行），402
Functionality of（的功能特性），420
Implementation of（的实现），410-413
Interface of（的接口），401, 417, 402
Implementation Method（实现方法），407, 411
Interface Method（接口方法），406, 407, 411
Invariant of（的不变式），408
Multiple Roles of（的多角色），417, 516
Scheduling of（的调度），407, 420
State of（的状态），401, 402, 404, 408, 516
Structure of（的结构），402
Synchronization of（的同步），408, 420
Monitoring（监控），22, 111, 136
Moore's Law（摩尔定律），2, 554
MTV（音乐电视），552
Multicast（多播），11, 134, 555
Mutual Exclusion（互斥），20, 28, 205

N

Name Binding（名字绑定），23
Name Space（名字空间），23
Namespace（名字空间），51
Naming（命名），6, 12, 22
Nested Monitor Lockout（嵌套的监视器阻塞），349, 352, 420
NetPC（网络PC），6

Netscape Communicator（网景通信程序），135, 320
Network（网络），22, 24, 31, 75, 143, 232, 301, 369, 393, 433, 555
Byte Order（字节序），50, 61
Connection（连接），181, 451
interface（接口），555
Local Area（局域网），2, 23, 75
Management（管理），6
Neural（神经），70
Platform（平台），449
Programming（编程），xiv, 49
Protocol（协议），xv, xxix, 5, 7
Service（服务），xxix
Wide Area（广域网），75
Wireless（无线网），2
Networked System（网络化系统），xxvii, 2, 6-23, 44, 158, 176, 286, 287, 292, 311, 394, 506, 510, 513, 515, 520, 521, 524, 526, 528
Networking（连网），xiv, 366, 505, 524
Idiosyncrasy（特质），62
Non-functional Property（非功能性属性），555

O

Object（对象），20, 110, 139, 353, 555
Access of（的访问），139, 370, 372, 401, 478, 479
Atomic（原子的），478
Concurrent（并发的），400, 401
Scheduled（调度的），400
Synchronized（同步的），400, 401
Access to（访问），40
Composition of（合成），299
Creation of（的建立），55, 356, 552
Destruction of（销毁），55, 356
Idempotent Initialization of（的等幂初始化），354, 551
Interface of（的接口），401
Intra-object Method Call within（内的对象间方法调用），346, 347, 350
Invariant of（的不变式），401
Just Once Initialization of（的仅仅一次初始化），356
Manipulation of（操纵），324
Responsibility of（响应能力），73, 351
Self-deadlock of（自死锁），331, 407
Serialization of（的串行化），324, 419
State of（的状态），20, 400, 401, 478
Type of（的类型），478
Active（主动的），370, 371, 372, 375, 384
Composite（组装的），40
Concurrent（并发的），324, 370
Global（全局的），355, 356, 503, 504
Logically Global（逻辑全局的），477, 478, 479, 484, 489, 492
Opaque（不透明的），284
Passive（被动的），370, 396, 420
Physically Global（物理上全局的），478
Physically Local（物理上局部的），478, 489
Remote（远程），398
Shared（共享的），213, 367, 370, 419
Singleton（单件），40
Thread-specific（针对具体线程的/与线程有关的/线程特定），478
Object Management Group（对象管理组），548
Object Model（对象模型），173
Object Request Broker（对象请求代理），94, 104, 109, 110, 112, 129, 132, 133, 135, 207, 210, 280, 320, 441, 470, 534, 555
Client-side（客户端），112, 117, 119
Communication Mechanism within（内的通信机制），134
Core of（内核），207
Event Loop of（的事件循环），207
Fault Tolerant（容错），134
Flexibility of（的灵活性），134
Request Processing within（内的请求处理），134
Server-side（服务器端），112, 117, 119
ObjectSpace（对象空间），70
Observer（观察者），91, 124, 126, 127, 139, 213, 259
ODBC Database Toolkit（ODBC数据库工具箱），57
OMG。参见Object Management Group
On-the-wire Protocol（在线协议），555
Open Implementation（开放实现），64, 72, 111, 121, 514
OpenDoc，171
OpenGL，502
Open-source Software（开放源码软件），540
OpenSSL，12
Operating System（操作系统），xiv, 3, 10, 16, 35, 44, 88, 103, 180, 184, 192, 211, 216, 232, 235, 238, 243, 257, 298, 423, 438, 444, 449, 451, 458, 486, 500, 556
Application Programming Interface of（的应用编程接口），49, 66, 73

Defect of（的不足），47
Diversity of（的多样性），28
Feature of（的特性），47
Interrupt Handling within（内的中断处理），255
Kernel of（的内核），36, 103, 220, 426, 430, 431, 473, 556
Limitation of（的限制），18
Native Call to（本身就有的对……的调用），70
Service of（的服务），28
Thread Scheduler within（内的线程调度程序），396, 404, 408, 458
Threads Library of（的线程库），480, 485
Type of（的类型），333
Microkernel（微内核），439
Multi-threaded（多线程的），333, 479, 503
Real-time（实时），18, 28, 34, 485
Single-threaded（单线程的），333
Optimization Principle Pattern（最优化原则模式），536
ORB。参见Object Request Broke
ORBacus，210
Orbix，133
Filter（过滤器），134
MT-Orbix，441
Organizational Pattern（组织模式），540
Originator（发起者），284
Out-of Band（带外），556

