C++网络编程

卷1

C++网络编程　卷 1

C++Network Programming Volume 1

Douglas C. Schmidt Stephen D. Huston

Copyright© 2002 Addison Wesley Longman, Inc.

Simplified Chinese Copyright 2003 of the first publication by Huazhong Science and Technology University Press and Pearson Education North Asia Limited.

All rights Reserved.

Published by arrangement with Pearson Education North Asia Limited, a Pearson Education Company.

版权所有,翻印必究。

本书封面贴有华中科技大学出版社(原华中理工大学出版社)激光防伪标签,封底贴有"Pearson Education"激光防伪标签,无标签者不得销售。

图书在版编目(CIP)数据

C++网络编程　卷 1/(美) Douglas C. Schmidt　Stephen D. Huston 著;於春景 译

武汉:华中科技大学出版社,2003 年 12 月

ISBN 7-5609-3066-2

Ⅰ.C++…

Ⅱ.①D…　②S…　③於…

Ⅲ.C 语言-程序设计

Ⅳ.TP312

责任编辑:曾　光

出版发行:华中科技大学出版社　(武昌喻家山　邮编:430074)

http://press.hust.edu.cn

录　排:华中科技大学惠友科技文印中心

印　刷:湖北新华印务有限公司

开　本:787×1092　1/16　　印　张:20.75　　插　页:2　　字　数:308 000

版　次:2003 年 12 月第 1 版　　印　次:2003 年 12 月第 1 次印刷

ISBN 7-5609-3066-2/TP·517　　定　价:35.00 元

深入 C++丛书

Bjarne Stroustrup，编者

“我把这封信写得比平时要长，因为我没有时间去把它写得更短。”

——BLAISE PASCAL

对 C++程序员来说，ISO/ANSI C++标准的问世标志着一个新纪元的开始。C++标准为程序设计提供了很多新的便利和可能，但要想在如此众多的信息中挖掘到其中的精髓，作为现实世界中的程序员，我们缺少足够的时间。“深入 C++”丛书就是针对一定的主题，为程序员提供了简明扼要的指导，从而将学习时间和疑惑减至最少。

在这套丛书中，每一本书都呈现一个独立的主题，每本书的技术难度也和相应的主题相适应。本丛书所提供的有效途径，有助于每一位专业人员将自己的程序设计水平提升到更高的层次。这些专著由业内专家撰写，篇幅短小但内容深入。在阅读和参考这些专著时，我们不会被无关的内容所打搅。作为一个系列，这些书相互之间交叉引用；此外，它们也都引用了 Bjarne Stroustrup 的著作《The C++ Programming Language》。

在提高 C++程序设计能力的过程中，日益重要的一点是将重要的知识同华而不实的东西分离开来，然后找出更深入、更全面的内容，从而获得进一步的提高。“深入 C++”丛书所展示的 C++程序设计的工具、概念、技巧和新的方法，将为你提供一个重要的开端。

丛书书目

Accelerated C++: Practical Programming by Example, Andrew Koenig and Barbara E. Moo

The Boost Graph Library: User Guide and Reference Manual, Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine

C++ In-Depth Box Set, Bjarne Stroustrup, Andrei Alexandrescu, Andrew Koenig, Barbara E. Moo, Stanley B. Lippman, and Herb Sutter

C++ Network Programming, Volume1: Mastering Complexity Using ACE and Patterns, Douglas C. Schmidt and Stephen D. Huston

Essential C++, Stanley Lippman

Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions, Herb Sutter

Modern C++ Design: Generic Programming and Design Patterns Applied, Andrei Alexandrescu

More Exceptional C++: 40 New Engineering Puzzles, Programming Problems, and Solutions, Herb Sutter

欲获取更多信息，请访问“深入 C++”丛书网站 http://www.aw.com/cseng/series/indepth/

C++网络编程

卷 1

运用 ACE 和模式消除复杂性

C++ Network Programming

Volume 1

Douglas C. Schmidt　Stephen D.Huston　著

於春景　译

马维达　审校

华中科技大学出版社

中国·武汉

序

为本书作序之时，我正在欧洲畅游。这多亏了欧洲完善的公共交通设施。作为一个美国人，我不禁被这里的基础设施所倾倒。无论在什么地方，只要我一下飞机，就可以方便地找到火车或巴士，享受快速、整洁、安全、准时的服务——或许，更重要的是：将我直接送到目的地。此外，离站和到站的通告用多种语言表示，各种标志和指示也易于理解，即使是我这样一个不懂当地语言的人，也没有感到丝毫不便。

我生活、工作在 Boston。和大多数美国人一样，我几乎完全靠自己的汽车辗转于各个场所。除了偶尔光顾一下 Boston 的地铁系统之外，我总是自己开车外出；因为公共交通设施有太多的限制，它很难将我送到目的地。数百万 Boston 居民以及其他地方的人们也都处于同样的困境。可见，我们的公路系统已经过时，远不能承受今天的交通量。我知道，如果仔细算一算自己一生中因为塞车而浪费的时间，一定会吓一跳。

有趣的是，网络计算系统（networked computing system）和交通系统之间有某些相似之处，其中最显著的一点是：二者的成功都离不开“可伸缩（scalable）”的基础设施。所谓可伸缩的交通系统，不只包括那些显而易见的设施，如火车和铁轨、飞机和机场，还需要有调度、路线选择、维修、检票、监控，而这一切还要能够随着物理上的交通系统的变化而伸缩、变化。类似地，网络计算不仅需要主机和网络——物理上的计算和通信基础设施——还需要软件上的调度、路由选择、分发、配置、转译、验证、授权和监控——只有这样，网络系统才能根据需要调节、变化。

说到基础设施，有这样一个具有讽刺意味的事实：建设好基础设施，这本身已经非常之难；但更难的是，只有让用户越感觉不到它的存在，人们才会认为它越成功。举个例子：瑞士境内的阿尔卑斯山脉地势崎岖，但是，为数不多的设计师、工程人员和建筑工人却凭借他们的技能，为无数的瑞士居民提供了有效的交通系统，从而给他们的日常生活带来了便利。事实上，这套系统如此可靠易用，以至于你很快将它的存在当成是理所当然之事。它因此变得对你透明了。譬如，当你踏上瑞士的铁路之旅时，你的心思只会放在该从哪一个地方到达另一个地方上，而不会关心是什么样的交通工具将你载到那儿。除非你是个游客，否则，你大概不会注意到你正在穿越一个历经数年设计、修建的隧道，也不会注意到火车正攀行在如此陡峭的斜坡上，更不会注意到铁路上还安装了齿轨，以帮助列车爬行。铁轨设施完美地承担了它的职能，所以，你甚至没有注意到它的存在。

本书针对网络计算系统，探讨了一种基础设施软件（infrastructure software），通常称为*中间件（middleware）*。之所以称之为“中间件”，是因为它就像是“沙漏的腰（waist in the hourglass）”，位于操作系统和网络之上、应用程序之下。中间件具有多种形态（shapes）、规模（sizes）和能力（capabilities）：从 J2EE 应用服务器、异步消息处理系统、CORBA ORB，到小型嵌入式系统中的 Socket 监控软件。中间件必须支持日益繁多的应用程序、操作系统、网络协议、编程语言和数据格式。如果没有中间件，要想应对网络计算系统中日益增长的多样性（diversity）和异种性（heterogeneity），将十分麻烦，而且容易出错且代价昂贵。

中间件种类繁多，要解决的问题也各式各样，但在解决各种复杂难题时，不同类型的中间件往往采用相同的模式（patterns）和共同的抽象（abstractions）。例如，如果去窥探一个具有可伸缩性和灵活性的应用程序服务器、消息系统或 CORBA ORB 的内部，就会发现，它们采用了类似的技术来完成诸如连接管理、并发、同步、事件多路分离、事件处理程序分发、错误记录、监控等任务。如同瑞士铁路系统的用户数量远远大于设计和建造铁路的工程人员的数量那样，一套成功的中间件的用户数量也远远大于设计和构造中间件的开发人员的数量。如果要设计、构造或使用中间件，那么，只有熟悉、理解和运用这些常见模式和抽象，才会获得成功。

虽然很多人都知道，中间件必须具有可伸缩性和灵活性，但很少有人能够高效地提供这种中间件。然而，Doug Schmidt 和 Steve Huston 在本书介绍的 ADAPTIVE 通信环境（ACE）中却做到了这一点。ACE 是一套应用广泛的 C++工具包，它汇集了很多常用的模式和抽象，这些模式和抽象在各种极为成功的中间件和网络程序中都有广泛的应用。ACE 已经成为许多网络计算系统的基础：从实时的航空电子应用到 CORBA ORB，以及对主机“端对端（peer-to-peer）”通信的支持。

和其他所有优秀的中间件一样，ACE 也将异种环境下各式各样的复杂性隐藏在底层；和其他众多基础设施中间件不同的是：ACE 能够在程序需要的任何地方提供最大的灵活性，但不会损及系统的性能或可伸缩性。作为一名长期从事中间件开发工作的设计师，我清楚地知道，要想在同一个软件包中兼顾性能和灵活性，难度该有多大！

但从某种意义上说，ACE 在灵活性和性能上的优越表现没有出乎我的意料。和 Doug 的长期合作让我清楚地知道，他无愧于这个领域的先锋。当今各种具有可伸缩性、高性能和灵活性的中间件，都明显地留下了他的痕迹，并深受他的影响。Steve 是一位天才的 C++开发者和作家，他的工作使 ACE 多年来取得了长足进步。Doug 和 Steve 联手，为所有设计、构造、甚至使用中间件的人奉献了一部“必读”之作。万维网（World Wide Web）和互连嵌入式系统（interconnected embedded systems）的日益普及，意味着网络计算系统的数量、规模和重要性将持续增长。只要理解了 Doug 和 Steve 在书中讲述的重要模式、技术、class（类）和经验教训，我们就有希望创建出完全透明、高效、可靠的中间件基础设施。

Steve Vinoski
总设计师兼副总裁，Platform Technologies
IONA Technologies
2001 年 9 月

译　　序

计算机网络的普及，给软件开发者带来更大的挑战。硬件设备、软件环境的多样性（variety）和异种性（heterogeneity），使得网络程序设计的复杂性（complexity）大大提高。应用程序作为软件功能的最终实现者，不应当（有时甚至不可能）去直接应对这所有的复杂性。因此，一套专门处理“多平台”差异和编程复杂性的中间件（middleware），对网络程序开发具有重要意义。ACE 就是这样一套优秀的中间件。

本书是 ACE 的缔造者撰写的一部网络中间件专著。它以 ACE 的设计为核心，深入探讨了网络中间件开发中面临的各种挑战，以及设计上的各种选择，并为你展示了 ACE 针对这些问题的解决之道。

本书适合 ACE 的使用者或者（潜在的）ACE 的开发者阅读。本书对 ACE 的能力、ACE 要解决的问题，以及如何通过 ACE 提高网络程序的可伸缩性和可移植性等议题作了详尽的论述。对 ACE 使用者来说，本书是迄今为止最为权威、详尽、系统的 ACE 专著。

但是，本书并不只针对 ACE 的使用者，任何一位网络程序开发者——无论应用程序开发者或中间件开发者——都能从本书受益。作者在书中就网络编程领域存在的复杂性作了全面、细致的阐述，对各种设计方案的利弊作了深入的剖析，并给出了 ACE 的解决之道。这些内容，是 ACE 的开发者十多年来经验的积累，任何一位网络开发者都能从中汲取养分，增添功力。

对 C++程序员和模式爱好者来说，本书也不容错过。ACE 运用了 C++面向对象设计技术，也使用了模板、traits（特征类）等高级语言特性和其他 C++技术；此外，还运用了大量的设计模式。长期以来，高级 C++特性和技术在实际软件开发中的运用、

设计模式的实践性等问题，给不少编程爱好者带来很多困惑和疑问。本书通过现实世界的产品级代码，就这一问题作出了清晰而明确的回答。

虽然本书并不是一本网络编程、C++或设计模式方面的教程，但在阅读本书之前，必须对这三方面的知识有所了解。此外，本书具有很强的实践性，它是作者十多年 ACE 实践经验的总结。就我看来，这也是“深入 C++”丛书中最具实践性的专著之一。

最后要说明的是，《C++网络编程》共两卷，本书是第一卷。这使得我在翻译过程中能有机会同第二卷的译者马维达先生进行多次深入的探讨。我十分高兴能和他共同交流学习。我也期望我们的工作能给各位读者带来帮助。

於春景，2003/04/22

深圳蛇口，海上世界

关于本书

过去 10 年，并发式“面向对象”网络编程（concurrent object-oriented network programming）已经成为一种有效的应用软件开发范式（paradigm）。在这些应用程序中，相互协作的对象可以：

1. 在一个进程或计算机中相互关联；
2. 分布在一组通过网络（如嵌入式互连系统、局域网（LAN）、企业内部网、互联网）相连的计算机中。

如果对象是分布（distributed）存在的，那么，组成这些对象的各种实体（entity）之间就要能够有效地通信和协调。而且，当应用程序在其生命周期内发生变化时，也必须始终这样。对象的布局、现有网络设施、平台并发机制的选择都允许有一定程度的自由，这种自由带来了强大的功能，同时也带来了挑战。

如果设计合理的话，并发式“面向对象”网络编程将提供强大的功能，从而大大提高应用程序的灵活性。例如，根据项目的需求和现有资源，可以使用：

- 实时（real-time）、嵌入式（embedded）或手持（handheld）式系统；
- 个人或膝上型电脑；
- 各式各样、不同规模的 UNIX 或 Linux 系统；
- “大型”主机甚至超级计算机。

然而，当你在多个操作系统（OS）平台上开发和移植网络应用程序时，将面临错综复杂的挑战。这些复杂性的表现形式各异：网络协议不兼容，在不同软、硬件平台上具有不同 API 和语义的组件库，由于 OS 本身的（native）进程间通信（IPC）机制

和并发机制的局限性造成的“*偶发复杂性（accidental complexity）*”。为减小这些问题的影响，ADAPTIVE Communication Environment（ACE）提供了一套“面向对象”工具包。此工具包可运行在大量硬件和 OS 平台上，包括 Win32 和 UNIX 的大多数版本，以及很多实时及嵌入式操作系统。

可能有人告诉过你：一些“事实上”或“官方”的 OS 标准（如 POSIX、UNIX98 或 Win32），是所有程序员的保护伞，它们可以让你的应用程序免受“可移植性”难题的困扰。不幸的是，有这样一句格言：“标准的最大好处是，它们提供了大量的选择”[Tan96]。这一格言在今天比在 10 年前更适用。今天，数十种不同的 OS 平台正应用于商业、学校和政府项目中；每一个新版本和变体的出现，都会使这个排列数增加。

过去 20 年里，我们开发过很多“跨平台”的并发式网络系统，因而，我们可以明确地告诉你：在不同的时期，OS 供应商使用的标准也往往不同。何况，标准也是不断变化和发展的。你很可能工作在多个平台上，而这些平台是在不同的时期、以不同的方式去实现标准的。因而，直接针对 OS API 编程会导致以下两个问题。

1. **容易出错。**因为，用 C 写成的、OS 本身具有的 API 一般都缺乏类型安全（type safe）、可移植（portable）、可重入（reentrant）、可扩充（extensible）的系统函数接口和函数库。例如，在广泛使用的 Socket API（将在第 2 章讲述）中，通信端点是通过“弱类型化（weakly typed）”的“整数”或“指针”型 I/O 句柄（handle）来标识的，这无疑增加了程序运行时产生微妙错误的可能性。

2. **助长“不恰当”的设计技术的运用。**因为，很多用 OS API 写成的网络应用程序都是基于“算法”设计（algorithm design），而不是“面向对象”设计（object-oriented design）的。算法设计需要根据特定的功能需求来分解程序的结构，而功能需求并非一成不变——随着时间的变化，它很可能也会发生变化。因而，这种设计范式（paradigm）导致的软件结构将无法扩充，也无法根据不断变化的应用需求作出快速响应[Boo94]。

在这个经济动荡、缺乏管制、全球激烈竞争的年代，如果完全通过原始的（native）OS API 和算法设计技术来开发应用程序，其昂贵的开销和时间上的浪费将让人无法忍受。

如果你多年来一直从事网络软件系统的开发，或许早已将其中的一些问题视作家常便饭。但是，还有更好的选择！在本书中，我们将向你介绍，C++和 ACE 如何为你提供“面向对象”能力，从而让你尽可能地避开很多陷阱和避免犯错误，同时还不失对标准——甚至，某些“和平台相关”的特性——的支持。随着时间的推移，“面向对象”设计体现出的稳定性将远胜于“算法”设计；因而，在开发多种类型的网络应用程序时，应该优先选用“面向对象”设计。

不必惊讶，这些灵活性也需要代价：可能需要学习一些新的概念、方法、模式（pattern）、工具和开发技术。根据个人经验的不同，你的学习曲线可能是平坦的，也有可能在一开始就显得很陡峭。但重要的是，“面向对象”设计范式（paradigm）为你提供了一套成熟的技术，可以让你应对网络应用程序开发中的很多挑战。本书给出了一系列具体的示例，向你展示如何运用“面向对象”技术开发和应用 ACE 工具包中的类。你可以运用相同的技术和 ACE 类来简化自己的应用程序。

目标读者

本书奉献给“一线”程序员或高年级学生，因为在运用 C++和“面向对象”设计技术进行并发网络编程时，他们需要了解其中的战略、战术。我们将讲述重要的设计空间（dimension）、模式和原则——要想快速方便地开发灵活高效的并发式网络程序，就需要了解它们。我们提供了大量的 C++代码范例，让你强化对设计概念的理解，还具体演示如何尽快使用 ACE 中的核心类。我们还将你带到“幕后”，让你理解 ACE 工具包中的 IPC 和并发机制是如何设计的，以及为什么要那样设计。这些内容将帮助你增强设计技能，在“面向对象”网络程序中更高效地运用 C++和*模式*。

本书并非是讲述“面向对象”开发、模式、UML、C++、系统程序设计或网络技术的详尽教程。所以我们假设，对于以下主题，本书读者应该有一定程度的了解。

- **面向对象设计及编程技术**。例如，框架（framework）[Joh97, FJS99b, FJS99a]、模式（pattern）[GHJV95, BMR+96, SSRB00]、模块化（modularity）[Mey97]、信息隐藏（Information hiding）[Par72]、建模（modeling）[Boo94]。

- **面向对象表示法及处理方式**。例如，Unified Modeling Language（UML，统一建模语言）[RJB98]、eXtreme Programming（极限编程）[Bec00]、Rational Unified Process（RUP，Rational 统一过程）[JBR99]。

- **C++语言的基本特性**。例如，类（class）、*继承（inheritance）*、*动态绑定（dynamic biding）*、*参数化类型（parameterized type）*[Bja00]。

- **核心系统编程机制**。例如，事件多路分离、进程和线程管理、虚拟内存、UNIX[Ste98, Ste99, Ste92, Lew95, KSS96, But97]和 Win32[Ric97, Sol98, JO99]平台上都存在的 IPC 机制和 API。

- **网络术语和概念**。例如，TCP/IP[Ste93]、远程操作请求[Obj01]、客户/服务器架构[CS92]。

我们鼓励你广泛使用参考资料，针对想深入学习的主题去追根溯源。

本书也不是一部针对 ACE 程序员的使用手册，也就是说，我们没有对 ACE 中的每一个类（class）、每一个方法（method）进行说明。如果想达到那样详细的程度，可以查看完整的 ACE 在线文档，这些文档用 Doxygen [Dim01]生成，由 http://ace.ece.uci.edu/Doxygen 和 http://www.riverace.com/docs 提供。相反，本书讲述的重点是：

- 设计成功的面向对象的网络应用程序和中间件需要了解的重要概念、模式和 C++特性；

- 最常用的 ACE TCP/IP 及并发式 wrapper facade 类背后的设计动机及其基本应用。

结构和内容

本书讲述的内容是：如何用 C++和中间件来解决网络应用程序开发中相关的重要课题。首先回顾主流 OS 平台本身具有的核心 OS 机制，并阐释如何在 ACE 中运用 C++和模式，将这些机制封装到类库（class library）wrapper facade 之中，从而提高应用程序的可移植性和健壮性。本书用到的主要例子是一个网络日志服务程序，其功能是：通过 TCP/IP，将日志记录从客户程序传输到一个日志服务器。我们将这个服务程序作为贯穿全书的示例，从而具体演示：

- C++和 ACE 如何有助于实现高效、可预测和可伸缩的网络应用程序；
- 开发并发式“面向对象”网络程序时，在设计和实现上要考虑到哪些因素和方案。

本书分为 11 章，其主要内容简介如下。

- **简介**——第 0 章是有关 C++网络编程的基本介绍。首先，勾勒出问题空间（problem space），提出应用程序运行在单进程、单线程的情况下可能面临的课题。然后，引入中间件层（middleware layer）的分类（taxonomy）概念，并讲述如何运用主机设施中间件和 ACE 工具包来解决常见的网络编程问题。
- **第 1 篇**——第 1~4 章概要介绍通信设计上的选择方案，讲述 ACE 中高效编写 OS IPC 机制时使用的面向对象技术。运用这些技术所得到的类构成了本书中的网络日志服务例程的第一个版本的基础。
- **第 2 篇**——第 5~10 章概要介绍并发设计上的选择方案，并讲述 ACE 中高效编写 OS 并发机制时使用的面向对象技术。

在整个第 1、2 篇中，我们针对那个网络日志服务程序，提供了一系列逐步复杂化的实现，以演示如何将 ACE IPC 和并发 wrapper facade 运用于实践。

附录 A 根据 ACE IPC 和并发 wrapper facade，总结了其底层的“类设计和实现”的原则。附录 B 介绍 ACE 的由来，以及作为开放源码，ACE 在过去 10 年的演变，此

外还描绘了它在未来的远景和发展方向。本书结尾提供了一个技术术语表（包括书中以斜体字形式出现的术语）、一个供你进一步阅读的参考文献的详细列表，以及一个综合性的主题索引。

相关材料

本书讲述的重点是如何通过特定的C++特性、模式和ACE来消除复杂性。这套丛书的第二卷——《C++ Network Programming：Systematic Reuse with ACE and Framework》[SH]——扩展了这些内容，对ACE提供的“面向对象”网络编程框架（framework）进行了说明。这些框架具体化了本书呈现的ACE wrapper facade类的常见使用模式（common usage pattern），支持更广泛、更深层次的系统复用（systematic reuse）。本书介绍的ACE wrapper facade类和卷2介绍的ACE框架类的区别是，ACE wrapper facade类很少有虚方法（virtual method），但ACE框架类中大部分方法都是虚方法。

本书基于ACE 5.2版，此版本发布于2001年10月。ACE软件和书中提供的所有示例程序都是开放源码，可以在http://ace.ece.uci.edu和http://www.riverace.com下载到它们。这些网站还提供了大量和ACE相关的其他资料，如教程、技术论文，以及一篇概括性的文档，此文档就本书没有提到的其他IPC和同步机制ACE wrapper facade作了介绍。我们提倡大家去下载一份ACE的副本，这样，就可以随着我们，将实际的ACE类和框架完完整整看个通透，并在阅读本书的过程中运行相应的示例程序。此外，还有已经编译好了的ACE版本，你可以向http://riverace.com购买，价格微不足道。

如果想更多地了解ACE，或者想报告你在本书中发现的任何错误，可以订阅ACE邮件列表ace-users@cs.wustl.edu。可以发送电子邮件至ace-users-request@cs.wustl.edu，向Majordomo列表服务器订阅邮件。在电子邮件正文中，请包含下面的命令（邮件主题省略）：

subscribe ace-users [emailaddress@domain]

只是当邮件信息中的“From”地址不是你想订阅的地址时，才需要提供emailaddress@domain。

发送给ACE邮件列表的帖子也会被同时转发至USENET新闻组comp.soft-sys.ace；发送给 ACE 邮件列表的所有帖子的历史档案则存储在 http://groups.yahoo.com/group/ace-users 上。

致谢

诚挚的谢意献给以下审阅者：Christopher Allen、Tomer Amiaz、Alain Decamps、Don Hinton、Susan Liebeskind、Dennis Mancl、Patrick Rabau、Eamonn Saunders、Johnny Willemsen。他们审阅了全部文稿并提供了广泛的意见，使本书在形式和内容上都有了实质性的提高。当然，书中遗留的任何问题，责任在我们。

来自世界各地的其他很多 ACE 用户对本书初稿提供了反馈意见，他们是 Mark Apple、Shahzad Aslam-Mir、Kevin Bailey、Barry Benowitz、Emmanuel Croze、Yasir Faiz、Gillmer Derge、Iain Hanson、Brad Hoskins、Bob Huston、Christopher Kohlhoff、Serge Kolgan、Andy Marchewka、Jeff McNiel、Phil Mesnier、Arturo Montes、Aaron Nielsen、Jeff Parsons、Pim Philipse、Yaron Pinto、Stephane Pion、Nick Pratt、Paul Rubel、Shourya Sarcar、Leo Stutzmann、Tommy Svensson、Alain Totouom、Roger Tragin、Reuven Yagel。

我们要感谢 Washington University, St. Louis 和 University of California, Irvine 的所有 Doc group 成员——无论是过去的还是现在的成员，以及 Object Computing Inc.和 Riverace Corporation 的成员，他们都为开发、改进和优化本书讲述的许多 ACE 功能作出了贡献。他们包括：Everett Anderson、Alex Arulanthu、Shawn Atkins、John Aughey、Darrell Brunsch、Luther Baker、Don Busch、Chris Cleeland、Angelo Corsaro、Chad Elliot、Sergio Flores-Gaitan、Chris Gill、Pradeep Gore、Andy Gokhale、Priyanka Gontla、Myrna Harbibson、Tim Harrison、Shawn Hannan、John Heitmann、Joe Hoffert、James Hu、Frank Hunleth、Prashant Jain、Vishal Kachroo、Ray Klefstad、Kitty Krishnakumar、Yamuna Krishnamurthy、Michael Kircher、Fred Kuhns、David Levine、Chanaka Liyanaarachchi、

Michael Moran、Ebranhim Moshiri、Sumedh Mungee、Bala Natarajan、Ossama Othman、Jeff Parsons、Kirthika Parameswaran、Krish Pathayapura、Irfan Pyarali、Sumita Rao、Carlos O'Ryan、Rich Siebel、Malcolm Spence、Marina Spivak、Naga Surendran、Steve Totten、Bruce Trask、Nanbor Wang、Seth Widoff。

我们还要感谢来自世界 50 多个国家、数以千计的 C++开发者，他们在过去 10 年里为 ACE 作出了贡献。ACE 的优秀和成功得益于许多有才华的开发者的精湛技能和高尚品德，也得益于许多高瞻远瞩的公司为 ACE 开放源码库所做的大量工作。没有他们的支持、不断的反馈和鼓励，我们不会写出这本书。为了感谢 ACE 开放源码社区作出的贡献，我们有一份名单，列出了所有的贡献者，提供在 http://ace.ece.uci.edu/ACE-members.html 上。

我们还要感谢来自同事以及支持我们开展模式研究和 ACE 工具包开发的赞助者们，尤其是以下人员：Ron Akers（Motorola）、Steve Bachinsky（SAIC）、John Bay（DARPA）、Detlef Becker（Siemens）、Dave Busigo（DARPA）、John Buttitto（Sun）、Becky Callison（Boeing）、Wei Chiang（Nokia）、Joe Cross（Lockheed Martin）、Lou DiPalma（Raytheon）、Bryan Doerr（Boeing）、Karlheinz Dorn（Siemens）、Matt Emerson（Escient Convergence Group, Inc.）、Sylvester Fernandez（Lockheed Martin）、Nikki Ford（DARPA）、Andreas Geisler（Siemens）、Helen Gill（NSF）、Bob Groschadl（Pivotech Systems, Inc.）、Jody Hagins（ATD）、Andy Harvey（Cisco）、Sue Kelly（Sandia National Labs）、Gary Koob（DARPA）、Petri Koskelainen（Nokia Inc）、Sean Landis（Motorola）、Patrick Lardieri（Lockheed Martin）、Doug Lea（SUNY Oswego）、Hikyu Lee（SoftLinx）、Joe Loyall（BBN）、Mike Masters（NSWC）、Ed Mays（U.S. Marine Corps）、John Mellby（Raytheon）、Jeanette Milos（DARPA）、Stan Moyer（Telcordia）、Russ Noseworthy（Object Sciences）、Dieter Quehl（Siemens）、Vijay Raghavan（Vanderbilt U.）、Lucie Robillard（U.S. Air Force）、Craig Rodrigues（BBN）、Rick Schantz（BBN）、Steve Shaffer（Kodak）、Tom Shields（Raytheon）、Dave Sharp（Boeing）、Naval Sodha（Ericsson）、Paul Stephenson（Ericsson）、Tatsuya Suda（UCI）、Umar Syyid（Hughes）、Janos Sztipanovits（Vanderbilt U.）、Gautam Thaker（Lockheed Martin）、Lothar Werzinger（Krones）、Don Winter（Boeing）。

特别感谢我们的文字编辑 Susan Cooper，她对我们的文字作了润色。此外，我们要感谢编辑 Debbie Lafferty、制作协调人 Elizabeth、丛书编辑和 C++的发明者 Bjarne Stroustrup，以及每一位为本书出版作出贡献的其他 Addison-Wesley 的工作人员，感谢他们的鼓励和耐心。

最后，我们还要表达对已故的“网络编程作品之父”W. Richard Stevens 的谢意和敬仰。他的著作对网络编程艺术和科学作了清晰透彻的诠释，使之达到了前所未有的高度。我们将站在 Richard 无形的肩膀上，努力将他著作中的理念在“面向对象”设计和 C++编程领域中发扬光大。

Steve 的致谢

我要感谢上帝，他给了我在计算机和网络世界中遨游的快乐，我也希望他快乐。感谢 Jane，陪伴我 20 年的妻子，谢谢你每一天爱我、鼓励我，没有你的支持，我无法完成这部作品——你是上帝给我的恩赐。感谢已故的 David N. Drummond，他给了没学位的孩子一个机会。感谢你，Doug Schmidt，一位学者、绅士，你的见识、热情和创造力每天感染并激励着我。

Doug 的致谢

写作本书花了 10 年。所以，看到它最终得以出版，我感到无比激动（和解脱）。为此，我要感谢 Steve Huston、Debbie Lafferty 和 Bjarne Stroustrup，他们见证了本书从开始策划直至最终结出硕果的历程，感谢他们在此期间对我的极大帮助和耐心。我还要感谢我的妻子 Sonja，谢谢她在我写作本书的过程中给我的爱和支持——既然现在书已经写完了，我们将有更多的时间去舞厅跳舞了！最后，感谢我的许多朋友和同事，他们来自 College of William and Mary、Washington University, St. Louis、University of California, Irvine、DARPA、Siemens，还有数以千计遍布于全球的 ACE 及 TAO 的开发者和使用者。在过去 20 年里，有了他们，我的学术研究和社会生活才变得如此丰富多彩。

C++ Network Programming Vol.1

目　录

图表一览 ix

序 xi

译序 xv

关于本书 xvii

第 0 章　设计课题、中间件方案、ACE 1

0.1　网络应用面临的挑战 1

0.2　网络应用的设计空间 5

0.3　面向对象中间件方案 7

0.4　ACE 工具包概览 12

0.5　示例：一个网络日志服务程序 17

0.6　小结 19

第 1 篇　面向对象网络编程 21

第 1 章　通信设计空间 23

1.1　无连接协议与面向连接协议 23

1.2　同步及异步消息交换 26

1.3　消息传递与共享内存 28

1.4　小结 31

第 2 章　Socket API 概述 33

2.1　操作系统 IPC 机制概述 33

2.2 Socket API 34
2.3 Socket API 的局限性 37
2.4 小结 43

第 3 章 ACE Socket Wrapper Facade 45
3.1 概述 45
3.2 ACE_Addr 类和 ACE_INET_Addr 类 49
3.3 ACE_IPC_SAP 类 52
3.4 ACE_SOCK 类 54
3.5 ACE_SOCK_Connector 类 56
3.6 ACE_SOCK_IO 类和 ACE_SOCK_Stream 类 60
3.7 ACE_SOCK_Acceptor 类 64
3.8 小结 68

第 4 章 网络日志服务程序的实现 71
4.1 概述 71
4.2 ACE_Message_Block 类 72
4.3 ACE_InputCDR 类和 ACE_OutputCDR 类 76
4.4 日志服务器的初始版本 80
4.5 客户程序 95
4.6 小结 99

第 2 篇 并发式面向对象网络编程 101

第 5 章 并发设计空间 103
5.1 循环、并发及反应式服务器 103
5.2 进程与线程 109
5.3 进程/线程创建策略 112
5.4 用户、核心及混合线程模型 114
5.5 分时及实时调度级别 119

5.6	“基于任务”与“基于消息”的体系	121
5.7	小结	123
第 6 章	**操作系统并发机制概述**	**125**
6.1	同步事件多路分离	125
6.2	多进程机制	127
6.3	多线程机制	129
6.4	同步机制	130
6.5	OS 并发机制的局限性	135
6.6	小结	137
第 7 章	**ACE 同步事件多路分离 Wrapper Facade**	**139**
7.1	概述	139
7.2	ACE_Handle_Set 类	141
7.3	ACE_Handle_Set_Iterator 类	147
7.4	ACE::select()方法	151
7.5	小结	157
第 8 章	**ACE 进程 Wrapper Facade**	**159**
8.1	概述	159
8.2	ACE_Process 类	161
8.3	ACE_Process_Options 类	165
8.4	ACE_Process_Manager 类	169
8.5	小结	182
第 9 章	**ACE 线程 Wrapper Facade**	**185**
9.1	概述	185
9.2	ACE_Thread_Manager 类	187
9.3	ACE_Sched_Params 类	198
9.4	ACE_TSS 类	201

9.5 小结 205

第 10 章 ACE 同步 Wrapper Facade 207
10.1 概述 207
10.2 ACE_Guard 类 209
10.3 ACE 互斥体类 212
10.4 ACE Readers/Writer Lock 类 218
10.5 ACE 信号量类 222
10.6 ACE 条件变量类 229
10.7 小结 233

附录 A ACE C++ Wrapper Facade 的设计原则 235
A.1 概述 235
A.2 通过 Wrapper Facade 增强类型安全性 236
A.3 简化常见情形 238
A.4 通过继承结构提高设计的清晰性和可扩充性 246
A.5 尽可能地隐藏平台间的差异 248
A.6 针对效率实施优化 255
A.7 小结 257

附录 B ACE 的过去、现在和将来 259
B.1 ACE 的演变 259
B.2 未来之路 267
B.3 结束语 268

术语表 269

参考文献 285

索引 293

图表一览

0.1 独立应用体系 2
0.2 一个普通的网络应用环境 3
0.3 网络应用的设计空间 6
0.4 应用环境下的面向对象中间件层次 8
0.5 ACE 的层次化体系结构 13
0.6 网络日志服务的参与者 18

1.1 可选的连接多路复用策略 25
1.2 同步及异步消息交换策略 26
1.3 消息传递与共享内存 29

2.1 Socket 空间分类图 42

3.1 ACE 面向连接的 Socket 类关系图 47
3.2 ACE Socket Wrapper Facade 分类图 48
3.3 ACE Socket Wrapper Facade 中的角色 49
3.4 ACE_Addr 和 ACE_INET_Addr 类图 51
3.5 ACE_IPC_SAP 和 ACE_SOCK 类图 54
3.6 ACE_SOCK_Connector 类图 57
3.7 ACE_SOCK_Stream 和 ACE_SOCK_IO 类图 62
3.8 ACE_SOCK_Acceptor 类图 65

4.1 ACE_Message_Block 类图 73
4.2 两种 ACE_Message_Block 75
4.3 ACE_InputCDR 和 ACE_OutputCDR 类图 79
4.4 日志服务器示例类 81
4.5 日志记录信息的消息块链 90

TP393.42/01

第0章

设计课题、中间件方案、ACE

本章提要

本章讲述“设计范式（paradigm）的转移（shift）”，即从*独立应用体系（stand-alone application architecture）*向*网络应用体系（networked application architecture）*的转移。这种转移会引发两类新的挑战：一类归属于问题空间（problem space），面对的是“软件体系及设计”方面的问题；一类归属于方案空间（solution space），涉及的是“实现网络应用程序的软件工具和技术”。本章首先就影响前一类问题的设计空间（design dimension）进行领域分析（domain analysis），并介绍了针对后一类问题且适用于这些问题的中间件（middleware）；然后介绍了 ACE 工具包和一个网络应用程序示例，这个示例贯穿本书，用来演示各种解决方案。

0.1 网络应用面临的挑战

大多数软件开发者都熟悉独立应用体系（stand-alone application architecture）。在这种体系中，一台独立的计算机包含软件的所有组成部分，包括图形用户接口（GUI）、应用服务处理和持久数据资源等等。例如，在图 0.1 所示的独立应用体系中，一台计算机融合了 GUI、服务处理和持久数据资源，所有外围设备都和计算机直接相连。在独立应用体系中，程序的执行始于哪一台计算机，控制流程就只位于哪一台计算机上。

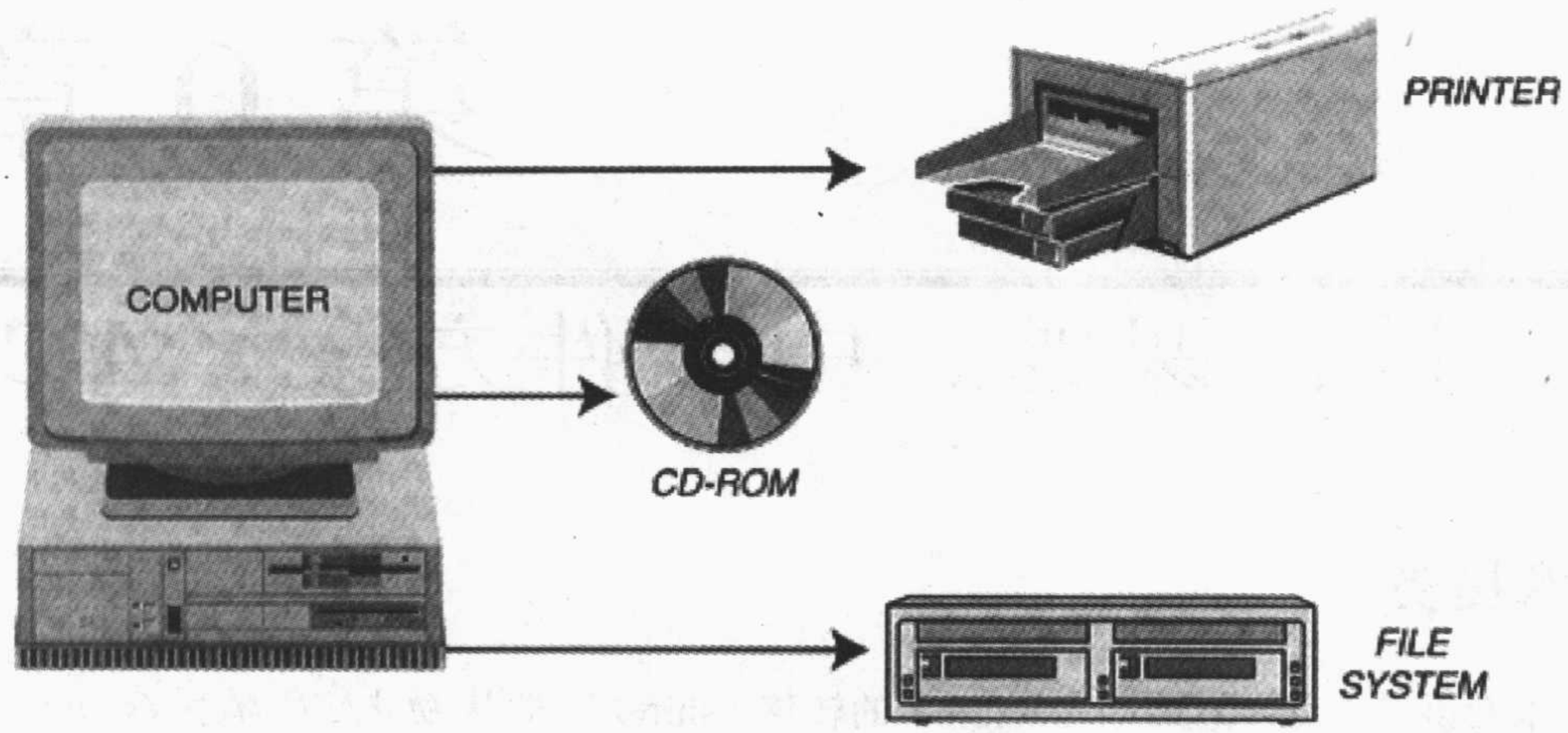


图 0.1：独立应用体系

相反，在网络应用体系（networked application architecture）中，应用系统被划分为所谓的“服务（service）”，供多个应用程序共享（shared）和复用（reused）。为了发挥最大的功效和作用，服务分布在通过网络相连的多个计算设备（computing device）中，如图 0.2 所示。这种环境下提供给客户（clients）的网络服务常有：分布式命名、网络文件系统、路由表管理、日志、打印、电子邮件、远程登录、文件传输、基于 Web 的电子商务服务、付款处理、顾客信息管理、服务台系统、MP3 交换、流式媒体、即时信息处理和社区聊天室等。

在图 0.2 所示的网络应用体系中，交互式 GUI、指令处理和持久数据资源被分割开来，分布在网络中的多个独立*主机（host）*中。在运行期间，一个网络应用程序的控制流程驻留在一个或多个主机中。系统中的所有组成部分协同通信，根据需要相互之间传输数据和控制命令。只要使用兼容的通信协议，即使底层的网络、操作系统、硬件和编程语言种类不同[HV99]，分离的组件之间也可以达成互操作性（interoperability）。这种跨越多个主机、将网络应用服务的职责进行委托的做法，可以带来以下好处：

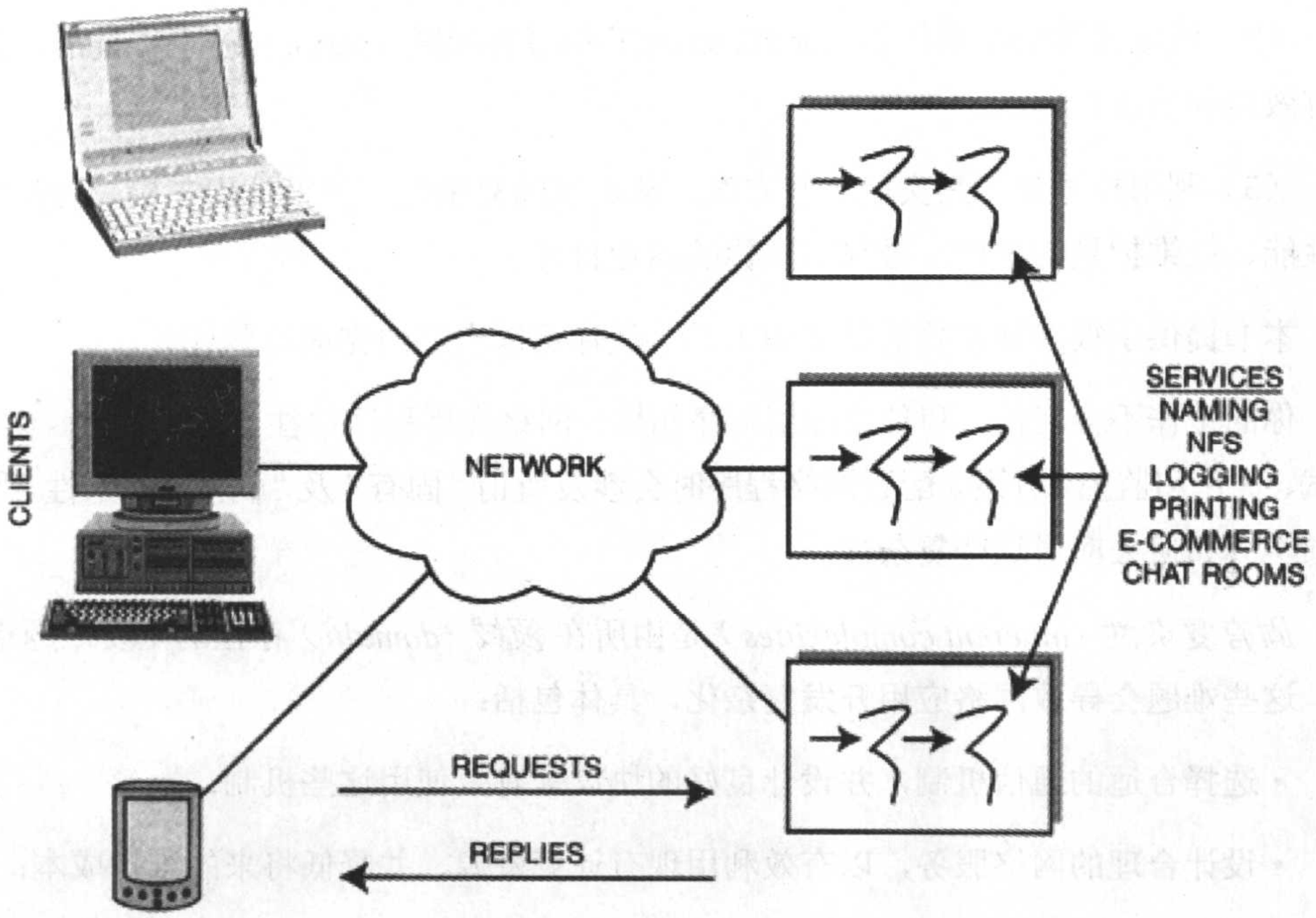


图 0.2：一个普通的网络应用环境

1．**增强可连接性和协作性**。信息可以迅速传播给更多的潜在用户。

2．**提高性能和可伸缩性**。根据目前和预计的系统需求，可以快速、安全地改变系统配置，以调整计算资源。

3．**降低成本**。用户和应用程序可以共享昂贵的外围设备和软件，如复杂的数据库管理系统。

作为一名网络应用程序开发者，你的任务是：首先了解你的应用程序将要提供哪些服务，有哪个（哪些）现有环境可以提供这些服务；然后

（1）针对服务（service）设计各种机制，供“服务和服务”之间、“服务和客户”之间通信时使用。

（2）确定有哪些体系结构（architecture）和服务配置（service arrangement）能够最有效地利用现有环境。

（3）利用技术和工具实现这些方案，从而消除复杂性，并设计出正确、可扩充、高性能、低维护量的软件，最终实现你的商业目标。

本书提供了很多你需要的信息和工具，使你可以出色地完成这些任务。

你的工作不会轻松。和独立应用程序相比，网络应用程序往往更难于设计、实现、调试、优化和监控。开发、配置网络程序时会涉及所谓“固有”及“偶发”复杂性[Bro97]，必须学会如何去应对这些复杂性。

*固有复杂性（inherent complexities）*是由所在*领域（domain）*存在的主要难题造成的，这些难题会导致网络应用开发复杂化，具体包括：

- 选择合适的通信机制，并设计良好的协议来有效使用这些机制；
- 设计合理的网络服务，以有效利用现有计算资源、并降低将来的维护成本；
- 有效使用*并发（concurrency）*机制，使系统获得可预计的、可靠的高性能；
- 管理和配置服务（service），以获得最大程度的系统可用性和灵活性。

应付固有复杂性需要经验，并需要对所在领域有深刻了解。固有复杂性涉及很多设计上的利弊权衡，我们将在第 1 章和第 5 章进行研究。

*偶发复杂性（accidental complexities）*源自开发网络应用软件时使用的工具和技术存在的局限性，具体包括：

- 缺乏类型安全、可移植、可扩充的原始（native）OS API；
- 算法分解（algorithm decomposition）的广泛使用，无谓地造成网络应用程序在维护和扩充上的困难；
- 网络应用中，核心（core）概念和功能的不断发现和创造，造成软件生命周期的成本无谓地居高不下。

网络应用开发者必须了解这些设计挑战，并运用相关技术有效地解决它们。在本书中，我们通过示例，演示了 ACE 如何运用“面向对象”技术和 C++语言特性来解决以上列出的偶发复杂性。

0.2 网络应用的设计空间

在某一领域（domain）中，没有评估其主要设计空间（design dimension）就去学习 API 和接口编程，这种情况是可能的。但从我们的经验来看，如果开发者对网络应用领域的基础知识有更深的了解，就能更有效地解决设计、实现和性能方面的重大难题。所以，首先我们来探讨一下网络应用开发中核心的结构化设计空间（core architectural design dimension）。我们将集中讨论支持“多服务”或“一个服务多个实例”的服务器，这些服务器同时和多个客户协同工作，情况类似于图 0.2 所示的网络应用环境。

本书讨论的设计空间是经过详尽的*领域分析（domain analysis）*后确定的，它们基于过去 10 多年亲手开发数百个网络应用产品和系统时积累的设计和实现经验。领域分析是一个归纳（inductive）和反馈驱动（feedback-driven）的过程，它对一个应用领域进行系统的分析，识别出其核心课题和设计空间，从而为它们找到相应的解决之道。这个过程带来以下好处。

- **为领域中的抽象概念定义了一个常用词汇表。**这一方面使得开发者能够更有效地交流[Fow97]；另一方面，一旦明确了问题空间（problem space）中的词汇，在方案空间（solution space）中寻找对应的模式和软件抽象体的工作就会得以简化。例如，对网络协议、*事件（event）*多路分离策略、并发体系有了共识，就可以将这些概念运用到关于 wrapper facade 的讨论中，也可以运用到[SH]中关于 ACE 框架的讨论中。

- **提高了可复用性。**它将设计问题分为两类：

（1）和“应用的类型（type）”相关的问题；

（2）这个领域中，所有应用共有的问题。

在一个领域中，如果将注意力集中在那些共有的设计问题上，应用程序和*中间件（middleware）*开发者就能够抓住时机，改造或构造出可复用的软件类库（software class library）。在这些类库中，如果规范的控制流程被提取出来，并经过重新整合，就可以构成中间件框架（middleware framework）（如 ACE 中的中间件框架），从而显著降低后续的应用程序开发工作。在一个成熟的领域中，借助面向对象语言特性，如继承、

动态绑定、参数化类型和异常，可以扩充、定制现有的中间件框架，从而系统地解决“和应用相关（application-specific）”的设计问题。

在网络应用领域中，对于图 0.3 所示的 4 个设计空间中的每一个，开发者都面临设计上的抉择。这些设计空间涉及的主要是固有复杂性的管理。所以，在很大程度上，它们独立于特定的生命周期过程、设计方法和表示法、编程语言、操作系统平台和网络硬件。每一个设计空间都由一组相对独立的选择（alternatives）构成。尽管它们之间大都是正交的，但在你的网络应用程序中，如果你改变了其中一个或多个空间，网络应用的“形状”就会相应发生变化。所以，设计上的变化不是孤立地发生的。在涉及以下设计空间时，一定要将这一点牢记在心：

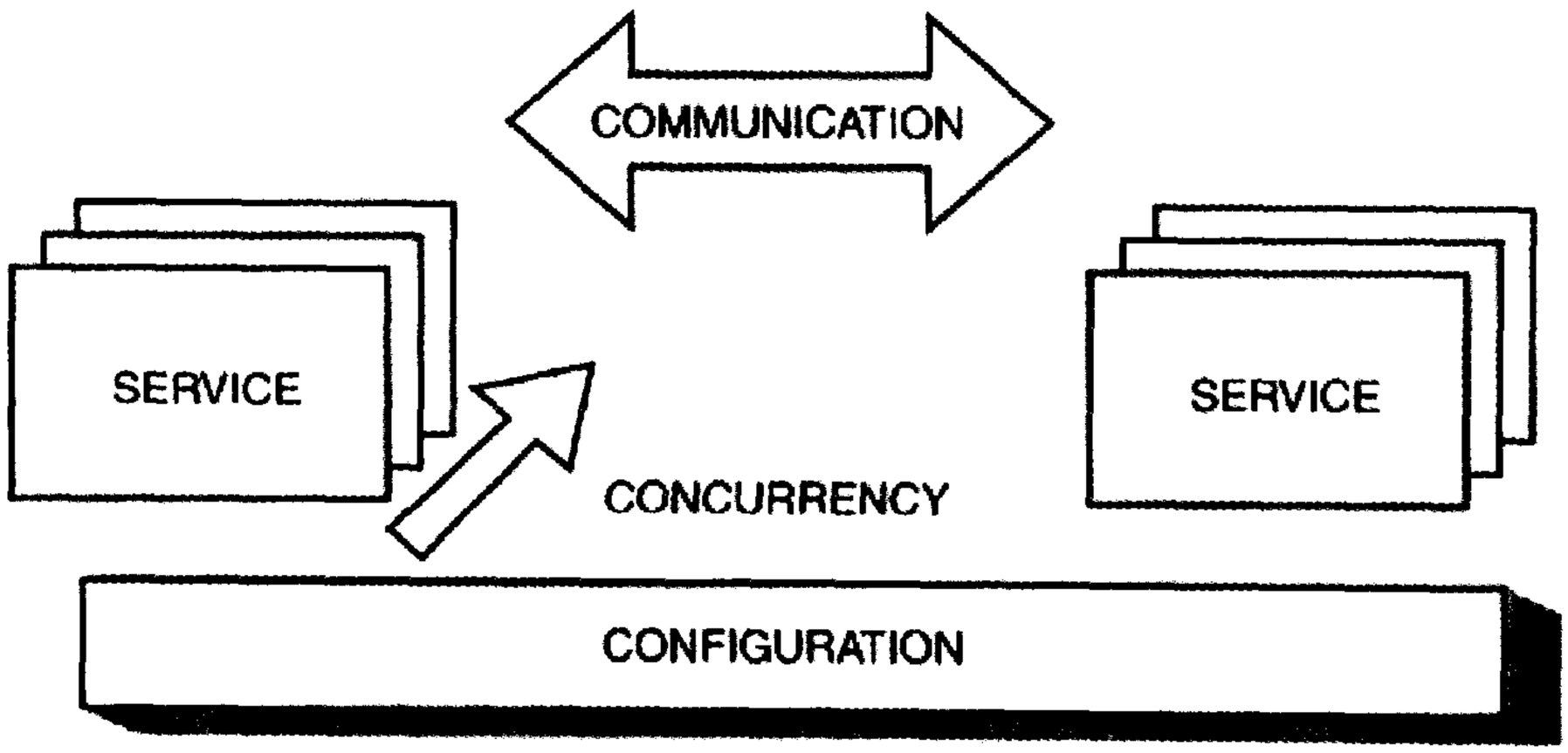


图 0.3：网络应用的设计空间

1．**通信（Communication）空间。**它指的是网络应用进行交互（interact）的规则、形式和抽象层次。

2．**并发（Concurrency）空间。**它指的是一些策略和机制，用来控制如何正确使用进程和线程，以表达多个服务实例，以及每个服务实例如何在内部使用多线程。

3．**服务（Service）空间。**它反映网络应用服务的主要性质，如每个服务实例的持续时间和结构。

4. **配置（Configuration）空间**。它反映的是如何识别网络服务，以及在何时将网络服务绑定在一起，以构成完整的应用。“配置”往往影响到多个服务，以及服务之间的关系。

在第 1 章和第 5 章，将更深入地分析头两个设计空间，第 3 和第 4 个设计空间则在参考文献[SH]中讨论。首先，解释重要的词汇、设计上的折中、方案的提炼；然后，介绍“和每一个设计空间有关”的平台能力、其相关的偶发复杂性，以及过去 10 多年来，在应对这些设计因素时，ACE 所提供的方案。你将看到，ACE 工具包使用了久经考验的“面向对象”分割（partitioning）技术、*接口（interface）*设计、数据封装模式和 C++特性，这样，在你的网络应用程序中，设计空间的变化就可以做到最大可能的独立性和可移植性。

0.3 面向对象中间件方案

在处理网络应用的偶发复杂性和固有复杂性时，一些最成功的技术和工具主要集中在“面向对象”中间件中，它有助于管理网络应用中的复杂性（complexity）和异种性（heterogeneity）。“面向对象”中间件提供了可复用的服务/协议组件和框架（framework）软件，在功能上为以下二者之间架设了桥梁：

（1）端对端（end-to-end）应用的功能需求；

（2）底层的操作系统、网络*协议栈（protocol stack）*和硬件设备。

面向对象中间件提供的能力十分重要，它可以对网络应用的连接和协作起到简化和协调的作用。

0.3.1 面向对象中间件层

网络协议栈（如 TCP/IP[Ste93]）可以分为多个层次（譬如 OSI 参考模型[Bla91]中定义的物理层、数据链路层、网络层、传输层、会话层、表示层和应用层）。同样，“面向对象”中间件也可以分为多个层次[SS01]，如图 0.4 所示。

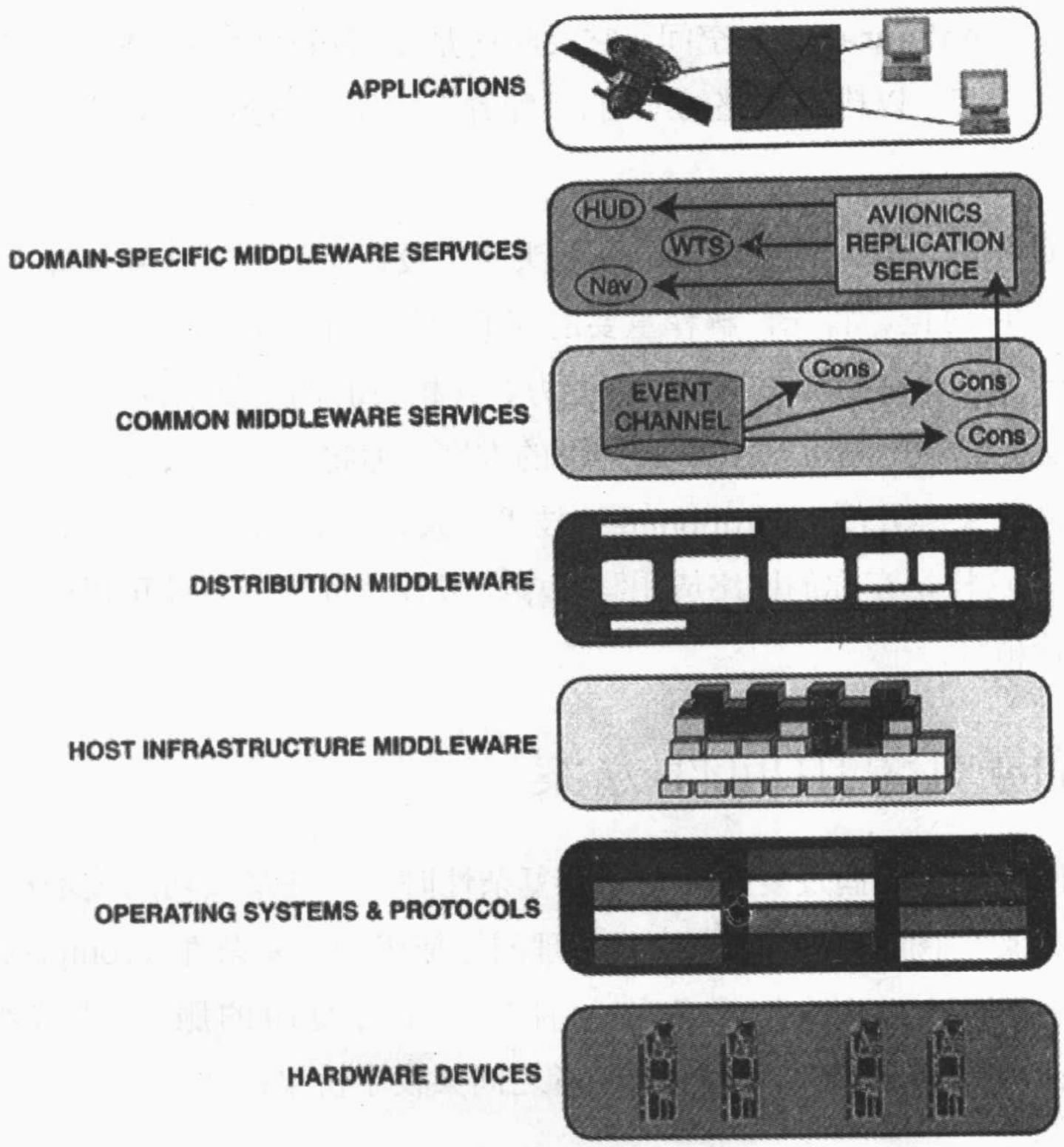


图 0.4：应用环境下的面向对象中间件层次

一个常见的面向对象中间件体系结构，一般包括如下所述的几个层次：

*主机基础设施中间件（Host infrastructure middleware）*对 OS“并发（concurrency）”机制和“进程间通信（IPC）”机制进行封装，以获得“面向对象”网络编程能力。有了这些能力，通过原始 OS API——例如，Socket、POSIX 线程（Pthreads）——开发网络应用程序时涉及的很多繁琐、容易出错、不可移植的情况就能得以避免。广泛应用的主机基础设施中间件的例子包括 Java Package[AGH00]和 ACE。

*分布式中间件（Distribution middleware）*使用并扩充了主机基础设施中间件，使得一些网络编程任务自动化（automate）。这些编程任务包括：连接管理和内存管理、*整编（marshaling）*、*解编（demarshaling）*、端点和请求的*多路分离（demultiplexing）*、