P

Package（包），51
Packet（包，信息包），556
Parallelism（并行性），212, 371, 395, 556
Parameter（参数），556
Parameterized Type（参数化类型），xxix, 72, 299, 302, 303, 556
Pascal，57
Passive Object（被动对象），370, 396, 420, 556
Pattern（模式），xiii, xxvi, 4, 27, 505, 506-509, 520, 528, 534, 556
Applicability（可应用性），525
Categorization of（分类），xxvi
Combination of（组合），525
Decomposition of（分解），508, 509, 537
Documentation of（的文档化），539
Experience Report on（经验报告），532
Formalization of（的形式化，定形），531, 539
Inter-relationships between（之间的内部关系），507, 508, 509, 510, 520, 525
Is-Used-By（被使用），510
Transitive（传递），510
Uses（使用），510
Introduction to（介绍），xxvi
Organization of（的组织），506, 508, 510, 524, 526
Property of（的属性），524
Pattern Almanac（模式年鉴），530
Pattern Community（模式团体），74, 532
Pattern Conference（模式会议），528
ChiliPLoP，528
EuroPLoP，528
KoalaPLoP，528
PLoP（Pattern Languages of Programs），528
Pattern Form（模式形式），xxvi, xxvii
Pattern Language（模式语言），xv, xxvii, 4, 505-526, 529, 537, 556
Entry Point Pattern（入口点模式），508, 510
Intermediate Pattern（中间模式），508, 510
Leaf Pattern（叶子模式），508, 510
Pattern Paradigm（模式范型），532
Alexander-oriented（面向Alexander的），532
Anti-patterns（反模式），532
Engineering-oriented（面向工程的），532
Pattern System（模式系统），524-526, 529
Classification Scheme of（的分类方案），524
Domain-specific（领域相关的），526
Problem Category of（的问题分类），524
Universal（通用的，全体的），526
Pattern Writer（模式记录器），508
Pedagogical Pattern（教学法模式），540
Peer（对等层，对等体），15
Peer-to-peer（对等，端到端），556
Performance（性能），6, 7, 14, 18, 19, 21, 27, 31, 32, 34, 72, 78, 169, 176, 213, 215, 216, 217, 257, 261, 262, 335, 351, 363, 366, 367, 396, 400, 423, 424, 425, 426, 437, 440, 442, 443, 444, 447, 450, 464, 472, 473, 478, 493, 532
Bottleneck（瓶颈），138
Degradation of（退化），72, 477
Enhancement of（改善），350, 471
Measurement of（度量），532
Optimization of（优化），532
Overhead of（的开销），107, 400
Persistence（持久性/持久状态），133, 153

Piecemeal Growth（逐渐增长），54, 110
Pipe（管道），9, 76, 139
Platform（平台），18, 55, 557
  Hardware（硬件），18, 50
  Heterogeneous（异构的），18, 268
  Homogeneous（同构的），267
  Software（软件），18
Polymorphism（多态性），xxix, 55, 73, 142, 172, 173, 189, 557
Port（端口），557
  Number（号），557
Portability（可移植性），11, 23, 28, 44, 47, 50, 51, 60, 64, 66, 69, 71, 73, 192, 207, 211, 232, 256, 321, 333, 355, 495, 503, 517
Portland Pattern Repository（Portland模式仓库），539
POSIX，xiv, 11, 18, 28, 56, 60, 237, 255, 256, 280, 360, 439
  aio_suspend()，238
  Asynchronous I/O（异步I/O），11, 432
  Condition Variable（条件变量），33, 403
  Mutex（互斥），403
  Network Programming（网络编程），53
  once Variable（一次性变量），360
  Pthreads，xiv, 10, 18, 29, 63, 487, 488, 489, 491, 495
    _POSIX_THREAD_KEYS_MAX，487
    Object_set_（常数），488
    pthread_exit()，490
    pthread_getspecific()，491
    pthread_key_create()，360, 489
    pthread_key_t()，489
    pthread_self()，488
    pthread_setspecific()，491
    phread_exit_hook()，489
    thread_state，488
  Real-time（实时），235
Predictability（可预测性），18, 106, 441, 536
Preprocessor（预处理器），58
Priority Inversion（优先级逆转），433, 465, 472, 557
Proactor（主动器），177, 218, 222, 233, 252, 257, 513
  Access of（访问），236
  Behavior of（的行为），224
  Completion Handler Dispatching within（内的完成处理程序分配），224, 228, 233, 235, 237, 240, 242, 245, 251, 252
  Configuration of（的配置），237, 257
  Event Demultiplexing within（内的事件多路分解），224, 230, 233, 235, 237, 239, 242, 244, 252
  Event Loop of（的事件循环），221, 224, 235, 236, 245, 251
  Handle Registration with（向……的句柄注册），236
  Hierarchy of（的层次），237
  Implementation of（的实现），237-242, 256
  Interface of（的接口），235-242, 256
  Management of（的管理），232
  Number of（的数量），242
  Structure of（的结构），221
  Type of（的类型），237
    Multi-threaded（多线程的），242
    Single-threaded（单线程的），226, 251
    Singleton（单件），236, 242, 244
Process（进程），16, 17, 20, 28, 44, 75, 77, 93, 106, 211, 270, 298, 378, 425, 426, 429, 430, 437, 485, 515, 557
  Address Space of（的地址空间），17
  as a Unit Of Protection（作为一个保护单元），17
  as a Unit Of Resource Allocation（作为一个资源分配单元），17
  Blocking of（阻塞），31, 212, 369, 370, 400
  Kernel Mode of（的内核模式），436
  Management of（管理），xxix, 29
  Multi-processing（多处理），32, 93, 430
  Process Pool（进程池），93
  Restart of（的重启动），75, 76, 515
  Scheduling of（的调度），430
  Shut Down of（的关闭），75, 515
  Single-processing（单处理），3
  Spawning of（的创建），430
  Synchronization of（的同步），430
  Termination of（的中止），76
Process Pattern（进程模式），529, 540
Producer（生产者），32, 445
Product Market-share（产品市场份额），50
Productivity（生产力），2, 10
Programming（编程/程序设计），49
  Asynchronous（异步的），424, 425
  Complexity（复杂性），173
  Concurrent（并发的），20, 425
  Error（错误），11, 50, 61, 71
  Model（模型），419
  Object-oriented（面向对象的），xxix, 523