同步、多线程。使用分布式中间件，程序员就可以像开发独立应用程序那样开发分布式应用程序；也就是说，只需向目标对象请求操作，不必关心目标对象的位置、语言、OS 或硬件[HV99]。分布式中间件的核心是 *Object Request Broker*（ORB），例如，COM+[Box97]、Java RMI[Sun98]、CORBA[Obj01]。

*公共中间件服务（Common middleware service）*扩展了分布式中间件，定义了更高层次、独立于特定领域的服务，例如，事件通知、日志记录、持久性（persistence）、安全且可恢复的事务（recoverable transaction）。但是，“分布式中间件”主要是管理支持“面向对象”分布式编程模型的终端系统（end-system）资源，而“公共中间件服务”则是对整个分布式系统中的各种资源进行分配、调度和协调。没有“公共中间件服务”，这些“端对端”功能就得由每一个网络应用程序*专门（ad hoc）*实现。

*特定领域中间件服务（Domain-specific middleware service）*可以满足特定领域（例如，电信、电子商务、医疗保健、过程自动化、航空电子）的特定需求。尽管其他“面向对象”中间件层提供的是可广泛复用（reusable）的“横向（horizontal）”机制和服务，但“特定领域”中间件服务面向的是“纵向市场（vertical market）”。从“商业现货”（COTS，commercial off-the-shelf）的角度来看，“特定领域”服务是如今最不成熟的中间件层。其部分原因是，一直以来都没有中间件标准来提供一个稳定的基础，用以创建“特定领域”服务。

“面向对象”中间件是开发网络应用程序的重要工具。为开发和改进网络应用，它对以下 3 个方面进行了改进：

1. **以战略为中心（strategic focus）。**开阔了应用程序开发者的视野，使之不拘泥于低级的 OS 并发及网络 API。牢固掌握这些 API 的概念和功能，对所有的网络应用开发者来说都是基础。但是，中间件有助于将这些细节抽象为更高级、更易于使用的技术。一旦无需过多地关心底层的细节，开发者就能将注意力集中在更具战略性、以应用为中心的问题上。

2. **有效的可复用性（effective reuse）。**通过利用已有的开发技能，并通过将重要模式[SSRB00,GHJV95]具体实现为可复用的中间件框架，来分期承担（amortize）软件生命周期中的开发工作。将来，大多数网络应用程序都可以组装而成，而且是通过整合、编写（script）“特定领域”和公共“可插入（pluggable）”中间件服务组件来进行组装，而不是完全白手起家重新编写程序[Joh97]。

3. **开放的标准（open standard）。**提供了一组可移植、可互操作（interoperable）的软件制品（software artifact）。这些制品有助于将开发者的注意力引导至更高级的软件应用体系和设计问题上，例如，可互操作的安全性、层次化的分布式资源管理、容错服务等。开放且（或）标准的 COTS“面向对象”中间件——例如，CORBA、Java 虚拟机、ACE——正扮演着日益重要的角色，这些中间件还能以“开放源码”的途径购买或得到。对那些面临时间和市场压力、缺乏软件开发资源的组织来说，COTS 中间件尤为重要。

虽然分布式中间件、公共中间件服务和“特定领域”中间件服务是重要的主题，但出于下一节所说的原因，本书将不对它们作进一步讨论。有关这些主题的进一步论述，请登录 http://ace.ece.uci.edu/middleware.html 网站，或参考《*Advanced CORBA Programming with C++*》[HV99]。

0.3.2 主机基础设施中间件的优势

如果程序员必须考虑服务质量（QoS, quality of service）上的要求和（或）成本上的限制，那么，和较高层（higher level）的中间件相比，主机基础设施中间件将更显优势。况且，要发展高级中间件，主机基础设施中间件可谓是一个基础领域。这些领域及其基本原理将在下文中进行论述。

满足严格的 QoS 需求 为满足“效率（efficiency）”和“可预见性（predictability）”方面严格的 QoS 需求，某些类型的应用程序需要访问原始的 OS IPC 机制和协议。例如，多媒体应用程序需要持续时间长、双向字节流的通信服务，因而不适合使用某些“分布式中间件”提供的“同步请求/应答”模式[NGSY00]。尽管优化技术已经有了巨大

进步[GS99, POS+00]，但在很多传统的“分布式中间件”实现中，吞吐量和*延迟（latency）*上的开销还是十分显著，而且缺乏足够的手段（hook）来处理其他“和 QoS 相关”的性质，譬如*抖动（jitter）*和可靠性（dependability）。

相反，主机基础设施中间件往往在保证“端对端”QoS 需求上更适合，因为它允许应用程序：

- 忽略不必要的功能，例如，在同种（homogeneous）环境中忽略整编（marshaling）和解编（demarshaling）；
- 对通信行为施加精细的控制，例如，支持 IP 多路传输和异步 I/O；
- 定制（customize）网络协议，以对网络*带宽（bandwidth）*的使用实施优化，或将“共享内存（shared memeory）通信”替换为“回送（loopback）网络通信”。

可以预计，10 年之后，分布式中间件和公共服务上的研究和开发（R&D）将会达到一个新的高度，它的 QoS 水平将能够和手工编写的主机基础设施中间件和网络应用程序相匹敌，甚至胜出。但此时此刻，我们还是得编写、使用大量的产品软件。正是在这种环境下，主机基础设施中间件扮演着重要的角色，提升了抽象的层次：在这一层次上，网络应用程序的开发不会过度影响它们的 QoS。

控制成本 为了在全球竞争环境中生存，很多组织正在向“面向对象”开发过程（process）和方法（method）过渡。基于这一背景，主机基础设施中间件提供了功能强大、久经考验的解决方案。这些方案在消除本书第 0.1 节讨论的固有和偶发复杂性时，有助于控制成本。

例如，使用新的编译器、开发环境、调试器和工具包的开销很大，而培训软件工程师的开销则更大。因为熟练掌握新技术的学习曲线十分陡峭。当着手一个软件项目，并在项目中评估中准备采用新技术时，控制以上成本十分重要。主机基础设施中间件

是一种有效的工具，它可以利用现有的 OS 和网络经验、知识及技能；还能够将开发经验扩展到新的平台上，并能攀越学习曲线，转移到至高级、更节约成本的软件技术上。

改进核心技术，提高现有水平 对主机基础设施中间件有了透彻的理解，将有助于开发者识别高级模式（pattern）和服务（service），从而在各自应用领域获得更大的产出。在今天的“面向方法”和“面向消息”中间件技术中，尚有许多新的技术难题需要攻克。基础设施中间件为将来的研究和开发提供了重要的构件。因为

- 开发者若牢固掌握了“主机基础设施中间件”涉及的设计方法和模式，就能更快地跟上软件技术的进步，从而，就能促进开发小组或团队使用更高级的中间件功能。
- 开发者若对中间件底层“发生了什么”有了透彻的理解，就更能发现新方法，进而改进网络应用。

0.4 ACE 工具包概览

*ADAPTIVE 通信环境（ACE, ADAPTIVE Communication Environment）*是一种广泛使用的主机基础设施中间件。ACE 库包含约 240,000 行 C++代码、约 500 个类。ACE 软件发布包还包括数百个自动化的回归测试（regression test）和示例程序。作为开放源码（open-source）软件，ACE 可以免费获得；可以在 http://ace.ece.uci.edu/ 或 http://www.riverace.com 下载到它。

为分离关系、降低复杂度、允许功能模块的划分，ACE 被设计为层次化体系结构[BMR+96]，如图 0.5 所示。ACE 软件包的基础是 ACE OS Adaptation *层（layer）*和 C++ wrapper facades[SSRB00]的结合，它封装了核心 OS 并发网络编程机制。ACE 的较高层建立在这一基础之上，提供了可复用的*框架（framework）*、网络服务组件、基于标准的中间件等。这些中间件层联合起来，简化了网络应用程序的创建、组成、配置和移植，但不会导致显著的性能开销。

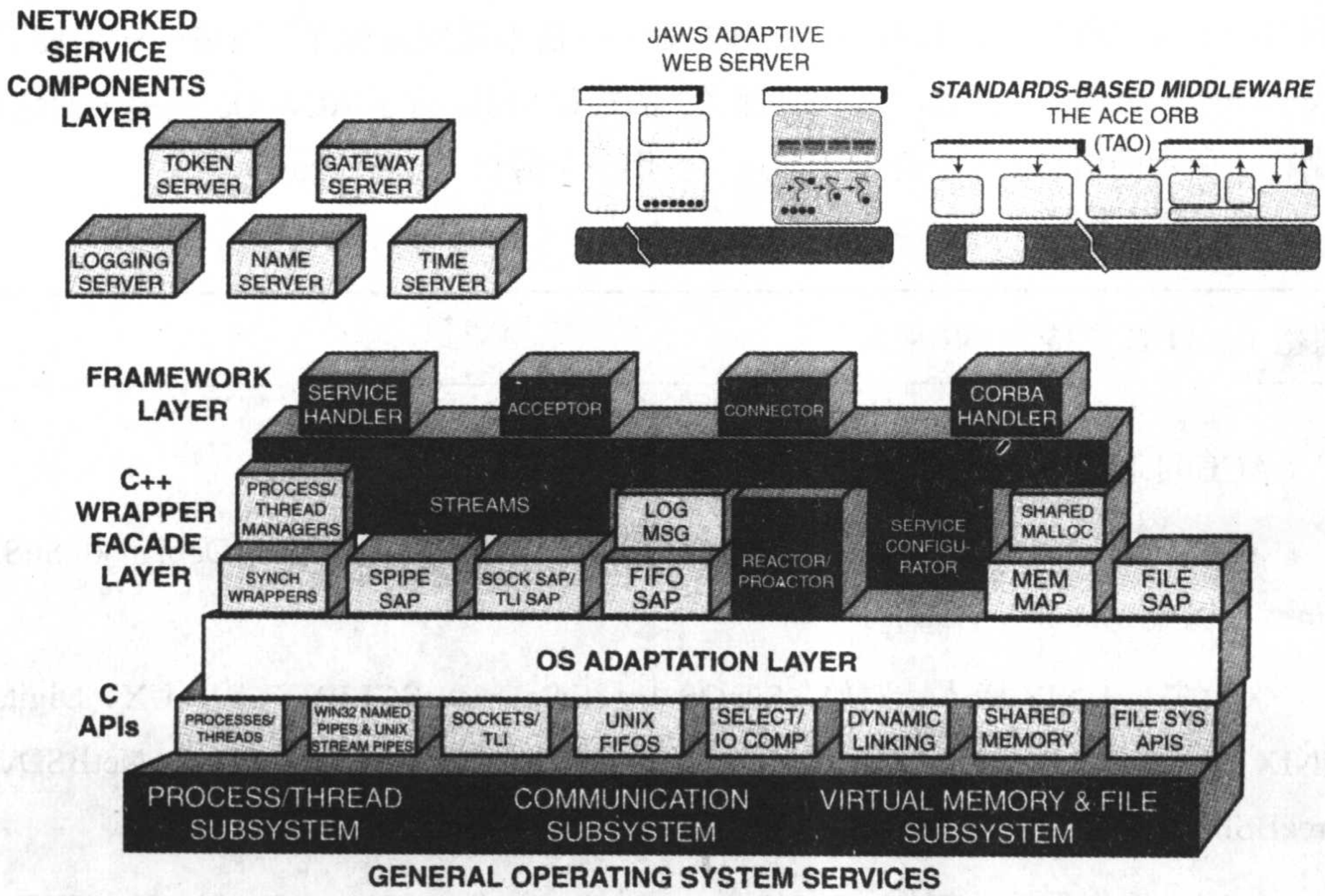


图 0.5：ACE 的层次化体系结构

本书集中讨论“针对原始 OS IPC 和并发机制”的 ACE wrapper facade。关于框架（framework）的优点，以及 ACE 框架的详尽介绍，请阅读《*C++网络编程*》[SH]第二卷。本章余下部分概要介绍 ACE 中各个层次的结构和功能。第 B.1.4 节介绍了基于标准的中间件（TAO[SLM98]和 JAWS[HS99]），它们基于 ACE 并和 ACE 捆绑在一起。

0.4.1 ACE OS Adaptation 层

ACE OS Adaptation（适应）层大约占 ACE 的 10%（约 27,000 行代码）。它由一个名为 ACE_OS 的类构成，此类包含 500 多个 C++静态方法（static method）。这些方法封装了原始的、基于 C 的 OS API，隐藏了“和平台相关”的细节，展示了统一的 OS 机制接口，供更高级的 ACE 层使用。ACE_OS adaptation 层使得 ACE 的可移植

性和可维护性变得单纯；只有 ACE 开发者——而不是应用程序开发者——需要了解 ACE wrapper facade 底层的、“和平台相关”的神秘知识。有了 ACE_OS 类提供的抽象，在副栏 1 所示的所有 OS 平台上，都只用使用一个源树（source tree）。

副栏 1：ACE 支持的 OS 平台

ACE 可运行于广泛的操作系统之上，包括

- PC：例如，Windows（所有 32/64 位版本）、WinCE；RedHat、Debian 和 SuSE Linux；以及 Macintosh OS X。
- 大多数 UNIX 版本：例如，SunOS 4.x 和 Solaris、SGI IRIX、HP-UX、Digital UNIX（Compaq Tru64）、AIX、DG/UX、SCO OpenServer、UnixWare、NetBSD、FreeBSD。
- 实时操作系统：例如，VxWorks、OS/9、Chorus、LynxOS、Pharlap TNT、QNX Neutrino 和 RTP RTEMS、pSoS。
- 大型企业系统：例如，OpenVMS、MVS OpenEdition、Tandem NonStop-UX、Cray UNICOS。

在这些平台上，ACE 可用于几乎所有主流 C++编译器。ACE 网站 http://ace.ece.uci.edu 上提供了一份完整的同时也是最新的 OS 平台列表，以及下载和编译 ACE 的说明。

0.4.2 ACE C++ Wrapper Facade 层

wrapper facade 由一个或多个类组成，这些类将函数和数据封装在类型安全（type-safe）的“面向对象”接口中[SSRB00]。ACE C++ wrapper facade 层位于 OS adaptation 层之上，并提供了大致相同的功能，如图 0.5 所示。由于这些功能被包装成 C++类，而不是独立的 C 函数，在学习和正确使用 ACE 时，需要的精力投入将大为减少。为降低或消除由于“可用性”和“安全性”的提高所导致的性能开销，ACE wrapper facade

的设计必须十分谨慎。附录 A 就 ACE 设计的基本原则作了论述。

ACE 提供了一套庞大的 wrapper facade，几乎占整个代码库的 50%。通过有选择地继承、聚合和（或）实例化这些 wrapper facade，应用程序可以结合、提炼这些 wrapper facade。本书将展示如何使用 socket、文件、并发及同步 wrapper facade 来开发高效、可移植的网络应用程序。

0.4.3 ACE framework 层

ACE 中剩下的 40%是*面向对象框架（object-oriented framework）*，它们是多组集成在一起、相互合作的类，用来为一组相关应用提供可复用的软件架构[FS97]。为获得成功的、系统化（systematic）的可复用性，面向对象框架很关键；因为它们补充、加强了其他复用技术，如类库、组件、模式[Joh97]。例如，较之复用（reuse）“单独的类”或“独立的函数”，ACE 框架加强了“和应用相关的类”和“独立于应用的类”之间的集成与合作，可以获得更大范围的软件可复用性。ACE 框架集成、扩充了 C++ wrapper facade 类；它们运用高级的并发和网络编程模式[BMR+96, SSRB00]，*具体化（reify）*了 ACE 中多组（families of）相关类之间的标准控制流程和合作。

在开发、配置并发网络应用程序和服务时，以下 ACE 框架提供了高效、稳定、灵活的支持：

“事件多路分离和分发”框架 ACE Reactor 和 Proactor 框架分别实现了 Reactor 和 Proactor 模式[SSRB00]。Reactor 和 Proactor 框架自动处理“和应用相关”的处理程序（handler）的多路分离（demultiplexing）和分发（dispatching），以响应各种基于 I/O、基于计时器、基于信号和基于同步的事件。

“连接建立和服务初始化”框架 ACE Acceptor-Connector 框架实现了 *Acceptor-Connector 模式* [SSRB00]。这个框架将“主动和被动初始化角色”同“初始化结束之后，通信对等服务所执行的应用处理”分离开来。

“并发”框架 ACE提供了Task框架，用来实现一些主要的并发模式[SSRB00, Lea99]，如Active Object、Half-Sync/Half-Async。二者分别将方法的执行和方法的请求、同步处理和异步处理分离开来，从而简化了并发编程。

“服务配置器（Service configurator）”框架 这个框架实现了*Component Configurator（组件配置器）模式*[SSRB00]，以支持应用程序的配置。应用程序中的服务可以在开发周期的后期（例如安装期间）动态组装。在运行期间，它还支持应用程序中服务的动态重新配置（reconfiguration）。

“流（stream）”框架 这个框架实现了*Pipes and Filters模式*[BMR+96]。其中，每一个处理步骤都被封装在一个过滤模块（filtering module）中，这种模块可以访问、操纵“经过这些模块流（stream of modules）”的数据。对那些可以灵活地组合起来、从而创建某种网络应用（譬如用户级协议栈和网络管理代理[SS94]）并具有层次化结构的服务来说，ACE的“流”框架可以简化其开发。

关于ACE框架的目的、设计和使用，其深入讨论请参见《*C++ Network Programming: Systematic Reuse with ACE and Frameworks*》[SH]。关于ACE wrapper facade和framework（框架）的更多知识，还可参阅《*The ACE Programmer's Guide*》[HJS]。

0.4.4 ACE网络服务组件层

除了前面讲述的主机基础设施中间件wrapper facade和framework（框架）之外，ACE还提供了一个网络服务库，库中的服务都被包装成组件。所谓*组件（component）*，指的是软件系统中一个“被封装的部分”，用来实现特定的一个或一组服务[Szy98]。虽然ACE库本身并不包含这些组件，但它们被捆绑在ACE软件的发行包中，用来提供以下功能：

- **演示“ACE功能的常见使用方式”**——这些组件用来演示如何通过主要的ACE框架和类来开发灵活、高效、健壮的网络服务。

• **提取“可复用的网络应用构件”**——这些组件可以为常见的网络应用服务（如名称管理、事件路由处理[Sch00]、日志记录、时间同步[SSRB00]和网络锁定）提供可复用（reusable）的实现。

0.5 示例：一个网络日志服务程序

在整本书中，我们用一个网络日志服务程序作为例子，以演示设计要点和 ACE 的功能。在这个服务程序中，可以收集、记录一个或多个客户程序发出的诊断信息。它不同于 Windows NT/2000 中事件记录的普通方式（Windows 95/98 中没有这种日志记录）。但是，如果你是一位有经验的 UNIX 程序员，你会认为这是一种时间上的浪费，因为 SYSLOGD 已经提供了这类服务。尽管如此，这个日志服务程序体现了一个重要优点：可移植。因而，在 ACE 支持的所有平台上，应用程序都能够记录信息。

这个日志服务程序示例是 ACE 实际日志服务程序的缩影。通过 Component Configurator 模式[SSRB00]和 ACE Service Configurator（服务配置器）框架[SH]，可以动态配置 ACE 的日志服务程序。采用 Adapter 模式[GHJV95]，记录可以被重定向（redirected）至 UNIX 的 SYSLOGD，或 Windows NT/2000 的事件记录，或二者——即使启动应用程序（initiating application）在另一类 OS 平台上。在本书中，这个日志服务程序示例被有意简化了。这样，我们就可以将注意力放在消除复杂性上。在图 0.6 中，我们描述了这个网络日志服务程序中的应用程序进程和服务器。下面，简要介绍图 0.6 中的主要组成部分。

客户应用程序进程运行在客户机（client host）上，会产生小至调试信息、大至严重错误信息的各种日志记录。客户应用程序发出的日志信息反映以下内容：

（1）日志记录创建的时间；

（2）应用程序的进程标识符；

（3）日志记录的优先级；

（4）一个字符串。这个字符串包含的是这个日志信息的文本，文本长度小至 0，大至一个“可配置的最大长度”，譬如 4KB。

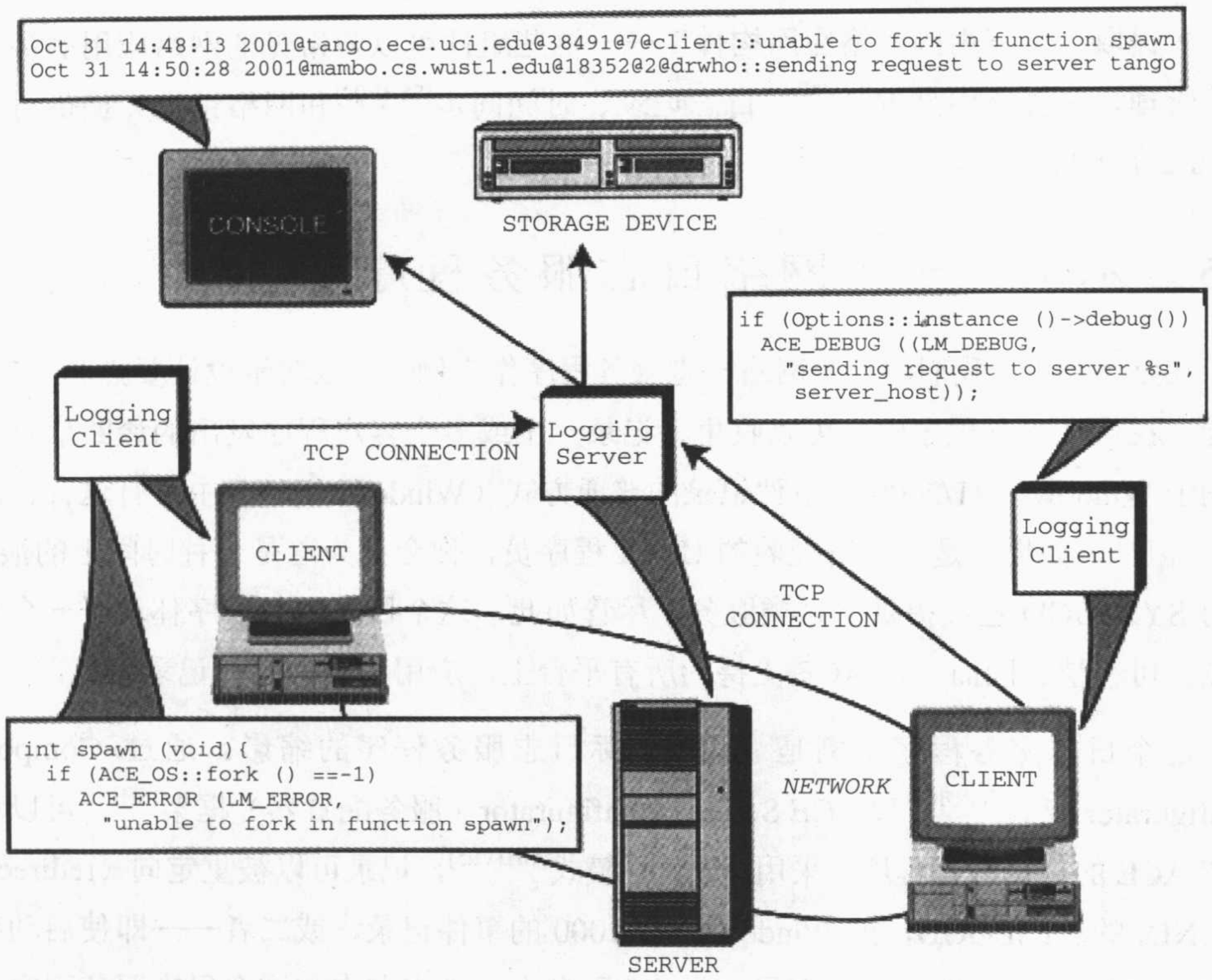


图 0.6：网络日志服务的参与者

日志服务器对来自客户应用程序的日志记录进行收集和输出。利用“从 Socket API 获得的地址信息”，日志服务器可以识别每一条消息来自哪一个客户机。一个系统一般配备一个日志服务器，但也可以重复配备，以增强容错性（fault tolerance）。

在整本书中，我们都借助这个网络日志服务程序，来讨论网络应用程序的领域分析空间（domain analysis dimensions），使讨论更为具体。我们的日志服务程序的结构（architecture）受这一领域分析左右。正如真正的产品在其生命周期中会有一定范围的变化一样，在本书以及参考文献[SH]中，随着我们不断前进，这个日志服务程序的设计、功能、可伸缩性、健壮性也会不断改进。我们将循序渐进地开发这个服务程序，演示常见设计挑战的解决之道。这之中运用了很多重要的模式，这些模式通过 ACE 中的类得以实现。副栏 2 描述如何生成（build）ACE 库。生成 ACE 库后，就可以实际操练本书提供的例子了。

副栏 2：如何生成 ACE 和“使用了 ACE”的程序

ACE 是开放源码（open-source）软件。所以，可以在 http://ace.ece.uci.edu 下载并且生成它。这里有一些要点，可以帮助你理解我们演示的源程序示例，以及学习如何生成 ACE、如何生成示例程序和你自己的应用程序：

• 将 ACE 安装到一个空目录中。发行包的顶级目录名为 ACE_wrappers。我们将这一顶级目录称为$ACE_ROOT。根据这个名称创建一个环境变量，此名称包含顶级 ACE 目录的完整路径。

• ACE 源代码和头文件位于$ACE_ROOT/ace。

• 本书的网络日志服务例程的源代码和头文件位于$ACE_ROOT/examples/C++NPv1。

• 当编译你的程序时，必须将$ACE_ROOT 目录添加到编译器的文件（包含路径）中，这通常是通过-I 或/I 编译器选项来指定的。

• $ACE_ROOT/ACE-INSTALL.html 文件完整地说明了如何生成、安装 ACE 和那些“使用了 ACE”的程序。

你还可以以微不足道的价格从 Riverace 购买到预先生成好（prebuilt）的 ACE 版本。Riverace 支持多种预先构建好（prebuilt）的编译器和 OS 平台，详细列表见 http://www.riverace.com。

0.6 小结

本章讲述的是，在开发“可以高效地运行在分布式计算环境中”的网络应用程序和中间件时，将面临哪些挑战。我们引入了“固有”和“偶发”复杂性的概念，在开发软件包括开发那些极为受限的实时和嵌入式系统[SLM98]，乃至近来不断发展的“中间件抽象”（middleware abstraction）[MSKS00]和“具有严格 QoS 要求的下一代网络应用程序”[SKKK00]时，我们会面临这些复杂性。我们对“中间件分层”提供了分类，强调了主机基础设施中间件的优点，而主机基础设施中间件正是本书讨论的核心。

本章还针对网络应用体系的主要设计空间，介绍了“领域分析”的结果。这些结果由四大类组成：

- 通信协议及机制；
- 并发体系；
- 服务体系；
- 服务配置策略。

过去 10 年里，在开发数百个网络应用程序和中间件组件的过程中，“领域分析”已经获得改进。这一分析还指导了 ACE 并发网络编程工具包的开发。将“网络应用程序中反复出现的结构和行为”*重构（refactoring）*[FBB+99]为“主机基础中间件”时使用的原则和得到的好处，在 ACE 中得到了验证。ACE 的“面向模式（pattern-oriented）”软件体系构成了具有工业强度的实例，证实了正确的“面向对象设计”和“C++的运用”如何显著加快开发进度，如何提高网络应用程序和中间件的质量、灵活性和性能。

最后，我们介绍了网络日志服务程序。这个程序的功能是存储“来自一个或多个客户程序”的诊断信息。在整本书中，我们通过这个例子来演示常见的设计问题，以及如何通过 ACE 提供有效的解决方案。本书后面两篇的内容组织如下：

- **第 1 篇**——第 1~4 章概要介绍通信设计的选择方案，讲述 ACE 中使用的“用于有效编写 OS IPC 机制”的“面向对象”技术。
- **第 2 篇**——第 5~10 章概要介绍并发设计的可选方案，讲述 ACE 中使用的“用于有效编写 OS 并发机制”的“面向对象”技术。

在这两篇中，我们就开发者设计网络应用程序时、直接使用原始 OS IPC 和并发 API 编写程序时经常出现的问题进行了说明；还展示了 ACE 如何运用面向对象设计技术、C++特性和模式来解决这些问题。

第 1 篇

面向对象网络编程

第 1 章

通信设计空间

本章提要

通信（communication）是网络应用程序设计的基础。本章就通信设计空间（design dimension）作了领域分析（domain analysis），介绍了网络应用程序之间相互作用的规则（rule）、形式（form）和抽象层次。在本章中，我们将论及以下设计空间：

- 无连接协议与面向连接协议
- 同步及异步*消息(message)*交换
- 消息传递与共享内存

1.1 无连接协议与面向连接协议

所谓协议（protocol），就是一组规则，用来指定“控制信息”和“数据信息”如何在通信实体（譬如，在网络计算环境内部相互作用的应用程序进程）之间交换。协议一般分为无连接（connectionless）和面向连接（connection-oriented）两种。在这一设计空间，需要权衡的地方主要涉及延迟（latency）、可伸缩性（scalability）和可靠性（reliability）。

无连接协议提供的是“面向消息的（message-oriented）”服务。其中，每一条消息都可以独立寻址和发送。无连接协议一般采用“尽力（best-effort）”发送语义。这些语义不保证某组消息会以特定次序到达目的地，甚至根本不会到达目的地！

广泛使用无连接协议的例子有：用户数据报协议（UDP，User Datagram Protocol）和网际协议（IP，Internet Protocol）。这些协议可直接被多媒体应用程序（例如，IP 语音或视频流[MSK00]）使用，这些应用允许一定程度的数据丢失。UDP/IP 还支持不可靠的多播（multicast）[DC90]和广播（broadcast）功能，允许一个发送者和一组接收者进行通信。

面向连接协议提供的是可靠、有序、不重复的发送服务。对那些不允许数据丢失的应用程序来说，这种协议十分有用。为了提高性能并保证可靠性，“面向连接”协议在发送端和(或)接收端交换并维护状态信息。传输控制协议(TCP，Transmission Control Protocol)是一种面向连接协议，它被运用于很多“面向会话(session-oriented)”的 Internet 应用中，如 Web 服务和电子邮件。

当使用面向连接协议时，应用程序和中间件开发者还必须作出以下设计选择：

- **数据成帧策略（data framing strategies）**。“面向连接”协议提供了不同种类的数据成帧策略。例如，“面向消息”发送策略为某些“面向连接”协议所支持，如 TP4 和 XTP。相反，TCP 是一种字节流（bytestream）协议，不保护应用程序消息的边界（boundaries）。因此，在 TCP 上，如果某一应用程序通过 4 个 send()调用传输 4 条不同的消息，会有一个或多个（可能大于 4 个）TCP 数据段（segment）被传输给接收端。所以，如果某一应用程序需要“面向消息”发送，发送端和接收端就必须执行额外的处理，以将 4 条“在 TCP 上交换”的消息分割成帧。如果消息的长度总相同，并且永远没有网络错误发生，成帧相对来说简单；否则，这会成为一个不小的问题。

- **连接多路复用策略（connection multiplexing strategies）**。在“面向连接”协议上传输数据，有这样两个一般性的策略，参见图 1.1 的描述和下面的说明。

1. **多路复用（multiplexed）**。一个进程中的多个线程发出的所有客户请求都通过

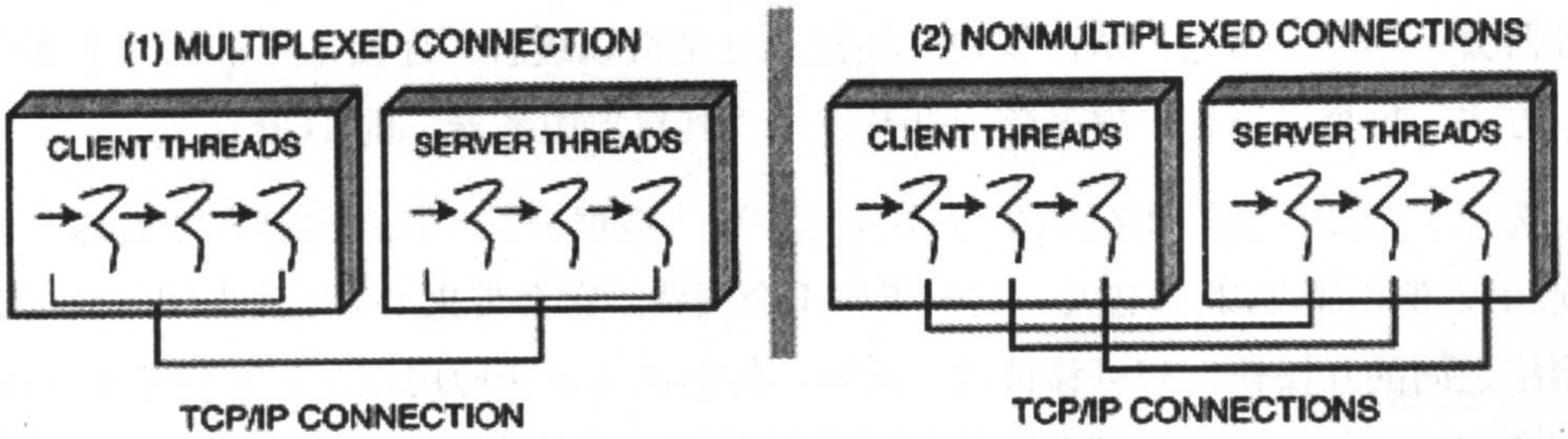


图 1.1：可选的连接多路复用策略

一条 TCP *连接*传递给一个服务器进程，如图 1.1(1)所示。连接多路复用的一个优点是，节省 OS 的通信资源，如 socket 句柄和连接控制块。这个策略的缺点是，难以编程[AOS+00]、缺乏效率[CB01]、缺乏确定性[SMFGG01]。这些问题源自“同步”机制和“环境切换”机制；在多路连接上，这些机制用来将“请求”和“应答”联系起来。

2. **非多路复用（nonmultiplexed）**。每一个客户都使用不同的连接和对等服务程序通信，如图 1.1(2)所示。“非多路复用连接”的主要优点是，可以更好地控制通信的优先级。在“非多路复用连接”设计中，每一个连接的优先级可以单独设定。这样一来，即使其他线程之间存在大量低优先级通信任务，高优先级通信也可以快速完成。这种设计避免了*优先级倒置（priority inversion）*现象——即低优先级线程占用一个单独的多路连接。此外，因为连接没被共享，这个策略的同步开销很小，因为发送和接收双向请求时不需要额外的锁定工作。和“多路复用连接”策略相比，“非多路复用”策略需要使用更多的 OS 资源，因而，在某些环境（譬如，大容量的 Internet 电子商务服务器）中，其伸缩性可能不是很好。

关于连接“多路复用”策略的设计及其利弊，请参阅[SMFGG01, SSRM00]提供的其他资料。

日志服务程序 => 在网络日志服务程序的实现中，我们通过“面向连接”的 TCP/IP 协议将日志记录从客户应用程序传输给日志服务器。我们只建立一次连接，并在客户/

服务器会话期间保留它，从而使建立连接的开销得以缓解。连接采用的是“非多路复用”方式，每个日志客户程序分别打开一个“到日志服务器”的 TCP 连接。

选择了 TCP，就需要在 TCP 字节流上实现“数据分帧”机制（副栏 9 就这一点作了演示）。但由于 TCP 的普及，这一工作不会白费，它会带来 OS 平台之间、数据链路层网络之间的互操作性和可移植性。此外，传输层（而非应用程序）负责流量（flow）控制和拥塞（congestion）控制，负责重新传输丢失的数据包，并保证数据以正确的顺序发送。对那些不能丢失日志记录的应用程序来说，这些能力很重要。

1.2 同步及异步消息交换

在很多网络应用程序中，客户向服务器发送一个请求，后者处理这个请求，然后返回一个应答。这种请求/应答协议既可以在无连接协议上实现，也可以在面向连接协议上实现。管理请求/应答协议交换的可选策略有两种：同步（synchronous）和异步（asynchronous），如图 1.2 所示。是选择同步还是异步消息交换策略，要考虑两个以下因素：

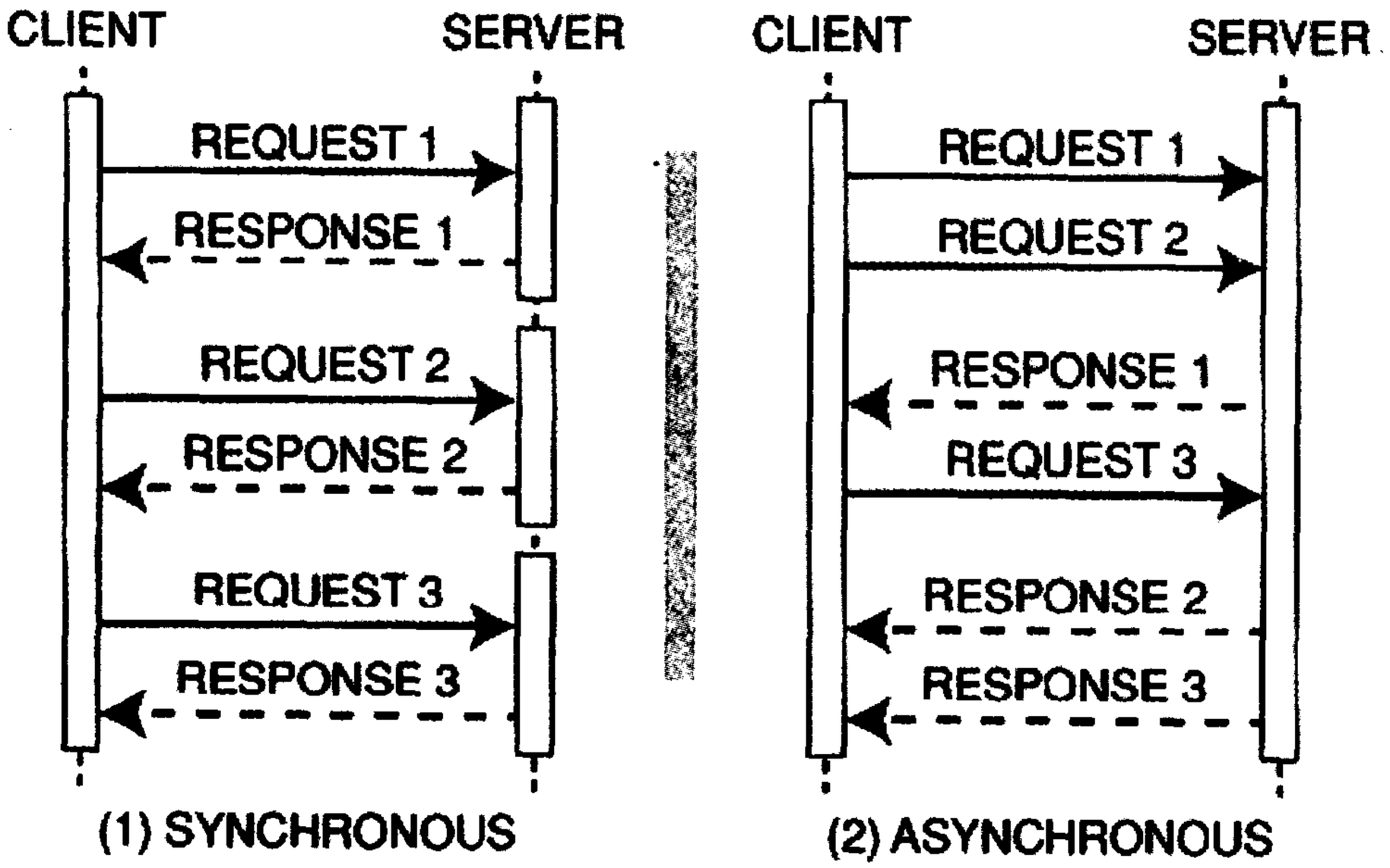


图 1.2：同步及异步消息交换策略

（1） 请求之间的关联性（interrelatedness）；

（2） 底层协议或传输介质的延迟。

以下，我们来看看这些因素如何影响到策略的选择。

同步请求/应答协议。这是最容易实现的形式。在这种协议中，请求（request）和应答（response）是以*锁步（lock-step）*次序交换的。每一个请求必须同步地接收到一个应答，然后才能发送下一个请求，如图 1.2(1)所示。同步请求/应答协议适用于以下场合。

• 请求的结果决定后续的请求。例如，某一应用程序需要交换验证信息，那么，在安全认证信息成功交换之前，它不会发送机密信息请求。

• 应用程序中交换的信息需要在低延迟（low-latency）网络中执行短期处理，如高速 LAN 中的 NFS read()和 write()操作。

• 较之后文介绍的异步请求/应答协议有可能获得的“性能上的提高”，“实现（implementation）的简易性”或“协议交换的少量性”更为重要。

异步请求/应答协议。它将请求从客户连续发送至服务器，无需同步地等待应答[AOS+00]。多个客户请求可以在服务器应答到达之前发送，如图 1.2(2)所示。因此，异步请求/应答协议往往需要一种策略，用来检查请求的丢失或失败，然后重新发送。异步请求/应答协议适用于以下场合：

• 无需应答就可以决定后续请求。例如，Web 浏览器可以使用异步策略，用于从同一服务器获取多个嵌入图像。因为每一个请求都是独立的，它们全都可以异步发送，不必等待应答。浏览器利用每一个应答中包含的信息，将应答匹配至相应的请求——即使应答的顺序不同于请求的发送顺序。

•“通信的延迟”和“请求所需的处理时间”密切相关。异步请求策略有助于有效地利用网络，减少高延迟带来的影响。较之将“应答”和“请求”关联起来并实现“重试”策略所带来的额外复杂度，它在性能上的提高更为重要。

日志服务程序 => 在网络日志服务器中，我们使用的是异步请求/应答协议的单向形式（one-way variant），所以不需要应用程序级（application-level）响应。日志记录只是从客户应用程序传给日志服务器；也就是说，不需要服务器在应用程序级做出应答。日志服务器在接收到每一条日志记录后，都会将记录立即写入磁盘，并假设每一条发送出来的日志记录都被可靠地记录了下来。只要客户程序不需要网络日志服务程序采取“强烈”手段来确保所有日志记录都被永久存储——即使有灾难性的故障发生——这个设计就已经足够了。如果真的有那样的需求，我们就得开发“基于事务（transaction-based）”的日志服务；那将会更复杂，还会带来显著的时间/空间开销。

1.3 消息传递与共享内存

到目前为止，我们关于协议（protocol）的讨论都基于一个假设：信息是在不同计算机的对等体之间传输。这类交换（传输）被称为“消息传递”。如果通信实体可以访问一块共享内存，就会导致另一种不同的信息交换方式。本节将讲述这两种机制，以及它们的利弊。

*消息传递（message passing）*明确地通过 IPC 机制来交换字节流（bytestream）和“面向记录（record-oriented）”的数据；关于 IPC 机制，我们将在第 2 章讲述。通常，应用程序开发者得定义这些信息的格式和内容，以及应用协议；参与者必须遵守这一协议，以交换信息。这一协议还定义了每个交换中可能的参与者的数量（例如，“点对点”（单播）、多播，或广播），以及参与者如何启动、管理和终止“消息传递”会话。

“消息传递”IPC 机制通过 IPC 信道（channel），将数据以消息的形式从一个进程或线程传输给另一个进程或线程，如图 1.3(1)所示。如果要传输的数据量很大，这些数

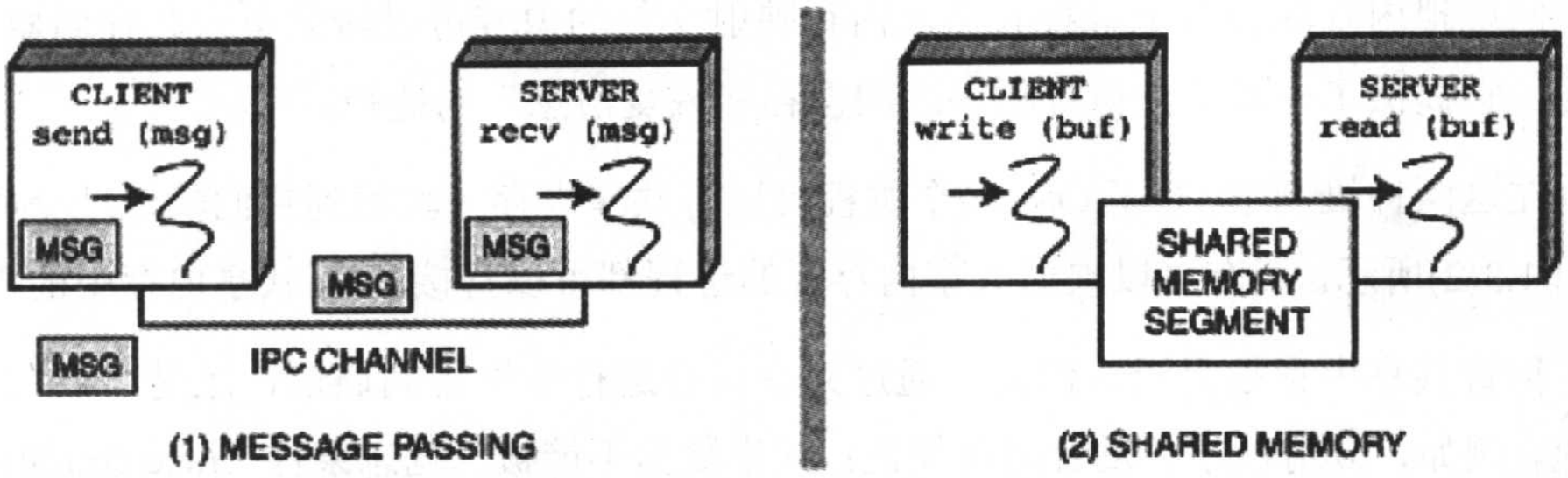


图 1.3：消息传递与共享内存

据就会被分段，以消息序列的形式发送。如果有一个以上的进程接收数据，则每一条消息会发送多次，每一次针对一个接收者。很多流行的中间件体系结构，譬如，RPC、CORBA 和“面向消息”中间件（MOM, message-oriented middleware），其内部都基于“消息传递”通信模型。

*共享内存（shared memory）*允许相同或不同主机上的多个进程访问、交换数据，就像数据位于每一个进程的本地地址空间一样。在网络应用程序中，如果数据必须被多个进程读取和（或）处理，那么，较之“消息传递”，“共享内存”设施将是一种更有效的通信机制。如果要在应用程序中使用原始（native）的 OS“共享内存”机制，则应用程序不用定义“信息在实体间的传输方法”，它要定义如何定位（locate）和映射（map）共享内存区，还要定义“放在共享内存中”的数据结构。

共享内存有“本地（local）”和“分布式（distributed）”两种形式，如下文所述。

• **本地共享内存（Local shared memory）**允许同一计算机上的进程拥有一个或多个共享内存区，这些内存区被映射为不同的虚拟地址范围。常见的共享内存机制有两种：

1. **System V UNIX 共享内存。**shmget()系统函数创建一块新的共享内存区，或返回一块可用的共享内存区。进程可以通过 shmat()系统函数将一块共享内存区挂接（attach）到它的虚拟地址空间上。

2. **内存映射文件。**一个文件的“全部”或“局部”可以映射到一块虚拟内存区

中，此虚拟内存区被多个进程共享。内存映射文件的内容可以转储至永久存储器中；因而，它提供了一条“在执行程序之间保存和恢复信息”的捷径。

在这两种机制中，有了 OS，多个进程可以将共享内存区映射到它们的地址空间中，如图 1.3(2)所示。所有可以映射共享内存区的进程都可以直接读/写共享内存中的内容。

尽管共享内存很灵活，但是，通过共享内存进行多进程间通信，还是需要细致的编程。例如，应用程序开发者必须保证：共享数据不能被“竞态条件（race condition）”破坏。在第 10 章，我们将讲述系统范围（system-scope）同步机制，这种机制可以对一块共享内存区实施*串行化（serialization）*控制，使得共享信息能够以一种有序的方式被访问。同样，如果应用程序需要在共享内存中存储 C++对象，也得认真设计，如副栏 3 所述。

副栏 3：C++对象与共享内存

在 C++中， new 运算符可用来在共享内存中初始化 C++对象。这一特性对 concrete class（具体类）[Bja00]来说是可行的，因为每一个 concrete class 都包含“提供这一支持所需的全部操作”。*concrete class* 中的所有方法（method）都是非虚方法（nonvirtual）；也就是说，它们是被直接调用的，而不是通过函数指针间接调用。很多 ACE wrapper facade 都是 concrete type（具体类型），原因参见附录 A.6 的论述。

相反，*abstract* type（*抽象*类型）包含虚方法，如果将其置于共享内存中，则在编写程序时将很棘手。通常，虚方法是间接调用的，它要借助一个“位于对象内存中”的函数指针表（vtable）。在每一个映射了共享内存的进程中，此共享内存区可能位于不同的虚拟内存区域（location）[BC94、Jor91]。而且，C++编译器/链接器不一定会将 vtable 放在各个进程的相同地址中。因此，vtable 及其指向的函数可能会被映射到各个进程中的不同虚拟地址中，因而必定会在运行时产生问题。

• **分布式共享内存（DSM）**是程序设计中的一种抽象，它扩展了 OS 虚拟内存机制[JM98]，以供应用程序所需。虚拟内存提供了一组策略（policy），用于在需要时获取

（fetching）、放置（placing）、替换（replacing）一组内存页，从而给程序造成一种假象，认为自己拥有一块比实际物理内存大得多的地址空间。DSM 在网络上扩展了虚拟内存的概念，以通过全局/共享内存中的数据进行透明的进程间通信。它体现了两种计算范式（computing paradigm）的结合：共享内存多处理器和分布式系统[PTM97]。

例如，在一些硬件/软件平台上，多台计算机被群集（cluster）为一套逻辑系统，系统中的内存在群集（cluster）中共享。应用程序通过共享内存通信，共享内存由网络计算机共同管理。DSM 系统提供了相应的机制，以配合进程和网络节点的更新。通常，这种通信模型会在底层使用“消息传递”机制，用来协调数据交换，使读取者和写入者同步。

日志服务程序 => 在网络日志服务程序中，我们使用了一种实现在 TCP/IP 之上的“消息传递”IPC 模型。一般来说，较之开发 DSM 机制并通过复杂的“分布式高速缓存一致性协议（distributed cache coherency protocol）”来管理它，使用消息传递 IPC 更实际。由于 DSM 是一项高级研究课题，不常见于实际应用之中，所以，在本书中不会对其作进一步讨论。关于 DSM 方面的研究，介绍得比较好的是参考文献[NL91, PTM97]。

1.4 小结

一般来说，在独立（stand-alone）应用程序中，位于单个地址空间中的各个实体（entities）是通过“在函数调用中传递参数”和（或）“访问全局变量”相互作用的。但在网络应用程序中，位于不同地址空间中的实体需要的是不同的“进程间通信”机制。本章讲述 IPC 机制如何产生几个通信设计空间，使得你在设计网络应用程序时必须考虑它们。我们讲述了连接模式、请求/应答协议、信息交换的选择方案，以及一些“指导你做出选择”的因素和需要权衡的地方。

第 2 章讨论 Socket API。这是在你实现自己的设计时会碰到的最常见的 IPC 工具；所以，了解它的能力和缺点以及 ACE 的解决方案十分重要。ACE 工具包提供了一套类，指导你根据本章讲述的设计空间，做出通信上的选择。例如：

• 封装了“面向连接”和“同步消息传递”功能的 ACE wrapper facade 在第 3 章中介绍。

• 异步连接建立和数据传输在参考文献[SH]中说明。

• ACE“内存映射文件”机制在副栏 7 中介绍；ACE“共享内存”机制的说明见参考文献[HJS]。

在网络应用程序中运用 ACE 的这些功能，将有助于减少“固有”及“偶发”复杂性。

第 2 章

Socket API 概述

本章提要

在第 1 章，我们集中建立了网络程序通信的设计空间模型，从而为解除“固有”复杂性提供了很多机会。本章开始深入分析方案空间（solution space）：首先讨论最常见的进程间通信（IPC）机制 *Socket*，讲述 Socket API 及其在 TCP/IP 中的应用；接着分析程序员在使用原始 OS API 开发网络应用程序时，会遇到哪些常见的“偶发”复杂性。

2.1 操作系统 IPC 机制概述

为了使客户和服务器之间能够交换信息，网络应用程序需要使用“进程间通信（IPC）”机制。操作系统提供的 IPC 机制一般可分为两类：

- **本地 IPC**——某些 IPC 机制，如共享内存（shared memory）、管道（pipe）、“UNIX 领域”socket、门（door）、信号（signal）等，只允许“位于同一计算机上”的实体之间通信[Ste99]。

- **远程 IPC**——其他 IPC 机制，如“Internet 领域”socket[Ste98]、X.25 电路、Win32 命名管道（named pipe），允许“配置或分布在一个网络上”的实体之间通信。

网络应用程序大都和“远程 IPC”机制有关，但是，我们在本书提供的模式和 ACE wrapper facades 对大多数“本地 IPC”机制也适用。

OS IPC 机制及其 APIs 的完整介绍超出了本书的范围。要想全面了解它们，请参考文献[Rag93, MBKQ96, Ric97, Sol98, Ste98, Ste99]。本章集中讲述 Socket API 和 TCP/IP，因为它们分别是最常见的 IPC API 和网络协议；同时向你展示：依靠 Socket API 为异种（heterogeneous）平台编写可移植、健壮的程序，该有多么困难。在第 3 章和第 4 章，我们将演示 ACE 是如何运用 C++和模式来解决这些问题的。

2.2 Socket API

Socket API 最初是在 BSD UNIX[MBKQ96]中开发的，用来为 TCP/IP 协议簇提供应用程序级接口；后来，被移植到大多数操作系统中，包括 ACE 支持的所有操作系统。现在，它是在 TCP/IP 上编写进程间通信的“事实上”的标准。

在应用程序中可以用 Socket API 中的 C 函数来创建、管理本地通信*端点(endpoint)*，即 socket（套接字）。每一个 socket 可以通过一个*句柄（handle）*访问；在 UNIX 环境下，句柄也被称作描述符（descriptor）。一个 socket 句柄可以标识 OS 维护的一个通信端点，同时还可以使程序不受“底层 OS 内核”实现细节的影响，而且不依赖于那些细节，譬如：

- 在 UNIX 中，对大多数操作（operation）来说，socket 句柄和其他 I/O 句柄（例如，文件、管道和终端设备句柄）可以互换（interchangeably）使用。
- 在 Microsoft Windows 中，对大多数操作来说，socket 句柄和 I/O 句柄不能互换使用，虽然它们的使用目的类似。

每一个 socket 都可以绑定（bound）至一个本地地址和一个远程地址上。这些地址定义了两个或多个“通过 socket 通信”的对等实体（peers）之间的关系。

Socket API 包含大约二十多个系统函数，这些函数可以分为以下五类：

1. **本地环境管理**。Socket API 提供了一些函数，用来管理本地环境信息（local context information），这些信息通常存储在 OS 内核或系统库中。

函　数	说　明
socket()	工厂（factory）函数。用于分配一个 socket 句柄，并返回给调用者。
bind()	将一个 socket 句柄和一个本地或远程地址关联起来。
getsockname()	返回 socket 绑定的本地地址。
getpeername()	返回 socket 绑定的远程地址。
close()	释放 socket 句柄，使其可复用。

2. **连接的建立和终止**。Socket API 提供了一些函数，用来建立和终止连接。

函　数	说　明
connect()	*主动*在一个 socket 句柄上建立连接。
listen()	表示愿意*被动*侦听来自客户的连接请求。
accept()	*工厂*（factory）函数。响应客户请求，创建一个新的连接。
shutdown()	有选择地终止双向连接中读取方和（或）写入方的数据流。

3. **数据传输机制**。Socket API 提供了一些函数，用来通过 socket 句柄发送和接收数据。

函　数	说　明
send() recv()	通过某一特定的 I/O 句柄，传送和接收数据缓冲区。
sendto() recvfrom()	交换“无连接”数据报。其中，每一个 sendto()调用都要提供接收方的网络地址。

在 UNIX 上，这些函数还可用于其他类型的 I/O 句柄，如文件和终端设备。UNIX 平台还提供了下面的数据传输机制：

函　数	说　明
read() write()	通过某一句柄，接收和传送数据缓冲区。
readv() writev()	分别支持“分散读取”和“集中写入”语义，以优化“模式切换”，简化“内存管理”。
sendmsg() recvmsg()	通用函数，包含其他数据传输函数的行为。

4. **选项管理**。Socket API 定义了一些函数，允许程序员改变缺省的 socket 行为，从而支持多播和广播，并能修改/查询传输缓冲区的大小：

函　数	说　明
setsockopt	在协议栈的不同层修改选项。
getsockopt	在协议栈的不同层查询选项。

5. **网络地址**。除了以上介绍的函数之外，网络应用程序还经常使用一些函数来将具有可读性的名称（如 tango.ece.uci.edu）解析为低级网络地址（如 128.195.174.35）：

函　数	说　明
gethostbyname() gethostbyaddr()	处理主机名和 IPv4 地址之间的网络地址映射。
getipnodebyname() getipnodebyaddr()	处理主机名和 IPv4/IPv6 地址之间的网络地址映射。
getservbyname()	通过具有可读性的名称标识服务（service）。

虽然 Socket API 常用来编写 TCP/IP 应用程序，但它用途之广，足以支持多个通信领域（communication domain）。一个通信领域由一个“协议簇”和一个“地址簇”确定，如下所述。

• **协议簇（protocol family）**。今天的网络环境包含大量协议，这些协议可以提供广泛的通信服务，如“面向连接”的可靠传送、不可靠的多播，等等。一个协议簇是一个协议集合（collection），用来提供一组特定的相关服务。在通过 Socket API 创建 socket 时，协议是通过以下两个参数共同指定的。

1. **协议簇**——例如，“UNIX 领域”（PF-UNIX）、“Internet 领域”的 IPv4（PF-INET）和 IPv6（PF-INET6）、ATM（PF-ATMSVC）、X.25（PF-X25）、Appletalk（PF-APPLETALK），等等。

2. **服务类型**——例如，有序、可靠的字节流（SOCK_STREAM）、不可靠的*数据报*（SOCK_DGRAM），等等。

例如，通过传递 PF_INET（或 PF_INET6）和 SOCK_STREAM 标志给 socket()函数，就指定了 TCP/IP 协议。

• **地址簇（address family）**。地址簇定义地址的格式，表达地址的字节大小，以及它所包含的字段（field）的数量、类型、顺序等。此外，地址簇还定义了一组解释地址格式的函数，可用来——举例来说——确定 IP 数据报将要到达的子网。地址簇和协议簇密切相关，例如，IPv4 地址簇 AF_INET 只能用于 IPv4 协议簇 PF_INET。

2.3 Socket API 的局限性

原始 Socket API 有几个局限性：容易出错、过于复杂、不可移植或不统一。以下讨论虽然针对的是 Socket API，但也适用于其他原始的 OS IPC APIs。

2.3.1 容易出错的 API

如第 2.2 节所述，Socket API 通过句柄标识 socket 端点。一般来说，操作系统使用句柄来标识其他 I/O 设备，如文件、管道、终端。这些句柄是通过*弱类型*（weakly typed）的整数或指针类型实现的，因而会在运行时引发微妙的错误。为了演示它所产生的各种问题，给出下面这个 echo_server()函数：

```
0 // This example contains bugs!  Do not copy this example!
1 #include <sys/types.h>
2 #include <sys/socket.h>
3
4 const int PORT_NUM = 10000;
5
6 int echo_server ()
7 {
8   struct sockaddr_in addr;
9   int addr_len;
10  char buf[BUFSIZ];
11  int n_handle;
12  // Create the local endpoint.
13  int s_handle = socket (PF_UNIX, SOCK_DGRAM, 0);
14  if (s_handle == -1) return -1;
15
16  // Set up the address information where the server listens.
17  addr.sin_family = AF_INET;
```

```
18  addr.sin_port = PORT_NUM;
19  addr.sin_addr.addr = INADDR_ANY;
20
21  if (bind (s_handle, (struct sockaddr *) &addr,
22            sizeof addr) == -1)
23    return -1;
24
25  // Create a new communication endpoint.
26  if (n_handle = accept (s_handle, (struct sockaddr *) &addr,
27                         &addr_len) != -1) {
28    int n;
29    while ((n = read (s_handle, buf, sizeof buf)) > 0)
30      write (n_handle, buf, n);
31
32    close (n_handle);
33  }
34  return 0;
35 }
```

这个函数至少包含 10 个微妙的错误，而且全都是使用 Socket API 时十分常见的错误。你可先阅读上面的代码，看看能否找出这 10 个错误；然后阅读以下关于这些缺陷的剖析。括号内的数字是 echo_server()函数中出现错误的行号。

1. **忘记初始化一个重要的变量。（8~9）**addr_len 变量必须被设置为 sizeof(addr)。如果忘记初始化这个变量，在运行时，第 26 行的 accept()调用将会失败。

2. **句柄使用了不具可移植性的数据类型。（11~14）**这几行代码看起来似乎没有什么危害，其实危机四伏。这段代码不能移植到 Windows Sockets（WinSock）平台上。因为在 WinSock 平台上，socket 句柄的类型为 SOCKET，不是 int；而且，WinSock 的失败是用一个非标准的宏（称为 INVALID_SOCKET_HANDLE）来表示的，不是返回 -1。在这几行代码中，还存在几个不太明显的错误，等分析了函数的其余部分之后，你自然会明白。

接下来的 3 个网络地址方面的错误很微妙，它们只是在运行时才会出现。

3. **未使用的 struct 成员未被清零。（17~19）**在设置每一个地址成员之前，整个 addr 结构应被初始化为 0。Socket API 使用了一种很基本的地址结构（sockaddr），根据地址簇不同，结构成员的占位（overlays）也不同，例如，对于 IPv4 来说，它是 sockaddr_in。若不将整个结构设置为 0，部分字段就会包含不确定的内容，因而会产生随机的运行时

错误。

4. **地址/协议簇不匹配。（17）**add.sin_family 字段被设置为 AF_INET，它指定的是 Internet 地址簇。它将被用于一个“通过 UNIX 协议簇创建的”socket（s_handle）。这种用法不一致。其实，传给 socket()函数的协议簇类型应该是 PF_INET。

5. **错误的字节顺序。（18）**赋给 addr.sin_port 的值没有使用网络字节顺序；也就是说，程序员没有使用 htons()将*端口号（port number）*从“主机字节顺序”转换为“网络字节顺序”。如果在使用 little-endian 字节顺序的计算机上运行这段代码，执行时不会出现错误；但是，客户将无法连接到预定的端口号上。

如果这些网络地址错误能被修改过来，第 21~23 行就不会有问题！

第 25~27 行有一组相互关联的错误。它们证实：在直接使用 C Socket API 编程的时候，发现错误该有多么困难。

6. **未调用一个重要的 API 函数。（25）**listen()函数不小心被遗漏了。这个函数必须在 accept()之前调用，以将 socket 句柄设置为所谓“被动模式”。

7. **API 调用中使用了错误的 socket 类型。（26）**accept()函数是针对 s_handle 调用的，这完全没有问题。但是，s_handle 被创建为 SOCK_DGRAM 类型的 socket，对 accept()来说，这是非法的 socket 类型。因而，在前面第 13 行的 socket()调用中，传递的应该是 SOCK_STREAM 标志。

8. **运算符优先级错误。（26~27）**还有一个和 accept()调用有关的错误。就像代码中所写的那样，如果 accept()成功，n_handle 会被设置为 1，否则会被设置为 0。但是，如果程序是运行在 UNIX 上（而且其他错误都已改正），数据就会被分别写到 stdout 或 stdin 上，而不是已连接的 socket 上。尽管使用 ACE Wrapper façade 类，上面例子中的大多数错误都可以避免；但本条错误仅仅是运算符优先级方面的错误，使用 ACE 或其他库（library）也爱莫能助。这是 C 和 C++程序中的常见缺陷 [Koe88]，只有了解了运算符优先级顺序才能补救。为了修正这个错误，可以在赋值表达式外添加一对括号，像下面这样：

```
if ((n_handle = accept (s_handle, (struct sockaddr *) &addr,
                        &addr_len) != -1)) {
```

但更好的做法是，遵循在 ACE 中使用的规范，将 n_handle 的赋值放在单独一行：

```
n_handle = accept (s_handle,
                   (struct sockaddr *) &addr, &addr_len);
if (n_handle ! = -1) {
```

这样，你就不必劳神去记住神秘的 C++运算符优先级规则了！

关于 echo_server()函数，我们的讨论就快要结束了；但还剩下几个错误。

9. **API 调用中使用了错误的句柄。（29）**read()函数被调用后，会从 s_handle 接收多达 sizeof buf 个字节，这是被动模式的“侦听”socket；但我们本应该在 n_handle 上调用 read()。这是一个不明显的运行时错误，在编译时发现不了，因为 socket 句柄属于弱类型（weakly typed）。

10. **数据有可能丢失。（30）**没有检查 write()的返回值，看看所有字节是否都已被写入（或根本没有任何字节写入）。这有可能造成数据丢失。出于 socket 缓冲（buffering）和*流量控制（flow control）*方面的原因，在通过 write()向“字节流”模式 socket 写入数据时，发送的有可能只是一部分字节；如果出现这种情况，必须随后发送剩余字节。

经历过多次挫折之后，对于上面列出的大多数问题，有经验的网络程序员都会有所警觉。但是，更本质的设计问题是，Socket API 缺乏足够的类型安全（type safety）和数据抽象（data abstraction）。在某些 OS 平台上，上面的源代码会顺利通过编译，但有一点：在任何平台上，它都不能正确运行！

多年来，程序员和程序库都采取了很多手段，试图消除这些问题。一个常见的方案是：通过 typedefs 清楚地告诉程序员，代码中所指的是何种类型。这些方案虽然可以消除“可移植性”方面的某些问题，但没有覆盖上文列出的其他问题。特别是，IPC 地址策略中的“多样性”所带来的变化，并不是仅仅一个 typedef 所能隐藏得了的。

2.3.2 过于复杂的 API

Socket API 通过一个接口来支持多个：

- **协议簇**，例如，TCP/IP、IPX/SPX、X.25、ISO OSI、ATM、“UNIX 领域” Socket；
- **通信/连接角色**，例如，*主动连接建立（active connection establishment）*、*被动连接建立（passive connection establishment）*、数据传输；
- **通信优化**，例如，“集中写入” 函数 writev()，用来在一个系统函数中发送多个数据缓冲区；
- **选项**，用来设置不大常用的功能，例如，广播、多播、异步 I/O、紧急数据发送。

Socket API 将这些所有功能都结合到一个 API 中——详见第 2.2 节表格中的内容。其造成的结果复杂且难以掌握。但是，如果仔细分析 Socket API 之后，就会发现，它的接口可以分解到以下几个维度（dimensions）中。

1. **通信服务的类型**，例如，流、数据报、连接的数据报（connected datagram）。
2. **通信/连接角色**，例如，客户通常主动启动连接，服务器通常被动接受这些连接。
3. **通信领域**，例如，“仅本地主机（local host only）”、“本地或远程主机”。

在图 2.1 中，我们依据这三个维度，对相关 Socket 函数进行了分组（clusters）。然而，这种很自然的分组（clustering）在 Socket API 中却被模糊化了，因为所有这些功能都被塞进了惟一一组函数中。而且，Socket API 不能在编译时保证，它的函数会根据不同的“通信”和“连接”角色（如主动/被动连接建立或数据报/流通信）而得到正确的使用。

2.3.3 不可移植、不统一的 API

尽管 Socket API 普遍存在，但它并不具有可移植性。以下是各个平台之间存在的一些分歧：

- **函数名称**——第 37 页 echo_server()函数中的 read()、write()和 close()函数无法移植到所有的 OS 平台上。例如，Windows 定义了一组不同的函数（ReadFile()、WriteFile()、closesocket()）来提供这些行为。

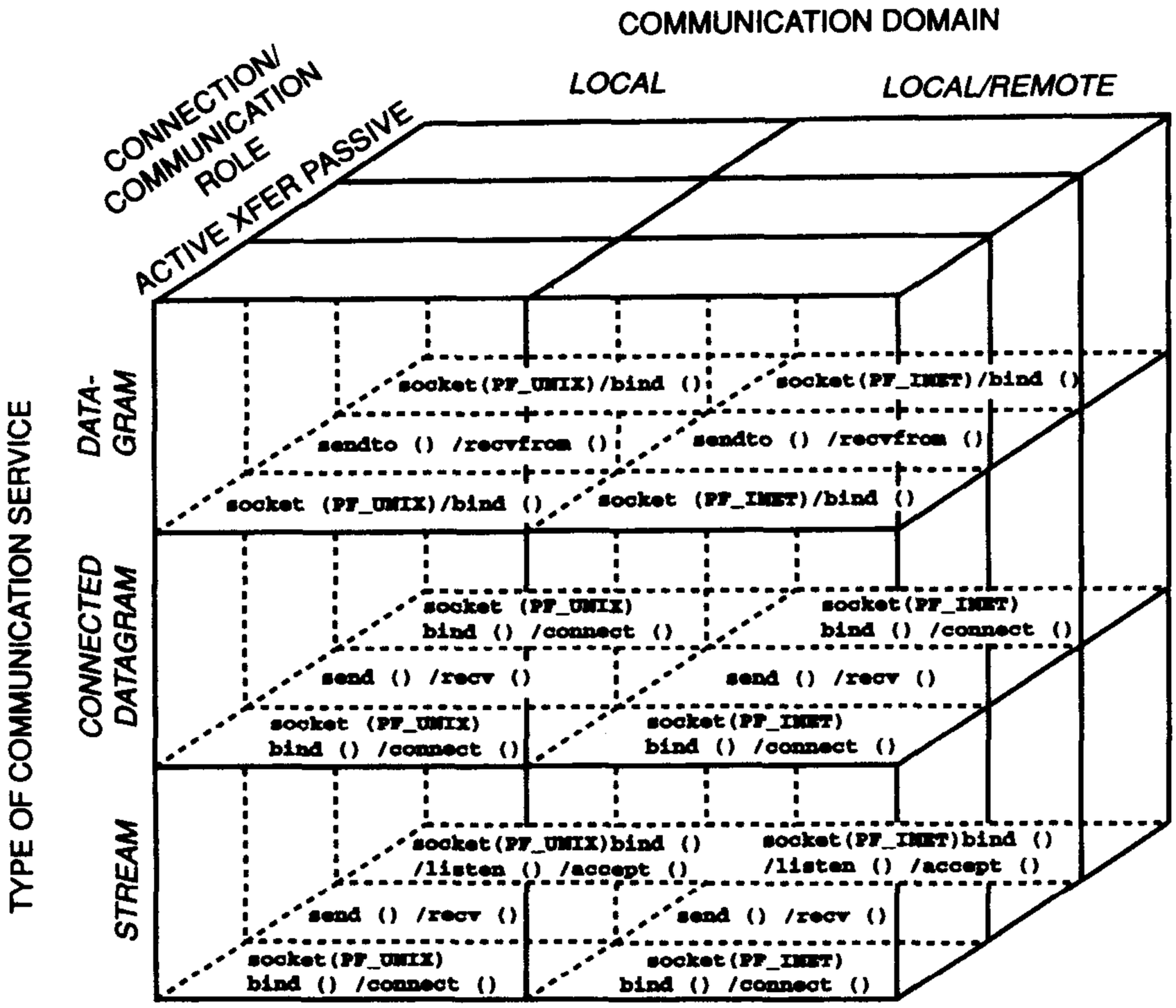


图 2.1：Socket 空间分类图

• **函数的语义**——某些函数在不同的平台上表现出不同的行为。例如，在 UNIX 和 Win32 中，可以将 NULL 指针传给 accept()函数，作为客户地址和客户长度字段。但在某些实时平台如 VxWorks 上，将 NULL 指针传给 accept()会使计算机系统崩溃。

• **Socket 句柄的类型**——对于 socket 句柄，不同的平台使用了不同的表示法。例如，在 UNIX 平台上，socket 句柄是整数；但在 Win32 上，它们使用的实际上是指针。

• **头文件**——在不同的 OS/编译器平台上，“包含 Socket API 函数原型”的头文件使用的是不同的名称。

Socket API 的另一个问题是，它的几十个函数缺乏统一的命名规范，这使得我们很难确认 API 的“范围（scope）”。例如，socket()、bind()、accept()和 connect()同属一个 API，但这一点并不直观。其他网络 APIs 为了解决这个问题，会在每个函数之前添

加一个共同的前缀。例如，在 TLI API [Rag93]中，每个函数之前都被添加了一个 t_。

2.4 小结

无论何种工作，如果想出色地完成它，一个前提是：要知道如何有效地使用工具。很多工具很有用，但如果误用它，反而会伤害到你。没有人曾被 UNIX“管道（pipe）”或“套接字（socket）”击中脑袋；但是，在不必要的开发和调试中浪费时间和资源，确实会消耗大量的金钱，并让人华发早生，还让人失去很多参加社会活动的时间。此外，和学习使用扳手、锤子这类工具不同，在将你的网络应用程序移植到新的平台时，学习使用那个平台上的“低级”工具没有什么帮助。

软件开发者在构造网络应用程序时，必须掌握“和进程间通信（IPC）相关”的概念和工具。本章介绍了 Socket API，讨论了这一普遍存在的 API 提供的能力范围，以及与其相关的一组偶发复杂性。我们给出了一个简短的程序，演示了这些复杂性造成的错误的类型。在第 3 章，我们将演示 ACE 如何运用 C++特性和 Wrapper Facade[SSRB00]模式来解决这些问题。

第 3 章

ACE Socket Wrapper Facade

本章提要

本章介绍的是 ACE 中使用的模式（pattern）和它提供的 wrapper façade 类，可用来解决第 2 章所说的“Socket API 中存在的偶发复杂性”问题。这些 ACE classes 使用了 Wrapper Facade 模式，将“面向连接”的 Socket API 功能封装成了可移植的 C++类。我们还演示了这些类的用法，采用的例子是一个简短、熟悉的 Web 服务器和客户程序。在第 4 章，我们还将使用一个较大型的网络日志服务程序示例。

3.1 概述

ACE 定义了一组 C++类，用来解决第 2.3 节提及的“Socket API 的局限性”。这些类都是根据 Wrapper Facade 模式[SSRB00]设计而成的。[1] 它们将现有的“非面向对象”API 所提供的函数和数据封装到了简明、健壮、可移植、可维护、一致的“面向对象”类接口中。本章提供的 ACE Socket wrapper facade 类有：

[1] 在本书余下部分，我们用 *ACE wrapper facade* 这个术语来表示“通过 Wrapper Facade 模式设计出来的 ACE 类”。

ACE 类	说 明
ACE_Addr	ACE“网络地址”继承结构的根（root）。
ACE_INET_Addr	这个类封装了“Internet 领域”地址簇。
ACE_IPC_SAP	ACE IPC wrapper facade 继承结构的根。
ACE_SOCK	ACE Socket wrapper facade 继承结构的根。
ACE_SOCK_Connector	这是一个工厂（factory），它连接至一个对等的接收者，然后在一个 ACE_SOCK_Stream 对象中初始化一个新的通信端点。
ACE_SOCK_IO ACE_SOCK_Stream	它们封装了“数据模式（data-mode）”socket 支持的数据传输机制。
ACE_SOCK_Acceptor	这是一个工厂，它在一个 ACE_SOCK_Stream 对象中初始化一个新的通信端点，对“来自对等连接者的连接请求”做出响应。

图 3.1 演示了这些类之间的主要关系。这些 ACE Socket wrapper facade 提供了以下好处。

- **提高了“类型安全”**，可以很快发现应用程序中很多微妙的“类型错误”，例如，若用于“被动和主动建立连接”的工厂（factory）没有提供“发送和接收数据”的方法（method），那么，类型错误在编译时就可以发现，而不是在运行时。

- **保证了可移植性**，使用的是“和平台无关”的 C++类。

- **简化了常见的使用情况**，因为减少了应用程序代码量，节省了耗费在“低级网络编程细节”上的开发量，开发者可以将注意力集中在高级的、“以应用为中心”的问题上。

此外，ACE Socket wrapper facade 还保留了效率。它通过使用内联函数，提高了上文列出的各项软件质量，且没有牺牲性能。关于 ACE 中“和效率有关”的设计原则，将在第 A.6 节作进一步讨论。

ACE Socket wrapper facade 的结构（structure）符合图 3.2 中所示的关于“通信服务”、“连接/通信角色”、“通信领域”的分类。这幅图中所示的类提供了以下功能：

- ACE_SOCK_*类封装了“Internet 领域”Socket API 的功能。

- ACE_LSOCK_*类封装了“UNIX 领域”Socket API 的功能。

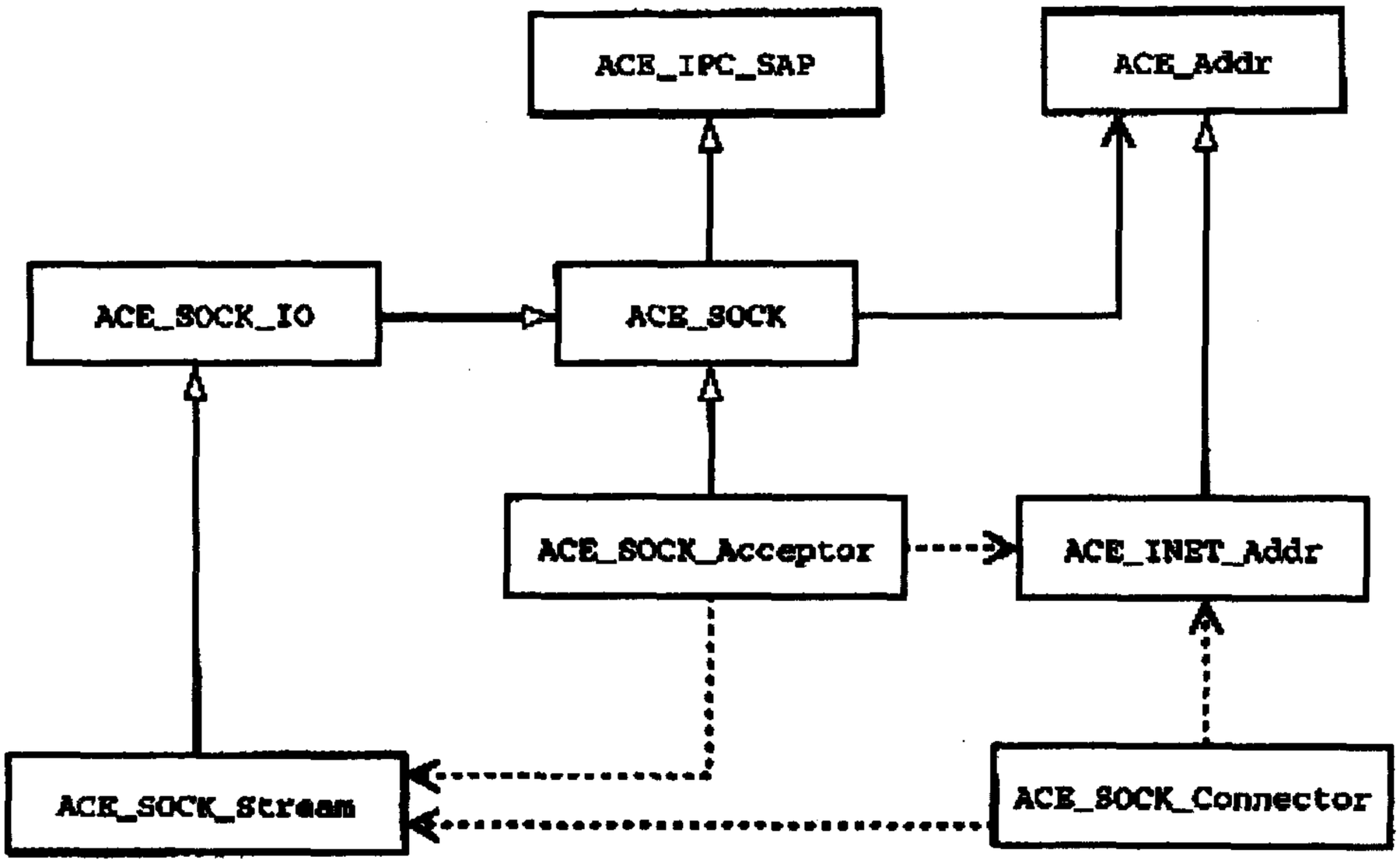


图 3.1：ACE 面向连接的 Socket 类关系图

如果将图 3.2 中所示的 ACE 类的结构和图 2.1 所示的函数作比较，将会受到启发。图 3.2 中所示的类更为简明，因为它们通过 ACE Socket wrapper facade，将多个 socket 函数的行为封装到了“通过继承（inheritance）”相互联系的 C++类中。除了支持“连接建立”和“通信”的类之外，ACE 还提供了一组和“地址”有关的类。在第 3.2 节，将讲述这些类的功能；在第 A.4.2 节，还将讨论相关的设计原则。图 3.1 描绘了这些类同“面向连接”Socket API 系列中其他 ACE 类之间的关系。

在本章余下部分，将集中讨论 ACE 中“面向连接”的“Internet 领域”wrapper facade，因为它们是使用最广的 ACE IPC 机制。这些类分布在图 3.1 所示的继承结构中。为避免代码重复，公共的方法（method）和状态（state）都被提取到了抽象类（abstract class）中，而抽象类是不能被直接实例化（instantiated）的。因而，可以在子类（subclasses）中添加合适的方法，并正确运用模式。例如，在“面向连接”的网络应用程序中，客户和服务器扮演的是“非对称”的连接角色。服务器*被动（passively）*侦听，等待客户*主动（actively）*发起连接[SSRB00]，如图 3.3 所示。即使在*端对端（peer-to-peer）*应用程

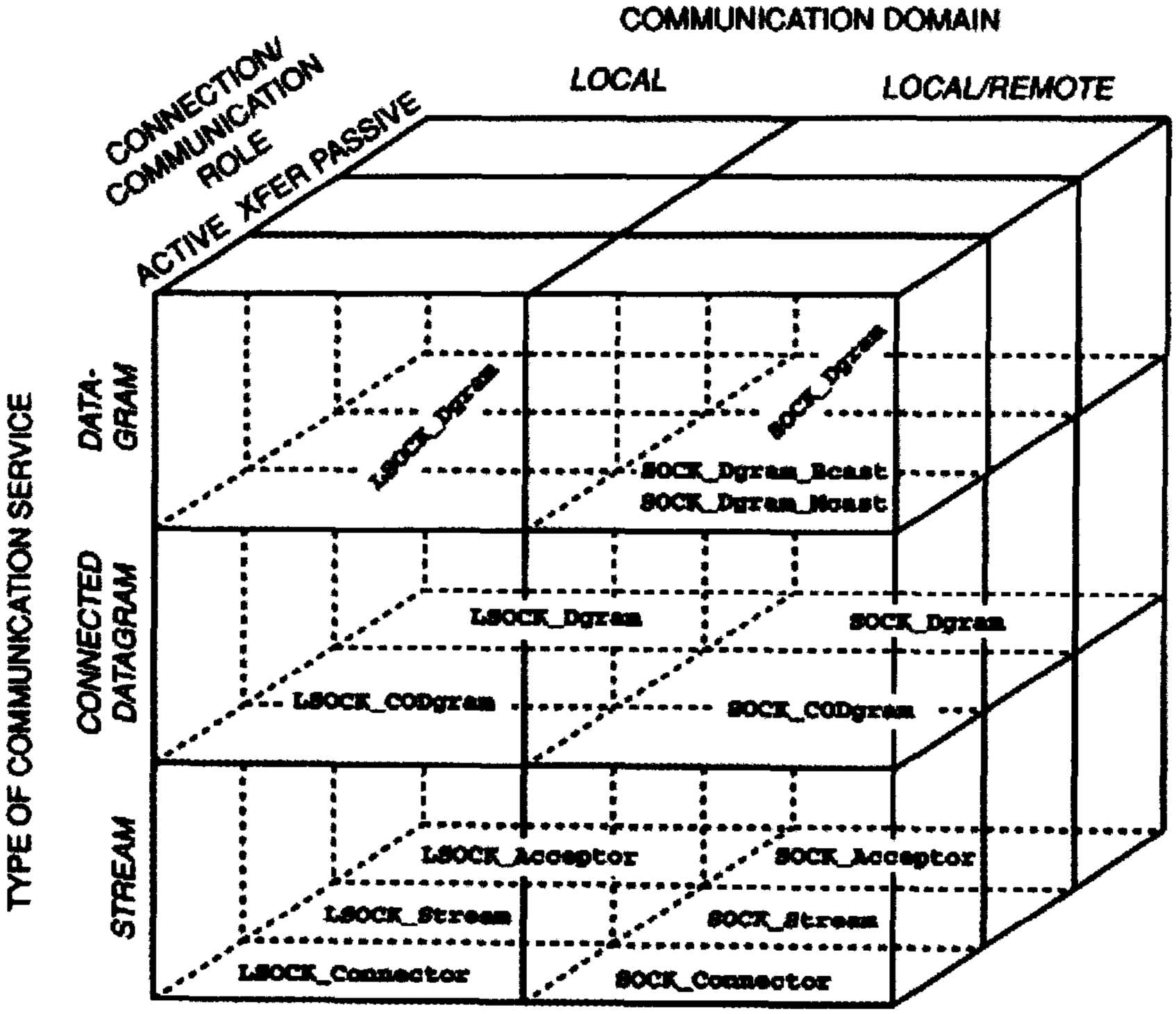


图 3.2：ACE Socket Wrapper Facade 分类图

序中，也就是说，即使应用程序同时扮演客户和服务器这两种角色，连接也必须由其中一方主动发起，另一方被动接受。

为了区分 C Socket API 中的函数，请认清这些函数在面向连接协议（譬如 TCP）中扮演的 3 种不同的角色：

1. 主动连接角色（连接者），由对等应用程序（peer application）中的一方扮演，用来发起一个“对远程对等方（remote peer）的连接”；

2. 被动连接角色（接受者），由对等应用程序中的一方扮演，用于接受一个“来自远程对等方的连接”；

3. 通信角色（流），由对等应用程序中的双方共同扮演，用于在建立连接后交换数据。

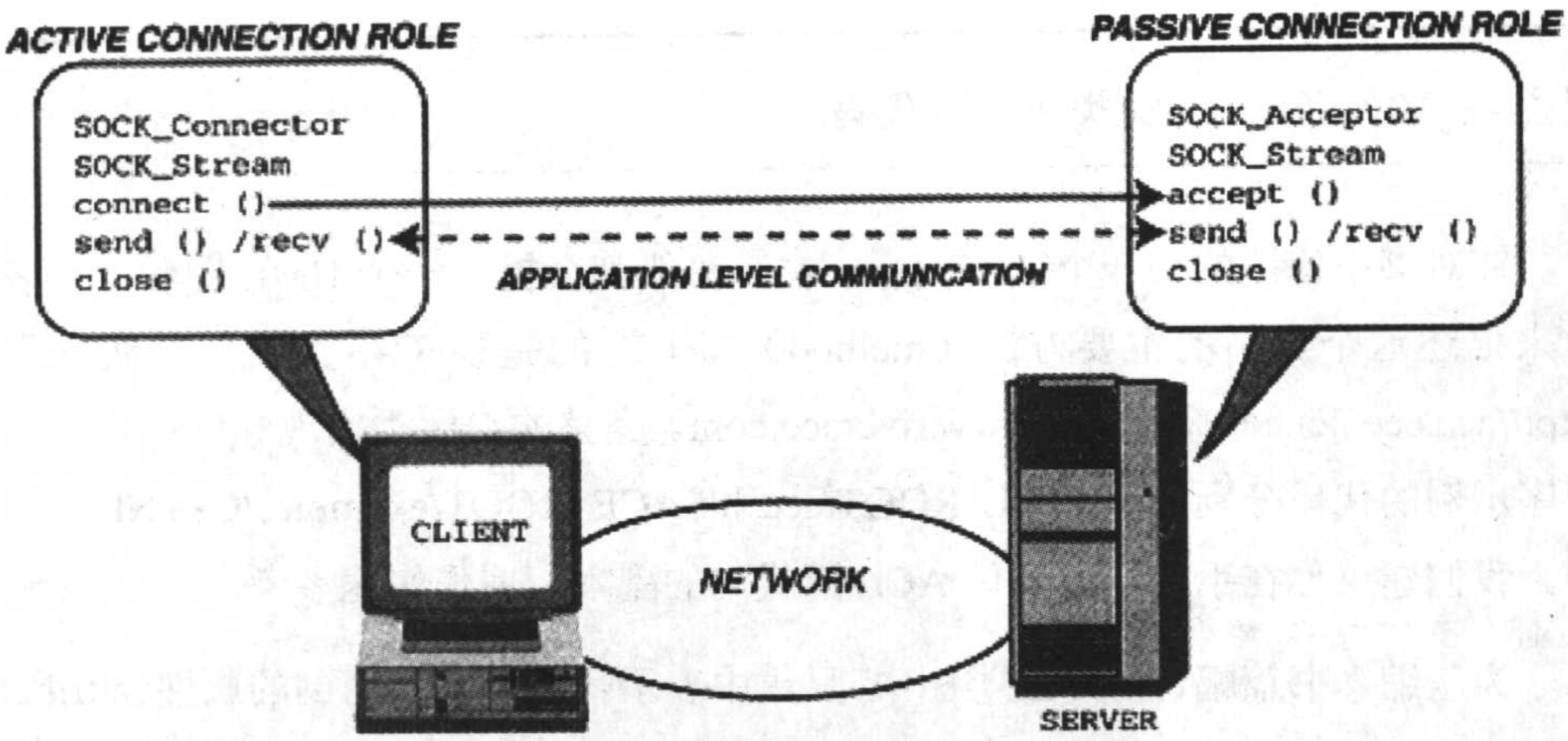


图 3.3：ACE Socket Wrapper Facade 中的角色

ACE Socket 类共同扮演着以上所说的“连接”和“通信”角色。本章将逐一讲述这些类的功能。副栏 4 是关于 UML 图和本书 C++代码的介绍，请牢记其中讲述的要点。

3.2 ACE_Addr 类和 ACE_INET_Addr 类

设计意图

“网络地址”是 Socket API 中的一个薄弱点。在 Socket API 的“网络地址”机制中使用了 C 结构（structures）和类型转换（typecasts），这对编程来说既麻烦又容易出错。在通用地址结构 sockaddr 中，“地址簇（address family）”是第一个成员。其他地址结构，例如，“Internet 领域”地址 sockaddr_in 和“UNIX 领域”地址 sockaddr_un，也有一个“地址簇”成员。和在 sockaddr 结构中一样，这个成员占用相同的位置和大小。应用程序可以像下面这样使用这些特殊的“地址簇”结构：

1. 为预定的“地址簇”分配一个结构，例如 struct sockaddr_in；
2. 填充“地址簇”成员，向 Socket API 表明它的真实类型；
3. 提供地址，譬如 IP 和端口号；
4. 将类型转换为 sockaddr *，然后传递给 Socket API 函数。

副栏 4：如何表示 ACE 类和 C++代码

针对书中使用的每一个 ACE C++ 类，一般都会提供一个 UML 图和一个表，用来描述这个类中的主要方法（method）。ACE 的完整 C++类接口在线提供于 http://ace.ece.uci.edu 和 http://www.riverace.com。完整的 C++类实现和网络日志服务程序示例的代码分别位于$ACE_ROOT/ace 和$ACE_ROOT/examples/C++NPv1 目录中。我们建议你在手边保留一份 ACE 源代码的副本，用作快速参考。

为压缩本书篇幅，在UML类图中，只集中表示代码示例中用到的属性（attribute）和操作（operation）。如果类中没有和讨论直接相关的属性，图中将不会显示这个类的属性区。如果需要快速了解或重温 UML，可以阅读《UML Distilled》[WKS00]。

为进一步缩减篇幅，我们还在编程中使用了一些“ACE 中没有使用”的编程捷径。例如：

- 在 C++示例中，省略了很多“出错处理”代码。当然，ACE 总是会对出错条件进行检查，并采取相应的补救措施，就像你的程序应该做的那样。
- 在一些 C++示例中，方法（method）是在类的定义中实现的。但在 ACE 中，我们不会这么做；因为这会给类接口造成混乱，还会降低编译速度。

ACE 的编程准则请见$ACE_ROOT/docs/ACE_guidelines.html。

为了最大限度地降低这些底层细节的复杂性，ACE 定义了一个类的继承结构，为所有的 ACE“网络地址”对象提供统一的接口。

功能

ACE_Addr 类位于 ACE“网络地址”继承结构的根部。这个类的接口如图 3.4 所示。它的主要方法（method）所提供的功能对所有 ACE“网络地址”类都是公用的，如下表所示。

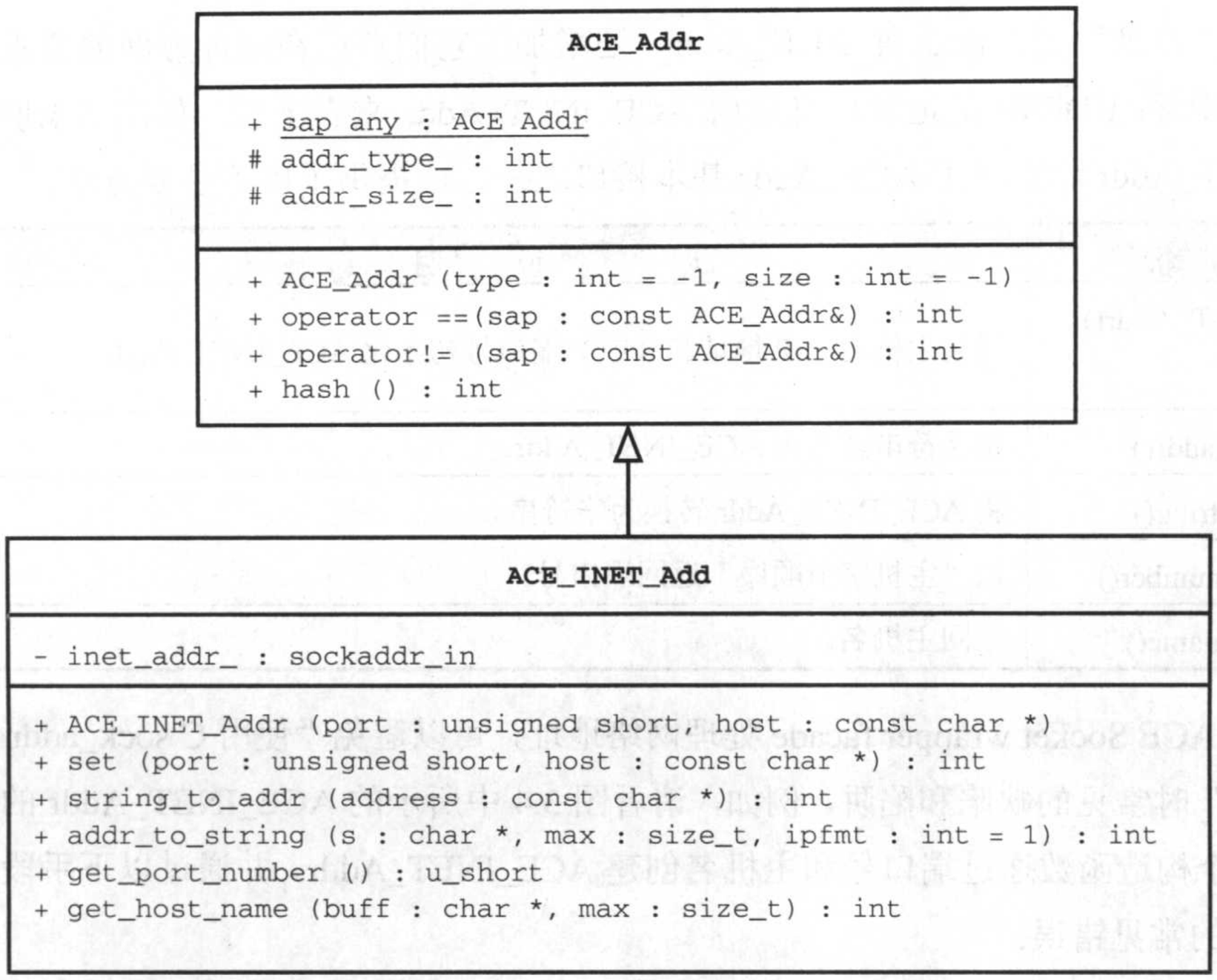


图 3.4：ACE_Addr 和 ACE_INET_Addr 类图

方　法	说　明
operator == ()	比较“地址的相等性”。
operator != ()	比较“地址的不等性”。
hash ()	计算某个地址的散列（hash）值。

ACE_Addr 类还定义了一个静态（static）数据成员 sap_any。如果客户和服务器程序不在意它们被分配为何种地址，可以将 sap_any 用作“通配符”。例如：

•客户程序可以用 sap_any 来创建“OS 分配的”临时端口号，即所谓“短暂端口”；在连接关闭之后，这些端口可以再利用。

• 服务器程序可以通过 sap_any 来选择它们的端口号，只要它们通过某种定位机制（例如，命名或交易服务）向客户输出了被分配的端口号。

在每一种 IPC 机制（譬如，“Internet 领域”socket 和“UNIX 领域”socket）中，

其具体的“地址”类都派生自 ACE_Addr，还增加了它们自己在地址方面的要求。例如，TCP/IP 和 UDP/IP 的地址信息是用 ACE_INET_Addr 来表示的，如图 3.4 所示。ACE_INET_Addr 除实现了 ACE_Addr 基本接口之外，还提供了以下主要方法。

方　法	说　明
ACE_INET_Addr() set()	通过主机名、IP 地址和（或）端口号初始化 ACE_INET_Addr。
string_to_addr()	将字符串转换为 ACE_INET_Addr。
addr_to_string()	将 ACE_INET_Addr 转换为字符串。
get_port_number()	以“主机字节顺序”返回端口号。
get_host_name()	返回主机名。

通过 ACE Socket wrapper facade 处理网络地址，可以避免“使用 C sock_addr 系列数据结构”时常见的缺陷和陷阱。例如，请看图 3.4 中所示的 ACE_INET_Addr 的构造函数。这个构造函数通过端口号和主机名创建 ACE_INET_Addr，并通过以下手段消除了编程中的常见错误：

- 将底层 sockaddr_in 中的所有字节初始化为 0；
- 将端口号和 IP 地址转换为网络字节顺序。

有了 ACE_INET_Addr，在编写网络应用程序时，开发者就无需担心低级地址（low-level address）的初始化细节。ACE_INET_Addr 中还有其他很多重载（overloaded）的构造函数，以及 set()方法。利用它们，我们可以通过“主机名”、“IP 地址”和（或）“TCP/UDP 端口名称和端口号”的各种组合，初始化“Internet 领域”网络地址对象。

3.3 ACE_IPC_SAP 类

设计意图

我们第 2.3.1 节曾讨论过：原始 IPC API 中使用的 I/O 句柄会造成“可移植性”方面的问题。ACE 通过以下方式解决 I/O 句柄的“可移植性”问题：

- 提供了 ACE_HANDLE 类型定义（type definition），在每一个 OS 平台上指定了合适的句柄类型。

• 定义了可移植的 ACE_INVALID_HANDLE 宏，应用程序可用来检测错误。

ACE 自定义的这些“类型和值”的抽象虽然简单，但有助于提高程序的可移植性。

然而，深谙“对象”概念的软件开发者会注意到：一个句柄——即使一个具有“可移植性”的句柄——也不是“面向对象”网络应用程序中合适的抽象层。相反，更合适的编程抽象应该是某种“I/O 句柄”类，这也就是 ACE 提供 ACE_IPC_SAP 类的原因。[2]

功能

ACE_IPC_SAP 类是 ACE IPC wrapper facade 继承结构的根（root）。它为其他 ACE wrapper facade 提供了基本的“I/O 句柄”操作能力。它的接口如图 3.5 所示（图中还给出了第 3.4 节讲述的 ACE_SOCK 类），它的主要方法（method）如下表所示：

方　法	说　明
enable() disable()	启用或禁止各种“I/O 句柄”选项，例如，启用/禁止“非阻塞”I/O。
set_handle() get_handle()	设置（set）或得到（get）底层 I/O 句柄。

虽然 ACE_IPC_SAP 定义了有用的方法和数据，但它的目的并不是供应用程序直接使用。相反，它是供子类（subclass）（譬如针对文件、STREAM 管道、命名管道、System V 传输层接口（TLI）的 ACE wrapper facade）使用的。为了实施设计上的这种约束，并且为了防止 ACE_IPC_SAP 被直接实例化，我们将 ACE_IPC_SAP 的构造函数声明在这个类的 protected 部分，使其成为一个*抽象类（abstract class）*。通过使用这一规范，而不是定义纯虚方法，每一个子类实例（instance）就不需要一个虚拟指针（virtual pointer）。

[2] 在“网络标准”文献[Bla91]中，缩写词“SAP”代表“service access point （服务访问点）”。

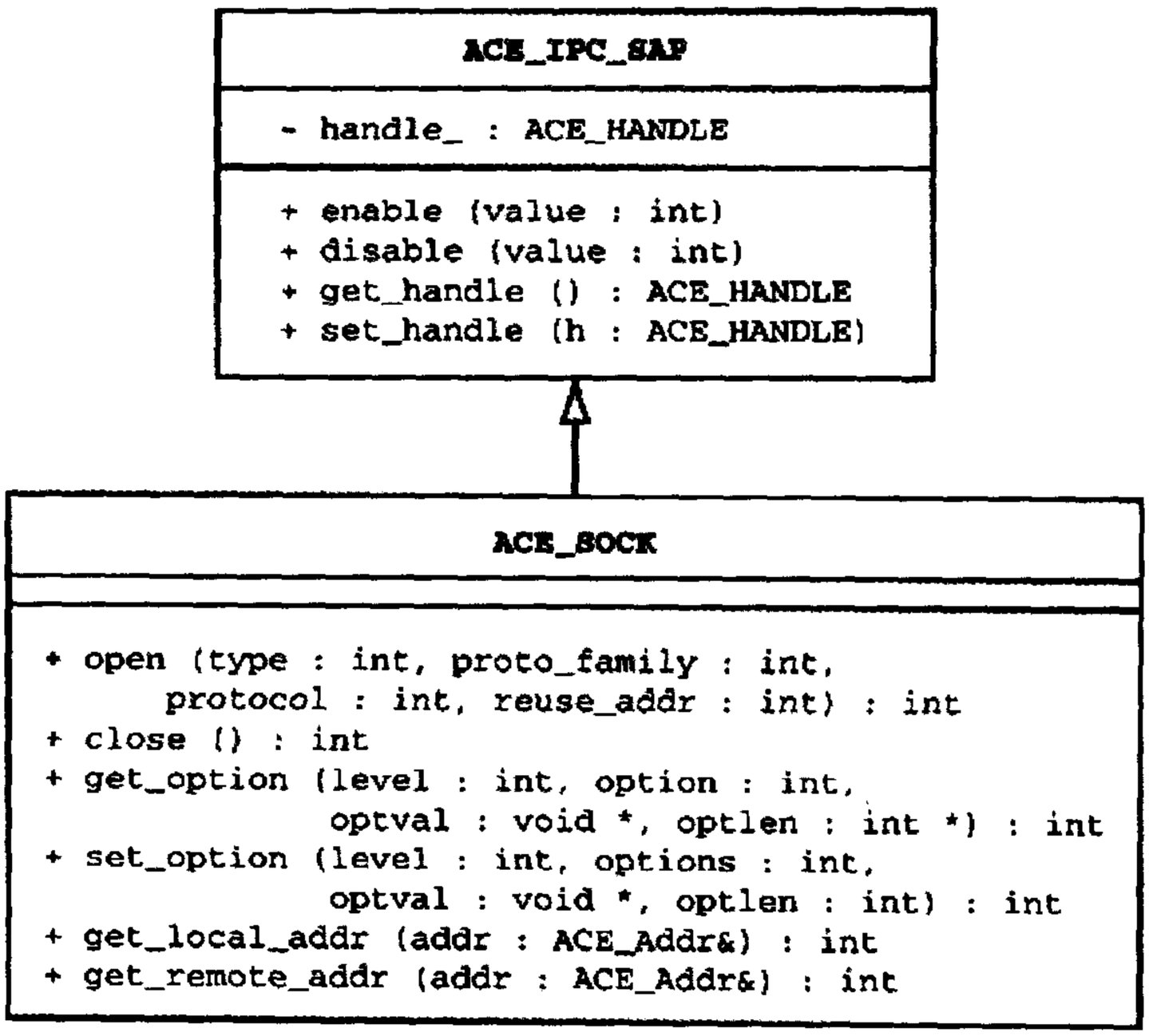


图 3.5：ACE_IPC_SAP 和 ACE_SOCK 类图

3.4 ACE_SOCK 类

设计意图

如第 2.3 节所述，在 Socket API 中，产生偶发复杂性的一个主要原因是：编译器没有能力在编译时检测到 socket“句柄被误用”的情况。为解决这一问题，第 3.3 节介绍的 ACE_IPC_SAP 类迈出了第一步。在本章余下部分，我们将讲述其他一些 ACE Socket wrapper facade，用以解决第 2.3 节演示的其他问题。我们沿着图 3.1 所示的继承结构向下移动，对每一层中有意义的行为进行探讨。我们首先从 ACE_SOCK 类开始。

功能

ACE_SOCK 类是 ACE Socket wrapper facade 继承结构的根（root）。除了输出“从 ACE_IPC_SAP 类派生而来”的方法（method）之外，ACE_SOCK 类还提供了其他许

多功能，这些功能对其他 ACE Socket wrapper facade（包括本章后面将要讲述的类）来说都是公用的。这些功能包括：

- 创建和销毁 socket 句柄；
- 获得“本地”和“远程”对等端（peer）的网络地址；
- 设置和读取 socket 选项，例如，socket 队列（queue）的长度、提供广播/多播通信功能、禁用 Nagle 算法（将在副栏 6 中讲述）。

ACE_SOCK 的接口以及它与 ACE_IPC_SAP 基类（ base class）的关系如图 3.5 所示。它的主要方法（method）见下表：

方　法	说　明
open() close()	创建和销毁一个 socket 通信端点。
set_local_addr() get_remote_addr()	分别返回本地和远程对等端的地址。
set_option() get_option()	设置和读取 socket 选项。

为防止意外误用，ACE_SOCK 被定义为抽象类（abstract class）；也就是，它的构造函数被声明在这个类（class）的 protected 访问控制代码段中。因而，和 ACE_IPC_SAP 一样，ACE_SOCK 的对象不能被直接实例化，因为它们只能被子类（例如，第 3.5~3.7 节讨论的面向连接 wrapper facade）访问。

因为 ACE_SOCK 类没有在析构函数中关闭 socket 句柄，所以它提供了一个 close() 方法。正如第 A.6.1 节所介绍的那样，这种设计是有目的的，它可以防止“将 ACE_SOCK_Stream 作为值（value）传递或者复制给别的对象”时发生错误。有经验的 ACE 开发者会使用较高层次的类来自动关闭底层 socket 句柄。例如，在第 4.4.2 节中，我们介绍了 Logging_Handler 类；当高层处理器（handler）对象关闭时，这个类会关闭它的 socket。[SH]中介绍的 ACE_Svc_Handler 类也会自动关闭底层 socket 句柄。

3.5 ACE_SOCK_Connector 类

设计意图

虽然 Socket API 在面向连接协议中扮演了三种不同角色，但它只支持以下两种 socket 模式：

（1）*数据模式（data-mode）socket*。扮演“通信”角色的对等应用程序使用这一模式，在“已连接的对等端（peer）”之间交换数据。

（2）*被动模式（passive-mode）socket*。扮演“被动连接”角色的对等应用程序将其用作工厂（factory），返回一个已连接的“数据模式”socket 的句柄。

但是，没有任何一种 socket 模式专门承担“主动连接”角色。相反，常规做法是，“数据模式”socket 会扮演这一角色。因此，在 connect()函数成功建立连接之前，应用程序不能在数据模式 socket 上调用 send()或 recv()。

Socket API 在“连接角色”和“socket 模式”上的不对称性不仅让人困惑，而且容易引发错误。例如，在某一“数据模式”socket 句柄被连接之前，应用程序有可能无意中对它调用 recv()或 send()。不幸的是，这种问题只有在运行时才能发现，因为 socket 句柄属于“弱类型”。因此，“数据模式”socket 的双重角色只有通过编程规范才能实施，而不是通过编译器的类型检查特性。基于以上原因，ACE 定义了 ACE_SOCK_Connector 类。这个类只展示了用于“主动建立连接”的方法，从而防止了对它的意外误用。

功能

ACE_SOCK_Connector 类是一个工厂（factory），用来主动建立一个新的通信端。它提供的功能如下：

- 发起一个到“对等接受者（peer acceptor）”的连接，并在连接建立后初始化一个 ACE_SOCK_Stream 对象。
- 连接可以通过“阻塞”、“非阻塞”或“定时”方式发起。
- 运用 C++ traits，以支持泛型编程技术，从而可以通过 C++“参数化类型”进行功能上的批量替换，如副栏 5 所述。

副栏 5：在 ACE Socket Wrapper Facade 中使用 Traits

为了简化 IPC 类和相关“地址”类的批量替换，ACE Socket wrapper facade 定义了 *traits（特征类）*。Traits 是一种 C++泛型编程 [Ale01]技术，用来定义和结合一组特征，以改变模板类（template class）的行为 [Jos99]。ACE Socket wrapper facade 通过 traits 定义的“类关系”如下：

• PEER_ADDR——这个 trait 定义的是和 ACE Socket wrapper façade 类关联的 ACE_INET_Addr“地址”类。

• PEER_STREAM——这个 trait 定义的是和 ACE_SOCK_Acceptor 以及 ACE_SOCK_Connector 工厂关联的 ACE_SOCK_Stream“数据传输”类。

在 ACE 中，这些 traits 被实现为 C++类型定义（type definitions），在本章的 UML 图中则以附注（notes）的形式出现。在第 A.5.3 节，我们将演示如何使用这些 traits，从而编写出简明的泛型函数和类。在[SH]中，我们探讨了 ACE traits 和 traits 类的其他用法。

ACE_SOCK_Connector 的接口如图 3.6 所示。

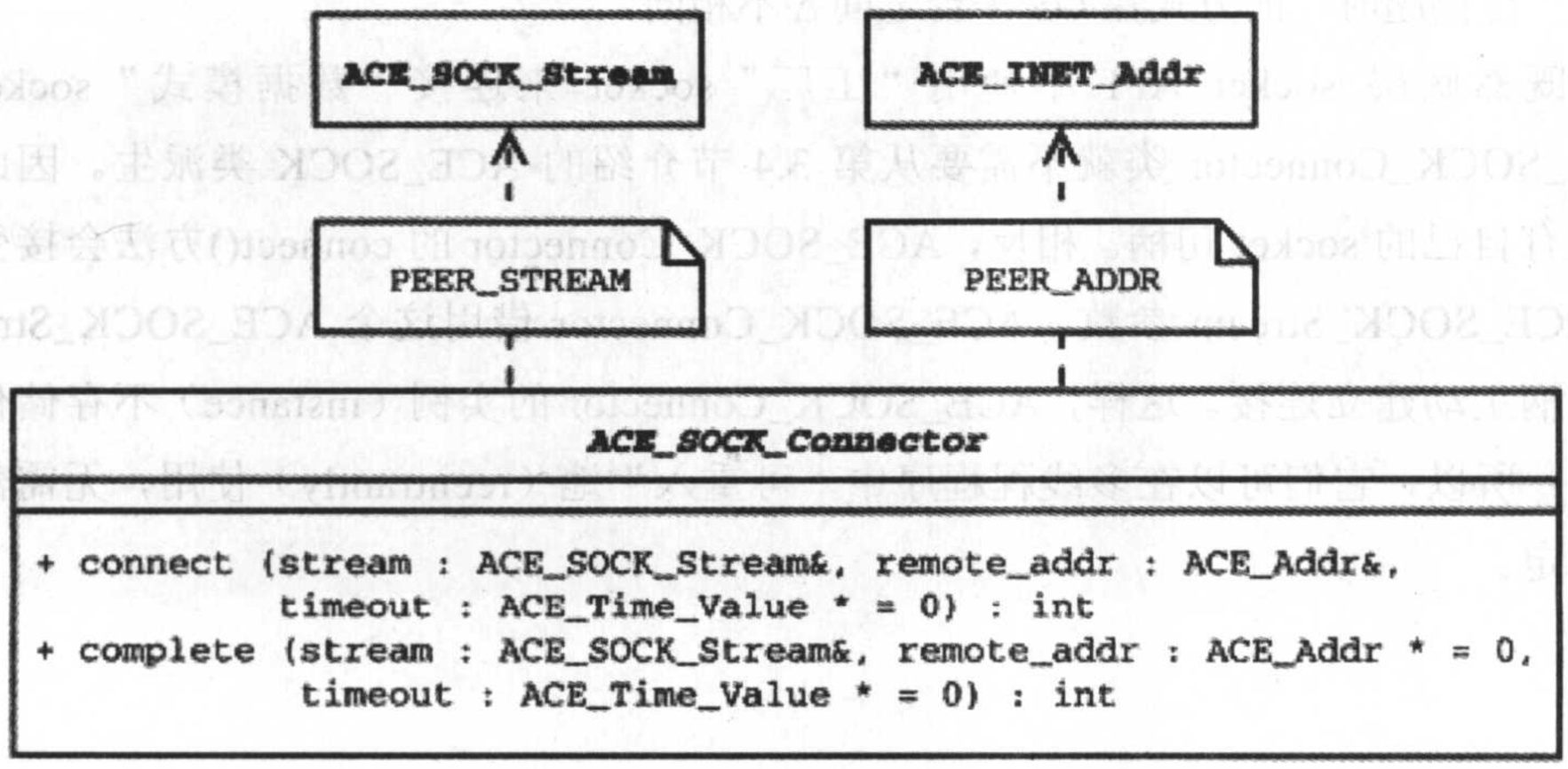


图 3.6：ACE_SOCK_Connector 类图

ACE_SOCK_Connector 的两个主要方法见下表：

方 法	说 明
connect()	通过“阻塞”、“非阻塞”或“定时”模式之一，主动连接某一网络地址上的 ACE_SOCK_Stream。
complete()	结束一个“非阻塞”连接并初始化一个 ACE_SOCK_Stream。

ACE_SOCK_Connector 支持“阻塞”、“非阻塞”和“定时”连接，缺省为“阻塞”连接。如果是在“高延迟”链路（link）上建立连接，或是使用单线程程序，或者要初始化很多“可以按任意次序连接”的对等端（peer）时，则“非阻塞”和“定时”连接很有用。可以将以下三类 ACE_Time_Value 值传给 connect()方法，以控制它的行为：

方 法	说 明
NULL ACE_Time_value 指针	表示 connect()应该无限期等待；即，connect()会阻塞，直至连接成功建立，或者 OS 认为无法连接到服务器主机。
非 NULL ACE_Time_value 指针，sec()和 usec()方法返回的值为 0	表示 connect()应当执行“非阻塞”连接；即，如果连接没有很快建立成功，它会返回–1，还会将 errno 设置为 EWOULDBLOCK。
非 NULL ACE_Time_value 指针，sec()或 usec()方法返回的值 >0	表示 connect()会等待一段时间，以建立连接。如果届时无法建立连接，它会返回–1，还会将 errno 设置为 ETIME。

ACE_SOCK_Connector 的“连接超时”功能在实践中特别有用，因为 Socket API 实现“连接超时”的方式在 OS 平台之间各不相同。

既然底层 socket API 不使用“工厂”socket 来连接“数据模式”socket，ACE_SOCK_Connector 类就不需要从第 3.4 节介绍的 ACE_SOCK 类派生。因此，它没有自己的 socket 句柄。相反，ACE_SOCK_Connector 的 connect()方法会接受一个 ACE_SOCK_Stream 参数，ACE_SOCK_Connector 借用这个 ACE_SOCK_Stream 的句柄主动建立连接。这样，ACE_SOCK_Connector 的实例（instance）不存储任何状态；所以，它们可以在多线程程序中“可重入”地（reentrantly）使用，无需额外的锁定。

示例

在副栏 2 中，我们已经告诉你如何去构建（build）ACE 库；这样一来，你就可以去实际体验本书提供的例子了。在本书的第一个例子中，我们将演示如何使用 ACE_SOCK_Connector，将客户程序连接到 Web 服务器上。

首先，我们得包含必要的 ACE Socket wrapper 头文件：

```
#include "ace/INET_Addr.h"
#include "ace/SOCK_Connector.h"
#include "ace/SOCK_Stream.h"
```

然后，我们来定义 main()函数，该函数的功能是同一个在端口 80 侦听的 Web 服务器建立连接。这是支持“超文本传输协议（HTTP，Hypertext Transport Protocol）”的 Web 服务器使用的标准端口号 [Ste96]。HTTP 是一种建立在 TCP 之上的简单协议，用户通过它从 Web 服务器下载内容。

```
int main (int argc, char *argv[])
{
  const char *pathname =
    argc > 1 ? argv[1] : "index.html";
  const char *server_hostname =
    argc > 2 ? argv[2] : "ace.ece.uci.edu";

  ACE_SOCK_Connector connector;
  ACE_SOCK_Stream peer;
  ACE_INET_Addr peer_addr;

  if (peer_addr.set (80, server_hostname) == -1)
    return 1;
  else if (connector.connect (peer, peer_addr) == -1)
    return 1;
  // ...
```

我们首先介绍 ACE_SOCK_Stream 这个“数据传输”类的功能；在第 3.6 节，我们将介绍这个例子的其余部分。

请注意，上面的 connector.connect()调用是同步的；也就是说，它会阻塞，直至连接成功，或者“连接请求”失败。如第 58 页上的表中所述，ACE Socket wrapper facade 能方便地完成“非阻塞”或“定时”连接。例如，在下面的代码中，我们演示如何修改我们的代码，从而“非阻塞”式地 connect()至 Web 服务器：

```
// Designate a nonblocking connect.
if (connector.connect (peer,
                       peer_addr,
                       &ACE_Time_Value::zero) == -1) {
  if (errno == EWOULDBLOCK) {
    // Do some other work ...

    // Now, try to complete the connection establishment,
    // but don't block if it isn't complete yet.
    if (connector.complete (peer,
                            0,
                            &ACE_Time_Value::zero) == -1)
      // ...
```

同样，“定时”connect()可以像下面这样执行：

```
ACE_Time_Value timeout (10); // Set time-out to 10 seconds

if (connector.connect (peer, peer_addr, &timeout) == -1) {
  if (errno == ETIME)
    // Time-out, do something else...
}
```

3.6 ACE_SOCK_IO 类和 ACE_SOCK_Stream 类

设计意图

在第 2.3 节确定的偶发复杂性中，有一个问题是：编译器无法在编译时发现“误用 socket”的情况。如第 3.1 节所述，“连接管理”涉及三个角色：*主动连接角色（active connection role）*、*被动连接角色（passive connection role）*和*通信角色（communication role）*。但 Socket API 只定义了两种 socket 模式：*数据模式（data mode）*和*被动模式（passive mode）*。所以，开发者可能会以某种方式误用了 socket，但编译器无法在编译时发现这种情况。为了消除这方面的复杂性，ACE_SOCK_Connector 类迈出了第一步，ACE_SOCK_Stream 类则迈出了另一步。ACE_SOCK_Stream 定义了一个“数据模式”的“只传输（transfer-only）”对象。ACE_SOCK_Stream 对象不能用作其他任何角色；若不有意破坏它的接口，它就只能用作“数据传输”角色。

功能

ACE_SOCK_Stream 类封装了“数据模式”socket 支持的数据传输机制。它提供的功能如下：

- 支持数据发送和接收。发送和接收的字节数最多 n 个或刚好 n 个。
- 支持“分散读取（scatter-read）”操作。这些操作涉及多个“调用者提供（caller-supplied）”的缓冲区，而不是单个连续的缓冲区。
- 支持“集中写入（gather-write）”操作。在单个操作中传输多个非连续数据缓冲区中的数据。
- 支持“阻塞”、“非阻塞”、“定时”I/O 操作。
- 支持泛型编程（generic programming）技术 [Ale01]，可以借助“参数化类型”实现功能上的批量替换，如副栏 5 所述。

ACE_SOCK_Stream 实例（instance）在 ACE_SOCK_Acceptor 或 ACE_SOCK_Connector“工厂（factory）”中初始化。ACE_SOCK_Stream 和 ACE_SOCK_IO 类的接口如图 3.7 所示。ACE_SOCK_Stream 派生自 ACE_SOCK_IO；ACE_SOCK_IO 本身派生自 ACE_SOCK，并且定义了一些基本的数据传输方法，供 ACE UDP wrapper facade 复用（reuse）。

ACE_SOCK_Stream 输出的主要方法见下表：

方　法	说　明
send() recv()	发送、接收数据缓冲区数据。读、写的字节数可能比请求的字节数少，因为它们要受“OS 中的缓冲机制”和“传输协议中的流量控制”影响。
send_n() recv_n()	发送、接收恰好 n 个字节的缓冲区数据，简化“少量写入”和“少量读取”时的应用程序处理。
recvv_n()	使用 OS 的“分散读取”系统函数，高效、完整地接收多个缓冲区数据。
sendv_n()	使用 OS 的“集中写入”系统函数，高效、完整地发送多个缓冲区数据。