Overhead（开销），400
Paradigm（范型），401
Sequential（顺序的），20
Simplification（简化），19
Synchronous（同步的），424, 425, 429
Programming Language（编程语言），151, 173
Limitation of（的限制），72
Non-object-oriented（非面向对象的），56, 83
Object-oriented（面向对象的），49, 169
Programming Pattern参见Idiom
Project Spectrum（Spectrum项目），210, 281, 320
Protocol（协议），25, 458, 557
Bytestream-oriented（面向字节流的），456
Chain（链），107
Communication（通信），429
Configuration（配置），107
Connection-oriented（面向连接的），47, 179, 285, 286, 369, 400
Handler（处理器/处理程序），425
Message Passing（消息传递），430
Peer-to-peer（对等），15
Processing（处理），8, 107
Record-oriented（面向记录的），456
Stack（栈），557
Prototype（原型），49
Proxy（代理），371, 372, 373, 379, 382, 384, 389, 390, 392, 397, 398, 516
Behavior of（的行为），374
Data of（的数据），492, 493
Implementation of（的实现），377
Interface of（的接口），371, 373, 377, 492, 493
Logic within（内的逻辑），391
Structure of（的结构），371
Type of（的类型），386
Abstract（抽象），492
Client-side（客户端），169, 393
Local（局部），386, 413
Multiple Roles（多角色），389
Remote Proxy（远程代理），521
Role-specific（与角色有关的），516
Server-side（服务器端），393
pSoS，503
Publisher（出版者），139

Q

QoS。参见Quality Of Service
Quality Object（质量对象），135
Quality Of Service（服务质量），3, 76, 196, 232, 242, 369, 370, 535, 557
Best-effort（最佳效果），3
CPU Speed（CPU速度），536
End-to-end（端到端），3, 31, 369, 536
Latency（延迟），76, 536
Memory Access Time（存储访问时间），536
Network Bandwidth（网络带宽），536
Power level（功能层），536
Stringent（严厉的/迫切的/必须遵守的），3, 535
Throughput（吞吐量），76

R

Race Condition（竞争条件），8, 18, 19, 33, 90, 93, 205, 346, 354, 355, 360, 401, 402, 408, 418, 433, 443, 450, 456, 459, 476, 477, 478, 486, 492, 495, 496, 516, 558
Reactivation（重新激活），22
Reactor（反应器），30, 176, 181, 184, 186, 187, 188, 190, 207, 211, 440, 443, 512
Behavior of（的行为），184
Demultiplexing Table of（的多路分解表），190, 191, 193, 205, 206, 207
Event Demultiplexing within（内的事件多路分解），183, 191, 195
Event Handler Dispatching within（内的事件处理程序分配），183, 191, 194, 195, 199, 200, 202, 212
Event Handler Registration with（向……注册事件处理程序），183, 184, 189, 190-191, 198, 205, 207
Event Loop of（的事件循环），185, 189, 191, 198, 204, 206, 207
Hierarchy of（的层次），191
Implementation of（的实现），191-195, 211
Initialization of（的初始化），206
Interface of（的接口），183, 189-191, 211
Notification of（的通知），206
Number of（的数量），195
Re-entrancy of（的重入），207

State of（的状态），205
Structure of（的结构），183
Thread-safety of（的线程安全性），204
Timer Mechanism of（的定时器机制），208
Type of（的类型），186, 191
Multi-threaded（多线程的），205, 211
Platform-specific（针对具体平台的），189
Single-threaded（单线程的），186
Singleton（单件），195
Redundancy（冗余），447
Re-entrancy（重入），443
Refactoring（再分解），54, 152, 154, 156, 509, 540, 558
Reference Counting（引用计数），207
Reflection（映像），140, 536
Lightweight（轻量级），140
Register（注册），17
Reification（具体化），18, 558
Relationship（关系），558
Reliability（可靠性），6, 21, 23, 106, 261, 447
Remote log-in（远程登录），9, 17
Remote Procedure Call（远程过程调用），23
Replication（复制），7, 22, 23, 129, 136
Active（主动的），22
Framework（框架），22
Passive（被动的），22
Replica Manager（复制品管理器），22
Request（请求），16, 32, 47, 224, 263
Forwarding of（转发），138
Issuing of（发出/发布），397
Ordering of（预订），473
Priority of（优先级），36
Processing of（处理），31, 32, 38, 217, 367, 397
Asynchronous（异步的），217
Concurrent（并发的），32
Iterative（迭代），216
Proactive（主动的），38, 176, 217, 222
Reactive（反应的），30, 31, 176, 184, 207
Simultaneous（同时的），370, 448
Synchronous（同步的），32
Receiver of（接收者），138
Retransmission of（再发送），22
Sender of（发送者），138
Resource（资源），45, 180, 433, 523
Acquisition of（获取），50, 60, 332, 523
Activation of（激活），157
Consumption of（消耗），77, 355
Deactivation of（使……无效，停用），157
Leak（泄漏），357
Limited（受限的），495
Management of（管理），19, 66, 157
Manipulation of（处理），326
Release of（的释放），50, 60, 81, 151, 294, 332, 523
Shared（共享的），20, 158, 326, 355, 366, 367
Utilization of（利用），78
Resource-Acquisition-is-Initialization（资源获取即初始化），325
Responsibility（责任），49, 558
Responsiveness（响应），18, 181, 212, 251, 393, 429
Reusability（可重用性），18, 72, 105, 110, 137, 186, 197, 210, 256, 312, 321, 342, 376, 390, 416, 420, 502, 540
Robustness（健壮性），5, 6, 18, 28, 44, 47, 49, 51, 71, 73, 90, 91, 93, 268, 269, 321, 324, 331, 350, 416, 430, 517
Rogue Wave（异常波），70
Net.h++，70
Threads.h++，70, 331
Role（角色），52, 143, 144, 150, 516, 517, 558
Run-time Stack（运行时栈），17