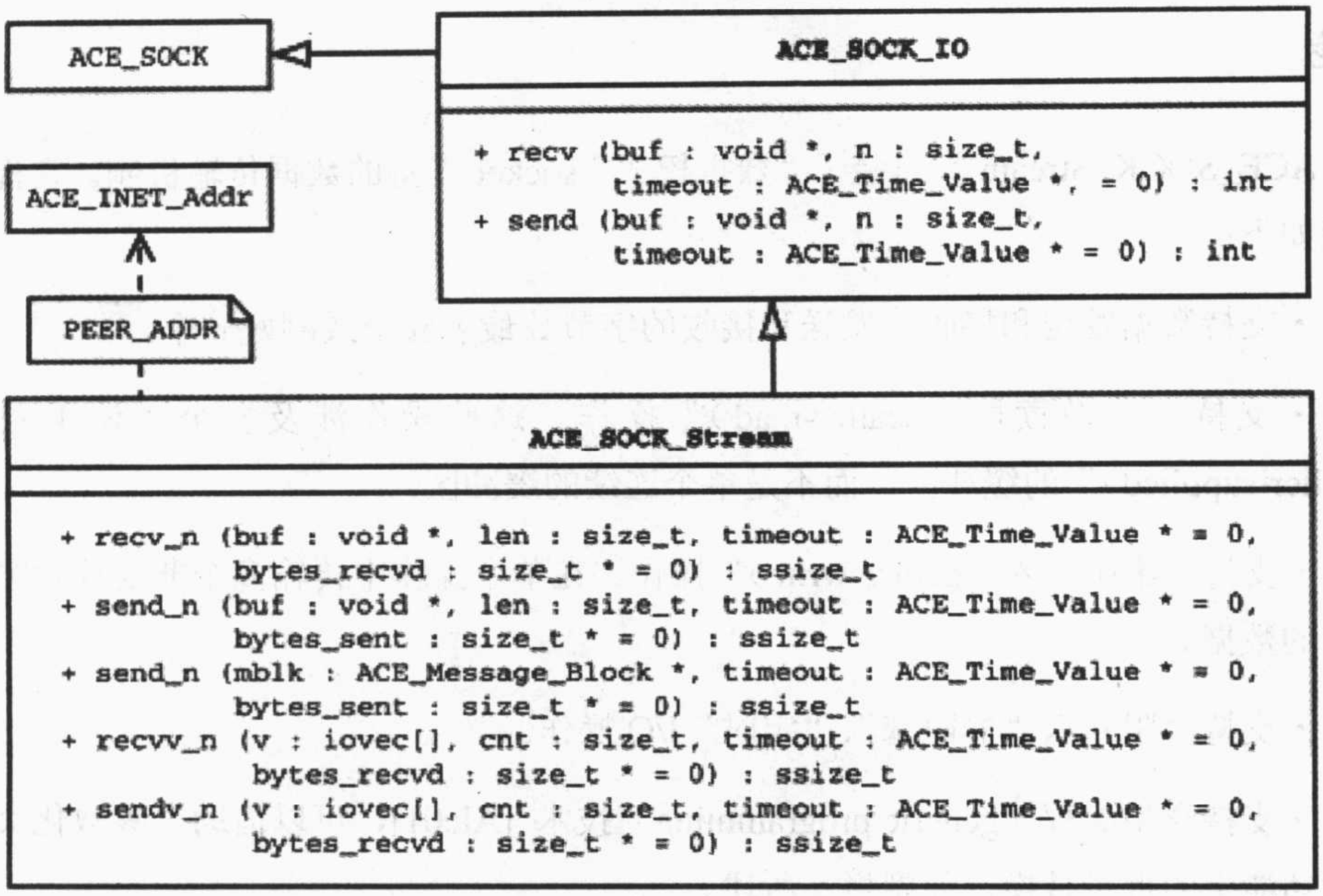


图 3.7：ACE_SOCK_Stream 和 ACE_SOCK_IO 类图

ACE_SOCK_Stream 类支持“阻塞”、“定时”、“非阻塞” I/O；如果使用“阻塞”式 I/O，调用者线程会无限期等待，直至 I/O 操作结束。如果在通信中，对方可以无限期挂起（hang）或阻塞，“定时” I/O 将很有用处。我们可以将两类 ACE_Time_Value 值传给 ACE_SOCK_Stream I/O 方法，以控制它们的“超时”行为：

方　法	说　明
NULL ACE_Time_Value 指针	表示 I/O 方法会阻塞，直至数据传输结束，或有错误发生。
非 NULL ACE_Time_Value 指针	表示 I/O 方法会等待一段时间，以传输数据。如果在数据发送或接收完成之前超时，它会返回–1，并将 errno 设置为 ETIME。

在数据没有被立即发送或接收的情况下，如果应用程序不能忍受此时发生阻塞，那么，使用“非阻塞” I/O 将很有帮助。“阻塞”和“非阻塞” I/O 可以通过 enable()和 disable()方法来调节，这两个方法继承自 ACE_IPC_SAP：

```
peer.enable (ACE_NONBLOCK); // Enable nonblocking I/O.
peer.disable (ACE_NONBLOCK); // Disable nonblocking I/O.
```

如果在处于“非阻塞”模式的 ACE_SOCK_Stream 实例上调用 I/O 方法，而且调用会阻塞，那么，这个方法将返回–1，并将 errno 设为 EWOULDBLOCK。

示例

探讨了 ACE_SOCK_Stream 的功能之后，对于第 3.5 节的 Web 客户示例，现在可以给出“数据传输”部分的代码了。这段代码先向一个 URL 路径名发送 HTTP GET 请求，然后打印出从 Web 服务器下载到的文件内容。

```
// ...Connection code from example in Section 3.5 omitted...
char buf[BUFSIZ];
iovec iov[3];
iov[0].iov_base = "GET ";
iov[0].iov_len = 4; // Length of "GET ".
iov[1].iov_base = (char*)pathname;
iov[1].iov_len = strlen (pathname);
iov[2].iov_base = " HTTP/1.0\r\n\r\n";
iov[2].iov_len = 13; // Length of " HTTP/1.0\r\n\r\n";

if (peer.sendv_n (iov, 3) == -1)
  return 1;

for (ssize_t n; (n = peer.recv (buf, sizeof buf)) > 0; )
  ACE::write_n (ACE_STDOUT, buf, n);

return peer.close () == -1 ? 1 : 0;
}
```

代码中用到了一个类型为 iovec 结构的数组，使用这个数组，并通过 ACE_SOCK_Stream::sendv_n()这个“集中写入”方法，可高效地向 Web 服务器发送 HTTP GET 请求。这避免了副栏 6 介绍的 Nagle 算法 [Ste93]存在的性能方面的问题。在 UNIX/POSIX 平台上，这个方法是通过 writev()实现的；在 WinSock2 平台上，则是通过 WSASend()实现的。

如果 main()函数中的 I/O 方法遇上了 TCP 流量控制（flow control），或者 Web 服务器发生故障，就会发生阻塞。为防止客户无限期挂起（hang），可以为这些方法添加超时控制。例如，在下面的代码中，如果服务器在 10 秒钟内没有接收到数据，就会返回–1，并将 errno 设置为 ETIME：

```
// Wait no more than 10 seconds to send or receive data.
ACE_Time_Value timeout (10);

peer.sendv_n (iov, 3, &timeout);

while (peer.recv (buf, sizeof buf, &timeout) > 0)
  // ... process the contents of the downloaded file.
```

副栏 6：使用（或避开）Nagle 算法

缺省情况下，大多数 TCP/IP 实现（implementation）都会使用 Nagle 算法（Ste93）；这一算法会在发送端的 TCP/IP 栈（stack）中缓存“少量”、“依次发送”的数据报。这个算法虽然缓解了网络拥塞，但如果不清楚它“何时”、以及“如何”起作用，反而会增加延迟与降低吞吐量。如果将几个小的缓冲区放在“连续”、“单向”操作中发送，这些问题就会产生。例如，下面的代码会触发 Nagle 算法：

```
peer.send_n ("GET ", 4);
peer.send_n (pathname, strlen (pathname));
peer.send_n (" HTTP/1.0\r\n\r\n", 13);
```

应用程序开发者可以禁用 Nagle 算法，方法是调用 peer.set_option()，并使用 TCP_NODELAY 标志，强制 TCP 尽快将报文发送出去。第 63 页的 main()函数演示了一个更为高效的方案，它使用了 sendv_n()，在一个系统函数中将所有数据缓冲区都发送出去。这个方法接受一个类型为 iovec 结构的数组，该结构的定义如下：

```
struct iovec {
  // Pointer to a buffer.
  char *iov_base;

  // Length of buffer pointed to by <iov_base>.
  int iov_len;
};
```

一些 OS 平台本身就定义了 iovec；但在另外一些平台上，它是由 ACE 定义的。所有情况下，结构中成员的名称是相同的，但顺序不一定相同，所以要显式地设置它们，不要通过 struct 初始化。

3.7 ACE_SOCK_Acceptor 类

设计意图

ACE_SOCK_Connector 类和 ACE_SOCK_Stream 类解决了“通信角色”和“Socket API 函数”不匹配所导致的复杂性问题。虽然 Socket API 单独定义了一组函数来填补

“被动连接建立”角色，但另一组复杂性还是存在。Socket API 中的 C 函数属于“弱类型（weakly typed）”，很容易被误用。

例如，accept()函数可以在一个“数据模式”的 socket 句柄上调用，但这个句柄本来是准备通过 recv()和 send() I/O 操作来传输数据的。类似地，I/O 操作可以在一个“被动模式”socket 句柄工厂上调用，但这个句柄工厂原本只是用来接受连接的。不幸的是，这些错误只有在运行时才能被发现。ACE 通过“强类型（strongly typed）”的 ACE_SOCK_Acceptor 类来消除这些复杂性。和直接的 Socket API 调用不一样，在编译时，编译器可以很容易地发现对 ACE_SOCK_Acceptor 的误用。

功能

ACE_SOCK_Acceptor 类是一个“工厂（factory）”[GHJV95]，用来“被动”地建立一个新的通信端点。它提供以下功能：

- 接受来自对等连接端（peer connector）的连接，并在连接建立后初始化 ACE_SOCK_Stream 对象。
- 连接可以通过“阻塞”、“非阻塞”或“定时”的方式接受。
- 运用 C++ traits 支持泛型编程技术；从而，可以通过 C++“参数化类型（parameterized types）”来进行功能（functionalities）上的批量替换，如副栏 5 所述。

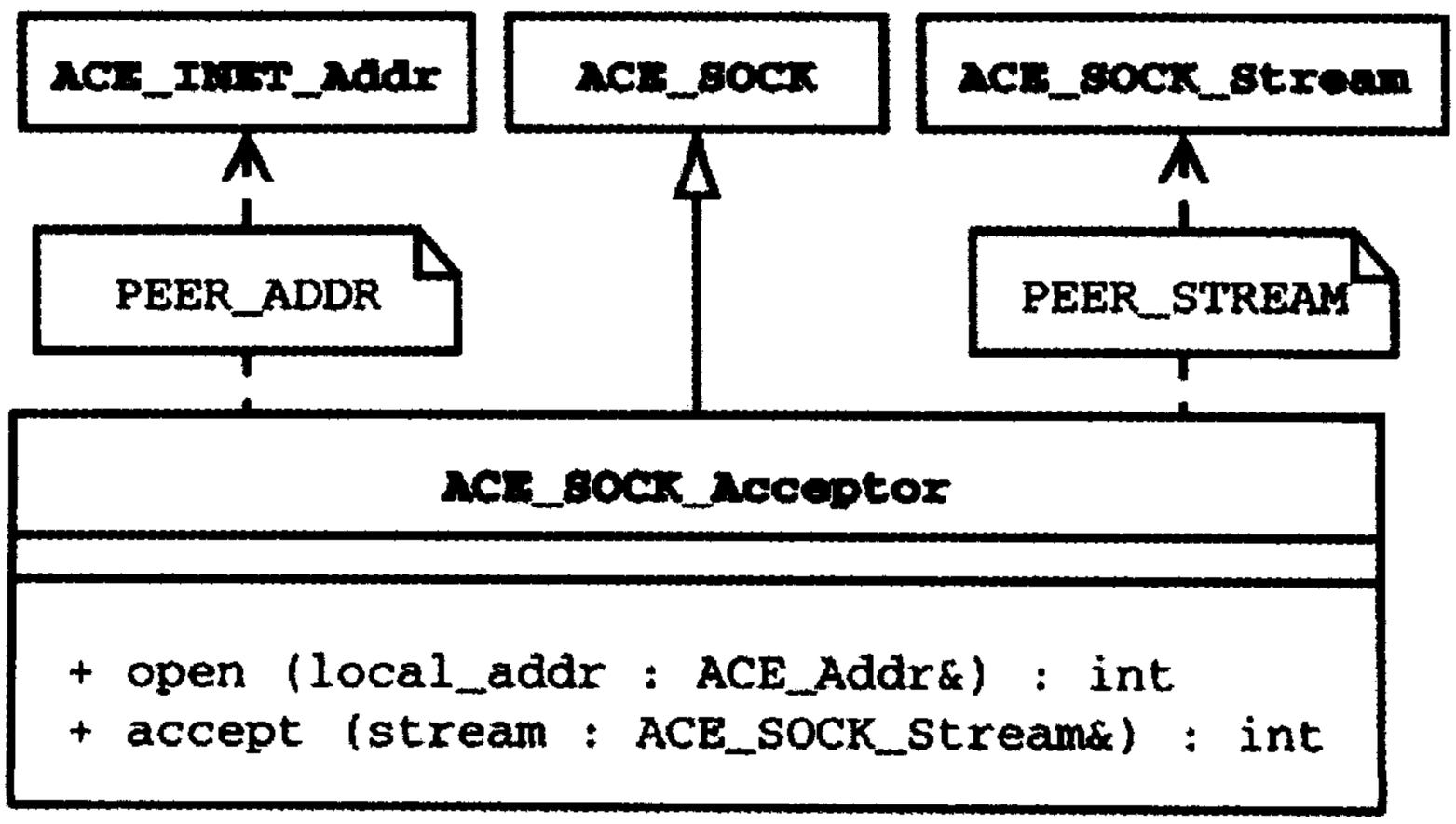


图 3.8：ACE_SOCK_Acceptor 类图

ACE_SOCK_Acceptor 类的接口如图 3.8 所示，它的两个主要方法见下表：

方 法	说 明
open()	初始化“被动模式”、“工厂”socket，被动侦听一个指定的 ACE_INET_Addr 地址。
accept()	通过一个新接受的客户连接（client connection）初始化 ACE_SOCK_Stream 参数。

在 ACE_SOCK_Acceptor 中，open()和 accept()方法使用的 socket 句柄继承自 ACE_IPC_SAP。这个设计很好地利用了 C++类型系统，使得应用程序开发者可以遵循以下原则，避免“误用（misuse）”情况发生：

• ACE_SOCK_Acceptor 的 open()方法总是以正确的顺序调用底层 socket()、bind()和 listen()函数。

• 这些函数所调用的 socket 句柄，只会是那些已经被“被动模式”socket 句柄工厂初始化了的 socket 句柄。

关于这些（以及其他）ACE wrapper facade 设计选择（design choices）所体现的设计原则，详细介绍参见附录 A。

示例

为了演示 ACE_SOCK_Acceptor，我们给出一个循环式（iterative）Web 服务器的例子，演示建立“被动连接”和下载内容的情况。这个服务器是像这样处理客户请求的：

（1）在 80 端口接受一个客户连接；

（2）读取 GET 请求中的路径名，然后下载请求的文件；

（3）关闭连接；

（4）回到第 1 步。

为了高效地读取文件，我们使用了 ACE_Mem_Map。ACE_Mem_Map 的介绍详见副栏 7。为了利用 ACE_Mem_Map 的能力，除了所需的其他 ACE Socket wrapper facade 头文件外，还包含了 ace/Mem_Map.h；这样，就可以开始编写我们的 Web 服务器了。

```
#include "ace/Auto_Ptr.h"
#include "ace/INET_Addr.h"
#include "ace/SOCK_Acceptor.h"
```

```
#include "ace/SOCK_Stream.h"
#include "ace/Mem_Map.h"

// Return a dynamically allocated path name buffer.
extern char *get_url_pathname (ACE_SOCK_Stream *);
```

我们的 Web 服务器支持 HTTP 1.0 版本；所以，每一个“客户请求”都有一个单独的 TCP 连接。在下面的 Web 服务器代码中，首先初始化 ACE_SOCK_Acceptor 工厂，让它在端口 80 上侦听连接；然后启动一个事件循环，不断地（iteratively）接受新的客户连接，将文件映射到内存；最后将其下载到客户端，并在文件传输结束时关闭连接。

```
int main ()
{
  ACE_INET_Addr server_addr;
  ACE_SOCK_Acceptor acceptor;
  ACE_SOCK_Stream peer;

  if (server_addr.set (80) == -1) return 1;
  if (acceptor.open (server_addr) == -1) return 1;

  for (;;) {
    if (acceptor.accept (peer) == -1) return 1;
    peer.disable (ACE_NONBLOCK); // Ensure blocking <send_n>.

    ACE_Auto_Array_Ptr <char> pathname (get_url_pathname (&peer));
    ACE_Mem_Map mapped_file (pathname.get ());

    if (peer.send_n (mapped_file.addr (),
                     mapped_file.size ()) == -1) return 1;
    peer.close ();
  }

  return acceptor.close () == -1 ? 1 : 0;
}
```

在大量客户同时发送请求的情况下，这个循环式 Web 服务器的主要缺陷将会暴露。这种情况下，在 Web 服务器处理当前“连接请求”的同时，OS 的 Socket 实现（implementation）会将少量（例如，5~10 个）“连接请求”排成队列。如果 Web 服务器实际投入运行，OS 很快就会不堪重负。出现这种情况时，如果 Web 服务器的 OS 拒绝接受客户的连接请求，客户端就会出错。在第 8 章和第 9 章，我们将介绍一些模式和 ACE wrapper facade，用以解除这一缺陷。

副栏 7：ACE_Mem_Map 类

Win32 和 POSOX 平台都提供了“内存映射（memory-mapped）” 文件机制；ACE_Mem_Map wrapper facade 封装了这一机制。这些调用（call）使用了 OS 虚拟内存机制，将文件映射到一个进程的地址空间中。被映射的文件内容可以通过指针直接访问。较之通过读/写“系统 I/O 函数”来访问数据块，“内存映射文件”更方便、更高效。此外，“内存映射文件”可以被同一台机器上的多个进程共享，如第 1.3 节所述。

Win32 和 POSIX 本身的“内存映射文件”API 不具可移植性，而且很复杂。例如，开发者必须手工完成很多簿记（bookkeeping）工作，如必须显式地（explicitly）打开一个文件，确定文件的长度，然后完成多个映射。相反，ACE_Mem_Map wrapper facade 提供了一个接口，这个接口使用了缺省值、以及“具有多个类型签名（type signature）”（例如，“从一个打开的文件句柄映射”，或“从一个文件名映射”）的构造函数，从而简化了“内存映射文件”的常见用法。

3.8 小结

本章讲述的是，ACE 如何在网络应用程序中使用 C++特性和 Wrapper Facade 模式，从而更容易地编写正确、具有可移植性的“面向连接”TCP/IP 机制。我们集中讨论了 ACE Socket wrapper façade，它简化了：

- 网络地址、I/O 句柄、基本 socket 操作；
- TCP 连接的建立和数据传输操作。

第 4 章演示如何运用这些 Socket wrapper facade，开发网络日志服务程序的第一个版本。

ACE Socket wrapper facade 解决了 Socket API 存在的问题，这些问题在第 2.3 节曾作过介绍，即

- **容易出错的 API**——ACE Socket wrapper facade 可以保证：传递给 Socket API 的所有参数（argument）都是类型安全（type-safe）的。此外，地址查找（address lookup）时必须当心字节顺序（byte-ordering）问题，而这些问题也都由 ACE 透明地处理掉了。

• **过于复杂的 API**——ACE Socket wrapper facade 将 Socket API 群集（cluster）到了单独的、分别用于“主动连接”、“被动连接”和“传输数据”的类中，使得网络应用程序开发者面临的难题大大减少。这样一来，在编写网络应用程序时，开发者就不必了解所有的 ACE Socket wrapper facade，而只是使用其中的一部分。

• **不可移植、不统一的 API**——在 ACE 支持的所有平台上，客户程序和 Web 服务器源代码都能正确、高效地编译和运行。

采用 ACE Socket wrapper facade 的“面向对象”设计，应用程序的 IPC 机制可以很容易改变，并且无需更改 API 参数，也无需手工修改地址结构和函数。如图 3.2 所示，每一个通信服务类型（流、连接的数据报或数据报）中的类都提供了一套公共方法；通过*泛型编程*（generic programming）技术[Ale01]，这些方法很容易互换。如副栏 5 所述，有了泛型编程和 C++ trait *手法（idiom）*，我们可以通过参数化类型（parameterized type）进行功能上的批量替换。这些技术也适用于其他 ACE IPC wrapper façade 类，请参见 ACE 网站 http://ace.ece.uci.edu 上的介绍。

第 4 章

网络日志服务程序的实现

本章提要

本章提供了网络日志服务程序的第一个实现：通过第 3 章介绍的 ACE Socket wrapper facade，并结合使用其他处理“消息缓冲（buffering）和整编/解编（marshaling/demarshaling）”的 ACE 类，以一种具有可移植性、正确、高效的方式发送、接收和处理日志记录。在本书余下部分，我们会不断改进这个例子，并就开发“面向对象”网络应用程序时需要解决的常见问题和难点进行论述。

4.1 概述

大多数网络应用程序都需要执行以下行为：

- 主动/被动地建立连接；
- 和对等应用程序进行数据交换；
- 管理数据缓冲区；
- 整编（marshal）/解编（demarshal）“类型化（typed）”数据，使得数据在各类异种（heterogeneous）系统中都能得以正确处理。

本章将演示，如何借助 ACE 的消息“缓冲”及“整（解）编”类，来壮大第 3 章的 ACE Socket wrapper façade 的功能，从而使我们的网络日志服务程序的第一个实现得以简化。这个服务程序的“参与者”如图 0.6 所示。

4.2 ACE_Message_Block 类

设计意图

网络应用程序大都需要一种方法来高效地处理消息 [SS93]。标准的“消息管理”操作包括：

- 收到来自“网络”或来自“同一主机上其他进程”的消息后，将这些消息保存到缓冲区中；
- 当消息经过用户级协议栈时，添加和（或）删除消息的头和尾；
- 将消息分段（fragmenting）并重新组装（resembling），使其适合网络的最大传输单元（MTU，maximum transmission unit）；
- 将消息保存到缓冲区中，以进行传输，或再次传输；
- 对“未按顺序（out-of-sequence）”接收到的消息作好记录。

为了提高效率，在这些操作中，动态内存管理的开销必须降至最低，不必要的数据复制也必须避免。基于此，ACE 提供了 ACE_Message_Block 类。

功能

有了 ACE_Message_Block 类，我们可以高效地管理具有“固定”和“可变”长度的消息。ACE_Message_Block 实现了 Composite 模式 [GHJV95]，并提供以下功能：

- 每一个 ACE_Message_Block 都包含一个指针，指向“带引用计数（referenced-counted）”的 ACE_Data_Block，而 ACE_Data_Block 则指向“和消息关联”的实际数据。采用这一设计，我们可以灵活、高效地共享数据，并能降低内存复制带来的额外开销。
- 允许多条消息连接在一起，形成一个单链表，从而支持复合消息（composite message）。这种复合消息可用于多态链表（polymorphic lists），也可用于“需要高效地添加/删除消息头/尾”的层次化协议栈。
- 允许将多条消息连接起来，形成一个双链表，构成 ACE_Message_Queue 类的基础。关于 ACE_Message_Queue，本书第 228 页及 [SH]中作了介绍。

• 将“同步”和“内存管理”特性作为“无需修改底层 ACE_Message_Block 实现，应用程序就可以改变”的*特征（aspect）*[Kic97, CE00]来对待。

ACE_Message_Block 类的接口如图 4.1 所示。这个设计的模型大致基于 System V STREAMS [Rag93]的消息缓冲机制；该类支持两种消息：

• **简单消息（Simple message）**只包含一个 ACE_Message_Block，如图 4.2(1)所示。

• **复合消息（Composite message）**包含多个 ACE_Message_Blocks，它们依据 Composite 模式 [GHJV95]连接在一起，形成了一种结构（structure），用来构造“递归聚合体（recursive aggregates）”，如图 4.2(2)所示。复合消息中通常包含一条控制消息，用来保存簿记信息，如目标地址；接下去是一条或多条数据消息，包含的是消息的实际内容。

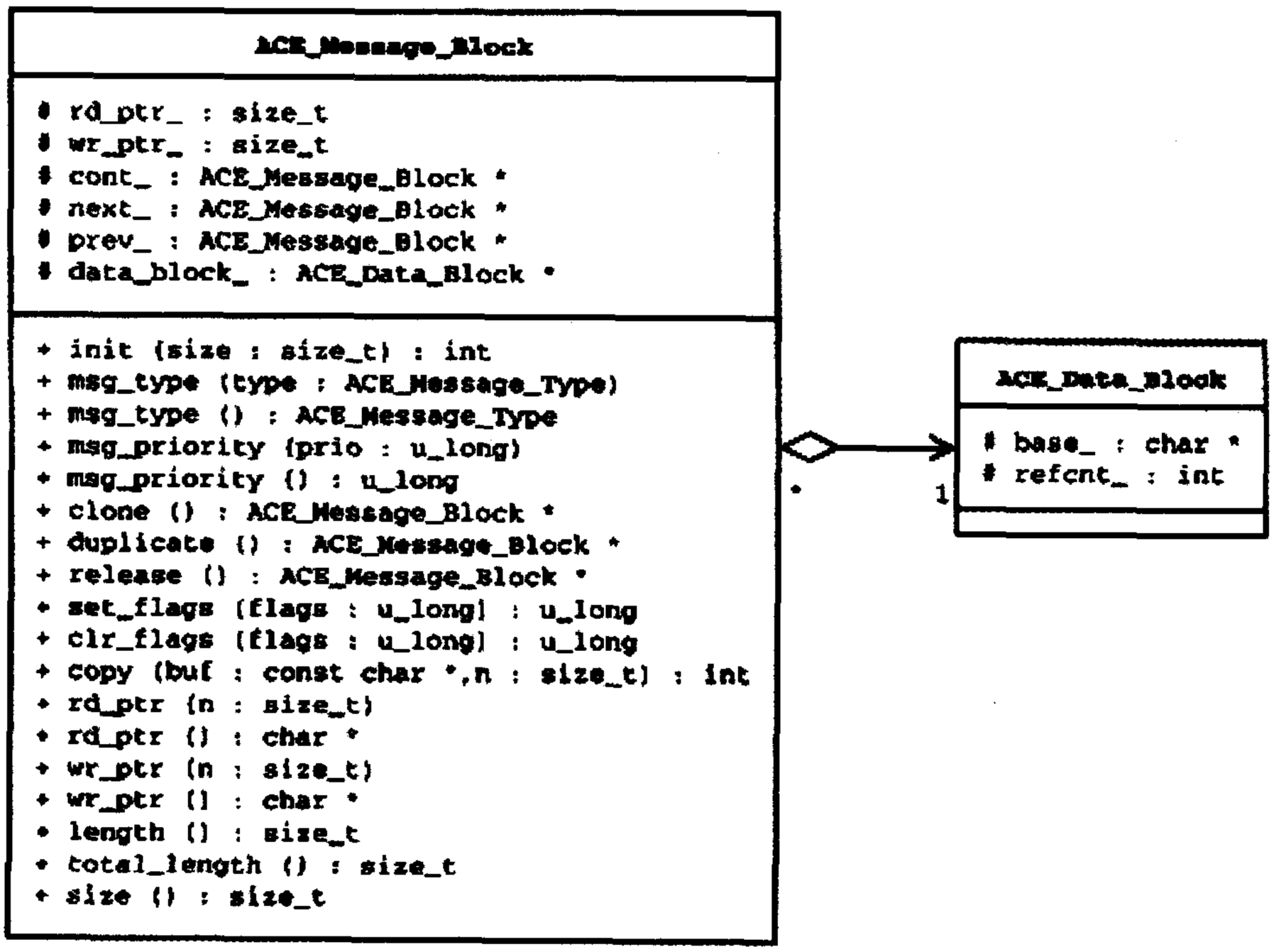


图 4.1：ACE_Message_Block 类图

ACE_Message_Block 的主要方法如下表所示：

方 法	说 明
ACE_Message_Block() init()	对消息进行初始化。
msg_type()	设置和获取消息的类型。
msg_priority()	设置和获取消息的优先级。
clone()	返回整条消息的完整“深复制（deep copy）”。
duplicate()	返回消息的“浅（shallow）”复制，将其引用计数增 1。
release()	将引用计数减 1，如果计数值降至 0，则释放消息的资源。
set_flags()	将指定的数据位（bit）同一组已有的标志（flag）执行“按位与（bitwise-OR）”操作，用以指定消息的语义（例如，消息释放时是否删除缓冲区，等等）。
clr_flags()	清除指定的标志位。
copy()	从缓冲区中复制 n 个字节到消息。
rd_ptr()	设置和获取“读”指针。
wr_ptr()	设置和获取“写”指针。
cont()	设置和获取消息中的“连接（continuation）”字段，这个字段用于将复合消息连接在一起。
next() prev()	设置和获取“指向 ACE_Message_Queue 中的双向消息链表”的指针。
length()	设置和获取消息的当前长度，即 wr_ptr() – rd_ptr()。
total_length()	获取消息的长度，包括所有被连接的消息块。
size()	设置和获取消息的总容量，包括分配在[rd_ptr(), wr_ptr()]范围之前及之后的存储容量。

每一个 ACE_Message_Block 都包含一个指针，指向“带引用计数”的 ACE_Data_Block，后者则指向数据的有效空间（payload），如图 4.2(1)所示。请注意，rd_ptr()和 wr_ptr()分别指向的是数据存储空间中活动部分的头部和尾部。图 4.2(2)演示的是，ACE_Message_Block::duplicate()方法如何用来生成一条消息的“浅复制”。这样做可以将内存分配和复制的开销降至最低，从而灵活、高效地共享同一数据。因为消息块本身会被复制，所以各个 rd_ptr()和 wr_ptr 可以指向被共享的同一有效空间（payload）的不同部分。

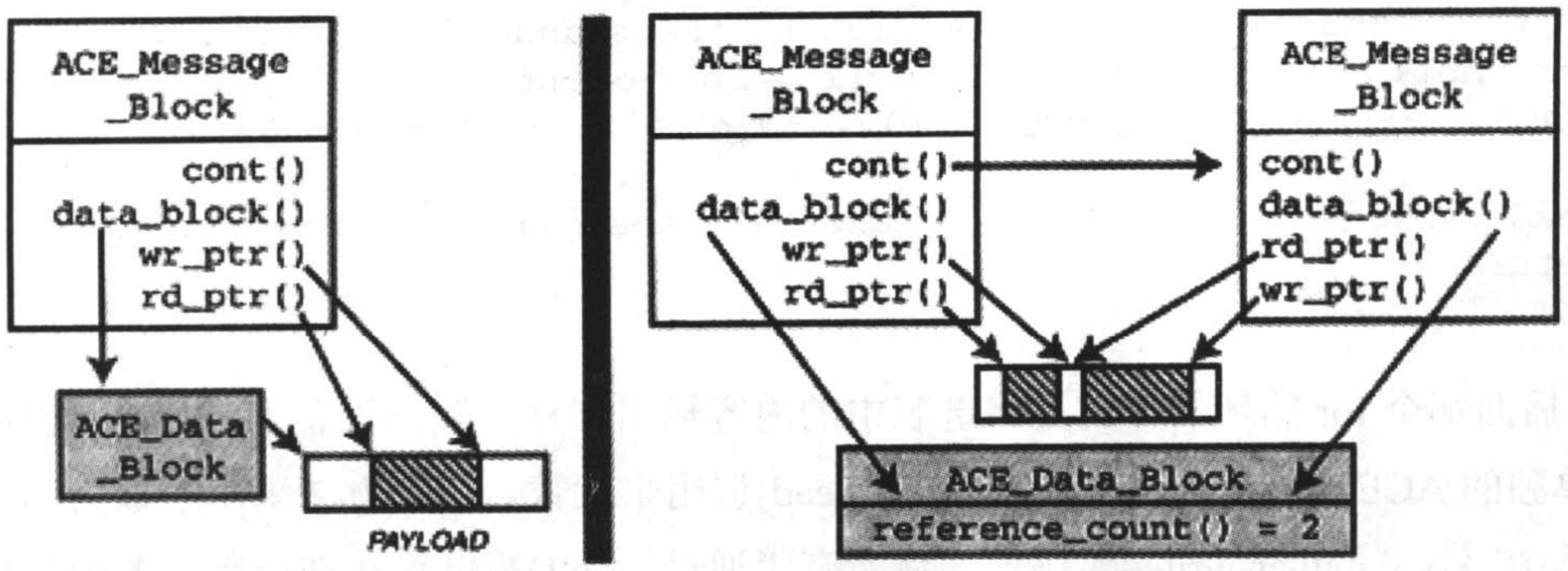


图 4.2：两种 ACE_Message_Block

示例

下面的程序将标准输入设备（standard input）中的所有数据读到一个单链表中；单链表中是动态分配的 ACE_Message_Blocks。首先，它们被“连接指针（continuation pointer）”串接在一起；然后，这个程序会将所有已连接的消息块中的内容输出到标准输出设备（standard output）中，并释放动态分配的内存。

```
#include "ace/OS.h"
#include "ace/Message_Block.h"

int main (int argc, char *argv[])
{
  ACE_Message_Block *head = new ACE_Message_Block (BUFSIZ);
  ACE_Message_Block *mblk = head;

  for (;;) {
    ssize_t nbytes = ACE::read_n (ACE_STDIN,
                                  mblk->wr_ptr (),
                                  mblk->size ());
    if (nbytes <= 0)
      break; // Break out at EOF or error.

    // Advance the write pointer to the end of the buffer.
    mblk->wr_ptr (nbytes);

    // Allocate message block and chain it at the end of list.
    mblk->cont (new ACE_Message_Block (BUFSIZ));
    mblk = mblk->cont ();
  }
```

```
  // Print the contents of the list to the standard output.
  for (mblk = head; mblk != 0; mblk = mblk->cont ())
    ACE::write_n (ACE_STDOUT, mblk->rd_ptr (), mblk->length ());

  head->release (); // This releases all the memory in the chain.
  return 0;
}
```

后面那个 for 循环的作用是将链表中的内容输出到标准输出设备上，它也可以用一个单独的 ACE::write_n(ACE_STDOUT, head)调用来代替。这个方法采用了极为高效的“集中写入（gather-write）”操作，将所有“通过 cont()指针连接在一起”的消息块输出到标准输出设备。第 4.4.2 节中介绍的 Logging_Handler::write_log_record()方法就使用了一种类似的优化方式。

之所以使用 ACE::read_n()和 ACE::write_n()方法，而不使用 C++的“输入/输出流”机制，是因为 ACE 所支持的 OS 平台并不全都充分地支持“输入/输出流”。在正确支持 cin 和 cout 的平台上，我们的确可以用它们来代替。但是，由于“内部缓冲”的原因，它们会引发额外的数据复制，而且不能利用前文所说的“集中写入”优化。

4.3 ACE_InputCDR 类和 ACE_OutputCDR 类

设计意图

发送/接收消息的网络应用程序经常需要提供以下功能：

- *线性化（linearization）*。负责“丰富类型（richly-typed）数据”（例如数组、链表、图）与“原始内存缓冲区”之间的转换。

- *整编/解编（marshaling/demarsheling）*。在某些环境下，不同的编译器具有不同的“对齐（alignment）”要求，硬件指令也有不同的“字节次序（byte-ordering）”规则。“整编/解编”的目标是：在这些环境下，程序都能够正确地工作。

线性化、整编/解编难以正确、高效地通过手工完成；所以，这些操作最好被封装在可复用（reusable）的类中。因此，ACE 提供了 ACE_OutputCDR 和 ACE_InputCDR。

功能

ACE_OutputCDR 和 ACE_InputCDR 类提供了高度优化、可移植、便利的方法，并通过标准的 CORBA 公共数据表示（CDR，Common Data Representation）格式[Obj01]，对数据进行整编和解编。ACE_OutputCDR 根据数据结构创建 CDR 缓冲区（整编），ACE_InputCDR 从 CDR 缓冲区提取数据（解编）。

ACE_OutputCDR 类和 ACE_InputCDR 类支持以下特性。

• 为以下类型提供整（解）编操作：

——**原始类型**，例如，布尔类型，16、32 和 64 位整数，8 位字节，单精度及双精度浮点数，字符，字符串；

——**由原始类型组成的数组**。

•“插入”（<<）和“提取”（>>）运算符可用于对原始类型执行整编和解编操作，使用的语法和 C++输入/输出流一样。

• 在内部使用 ACE_Message_Block 链，以避免昂贵的内存复制操作。

• 利用 CORBA CDR 的“对齐”和“字节次序”规则，避免昂贵的“内存复制”和“字节交换”操作。

• 提供优化的“字节交换”代码，在常见硬件平台（例如 Intel x86）上使用内联汇编语言指令，在其他平台上使用标准 htons()、htonl()、ntohs()和 ntohl()宏/函数。

• 支持 8 位字节数据缓冲区的“零复制（zero copy）”整编和解编。

• 在“不将 ASCII 或 UNICODE 用作原始字符集”的平台上，用户可以定制字符集转换器（character set translators）。

ACE CDR “流（streaming）”类的接口如图 4.3 所示。ACE_OutputCDR 类的主要方法见下表：

方 法	说 明
ACE_OutputCDR()	创建一个空的 CDR 流，以执行“插入”操作。
write_*()	将“原始类型”数据“插入”到流中。例如，write_ushort()。
write_*_array()	将一个由“原始类型”数据组成的数组“插入”到流中。例如，write_long_array()。
operator<<()	针对每一种“原始类型”定义的“插入”运算符。
good_bit()	如果在流中发现错误，则返回 0。
total_length()	返回流中的字节数。
begin()	返回一个指针，指向链表中的第一个消息块。
end()	返回一个指针，指向链表中的最后一个消息块。

类似地，ACE_Input_CDR 类的主要方法见下表：

方 法	说 明
ACE_InputCDR()	创建一个空的 CDR 流，以执行“提取”操作。
read_*()	从流中“提取”一个“原始类型”数据。例如，read_char()。
read_*_array()	从流中“提取”一个由“原始类型”数据组成的数组。例如，read_octet_array()。
operator>>()	针对每一个“原始类型”定义的“提取”运算符。
good_bit()	如果在流中发现错误，则返回 0。
steal_contents()	返回“包含当前 CDR 流的底层 ACE_Message_Block”的副本。

ACE_InputCDR 类和 ACE_OutputCDR 类既能将“无类型（untyped）”的数据转化到“类型化（typed）”的缓冲区中，也能执行逆操作；所以，程序员在使用这些类时必须十分小心，以防止破坏类型系统（type system）。为避免所有这些问题，一个方法是用“分布式中间件”（例如，CORBA [Obj01]，以及第 B.1.4 节介绍的 ACE ORB(TAO) [SLM98]）来编程。

示例

ACE CDR 流类针对原始类型及数组，预定义了 operator <<和>>。针对自己的数据类型，ACE 应用程序也必须定义这些运算符。[1] 为了演示这些运算符，我们针对客户程

[1] 这些运算符也可以借助工具自动生成，如 TAO 提供的 CORBA 接口定义语言（IDL，Interface Definition Language）编译器 [AOSK00]。

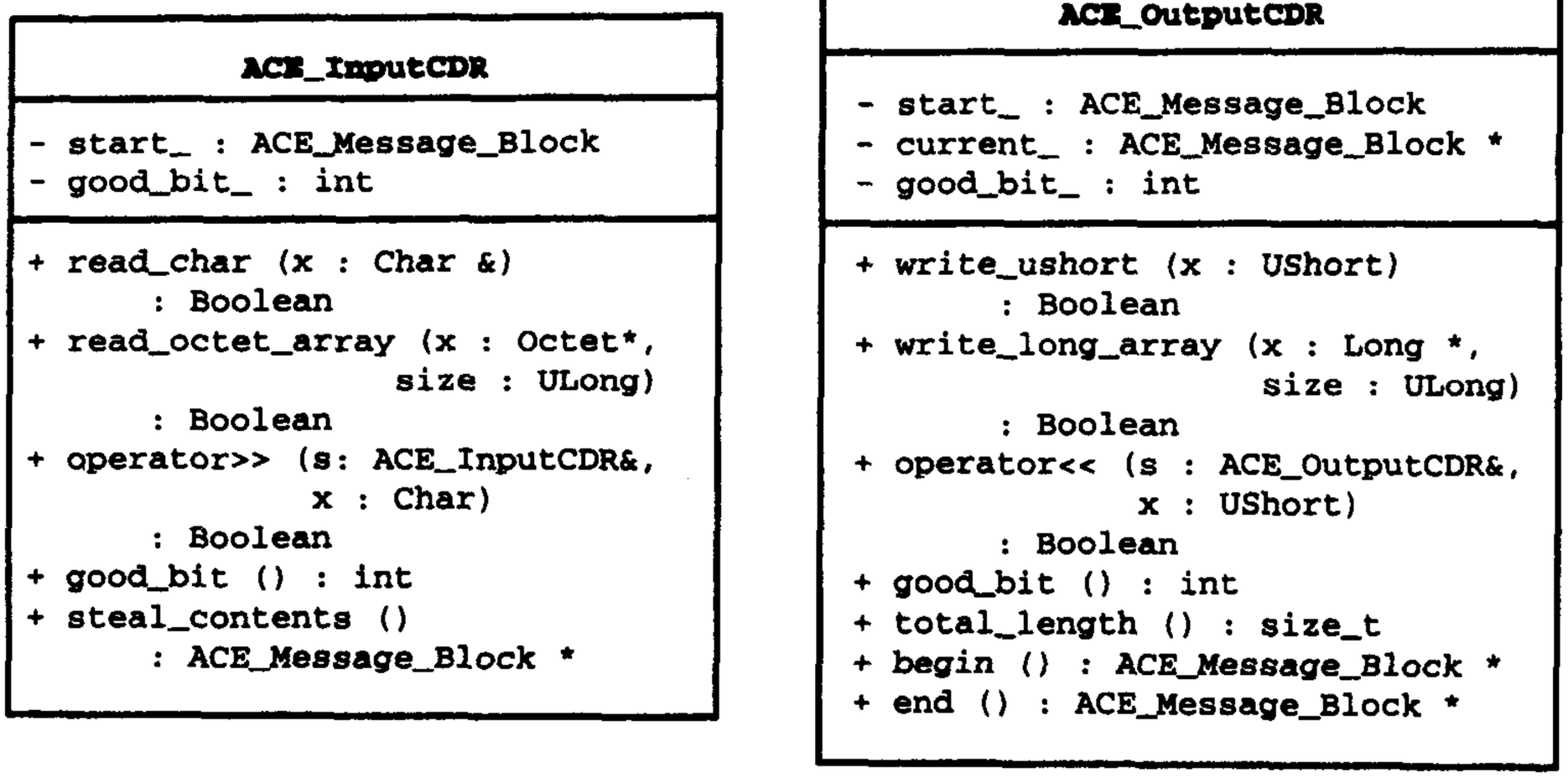


图 4.3：ACE_InputCDR 和 ACE_OutputCDR 类图

序和日志服务器中使用的 ACE_LOG_Record 类，提供了 ACE CDR 的“插入”和“提取”运算符。这个 C++类包含数个字段，以及一个具有可变长度的消息缓冲区。因此，我们的“插入”操作要将每一个字段整编（marshal）到一个 ACE_OutputCDR 对象中，如下所示：

```
int operator<< (ACE_OutputCDR &cdr,
                const ACE_Log_Record &log_record)
{
  size_t msglen = log_record.msg_data_len ();

  // Insert each <log_record> field into the output CDR stream.
  cdr << ACE_CDR::Long (log_record.type ());
  cdr << ACE_CDR::Long (log_record.pid ());
  cdr << ACE_CDR::Long (log_record.time_stamp ().sec ());
  cdr << ACE_CDR::Long (log_record.time_stamp ().usec ());
  cdr << ACE_CDR::ULong (msglen);

  cdr.write_char_array (log_record.msg_data (), msglen);

  return cdr.good_bit ();
}
```

相应地，我们的“提取”运算符将每一个字段从 ACE_InputCDR 对象中解编（demarshal）出来，填充到 ACE_Log_Record 对象中：

```
int operator>> (ACE_InputCDR &cdr,
                ACE_Log_Record &log_record)
{
  ACE_CDR::Long type;
  ACE_CDR::Long pid;
  ACE_CDR::Long sec, usec;
  ACE_CDR::ULong buffer_len;

  // Extract each field from input CDR stream into <log_record>.
  if ((cdr >> type) && (cdr >> pid) && (cdr >> sec)
      && (cdr >> usec) && (cdr >> buffer_len)) {
    ACE_TCHAR log_msg[ACE_Log_Record::MAXLOGMSGLEN+1];
    log_record.type (type);
    log_record.pid (pid);
    log_record.time_stamp (ACE_Time_Value (sec, usec));
    cdr.read_char_array (log_msg, buffer_len);
    log_msg[buffer_len] = '\0';
    log_record.msg_data (log_msg);
  }

  return cdr.good_bit ();
}
```

一会儿，我们将使用上面的“插入”和“提取”运算符，来简化我们的网络日志服务程序。

4.4 日志服务器的初始版本

在网络日志服务程序的第一个版本中，我们将确定初步设计，以及“可复用（reusable）”类的核心实现。这个日志服务器在一个 TCP 端口号上侦听；这个端口号在操作系统的网络服务文件中被定义为 ace_logger——这也是很多网络服务器习惯的做法。例如，在 UNIX 的/etc/service 文件中，可能会有下面这样一行：

```
ace_logger 9700/tcp # Connection-oriented Logging Service
```

根据需要，客户程序可以指定一个 TCP 端口和 IP 地址，让客户程序和日志服务器在那里交换日志记录。如果没有指定这一信息，则端口号会位于服务程序的数据库中，且主机名被假定为 ACE_DEFAULT_SERVER_HOST；在大多数 OS 平台上，这个主机名会被定义为“localhost”。一旦建立连接，客户便可以向日志服务器发送日志记录，而服务器则将每一条记录写入文件。本节将向你介绍一组可复用的类；在本书演示的

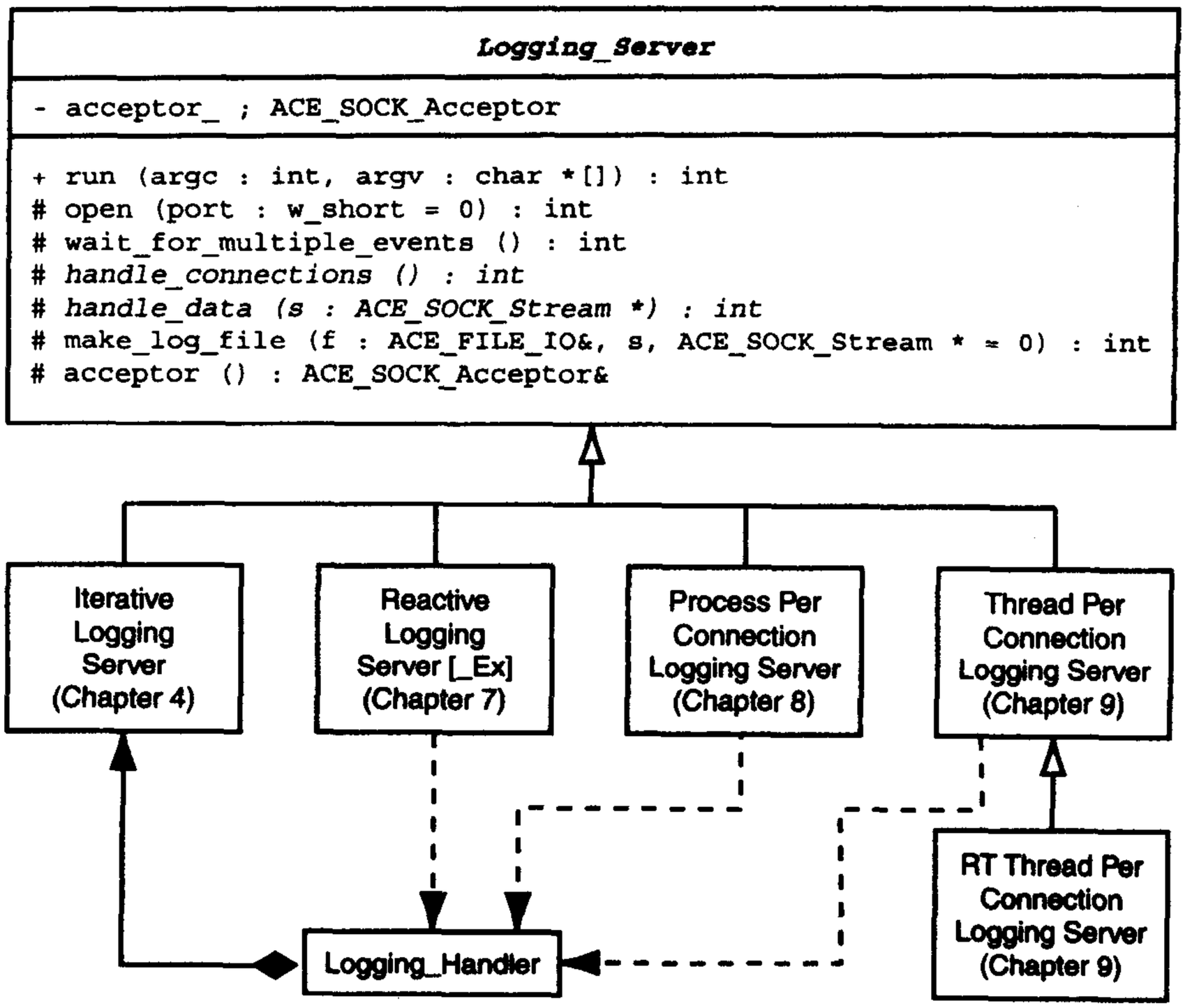


图 4.4：日志服务器示例类

所有日志服务器程序中，这些类被用来处理“被动连接”的建立和数据传输。

4.4.1 Logging_Server 基类

在 4.3 节，我们介绍了 ACE_Message_Block、ACE_Output_CDR、ACE_InputCDR；接下来，我们将在一个新的基类（base class）中使用这些类。在整本书中，这个基类都会被用来简化我们的日志服务器的实现。图 4.4 演示了我们的 Logging_Server 抽象基类（abstract base class）、将要在第 4.4.2 节开发的 Logging_Handler 类以及将要在后续章节开发的具体的日志服务器类。我们将 Logging_Server 基类的定义放在下面这个名为 Logging_Server.h 的头文件中：

```
#include "ace/FILE_IO.h"
#include "ace/SOCK_Acceptor.h"

// Forward declaration.
class ACE_SOCK_Stream;

class Logging_Server
{
public:
  // Template Method that runs logging server's event loop.
  virtual int run (int argc, char *argv[]);

protected:
  // The following four methods are "hooks" that can be
  // overridden by subclasses.
  virtual int open (u_short logger_port = 0);
  virtual int wait_for_multiple_events () { return 0; }
  virtual int handle_connections () = 0;
  virtual int handle_data (ACE_SOCK_Stream * = 0) = 0;
  // This helper method can be used by the hook methods.
  int make_log_file (ACE_FILE_IO &, ACE_SOCK_Stream * = 0);
  // Close the socket endpoint and shutdown ACE.
  virtual ~Logging_Server () {
    acceptor_.close ();
  }
  // Accessor.
  ACE_SOCK_Acceptor &acceptor () {
    return acceptor_;
  }

private:
  // Socket acceptor endpoint.
  ACE_SOCK_Acceptor acceptor_;
}
```

后续所有网络日志服务例程都包含这个头文件，继承 Logging_Server 基类，并重写（override）和复用（reuse）其方法。在后文中，我们将对这些方法一一作介绍。“实现”文件 Logging_Server.cpp 包含下面的 ACE 头文件：

```
#include "ace/FILE_Addr.h"
#include "ace/FILE_Connector.h"
#include "ace/FILE_IO.h"
#include "ace/INET_Addr.h"
#include "ace/SOCK_Stream.h"
#include "Logging_Server.h"
```

Logging_Server::run()模板方法

这个公有（public）方法执行规范的“初始化”和“事件循环”步骤，本书介绍的大多数日志服务器都要执行这一步骤。

```
int Logging_Server::run (int argc, char *argv[])
{
  if (open (argc > 1 ? atoi (argv[1]) : 0) == -1)
    return -1;

  for (;;) {
    if (wait_for_multiple_events () == -1) return -1;
    if (handle_connections () == -1) return -1;
    if (handle_data () == -1) return -1;
  }
  return 0;
}
```

上面是 Template Method（模板方法）模式 [GHJV95]的一个例子，它定义了“一个操作中的一个算法”的骨架，但将某些执行步骤延迟到了 hook（挂钩）方法中，而子类可以重写这些 hook 方法。在 run()这个模板方法中，所调用的 open()、wait_for_multiple_events()、handle_data()和 handle_connections()都是 hook 方法，它们可以被子类重写。

Logging_Server 中的 Hook 方法

Logging_Server 类中的每一个 hook 方法都承担一个角色，且具有缺省行为，以下是对它们的简要介绍。

Logging_Server::open()。这个方法用来初始化“服务器的地址”和“Logging_Server 的接受者端点（acceptor endpoint）”，以被动地在一个指定端口号上侦听。这个方法虽然可以被子类重写，但以下演示的缺省实现可以满足本书所有例子的需要：

```
int Logging_Server::open (u_short logger_port)
{
  // Raises the number of available socket handles to
  // the maximum supported by the OS platform.
  ACE::set_handle_limit ();

  ACE_INET_Addr server_addr;
  int result;
```

```
  if (logger_port != 0)
    result = server_addr.set (logger_port, INADDR_ANY);
  else
    result = server_addr.set ("ace_logger", INADDR_ANY);
  if (result == -1)
    return -1;

  // Start listening and enable reuse of listen address
  // for quick restarts.
  return acceptor_.open (server_addr, 1);
}
```

虽然 ACE_INET_Addr 的构造函数可以接受端口号和主机名作为参数，但我们在此避免这样做，原因是：

1. 我们允许用户传递一个端口号，或者使用缺省服务名称，这需要不同的调用。

2. 网络地址函数（例如 gethostbyname()）可能会被调用。这会让程序员感到意外，因为他们不希望构造函数的执行延迟至服务器主机地址解析完之后。

3. 如果有错误发生，我们需要发出正确的诊断信息。

在第 A.6.3 节，我们还解释了 ACE 为什么不使用原始 C++异常（exception）来传播错误。

请注意 acceptor_.open 调用中的第二个参数。它将 SO_REUSEADDR 这个“socket 选项”设置在“接受者（acceptor）”socket 上，允许程序停止之后立即重新启动。这就避免了和“前一次运行留下的 TIME_WAIT 状态”中的 socket 发生任何冲突；否则，它会使新的日志服务器在数分钟内无法侦听日志客户。

Logging_Server::wait_for_multiple_events()。这个方法的作用是：等待多个事件发生。它的缺省实现是一个空操作，即简单地返回 0。第 7 章介绍了基于 select()的“同步事件多路分离”机制；在使用这一机制的日志服务器实现中，前面所说的缺省实现会被重写。

Logging_Server::handle_connections()。这个 hook 方法的任务是：接受一个或多个来自客户的连接。我们将它定义为纯虚（pure virtual）方法，以确保子类（subclass）会实现它。

Logging_Server::handle_data()。这个 hook 方法的任务是：接收来自客户端的日志记录，并将其写入日志文件。我们也将其定义为纯虚方法，以保证子类会去实现它。

副栏 8：ACE“文件”Wrapper Facade

ACE 遵循 Wrapper Facade 模式，对操作系统平台上的“非缓冲（unbuffered）”文件操作机制进行了封装。和大多数 ACE IPC wrapper facade 一样，ACE“文件”类将以下二者实施了分离：

- **初始化工厂**，例如，ACE_FILE_Connector，用于打开和（或）创建文件。
- **数据传输类**，例如，ACE_FILE_IO，用于为“通过 ACE_FILE_Connector 打开的文件”读/写数据。

ACE IPC 和“文件”wrapper facade 的对称性体现了 ACE 设计的普遍性，也为将 IPC 机制融入到更高级的 ACE 框架中奠定了基础。关于 ACE 框架的介绍，请参阅 [SH]。

Logging_Server::make_log_file()方法

这个辅助方法（helper method）会被 hook 方法使用，以通过副栏 8 介绍的 ACE“文件”wrapper facade 来初始化日志文件。

```
int Logging_Server::make_log_file (ACE_FILE_IO &logging_file,
                                   ACE_SOCK_Stream *logging_peer)
{
  char filename[MAXHOSTNAMELEN + sizeof (".log")];

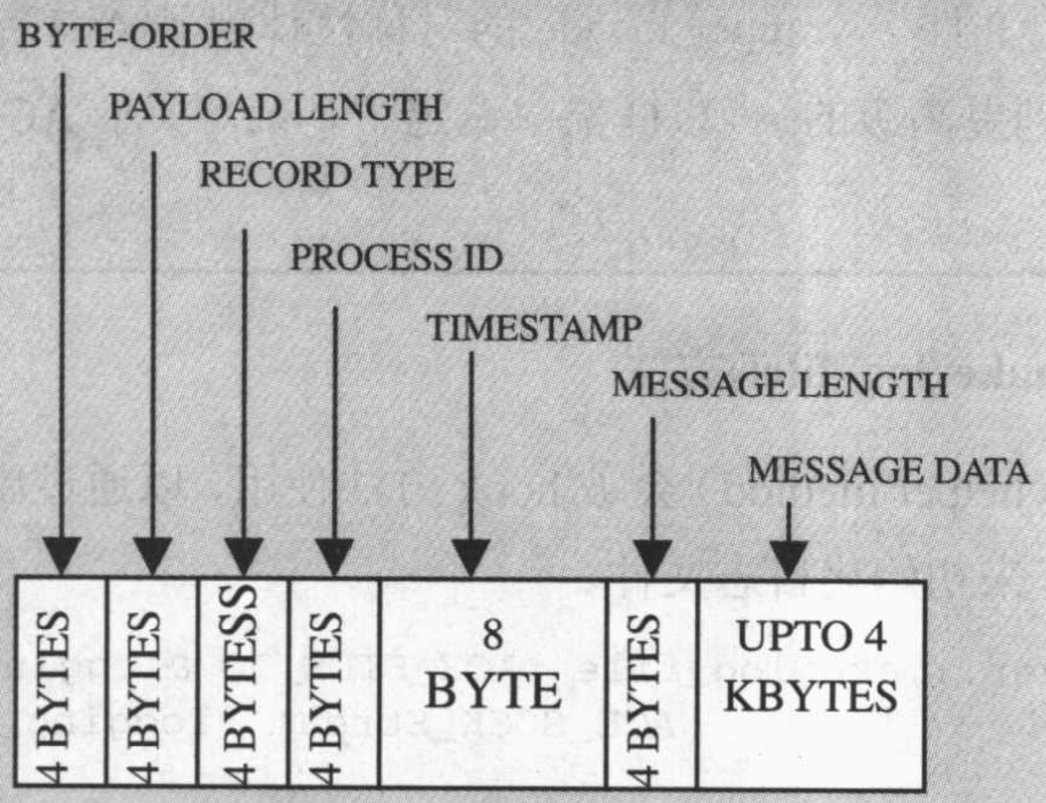
  if (logging_peer != 0) { // Use client host name as file name.
    ACE_INET_Addr logging_peer_addr;
    logging_peer->get_remote_addr (logging_peer_addr);
    logging_peer_addr.get_host_name (filename, MAXHOSTNAMELEN);
    strcat (filename, ".log");
  }
  else
    strcpy (filename, "logging_server.log");

  ACE_FILE_Connector connector;
  return connector.connect (logging_file,
                            ACE_FILE_Addr (filename),
                            0, // No time-out.
                            ACE_Addr::sap_any, // Ignored.
                            0, // Don't try to reuse the addr.
                            O_RDWR|O_CREAT|O_APPEND,
                            ACE_DEFAULT_FILE_PERMS);
}
```

默认情况下，我们将这个日志文件命名为“logging_server.log”。我们也可以用“被连接客户的机器名”重写这个名称，就像第 7.4 节演示的那样。

副栏 9：日志服务程序的消息分帧协议

TCP 是一种“字节流”协议；所以，为了分离数据流中的日志记录，我们需要“应用程序级”的“消息分帧”协议。我们将使用一个“长度为 8 个字节、采用 CDR 编码”的数据头，这个数据头包含“字节顺序标志”、“数据有效内容的长度（payload length）”等信息；数据头之后是日志记录的内容，如下所示：



4.4.2 Logging_Handler 类

这个类用在日志服务器中，它基于副栏 9 介绍的“消息分帧”协议，对 I/O 和日志记录的处理工作进行了封装。在这个协议实现（implementation）中，发送方的情况如第 4.5 节中介绍的 Logging_Client::send() 方法所示，接收方如稍后介绍的 Logging_Handler::recv_log_record()方法所示。

Logging_Handler 类的定义位于一个名为 Logging_Handler.h 的头文件中。

```
#include "ace/FILE_IO.h"
#include "ace/SOCK_Stream.h"

class ACE_Message_Block; // Forward declaration.
```

```
class Logging_Handler
{
protected:
  // Reference to a log file.
  ACE_FILE_IO &log_file_;
  // Connected to the client.
  ACE_SOCK_Stream logging_peer_;
public:
  // Initialization and termination methods.
  Logging_Handler (ACE_FILE_IO &log_file)
    : log_file_ (log_file) {}
  Logging_Handler (ACE_FILE_IO &log_file,
                   ACE_HANDLE handle)
    : log_file_ (log_file) { logging_peer_.set_handle (handle); }
Logging_Handler (ACE_FILE_IO &log_file,
                 const ACE_SOCK_Stream &logging_peer)
  : log_file_ (log_file), logging_peer_ (logging_peer) {}
int close () { return logging_peer_.close (); }
// Receive one log record from a connected client.  Returns
// length of record on success and <mblk> contains the
// hostname, <mblk->cont()> contains the log record header
// (the byte order and the length) and the data. Returns -1
// on failure or connection close.
int recv_log_record (ACE_Message_Block *&mblk);

// Write one record to the log file. The <mblk> contains the
// hostname and the <mblk->cont> contains the log record.
// Returns length of record written on success; -1 on failure.
int write_log_record (ACE_Message_Block *mblk);

// Log one record by calling <recv_log_record> and
// <write_log_record>. Returns 0 on success and -1 on failure.
int log_record ();

  ACE_SOCK_Stream &peer () { return logging_peer_; } // Accessor.
};
```