S

Scalability（可扩展性），6, 7, 11, 18, 27, 31, 32, 93, 176, 180, 181, 192, 195, 216, 217, 252, 262, 420, 441, 449, 536
Scheduler（调度程序），18, 371, 373, 378, 379, 380, 384, 392, 393, 395, 396, 397, 400, 420, 440, 516, 559
Activation of（激活），18
Behavior of（的行为），374
Event Loop（事件循环），516
Implementation of（的实现），381-384
Interface of（的接口），381
Method Dispatching by（进行的方法分配），374
Structure of（的结构），372
Type of（的类型），18
Integrated（集成的），389
Message Passing Based（基于消息传递的），390
Real-time（实时），18
Scheme（方案，图式），57, 260
Scripting（编写脚本），78, 80, 86
Security（安全性），12, 93, 106, 109, 110, 119, 131, 133, 532, 534, 536

Self deadlock（自死锁），40, 205, 324, 345, 346, 350, 351, 407, 516, 518
Semaphore（信号灯），20
SEP，559
Separation Of Concerns（事务分离），106, 136, 172, 210, 256, 321, 389, 396, 401, 443
Serialization（串行化），21, 158, 176, 204, 205, 211, 324, 325, 349, 355, 356, 372, 378, 401, 402, 420, 422, 443, 447, 468, 473, 516, 559
Servant（服务者），371, 372, 373, 377, 378, 379, 381, 384, 385, 389, 390, 393, 397, 398, 516, 559
  Behavior of（的行为），373, 374, 375
  Critical Section within（内的临界区），376
  Implementation of（的实现），373, 375
  Logic within（的逻辑），391
  Predicate of（的断言/的谓词），373, 375, 378
  State of（的状态），373, 374, 375, 383
  Structure of（的结构），373
Server（服务器），16, 37, 45, 110, 169, 173, 180, 217, 286, 288, 299, 398, 467, 469, 472, 512, 513, 559
  Failure of（失效），22
  Request Processing within（的请求处理），472
  Response of（的相应），22
  Restart of（的重启），22
  Role of（的角色），16
  Status of（的状况），22
  Termination of（中止），22
  Type of（的类型），134
    Back-end（后端），447
    Communication（通信），447
    Database（数据库），447, 466
    Event-driven（事件驱动），179
    Front-end（前端），447
    Multi-threaded（多线程的），134, 180
    Replicated（重复的），136
Service（服务），45, 51, 52, 53, 109, 112, 181, 182, 210, 217, 219, 227, 256, 263-267, 270, 271, 286, 311, 366, 369, 423, 425, 434, 450, 452, 478, 511, 513, 514, 516, 517, 554, 559
  Access of（访问），xvi, xxvi, 5, 9-13, 43, 370, 520, 524, 525
  ACT Holding within（在服务内拥有异步完成标记），270
  Blocking of（阻塞），401, 426, 429, 433
  Chain of（服务链），277
  Closure of（闭包），377
  Communication with（与……的通信），430
  Configuration of（的配置），xvi, xxvi, 5, 9-13, 43, 78, 515, 524, 525
  Connection to（到……的连接），30, 177, 285, 287
  Context of（语境），371, 377
  Control of（的控制），78, 515
  Cooperation of（与……的协作），177, 285
  Dispatching of（分配），374
  Evolution of（演化），12, 44, 136
  Failure of（失效），269, 283
  Implementation of（的实现），45, 180, 286, 430-432
  Initialization of（的初始化），7, 30, 177, 199, 285, 286, 287, 514
  Integration of（的集成），109, 136, 181, 217
  Interface of（的接口），347
    Abstract Method（抽象方法），545
    Implementation Method（实现方法），347, 348, 349, 351
    Interface Method（接口方法），347, 348, 349, 351
  Invocation of（调用），213, 259, 265, 324, 345, 351, 366, 369, 371, 373, 374, 400, 435, 482, 483, 516, 518
    Asynchronous（异步的），385, 517
    Concurrent（并发的），400, 434, 435, 443
    Timed（定时的），392
  Invocation Protocol of（调用协议），145
  Lifecycle of（的生命期），45
  Location of（的位置），22, 23, 45
  Naming of（命名），22
  Notification of（的通知），438, 439
  Parameter of（的参数），371, 377, 393
  Partitioning of（的划分），8
  Processing of（的处理），6, 16, 30, 33, 148, 177, 197, 199, 210, 213, 243, 259, 285, 286, 287, 294, 295, 296, 347, 366, 369, 371, 373, 385, 400, 401, 422, 423, 434, 435, 483, 511, 512, 513, 514, 516
    Asynchronous（异步的），19, 31, 217, 270, 366, 395, 423, 424, 425, 429, 431, 434, 435, 445, 511
    Concurrent（并发的），366, 399, 513, 515, 516
    Iterative（迭代），512
    Synchronous（同步的），19, 31, 180, 270, 366, 423, 424, 425, 429, 434, 435, 438, 439, 440, 441, 445, 511
  Property of（的属性），23
  Proxv for（代理），271
  Removal of（的删除），136
  Request for（请求），176, 179, 180, 215, 366, 447, 449, 512, 513, 514, 515
  Result of（的结果），148, 183, 264, 266, 268, 282, 371,

373, 377, 432, 513, 516
Role of（的角色），44
Scheduling of（的调度），33, 366, 370, 396, 399, 400, 422, 516
Selection of（的选择），22
Semantically-related（语义相关的），144
Synchronization（同步），401
Synchronization of（的同步），366, 370, 422
Type of（的类型），288
End-to-end（端到端），288, 311, 514
Layered（分层的），136
Long-duration（耗时长的），212, 216, 217, 218, 227, 251, 257, 370, 425, 429, 430, 447
Networked（网络化的），292
One-way（单向），374, 555
Out-of-band（带外），45, 111, 139, 514
Peer（对等体），286, 514
Remote（远程），45
Semantically-related（语义相关的），144
Short-duration（耗时短的），213, 251, 425, 429, 431
Two-way（双向），374, 379, 467, 562
Service Handler（服务处理程序），197, 211, 213, 243, 287, 298, 321, 507, 509, 513, 514, 515, 521
Behavior of（的行为），293-296
Communication with（与……的通信），299
Concurrency Strategy for（的并发策略），311
Configuration of（的配置），299
Connection to（到……的连接），287, 289, 294, 295, 296, 304, 306, 321
Cooperation Protocol for（的协作协议），311
Creation of（的建立），289, 293, 299, 300, 304, 311, 312
Customization of（的定制），299
Data Exchange with（与……的数据交换），289
Deletion of（的删除），294
Implementation of（的实现），311
Initialization of（的初始化），287, 288, 289, 290, 291, 293, 294, 295, 296, 299, 300, 301, 304, 306, 309, 321, 515
Interface of（的接口），299, 311
Hook Method（钩子方法），288, 290, 291, 293, 295, 296, 308
IPC mechanism for（IPC机制），299
Notification by Dispatcher（分配器的通知），292
Processing within（内的处理），288, 294, 295, 296, 299, 321, 515
Registration with Dispatcher（向分配器注册），292
Structure of（的结构），288
Service Provider（服务提供者），15
Shared Library（共享库），515
Shared Memory（共享内存），9, 76, 84, 109, 444, 470, 559
Siemens（西门子），70
FlexRouting，394
REFORM，70
Syngo，394
Signal（信号），54, 433
Signal Handler（信号处理程序），16, 17, 425, 439
Simplicity（简单性），122, 257, 282, 312, 351, 395, 418, 419, 424, 425, 440, 442, 472
Singleton（单件），155, 355, 358
Canonical Implementation of（的典型实现），353
Constructor of（的构造函数），354
Initialization of（的初始化），353, 363
Method invocation on（上的方法调用），353
Per-thread Singleton（每个线程一个单件），504
Pre-initialization of（的预初始化），359
Template Adapter for（模板适配器），358
Thread-safety of（线程安全），354
Skeleton（构架），393
Smalltalk，73, 529
SMTP（简单邮件传输协议），9, 423
Socket（套接字），xiv, 9, 10, 11, 16, 28, 29, 34, 44, 47, 54, 60, 61, 66, 67, 70, 73, 90, 94, 185, 196, 199, 201, 233, 243, 292, 298, 303, 312, 321, 430, 436, 438, 441, 475, 476, 560
Acceptor（接受器），50
Application Programming Interface(应用程序编程接口），51, 60, 288, 456, 458
Concurrent Accept Function（并发接受函数），34, 35
Connection（连接），62, 206
Datagram（数据报），436
Endpoint（端点），11, 239
Handle（句柄），11, 29, 34, 35, 38, 47, 61, 69, 71, 181, 182, 198, 199, 218, 227, 288, 289, 448, 449, 457, 466, 467, 476
Handle Factory for（的句柄工厂），47
Layer（层），427
Loopback Connection（回环连接），76
Operation on（对…的操作），29
Serialization of（的串行化），456
Stream（流），436
Type of（的类型），49
Data-mode（数据模式），34, 47, 288, 289, 298, 549