以下给出 recv_log_record()、write_log_record()和 log_record()方法的实现。

Logging_Handler::recv_log_record()。在网络日志服务程序中，这个方法实现了“消息成帧”协议（见副栏 9）的接收方。它使用了 ACE_SOCK_Stream、ACE_Message_Block、ACE_InputCDR 等类、以及第 4.3 节中定义的“提取（extraction）”

运算符 operator>>，用以从客户端读取一条完整的日志记录。这段代码具有“可移植性”和“互操作性”，因为在解编（demarshal）“从网络接收到”的内容时，使用的是 ACE_InputCDR 类。

```
1 int Logging_Handler::recv_log_record (ACE_Message_Block *&mblk)
2 {
3   ACE_INET_Addr peer_addr;
4   logging_peer_.get_remote_addr (peer_addr);
5   mblk = new ACE_Message_Block (MAXHOSTNAMELEN + 1);
6   peer_addr.get_host_name (mblk->wr_ptr (), MAXHOSTNAMELEN);
7   mblk->wr_ptr (strlen (mblk->wr_ptr ()) + 1); // Go past name.
8
9   ACE_Message_Block *payload =
10    new ACE_Message_Block (ACE_DEFAULT_CDR_BUFSIZE);
11  // Align Message Block for a CDR stream.
12  ACE_CDR::mb_align (payload);
13
14  if (logging_peer_.recv_n (payload->wr_ptr (), 8) == 8) {
15    payload->wr_ptr (8);     // Reflect addition of 8 bytes.
16
17    ACE_InputCDR cdr (payload);
18
19    ACE_CDR::Boolean byte_order;
20    // Use helper method to disambiguate booleans from chars.
21    cdr >> ACE_InputCDR::to_boolean (byte_order);
22    cdr.reset_byte_order (byte_order);
23
24    ACE_CDR::ULong length;
25    cdr >> length;
26
27    payload->size (length + 8 + ACE_CDR::MAX_ALIGNMENT);
28
29    if (logging_peer_.recv_n (payload->wr_ptr(), length) > 0) {
30      payload->wr_ptr (length);   // Reflect additional bytes.
31      mblk->cont (payload);
32      return length; // Return length of the log record.
33    }
34  }
35  payload->release ();
36  mblk->release ();
37  payload = mblk = 0;
38  return -1;
39 }
```

第 3~7 行 分配了一个新的 ACE_Message_Block，用来保存 Logging_peer_的主机名。我们十分小心地为主机名添加了 NULL 终止符，从而保证消息块的当前长度中只包含主机名，此长度为 wr_ptr() – rd_ptr()。

第 9~12 行 单独创建了一个 ACE_Message_Block，用来保存日志记录。因为数据头的长度已知（8 字节），所以，首先接收数据头。数据头接收完之后，要用 CDR 设施来对它进行解编（demarshal）；所以，对于新消息块，得采取一些和 CDR 有关的准备措施。CDR 能够可移植地对数据解编，但它也要求：被解编的数据必须始于“按 8 字节对齐”的数据边界。ACE_Message_Block 没有提供任何“对齐”保证；所以，在通过 payload->wr_ptr()接收数据头之前，要调用 ACE_CDR::mb_align()，强行将数据正确对齐。对齐操作可能会在数据块内部缓冲区的开始处产生一些无用的字节；所以，首先必须调整 payload，使它的长度大于要接收的 8 字节（ACE_DEFAULT_CDR_BUFSIZE 的缺省值为 512）。

第 14~15 行 在成功调用 recv_n()并接收到“具有固定长度”的数据头之后，payload 的“写”指针会前进，以反映消息块增加了 8 个字节的长度。

第 17-25 行 首先创建一个 CDR 对象，用来解编 payload 消息块中 8 字节的数据头。CDR 对象复制这个数据头，避免解编数据头时会对 payload 消息块作任何改动。因为“字节顺序标志”的类型为 ACE_CDR::Boolean，所以不管客户的字节顺序如何，它都可以被提取出来。然后，根据输入中指定的字节顺序，确定 CDR 流的字节顺序，提取具有不定长度的日志记录 payload 的长度。

第 27~32 行 既然知道了日志记录 payload 的长度，就可以调整 payload 消息块的大小，以容纳整个记录。紧接着数据头，第二个 recv_n()调用将日志记录的余下部分添加到 payload 消息块中。如果一切进展顺利，则在 payload 中更新“写”指针，以反映数据的增加，并通过“连接字段（continuation field）”将 payload 链接到 mblk，然后返回日志记录的长度。

第 35~38 行 出错情况在此结束。所以，需要释放 mblk 和 payload 中的内存，以防止泄漏。

为了可移植、高效地接收一条日志记录，recv_log_record()方法有效地利用了一个 CDR 流和一个消息块。它利用消息块的“连接链（continuation chain）”来存储客户程

序的主机名，接下去是保存日志记录的消息块。虽说这两项内容是独立的，但它们是作为一个整体在应用程序间传递的，如图 4.5 所示。

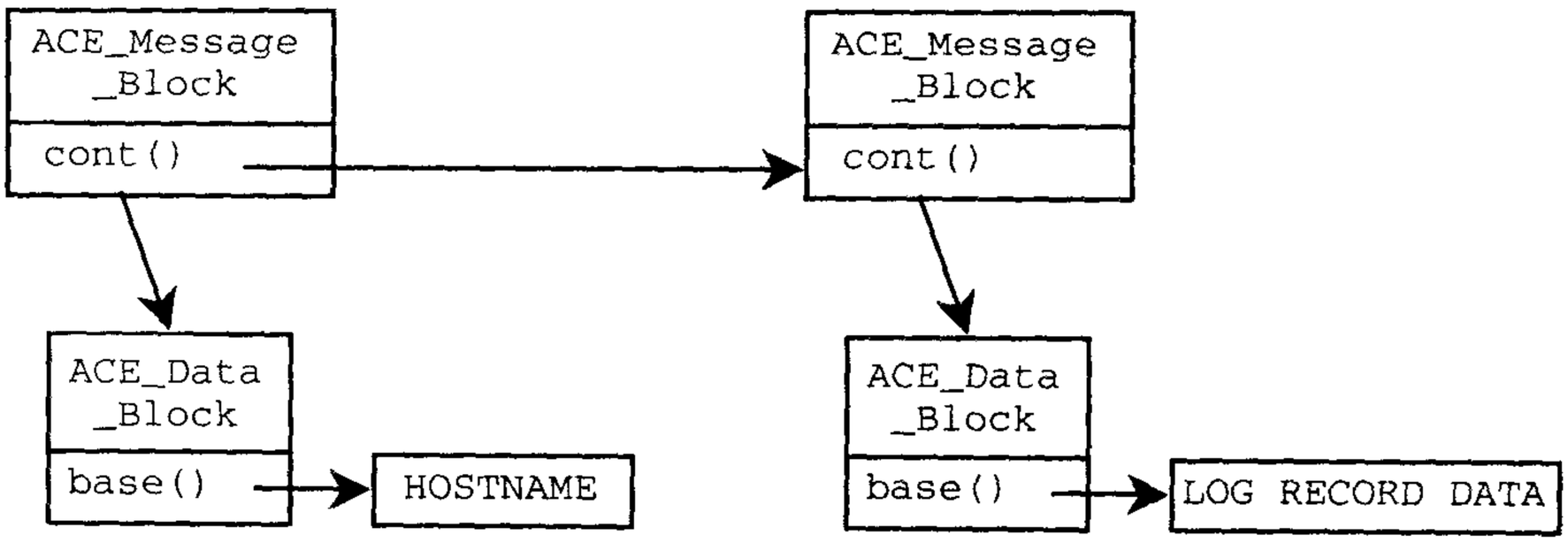


图 4.5：日志记录信息的消息块链

Logging_Handler::write_log_record()。这个方法结合使用了 ACE_FILE_IO、ACE_Message_Block、ACE_InputCDR 等类，以及第 4.3 节定义的“提取”运算符 operator>>；它会对“从客户接收到”的主机名和日志记录数据执行格式化操作，然后将其写入日志文件。

```
1 int Logging_Handler::write_log_record (ACE_Message_Block
  *mblk)
2 {
3   if (log_file_->send_n (mblk) == -1) return -1;
4
5   if (ACE::debug ()) {
6     ACE_InputCDR cdr (mblk->cont ());
7     ACE_CDR::Boolean byte_order;
8     ACE_CDR::ULong length;
9     cdr >> ACE_InputCDR::to_boolean (byte_order);
10    cdr.reset_byte_order (byte_order);
11    cdr >> length;
12    ACE_Log_Record log_record;
13    cdr >> log_record;  // Extract the <ACE_log_record>.
14    log_record.print (mblk->rd_ptr (), 1, cerr);
15  }
16
17  return mblk->total_length ();
18 }
```

第 3 行 对方主机名位于 mblk 中，日志记录位于 mblk 的“连接链”中。我们用 ACE_FILE_IO 的 send_n()方法来写（write）所有“通过 cont()指针链接起来”的消息

块。在内部，这个方法使用了 OS 的“集中写入（gather-write）”操作，以减少*跨领域代价（domain-crossing penalty）*。既然日志文件已经以 CDR 格式写出（write out），就需要另外编写一个程序，用来读取文件，并以一种可读的格式来显示它的值。

第 5~18 行 如果是在调试模式下，我们会根据日志记录数据构造一个 CDR 流，并将它的内容输出到 cerr。因为日志记录的头部还是在日志记录消息块中，所以“字节顺序”必须被提取出来，以保证日志记录本身得以正确解编。虽然头部的 length 没有被使用，我们还是将它提取出来，以保证：在为了解编日志记录而调用“提取”运算符的时候，CDR 流会被正确地置于日志记录的起始处。这个方法返回的是“写到日志文件中”的字节数。

Logging_Handler::log_record()。这个方法使用了 recv_log_record()和 write_log_record()方法；其作用是：从 socket 中读取一条日志记录，并将它写到日志文件中。

```
int Logging_Handler::log_record ()
{
  ACE_Message_Block *mblk = 0;
  if (recv_log_record (mblk) == -1)
    return -1;
  else {
    int result = write_log_record (mblk);
    mblk->release (); // Free up the entire contents.
    return result == -1 ? -1 : 0;
  }
}
```

release()调用会删除 mblk 中的客户主机名和 mblk->cont()中的日志记录。

4.4.3 Iterative_Logging_Server 类

以下代码演示的是：如何通过 ACE Socket wrapper facade 实现“循环式”日志服务器。这个例子继承了 Logging_Server 基类（base class）中的各个 hook 方法，像下面这样循环处理客户的日志请求：

1. hook 方法 handle_connection()用于接受“来自客户的新连接”。

2. hook 方法 handle_data()用来读取和处理“来自客户的日志记录”，直至连接关闭或发生错误。

3. 回到第1步。

首先，若要包含（include）这个例子则需要的头文件。

```
#include "ace/FILE_IO.h"
#include "ace/INET_Addr.h"
#include "ace/Log_Msg.h"

#include "Logging_Server.h"
#include "Logging_Handler.h"
```

接着，创建一个“继承自 Logging_Server”的 Iterative_Logging_Server 类，并将这个类放进一个名为 Iterative_Logging_Server.h 的头文件中。我们还定义一些数据成员，用来接收来自客户端的日志记录，并将数据写入日志文件。

```
class Iterative_Logging_Server : public Logging_Server
{
protected:
  ACE_FILE_IO log_file_;
  Logging_Handler logging_handler_;

public:
  Iterative_Logging_Server (): logging_handler_ (log_file_) {}

  Logging_Handler &logging_handler () {
    return logging_handler_;
  }

protected:
  // Other methods shown below...
};
```

open()是个 hook 方法，它通过第 4.4.1 节中定义的 make_log_file()辅助方法（helper method），创建并（或）打开一个日志文件。如果 make_log_file()调用失败，则使用副栏 10 介绍的 ACE_ERROR_RETURN 宏来输出失败原因，并返回一个错误值。如果调用成功，则通过父类的 hook 方法 open()来初始化 ACE_INET_Addr 网络地址，日志服务器正是在这个地址上侦听客户连接的。

```
virtual int open (u_short logger_port) {
  if (make_log_file (log_file_) == -1)
    ACE_ERROR_RETURN ((LM_ERROR, "%p\n", "make_log_file()"),
                      -1);
  return Logging_Server::open (logger_port);
}
```

析构函数用于关闭日志文件，如下所示：

```
virtual ~Iterative_Logging_Server () {
  log_file_.close ();
}
```

副栏 10：ACE“调试”及“错误”宏

ACE 提供了一组宏（macro），它们通过类似 printf()的格式，加强了调试及出错信息的输出功能。这之中最常用的宏有 ACE_DEBUG、ACE_ERROR、ACE_ERROR_RETURN 等，它们封装了 ACE_Log_Msg::log()方法。因为这个方法接受可变数量的参数，因此所有这些宏的第一个参数实际上是一个“复合”参数，此参数被包装在另一对括号中，使得在 C++预处理器看来是一个参数。

ACE_Log_Msg::log()方法的第一个参数是“严重度（severity）”代码。例如，LM_ERROR 表示普通错误，LM_DEBUG 表示诊断信息。下一个参数是一个“格式”字符串，它可以接受 printf()的大多数转义说明符。ACE 还定义了另外一些格式说明符，这些说明符在“跟踪”和“记录”操作中十分有用：

格式	行为
%l	显示出现错误的行号
%N	显示出现错误的文件名
%n	显示程序名
%p	显示当前进程 ID
%P	接受一个 const char * 参数，然后显示这个字符串以及 error 对应的错误字符串（如 perror()）
%T	显示当前时间
%t	显示调用者线程的 ID

ACE_ERROR_RETURN 宏是记录错误信息并从当前函数返回一个值的“捷径”。因此，它取两个参数：第一个参数和其他宏一样；第二个参数是消息记录结束之后从函数返回的值。

既然要实现的是循环式服务器，就可以复用（reuse）派生的 wait_for_multiple_events()方法。因此，在接收到下一个客户连接之前，handle_connection()方法会阻塞。

```
virtual int handle_connections () {
  ACE_INET_Addr logging_peer_addr;

  if (acceptor ().accept (logging_handler_.peer (),
                          &logging_peer_addr) == -1)
    ACE_ERROR_RETURN ((LM_ERROR, "%p\n", "accept()"), -1);
  ACE_DEBUG ((LM_DEBUG, "Accepted connection from %s\n",
              logging_peer_addr.get_host_name ()));
  return 0;
}
```

服务器接受来自客户的新连接之后，hook 方法 handle_data()继续接收来自客户的日志记录，直至连接关闭或有错误发生。

```
virtual int handle_data (ACE_SOCK_Stream *) {
  while (logging_handler_.log_record () != -1)
    continue;

  logging_handler_.close (); // Close the socket handle.
  return 0;
}
```

main()函数实例化（instantiate）一个 Iterative_Logging_Server 对象，并调用 run()方法来循环处理所有的客户请求。这段代码位于文件 Iterative_Logging_Server.cpp 中。

```
#include "ace/Log_Msg.h"
#include "Iterative_Logging_Server.h"

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

  if (server.run (argc, argv) == -1)
    ACE_ERROR_RETURN ((LM_ERROR, "%p\n", "server.run()"), 1);
  return 0;
}
```

这个例子证明，在编写网络应用程序时，使用 ACE 要比使用“原始 C Socket API 调用”容易得多。使用 ACE wrapper façade 类时，第 2.3 节确定的所有关于 Socket API 的偶发复杂性都能得以消除。消除了这些复杂性，应用程序开发者就能被解放出来，将注意力专注在重要的程序逻辑上。请注意，上面给出的大多数源代码都和第 4.4.1 节给出的 Logging_Handler 方法中的代码有关，也就是说，和那些“针对特定应用”的整

（解）编及 I/O 代码有关。而且，直接用 Socket API 编写应用程序时会出现的“传统的”socket 操作、缓冲区管理和字节顺序代码，在这里完全不存在。

以上是我们的日志服务器的第一个实现。尽管它有了很多改进，但仍存在着循环式服务器设计上的局限性。在本书后续示例中，将介绍如何运用“并发”模式和 ACE wrapper facade 来解决这些问题。因为日志服务器分离了 Logging_Server 和 Logging_Handler 这两个角色，所以，可以在不改动整个设计的情况下修改它的实现细节。

4.5 客户程序

在下面的客户应用程序中，将演示如何使用 ACE wrapper facade 建立连接、整编日志记录、向日志服务器发送数据。这个例程从标准输入设备读取数据行，并将每一行数据作为一个单独的日志记录发送给日志服务器。如果从标准输入设备读入 EOF，程序将停止执行。首先，针对代码中用到的类，得包含所需的 ACE 头文件。一个好的做法是，针对用到的类，明确地包含所有的 ACE 头文件。例如，ace/OS.h 包含 ACE_DEFAULT_SERVER_HOST 的定义，后者被用作缺省主机名。

```
#include "ace/OS.h"
#include "ace/CDR_Stream.h"
#include "ace/INET_Addr.h"
#include "ace/SOCK_Connector.h"
#include "ace/SOCK_Stream.h"
#include "ace/Log_Record.h"
#include "ace/streams.h"
#include <string>
```

上面的代码中包含了 ace/streams.h，后者提供了条件代码，使得 C++ I/O 流的使用更具可移植性。基于 ACE 中的配置设定，它包含具有正确形式的头文件（例如，是<iostream>还是<iostream.h>）。如果 I/O 流是在 std 命名空间中提供，ace/streams.h 文件会将用到的类引入全局命名空间。

首先定义 Logging_Client 类。这个类有一个 send()方法，用来在已连接的 ACE_SOCK_Stream 上传输 ACE_Log_Record。这个类还实现了一个辅助方法和一个析构函数，前者返回 ACE_SOCK_Stream 实例的引用，后者关闭底层 socket 句柄。

```
class Logging_Client
{
public:
  // Send <log_record> to the server.
  int send (const ACE_Log_Record &log_record);

  // Accessor method.
  ACE_SOCK_Stream &peer () { return logging_peer_; }

  // Close the connection to the server.
  ~Logging_Client () { logging_peer_.close (); }

private:
  ACE_SOCK_Stream logging_peer_; // Connected to server.
};
```

Logging_Client::send()方法实现的是“消息分帧”机制的发送方（关于“消息分帧”机制，参见副栏 9 的介绍）。这个方法具有可移植性和互操作性，因为它使用了第 4.3 节中定义的“插入”运算符（operator<<），将日志记录的内容整编到了 ACE_OutputCDR 流中。

```
1 int Logging_Client::send (const ACE_Log_Record &log_record)
  {
2   const size_t max_payload_size =
3       4 // type()
4     + 8 // timestamp
5     + 4 // process id
6     + 4 // data length
7     + MAXLOGMSGLEN // data
8     + ACE_CDR::MAX_ALIGNMENT; // padding;
9
10  ACE_OutputCDR payload (max_payload_size);
11  payload << log_record;
12  ACE_CDR::ULong length = payload.total_length ();
13
14  ACE_OutputCDR header (ACE_CDR::MAX_ALIGNMENT + 8);
15  header << ACE_OutputCDR::from_boolean
    (ACE_CDR_BYTE_ORDER);
16  header << ACE_CDR::ULong (length);
17
18  iovec iov[2];
19  iov[0].iov_base = header.begin ()->rd_ptr ();
20  iov[0].iov_len  = 8;
21  iov[1].iov_base = payload.begin ()->rd_ptr ();
22  iov[1].iov_len  = length;
```

```
23
24    return logging_peer_.sendv_n (iov, 2);
25 }
```

第 2~12 行 为 ACE_Log_Record 分配足够的空间，将 log_record 的内容“插入”到 playload CDR 流中，并得到流（stream）使用的字节数。

第 14~16 行 创建一个“采用 CDR 编码”的数据头；这样，接收者就可以确定“接收到的 CDR 流”的字节顺序和大小。

第 18~24 行 使用了一个 iovec，并调用 sendv_n()这个“集中写入”方法，有效地发送数据头和 payload CDR 流。

最后，给出客户应用程序的 main()函数。

```
 1  int main (int argc, char *argv[])
 2 {
 3   u_short logger_port =
 4     argc > 1 ? atoi (argv[1]) : 0;
 5   const char *logger_host =
 6     argc > 2 ? argv[2] : ACE_DEFAULT_SERVER_HOST;
 7   int result;
 8
 9   ACE_INET_Addr server_addr;
10
11   if (logger_port != 0)
12     result = server_addr.set (logger_port, logger_host);
13   else
14     result = server_addr.set ("ace_logger", logger_host);
15   if (result == -1)
16     ACE_ERROR_RETURN((LM_ERROR,
17                       "lookup %s, %p\n",
18                       logger_port == 0 ? "ace_logger" : argv[1],
19                       logger_host), 1);
20
21   ACE_SOCK_Connector connector;
22   Logging_Client logging_client;
23
24   if (connector.connect (logging_client.peer (),
25                          server_addr) < 0)
26     ACE_ERROR_RETURN ((LM_ERROR,
27                        "%p\n",
28                        "connect()"),
29                       1);
30
31   // Limit the number of characters read on each record.
32   cin.width (ACE_Log_Record::MAXLOGMSGLEN);
```

```
33   for (;;) {
34     std::string user_input;
35     getline (cin, user_input, '\n');
36
37     if (!cin || cin.eof ()) break;
38
39     ACE_Time_Value now (ACE_OS::gettimeofday ());
40     ACE_Log_Record log_record (LM_INFO, now,
41                                ACE_OS::getpid ());
42     log_record.msg_data (user_input.c_str ());
43
44     if (logging_client.send (log_record) == -1)
45       ACE_ERROR_RETURN ((LM_ERROR,
46                          "%p\n", "logging_client.send()"), 1);
47   }
48
49   return 0; // Logging_Client destructor closes TCP
               // connection.
50 }
```

第3~19行 如果“TCP端口”和“主机名”被作为命令行参数传递到程序中，则可以用它们来初始化网络节点，并让日志服务器在这个节点上被动侦听，以接受连接。否则，使用缺省设置。

第21~29行 利用已经初始化了的ACE_INET_Addr，尝试连接日志服务器。连接是通过一个ACE_SOCK_Connector进行的。如果连接成功，logging_client对象的socket就会被连接到服务器上。

第31~47行 给出客户事件主循环。这个循环连续从标准输入设备读取缓冲数据，并将其转发到日志服务器，直至最后读入EOF。ACE_LOG_Record类中包含“保存记录时所需”的信息，还包含“设置和读取日志记录数据”的方法。运用前面定义的Logging_Client::send()方法，可以简化ACE_Log_Record的整编（marshaling）和传输。

这个客户日志程序是ACE实际网络日志服务程序的简化版本。实际的ACE日志服务设施基于ACE_Log_Msg类。副栏10介绍的宏使用了ACE_Log_Msg类（特别是它的ACE_Log_Msg::log()方法）。在消除了偶发复杂性之后，日志客户程序的代码相对简单。因而，在继续开发本书日志服务程序的过程中，我们会将焦点集中在服务器端；因为后者向我们展示了最引人注目的多路分离、并发、同步等课题。

4.6 小结

在本章中，我们运用 ACE Socket wrapper facade、ACE“消息缓冲”和“整（解）编”类，为网络日志服务程序提供了一个初步的“客户/服务器”实现。在本书中，我们还会继续使用这些 ACE 类，以保持我们这个网络日志服务程序的可移植性、简明性和高效性。作为 ACE 网络日志服务组件的一部分，我们在 ACE 中提供了一个完整的日志服务程序——在第 0.4.4 节曾就此作过介绍。

第 2 篇

并发式面向对象网络编程

第 5 章

并发设计空间

本章提要

在开发具有“可伸缩性”和“健壮性”的网络应用程序，特别是网络服务器时，并发性（concurrency）显得十分重要。本章提供了“并发设计空间（concurrency design dimension）”的领域分析（domain analysis），“并发设计空间”涉及正确使用进程、线程和同步装置（synchronizer）的策略（policy）和机制。本章讨论以下设计空间：

- 循环、并发及反应式服务器；
- 进程与线程；
- 进程/线程创建策略；
- 用户、核心及混合线程模型；
- “分时”及“实时”调度级别；
- “基于任务”与“基于消息”的体系。

5.1 循环、并发及反应式服务器

服务器可以分为循环式（iterative）、并发式（concurrent）和反应式（reactive）等几大类。在这一设计空间中，需要权衡的地方主要在于：是要简化编程；还是要提高程序的可伸缩性，使之日后能够增强“提供服务（service-offering）”的能力和“主机承担负荷”的能力。

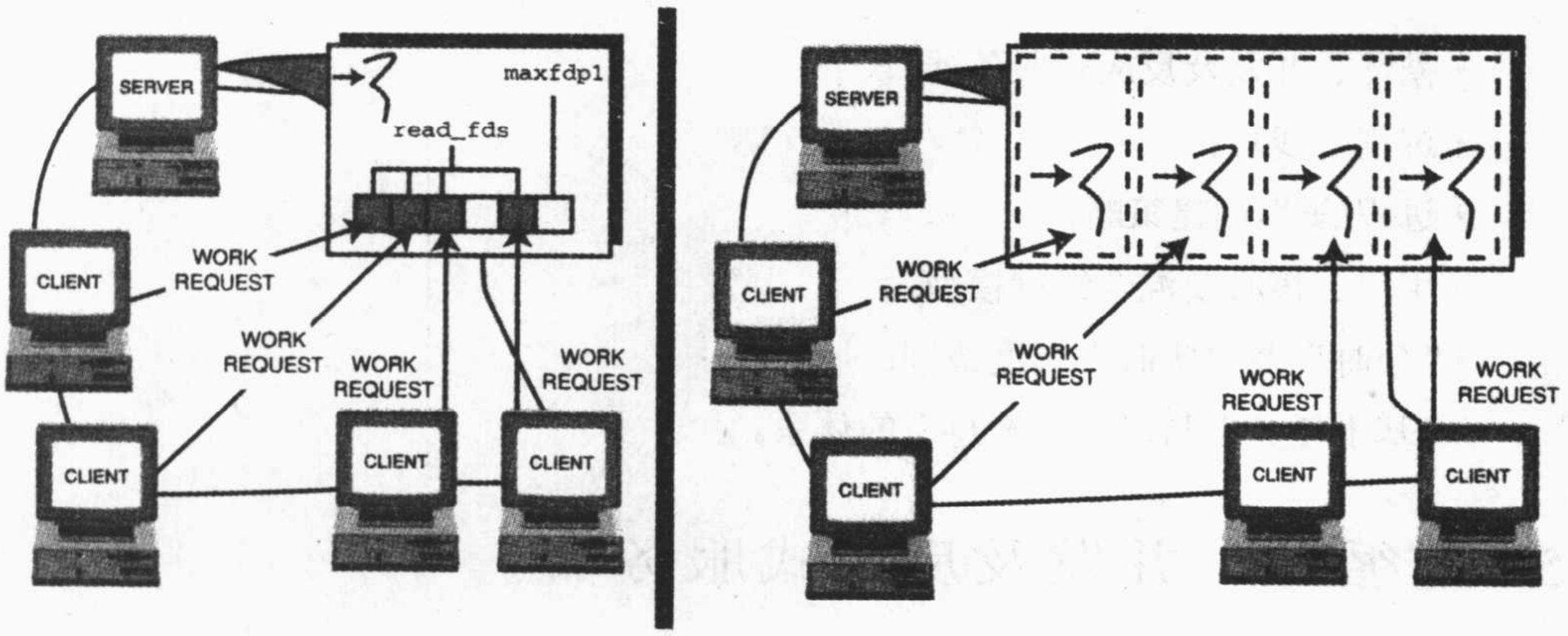
循环式服务器在处理后续请求之前，会完整地处理每一个客户请求。因此，在处理一个请求时，循环式服务器要么将其他请求排成队列，要么忽略它们。循环式服务器最适合以下两种服务：

• **短期服务**，例如，执行时间短、标准的 Internet ECHO 和 DAYTIME 服务；

• **不经常运行的服务**，例如，在平台负载较轻的夜间才运行的“远程文件系统备份服务”。

开发“循环式服务器”相对来说较直接。图 5.1(1)告诉我们：在内部，循环式服务器常常在“单进程”地址空间中执行服务请求，如以下伪代码所示：

```
void iterative_server ()
{
  initialize listener endpoint(s)
  for (each new client request) {
    retrieve next request from an input source
    perform requested service
    if (response required) send response to client
  }
}
```



(1) ITERATIVE/REACTIVE SERVER (2) CONCURRENT SERVER

图 5.1：循环式/反应式服务器与并发式服务器

由于这样一种循环式结构，每一个请求处理都会在一个相对粗糙的层次上被串行化——例如，在应用程序和 OS 同步事件多路分离程序（如 select() 或 WaitForMultipleObjects()）之间的接口上。但是，这种粗糙层次上的并发性不会充分利用主机平台上的某些处理资源（例如，多 CPU）和 OS 特性（例如，并行 DMA 和 I/O 设备之间的传输）。

循环式服务器的另一个特点是：当客户因为等待服务器处理请求而被阻塞时，会阻止客户程序继续往下执行。如果服务器端延迟过久，则应用程序和中间件层中用于“重新传输”的“超时”计算会变得很复杂，从而引发严重的网络阻塞。根据客户和服务器交换请求时使用的协议类型，服务器还有可能接收到重复请求。

并发式服务器同时处理多个客户请求，如图 5.1(2)所示。根据 OS 和硬件平台不同，并发式服务器在执行它的服务时，要么使用多线程，要么使用多进程。如果是“单服务（single-service）”服务器，则同一服务的多个副本可以同时运行。如果是“多服务（multiservice）”服务器，则不同服务的多个副本也可以同时运行。

并发式服务器非常适合“I/O 操作频繁（I/O-bound）”的服务和（或）“执行时间会变化”的长周期服务。和循环式服务器不同，并发式服务器可以使用更精细的同步技术，能够在“应用程序定义的层次”将“请求”串行化。这种设计需要使用同步机制，如信号量（semaphore）或互斥（mutex）*锁（lock）*[EKB+92]，以保证同时运行的进程和线程之间的稳固合作和数据共享。我们将在第 6 章探讨这些机制，并在第 10 章演示它们的应用示例。

我们将在第 5.2 节看到，并发式服务器可以通过各种方式构成，例如，通过使用多进程或多线程。并发式服务器的一个常见设计是*“一个请求一个线程（thread-per-request）”*：为了并发处理客户请求，针对每一个客户请求，主线程都会单独创建一个工作者线程。

```
void master_thread ()
{
  initialize listener endpoint(s)

  for (each new client request) {
    receive the request
    spawn new worker thread and pass request to this thread
  }
}
```

主线程继续侦听新的请求，工作者线程则处理客户请求，如下所示：

```
void worker_thread ()
{
  perform requested service
```

```
  if (response required) send response to client
  terminate thread
}
```

可以直接修改这个“thread-per-request”模型，使之支持其他“并发式”服务器模型，如*“thread-per-connection（一个连接一个线程）”*：

```
void master_thread ()
{
  initialize listener endpoint(s)

  for (each new client connection) {
    accept connection
    spawn new worker thread and pass connection to this thread
  }
}
```

在这个设计中，主线程继续侦听新的连接，工作者线程则处理来自这个连接的客户请求，如下所示：

```
void worker_thread ()
{
  for (each request on the connection) {
    receive the request
    perform requested service
    if (response required) send response to client
  }
}
```

“thread-per-connection”出色地支持“客户请求”的优先级控制。例如，来自高优先级客户的连接会被关联到高优先级线程上。因而，来自高优先级客户的请求会在低优先级客户请求之前被处理，因为OS可以抢占低优先级线程。

在第5.3节，我们还介绍了另外几个并发服务器模型，例如*线程池（thread pool）*和进程池（process pool）。

反应式服务器几乎是同时处理多个请求——尽管所有处理实际上在一个线程中完成。多线程尚未在OS平台上广泛普及之前，并发处理通常是通过“同步事件多路分离”策略来实现的：多个服务请求由一个“单线程（single-threaded）进程”依次循环处理。例如，标准的X Windows服务器就是以这种方式运作的。

通过“同步事件多路分离”机制，如第 6 章介绍的 select()和 WaitForMultipleObjects()，可以显式执行分时处理，将服务器的注意力分散到每一个请求上，以此方式实现反应式服务器。以下伪代码演示的是，在基于 select()的反应式服务器中，我们使用的一种典型的编程风格：

```
void reactive_server ()
{
  initialize listener endpoint(s)

  // Event loop.
  for (;;) {
    select() on multiple endpoints for client requests
    for (each active client request) {
      receive the request
      perform requested service
      if (response is necessary) send response to client
    }
  }
}
```

这个服务器虽然能够在一段时间内服务多个客户，但从服务器的角度来看，它本质上是循环式的。因此，较之利用“OS 提供的完善的多线程支持”，利用这种技术开发的应用程序存在以下局限性。

- **编程的复杂性增加**。某些类型的网络应用程序（例如，“I/O 操作频繁”的服务器）难以用“反应式服务器”模型来编程。例如，开发者必须显式创建事件循环线程，还要负责手工保存和恢复环境信息。所以，为了让客户认为它们的请求是在被“并发”处理，而不是“循环”处理，每一个请求就都必须在一段相对较短的时间内执行。同样，长时间的操作（例如，下载大型文件）得被显式编写为一个“有限状态机”，在对其他对象的事件作出反应的过程中，这个状态机会记录这个对象的处理过程。随着状态数量的增加，设计会变得很笨重。

- **可靠性和性能降低**。如果一个操作失败（例如，一个服务进入死循环，或陷入*死锁（deadlock）*而被无限期挂起），则整个服务器进程将会挂起（hang）。而且，即使整个进程没有失败，但只要有一个服务调用了系统函数，或产生了分页错误，OS 就会阻塞整个进程，从而降低整个服务器进程的性能。另一方面，如果只使用“非阻塞”方法，那将很难运用到 DMA 之类的高级技术，从而无法利用*数据和指令缓存（data and*

instruction cache）的“引用局部性（locality of reference）”来提高性能。如第 6 章所述，OS“多线程”机制可以克服这些性能上的局限性；因为对那些运行在单独线程上的独立服务来说，多线程的出现能使它们的“抢占式”和“并行”执行自动化。若想不通过完整的“并发式服务器”方案来解决这些问题，一个方法是使用“异步 I/O”，如副栏 11 所示。

副栏 11：异步 I/O 和前摄式服务器

另一个在单线程服务器中处理多 I/O 流的机制是*异步 I/O*。这一机制允许服务器通过一个或多个 I/O 句柄来启动 I/O 请求，不用因为要等待请求完成而被阻塞。而当请求结束时，OS 会通知调用者，服务器则可以在已经结束的 I/O 句柄上继续处理。以下 OS 平台提供了异步 I/O 机制：

- 通过“重叠（overlapped）I/O”和“I/O 完成端口（completion port）”支持异步 I/O 的 Win32 平台 [Ric97, Sol98]。
- 一些实现了 aio_*()“异步 I/O”函数、符合 POSIX 标准的平台 [POS95, Gal95]。

但是，和“多线程”及“同步事件多路分离”机制相比，“异步 I/O”的实现不像前者那样具有可移植性；所以，在本书中不再对其作进一步讨论。

在 [SH]一书中，我们会讲述“使用了 Proactor 模式” [SSRB00]的 ACE Proactor 框架；还会讨论“异步 I/O”。在使用了 Proactor 模式的“事件驱动型（event-driven）”应用程序中，在受到“异步操作完成”事件的触发时，程序可以高效地多路分离（demultiplex）和分发（dispatch）服务请求，从而获得并发性所带来的性能提高，同时无需承担并发处理上的任何工作。在 Win32 和支持 aio_*()同步 I/O 函数的 POSIX 系列平台上，都可以使用 ACE Proactor 框架（framework）。

日志服务程序 => 为简单起见，在第 4 章的网络日志服务程序的初始版本中，使用的是“循环式服务器”设计。在后续章节中，将对这个日志服务器的功能和伸缩性

进行扩充：在第 7 章，扩充后的服务器展现的是“反应式”服务器风格；在第 8 章，展现的是使用了“多进程”的“并发式”服务器风格；在第 9 章，演示了几个“使用多线程”的“并发式”服务器设计。

5.2 进程与线程

第 5.1 节介绍了“并发式服务器”的优点，而“并发式服务器”可以通过多个*进程（process）*或*线程（thread）*来实现。这一设计空间涉及的主要取舍包括：健壮性、效率和可伸缩性。

多进程。进程是一种 OS 实体（entity），用以提供“执行程序指令”的环境（context）。每一个进程都管理某些资源，如虚拟内存、I/O 句柄、信号处理器等，并借助*内存管理单元（MMU，memory management unit）*硬件，防止自身遭受其他 OS 进程破坏。在 UNIX 中，进程是通过 fork()创建的，Win32 进程则通过 CreateProcess()创建；这些进程并发执行，但位于“和调用者不同”的地址空间。这些机制在第 8 章作了详尽介绍。

在上一代操作系统（如 BSD UNIX [MBKQ96]）中，进程只能控制一个线程。这种“单线程”的进程模型可以促进程序的健壮性；因为如果没有程序员明确介入，进程之间不会相互干扰。例如，进程只能通过“共享内存”或“本地 IPC”机制相互合作，如图 5.2(1)所示。

图 5.2：多进程与多线程

但是，通过“单线程”进程难以开发某些类型的应用程序，特别是高性能服务器或实时服务器。需要相互通信或需要响应“管理请求”的服务器必须使用某种形式的 IPC，这增加了它们的复杂性。此外，如果使用多进程，将难以对“调度”和“进程优

先级”实施高效率、高精度的控制。

多线程。为了减轻上面提及的进程存在的问题，如今，大多数 OS 平台都能在一个进程中支持多个线程。一个线程是一组单独的指令序列，执行在进程的保护范围之内，如图 5.2(2)所示。除了一个指令指针（instruction pointer）之外，线程还要管理某些资源，例如，保存“函数活动记录”的运行时栈（stack）、一组寄存器、信号屏蔽设备、优先级以及和线程有关的数据等等。如果有多个 CPU，多线程服务器中的服务则可以并行（parallel）执行[EKB+92]。在很多 UNIX 版本中，线程由 pthread_create()创建；在 Win32 中，线程由 CreateThread()创建。

在并发式网络应用程序的实现中，如果多个操作在各自的“线程”中执行，而不是在各自的“进程”中执行，以下“并发”开销就能得以降低：

- **线程创建与环境切换**。和进程相比，线程维护的状态信息要少；因而，和对应的进程的生命活动相比，“线程创建”和“环境切换”的开销要少。例如，在一个进程中切换线程时，进程范围内的资源（如虚拟地址映射和缓存）不需要改变。

- **同步**。在调度、执行一个应用程序线程时，核心模式和用户模式之间的切换可能不必要。而且，“进程内”同步的开销通常没有“进程间”同步昂贵，因为“被同步对象”位于进程内部，因而不需要 OS 内核的干预。相反，“进程间”线程同步一般需要牵涉到 OS 内核。

- **数据复制**。线程可以通过“进程局部内存（process-local memory）”共享信息，这具有以下优点：

1. 较之通过“共享内存”或“本地 IPC”机制进行进程间通信，通过“进程局部内存”通信往往更有效率；因为数据不需要通过内核来复制。

2. 在“进程局部内存”中，使用 C++对象将更为容易，因为类的虚函数表不会存在“内存布局”问题（见副栏 3）。例如，有这样一些相互协作的数据库服务，它们都引用了“进程局部内存”中的公共数据结构，那么，在实现这些数据库服务时，使用“多线程”比使用“多进程”更简单、更高效。

有了这些优化，多线程可以显著地提高应用程序的性能。例如，多线程会使“I/O操作频繁”的应用程序受益，因为“计算密集（compute-intensive）”型服务可以同磁盘和网络操作重叠执行。但是，仅仅因为 OS 平台支持线程，并不就意味着所有应用程序都应该使用多线程。特别是，使用“多线程”实现“并发式”应用程序，也存在以下局限性：

• **性能损失**。一个常见的误解是：线程天生就是用来提高程序性能的。其实在很多时候，线程并不能提高性能，原因如下：

1. 在单处理器上，“计算任务繁重”的应用程序不会从多线程受益，因为计算和通信无法并行执行。

2. 高精度的锁定策略会带来高同步开销，这会阻碍应用程序充分挖掘和利用“并行处理”[SS95]的好处。

• **健壮性降低**。为减少“环境切换”和“同步”开销，线程之间接收到的“MMU保护”很少或没有。通过“单进程地址空间”中的线程执行所有任务，会降低应用程序的健壮性。这是因为：

1. 在同一进程地址空间中，各个线程之间缺乏很好的保护。因此，只要进程中有一个服务出错，全局数据就有可能被破坏；而这个数据可能会被“运行在这个进程中的其他线程上”的服务所共享。这就会造成不正确的执行结果，或使整个进程崩溃，或使应用程序无限期挂起。

2. 在一个线程中调用某些 OS 函数，会对整个进程带来不利的副作用（side effect）。例如，UNIX 的 exit()和 Win32 的 ExitProcess()函数具有的副作用是：它们会终止一个进程中的所有线程。

• **缺乏高精度的访问控制**。在大多数操作系统上，进程是“访问控制（access control）”的基本粒度（granularity）。因此，多线程存在另一个限制：在同一进程中，所有线程都共享相同的用户 ID，共享相同的“对文件和其他被保护资源”的访问权。

为了防止无意或有意地访问“未经授权的资源”，那些将“安全机制”建立在“进程所有权”上的网络服务，例如 TELNET，通常运行在单独的进程上。

日志服务程序 => 在我们的网络日志服务程序中，日志服务器的“并发”版本可以通过各种方式实现。在第 8 章和第 9 章，我们分别通过“多进程”和“多线程”来实现并发日志服务器。

5.3 进程/线程创建策略

创建进程和线程有多种策略。在不同的环境下，可以通过不同的策略来优化“并发式服务器”的性能，从而，使开发者能够调整服务器的并发级别（concurrency level），使之适应“客户的需求”和“现有的 OS 处理资源”。如下文所述，这些不同的策略会增加启动（startup）时的开销，或带来更多的资源消耗。

急式创建（Eager spawning）策略。采用这一策略，在服务器创建期间，会有一个或多个 OS 进程/线程预先创建。这些“热启动（warm-started）”的执行资源（execution resource）形成了一个“池（pool）”：它改善了响应时间，但也增加了“请求被处理之前启动服务”的开销。这个“池”能以静态和（或）动态方式扩充或收缩，具体取决于各种因素，例如，可用 CPU 的数量、机器的当前负载、客户请求序列的长度。图 5.3(1)描述的是线程的“急式创建”策略。这个设计使用了 Half-Sync/Half-Async（半同步/半异步）模式[SSRB00]，将“来自 I/O 层”的请求向上提供给线程池中的工作者线程。

另一个“急式创建”策略是通过 *Leader/Followers（领导者/跟随者）*模式[SSRB00]来管理线程池，如图 5.3(2)所示。这个模式定义了一个高效的并发模型（concurrency model）；采用这一模型，多个线程轮流共享一组事件源（event source），以对这些事件源上发生的服务请求进行检测、多路分离、分发和处理。如果对在处理线程池中的请求没有“同步”或“排序”限制，则用 Leader/Followers 模式代替 Half-Sync/Half-Async 模式可以提高性能。

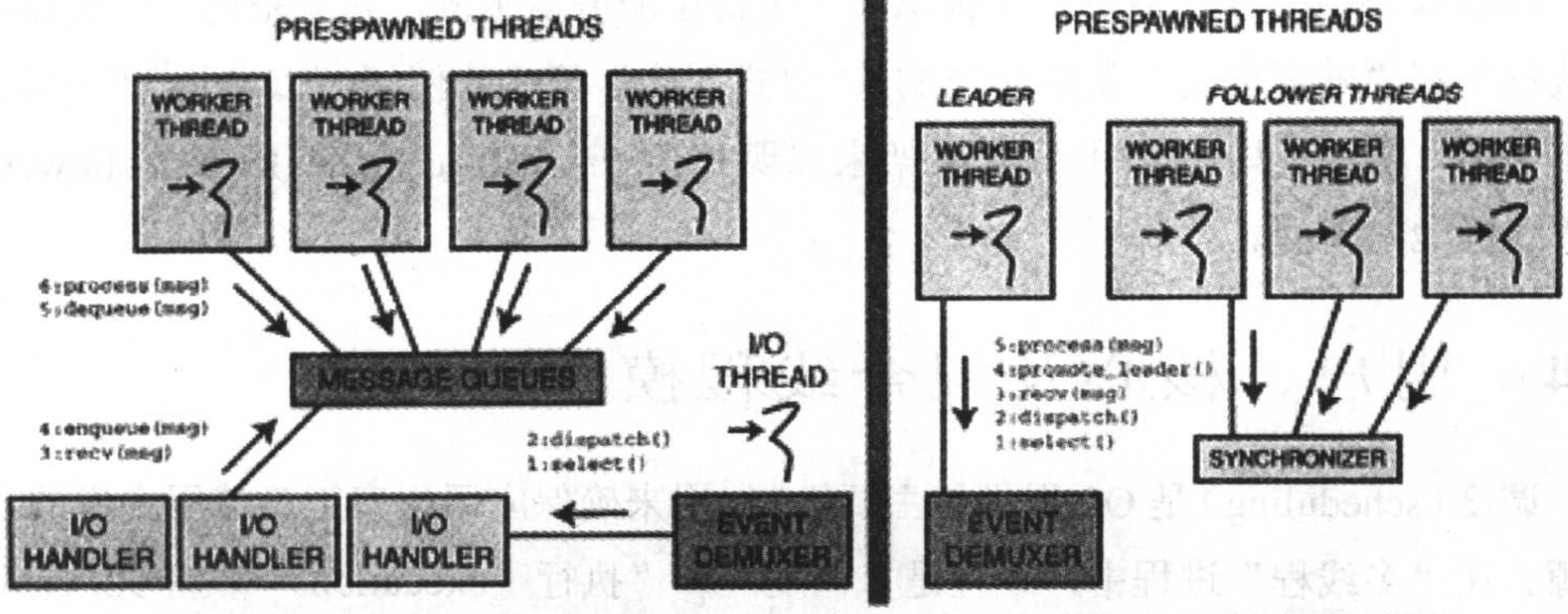


图 5.3：线程池“急式创建”策略

随需创建（on-demand spawning）策略是在“客户连接”和（或）“数据请求”到来时创建新进程或线程，就像第 5.1 节介绍的“一个请求一个线程”和“一个连接一个线程”模型那样。图 5.4 演示了线程的“随需创建”策略。“随需创建”策略的主要优点是资源消耗降低。它们的缺点是：由于创建进程/线程和启动服务时会带来开销，因而，这些策略会降低“重负载”服务器的性能和实时系统的确定性（determinism）。

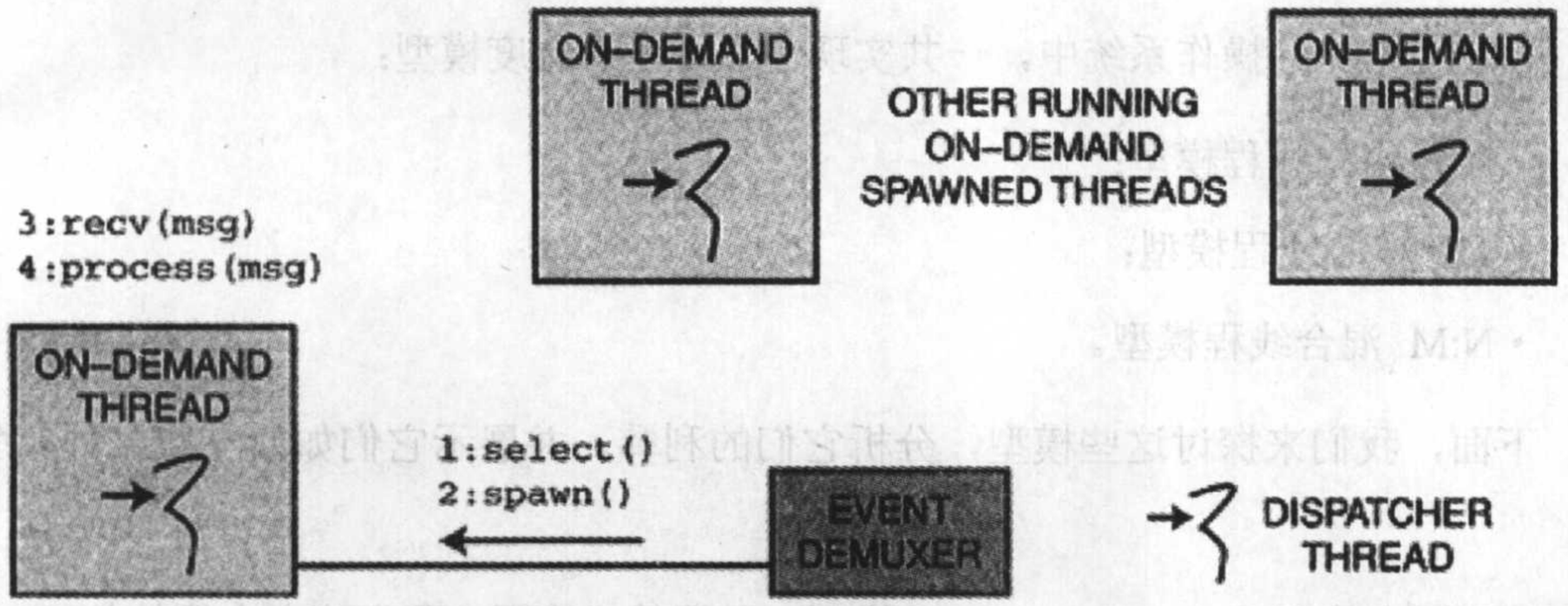


图 5.4：“一个请求一个线程”式的“随需创建”策略

日志服务程序 => 第 9.2 节提供了一个日志服务器示例，它就是用“一个连接一个线程”和“随需创建”策略实现的。“急式创建”策略的例子可以参见[SH]一书。在那本书中，演示了如何运用 ACE 框架来实现 half-sync/half-async 和 leader/followers 式的“线程池”模型。

5.4 用户、核心及混合线程模型

调度（scheduling）是 OS 提供的主要机制，用来确保应用程序正确使用主机的 CPU 资源。在“多线程”进程中，线程是“调度”和“执行（execution）”的单元（unit）。现代 OS 平台提供了各种模型，用来调度应用程序创建的线程。这些模型之间的主要不同是：在竞争系统资源（特别是 CPU 时间）时，线程所处的*竞争范围（contention scope）*不一样。有以下两个不同的竞争范围。

- *进程竞争范围（process contention scope）*：各个线程在同一进程中竞争“被调度的 CPU 时间”（但不直接和其他进程中的线程竞争）。

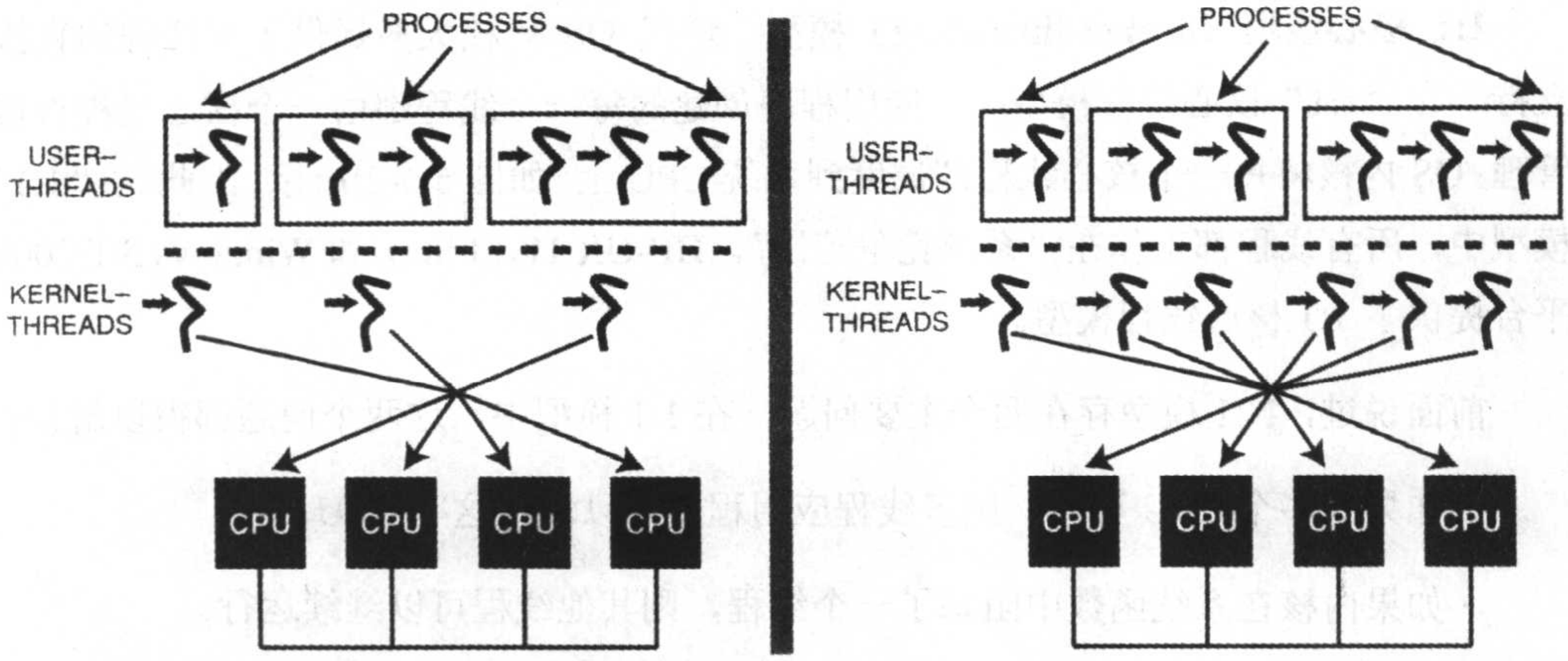
- *系统竞争范围（system contention scope）*：线程直接和“系统范围”内的其他线程竞争，无论它们和什么进程关联。

在当今常见的操作系统中，一共实现了三种线程调度模型：

- N:1 用户线程模型；
- 1:1 核心线程模型；
- N:M 混合线程模型。

下面，我们来探讨这些模型，分析它们的利弊，并展示它们如何支持各种竞争范围。

N:1 用户线程（user-threading）模型。早期的“线程实现”建立在原始的 OS“进程控制”机制之上，并由用户空间中的程序库来管理。因此，OS 内核完全不知道线程的信息。“调度进程和程序库”的内核在一个进程中管理 n 个线程，如图 5.5(1)所示。因此，这个模型被称为“N:1”用户线程模型，这些线程则被称为“用户空间线程”，或简称为“用户线程”。在 N:1 线程模型中，所有线程都工作在“进程竞争范围”。



(1) N:1 USER THREADING MODEL (2) 1:1 KERNEL THREADING MODEL

图 5.5：N:1 与 1:1 线程模型

HP-UX 10.20 和 SunOS 4.x 这两个平台提供了 N:1 用户线程模型。

在 N:1 线程模型中，内核不干涉线程的任何生命活动，也不干涉同一进程中的环境切换。因此，线程的创建、删除和环境切换都很高效。具有讽刺意味的是，N:1 模型存在两个主要问题，而这两个问题也同样源于内核对线程的忽视：

1. 无论主机有多少 CPU，一个进程只会被调度到一个 CPU 上。进程中所有的线程都会竞争那个 CPU，内核进行进程调度时分配的任何时间片（time-slice）也被所有线程共享。

2. 如果某个线程执行了一个“阻塞式”操作（例如，通过 read()或 write()读/写一个文件），那么，那个进程中的所有线程都会阻塞，直至那个操作结束。很多 N:1 实现，如最著名的 DCE Threads，都在 OS 系统函数之外提供了包装器（wrapper），以求消除这种限制。但是，这些包装器不是完全透明的，它们会限制你的程序行为。因此，你必须小心它们，以避免对你的应用程序带来负面影响。

1:1 核心线程（kernel-threading）模型。现代 OS 内核大多提供了对线程的直接支持。在“1:1”核心线程模型中，应用程序创建的每一个线程都由一个核心线程直接管理。OS 内核将每一个核心线程都调度到系统 CPU 上，如图 5.5(2)所示。因此，在“1:1”模型中，所有线程都工作在“系统竞争范围”。HP-UX 11、Linux 和 Windows NT/2000 平台提供了 1:1 核心线程模型。

前面说过，N:1 模型存在两个主要问题，在 1:1 模型中，这两个问题都得以解决：

- 如果有多个 CPU 可用，则多线程应用程序可以利用这些 CPU。
- 如果内核在系统函数中阻塞了一个线程，则其他线程可以继续运行。

但是，OS 内核涉及线程的创建和调度；所以，较之使用 N:1 模型，在使用 1:1 模型时，线程的生命周期操作（life-cycle operations）更昂贵（但一般来说，还是要比“进程”生命周期操作低廉）。

N:M 混合线程（hybrid-threading）模型。一些操作系统，如 Solaris [EKB+92]，提供了 N:1 和 1:1 模型的组合形式，即所谓“N:M”混合线程模型。这个模型支持用户线程和核心线程的混合形式，如图 5.6 所示。当应用程序创建线程时，它可以指明这个线程应该工作在哪一个竞争范围（在 Solaris 中，缺省选择是“进程竞争范围”）。OS 线程库创建一个用户空间（user-space）线程；但只是在需要的时候，或是在应用程序明确请求“系统竞争空间”的时候，它才会创建一个核心线程。就像在 1:1 模型中那样，OS 内核将核心线程调度到 CPU 上。但也像在 N:1 模型中那样，OS 线程库将“用户空间线程”调度到所谓“轻量级进程（LWP，lightweight process）”上，后者一一映射到核心线程。

机敏的读者会注意到，在 N:M 模型中，有一个问题会重新浮出水面；也就是说，多个“用户空间线程”中只要有一个线程调用了阻塞式系统函数，其他线程就都会阻塞。如果 OS 内核阻塞了一个 LWP，则被线程库调度到这个 LWP 之上的所有用户线程也都会阻塞——虽然在这个进程中，被调度到其他 LWP 上的线程可以继续运行。Solaris 内核通过二分法来解决这个问题；这种二分法基于“调度者激活（scheduler activations）”概念 [ABLL92]，如下所述：

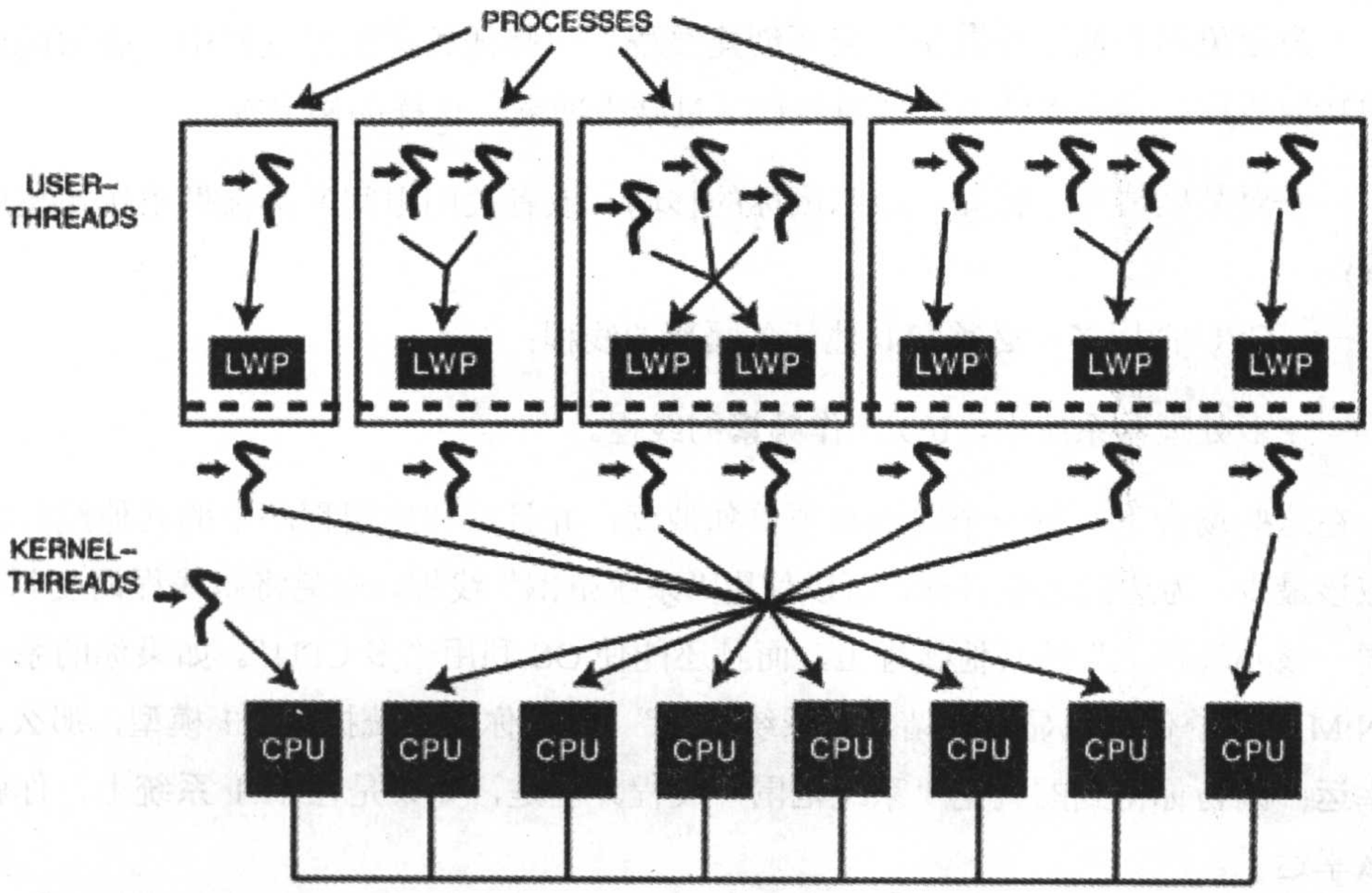


图 5.6：N:M 混合线程模型

1. OS 线程库维护着一个“LWP 池”，用来运行所有“进程范围”用户线程。在需要的时候，它可以将这些用户线程重新调度到池中的 LWP 上。池的大小可以通过 Pthreads 的 pthread_setconcurrency()函数来调整。