I/O（输入/输出），17
Passive-mode（被动模式），11, 34, 35, 47, 200, 201, 245, 288, 298, 556
Software Architecture（软件体系结构），4, 5, 19, 110, 112, 425, 508, 509, 560
Layered（分层的），14
Multi-tier（多层的），447
Software Development（软件开发），109, 149, 508
Cost of（的代价），2, 50, 536
Cycle-time of（的循环周期），536
Effort of（效率），2
Lifecycle of（的生命期），2, 77
Method for（方法），530, 538
Process（过程），508
Tool for（工具），530, 538
Software System（软件系统），9, 19, 111, 112, 114, 115, 176, 181, 183, 184, 195, 222, 235, 263, 334, 366, 371, 400, 425, 508, 528, 546, 560
Behavior of（的行为），106, 136, 140
Blocking of（的阻塞），181
Compilation of（编译），44, 75, 76, 515
Complexity of（复杂性），50
Configuration of（的配置），76, 536
Control of（的控制），536
Event Loop of（的事件循环），183, 204, 209, 233, 522
Evolution of（的演化），44, 49, 75, 76, 111, 140, 515
Failure of（的失效），3, 8
Family（族/系列），150, 508, 560
Initialization of（的初始化），328
Introspection of（反省），140
Linking of（连接），44, 75, 76, 515
Logic of（的逻辑），136, 140
Monitoring of（的监控），536
Property of（的属性），140
Component-based（基于组件的），xv, 77, 117, 119, 165, 536
Concurrent。参见Concurrent System
Decentralized（分散的），7
Distributed（分布式的），45, 107, 109, 117, 119, 131, 148, 169, 180, 217, 285, 369, 469, 475, 520, 528, 535, 536
Embedded（嵌入的），528, 534, 536
Event-driven（事件驱动的），14, 37, 176, 177, 180, 192, 210, 215, 216, 217, 263, 449, 512, 513
Fault-tolerant（容错的），22, 528
Interactive（交互的），xv
Mobile（移动的），535
Modular（模块），7
Multi-threaded（多线程的），205, 324, 346, 354, 476, 478
Networked。参见Networked System
Performance-sensitive（对性能敏感的），423
Reactive（反应性的），192
Real-time（实时），14, 19, 106, 196, 464, 528, 534, 536
Remote（远程），218, 225
Sequential（顺序的），324
Short-lived（生命期短的），49
Single-threaded（单线程的），204, 212, 476, 477, 478
Stand-alone（独立的），6
Transactional（事务的），535
Requirement for（对..的需求），142, 149, 150, 334, 520
Semantics of（的语义），140
Structure of（的结构），140
Subsystem of（的子系统），19, 49, 73, 349, 560
Solaris，29, 34, 48, 51, 65, 103, 232, 360, 439
Condition Variable API（条件变量API），59
Errno（错误号），500
Mutex API（互斥API），57, 60
Synchronization Function（同步函数），57
Threading API（用于线程的API），29, 47, 50, 63
Threads（线程），491, 495
SPX，287
SSL（安全套接字层），110
Starvation（饿死），106, 396, 478, 560
State Machine（状态机），117, 118, 120, 138
Storage Management（存储器管理），7
Structured Programming（结构化编程），xxviii
Stub（桩程序/存根），393
Subject（主题），91, 124, 126, 127, 139, 213, 259
Subscriber（发行者），139
Supplier（供应者），59, 369, 386, 397, 399, 413
Symmetry（对称），136
Synchronization（同步），xvi, xxvi, xxix, 8, 19-21, 32, 47, 54, 180, 196, 212, 217, 257, 262, 323, 331, 347, 353, 366, 369, 370, 376, 393, 395, 396, 399, 400, 401, 406, 407, 413, 441, 443, 448, 450, 464, 467, 471, 485, 489, 502, 516, 519, 520, 524, 525, 532, 560
Boundary（边界），401
Constraint（约束），372, 378, 381, 395, 396, 473

Mechanism（机制），58, 70, 324, 333, 336, 342, 380, 401, 408, 433, 519
Objectified（客体化的），334
Parameterized（参数化的），334
Overhead of（的开销），351
Primitive（原语），19
Strategy（策略），376, 383, 519
Multiple-readers/single-writer（多阅读器/单记录器），383
Synchronized Method（同步化方法），401, 402, 403, 407, 420, 516
Behavior of（的行为），404
Blocking of（的阻塞），404
Execution of（的执行），401, 402, 403, 404, 420
Implementation of（的实现），420
Invocation of（的调用），402, 404, 415
Resumption of（的恢复），401, 403, 405, 408, 420, 516
Scheduling of（的调度），403, 408
Serialization of（的串行化），404, 408
Structure of（的结构），402
Suspension of（的挂起），401, 403, 404, 408, 420, 516
Synchronous Completion Token（同步完成标记），279
Synchronous Event Demultiplexer（同步事件多路分解器），181, 182, 183, 184, 191, 198, 206, 207, 211, 213, 512
Blocking（阻塞），182
Concurrent（并发的），206
poll()，191, 192
select()，182, 191, 192, 197
Selection（选择），192
WaitForMultipleObjects()，191, 192
Synchronous Operation（同步操作），37
System Library（系统库），3, 49

T

TAO。参见ACE ORB
TCP。参见Transmission Control Protocol
TELNET（远程登录），9, 10, 287, 294, 423, 436
Testability（可测试性），105
The ACE ORB，xvii, 133, 210, 280, 320, 470
Thin Client（瘦客户机），6
Thread Of Control（控制线程），xiv, 16, 17, 19, 20, 28, 32, 44, 47, 54, 58, 63, 66, 67, 70, 73, 93, 106, 195, 205, 206, 211, 212, 213, 257, 262, 270, 298, 324, 358, 365-504, 516, 517, 518, 523, 526, 561
as a Unit Of Execution（作为执行单元），17
Blocking of（的阻塞），29, 35, 182, 218, 291, 295, 355, 384, 404, 464, 469, 470
Cancellation of（的取消），18, 64
Cooperation between（互操作），419
Coordination of（的协调），450
Dispatching of（分配），465
Exit Hook for（的exit钩子），489, 495
Exit Status of（的exit状态），50
Handle/Thread Association（句柄/线程关联），467
Bound（受限），467, 471
Hybrid（混合），469
Unbound（不受限），467, 471
Identifier of（的标识符），484, 485, 486
Joining of（的连接），64
Management of（的管理），xxix, 11, 29
Multi-threading（多线程），17, 27, 31, 35, 93, 180, 181, 216, 217, 242, 257, 262, 263, 294, 311, 331, 334, 346, 353, 355, 366, 396, 430, 439, 447, 448, 449, 450, 472, 475, 476, 478, 512, 518
Hazard（危害），18
Synchronous（同步的），36
Notification of（的通知），457, 468
Overhead of（的开销），28, 34
Priority of（的优先级），196, 232, 242, 464
Programming with（用……编程），49
Resumption of（的恢复），59, 408
Scheduling of（的调度），8, 34, 430, 464
Shut Down of（的关闭），18
Single-threading（单线程），3, 17, 31, 93, 186, 195, 252, 263, 311, 447, 476, 478
Spawning of（的产生），49, 63, 293, 430
Stack of（的栈），464
Suspension of（的挂起），33, 58, 60, 408
Synchronization of（的同步），430
Threading Model（线程模型），34
Thread Pool（线程池），33, 34, 35, 93, 216, 252, 367, 393, 448, 450, 453, 454, 460, 512, 526, 561
Thread-per-Connection（每个连接一个线程），180, 294, 475, 561
Thread-per-Request（每个请求一个线程），561
Thread-specific Data of（的与线程有关的数据），17
Thread-specific Storage of（的线程特定的存储器），18, 480
Type of（的类型），33

Consumer Thread（消费者线程），33
Detached（分开的），18, 50, 63
Follower Thread（追随者线程），35, 451, 453, 512
Leader Thread（领导者线程），35, 451, 453, 512
Preemptive Thread（抢先的线程），353, 476
Processing Thread（处理线程），35, 451, 453, 471, 512
Producer Thread（生产者线程），33
Thread Pool（线程池），450, 454, 457, 466, 467, 468, 472
Behavior of（的行为），454
Creation of（的建立），462
Follower Thread within（追随者线程），451, 453, 454, 462, 464, 467, 470, 471, 472
Multiple（多个），469
Promotion（提升），454, 459, 460, 463-465, 472,
Implementation of（的实现），460
Leader Thread within（领导者线程），451, 453, 454, 456, 462, 463, 464, 467, 470, 471, 472
Multiple（多个），469
Selection（选择），459, 460, 463
Processing Thread within（内的处理线程），451, 453, 455, 462, 465
Promotion Protocol of（提升协议），464
Implementation-defined（已定义的实现），465
Last-in, First-out（后进先出），464
Priority-based（基于优先级的），464
Structure of（的结构），453
Synchronizer within（内的同步器），453, 455, 460, 462, 468
Thread Rejoinment（线程再联合），455, 462, 469, 472
Thread-safety（线程安全），66, 324, 345, 353, 396, 400, 402, 418, 513, 514, 516, 519
Thread-specific Object（线程特定的对象），478, 479, 518
Access of（访问），480, 483, 486, 489, 492, 493, 496, 502
Extension Interface（扩展接口），493, 497
Smart Pointer（智能指针），493, 497
Behavior of（的行为），482
Creation of（的建立），483, 495, 496
On-demand（按需），496
Destruction of（的析构函数），491
Global Access Point of（全局访问点），367, 475, 518
Identification of（的标识），482, 484, 487
Implementation of（的实现），484
Initialization of（的初始化），495
Interface of（的接口），493, 496, 518
Constructor（构造函数），495
Destructor（析构函数），495
Matrix Analogy for（矩阵类比），482
Method Invocation on（方法调用），493
Overuse of（滥用），503
Per-thread Singleton（每个线程一个单件），504
Retrieval of（获取/检索），486, 490
State of（的状态），486
Type of（的类型），484, 492, 493, 496
Thread-specific Object Proxy（线程特定的对象代理），480, 503, 504, 518
Access of（的访问），493
Behavior of（的行为），482
Creation of（的建立），494
Destruction of（的销毁），494
Implementation of（的实现），491-498
Interface of（的接口），480, 482, 492, 496
Parameterized Type（参数化类型），492, 496
Polymorphism（多态性），492, 496
Matrix Analogy for（矩阵类比），482
Method Invocation on（方法调用），492
Structure of（的结构），480
Thread-specific Object Set（线程相关的对象集），479, 492, 493, 496, 518
Access of（的访问），480, 485
Behavior of（的行为），482
Identification of（的标识），486
Implementation of（的实现），484-491
Interface of（的接口），479
Location of（的定位），484
Thread-external（线程内的），485, 487
Thread-internal（线程外的），486, 487
Matrix Analogy for（矩阵类比），482
Structure of（的结构），479
Throughput（吞吐量），181, 207, 217, 393, 447
Time Service（限时服务），75, 96
Berkeley Algorithm（Berkeley算法），76, 97
Clerk（职员），75, 98
Clock Synchronization（时钟同步），75
Cristian Algorithm（Cristian算法），76, 97
Time Server（时间服务器），75, 96
Timer（定时器），208, 513
Delta-list-based（基于增量表的），208
Heap-based（基于堆的），208