2. 如果 OS 内核发现进程中所有核心线程都被阻塞，则会向受影响的进程发送 SIGWAITING 信号。线程库捕获到信号后，会启动一个新的 LWP。然后，它将一个“进程范围”线程重新调度到这个新的 LWP 上，使得应用程序可以继续运行。

关于线程如何映射到系统资源，如何分配系统资源，并不是所有的 OS 平台都允许你去干预的。你应该知道你的平台允许你做什么，它们的工作情况如何；只有这样，你所做的那些工作才会达到最大的效用。关于 OS 并发机制的详细论述，请参见[Lew95, But97, Ric97, Sol98, Sch94]。和使用任何一种威力强大、功能齐全的工具一样，滥用线程也有可能给你带来伤害。那么，如果有机会在“竞争空间”之间作选择，你应该选择哪一个呢？答案取决于：你创建线程的原因最符合下面哪一条，以及在你的程序中，它必须和其他线程保持怎样的独立性。

• 为避免和其他任务发生冲突而创建线程。一些任务在执行过程中，必须同进程中的其他线程，甚至系统中的其他线程尽量减少冲突。这样的例子有：

——对某些情况（譬如，跟踪鼠标的移动，或者关闭电闸）必须做出快速反应的线程；

——CPU 占用多，必须和其他任务隔离的线程；

——多处理器系统中，I/O 操作频繁的线程。

在这些场合下，每一个线程必须单独调度，并且，和应用程序中的其他线程的竞争应该最少。为达到这个目标，可以使用“系统范围”线程，避免将新线程调度到“位于同一核心线程上”的其他线程上，而且还能使 OS 利用“多 CPU”。如果你的系统支持 N:M 模型，你可以显式地请求“系统竞争”。如果你的系统提供 1:1 模型，那么，你很幸运；因为你得到的就是“系统范围”线程。但是，如果是在 N:1 系统上，你就不那么幸运了。

• 为简化应用程序设计而创建线程。通常有这样一个明智的做法：将可用 OS 资源保留下来，到需要的时候再使用。你创建线程的主要动机可能是简化应用程序的设计，做法是：将应用程序分解为各个独立的逻辑任务；例如，将其分解为各个“处理阶段（processing stage）”，先对数据进行操作，然后将其传递到下一个阶段。这种场合下，如果可以使用“进程范围”线程，就不必承担“使用核心线程”的开销；因为“进程范围”线程总是用在 N:1 系统中，或是在 N:M 模型中经过请求得到。使用“进程范围”线程会带来以下后果：

——避免线程创建、调度及同步过程中内核的参与，但还是分离了应用程序中的关系（concern）。

——如果“进程范围”线程的等待状态是因为同步（synchronization）造成的，例如，是因为等待一个互斥锁（mutex）而不是阻塞式系统函数，那么，整个进程或核心线程就不会被无端阻塞。

“多线程”初看上去似乎很吓人，但是，一旦掌握了同步模式 [SSRB00]和 OS 并发机制，就可以帮助你简化应用程序的设计。例如，可以通过一个或多个线程来执行

同步（synchronous）I/O；较之使用“同步”或“异步”事件处理模式，如 Reactor 或 Proactor，使用多线程的设计更为直接。在第 6 章，将讨论 OS 并发机制；在第 9、10 章，将讨论封装了这些机制的 ACE“线程”及“同步”wrapper facade。

日志服务程序 => 在本书余下章节的日志服务器实现中，我们演示了各种不同的 ACE“并发式”wrapper facade。在这些例子中，当使用线程的目的是为了执行 I/O 操作的时候，例如，从客户获取日志记录的时候，将使用“系统竞争范围”线程，即 1:1 核心线程。这种设计将保证：在调用阻塞式函数接收 socket 数据时，其他任何线程或整个进程都不会被无端阻塞。

5.5 分时及实时调度级别

除了第 5.4 节介绍的“竞争范围（contention scope）”和“线程模型（threading model）”之外，OS 平台一般还会定义一些策略和优先级，用以进一步影响“调度”行为[But97]。有了这些能力，现代操作系统可以运行各种应用程序，满足通用和实时调度的需要。OS 内核可以为线程分配不同的调度级别（class），并根据一些标准（如它们的优先级，或它们的资源使用情况）来调度这些线程。在下文讲述的各个“调度级别”策略中，我们用所谓“公平性（fairness）”换取“可预见性”和“可控制性”的提高。

分时调度级别（Time-shared scheduling class）。通用型的 OS *调度器（scheduler）*着眼于传统的分时、交互式环境。这些操作系统上的调度器往往具有以下特征：

- **基于优先级（priority-based）**——具有最高优先级的可运行（runnable）线程是下一个将被调度执行的线程。

- **公平（fair）**——在分时调度器中，根据线程使用 CPU 的情况，线程的优先级会发生变化。例如，一个“长期运行”且“计算任务繁重”的线程如果占用了过多的 CPU 时间，分时调度器会逐渐降低它的优先级，直至降为一个较低的极限值。

• **抢占（preemptive）**——如果一个低优先级线程正在执行，此时，一个较高优先级的线程转为可运行状态，那么，调度器会抢占低优先级线程，让高优先级线程执行。

• **时间片（time-sliced）**——用于循环执行“具有相同优先级”的线程。采用这一技术，每一个线程都只运行一个有限的时段（例如 10 毫秒）。如果当前正在运行的线程的时间片耗尽，则调度器会先选择另一个可用线程，并执行环境切换，然后将“被抢占的线程”排到一个队列中。

实时调度级别（Real-time scheduling class）。虽然分时调度器适合传统的网络应用程序，但在有“实时”要求的应用程序中，它们很少能够满足需要。例如，在分时调度器级别中，线程的执行通常没有固定次序，因为随着时间的流逝，调度器会改变线程的优先级。而且，当一个高优先级线程变为“就绪”状态时，分时调度器不会限制“抢占低优先级线程”所需的时间。

因此，实时操作系统（如 VxWorks 或 LynxOS）和一些通用操作系统（如 Solairs 和 Windows NT）提供了“实时调度级别”[Kha92]，以限制“最坏情况（worst-case）”下调度“用户线程”或“核心线程”所需的时间。具有“实时调度级别”的 OS 通常支持以下两种调度策略[Gal95]，或支持其中之一：

• **轮流（round-robin）**——根据一个时间量，指定一个线程被另一个“具有相同优先级”的实时线程抢占之前，能够运行的最长时间。

• **先进先出（FIFO，first-in, first-out）**——高优先级线程能根据需要长时间运行，直到它自愿放弃控制，或被另一个具有更高优先级的实时线程抢占。

如果一个 OS 同时支持“分时”和“实时”调度级别，则“实时”线程总是比任何“分时”线程都具有更高的运行优先级。

长期占用 CPU 的实时程序会占据系统，使其他系统活动全部陷入瘫痪。因此，为了控制这种不良情况，对那些“以超级用户（superuser）特权”运行的程序，大多数通用操作系统都会限制其使用“实时调度级别”。

日志服务程序 => 一般来说，日志服务程序对时间的要求不是很高；所以，在本书使用的例子中，我们简单地使用了缺省的 OS“分时调度”级别。但出于完整性考虑，在第 9.3 节的网络日志服务器示例中，还是向大家展示如何通过 ACE 编写一个“运行在实时调度级别上”的、“可移植”的日志服务器。

5.6 “基于任务”与“基于消息”的体系

并发体系（concurrency architecture）是联系以下要素的纽带：

1. **CPU**——为应用程序代码提供执行环境（execution context）；

2. **“数据”和“控制”消息**——在一个或多个应用程序或网络设备之间发送和接收；

3. **执行任务的服务**——当消息到达、离开时，基于消息执行服务。

在一个网络应用程序中，并发体系是影响“环境切换”、“同步”、“调度”和“数据移动”的成本，从而影响程序性能的几大要素[1]之一。并发体系的规范（canonical）类型有两种：“基于任务”的体系和“基于消息”的体系[SS93]。在这一设计空间，主要的取舍在于编程的简化和程序性能的提升。

基于任务的并发体系根据应用程序中的“服务功能单元（units of service functionality）”来组织多个 CPU。在这一体系中，任务是主动的，任务中处理的消息是被动的，如图 5.7(1)所示。并发性是通过在各个 CPU 中执行服务任务，并在任务/CPU 之间传递数据消息和控制消息而获得的。“基于任务”的并发体系可以通过“生产者/消费者（producer/consumer）”模式（譬如，Pipes 和 Filters [BMR+96]，以及 Active Objects [SSRB00]；说明请参见[SH]）来实现。

基于消息的并发体系根据“从应用程序和网络设备接收到的消息”来组织 CPU。在这一体系中，消息是主动的，任务是被动的，如图 5.7(2) 所示。并发性是借助一个

[1] 影响性能的其他要素包括协议、总线、内存和硬件设备特性。

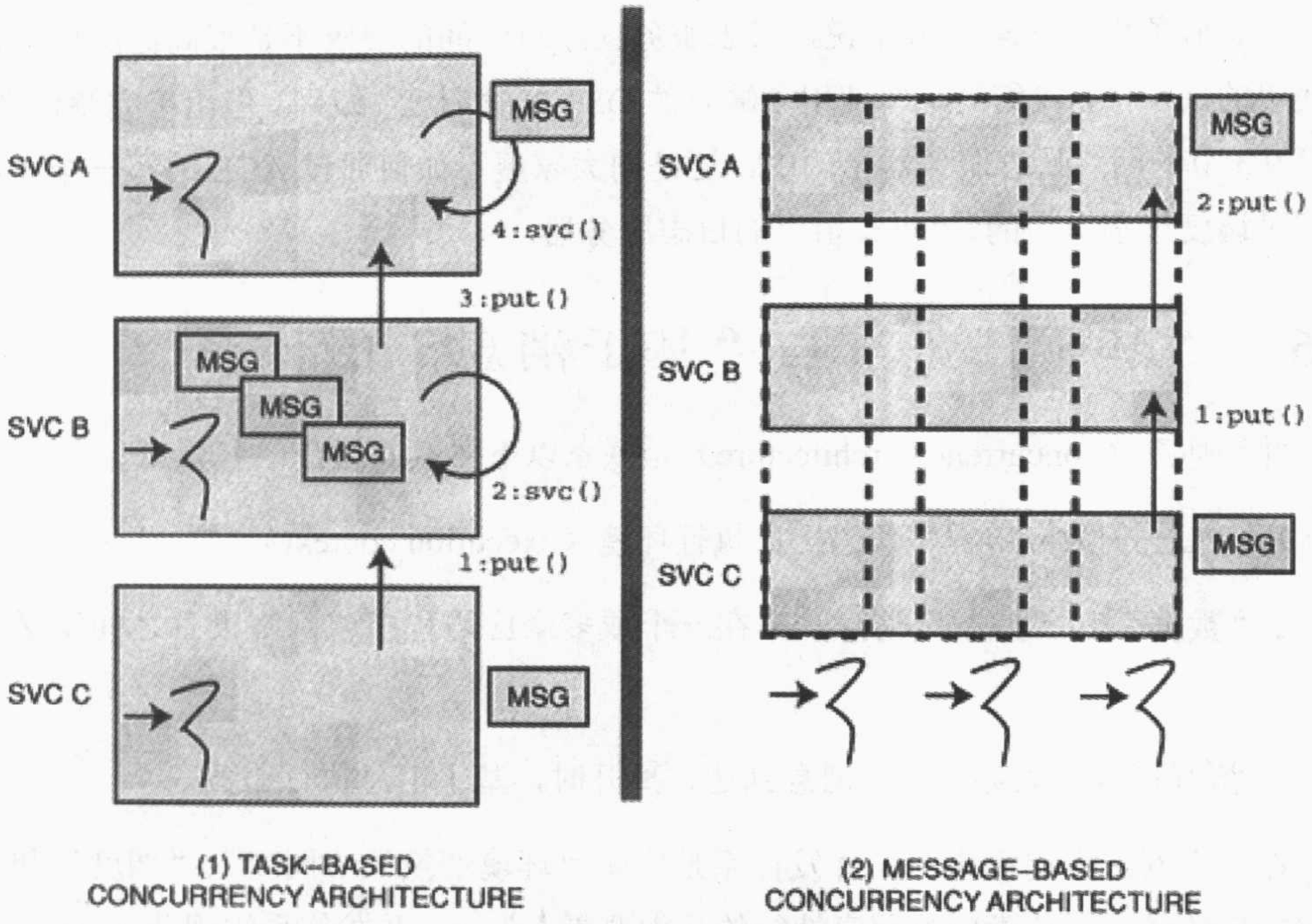


图 5.7：“基于任务”与“基于消息”的并发体系

服务任务栈，通过同时在各个 CPU 上处理多个消息而获得的。“一个请求一个线程”、“一个连接一个线程”及“线程池”模型可用来实现“基于消息”的并发体系。第 9 章介绍了“一个连接一个线程”的使用。关于“线程池”的介绍则需参阅[SH]。

日志服务程序 => 在本书中，我们是通过“基于消息”的并发体系来构造网络日志服务程序的线程设计的，特别是“一个连接一个线程”模型，它们往往比“基于任务”的体系更有效率[HP91, SS95]。但是，“基于任务”的体系一般更易于编程；因为在一个任务中，或在一个层次上，一般不需要同步，因为并发的串行化是在任务访问点上，例如，在协议栈的各层之间。相反，“基于消息”的体系更难实现，因为它需要更复杂的并发控制。

5.7 小结

网络服务器可以将一个或多个进程或线程关联到一个或多个应用服务上。这需要我们在很多设计空间做出设计决策，因为它们影响线程和进程的使用、CPU 资源的使用、调度行为、应用程序的性能等。本章就这些内容丰富的并发设计空间做了一次领域分析，其分析结果可以被应用到网络应用程序的设计之中。在本书余下章节，我们将介绍“根据这一分析结果设计而成”的 ACE wrapper facade。我们针对当今操作系统上的同步事件多路分离、多进程、线程、同步等机制，就封装和使用上的优劣利弊作了专门的探讨。

1 甲
2 乙
3 丙
4 甲
5 乙
6 丙
7 甲
8 乙

第 6 章

操作系统并发机制概述

本章提要

网络应用程序、特别是服务器，经常需要并发地处理请求，以满足“服务质量（quality of service）”的要求。在第 5 章，我们讲述了“和并发选择（concurrency alternative）相关”的一般性设计权衡（design tradeoff）。在本章，我们将对同步事件多路分离（synchronous event demultiplexing）、多进程、多线程和同步机制作简要概述；利用这些机制，就可以实现第 5 章介绍过的那些设计。如果通过原始的“C 层次”的并发 API 来开发网络应用程序，将会遇到一些常见的可移植性问题和编程方面的问题，本章也对这些问题进行了讨论。

6.1 同步事件多路分离

“同步事件多路分离器（synchronous event demultiplexer）”是 OS 提供的一个函数，用于在一组事件源（event source）上等待特定事件的发生。当一个或多个事件源被激活时，函数将返回至调用者。于是，调用者可以处理这些“来自多个源”的事件。通常，同步事件多路分离器被用作“反应式服务器”*事件循环（event loop）*的基本构件；它以一种“连续”而且“有序”的方式，对“来自客户的事件”进行检查并作出反应。

大多数操作系统都支持一个或多个“同步事件多路分离”函数，例如：

- poll()函数[Rag93]，出自 System V UNIX；
- WaitForMultipleObjects()函数[Sol98]，由 Win32 提供；
- select()函数[Ste98]，在 I/O 句柄上对事件源进行多路分离[1]。

我们将重点讨论 select()，因为它最常见。在事件驱动型（event-driven）网络应用程序中，可以运用 select()，确定 I/O 操作可以在哪些句柄上同步调用，但不会阻塞那些发起这些调用的应用程序线程。select()函数的 C API 版本大致如下所示：

```
int select (int width,              // Maximum handle plus 1
            fd_set *read_fds,       // Set of "read" handles
            fd_set *write_fds,      // Set of "write" handles
            fd_set *except_fds,     // Set of "exception" handles
            struct timeval *timeout);// Time to wait for events
```

fd_set 是一个结构，表示一组句柄（一个句柄集），用于检查 I/O 事件。三个 fd_set 地址参数分别是 read_fds、write_fds 和 except_fds；select()分别检查它们，看它们的句柄是否处于“读”、“写”活动状态，或处于“异常条件”（例如，*带外数据（out-of-band data）*）。select()函数通过修改它的 fd_set 参数，向调用者通知“活动”和“非活动”句柄的情况：

- 在 fd_set 中，如果一个句柄值是“非活动”的，那个句柄将被忽略；在 select()返回的 fd_set 中，它的值将是“非活动”的。

- 如果 fd_set 中的一个句柄值是“活动”的，则 select()将会在相应的句柄上检查是否有未处理的事件。如果在这个句柄上有一个未处理事件，则在 select()返回的 fd_set 中，相应的值将是“活动”的。如果这个句柄没有未处理事件，则在返回的 fd_set 中，它的值将是“非活动”的。

每一个 OS 平台都会提供一组基本操作，允许应用程序“查询”、“操纵”fd_set。这些操作虽然在某些平台上是宏（macro），在另外一些平台上是函数，但它们具有一组统一的名称和行为：

---

[1] Win32 版本的 select()只对 socket 句柄有效。

函　数	说　明
FD_ZERO()	将 fd_set 中所有句柄的值标记为“非活动”
FD_CLR()	将 fd_set 中的一个句柄标记为“非活动”
FD_SET()	将 fd_set 中的一个句柄标记为“活动”
FD_ISSET()	检测 fd_set 中某一句柄是否“活动”

select()的最后一个参数是一个“指向 struct timeval”的指针；这个结构包含以下两个字段：

```
struct timeval
{
  long tv_sec;  /* seconds */
  long tv_usec; /* microseconds */
};
```

以下三种 timeval 值可以传给 select()，以控制“超时”行为：

值	行　为
NULL timeval 指针	表示 select()应该无限期等待，直至至少有一个 I/O 事件发生，或有一个信号中断了这个调用。
非 NULL timeval 指针，tv_sec 和 tv_usec 字段为 0	表示 select()应该执行“轮询（poll）”；即，检查所有句柄，并立即返回所有结果。
tv_sec 或 tv_usec 字段 > 0 的 timeval	表示 select()应该在这段时间上等待 I/O 事件发生。

6.2 多进程机制

所有现代通用操作系统都提供了相应的机制，用来创建多个进程，并对多进程的执行实施管理。进程是资源管理和保护的基本单位。一些嵌入式操作系统，如 VxWorks 和 pSoS 的很多版本，只支持单地址空间，类似单进程。这种限制通常是合理的，例如，你的移动电话的处理机可能不需要运行多个进程。但是，这些嵌入式系统实际上大多可以利用多线程。关于多线程，我们将在第 6.3 节讨论。

不同的操作系统提供的“进程控制”功能大不相同。但是，在多进程操作系统中，以下几方面的功能是共有的。

进程生存期（process lifetime）操作。如果 OS 或应用程序调用了进程“创建”和“执行”函数，例如，fork()和 exec*()系列函数（POSIX）或 CreateProcess()函数（Win32），进程就会应运而生。这些函数会创建新的地址空间，以运行某一程序。启动进程（initiating ）还可以设置新程序的命令行参数和环境变量，以将命令、参数、文件或目录的位置之类的信息传递给新进程。进程的终止有以下两种方式：

- **自愿（Voluntarily）终止**。例如，程序执行到 main()函数的结尾处；或者，程序调用了一个“进程退出”API，如 ExitProcess()（Win32）或 exit()（POSIX），这个 API 允许将进程的“退出状态”设置为某一特定值。

- **无意（Involuntarily）终止**。例如，通过一个信号（POSIX），或通过一个外部“删除操作”如 TerminateProcess()（Win32）而终止。

进程同步（process synchronization）操作。当进程退出时，大多数操作系统都会保留这个进程的标识信息（identity）和退出状态；这样，父进程可以与之同步，并获得它的退出状态。常见的进程同步函数有：

- POSIX 的 wait()和 waitpid()函数；
- Win32 的 WaitForSingleObject()和 WaitForMultipleObjects()函数。

很多操作系统还会维护“创建者进程”和“被创建进程”固有的“父/子”关系。一个子进程退出时，一些操作系统会根据所维护的这种关系，将退出信息通知给“可能还关注着这个子进程”的父进程。例如，在 POSIX 系统上，当子进程退出时，OS 会发送 SIGCHLD 信号给父进程。

进程属性（process property）操作。大多数操作系统都会提供一些函数，用来获取和设置进程的各种属性。可以在“进程”级设置的属性包括：文件访问的缺省权限、用户识别、资源限制、调度优先级、当前工作目录，等等。一些操作系统还允许锁定进程的虚拟内存区域，以避免分页错误，或允许内存和 I/O 设备之间交换数据。

6.3 多线程机制

线程是进程中的执行单元。现代操作系统大都提供了相应的机制，用来处理线程的生存期、同步，以及其他“和线程有关”的属性，如优先级、线程专有存储空间（thread-specific storage）等。以下简要介绍了这些机制。

线程生存期操作。程序的主线程在一个进程开始执行时隐含创建。其他线程大都通过显式调用“线程创建”函数（如 pthread_create（Pthreads）或 CreateThread()（Win32））创建。这些函数会创建并启动一个新线程，新线程则会执行“调用者指定”的某个“入口点（entry-point）”函数。和进程一样，线程的终止也是通过以下两种方式之一：

- **自愿终止**。例如，线程执行到“入口点”函数的结尾处；或者，调用了一个“线程退出”函数，如 pthread_exit（Pthreads）或 ExitThread()（Win32）,这些函数允许线程将“退出状态”设置为某个值。

- **无意终止**。例如，通过一个信号，或通过一个异步“线程删除”操作（如 pthread_cancel()（Pthreads）或 TerminateThread()（Win32））而终止。

线程同步操作。很多 OS 线程机制允许创建以下两种线程：

- **分离式（detached）**——当“分离式”线程退出时，OS 会自动回收这个线程的“状态（state）”和“退出状态（exit statu）”使用过的空间。

- **连接式（jóinable）**——当“连接式”线程退出时，OS 会保留它的“标识信息”和“退出状态”；这样，另一个线程可以与之同步，并得到它的“退出状态”。

线程同步机制的常见例子有：

- Pthreads 的 pthread_join()函数；

- Win32 的 WaitForSingleObject()和 WaitForMultipleObjects()函数。

一些 OS 平台提供了同步函数，允许线程之间执行挂起（suspend）和恢复（resume）操作。一些平台允许线程向同一进程中的其他线程发送信号。但是，这些操作一般不具可移植性，并且难以正确使用[But97]；因此，在本书中，我们不会对其作进一步讨论——虽然在它们所处的平台上，ACE 的确封装了它们。

线程属性操作。大多数操作系统都提供了相应的函数，用来获取和设置线程的各种属性。例如，在 Pthreads 中，线程的优先级可以通过 pthread_getschedparam()和 pthread_setschedparam()函数来查看和修改；在 Win32 中，则可以使用 GetThreadPriority()和 SetThreadPriority()函数。同样，很多操作系统还允许应用程序去选择更基本的属性，譬如，线程的调度级别。例如，Solaris 支持第 5.5 节讲述的“实时”和“分时”调度级别[Kha92]。

线程专有存储空间。从“空间（scope）”角度来看，线程专有存储空间（TSS，thread-specific storage）类似全局数据；也就是说，它并不为任何特定的函数或对象局部所有。和全局数据不同的是，每一个线程都实际拥有 TSS 数据的一个独立副本。应用程序可以通过 TSS 来维护具有这种特点的数据——很多函数或对象需要访问它们，但它们属于某一线程的私有数据。一个常见的例子是 errno，errno 用来传递系统函数的错误状态。在多线程程序中，每一个线程都各有一个独立的 errno 副本。

每一个 TSS 项（item）都和一个键（key）关联，进程中所有的线程都可以访问这个键。这些键是通过一个工厂函数，譬如 pthread_key_create()（Pthreads）或 TlsAlloc()（Win32）来创建的。任何线程都可以使用这个键，并通过指针来访问这个存储空间的副本。键/指针的关系是通过 TSS 函数来管理的；这样的函数有：Pthreads 中的 pthread_setspecific()和 pthread_getspecific()，或 Win32 中的 TlsGetValue()和 TlsSetValue()。

6.4 同步机制

如第 6.2 节和第 6.3 节所述，OS 平台初步提供了一些同步机制，允许进程和线程等待其他进程和线程结束。对那些并发执行多个独立执行路径、具有较低精度的应用程序来说，这些机制已经够用了。但是，很多并发应用程序需要更精细的同步机制，以允许进程和线程协调它们的执行次序和访问共享资源（例如文件、数据库记录、网络设备、对象成员、共享内存）的次序。为了防止“竞态条件（race condition）”的发生，对这些资源的访问必须保持同步。

当两个或多个并发线程的执行次序造成了意想不到的错误结果时，“竞态条件”就会产生。防止“竞态条件”的一个方法是使用*同步机制（synchronization mechanisms）*，对访问“共享资源”的代码中的关键段（critical section）实施“串行访问”控制。常用的 OS 同步机制有：互斥体（mutex）、“多读取者/单写入者”锁（readers/writer locks）、信号量（semaphores）和条件变量（condition variable）。

为了说明这些机制的必要性，请看看在第 4.4.3 节所定义的 Iterative_Logging_Server::handle_data()方法中，我们添加以下代码：

```
typedef u_long COUNTER;

// File scope global variable.
static COUNTER request_count;

// ...

  virtual int handle_data (ACE_SOCK_Stream *) {
    while (logging_handler_.log_record () != -1)
      // Keep track of number of requests.
      ++request_count;

    ACE_DEBUG ((LM_DEBUG,
                "request_count = %d\n", request_count));
    logging_handler_.close ();
    return 0;
  }
```

handle_data()方法等待客户日志记录的到来。当一个记录到达时，这个记录会被删除，并会接受处理。request_count 变量用来记录“接收到的客户日志记录”的数目。

只要 handle_data()是在“一个进程中的一个线程中”执行，上面的代码就会正常工作。但是，在很多平台上，当多个线程在同一个进程中同时执行 handle_data()时，会造成不正确的结果。问题出在：全局变量 request_count 上存在“竞态条件”，因而，这段代码不是“线程安全（thread-safe）”的。特别是，在具有以下特点的平台上，不同的线程会去递增、打印 requeset_count 变量的“陈旧”版本：

- “自增（auto-increment）”操作中编译进（compile into）了多个汇编语言操作，如 load、*add* 和 *store* 赋值”；
- 整个内存的排序没有通过硬件得以保证。

为保证上述以及类似情况下，程序会得到正确的结果，操作系统提供了一些同步机制。以下是最常见的同步机制。

6.4.1 互斥体（Mutex，Mutual Exclusion）锁

当共享资源被多个线程并发访问时，为了确保这些资源的完整性，我们可以使用互斥体（*mutex*）锁。互斥体可用来串行执行多个线程，这需要在代码中确定*关键段*（*critical section*）——即，一次只能由一个线程执行的代码。“互斥体”语义像双括号那样具有“对称性”；也就是说，如果一个线程拥有互斥体，那么，它还得负责释放这个互斥体。这种简单的语义有助于保证互斥体的高效实现——如通过硬件自旋锁（spin lock）来实现。[2]

常见的互斥体有两种：

- *非递归互斥体（nonrecursive mutex）*——如果当前拥有互斥体的线程在没有首先释放它的情况下，试图再次获得它，就会导致死锁或失败。

- *递归互斥体（recursive mutex）*——拥有互斥体的线程可以多次获得它而不会产生自死锁，只要这个线程最终以相同次数释放这个互斥体即可。

6.4.2 Readers/Writer 锁

*“Readers/writer（多读取者/单写入者）锁”*可以通过以下方式之一访问共享资源：

- 多个线程并发读取资源，但不修改它；

- 一次只有一个线程修改资源，此时其他线程都不能对其进行读/写访问。

“readers/writer 锁”可用来保护“读操作比写操作频繁”的资源，从而提高并发应用程序的性能。在实现“readers/writer 锁”时，要么给“读取者”以优先权，要么给“写入者”以优先权[BA90]。

“readers/writer 锁”和互斥体有某些共性，例如：获得锁的线程也必须释放这个锁。如果一个“写入者”希望获得这个锁，那么，这个线程必须等待其它所有“拥有

---

[2] 虽然自旋锁本身开销低，但是，如果线程必须长时间处于“*忙等待（busy wait）*”状态，以求某一特定条件的发生，此时自旋锁也会消耗大量的 CPU 资源。

这个锁”的线程释放它；然后，这个“写入者”线程单独占有这个锁。但和互斥体不同的是，多个线程可以同时获得一个“readers/writer 锁”执行“读”操作。

6.4.3 信号量锁

从概念上说，*信号量（semaphore）*是可以原子（atomically）递增和递减的非负整数。如果一个线程试图递减一个信号量，但这个信号量的值已经为 0，则线程将会阻塞。另一个线程“发出（post）”这个信号（semaphore），使信号量的值大于 0 之后，被阻塞的线程才会被释放。

信号量维护状态信息，对信号计数值（count）和被阻塞线程的数量进行记录。它们一般是通过*“休止锁（sleep lock）”*来实现的；休止锁用来触发环境切换，以允许其他线程执行。和互斥体不同的是，释放信号量的线程不必是最初获得这个信号量的线程。这使得信号量适用于更广泛的执行环境，如信号处理程序或中断处理程序。

6.4.4 条件变量

和“互斥体锁”、“readers/writer 锁”、“信号量锁”不同，*条件变量（condition variable）*提供了不同风格的同步方式。在前三种机制中，当“占有锁的线程”在关键段中执行代码时，其他线程会等待。与此相反，使用条件变量，线程可以调整和调度自己的处理过程。

例如，某一数据被其他线程共享；条件变量可以使自己处于等待状态，直至“涉及这个数据”的一个条件表达式达到某一状态。当一个“合作线程（cooperating thread）”显示共享数据的状态已经改变时，阻塞在条件变量上的线程会被唤醒；然后，被唤醒的线程重新计算它的条件表达式，如果共享数据达到预期状态，则恢复处理。再复杂的条件表达式也可以通过条件变量来等待；所以，较之前面所说的同步机制，条件变量允许更复杂的调度决策。

条件变量为高级并发模式（如 Active Object 和 Monitor Object [SSRB00]）不仅提供了核心同步机制，还提供了“进程内”通信机制（如同步消息队列）。Pthreads 和（实

现在 Solaris 之上的）UNIX International（UI）线程支持条件变量；但其他平台（如 Win32 和很多实时操作系统）不支持条件变量。

在副栏 12 中，针对上文介绍的 OS 同步机制，就其性能特征作了比较和评估。一般来说，应该尽量使用那些“提供了所需语义”、“最有效率”的同步机制。如果效率很重要，就得针对你的应用程序的目标平台，找到相应的信息。然后，可以通过变换使用各种 ACE“同步”类来调整你的应用程序，以充分利用平台的差异性。而且，通过使用 *Strategized Locking 模式* [SSRB00]，并遵循第 10.1 节介绍的 ACE_LOCK*伪类（pseudo-class）的方法签名（method signature），其代码还可以保持可移植性。

副栏 12：同步机制的性能评估

操作系统提供了一些同步机制，用以满足不同应用程序的需要。虽然根据具体的 OS 实现和硬件的不同，本章讨论的同步机制在性能上各有差异，但以下常识一定要知道：

- 和互斥体相比，条件变量和信号量的开销往往更大，因为它们的实现更为复杂。但是，和那些用户创建的东西相比，原始 OS 机制总是表现得更出色，因为后者可以利用“OS 的内部特性”和“专门针对硬件的优化”。

- 和“readers/writer 锁”相比，互斥体的开销通常更低，因为它们不需要管理多个“等待者集（waiter set）”。但是，多个“读取者”线程可以并行执行；所以，在多处理器上，如果将“readers/writer 锁”用于访问那些“读操作”比“写操作”多的数据，“readers/writer 锁”的伸缩性会表现得更好。

- 和递归互斥体相比，非递归互斥体效率更高。此外，如果程序员忘记释放递归互斥体，就会造成不易察觉的编程错误 [But97]；但如果忘记释放的是非递归互斥体，则系统会立即通过“死锁”来暴露这些问题。

6.5 OS 并发机制的局限性

如果通过以上介绍的原始 OS 并发机制来开发网络应用程序，则会造成“可移植性”和“可靠性”方面的问题。这里通过一个函数来演示这其中的一些问题。这个函数使用了一个 UI 线程互斥体，以解决第 6.4 节中 request_count 存在的“自增”串行化问题：

```
typedef u_long COUNTER;

static COUNTER request_count; // File scope global variable.

static mutex_t m; // Protects request_count (initialized to // zero).
// ...

  virtual int handle_data (ACE_SOCK_Stream *) {
    while (logging_handler_.log_record () != -1) {
      // Keep track of number of requests.
      mutex_lock (&m);   // Acquire lock
      ++request_count;   // Count # of requests
      mutex_unlock (&m); // Release lock
    }

    mutex_lock (&m);
    int count = request_count;
    mutex_unlock (&m);
    ACE_DEBUG ((LM_DEBUG, "request_count = %d\n", count));
    logging_handler_.close ();
    return 0;
  }
```

在上面的代码中，m 是 mutex_t 类型的变量，而且会被自动初始化为 0。在 UI 线程中，任何“被设置为零”的同步变量都会以缺省语义（default semantics）被“隐式”初始化[EKB+92]。例如，mutex_t 变量 m 是一个静态变量，它是在“未锁定”状态被缺省初始化的。因此，在 mutex_lock()函数第一次调用时，它将获得这个锁的拥有权。其他想得到这个锁的线程都必须等待，直至拥有锁 m 的线程将其释放。

以上代码虽然解决了最初的同步问题，但具有如下缺点：

- **强制性（obtrusive）**。这个方案需要修改源代码，以添加互斥体及相关的 C 函数。

在开发大型软件系统时，这种手工修改如果导致“不一致”的情况发生，将会给维护带来困难。

• **容易出错（error-prone）**。handle_data()方法相对简单；但在更复杂的方法中，程序员容易忘记调用 mutex_unlock()。如果忽略了这个调用，则其他想获得这个互斥体的“被阻塞线程”将处于“饥饿（starvation）”状态。而且，因为 mutex_t 是非递归互斥体，因此，如果拥有互斥体 m 的线程试图在释放它之前再次获得它，就会造成死锁。此外，我们没有检查 mutex_lock()的返回值，故不能确保它已调用成功，在产品级程序中，这会造成微妙的错误。

• **不可见的副作用（unseen side-effect）**。程序员还有可能忘记初始化互斥体变量。前面说过，在 UI 线程中，静态变量 mutex_t 会被隐式初始化。但是，对于“动态分配的对象”中的 mutex_t 成员来说，这一点无法保证。而且，其他 OS 线程实现（如 Pthreads 和 Win32 线程）不支持这些“隐式”初始化语义；也就是说，所有同步对象必须被“显式”初始化。

• **不可移植性（non-portable）**。这段代码只对“UI 线程”同步机制有效。所以，如果想移植 handle_data()方法，使之适用于 Pthreads 和 Win32 线程，将需要修改“锁定”代码，以使用各种不同的“同步”API。

总之，和第 2.3 节介绍的 Socket API 存在的问题一样，原始 OS“并发”API 也表现出很多类似的问题。除此之外，“并发”API 在 OS 平台上更缺乏标准。即使是类似的 API，各个实现（implementation）之间也往往存在微妙的语法和语义差异。例如，在不同操作系统供应商实现的 Pthreads 标准的不同版本中，函数的参数列表和返回的错误标识符都各不相同。

在副栏 13 中，列出了不同“并发”API 的“错误传播策略（error propagation strategies）”的一些不同。这种“可移植性”上的不足会增加并发网络应用程序的“偶发复杂性”。因此，设计更高级的编程抽象体（programming abstractions），例如 ACE 中的那些抽象体，将有助于开发者避免“不可移植”和“不统一”的 API 所带来的问题。

副栏 13：并发 API 的“错误传播策略”

不同的并发 API 采用不同的方式向调用者报错。例如，一些 API（如 UI 线程和 Pthreads）在成功时返回 0，在失败时返回非 0 值。其他 API（如 Win32 线程）在失败时返回 0，并通过“线程专有存储空间”显示错误值。这种行为上的多样性容易让人混淆，而且不具可移植性。反之，ACE“并发”wrapper facade 定义并采用了统一的方式——即，在出错时总是返回–1，并会设置一个“和线程有关”的 errno 值，以表示失败的原因。

6.6 小结

操作系统提供了并发机制，用来管理一个终端系统中的多个进程以及一个进程中的多个线程。任何象样的通用 OS 都允许多个进程并发执行。现代通用操作系统和实时操作系统还允许多个线程并发运行。在使用并发机制时，如果能结合使用适当的模式和应用程序结构，将有助于性能的提高和程序结构的简化。

在以下 4 章，我们就 ACE 提供的“并发”wrapper façade 类进行了阐述：

- 第 7 章讲述 ACE“同步事件多路分离”类。
- 第 8 章讲述 ACE“OS 进程机制”类。
- 第 9 章讲述 ACE“OS 线程机制”类。
- 第 10 章讲述 ACE“OS 同步机制”类。

在这 4 章中，将讨论 ACE 如何运用 C++特性和 Wrapper Facade 模式来解决原始 OS 并发 API 存在的问题，以及如何增强并发网络应用程序的功能性、可移植性和健壮性；在合适的地方，还会给出 ACE 并发 wrapper facade 的实现，演示它们如何映射（mapped）到底层 OS 并发机制上；我们还会指出 OS 平台特性的差异之处，以及 ACE 如何让开发者免受这些差异所带来的影响。

第 7 章

ACE 同步事件多路分离 Wrapper Facade

本章提要

在本章讲述的 ACE wrapper facade 中，我们通过可移植的 C++类，对 OS 同步事件多路分离（synchronous event demultiplexing）机制进行了封装。还运用这些 ACE wrapper facade，为我们的网络日志服务器提供了几个新的改进版本。

7.1 概述

第 5.1 节讨论了服务器并发空间（server concurrency dimension），第 5.1 节中的“反应式服务器”是其中的设计选择之一。我们可以将“反应式服务器”模型看作“轻量级的多任务”：一个单线程服务器以轮流（round-robin）方式和多个客户通信；线程和同步机制的开销和复杂性在此不存在。这种服务器并发策略使用了一个事件循环，连续对来自客户的事件进行检查并做出反应。事件循环将“来自各个事件源（event source）”的输入进行多路分离，使它们能够以有序的方式接受处理。

网络应用程序中的事件源主要是 socket 句柄。对 socket 句柄来说，其最具可移植性的同步事件多路分离器是 select()函数和它的句柄集（handle set），如第 6.1 节所述。针对反应式服务器中的这些设施，本章介绍了以下类，以对它们的使用方式进行简化和优化：

ACE 类/方法	说 明
ACE_Handle_Set	封装了 fd_set 及其常用操作。
ACE_Handle_Set_Iterator	提供了一种可移植、高效的机制，用来遍历 ACE_Handle_Set 对象中的活动句柄集。
ACE::select()	简化 select()的最常用方式，并提高可移植性。

ACE_Handle_Set 和 ACE_Handle_Set_Iterator 类的接口如图 7.1 所示。这些 wrapper facade 具有以下优点：

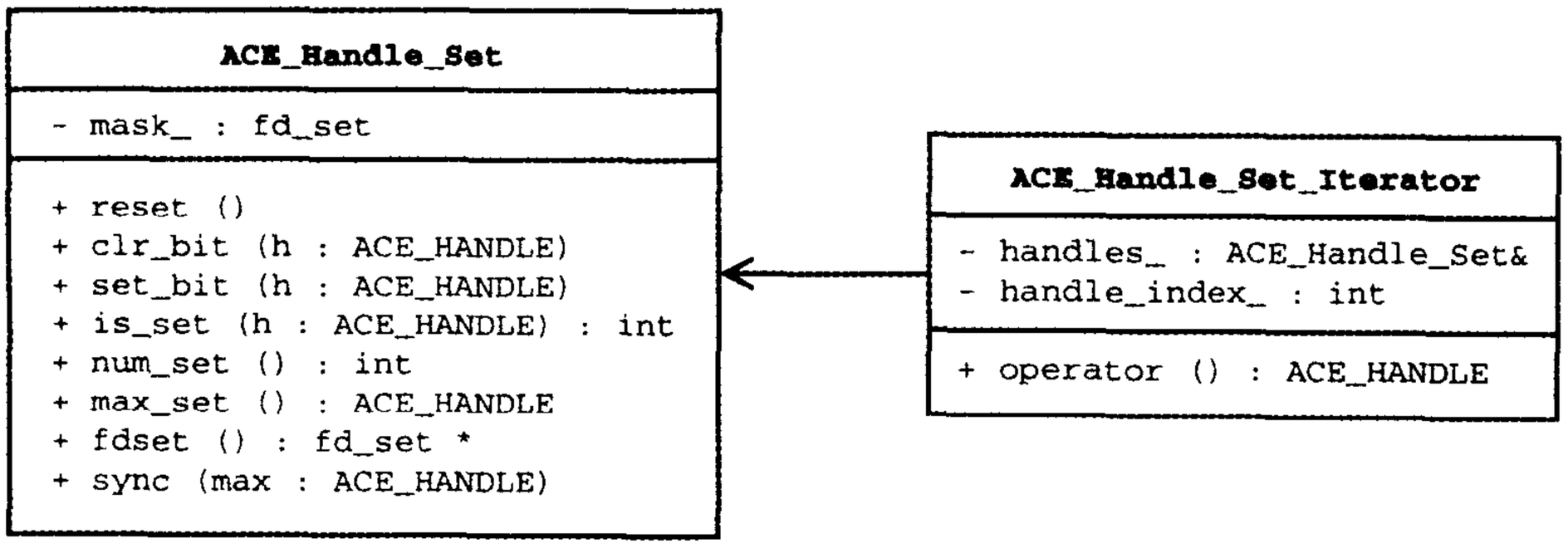


图 7.1：ACE_Handle_Set 和 ACE_Handle_Set_Iterator 类图

• 保证 OS 平台上的可移植性。ACE_Handle_Set 和 ACE_Handle_Set_Iterator 类可用在所有支持 select()和 fd_set 的 OS 平台上。在同步地对网络事件进行多路分离时，这些类能简化 I/O 句柄的使用。

• 可以高效遍历活动句柄集，同时不损及可移植性。高效地遍历活动句柄集（set of active handles）是提高“反应式服务器”事件循环的性能的一个重要因素。为了改善事件循环的性能，程序员往往得分析 fd_set 的内部结构，以找到某种更高效的方法来对它们进行搜索。为了消除网络应用程序对 fd_set 内部表示法的依赖，ACE_Handle_Set_Iterator 做了很大的优化。

ACE::select() 是一个包装器（wrapper）方法，它封装了 select()系统函数，并用 ACE_Handle_Set 类来简化“句柄传递”。我们将通过日志服务器的“反应式”版本向你演示：在编写网络应用程序时，这些 ACE 包装器是如何简化你的工作的。

7.2 ACE_Handle_Set 类

设计意图

在第 2.3 节，我们讨论了“直接使用 I/O 句柄”存在的复杂性。在以下方面，fd_set 体现了“偶发复杂性”的另一根源：

- OS 平台提供了一些宏和函数，用来操纵和搜索 fd_set，在使用这些宏或函数时必须十分小心，以避免处理“非活动句柄”，或无意中破坏 fd_set。

- 在代码中，搜索活动句柄的地方通常是程序中的一个*热点（hot spot）*，因为它是在一个密闭循环中连续执行的。因此，它显然是性能优化的一个候选者。

- fd_set 定义在系统提供的头文件中；所以，它的表示法暴露给了程序员。很自然，这会引诱程序员，使得他们想以某种“创造性”的方式利用这种“平台内部”的信息。但是，这样做会导致可移植性方面的问题，因为表示法在各个平台上各不相同。

- 将 fd_set 和 select()函数结合起来使用时，会产生微妙的“不可移植”问题。

为说明后面一点，让我们再次看看 select()函数的签名：

```
int select (int width,              // Maximum handle plus 1
            fd_set *read_fds,      // Set of "read" handles
            fd_set *write_fds,     // Set of "write" handles
            fd_set *except_fds,    // Set of "exception" handles
            struct timeval *timeout);// Time to wait for events
```

三个 fd_set 参数指定的是：选择每一种事件类型时使用的句柄。timeout 参数指定的是：等待事件发生的时间限制。这些参数的具体细节在各个平台上只有微小的差异。但是，I/O 句柄（从而，还有 fd_set）在各个平台之间天生就存在差异；所以，第一个参数（width）的含义在很多操作系统上是不同的：

• 在 UNIX 中，I/O 句柄的值从 0 开始，并且可以增长至“OS 定义的一个最大极限值”。因而，在 UNIX 中，width 参数的含义是，可以在 fd_set 中搜索“多远”的句柄；也就是说，三个 fd_set 参数中的最大句柄值加 1。

• 在 Win32 中，socket 句柄被实现为指针，而不是数值较小的整数。因此，传递给 select()的 width 参数会完全被忽略。相反，每一个 fd_set 都包含它自己的句柄数。

• 在其他平台上，句柄可能是整数，但并非从 0 开始，而且定义也会不同。

在 fd_set 中，那些“和大小（size）相关”的值，其计算也不同；这具体取决于 OS 平台中表示法（presentation）的具体特点，以及它的 select()语义。因此，直接针对原始 OS API 编写的任何应用程序，都必须适应这一切微妙的差异。为了向 width 提供正确的值，相应的代码得作修改，每一个项目最终可能都得重新设计和实现。要想在每一个应用程序中消除这些“可移植性”方面的差异，将十分麻烦，而且容易出错。正因为如此，ACE 提供了 ACE_Handle_Set 类。

功能

ACE_Handle_Set 类利用 Wrapper Facade 模式[SSRB00]来指导 fd_set 的封装。这个类提供了以下功能：

• 简化 fd_set 和 select()在各个 OS 平台上的使用，从而提高“事件驱动型”应用程序的可移植性、易用性和类型安全性。

• 在增加、删除句柄时，自动跟踪、调整 fd_set 中“和大小相关”的值。

fd_set 在表示法上的差异被隐藏在了 ACE_Handle_Set 的实现中。所以，应用程序代码只需编写一次，如果要将它移植到新的平台上，只用重新编译它就可以了。

ACE_Handle_Set 的接口如图 7.1 所示，其主要方法见下表：

方　法	说　明
ACE_Handle_Set() reset()	将句柄集初始化为缺省值。
clr_bit()	清除集中的一个句柄。
set_bit()	设置集中的一个句柄。
is_set()	检查集中的某一句柄是否活动。
num_set()	返回集中活动句柄的数目。
max_set()	返回集中最大句柄的值。
fdset()	返回一个指向内部 fd_set 的指针；或者，如果集中没有活动句柄，则返回 NULL。
sync()	使句柄集再次同步，以得到新的最大活动句柄和活动句柄的数目。在外部操作（例如对 select()的调用）导致句柄集被修改之后，这一功能很有用。

示例

在第 4.4.3 节和第 5.1 节中，我们讨论了循环式服务器存在的一些问题。为避免这些问题，一个策略是通过“同步事件多路分离”来实现“反应式服务器”模型。在下面的例子中，我们结合使用 select()和 ACE_Handle_Set 类，改善了第 4 章的循环式日志服务器的实现。这个服务器对以下两类事件实施多路分离：

1. “客户新连接”到达；
2. “客户连接上的日志记录”到达。

我们将“实现”（implementation）放进名为 Reactive_Logging_Server.h 的头文件中，这个头文件的开始处包含了 ACE 的 Handle_Set.h 文件和 Iterative_Logging_Server.h 文件。从第 4.4.3 节定义的 Iterative_Logging_Server 类派生一个新类：Reactive_Logging_Server。我们复用其数据成员，并定义另外两个成员，用于管理“多个客户”。

```
#include "ace/Handle_Set.h"
#include "Iterative_Logging_Server.h"

class Reactive_Logging_Server : public Iterative_Logging_Server
```

```
{
protected:
  // Keeps track of the acceptor socket handle and all the
  // connected stream socket handles.
  ACE_Handle_Set master_handle_set_;

  // Keep track of handles marked as active by <select>.
  ACE_Handle_Set active_handles_;

  // Other methods shown below...
};
```

为了实现“反应式服务器”语义，我们需要重写 Iterative_Logging_Server 中的 hook 方法。首先从 open()着手：[1]

```
virtual int open (u_short logger_port) {
  Iterative_Logging_Server::open (logger_port);
  master_handle_set_.set_bit (acceptor ().get_handle ());
  acceptor ().enable (ACE_NONBLOCK);
  return 0;
}
```

向下调用父类的 open()方法，从而在初始化了“接受者”之后，在 master_handle_set_ 上调用 set_bit()方法，以跟踪接受者的 socket 句柄。基于副栏 14 提到的原因，还将 acceptor 设置为“非阻塞”模式。

接着，为“反应式服务器”实现 wait_for_multiple_events()方法。将 master_handle_set_ 复制到 active_handles_中，后者会被 select()修改。我们要为 select()计算 width 参数，在 Win32 上，为了通过编译，这里需要一个转换（在这个平台上，传给 select()的实际值会被忽略）。select()的第二个参数是一个指针，指向 active_handles_内部的 fd_set，fd_set 可以通过 ACE_Handle_Set::fdset()方法访问。

```
virtual int wait_for_multiple_events () {
  active_handles_ = master_handle_set_;
  int width = (int) active_handles_.max_set () + 1;
  if (select (width,
              active_handles_.fdset (),
              0,  // no write_fds
              0,  // no except_fds
              0)) // no timeout
    return -1;
```

[1] 为节省篇幅，和前面的“循环式”日志服务器实现相比，我们在此省略了更多的出错处理代码。

```
active_handles_.sync
      ((ACE_HANDLE) active_handles_.max_set () + 1);
    return 0;
  }
```

副栏 14：“非阻塞”接受者的使用意图

如果将一个“接受者”socket 传递给 select()，则当接收到连接的时候，这个 socket 会被标记为“活动（active）”。很多服务器都通过这个事件来表明“可以无阻塞地调用 accept()”。不幸的是，TCP/IP 的异步行为会导致“竞态条件”。具体来说，当 select()将一个接受者 socket 标记为“活动”之后（但在调用 accept()之前），客户可以关闭它的连接，因而 accept()会阻塞，而且还可能使整个应用程序进程挂起（hang）。为避免这个问题，在使用 select()的时候，“接受者”socket 一定要被设置为“非阻塞”模式。在 ACE 中，我们的做法具有可移植性而且方便，它是通过将 ACE_NONBLOCK 标志传递给“ACE_IPC_SAP 输出（exported）的 enable()方法”来实现的，因而，也就是通过 ACE_SOCK_Acceptor 来实现的。

如果发生错误，select()会返回–1，这也是我们这个方法的返回值。如果成功，我们会返回活动句柄的数量，active_handles_fd_set 也会被修改，以标识每一个当前活动的句柄。调用 ACE_Handle_Set::sync()时，句柄的数目和 active_handles_中“和大小有关”的值会复位，以反应 select()所做的修改。

现在，给出 hook 方法 handle_connections()的实现。如果接受者的句柄是“活动”的，则会接受所有已经到达的连接，并将它们添加到 master_handle_set_中。因为我们将接受者设置为了“非阻塞”模式，所以，在接受了所有的等待连接（pending connections）之后，它会返回–1，并将 errno 设置为 EWOULDBLOCK，如副栏 14 所述。所以，不必担心进程会被无限期挂起。

```
virtual int handle_connections () {
  if (active_handles_.is_set (acceptor ().get_handle ())) {
    while (acceptor ().accept (logging_handler ().peer ()) == 0)
      master_handle_set_.set_bit
        (logging_handler ().peer ().get_handle ());
```

```
      // Remove acceptor handle from further consideration.
      active_handles_.clr_bit (acceptor ().get_handle ());
    }
    return 0;
  }
```

请注意此处 get_handle()方法的使用，以及下面的 handle_data()方法中 set_handle()的使用。之所以直接使用这种低级 socket 句柄，是因为“可以在一定程度上破坏类型安全”这一设计原则允许我们这样做。关于这一设计原则的具体说明，请阅读第 A.2.2 节。

handle_data()是一个 hook 方法，它循环遍历句柄集；对每一个活动句柄，它都会调用第 4.4.2 节中定义的 log_record()方法。这个方法对来自每一个活动连接的日志记录进行了处理。

```
// This method has problems. Do not copy this example!
virtual int handle_data (ACE_SOCK_Stream *) {
  for (ACE_HANDLE handle = acceptor ().get_handle () + 1;
       handle < active_handles_.max_set () + 1;
       handle++) {
    if (active_handles_.is_set (handle)) {
      logging_handler ().peer ().set_handle (handle);

      if (logging_handler ().log_record () == -1) {
        // Handle connection shutdown or comm failure.
        master_handle_set_.clr_bit (handle);
        logging_handler ().close ();
      }
    }
  }
  return 0;
}
```

在 handle_data()中，我们通过 ACE_Handle_Set 中的方法完成以下任务：

- 检测一个句柄是否“活动”；
- 当客户关闭连接时，清除 master_handle_set_中的句柄。

这个方法十分简明。但是，通过上面的方法使用 ACE_Handle_Set 类，还有几个缺点。第 7.3 节将讲述如何通过 ACE_Handle_Set_Iterator 类来解决这些问题；所以，在将这个方案运用到你自己的代码之前，请一定阅读第 7.3 节！

我们的 main()函数几乎和第 4.4.3 节的循环式服务器一模一样。惟一的区别是，我们定义的是一个 Reactive_Logging_Server 实例（instance），而不是 Iterative_Logging_Server 实例。

```
#include "ace/Log_Msg.h"
#include "Reactive_Logging_Server.h"

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

  if (server.run (argc, argv) == -1)
    ACE_ERROR_RETURN ((LM_ERROR, "%p\n", "server.run()"), 1);
  return 0;
}
```

7.3 ACE_Handle_Set_Iterator 类

设计意图

在前面的例子中，我们演示了“反应式服务器”如何将事件（event）多路分离（demultiplex）给事件处理程序。这需要以下几个步骤：

1. select()函数返回“被修改的句柄集”中的活动句柄数。

2. 服务器的事件循环搜索句柄集，找到活动句柄，并针对每一个句柄执行“事件处理”代码。

这些步骤看起来简单，但在实践中很复杂，因为 select()返回了“活动句柄的总数”，但没有标识它们位于哪个（些）句柄集。因而，应用程序必须搜索每一个句柄集中的内容，以找到活动句柄。搜索句柄集的一个方法是：在每一个（可能是“活动”的）句柄上调用 ACE_Handle_Set::is_set()，就像我们在第 7.2 节中介绍的 handle_data()方法中所做的那样。但是，这个方法存在以下两个问题。

• 只是在“将 socket 句柄表示为连续无符号整数”的 OS 平台上，那个用来遍历句柄的 for 循环才有效。这一技术虽然在 POSIX/UNIX 平台上可行，但在 Win32 上却无效，所以这段代码不具可移植性。而且，这个循环还假设，“接受者”socket 是“最小”的 socket 句柄。在复杂的应用程序中，如果句柄在“接受者”之前打开，随后又

被关闭，这一假设就不成立；这种情况下，句柄值会被重复利用（recycled）。

• 在支持“连续句柄值”的平台上，每一个 is_set()操作都会在所有可能的句柄值上连续调用。但是，网络应用程序中的常见情况是，在任何执行点上，即使 socket 句柄很多，但只有其中某一部分是活动的。在稀疏分布的“句柄集”中连续搜索是非常低效的——特别是句柄的数目很多的话，例如，在很多 UNIX 平台上，一个 fd_set 缺省有 1024 个句柄。虽然 ACE_Handle_Set 的 max_set()和 num_set()方法可用来限制搜索范围，但连续搜索句柄集还是会导致不必要的低效率。

在每一个应用程序中应付这些“可移植性”方面的问题，并针对这些瓶颈进行优化，不仅十分麻烦，而且容易出错。正因为如此，ACE 提供了 ACE_Handle_Set_Iterator 类。

功能

ACE_Handle_Set_Iterator 基于 Iterator 模式 [GHJV95]。它提供了连续访问聚合对象（aggregate object）中的元素的方法，但不暴露对象的底层结构。这个类可以高效地搜索 ACE_Handle_Set 中的句柄，并运用 operator()方法，一次只返回一个活动句柄。ACE_Handle_Set_Iterator 的接口以及它与 ACE_Handle_Set 类的关系如图 7.1 所示。它的主要方法见下表：

方　法	说　明
ACE_Handle_Set_Iterator()	初始化“句柄集迭代器(handle set iterator)”，以在 ACE_Handle_Set 上进行迭代。
operator()	返回句柄集中下一个未搜索到（unseen）的句柄。如果所有句柄都已被搜索到（seen），则返回 ACE_INVALID_HANDLE。

ACE_Handle_Set_Iterator 并没有被设计为“健壮的迭代器（robust iterator）”[Kof93]；所以，在一个正在被遍历的 ACE_Handle_Set 中，一定不要清除（clear）句柄。这一点很重要。

虽然 ACE_Handle_Set_Iterator 的接口（interface）很简单，但其实现（implementation）却比第3章介绍的 ACE Socket wrapper facade 更为强大。特别是，ACE_Handle_Set_Iterator 封装了所有“和平台相关”的 fd_set 的表示细节（representation detail），有助于移植以及高效率地遍历 ACE_Handle_Set 中所有的活动句柄，如下所述：

• Winsock 的 fd_set 中有一个字段，包含的是集中句柄值的数目。句柄本身存储在“句柄值数组（array of handle values）”中各个相邻的位置中。ACE_Handle_Set_Iterator 利用这些信息访问每一个句柄；它不必搜索下一个句柄，也不必为了得到活动句柄而搜索所有可能的句柄值。

• 大多数 UNIX fd_set 表示法（representation）都利用了 0~n 这个区间中所有可能的句柄值，做法是将每一个可能的句柄值都分配为一个比特（bit）。因而，一个 fd_set 包含一个整数数组，数组大小足以覆盖整个比特值区间。ACE_Handle_Set_Iterator 类会跳过整数数组中值为 0 的元素，从而可以快速略过一连串“非活动”句柄值。数组中的非零元素可以通过移位（shifting）和屏蔽位（masking）得以快速扫描，这之中使用了私有类成员来记住数组元素，以及 operator()调用之间的移位（shift position）。

• 通过使用更合适的移位和数组元素访问策略，可以对具有不同“大小”和“比特顺序”的底层 fd_set 进行调整。例如，可以配置 ACE_Handle_Set_Iterator，让它使用一个高度优化的位操作算法，此算法基于下面两个假设：

——fd_set 由一个 word 数组来表示（大多数 UNIX 平台上都这样）。

——如果 n 是一个 word，表达式 n &~(n–1)将计算一个 word；此 word 仅有的一个设置位（set bit）是 n 的最低设置位（lsb, least significant set bit）。

这个算法中的主要的优化在于：在 fd_set 的每一个 word 中，受检测的比特数完全等于活动比特数，而不是 word 数组中的总比特数。在活动句柄的数目非常少的情况下，这个算法可以大幅度地提高服务器的性能。

在这个算法中，ACE_Handle_Set_Iterator 首先在 fd_set 中找到“有活动位（active bit）的第一个 word”。因为表达式 word &~(word–1)在 word 中发现 lsb 为“活动”，所以必然找到 n，使得 lsb = 2^n。对于 fd_set 中的每一个 word，我们的成本为 O(word 中活动位的数目)。因而，如果一个 word 有 64 个比特，但只有一个活动位，那么，运行时成本将是 1，而不是 64。

• 可以配置 ACE_Handle_Set，让它使用另一个快速算法：在 select()设置活动句柄之后，这个算法会生成活动句柄的数目。此外，它还维护着一个包含 256 个元素的字符数组，每一个元素包含的是“生成索引值（index value）”需要的活动位的数目。例如，第 5 个元素包含的值为 2（2 个比特，即 0 和 2，表示二进制的 5）。略过 fd_set 元素并统计相应的数组值后，fd_set 中的活动位的数目可以快速计算出来，而不用连续搜索所有元素。

• ACE_Handle_Set_Iterator 是 ACE_Handle_Set 的友元（friend），因而，它可以用计算出的“活动句柄数”和“最大句柄值”来限定搜索空间。采用这一设计，ACE_Handle_Set_Iterator 的搜索范围将是一个最小的、必需的“活动句柄值”子集。

ACE 将所有这些优化细节都隐藏在了 ACE_Handle_Set_Iterator 之中，从而大大提高了主要的“同步事件多路分离”操作的性能，同时不损及应用程序的可移植性。这个类向我们展示：通过使用合适的抽象层，应用程序的性能将会显著提高。