Queue（队列），513
Timing Wheel-based（基于时间轮的），208
Time-to-market（进入市场的时间），5
TLI。参见Transport Layer Interface
Token Ring（令牌环），12
Tool（工具），3, 8, 18, 530, 538
Administration（管理），136
Auto Configuration（自动配置），56
Debugging（调试），109
Forward Engineering（前向工程），530
Limitation of（限制），18
Monitoring（监控），109
Parser Generator（分析程序生成器），88
Pattern-based（基于模式的），530
Reverse Engineering（逆向工程），530
TP4（第四类传输协议），287
Trading（交易），12
Transaction（事务），109, 131, 133, 447
Commitment of（的提交），447
Monitor（监视器），471
Request for（请求），447
Transmission Control Protocol（传输控制协议），9, 15, 24, 47, 61, 94, 179, 201, 206, 216, 285, 287, 289, 369, 400, 430, 434, 435, 437, 449, 456, 457, 475, 561
Connection（连接），369, 387, 400
Flow Control（流控制），369, 387, 400
Full Association（全关联），179, 287
Packet（包），436, 437
Port Number（端口号），179, 285, 288
Transport Endpoint（传输端点），179
Transport Layer（传输层），369
Transport Address（传输地址），47, 285, 289, 290, 293, 298, 303
Transport Endpoint（传输端点），34, 107, 182, 201, 285, 286, 287, 289, 290, 291, 292, 294, 513, 515, 562
Connected（连接的），288
Creation of（的建立），293, 298
Deletion of（的删除），294
Encapsulation of（的封装），289, 298, 299
Initialization of（的初始化），293
Passive-mode（被动模式），288, 293, 298, 301, 321
Selection of（的选择），298
Transport Handle（传输句柄），287, 289, 291, 292, 293, 295, 296, 298, 303
Creation of（的建立），288, 289
Deletion of（的删除），294
Encapsulation of（封装），299
Initialization of（的初始化），289, 290
Transport Layer Interface（传输层接口），30, 51, 94, 292, 303, 321, 562
Transport Layer System（传输层系统），503
Transport Mechanism（传输机制），298
Transport Protocol（传输协议），109, 132
Tuxedo，471
Type-safety（类型安全性），52, 64, 321

U

UDP。参见User Datagram Protocol
UML。参见Unified Modeling Language
Unicode（统一的字符编码标准），562
Unified Modeling Language（统一建模语言），xxviii, 530, 539
Class Diagram（类图），563
Sequence Diagram（顺序图），565
Statechart Diagram（状态图），567
Unified Software Development Process（统一软件开发过程），538
Uniformity（统一性），105
UNIX，11, 13, 18, 28, 60, 90, 103, 182, 193, 211, 254, 470, 476
BSD UNIX，17, 423, 426, 427, 435, 440
I/O Subsystem（I/O子系统），444
Kernel（内核），423, 436, 438
Network Packet Processing（网络数据包处理），431
Networking Subsystem（连网子系统），436
Protocol Processing（协议处理），423, 437
read()，436, 438
sbwait()，437
sbwakeup()，438
Scheduler（调度程度），437
soreceive()，436, 438
Network Daemon（网络精灵/网络守护进程），441
Network Superserver（网络超级服务器），319
Networking Subsystem（连网子系统），441
pipe（管道），206
poll()，211
Processes（进程），10
select()，211

Signal Handler（信号处理程序），16
Sockets（套接字），10
STREAM Pipe（STREAM管道），303
STREAMS（流），441
System V Release 4，88, 192
System V STREAMS，xiv
Upcall（上调），562
Usability（可用性），536
User Datagram Protocol（用户数据报协议），9, 456, 458, 562
User Interface（用户接口），6
User Interface Library（用户接口库），49

V

Validation（确认），424
Variation（变体），54, 55, 71, 73, 396
Versioning（版本控制），12
View（视图），141, 562
Virtual Machine（虚拟机），562
Virtual Memory（虚拟内存），17, 28, 36, 562
Visibroker，134
Visual Basic，171
VLSI（超大规模集成电路），2
VMS，232
VxWorks，18, 503

W

Web Browser（Web浏览器），24, 135, 215, 244, 273, 279, 320
Graphical User interface of（图形用户界面），24
Plug-in for（插件(程序)），135
Web Server（Web服务器），xvi, xxvi, 9, 10, 24-41, 112, 215, 244, 261, 273, 279, 325, 333, 429, 435, 470, 506, 532
Behavior of（的行为），39
Blocking of（阻塞），29, 326
Cached Virtual Filesystem of（缓存的虚拟文件系统），27, 39, 40
Configuration of（的配置），39
Connection Management within（连接管理），30
Design of（的设计），27-41
Event Dispatcher of（的事件分配器），26, 33, 35, 40
Event Loop of（的事件循环），36
Event Processing within（的事件处理），36
File Cache of（的文件缓存），10, 333, 340, 345
File Transmission within（的文件传输），36
Framework for（的框架），26
Hit Count of（的点击率），325
Latency of（的延时），34
Protocol Handler of（的协议处理程序），26, 35, 37, 39, 41
Protocol Processing within（的协议处理），27, 29, 30, 31, 35, 37
Request Queue of（的请求队列），27, 32, 36
Dequeuing（出队列），33
Enqueuing（入队列），33
Synchronized（同步的），33, 34
Robustness of（的健壮性），28
Type of（的类型），24
Concurrent（并发的），24
Multi-threaded（多线程的），325, 340
Single-threaded（单线程的），340
Wide Area Network。参见Network
WikiWikiWeb（允许任何人在任何内容上进行协作的网页），539
Wireless Network（无线网）。参见Network
Workload（工作量），25, 27, 366, 448
World Wide Web（万维网），7, 535, 539
Traffic on（流量），24
Transfer Protocol for（传输协议），26
HTTP/1.0，25, 26
HTTP/1.1，25, 26
HTTP-NG，25, 26
Syntax（语法），24
Wrapper Facade（包装器外观），51, 52, 299
Behavior of（的行为），53
Escape Hatch from（安全舱口），64
Implementation of（的实现），53-67
Level Of Indirection within（间接级别），55
Private Portion of（的私有部分），52
Structure of（的结构），52
Type of（的类型），52
Strongly-typed（强类型的），52
System-level（系统级），66
Writers Workshop（作者专题研讨会），528

WWW。参见World Wide Web

X

X Windows，57, 443, 522
  GUI Event Dispatching within（中的GUI事件分配），53
  Xt toolkit，209, 522
X.500，23
XML 。参见eXtended Markup Language（扩展标记语言）
XP。参见 eXtreme Programming
X-terminal，6

Y

yacc，88, 212