示例

在这个例子中，我们将演示如何使用 ACE_Handle_Set_Iterator，从而克服第 7.2 节末尾部分提到的“反应式日志服务器”的缺陷。将把重点放在第 7.2 节演示的 handle_data()方法上，并将它的很多代码用一个 ACE_Handle_Set_Iterator 来代替，后者在 active_handles_上实施遍历，如下所示：

```
virtual int handle_data (ACE_SOCK_Stream *) {
  ACE_Handle_Set_Iterator peer_iterator (active_handles_);
```

```
    for (ACE_HANDLE handle;
         (handle = peer_iterator ()) != ACE_INVALID_HANDLE;
         ) {
      logging_handler ().peer ().set_handle (handle);

      if (logging_handler ().log_record () == -1) {
        // Handle connection shutdown or comm failure.
        master_handle_set_.clr_bit (handle);
        logging_handler ().close ();
      }
    }
}
```

这段代码更简明、更具可移植性；此外，由于 ACE_Handle_Set_Iterator 的优化，它还更为高效。

7.4 ACE::select()方法

设计意图

大多数 OS 平台都提供了 select()函数。即使是这样一个常见的函数，也存在一些微妙之处，使得它不那么容易使用。例如，回顾一下第 7.2 节中的 select()是如何使用的。在反应式服务器中，select()最常用的情况是在一组 socket 句柄上无限期等待输入，但此时程序员还是得为“写（write）”fd_set、“异常（exception）”fd_set，以及 timeout 指针提供“NULL 指针值”。此外，程序员还必须记住，一定要在 active_handles_上调用 sync()方法，以反映 select()所作的修改。如果在每一个应用程序中都要处理这些问题，那将十分麻烦，而且容易出错。因此，ACE 提供了包装器方法 ACE::select()。

功能

ACE 针对原始的 select()函数，定义了重载的静态包装器方法，这些方法简化了常见情况下的使用方式。它们的定义位于名为 ACE 的“公用类（utilities class）”中：

```
class ACE {
public:
  static int select (int width,
                     ACE_Handle_Set &rfds,
                     const ACE_Time_Value *tv = 0);
```

```
  static int select (int width,
                     ACE_Handle_Set *rfds,
                     ACE_Handle_Set *wfds = 0,
                     ACE_Handle_Set *efds = 0,
                     const ACE_Time_Value *tv = 0);
  // ... Other methods omitted ...
};
```

在 ACE 类中，第一个被重载的 select()方法省略了一些参数，并指定了一个缺省值，即“无超时值”，也就是“无限期等待”；第二个方法提供了“0”这个缺省值，用于不频繁使用的“写”及“异常”ACE_Handle_Set。当底层 select()方法回过头来复位“句柄集中的句柄数”和“同大小相关的值”，以反映 select()所作的任何变化时，两个方法都会自动调用 ACE_Handle_Set::sync()。

在设计这些包装器函数（wrapper function）时，要注意设计细节和常用情况，以简化编程工作与降低程序代码的出错几率。这个设计源于以下几个因素：

- **简化“常见情况（common case）”**。如上所述，在反应式服务器中，select()的最常见的使用方式是在一组 socket 句柄上无限期等待输入。ACE::select()方法简化了这种常用情况。在第 A.3 节，我们会进一步讨论这一设计原则。

- **封装“平台差异（platform variation）”**。所有版本的 select()都接受一个“超时（timeout）”参数；但只有在 Linux 版本中，这个超时值在返回时被修改，以反映一个句柄被选择（selected）之后，“超时”期限的剩余时间。在 ACE::select()包装器函数（wrapper function）中，“超时”值被声明为 const，从而明确地表明了它的行为特征；此外，这个函数还包含一些内部代码，用来解决 Linux 上出现的、非标准的“超时修改”情况。在第 A.5 节，我们将进一步讨论这一设计原则。

- **提供“类型可移植性（type portability）”**。OS 平台上使用的是 ACE_Time_Value 类，而不是原始的计时器类型；因为计时器类型在各个平台上并不是统一的。

示例

我们的日志服务器越有用、越具有可移植性，就越会有更多的客户程序使用它，它的负载也就会越重。因此，必须未雨绸缪：在后文设计的日志服务器中，一定要避免让它成为瓶颈。在后面几章，将探讨 OS“并发（concurrency）”机制及其相关的 ACE wrapper facade。但随着对并发机制使用的不断深入，目前还在使用的日志记录文件会成为瓶颈，因为所有的日志记录都汇集在那一个文件中。

因此，为了能够在以后扩充不同形式的并发机制，我们要对上文那个新的反应式服务器示例作改进：将不同客户的日志记录写到不同的日志文件中，一个客户对应一个文件。在图 7.2 中，我们展示了一个更具可伸缩性的“反应式”日志服务器的结构；它建立在本章前两个例子之上，但有所增强。如图 7.2 所示，这个反应式服务器实现（implementation）维护着一个 map 容器；可以用来为每一个客户保留单独的日志文件。还向我们展示如何使用 ACE::select()包装器方法（wrapper method）和 ACE_Handle_Set 类，以通过一个反应式服务器模型为多个客户提供服务。

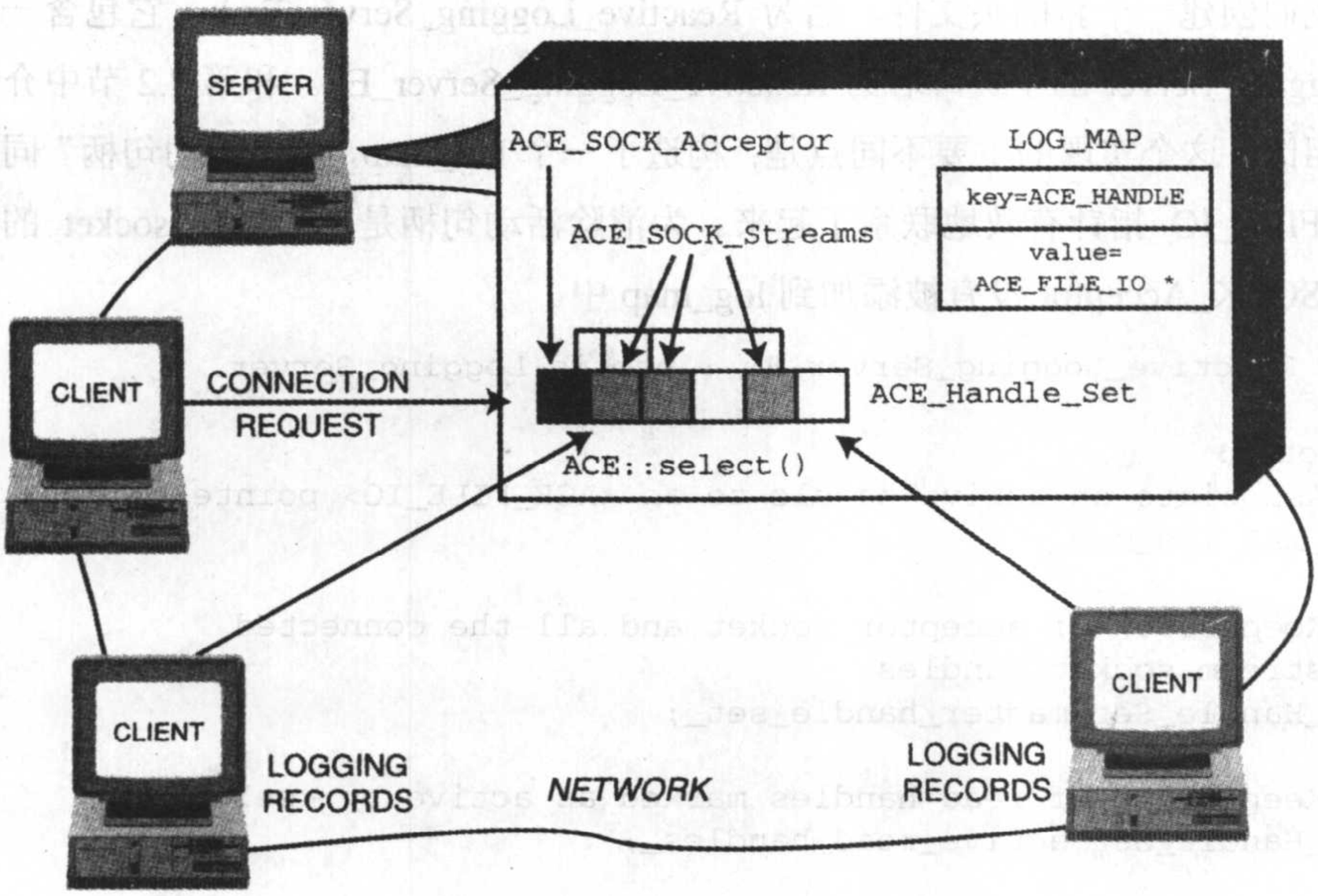


图 7.2 反应式日志服务器的结构

在实现（implementation）中，首先，包含几个新的头文件，这些头文件提供了日志服务器中将要用到的各项新功能。

```
#include "ace/ACE.h"
#include "ace/Handle_Set.h"
#include "ace/Hash_Map_Manager.h"
#include "ace/Synch.h"
#include "Logging_Server.h"
#include "Logging_Handler.h"
```

接着，提供 ACE_Hash_Map_Manager 模板的一个类型定义，说明见副栏 15。

```
typedef ACE_Hash_Map_Manager<ACE_HANDLE,
                             ACE_FILE_IO *,
                             ACE_Null_Mutex> LOG_MAP;
```

我们将使用这个模板的一个实例（instance），将一个活动的 ACE_HANDLE socket 连接（connection）高效地映射到 ACE_FILE_IO 对象上，这个对象和它的日志文件是对应的。将 ACE_HANDLE 用作这个映射的键（key），就解决了一个重要的“可移植性”问题：在 UNIX 上，socket 句柄是小的无符号整数，而在 Win32 上，它们是指针。

我们创建一个新的头文件，名为 Reactive_Logging_Server_Ex.h，它包含一个继承自 Logging_Server 的子类，名为 Reactive_Logging_Server_Ex。和第 7.2 节中介绍那个版本相比，这个实现的主要不同点是：构造了一个 log_map，将“活动句柄”同相应的 ACE_FILE_IO 指针有效地联系了起来。为消除活动句柄是一个流式 socket 的怀疑，ACE_SOCK_Acceptor 没有被添加到 log_map 中。

```
class Reactive_Logging_Server_Ex : public Logging_Server
{
protected:
  // Associate an active handle to an <ACE_FILE_IO> pointer.
  LOG_MAP log_map_;

  // Keep track of acceptor socket and all the connected
  // stream socket handles.
  ACE_Handle_Set master_handle_set_;

  // Keep track of read handles marked as active by <select>.
  ACE_Handle_Set active_read_handles_;

  // Other methods shown below...
};
```

副栏 15：ACE 容器类（container class）

ACE 提供了一套容器类，包括：

- 单链表和双链表
- 集和多重集
- 栈（stack）和队列（queue）
- 动态数组
- “字符串处理”类

在一切可能的地方，这些类都模拟了 C++标准库中的类；因而，随着 C++编译器逐渐成熟，二者很容易切换。

ACE_Hash_Map_Manager 定义了一个集抽象体，将“键”和“值”有效地关联了起来。之所以使用这个类而不是“标准的”std::map [Aus98]，基于以下原因：

1. std::map 并非那么标准——并不是所有支持 ACE 的编译器都实现了它；就算在实现了它的那些编译器上，它的接口也并非以相同的方式实现。

2. ACE_Hash_Map_Manager 基于散列技术(hashing)执行高效搜索；而 std::map 不支持这一点。

关于 ACE 容器类的更多介绍，请参考 [HJS]。

在 open()这个 hook 方法中，执行的只不过是反应式服务器初始化必需的步骤。

```
virtual int open (u_short logger_port) {
  Logging_Server::open (logger_port);
  master_handle_set_.set_bit (acceptor ().get_handle ());
  acceptor ().enable (ACE_NONBLOCK);
  return 0;
}
```

这个反应式服务器中的 hook 方法 wait_for_multiple_events()类似于第 7.2 节中介绍的那个版本。但在这个方法中，我们调用的是 ACE::select()；这是 ACE 中的一个静态包装器方法（wrapper method），它为 select()函数中“较少使用”的参数提供了缺省值。

```
virtual int wait_for_multiple_events () {
  active_read_handles_ = master_handle_set_;
  int width = (int) active_read_handles_.max_set () + 1;

  return ACE::select (width, active_read_handles_);
}
```

hook 方法 handle_connections()的实现类似于 Reactive_Logging_Server 中的版本。我们接受新的连接，并更新 log_map_和 master_handle_set_。

```
virtual int handle_connections () {
  ACE_SOCK_Stream logging_peer;

  while (acceptor ().accept (logging_peer) != -1) {
    ACE_FILE_IO *log_file = new ACE_FILE_IO;

    // Use the client's hostname as the logfile name.
    make_log_file (*log_file, &logging_peer);

    // Add the new <logging_peer>'s handle to the map and
    // to the set of handles we <select> for input.
    log_map_.bind (logging_peer.get_handle (), log_file);
    master_handle_set_.set_bit (logging_peer.get_handle ());
  }
  return 0;
}
```

请注意，我们使用了 make_log_file()方法（见第 85 页）；它继承自第 4.4.1 节介绍的 Logging_Server 基类。

handle_data() hook 方法只遍历活动连接，并在每一个连接上接收一个日志记录，然后将记录写入到“和这个客户连接关联”的日志文件中。

```
virtual int handle_data (ACE_SOCK_Stream *) {
  ACE_Handle_Set_Iterator peer_iterator (active_read_handles_);

  for (ACE_HANDLE handle;
       (handle = peer_iterator ()) != ACE_INVALID_HANDLE;
       ) {
    ACE_FILE_IO *log_file;

    log_map_.find (handle, log_file);
    Logging_Handler logging_handler (*log_file, handle);
```

```
    if (logging_handler.log_record () == -1) {
      logging_handler.close ();
      master_handle_set_.clr_bit (handle);
      log_map_.unbind (handle);
      log_file->close ();
      delete log_file;
    }
  }
  return 0;
}
```

客户关闭连接时，我们关闭相应的 Logging_Handler，在 master_handle_set_中清除句柄，在 log_map_中删除“句柄到文件”的关联，并删除动态分配的 ACE_FILE_IO 对象。在这个例子中，没有提供多少出错处理代码；但在产品级实现中，如果有出错情况发生，一定得采取适当的防范措施。

最后给出 main()程序。它和我们前面给出的那个版本基本相同；但这一次，定义的是一个 Reactive_Logging_Server_Ex 实例。

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

  if (server.run (argc, argv) == -1)
    ACE_ERROR_RETURN ((LM_ERROR, "%p\n", "server.run()"), 1);
  return 0;
}
```

7.5 小结

通过“反应式服务器模型”设计网络应用程序时存在偶发复杂性，还会造成性能降低、可移植性等问题，本章就这些问题作了分析。作为解决这些问题的一个方法，本章还介绍了 ACE_Handle_Set 和 ACE_Handle_Iterator wrapper facade。这些类有助于设计出可移植、健壮的服务器，同时不损及网络应用程序的性能。

反应式服务器作为一种常见的编程模型，允许一个服务器并发地处理“它与多个通信端或客户的交互”。尽管本章提供的网络日志服务器新版本向我们展示，要想“可移植地”做到这一点很容易，但这个版本还是有进一步改进的余地。在第 7.4 节提供的例子中，为了确定是否接受一个新的连接或接收日志记录，我们要去检查 socket 句柄在 ACE_Handler_Set 中是否“活动”。尽管这个设计是可行的，但它将“事件多路分离”

代码同“连接管理”代码紧密地联系在了一起。所以，尽管这些代码实际上是通用的，因而可以被重构（refactored）为可复用的框架（framework），但应用程序开发者还是需要重写它。

这里的基本问题是：虽然 select()和 fd_set 为功能强大的“事件多路分离”和“事件处理”能力提供了基础，但它们很难被封装在一个简单的 wrapper facade 中。所以，ACE 提供了一个基于 *Reactor 模式* [SSRB00]的框架，从而可以更灵活地重构（refactor）和复用（reuse）通用的事件多路分离（demultiplexing）和分发（dispatching）行为。我们将在[SH]中详尽讲述 ACE Reactor 框架。

第 8 章

ACE 进程 Wrapper Facade

本章提要

本章介绍 ACE 进程 wrapper facade。我们通过可移植的 C++类，对 OS 多进程机制进行了封装。我们演示了多进程设施（facility）在主流操作系统中的多种变化，还演示 ACE 如何运用一些技术，将“可移植的概念”抽象为“有用的类”。最后我们演示，在网络日志服务程序的多进程版本中，我们如何运用这些类。

8.1 概述

在网络应用程序中，OS 对“多进程”的支持很重要；因为多进程有助于：

- **实现并发性（Enable concurrency）**——允许 OS 在不同的 CPU 上调度（schedule）和运行（run）各个进程。

- **提高健壮性（Increase robustness）**——通过使用 MMU 硬件，使各个进程的地址空间免受系统中其他活动进程的无意或恶意破坏。

- **增强安全性（Enhance security）**——允许每一个进程校验或控制每一个用户、每一次会话的安全及验证信息。

本章介绍了以下 ACE 类；利用这些类，网络应用程序可以创建、管理一个或多个进程。

ACE 类	说 明
ACE_Process	可移植地创建和同步（synchronize）进程。
ACE_Process_Options	指定“与平台无关”和“与平台相关”的选项。
ACE_Process_Manager	可移植地创建和同步（synchronize）多组进程。

图 8.1 表示了这些类之间的关系。这些 wrapper facade 具有以下优点：

• 提高异种 OS 平台之间的可移植性。各种 OS 平台的“进程管理”能力大不相同。ACE 的多进程 wrapper facade 能可移植地访问这些能力的一个“公共子集”。例如，ACE_Process 类和 ACE_Process_Options 类可以统一创建、管理和同步进程，无论底层使用何种 OS 多进程机制。

• 易于访问“和 OS 相关”的机制。某些平台提供了功能强大的进程创建机制，如 POSIX 的 fork()。本章介绍的 ACE wrapper facade 允许开发者访问这些机制，且无需针对不同的项目修改应用程序代码。

• 将一组进程作为一个有机的集合来管理。在“使用了多进程”的网络应用程序中，多个进程往往需要像一个组（group）那样集体启动和终止。ACE_Process_Manager 类为 ACE_Process 提供的“单个进程管理（individual process management）”功能添加了这一能力。

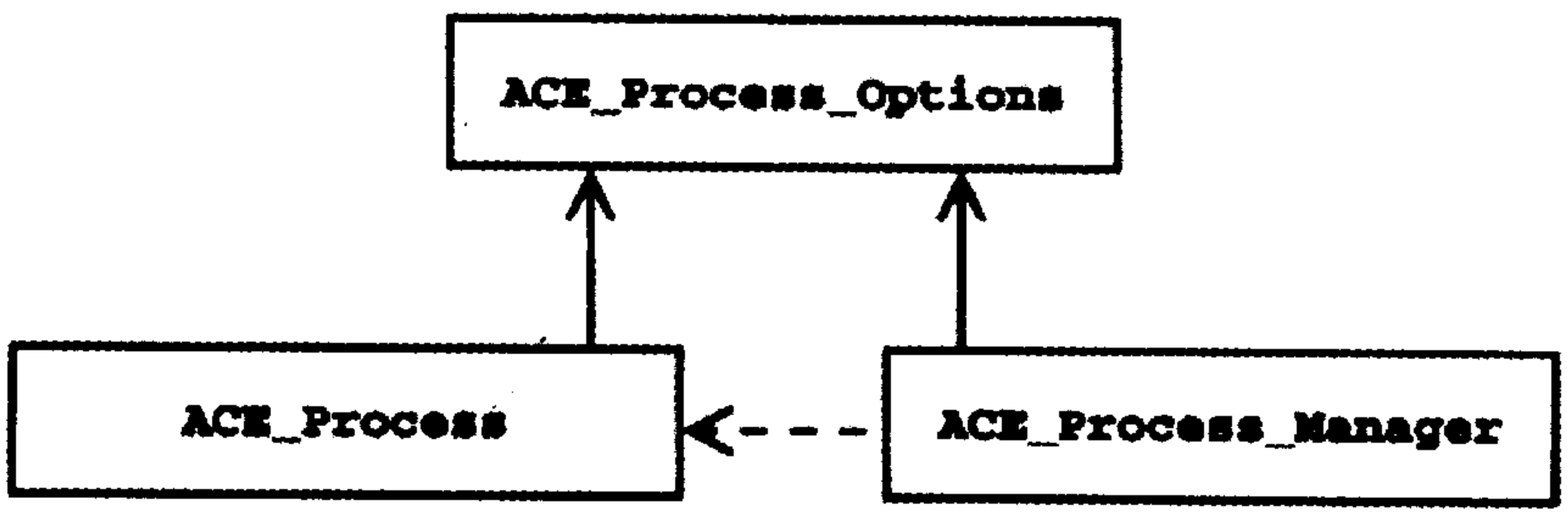


图 8.1：ACE 进程管理类关系图

本章提出并讨论了 ACE“多进程”类的功能。我们为每一个类提供了一个例子，通过这些例子，展示如何运用这些类，对网络日志服务程序的各个方面实施简化和加强。

8.2 ACE_Process 类

设计意图

进程的“创建”和“管理”是大多数通用操作系统的核心部分（但在第 6.2 节曾介绍过，一些嵌入式操作系统没有提供“进程”能力）。支持多进程的操作系统能提供以下功能：

- 创建一个进程，用来运行某一程序映像（program image）；
- “自愿”或“非自愿”地终止一个进程；
- 和另一个进程的“退出（exit）”保持同步。

有了这些完备的 OS 进程管理能力，可能会希望有一个统一的标准来管理这一领域。遗憾的是，出于很多原因，事实并非如此。例如，在下表中，我们给出了 POSIX/UNIX 和 Win32 进程管理的主要不同之处：

POSIX/UNIX	Win32
fork()复制调用者进程，包括打开 I/O 句柄；接着（可能）是一个 exec*()函数调用，将当前程序映像替换为新的程序映像。	CreateProcess()在一个系统调用中既创建新进程，又运行指定的程序映像。
kill()函数向一个进程发送信号，以终止这个进程。进程会截取一些信号，并将其解释为“程序终止”或“正常退出”请求。	TerminateProcess()强迫进程退出，不给进程留有清理自身的机会。
waitpid()函数等待子进程退出。父进程还可以捕获到 SIGCHLD 信号；当子进程退出时，OS 会将 SIGCHLD 信号发送给父进程。	WaitForMultipleObjects() 或 WaitForSingleObject() 函数可用来等待一个进程句柄；当那个进程退出时，函数会收到信号。

如果要完整讨论 OS 多进程机制，就超出本书的范围（参考 [Sch94, Ste92, Ric97] 可了解更多知识），但这个对照表还是展示了两大主流操作环境的主要不同之处。如你所见，OS 进程管理机制在语法和语义上各有差异。如果每一个应用程序都需要处理平台上的这些差异，那不仅困难、繁琐、容易出错，而且没有必要——因为 ACE 提供了 ACE_Process 类。

功能

运用 Wrapper Facade 模式，ACE_Process 类对不同 OS 多进程 API 之间的差异进行了封装。这个类定义了统一的、可移植的方法，赋予了应用程序以下能力：

- 创建和终止进程；
- 在进程退出时保持同步；
- 访问进程属性信息，如进程 ID。

ACE_Process 类的接口如图 8.2 所示，其主要的、“和平台无关”的方法见下表：

方　法	说　明
prepare()	子进程创建之前，由 spawn()调用。可以对新进程中将要用到的选项进行检测和（或）修改；可以方便地设置“和平台相关”的选项。
spawn()	创建新的进程地址空间，并运行指定的程序映像。
unmanage()	当 ACE_Process_Manager 管理的进程退出时，被 ACE_Process_Manager 调用。
getpid()	返回新子进程的进程 id。
exit_code()	返回子进程的退出代码（exit code）。
wait()	等待我们创建的进程退出。
terminate()	突然终止进程，不给进程清理自己的机会（所以应当慎用）。

ACE_Process 类提供了“创建新进程，并在新进程中执行新的程序映像”这一常见功能，并保证了可移植性。但是，在少数几个功能性领域（functional area）中，wrapper facade 也无法将所有的 OS 差异封装在一个标准的类中，无法做到在 ACE 支持的所有

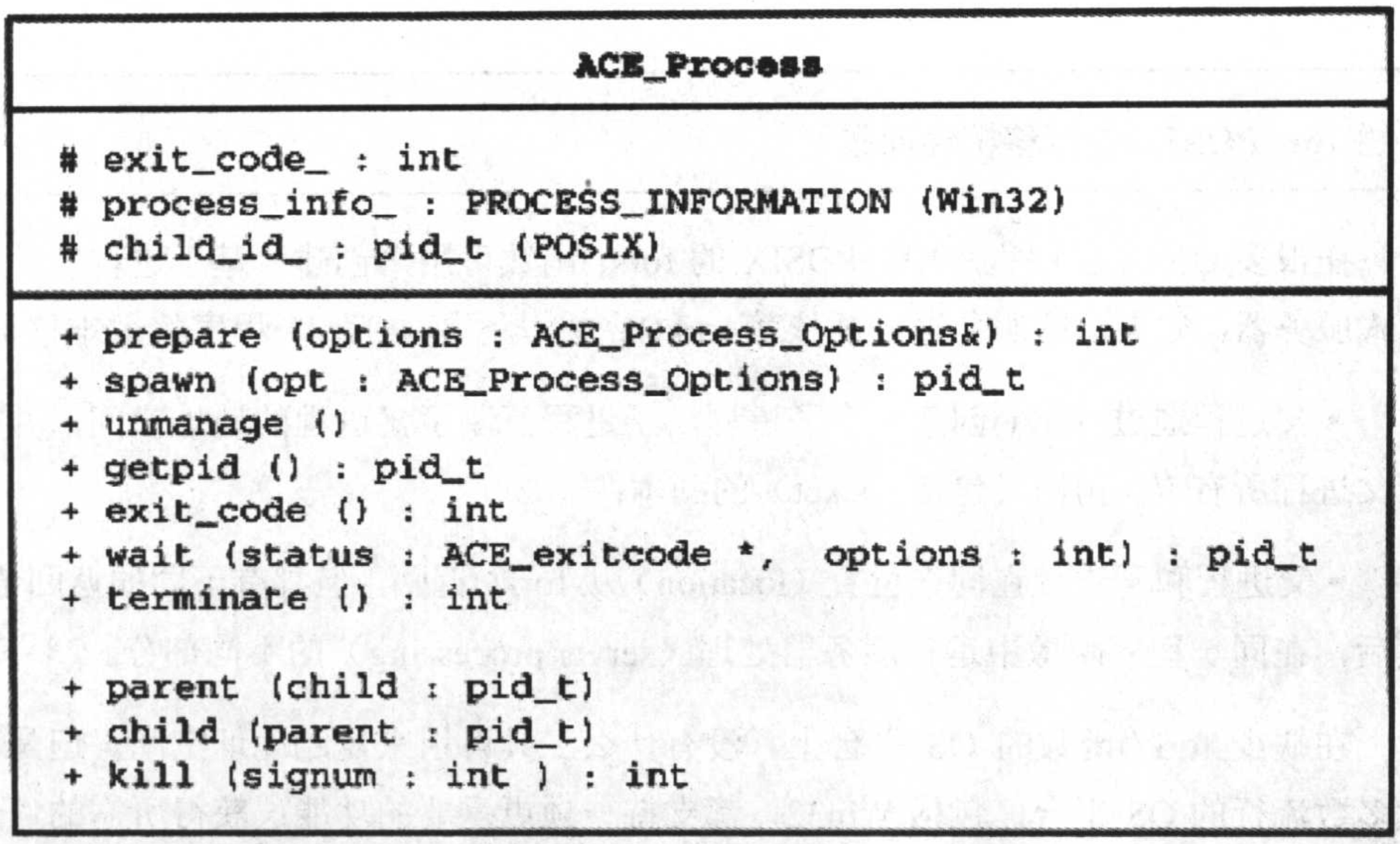


图 8.2：ACE_Process 类图

OS 平台上都确保可移植性——“进程管理”就属于这少数几个功能性领域之一。例如，有些服务器使用了 POSIX fork()机制中的某些独特语义，那将会导致某些可移植性方面的问题，副栏 16 就其中一些主要问题作了介绍。

UNIX 是网络应用程序的主要平台之一；因此，ACE_Process 类提供了几个“只在 POSIX/UNIX 环境下才有效”的方法。这些方法在很多设计场合下都十分有用（实际上，也很必要）。为避免混淆，下表专门列出了这些方法，以将它们同 ACE_Process 提供的那些具有可移植性的方法区分开来。

方　法	说　明
parent()	可以重写，以提供“和应用程序相关”的行为；在父进程环境中，如果 fork()调用之后子进程成功创建，这个方法将会被调用。
child()	可以重写，以提供“和应用程序相关”的行为；这个方法在子进程环境中调用，而且是在 fork()之后、exec()之前。
kill()	向进程发送信号。仅在能够向指定进程发送信号的 UNIX/POSIX 操作系统中具有可移植性。

副栏 16：POSIX 的可移植性问题

在很多 UNIX 应用程序中，POSIX 的 fork()函数可用来创建“基于进程”的并发式服务器。如果这些服务器需要依赖 fork()完成以下两项工作，程序将很难移植：

- 父进程通过 fork()创建一个子进程，子进程复制了父进程的地址空间，并拥有父进程所有 I/O 句柄（包括 socket）的副本；
- 父进程和子进程在同一位置（location）从 fork()返回，但具有不同的返回值，然后，在同一程序映像中运行服务器处理（server processing）的不同部分。

在缺少 fork()函数的 OS 平台上，没有什么工具可用来复制地址空间。但是，大多数流行的 OS 平台，包括 Win32，都支持一项更重要的功能：获得所需的“子进程中的 I/O 句柄”。ACE_Process_Options 类提供了一个可移植的方式来使用这项功能，如第 8.4 节日志服务器例程所示。

示例

在这个例子中，将演示如何运用 ACE_Process，可移植地创建一个“运行某一指定程序映像”的新进程。此程序通过以下方式“递归地（recursively）”计算阶乘：

1. 让一个父进程创建一个子进程，此子进程运行程序映像本身，以计算一个数的阶乘，此数通过 FACTORIAL 环境变量传递进来，如果没有设置环境变量，将缺省计算 10 的阶乘（10!）；

2. 每一个父进程等待子进程结束，用子进程的退出代码来计算 n 的阶乘，并将计算结果作为进程的退出代码返回。

显然，这不是一个高效的计算阶乘的方法，但它简明地演示了 ACE_Process 的功能，如下所示。

```
#include "ace/OS.h"
#include "ace/Process.h"
```

```
int main (int argc, char *argv[])
{
  ACE_Process_Options options;
  char *n_env = 0;
  int n;

  if (argc == 1) { // Top-level process.
    n_env = ACE_OS::getenv ("FACTORIAL");
    n = n_env == 0 ? 0 : atoi (n_env);
    options.command_line ("%s %d", argv[0], n == 0 ? 10 : n);
  }
  else if (atoi (argv[1]) == 1) return 1; // Base case.
  else {
    n = atoi (argv[1]);
    options.command_line ("%s %d", argv[0], n - 1);
  }

  ACE_Process child;
  child.spawn (options); // Make the "recursive" call.
  child.wait ();
  return n * child.exit_code (); // Compute n factorial.
}
```

ACE_Process::wait()调用总是可行的，无论被创建线程在调用之前是否结束。如果结束，调用会立即结束；否则，这个方法会等待，直至子进程退出。这个方法可以移植到所有支持多进程的操作系统上。

上面的例子演示了 ACE_Process_Options 的最基本、最具可移植性的使用方式：为“被创建程序”指定命令行。下一节讨论为什么需要一个独立的“选项”类，并演示了它的能力范围。

8.3 ACE_Process_Options 类

设计意图

操作系统提供了各种方法，用来设定“新创建进程”的属性。新进程的属性影响到它和其他进程及执行环境的关系。一些常见属性包括：

• **程序映像（program image）**。新进程应该执行哪个程序？

• **打开的 I/O 句柄（open I/O handles）**。子进程应该继承“打开的 I/O 句柄”或其他 OS 对象吗？如果继承了“打开的句柄”，子进程是否应该关闭其中的一部分或全部？

• **访问显示设备（access to display）**。子进程应该访问终端（terminal）或用户的显示器吗？

• **工作路径（working directory）**。子进程应该和父进程一样运行在相同的路径吗？或者，应该运行在不同的路径？

• **进程关系（process relationship）**。子进程是应该作为“独立的*监控程序（daemon）*”在后台运行，还是应该作为“相互关联的团体中的某一部分”在后台运行？

• **安全属性（security attribute）**。为了扩充或限制子进程的能力，子进程的安全属性或“缺省的文件保护设定”能否被修改？

这些选择所涉及的功能和 API 一般不具可移植性，即使在相似的平台上也不例外。若要确保正确地表达每一个应用程序的进程属性，就需要大量的经验和平台的具体信息。ACE_Process_Options 能够可移植地捕获这些信息。

功能

有了 ACE_Process_Options 类，我们就有了一种统一的方式来将进程属性传递给 ACE_Process 类和 ACE_Process_Manager 类；后者可以利用这些属性信息，对底层的 OS 进程创建和环境机制进行选择并施加影响。ACE_Process_Options 提供了以下功能：

• 允许应用程序指定想要的“进程控制信息”。

• 允许“进程控制项（process control item）”随着平台的改变而扩展。

• 提供了一种“去耦”机制，使得 ACE 可以提供这些功能，但不影响“进程创建”接口。

ACE_Process_Options 类的接口如图 8.3 所示，其“和平台无关”的主要方法见下表：

方　法	说　明
command_line()	通过 printf 风格的“格式字符串”，指定程序的命令和参数；在“被创建进程”中启动新程序时会使用它们。
setenv()	指定环境变量，并将其添加到被创建进程的环境中。
working_directory()	指定一个新路径，使被创建进程在启动新程序映像之前切换至这一路径。
set_handles()	设置某些特殊的文件句柄；被创建进程可将这些句柄用于它的 STDIN、STDOUT 和 STDERR。
Pass_handle()	表示一个“应被传递给新创建的进程”的句柄。

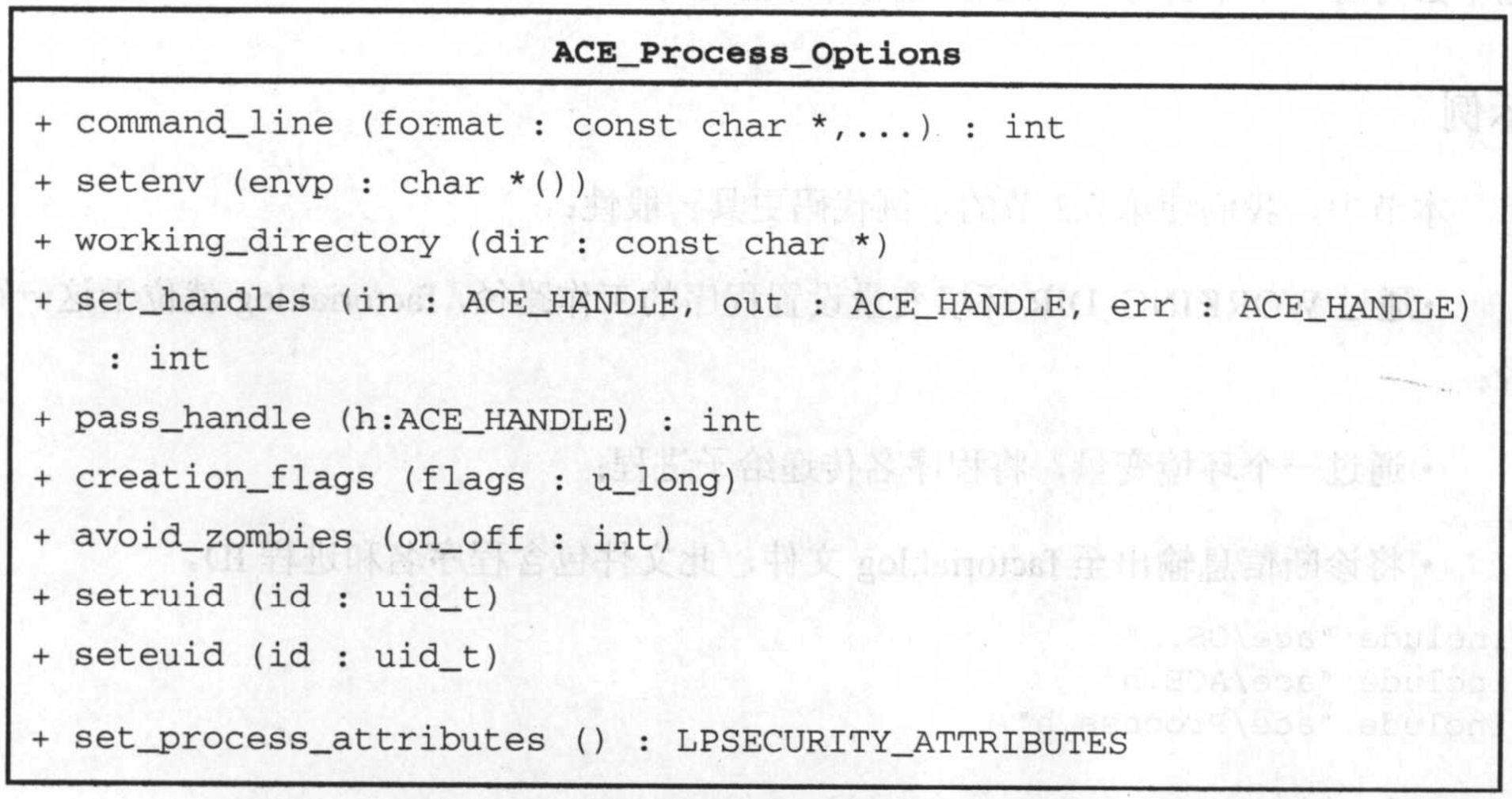


图 8.3：ACE_Process_Options 类图

在第 8.2 节，我们解释了 ACE_Process 提供以下两类方法的原因：

1. 在所有“支持进程”的操作系统上都可移植的方法；

2. 只能工作于特定平台上的方法。

ACE_Process_Options 类也采用了这一方法。针对所有的“多进程平台”，上面列出的方法提供了一组“可移植”的属性信息。下面列出的方法则用来访问“和平台相关”的属性：

方 法	说 明
creation_flags()	指定是否在被创建进程中运行新的程序映像（POSIX）。
avoid_zombies()	允许 ACE_Process 采取措施，确保被创建进程在退出时不会成为*“僵死（zombie）”*进程（死进程）（POSIX）。
setruid() seteuid()	分别为被创建进程指定“真实（real）”、“有效（effective）”的用户 ID（POSIX）。
set_process_attributes()	为被创建进程设置进程属性（Win32）。

有了这些方法，我们既可以在“多进程”应用程序中使用“不可移植”的功能，也不必为每一个平台重写“进程属性”管理代码。

示例

本节中，我们使第 8.2 节的示例代码更具一般性：

- 通过 WORKING_DIR 环境变量设置程序的工作路径，factorial.log 就位于这一路径；
- 通过一个环境变量，将程序名传递给子进程；
- 将诊断信息输出至 factorial.log 文件，此文件包含程序名和进程 ID。

```
#include "ace/OS.h"
#include "ace/ACE.h"
#include "ace/Process.h"

int main (int argc, char *argv[])
{
  ACE_Process_Options options;
  FILE *fp = 0;
  char *n_env = 0;
  int n;

  if (argc == 1) { // Top-level process.
    n_env = ACE_OS::getenv ("FACTORIAL");
    n = n_env == 0 ? 0 : atoi (n_env);
    options.command_line ("%s %d", argv[0], n == 0 ? 10 : n);
    const char *working_dir = ACE_OS::getenv ("WORKING_DIR");
    if (working_dir) options.working_directory (working_dir);
    fp = fopen ("factorial.log", "a");
```

```
    options.setenv ("PROGRAM=%s", ACE::basename (argv[0]));
  }
  else {
    fp = fopen ("factorial.log", "a");
    if (atoi (argv[1]) == 1) {
      fprintf (fp, "[%s|%d]: base case\n",
               ACE_OS::getenv ("PROGRAM"), ACE_OS::getpid ());
      fclose (fp);
      return 1; // Base case.
    } else {
      n = atoi (argv[1]);
      options.command_line ("%s %d", argv[0], n - 1);
    }
  }

  ACE_Process child;
  child.spawn (options); // Make the "recursive" call.
  child.wait ();
  int factorial = n * child.exit_code (); // Compute n factorial.
  fprintf (fp, "[%s|%d]: %d! == %d\n",
           ACE_OS::getenv ("PROGRAM"), ACE_OS::getpid (),
           n, factorial);
  fclose (fp);
  return factorial;
}
```

在不同的 OS 平台（如 Win32 和 POSIX）上，ACE_Process_Options 和 ACE_Process 代码调用的系统函数大不一样；但在所有这些平台上，上面的代码都可以移植，都能够正确地编译和运行。

8.4 ACE_Process_Manager 类

设计意图

复杂的网络应用程序中往往需要多组进程，以协同提供某一服务。例如，在一个具有多级工作流程的自动化应用程序中，可能需要创建多个进程，以分别处理一个大型问题中的各个部分。在进入工作流程的下一步之前，“主进程（master process）”会等待整组“工作者进程（worker processes）”退出。这是一个常见范式；因而，ACE 提供了 ACE_Process_Manager 类。

功能

ACE_Process_Manager 类通过运用 Wrapper Facade 模式，不仅具备了 ACE_Process 的可移植性和强大威力，还能将多组进程作为一个单元来管理。这个类具有如下功能：

- 保存内部记录，管理和监视 ACE_Process 类创建的多组进程。
- 允许一个进程创建一组进程，并且，在继续执行自己的处理之前，必须等待这一组进程退出。

ACE_Process_Manager 类的接口如图 8.4 所示，其主要方法见下表：

方　法	说　明
open()	初始化 ACE_Process_Manager。
close()	释放所有资源（不等待进程退出）。
spawn()	创建一个进程，将其添加到将被管理的一个进程组中。
spawn_n()	创建 n 个进程，这些进程都属于同一进程组。
wait()	等待进程组中的一些或全部进程退出。
instance()	返回一个指向 ACE_Process_Manager singleton 的指针的静态方法。

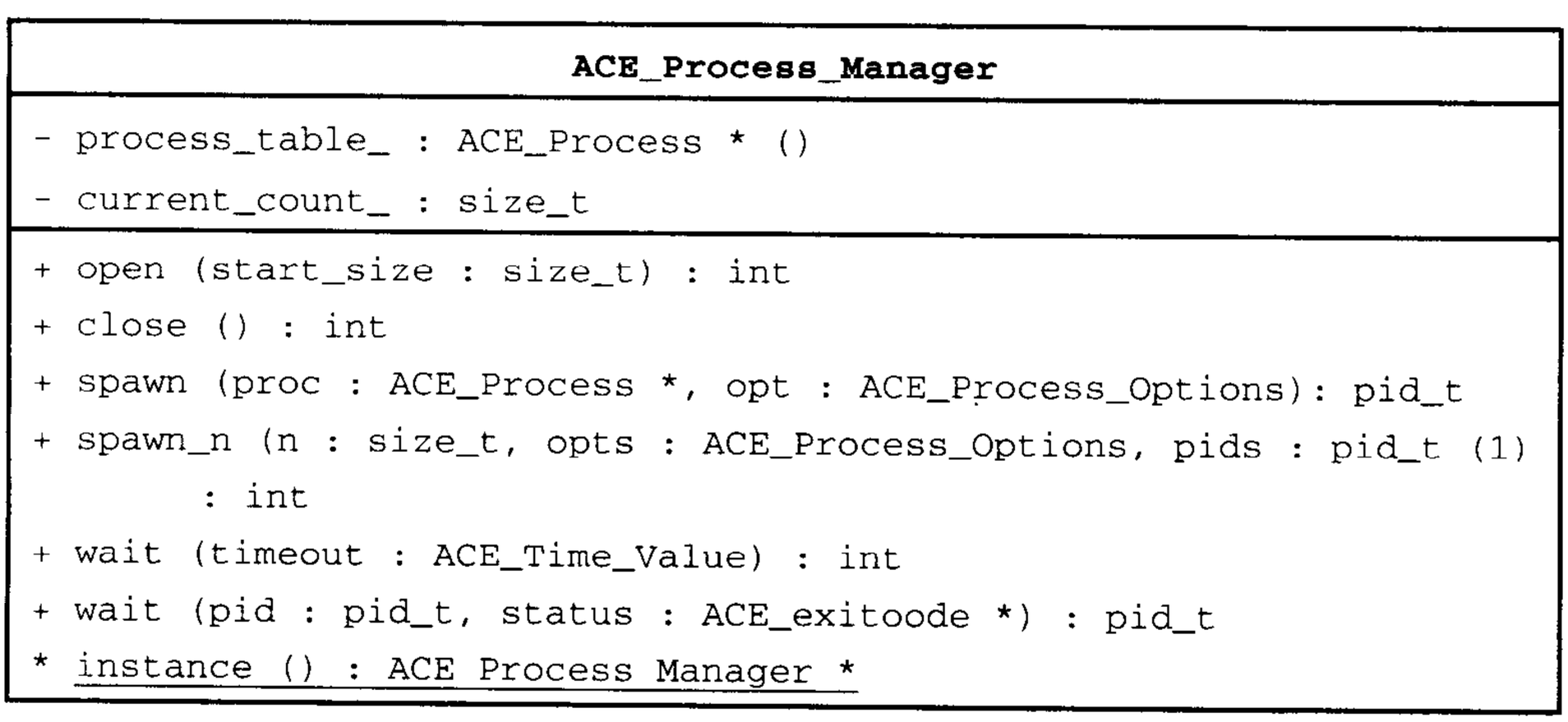


图 8.4：ACE_Process_Manager 类图

可以通过两种方式使用 ACE_Process_Manager：

- **作为 Singleton [GHJV95]使用。**这要用到它的 instance()方法。

- **实例化一个或多个实例**。此功能可用于支持“一个进程中的多组进程”。

示例

在第 7.4 节的例子中，演示了一个“反应式”日志服务器的设计和实现；那个例子消除了“纯循环式”服务器设计的缺陷。另一种处理客户请求的服务器模型是，创建一个新进程来处理每一个客户。当“多线程”方案面临以下情况时，“基于进程”的并发模型就有了用武之地：

- **不可能使用多线程方案**。例如，在缺乏有效的线程实现或完全缺乏线程支持的旧式 UNIX 系统中。
- **不适合使用多线程方案**。例如，受“不可重入”的第三方程序库限制；或者，所支持的硬件在时间和空间分割（partitioning）上要求具有高可靠性。

通过“多进程”而不是“多线程”来实现服务器，还有其他利弊。

基于“多进程”的日志服务器的结构如图 8.5 所示。这个修改版本使用的是“一个连接一个进程”并发模型。和第 4.4.3 节那个“使用了循环式服务器设计”的第一个版本相比，此版本有很多类似之处。这里的主要不同是，对于日志服务器端口上每一个已经接受的连接，主进程都会相应地创建一个新的工作者进程；然后，主进程继续接受新的连接。对于客户通过一个连接发送而来的所有日志请求，每一个工作者进程都会进行处理；在一个连接关闭时，相应的进程会退出。

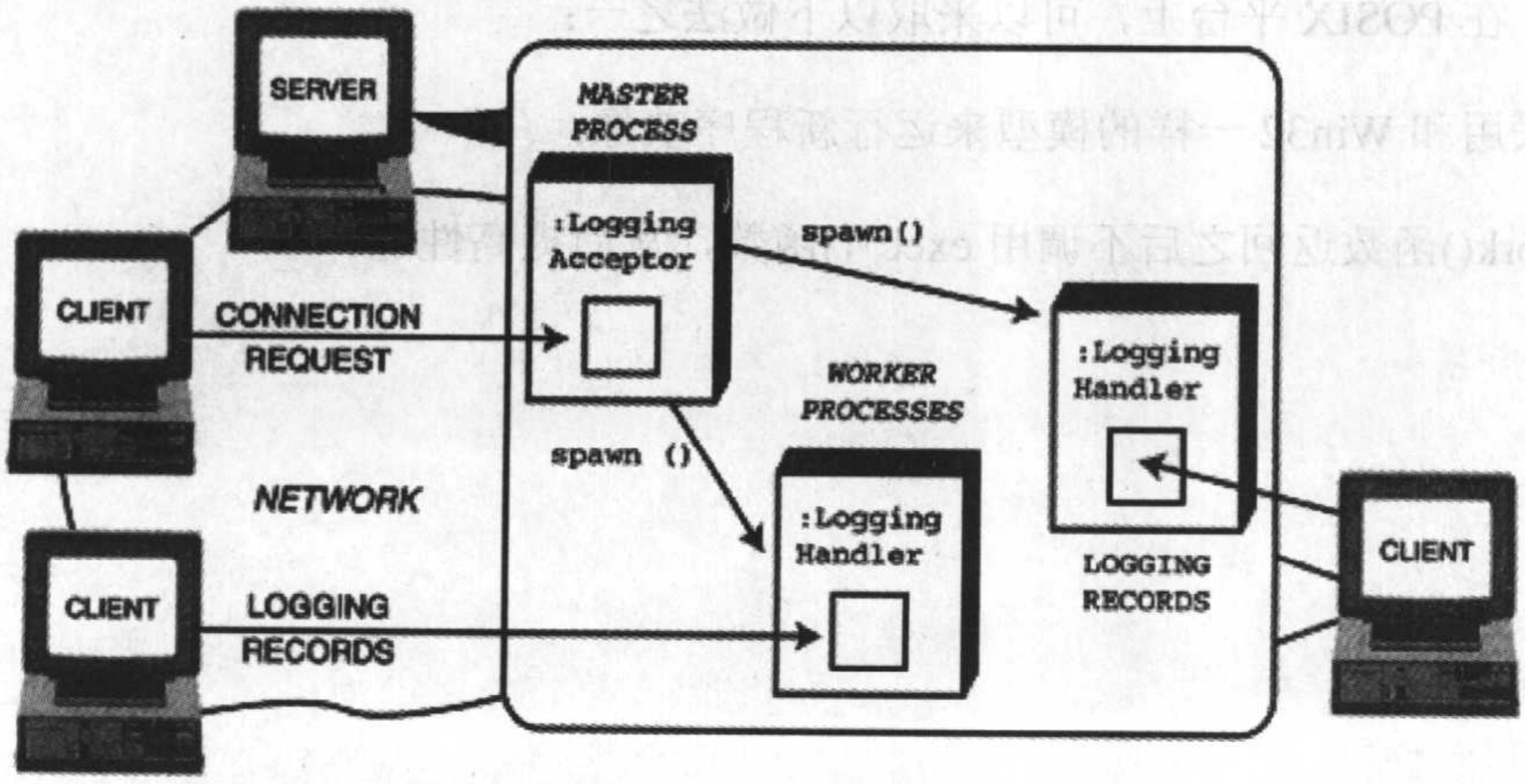


图 8.5：多进程日志服务器的结构

“一个连接一个进程”方案有两大优点：

1. 使得日志服务器更为稳固，可以避免“一个实例破坏其他实例”的问题。这个问题在某些情况下可能发生。例如，内部有 bug；或者，某一用户采取某种方法，恶意攻击“缓冲区溢出”这一弱点。

2. 很明显，下一步是去扩充日志服务器的功能，允许每一个进程可以检验或控制“每个用户”或“每个会话”的安全及验证信息。例如，在每一个具体用户的日志会话中，开始时都要有一个“用户/口令”交换。如果使用“基于进程”的服务器，而不是使用“反应式”或“多线程”服务器，在处理每一个进程中的具体用户信息时，就不会无意（或恶意！）造成和另一个用户信息相混淆的情况——因为每一个进程都有自己的地址空间和访问权限。

下面演示的“一个连接一个进程”服务器在 POSIX 和 Win32 平台上都可正常运行。从第 8.2 节和副栏 16 中关于平台差异性的说明可以知道，这是一个令人振奋的进步。由于在这个日志服务器示例的设计中，关系被彻底分离，再加上 ACE 对 wrapper facade 的明智使用，服务器代码的差异性被降至最低，并得以很好地封闭。特别是，应用程序中不需要条件编译代码。

但是，由于存在不同的进程创建模型，故首先得决定如何运行 POSIX 进程。Win32 强制程序映像运行在新进程中，POSIX 则不然。在 Win32 中，我们将服务器逻辑全都放在一个程序中，并通过不同的命令行参数，在工作者进程和主进程中都执行这个程序映像。在 POSIX 平台上，可以采取以下做法之一：

- 采用和 Win32 一样的模型来运行新程序映像；
- fork()函数返回之后不调用 exec*()函数，从而提高性能。

为了获得性能优势，并且为了演示 ACE 可以容易、正确地实现这两种方式，在 POSIX 上，我们不在工作者进程中运行新程序映像。

“一个连接一个进程”日志服务器的代码并不十分复杂，即使它在 POSIX 和 Win32 上的运作方式不同。但是，要想清楚地说明 ACE wrapper facade 在为我们提供这种简易性时所涉及的事件顺序和细节，却是十分困难。为了便于理解，将分开说明 Process_Per_Connection_Logging_Server 类、示例程序的进程管理，以及 Logging_Process 类。

Process_Per_Connection_Logging_Server 类。和第 4.4.3 节介绍的“循环式”日志服务器一样，在这里要定义一个类来表示我们的服务器。首先是包含所需的 ACE 头文件：

```
#include "ace/Log_Record.h"
#include "ace/Process.h"
#include "ace/Process_Manager.h"
#include "ace/Signal.h"
#include "Logging_Server.h"
```

从第 4.4.1 节定义的 Logging_Server 类，我们派生出 Process_Per_Connection_Logging_Server 类。我们还重写 run()方法，使得程序可以在 Win32 上以两种不同的方式执行：作为“主进程”接受连接并创建工作者进程；作为“工作者进程”为日志客户提供服务。其区别可以通过命令行参数来表达：

- 主进程启动时会使用命令行上的一个可选端口号，就像其他的网络日志服务器实现一样。
- 工作者进程启动时使用两个参数：(1) +H（ACE 为一个“被传递句柄”添加的“句柄”选项）；(2)从主进程派生而来的 socket 句柄的数值（numeric value）。

在图 8.6 和图 8.7 中，我们分别描述了在 POSIX 和 Win32 上，主进程和工作者进程之间的关系。

我们从第 180 页开始研究 Logging_Process 类，届时将看到命令行参数是如何实际传递的。首先，分析一下服务器类的定义：

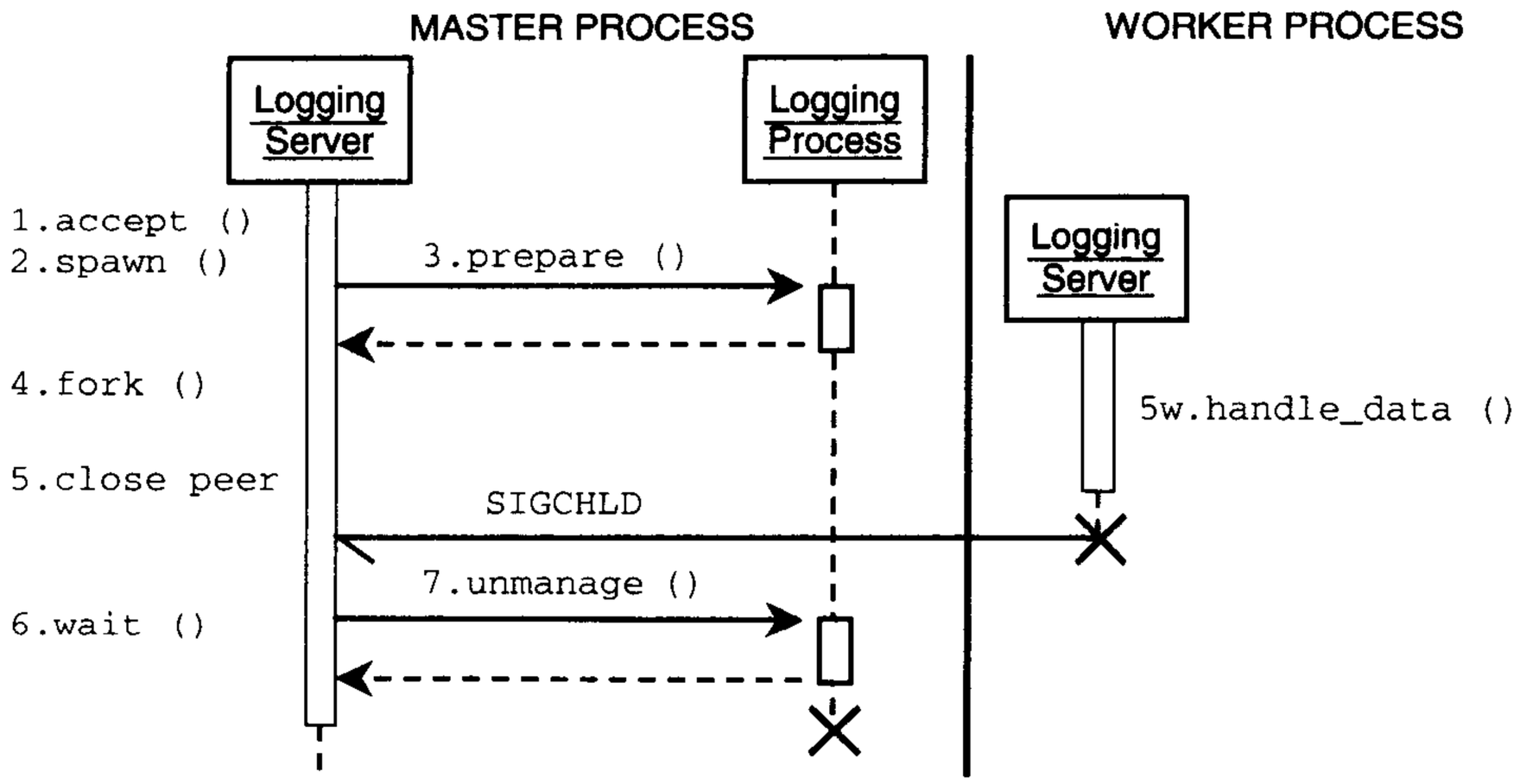


图 8.6：主进程/工作者进程的创建顺序（POSIX）

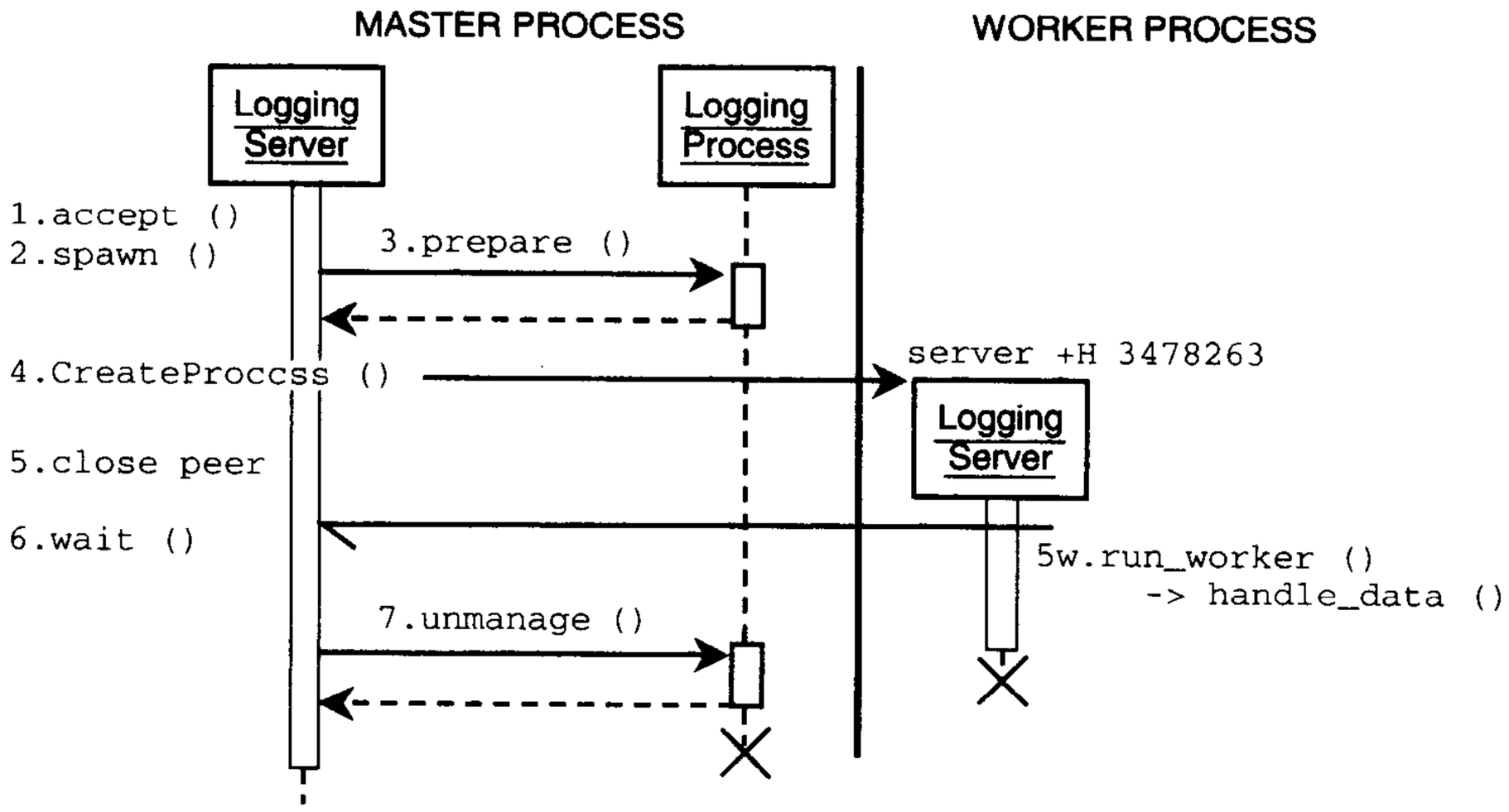


图 8.7：主进程/工作者进程的创建顺序（Win32）

```
class Process_Per_Connection_Logging_Server
  : public Logging_Server
{
protected:
  char prog_name_[MAXPATHLEN + 1];
```

从 argv[0]命令行参数中，prog_name_数据成员获得服务器程序的名称。用这个名称来创建工作者进程，以处理新连接。在 run()方法中，检查命令行参数的个数，从而确定服务器是应该作为主进程运行，还是应该作为工作者进程运行。如果有两个参数，这将是一个工作者进程，否则为主进程。

```
virtual int run (int argc, char *argv[]) {
  strncpy (prog_name_, argv[0], MAXPATHLEN);
  prog_name_[MAXPATHLEN] = '\0'; // Ensure NUL-termination.
  if (argc == 3)
    return run_worker (argc, argv); // Only on Win32.
  else
    return run_master (argc, argv);
}
```

run_master()方法和 Logging_Server::run()方法类似；例如，它会打开日志服务器的侦听端口，并调用 handle_connections()，以接受新的客户连接。但它不会调用 handle_data()这个 hook 方法，因为后者总是在工作者进程中调用。主服务器创建一个新的工作者进程，用来处理每一个客户的日志记录，如图 8.5 所示。

```
int run_master (int argc, char *argv[]) {
  u_short logger_port = 0;
  if (argc == 2) logger_port = atoi (argv[1]);
  if (open (logger_port) == -1) return -1;

  for (;;)
    if (handle_connections () == -1) return -1;

  return 0;
}
```

我们继承了 Logging_Server 基类（base class）的 open()方法的实现；此方法对“接受者”端点进行初始化，使之在某一指定端口上被动侦听。因为 wait_for_multiple_events()的实现是一个空操作，所以，在此简单地忽略它，并直接调用 handle_connections()，以运行主事件循环。

run_worker()方法只在 Win32 上执行。创建工作者进程时，主进程请求在命令行上将 socket 句柄传给工作者进程。run_worker()方法将这个命令行参数转换回句柄，用这个句柄构造 ACE_SOCK_Stream 对象，并调用 handle_data()来处理客户的日志记录。因为传入的数据类型是固定的，所以必须经过转化。为了达到 OS 平台的 C++编译器允

许的最大安全性， ACE 提供了一套可移植的、用于转换的宏（包括下面使用的 ACE_static_cast），如副栏 17 所述。

副栏 17：在所有 C++编译器上都可移植的类型转换

如果程序员们在学习 C++之前已经学习了 C，那么，他们大都熟悉“类型转换”运算符：(type) expression。这是一个功能强大的特性，但容易误用；因为它会破坏类型安全，会导致错误——特别是在将软件移植到新的平台的时候。虽然 C++标准为此引入了几个关键字，允许执行类型转换但不丢失所有的类型含义，但在早于这一特性的编译器中，这些特性没有得以实现。所以，ACE 提供了一组宏，以支持代码的可移植性。在支持新的类型转换特性的地方，它们会利用新特性；在需要使用旧式 C 风格类型转换的地方，它们会使用旧式类型转换。ACE 提供的最常用的类型转换如下表所示：

ACE 类型转换宏	可用的 C++类型转换
ACE_const_cast (TYPE, EXPR)	const_cast<TYPE> (EXPR)
ACE_static_cast (TYPE, EXPR)	static_cast<TYPE> (EXPR)
ACE_dynamic_cast(TYPE, EXPR)	dynamic_cast<TYPE> (EXPR)
ACE_reinterpret_cast (TYPE, EXPR)	reinterpret_cast<TYPR> (EXPR)

```
int run_worker (int argc, char *argv[]) {
  ACE_HANDLE socket_handle =
    ACE_reinterpret_cast (ACE_HANDLE, atoi (argv[2]));
  ACE_SOCK_Stream logging_peer (socket_handle);
  handle_data (&logging_peer);
  logging_peer.close ();
  return 0;
}
```

主服务器进程被动侦听新的客户日志连接。和前面的日志服务器一样，handle_connections()方法用来接受新连接。但是，在“一个连接一个进程”日志服务器中，handle_connections()会创建一个新的工作者进程来处理每一个新的客户连接。在图 8.6 中，我们演示了 handle_connections()在 POSIX 上接受一个新连接并创建一个工作者进程时发生的事件顺序。图 8.7 则演示了 Win32 上相同的事件顺序（在这两幅图中，为了适应页面的排版空间， Process_Per_Connection_Logging_Server 被缩写为

Logging_Server）。在图 8.6、图 8.7 及其说明中，我们都提到了将要介绍的 Logging_Process 类。在副栏 18 中，我们讲述了这两个图中的每一个步骤；而在下面的 C++代码段中，我们演示的是如何用 ACE_Process_Manager 来实现这些步骤。

以下是 Process_Per_Connection_Logging_Server 中 handle_connections()方法的实现：

```
1 virtual int handle_connections () {
2   ACE_SOCK_Stream logging_peer;
3   if (acceptor ().accept (logging_peer) == -1)
4     return -1;
5
6   Logging_Process *logger =
7     new Logging_Process (prog_name_, logging_peer);
8   ACE_Process_Options options;
9   pid_t pid = ACE_Process_Manager::instance ()->spawn
10    (logger, options);
11  if (pid == 0) {
12    acceptor().close ();
13    handle_data (&logging_peer);
14    delete logger;
15    ACE_OS::exit (0);
16  }
17  logging_peer.close ();
18  if (pid == -1)
19    ACE_ERROR_RETURN ((LM_ERROR, "%p\n", "spawn()"), -1);
20
21  return ACE_Process_Manager::instance ()->wait
22    (0, ACE_Time_Value::zero);
23 }
```

下面，我们来剖析 handle_connections()方法的实现，请参考第 174 页图 8.6 和图 8.7 中的各个步骤（各步骤的说明见副栏 18）。

第 2~4 行（第 1 步）调用 acceptor().accept()。此调用会被阻塞，直至有客户连接。

第 6~10 行（第 2~4 步）创建一个新进程，用来处理已接受的客户连接。“为每个平台正确设置选项”的工作在 Logging_Process::prepare()方法（将在第 180 页介绍）中完成。ACE_Process::spawn()调用合适的平台机制，以创建新进程。

副栏 18：如何创建工作者进程

在图 8.6 和图 8.7 中，那些带编号的步骤在两个图中是类似的，它们发生在 handle_connections()方法中：

第 1 步：接受一个新的日志客户连接。

第 2 步：调用 ACE_Process_Manager::spawn()，开始创建新的工作者进程。它会调用 ACE_Process::spawn()。

第 3 步：ACE_Process::spawn()回调 Logging_Server::prepare()方法，设置新进程的属性。

第 4 步：ACE_Process::spawn()方法调用 OS 平台的“进程创建”机制。例如，在 POSIX 上，fork()函数会创建工作者进程，使其成为主进程的完整副本。新进程的执行始于 handle_connections()方法中 fork()返回的位置。但在 Win32 中，创建新进程的是 CreateProcess()函数，并且没有复制主进程。新进程为新客户继承了 socket 句柄，如第 181 页将介绍的 Logging_Process::prepare()方法所示。新进程和主进程一样，运行的是同一程序映像，但用的是命令行上传递的新句柄值。

第 5 步：日志对等端（logging peer）的 socket 句柄在主进程中关闭，但工作者进程的副本还是继续打开，为日志客户提供服务。

第 5w 步：在工作者进程中，hook 方法 handle_data()会被调用。在 POSIX 上，它是在 handle_connections() 方法中直接调用的。在 Win32 上，Process_Per_Connection_Logging_Server::run()方法会注意到命令行上的句柄，并调用 run_worker()方法，后者则调用 handle_data()。当客户连接关闭时，工作者进程退出。

第 6 步：主进程周期性地调用 ACE_Process_Manager::wait()，检查工作者进程是否退出。

第 7 步：当工作者进程退出时，ACE_Process_Manager::wait()调用 Logging_Process::unmanage()。这给了主进程一个机会，来释放任何“和工作者进程有关”的资源。然后，Logging_Process 对象删除自身。

第 11~15 行（POSIX，第 5w 步[1]）在 Win32 中，spawn()不可能返回 0，因为它要么返回新进程的 PID，要么在出错时返回–1。因此，这几行代码一定只会在通过 fork()创建的 POSIX 进程环境中执行。工作者进程会关闭继承的 acceptor 对象，因为工作者进程只使用已建立的客户连接，不接受新连接。调用 handle_data()方法处理完所有的客户日志记录之后，Logging_Process 对象被释放，工作者进程退出。当主进程随后调用第 21 行的 ACE_Process_Manager::wait()时，它会注意到一个工作者进程已经退出。

第 17 行 如果 spawn()调用在父进程中返回（Win32 中总会是这样的情况），就不再需要 logging_peer 对象，因而会被关闭。但实际的 TCP 连接不会被关闭，因为 ACE“进程”wrapper facade 封装了以下“和平台相关”的行为：

• 在 POSIX 上，工作者进程继承了整个对象的副本，包括打开的句柄。OS 对这些句柄实施引用计数，因而连接不会被真正关闭，除非主进程和工作者进程都各自关闭了此句柄的副本。

• 在 Windows NT/2000 上，句柄的管理和在 POSIX 上类似；也就是说，只有在两个进程都关闭了句柄之后，连接才会关闭。

• Windows 95/98 没有对句柄自动实施引用计数。但是，Logging_Process::prepare()方法调用了 ACE_Process_Options::pass_handle()，后者复制了 logging_peer 中包含的 socket 句柄。所以，父进程可以安全地关闭它的句柄但不影响连接。

和以前一样，在这里，有了 ACE wrapper facade，应用程序开发者不必关心每一个 OS 平台上这些微妙的差异！

第 21~22 行（第 6、7 步）ACE_Process_Manager::wait()方法检查是否有任何工作者进程退出，并获取它们的状态，清除它们打开的任何句柄。

下面给出的 handle_data()方法在所有平台上完全一样。它将客户 socket 置于阻塞状态，打开日志文件以保留日志记录，并处理日志记录，直至日志客户关闭这个 socket 或有错误发生。最后，关闭日志文件。

---

[1] 在 Win32 中，第 5w 步由 run_worker()方法执行，如第 176 页所示。

```
virtual int handle_data (ACE_SOCK_Stream *logging_peer) {
  // Ensure blocking <recv>s.
  logging_peer->disable (ACE_NONBLOCK);
  ACE_FILE_IO log_file;
  make_log_file (log_file, logging_peer);
  Logging_Handler logging_handler (log_file,
                                   *logging_peer);

  while (logging_handler.log_record () != -1)
    continue;

  log_file.close ();
  return 0;
}
```

Logging_Process 类。为了正确设定新工作者进程，我们定义了一个 Logging_Process 类，它派生自第 8.2 节介绍的 ACE_Process 类。设定上的需求在 OS 平台和应用程序之间往往不同，所以，ACE_Process 提供了 prepare()和 unmanage()这两个 hook 方法。Logging_Process 类会通过 prepare()方法，将新日志客户的 socket 句柄传递给工作者进程。如果我们以后决定不在 POSIX 上运行新程序映像，需要的任何修改都可以集中在这里。

```
class Logging_Process : public ACE_Process
{
private:
  Logging_Process (); // Force desired constructor to be used.

  char prog_name_[MAXPATHLEN + 1];
  ACE_SOCK_Stream logging_peer_;

public:
  Logging_Process (const char *prog_name,
                   const ACE_SOCK_Stream &logging_peer)
    : logging_peer_ (logging_peer.get_handle ())
  { strcpy (prog_name_, prog_name); }
```

设定新工作者进程所需的参数被传递给 Logging_Process 类的构造函数。它们包括：用来创建新进程的命令的名称，工作者进程要用到的日志客户的 ACE_SOCK_Stream。这两个参数用在下面的 prepare() hook 函数中，此函数由 ACE_Process::spawn()在新进程被创建之前调用。

```
virtual int prepare (ACE_Process_Options &options) {
  if (options.pass_handle (logging_peer_.get_handle ()) == -1)
    ACE_ERROR_RETURN ((LM_ERROR, "%p\n", "pass_handle()"), -1);
  options.command_line ("%s", prog_name_);
  options.avoid_zombies (1);
  options.creation_flags (ACE_Process_Options::NO_EXEC);
  return 0;
}
```

prepare()方法在图 8.6 和图 8.7 的第 3 步演示。它仅有的一个参数是 ACE_Process_Options 对象的引用，ACE_Process::spawn()会利用这个对象来创建新进程。这给了 prepare()一个“根据需要修改或增加选项”的机会。我们像下面这样通过 prepare()设置选项：

- **传递日志客户的 socket 句柄**。为了正确得到工作者进程的 socket 句柄，ACE_Process 和 ACE_Process_Options 会在内部处理“和平台相关”的所有细节，包括在 Windows 95/98 上复制它，还要记得在进程退出时关闭这个复制的句柄。我们通过 pass_handle()方法，将这个句柄值传给工作者进程的命令行，表明某一日志客户需要服务。

- **设定程序名**。POSIX 的 fork()调用需要它，运行新程序映像时也会需要它。

- **避免“僵死进程”**。只有 POSIX 需要这个标志，Win32 会忽略它。

- **设置 NO_EXEC 标志**，使得 POSIX 系统只（通过 fork()）创建新工作者进程，但不（通过 exec()）执行新程序映像。此标志在 Win32 中也会被忽略。

尽管有些方法在 Win32 上没有效果，但还是会调用它们；这样，在所有支持多进程的 ACE 平台上，“设置工作者进程”的步骤就都具有可移植性。关于这“为什么是”（以及“什么时候不是”）一个正确设计的理由，请参见第 A.5 节的讨论。

为了确定进程何时退出、何时对“底层进程已经退出”的任何进程对象调用 unmanage() hook 方法，需要了解一些“和平台相关”的信息；而 ACE_Process_Manager 封装了这些信息。Logging_Process::unmanage()即第 174 页图 8.6 和图 8.7 的第 7 步，代码如下：

```
virtual void unmanage () {
  delete this;
}
```

它简单地清除一个 Logging_Process 对象；此对象在“日志客户连接”被接受时动态分配。但在 Windows 95/98 上，对等端（peer）socket 句柄在被传递给工作者进程之前，可能已被复制。通过在 ACE_Process 类中对“行为”和“状态”进行封装，那个句柄会在 ACE_Process 的析构函数中被关闭；所以，在 ACE 支持的所有平台上，我们的“清除”代码也是可移植的。

最后给出 main()程序；和前面那些服务器相比，它有了一点扩充。

```
static void sigterm_handler (int /* signum */) { /* No-op. */ }

int main (int argc, char *argv[])
{
  // Register to receive the <SIGTERM> signal.
  ACE_Sig_Action sa (sigterm_handler, SIGTERM);

  Process_Per_Connection_Logging_Server server;

  if (server.run (argc, argv) == -1 && errno != EINTR)
    ACE_ERROR_RETURN ((LM_ERROR, "%p\n", "server.run()"), 1);

  // Barrier synchronization.
  return ACE_Process_Manager::instance ()->wait ();
}
```

ACE_Sig_Action 类注册（register）这个进程，使之处理 SIGTERM 信号；管理员可以用这个信号来关闭“父服务器进程”。父进程在退出之前会调用 ACE_Process_Manager::wait()；这样，在父进程关闭自身之前，它可以和所有“工作者日志进程”的“退出”保持同步。如果父进程需要将最后的时间标记（time stamp）写入日志输出设备或文件，这种“*栅栏同步*（barrier synchronization）”能力将很有用。

8.5 小结

很多“多进程”操作系统都是在专有系统（proprietary system）方兴未艾之时开发的，所以，这些系统都开发了自己专有的“进程管理”机制。保持 OS 版本之间的兼容性很重要；所以，这些专有的进程机制和 API 大多都延续至今。

本章讲述如何运用 ACE 的“进程管理”能力，使得我们可以在广泛的平台上可移植地编写“多进程”服务器。我们将 ACE_Process_Manager 应用到一个新的日志服务器实现中，使得这个服务器可以将多个“新创建的进程”组合起来。这种组合允许主进程在关闭自身之前等待所有工作者进程退出。

OS“进程管理”存在语义上的内在差异；所以，网络程序开发者必须谨慎地评估系统的并发需求。在第 7 章，我们讨论的重点是如何消除“由于低级 API 导致”的偶发复杂性；在本章，我们的讨论延伸到“多平台软件开发”课题。本章不仅向我们展示可移植的 ACE wrapper facade 如何有助于编写“多进程”应用程序，还清楚地告诉我们，在评估“多进程”并发设计时，必须细致地考虑到应用程序目前和将来的平台所具有的能力。例如，ACE“进程”wrapper facade 可以让应用程序避免 OS 上的很多细节；但是，在根本就不支持进程机制的平台上，它们无论如何也无法提供进程抽象。

第 9 章

ACE 线程 Wrapper Facade

本章提要

在本章讲述的 ACE wrapper facade 中，我们通过可移植的 C++类，对 OS 多线程（multithreading）机制进行了封装。运用这些 ACE“线程”wrapper facade，加强了网络日志服务程序，并使得它更趋灵活、高效。

9.1 概述

很多网络应用程序天生适合多线程，因为“处理任务（processing task）”可以和“I/O 任务”分开。如果服务器需要同时为很多客户管理“面向连接”或“无连接”网络联系，那么，使用多线程将十分方便。今天，OS 对多线程提供了日益强大的支持，有助于应用程序——

- **利用硬件技术的进步**。例如，可以提供真正*并行（parallelism）*执行能力的对称多处理（symmetric multiprocessing）。
- 通过重叠（overlapping）计算和通信**提高性能**。
- 为 GUI 和网络服务器**改善响应时间**，确保在需要的时候可以对“时间紧迫”任务实施调度。

• 使用直观的“同步”编程机制，而不是更复杂的“异步”编程机制，从而**简化程序结构**。

本章将讲述以下 ACE 类；利用这些类，网络应用程序可以在一个进程中创建、管理一个或多个控制线程（threads of control）：

ACE 类	说 明
ACE_Thread_Manager	使得应用程序能够可移植地创建和管理线程的生存期、同步和属性。
ACE_Sched_Params	可移植地封装了 OS“调度”类的特性（feature）；可以结合 ACE_OS::sched_params() 包装器方法一起使用，以控制实时线程的各种属性。
ACE_TSS	封装了“线程专有存储”机制，使得在“物理”上“私有”于某一线程的对象可以被全局访问，就像它们在“逻辑”上“公有”于整个程序一样。

这些 wrapper facade 带来了以下好处：

• **确保异种 OS 平台间的可移植性**。操作系统具有各式各样的“多线程”语法和语义，ACE 通过 wrapper facade 隐藏了这种多样性。例如，无论底层是何种 OS 线程机制，ACE_Thread_Manager 类可以确保“分离（detached）”/“可连接（joinable）”线程具有相同的行为。

• **将一组线程作为一个有机的集合来管理**。在并发式网络应用程序中，多个线程经常需要像一个整体那样开始和结束。ACE_Thread_Manager 类提供了“线程组”能力，允许其他线程在继续执行自己的处理之前，先要等待整组线程退出。

• **管理线程的调度策略和优先级**。在很多网络应用程序中，需要对线程的优先级实施严格控制。在调度能力上，“实时”和“通用”操作系统往往有差异；而且，“多平台”网络应用程序都需要一种方式，用来可移植地管理完全不同的调度属性。之所以设计 ACE_Sched_Params 类，就是为了通过统一、可移植、易于使用的方式访问运行时调度信息。

• **高效地使用和管理“线程专有存储空间（TSS, thread-specific storage）”**。TSS是一种机制，它允许多个对象和方法管理每一个线程的专有信息。ACE 的 TSS wrapper façade 类克服了各个平台间的语法差异和能力差异。

本章提出并讲述了 ACE“多线程”类的能力。我们为每一个类都提供了示例，演示它们如何简化和优化网络日志服务程序的各个方面。

9.2 ACE_Thread_Manager 类

设计意图

不同的操作系统使用不同的 API 来创建、管理线程，以及执行“线程结束时的同步工作”。今天的“多线程”机制仍然受偶发复杂性的困扰，就像前面章节所讨论的那样。此外，“多线程”机制还引入了以下两种“不确定性”，使得我们很难编写出可移植的应用程序：

• **语法（syntactic）**——原始的 OS 线程 API 往往存在语法上的差异，因而不具可移植性——即使它们提供的功能相同。例如，Win32 的 CreateThread()和 Pthreads 的 pthread_create()函数提供了相似的线程创建功能，但它们的 API 在语法上有差异。

• **语义（semantic）**——wrapper facade 即使在语法上提供统一的 C++编程接口，也不一定能够解决各个 OS 线程机制在语义上的多样性。例如，Pthreads 和 UI 线程都支持“分离线程（detached thread）”，但 Win32 不支持，而 VxWorks 则“仅”支持“分离线程”。

在多线程应用程序中，一个令人苦恼的问题是：应当如何可移植地“删除（cancel）”线程。一些 OS 平台本身已经支持“线程删除”机制，例如：

• Pthreads 定义了一组功能强大的 API，允许线程通过 pthread_cancel()和 pthread_testcancel()函数而被“异步（asynchronously）”删除，或被“协同（cooperatively）”删除。

• 在某些 UNIX 平台上，还有可能通过信号（signal）来删除线程——例如，通过 Pthreads 的 pthread_kill()和 UI 线程的 thr_kill()函数。

• Win32 的 Terminate Thread()函数支持“异步删除”。

遗憾的是，以上归纳的原始 OS“线程删除”机制不具可移植性，还容易引发错误。例如，UNIX 信号和 Win32 的 Terminate Thread()函数会使线程在执行过程中“骤死”，从而无法释放它们拥有的资源。Pthreads 的“异步线程删除”机制对此提供了更好的支持，它们允许清除线程拥有的资源；但是，它们还是让人难以理解、难以正确编写、无法在非 Pthreads 平台上移植。

如果在每一个应用程序中都需要解决这些可移植性问题，那将十分繁琐，而且容易引发错误。所以，ACE 提供了 ACE_Thread_Manager 类。

功能

利用 Wrapper Facade 模式，ACE_Thread_Manager 类封装了不同 OS“多线程”API 在语法和语义上的多样性。这个类提供了以下颇具可移植性的功能：

• 一次创建一个线程或多个线程，每一个线程并发（concurrently）执行一个“应用程序指定”的函数。

• 针对每一个被创建线程，改变其最常见的线程属性，如调度优先级、栈的大小等。

• 创建并管理一组线程，将其视为一个“有机的集合”——称之为“线程组（thread group）”。

• 管理 ACE_Task 中的线程（ACE_Task 的介绍见 [SH]）。

• 实现线程的*协同删除（cooperative cancellation）*能力。

• 等待一个或多个线程退出。

ACE_Thread_Manager 类的接口如图 9.1 所示，其“和平台无关”的主要方法见下表：

方　法	说　明
spawn()	创建一个新的控制线程，并传递给它一个“函数”和“函数的参数”，作为线程执行的入口点。
spawn_n()	创建 n 个“同属一个线程组”的新线程，其他线程会等待这整组线程退出。
wait()	阻塞，直至线程管理器中的所有线程都已退出，并获得所有“可连接”线程的退出状态为止。
join()	等待某一线程退出，并获得其退出状态。
cancel_all()	请求 ACE_Thread_Manager 对象管理的线程全部退出。
testcancel()	询问某一线程是否已被请求退出。
exit()	退出一个线程，并释放这个线程的资源。
close()	关闭并释放所有被管理线程的资源。
instance()	这是一个静态方法，返回一个指向 ACE_Thread_Manager singleton 的指针。

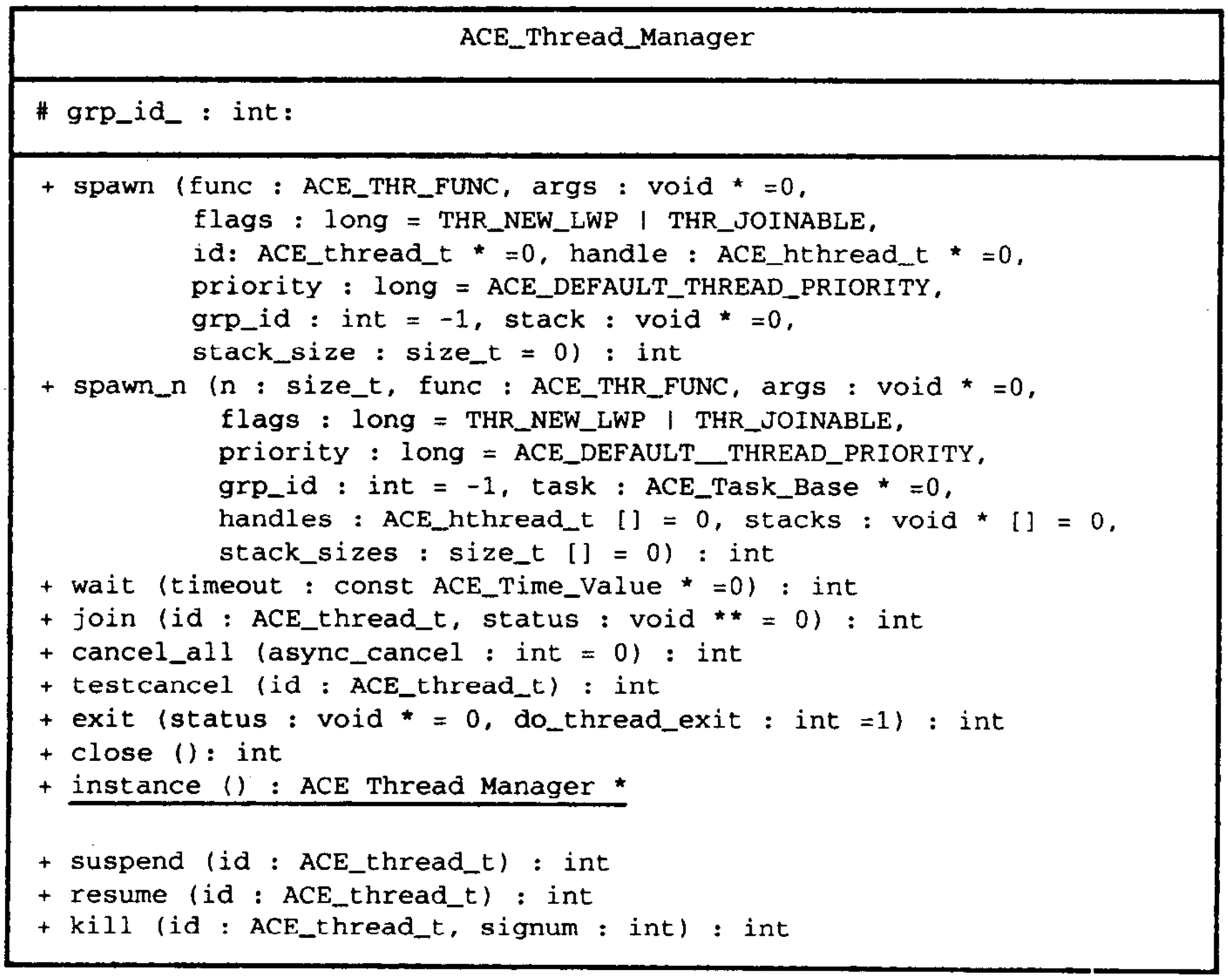


图 9.1：ACE_Thread_Manager 类图

我们可以向 ACE_Thread_Manager::spawn()方法传递一组标志，用以指定“被创建线程”的属性。它的值是以下标志按位“或”的结果：

标　志	说　明
THR_SCOPE_SYSTEM	新线程在*系统*调度争用空间中创建，永久绑定于新创建的核心线程。
THR_SCOPE_PROCESS	新线程在*进程*调度争用空间中创建，也就是说，它将作为一个用户线程运行。
THR_NEW_LWP	此标志影响进程的并发属性。对“未绑定线程”来说，其预想的并发级别是增 1，也就是添加一个新的核心线程到可用的线程池中，以运行用户线程。在不支持 N:M 用户/核心线程模型的 OS 平台上，此标志会被忽略。
THR_DETACHED	创建“分离”线程；这意味着它的退出状态不能被其他线程访问。当这个线程终止时，其线程 ID 和其他资源会立即被 OS 回收。此标志和 THR_JOINABLE 相反。
THR_JOINABLE	创建“可连接”线程；这意味着其他线程可以通过 ACE_Thread_Manager::join()方法得到它的退出状态。它的线程 ID 和其他资源不会被 OS 回收，除非另一个线程和它连接（join）。这个标志和 THR_DETACHED 相反。所有 ACE 线程创建方法的缺省行为都是 THR_JOINABLE。

ACE_Thread_Manager 不仅可以创建具有各种属性的线程，而且提供了安全、易于使用、可移植的“协同式线程删除”机制。为使用这种机制，删除线程要通过 ACE_Thread_Manager::cancel()设置一个标志，指示某一线程应该删除自身。在这一方案中，被删除方负责合作，周期性地调用 ACE_Thread_Manager::testcancel()，检查自己是否被请求删除。

线程是不会被立即删除的；所以，ACE 的“协同式线程删除”特性类似于使用了 Pthreads 的*延迟删除（deferred cancelation）*策略，删除点（cancellation point）为 ACE_Thread_Manager::testcancle()调用。ACE 的“协同式线程删除”同 Pthreads 的“延迟删除”在以下方面有所不同：

• 在 ACE 中，线程清理代码必须显式写在删除点之后。

• 在 ACE 中，当一个线程被删除之后，它可以选择结束“正在进行”的任何工作，或者完全忽略“删除请求”。

因此，ACE 的线程删除完全是主动的，这是可移植地、安全地删除一个线程的惟一方式。

和第 8.4 节介绍的 ACE_Process_Manager 一样，可以通过两种方式使用 ACE_Thread_Manager：

• **用作 singleton** [GHJV95]。这要用到它的 instance()方法。这个方法根据 Double-Checked Locking Optimization 模式实现，如副栏 19 所述。

• **实例化一个或多个实例**。此功能用于在一个进程中支持多个线程组。任何时候这都是一种合法的使用方式，但它最可取的使用时机是：在通过 ACE_Thread_Manager 类开发*共享库（DLL）*的时候。这样做，可以避免同“调用者程序”正在使用的 singleton 实例相冲突。

示例

在创建“线程”比创建“进程”开销小的操作系统上，“多线程”服务器很常用。在下面的例子中，我们使用 ACE_Thread_Manager，并基于“一个连接一个线程”的并发模式，实现了我们的第一个“多线程”日志服务器。如图 9.2 所示，主线程持续运行，扮演“工厂”的角色：

1. 接受一个连接，并动态创建一个 ACE_SOCK_Stream 对象。

2. 创建一个工作者线程，利用这个对象来处理客户的日志会话。

工作者线程在 ACE_SOCK_Stream 上完成后续所有“日志记录”处理，并在连接关闭时销毁它。第 8.4 节有一个“为每一个客户连接都创建一个新进程”的例子；这里的并发设计与之类似。但是，这种“一个连接一个线程”的日志服务器同“一个连接一个进程”的实现还是有所不同的，这主要表现在以下几个方面：

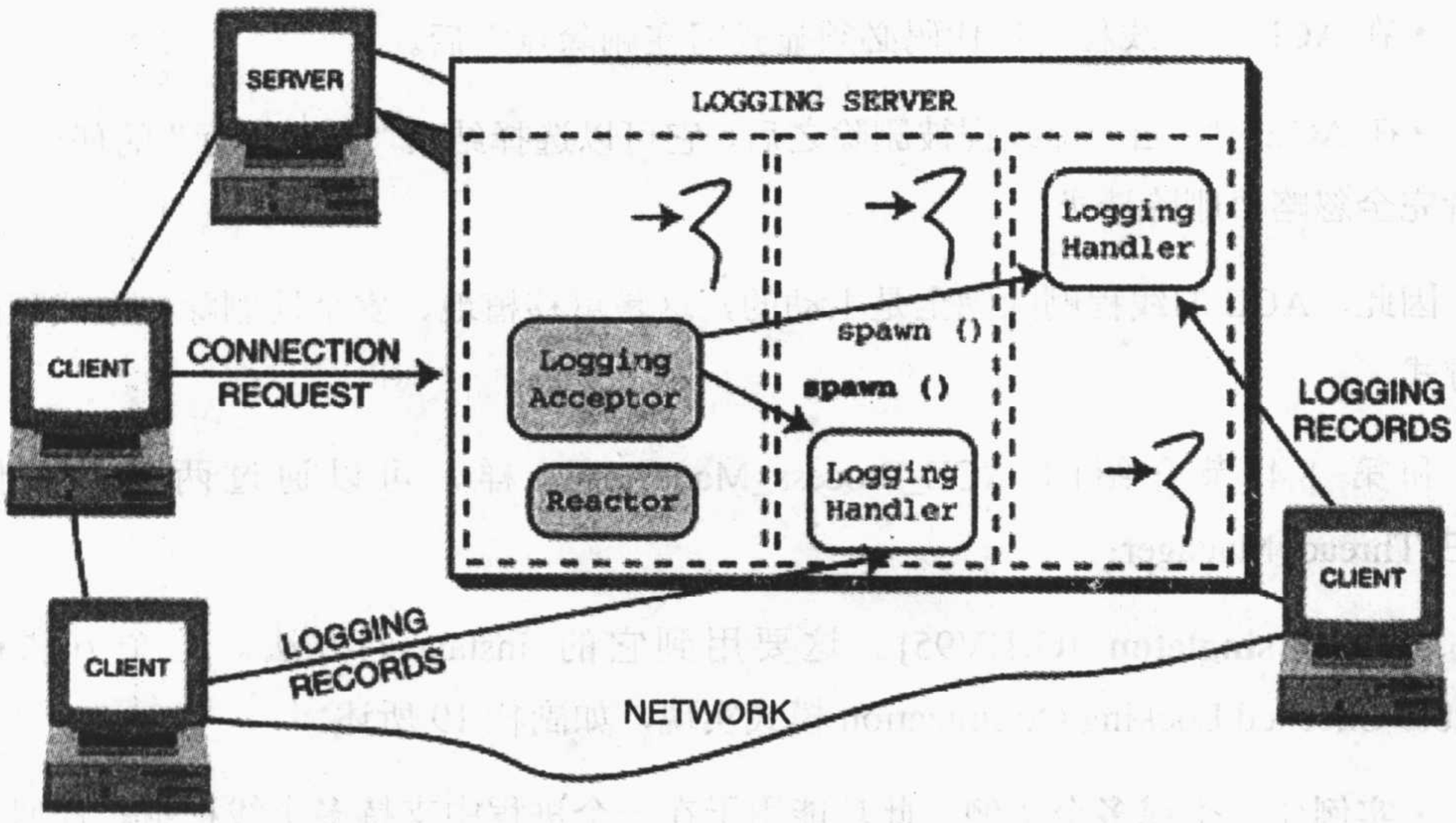


图 9.2：“一个连接一个线程”日志服务器的结构

副栏 19：在 ACE 中串行处理 singleton

Singleton 模式 [GHJV95]确保一个类只有一个实例，并提供一个全局点（global point）来访问它。ACE 中定义的 singleton 使用了 Double-Checked Locking Optimization（双检测锁定优化）模式 [SSRB00]——在程序执行期间，如果代码关键段必须以“线程安全”的方式获得锁（lock），而且只需一次，那么，使用这一模式可以降低“竞争”和“同步”的开销。在下面的代码中，我们演示静态 ACE_Thread_Manager::instance()方法如何使用这一模式：

```
ACE_Thread_Manager *ACE_Thread_Manager::instance () {
  if (ACE_Thread_Manager::thr_mgr_ == 0) {
    ACE_GUARD_RETURN (ACE_Recursive_Thread_Mutex, ace_mon,
                      *ACE_Static_Object_Lock::instance(),
                      0));
    if (ACE_Thread_Manager::thr_mgr_ == 0)
      ACE_Thread_Manager::thr_mgr_ = new ACE_Thread_Manager;
  }
  return ACE_Thread_Manager::thr_mgr_;
}
```

ACE_Static_Object_Lock::instance()返回一个预先初始化了的锁，这个锁在 main()函数运行之前就已经创建，请参阅副栏 23 的说明。

• 它为每一个连接创建一个新“线程”，而不是新“进程”，因而效率更高。

• 它更具可移植性，因为在支持线程的任何平台上，它都可行。

• 它更为简单，因为编写可移植的“多线程”服务器比编写“多进程”服务器更容易。

通过“多线程”而不是“多进程”来实现服务器，既有优点也有缺点；第 5.2 节就此作了更全面的讨论。

和以前一样，我们的第一步工作是包含所需的 ACE 头文件：

```
#include "ace/SOCK_Stream.h"
#include "ace/Thread_Manager.h"
#include "Logging_Server.h"
#include "Logging_Handler.h"
```

我们定义一个 Thread_Per_Connection_Logging_Server 类，这个类由 Logging_Server 派生而来。

```
class Thread_Per_Connection_Logging_Server : public Logging_Server
{
private:
  class Thread_Args {
  public:
    Thread_Args (Thread_Per_Conenction_Logging_Server *lsp)
      : this_ (lsp) {}

    Thread_Per_Connection_Logging_Server *this_;
    ACE_SOCK_Stream logging_peer_;
  };

  // Passed as a parameter to <ACE_Thread_Manager::spawn>.
  static void *run_svc (void *arg);

protected:
  // Other methods shown below...
};
```

和第 8.4 节的 Process_Per_Connection_Logging_Server 一样，我们继承并复用（reuse）了 open()和 wait_for_multiple_events()方法。

副栏 20：“混淆线程和对象”存在的陷阱和缺陷

作为参数传递给线程入口点函数（entry point function）的对象，其生命周期必须小心管理。很多程序员混淆了线程和对象，说出“这个对象运行在一个线程中”之类的话来。区分以下概念很重要：

- 一个线程是一个执行单元（unit of execution）。
- 一个对象是一块内存和一组与之相关的方法（method）。

所以，一个线程和“这个线程在执行期间访问的任何对象”之间没有内在（implicit）连接。因而，确保一个对象被删除之后没有线程能够访问它，这一点十分重要。

举例来说，如果 Thread_Args 对象是在 for 循环范围中构造/销毁的，那么，在这个线程中，这个对象会被后续“客户连接”改写（overwritten）。所以，为了避免这些问题，我们可以动态分配 Thread_Args 对象，让 run_svc()函数在返回之前删除它们。

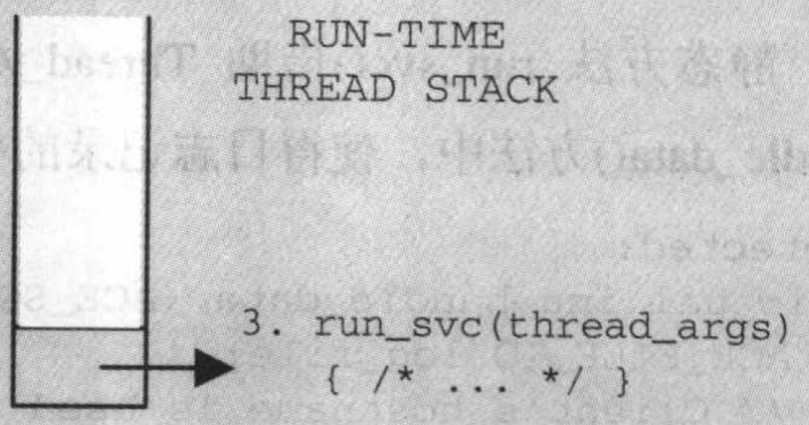
handle_connections()方法接受一个连接，并将这个连接放进 thread_args 对象的 ACE_SOCK_Stream 数据成员中（副栏 20 说明了为什么要动态分配 thread_args）。然后，通过 ACE_Thread_Manager singleton 创建一个新线程，用来处理新连接的客户。

```
virtual int handle_connections () {
  Thread_Args *thread_args = new Thread_Args (this);
  if (acceptor ().accept (thread_args->logging_peer_) == -1)
    return -1;
  else if (ACE_Thread_Manager::instance ()->spawn (
      // Pointer-to-function entry point.
      Thread_Per_Connection_Logging_Server::run_svc,
      // <run_svc> parameter.
      ACE_static_cast (void *, thread_args),
      THR_DETACHED | THR_SCOPE_SYSTEM) == -1)
    return -1;
  else
    return 0;
}
```

run_svc()静态方法（static method）是每个新线程的入口点函数（entry point function）。底层 OS“线程创建”函数会将 thread_args 指针传递给 run_svc()，用以控制 logging_peer 的生存期。因为不必为了获得这个线程的“退出状态”而和这个线程谋面，所以传递 THR_DETACHED 标志，命令 ACE_Thread_Manager 和“底层 OS 线程实现”在线程退出时立即释放资源。所有通过 ACE_Thread_Manager singleton 创建的线程都可以作为一个整体来管理。因此，可以等待它们的结束——即使在创建它们的时候，没有通过 THR_JOINABLE 标志将其指定为“可连接（joinable）”线程。副栏 21 演示如何通过这些标志来创建线程。

副栏 21：在 ACE 中如何创建线程

下图演示的是，在一个使用了 UI 线程的 thr_create()系统函数的平台上，执行 ACE_Thread_Manager::spawn()时发生的调用次序：




```
1. ACE_Thread_Manager::spawn
   (run_svc,
    thread_args,
    THR_DETACHED | THR_SCOPE_SYSTEM);
2. thr_create
   (0, 0,
    run_svc, thread_args,
    THR_DETACHED | THR_SCOPE_SYSTEM, &thread_id);
```

1. OS 创建一个“线程执行环境（thread execution context）”；

2. OS 为这个线程的栈（stack）分配内存；

3. 准备新线程的寄存器组（register set），这样，当新线程被调度执行时，它会调用“作为参数提供给 spawn()”的线程入口点函数；

4. 线程被标记为“可运行（runnable）”，于是 OS 可以启动其执行。

run_svc()方法是每一个新创建的线程的入口点，用来处理客户日志记录。以下给出了这个方法的代码。

```
void *Thread_Per_Connection_Logging_Server::run_svc (void *arg)
{
  Thread_Args *thread_args = ACE_static_cast (Thread_Args *, arg);

  thread_args->this_->handle_data (&thread_args->client);
  thread_args->logging_peer_.close ();
  delete thread_args;

  return 0; // Return value is ignored.
}
```

如副栏 21 所述，ACE_Thread_Manager::spawn()方法接受一个 Thread_Args 指针，供 run_svc()使用。这个指针必须被转换为 void *，以适用于 ACE 线程 API，从而保证在各个操作系统上的可移植性。run_svc()函数被 OS 线程库调用后，会将指针转换回 Thread_Arg *。ACE 的 ACE_static_cast()宏（见副栏 17）便于转换指针，并能做到编译器所能允许的最大安全性。请注意，我们是在“分离”线程中创建 run_svc()的，所以它的返回值会被忽略。

静态方法 run_svc()借助 Thread_Args 参数中的数据成员，将控制转回至下面的 handle_data()方法中，使得日志记录的处理工作可以继续进行。

```
protected:
  virtual int handle_data (ACE_SOCK_Stream * client) {
    ACE_FILE_IO log_file;
    // Client's hostname is used as the logfile name.
    make_log_file (log_file, logging_peer);

    // Place the connection into blocking mode.
    client->disable (ACE_NONBLOCK);

    Logging_Handler logging_handler (log_file, *logging_peer);
    ACE_Thread_Manager *tm = ACE_Thread_Manager::instance ();
    ACE_thread_t me = ACE_OS::thr_self ();

    // Keep handling log records until client closes connection
    // or this thread is asked to cancel itself.
    while (!tm->testcancel (me)
           && logging_handler.log_record () != -1)
      continue;

    log_file.close ();
    return 0;
  }
```

这个版本的 handle_data()类似于第 179 页“一个连接一个进程”的例子。但在这个版本中，可以请求“正在处理客户日志记录”的 handle_data()线程删除自身；这是“*协同式删除(cooperative cancelation)*”的一个例子。为了做到“协同”，每一个 handle_data()线程在处理一个日志记录之前，都会调用 ACE_Thread_Manager::testcancle()，检查主线程是否已经请求了删除。

“testcancel()调用的存在”和“Logging_Handler::log_record()方法的设计”有一个缺点：只有在接收到下一个日志记录之后，线程才会注意到“删除请求”。如果客户日志记录的发送十分频繁，这可能不是问题。在[SH]中，我们介绍的 ACE Reactor 和 Task 框架（framework）可用来避免这一缺点。

最后，main()程序在它的 run()模板方法（template method）中执行日志服务器的事件循环。

```
int main (int argc, char *argv[])
{
  // Register to receive the <SIGTERM> signal.
  ACE_Sig_Action sa (sigterm_handler, SIGTERM);

  Thread_Per_Connection_Logging_Server server;

  if (server.run (argc, argv) == -1)
    ACE_ERROR_RETURN ((LM_ERROR, "%p\n", "server.run()"), 1);

  // Cooperative thread cancelation.
  ACE_Thread_Manager::instance ()->cancel_all ();

  // Barrier synchronization, wait no more than a minute.
  ACE_Time_Value timeout (60);
  return ACE_Thread_Manager::instance ()->wait (&timeout);
}
```

在这个“一个连接一个线程”日志服务器实现中，服务器会捕捉 SIGTERM 信号，使得系统管理员可以关闭（shut down）它。但是，主线程不是无限期等待所有日志客户终止会话，而是通过 ACE_Thread_Manager 的“协同删除”机制，请求 ACE_Thread_Manager singleton 创建的所有服务线程关闭。

main()的最后一行是一个 wait()调用，它允许日志服务器等待一分钟时间，以和所有“被删除线程”的“完成（completion）”同步。这是*栅栏同步(barrier synchronization)*

的一个例子。在一些 OS 平台上，如果主线程退出而其他线程还在运行，则会导致意想不到的行为；“栅栏同步”可以防止发生这些问题。此外，将等待时间通过 timeout 参数进行限制，日志服务器就一定不会被无限期挂起。

9.3 ACE_Sched_Params 类

设计意图

Thread_Per_Connection_Logging_Server 类创建的所有线程都运行在相同的优先级（priority）上；这个优先级的缺省值也就是预定义（predefined）的 ACE_DEFAULT_THREAD_PRIORITY 宏的值。一些网络应用程序，特别是有“实时（real-time）”要求的应用程序，需要对线程的优先级实施更多的控制。在 ACE 中，可以通过向 ACE_Thread_Manager::spawn() 或 spawn_n()方法传递一个参数，为新创建的线程指定优先级。但是，这个方法有两个问题：

- **不具可移植性（non-potable）**——不同的 OS 平台以不同的方式表示优先级。例如，在一些操作系统中，高优先级具有较高的数值；但在另一些操作系统中，高优先级具有较低的值。

- **不具实时性（non-real-time）**——要使一个线程能够运行在第 5.5 节所述的“实时调度”级别中，大多数操作系统都需要额外（而且非标准）的措施。

如果需要在每一个应用程序中处理这些问题，那将十分麻烦，而且容易引发错误；正因为如此，ACE 提供了 ACE_Sched_Params wrapper facade 类和 ACE_OS::sched_params() 包装器方法（wrapper method）。

功能

ACE_Sched_Params 类基于 Wrapper Facade 模式，封装了第 5.5 节所述的 OS 调度级（OS scheduling class）API。这个类可以和 ACE_OS::sched_params()方法一起使用，以提供以下功能：

- 通过一种可移植的方式，指定调度策略（scheduling policy），如“先进先出（FIFO）”、“轮询（round-robin）”。

• 提供某种方式，为“轮询（round-robin）”调度策略指定“分时量（time-slice quantum）”。

• 提供某种方式，指定策略运用的范围。例如，是运用在当前“进程”还是当前“线程”上。

• 提供一致的“调度优先级”表示方式。这很必要；因为在一些 OS 平台上，较高调度优先级是用较低优先级值来表示的。

ACE_OS::Sched_params()方法封装了“用来操纵调度级别”、“和 OS 相关”的函数；ACE_Sched_Params 提供了这个方法用到的值。

ACE_Sched_Params 类的接口如图 9.3 所示，其主要方法见下表：

方　法	说　明
ACE_Sched_Params()	设置实时线程的调度策略、优先级、范围（scope）和量（quantum）。
priority_min()	返回某一调度策略和范围的最低优先级。
priority_max()	返回某一调度策略和范围的最高优先级。
next_priority()	给定一个策略、优先级和范围，返回下一个较高的优先级，其中“较高”指的是调度优先级，而不是优先级的值。如果给定的优先级已经是指定策略的最高优先级，那么返回的就是这个优先级本身。
previous_priority()	给定一个策略、优先级和范围，返回下一个较低的优先级，其中“较低”指的是调度优先级，而不是优先级的值。如果给定的优先级已经是指定策略的最低优先级，那么返回的就是这个优先级本身。

在 main()中，ACE_OS::sched_params()方法一般是在“还没有任何线程创建”的时候调用的。因为“被创建线程”会继承“父线程”的调度级别和优先级别，所以，这个调用设置的是缺省的基本优先级（base priority）。在通过 ACE_Thread_Manager::spawn()方法创建线程的时候，或是通过 ACE_OS::thr_prio()调整优先级的时候，线程的优先级可以单独指定。

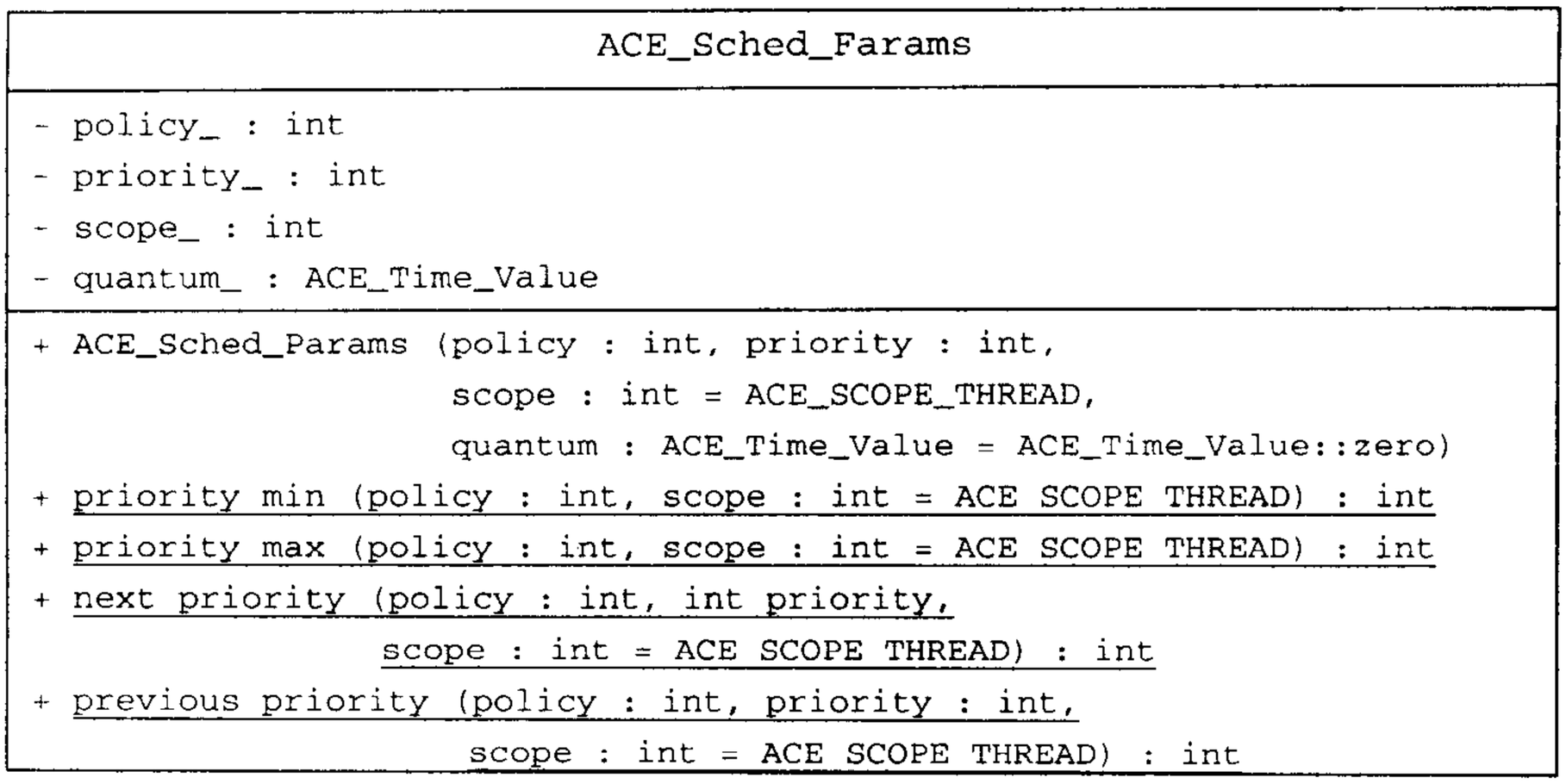


图 9.3：ACE_Sched_Params 类图

示例

为演示如何通过 ACE 可移植地编写 OS“实时调度级别”功能，我们将修改第 9.2 节的“一个连接一个线程”日志服务器示例。在对网络日志服务程序的使用情况作详细的经验分析之后，我们决定给予“已连接的客户”（而不是新的连接请求）更高的优先级。因此，较之将要创建的、用来“接收和处理日志记录”的线程，用来“接受连接”的线程将运行在较低的实时优先级上。

在 RT_Thread_Per_Connection_Logging_Server.h 文件中，我们通过继承 Thread_Per_Connection_Logging_Server 类，创建了一个网络日志服务器；此服务器的线程将运行在“实时 OS 调度级别”上。

```
class RT_Thread_Per_Connection_Logging_Server
  : public Thread_Per_Connection_Logging_Server
{
public:
```

接着，我们重写 hook 方法 open()，为这个进程中创建的所有线程实施 FIFO 调度。缺省情况下，被创建线程会运行在最低的“实时”线程优先级上。如果应用程序缺乏足够的权限，则会退而运行在“分时”调度级别上。

```
virtual int open (u_short port) {
  ACE_Sched_Params fifo_sched_params
    (ACE_SCHED_FIFO,
     ACE_Sched_Params::priority_min (ACE_SCHED_FIFO),
     ACE_SCOPE_PROCESS);

  if (ACE_OS::sched_params (fifo_sched_params) == -1) {
    if (errno == EPERM || errno == ENOTSUP)
      ACE_DEBUG ((LM_DEBUG,
                  "Warning: user's not superuser, so "
                  "we'll run in the time-shared class\n"));
    else
      ACE_ERROR_RETURN ((LM_ERROR,
                         "%p\n", "ACE_OS::sched_params()"), -1);
  }
  // Initialize the parent classes.
  return Thread_Per_Connection_Logging_Server::open (port);
}
```

最后重写 handle_data()方法。对每一个处理（来自“已连接客户”的）日志记录的线程，我们都提高它的优先级。

```
virtual int handle_data (ACE_SOCK_Stream *logging_client) {
  int prio =
    ACE_Sched_Params::next_priority
      (ACE_SCHED_FIFO,
       ACE_Sched_Params::priority_min (ACE_SCHED_FIFO),
       ACE_SCOPE_THREAD);
  ACE_OS::thr_setprio (prio);
  return Thread_Per_Connection_Logging_Server::handle_data
          (logging_client);
}
```

通过运用“面向对象”设计、C++特性和 ACE wrapper facade，可以为“一个连接一个线程”日志服务器增加“实时调度级”行为，但无需修改任何已有的类，也无需编写任何“和平台相关”的代码。main()函数和第 9.2 节一样；但在这里，使用的是 RT_Thread_Per_Connection_Logging_Server 类的实例，而非 Thread_Per_Connection_Logging_Server 类的实例。

9.4 ACE_TSS 类

设计意图

C++全局变量虽然有时很有用，但也有可能带来有害的副作用和不确定的初始化

语义 [LGS00]，从而产生微妙的问题。在“多线程”应用程序中，这些问题还会加剧。特别是，当多个线程同时访问“未同步的全局变量”时，信息有可能丢失，或者，会使用错误的信息。

以 C 的 errno 变量为例。假设一个进程中有一个 errno，那么，串行访问这个 errno 是没有意义的，因为它可以被并发地（concurrently）修改——例如，在“一个系统函数设置它”和“应用程序代码检测它”这两个时间点之间。所以，第 6.4 节介绍的同步机制在此不适用。每一个线程都需要自己的 errno 副本——这正是 OS“线程专有存储（TSS, thread-specific storage）”机制提供的特性。使用 TSS 机制，没有一个线程能够看到另一个线程的 errno 实例（instance）。

遗憾的是，操作系统提供的原始 TSS C API 存在以下问题：

- **容易出错**——在大多数 TSS C API 中，“线程专有（thread-specific）”对象的指针都被存储为 void *；对通用 API 来说，这是最合理的处理方式。然而，这也是偶发复杂性的又一根源，因为 void 指针不但容易出错，还丧失了“类型安全”。它们也必须被转换为实际类型；这给程序员带来了错误之源。

- **不可移植**——除了 API 差异之外，TSS 的功能和用法在各个平台上也有所不同。例如，Win32 不像 Pthreads 和 UI 线程那样，当一个线程退出时，它没有提供可靠的方式来释放“线程专有”对象。相反，在 UI 线程中，没有 API 可以删除一个键（key）。由于存在这些不同的语义，我们很难编写出“在各种 OS 平台上都能移植而且能稳定运行”的代码。

如果在每一个应用程序中都要去处理这些问题，那将十分麻烦，而且容易出错；所以，ACE 提供了 ACE_TSS 类。

功能

ACE_TSS 类实现了 *Thread_Specific Storage 模式*，封装并增强了原始 OS TSS API [SSRB00]。这个类提供以下功能：

- 支持“在物理上”为线程所专有（即，私有于一个线程）、但“逻辑上”能像程序中的全局变量那样访问的数据。

• 使用了 C++的“委托（delegation）运算符”operator–>()，来提供“线程专有”的*智能指针（smart pointer）*[Mey96]。

• 封装了“和 TSS 对象相关联”的键（key）的管理。

• 在“本身缺乏足够 TSS 支持”的平台（如 VxWorks）上，ACE_TSS 高效地模拟了 TSS。

ACE_TSS 类的接口如图 9.4 所示，其主要方法如下：

方　法	说　明
operator–>()	得到“和 TSS 键相关联”的“线程专有”对象。
cleanup()	静态方法；线程退出时删除 TSS 对象。

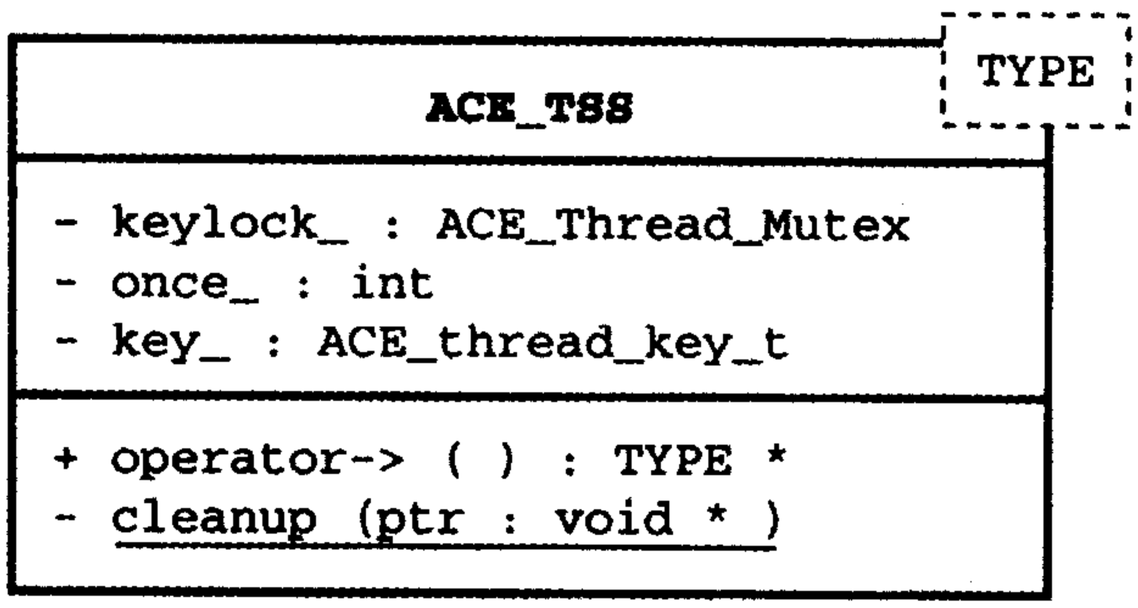


图 9.4：ACE_TSS 类图

ACE_TSS 模板是一个“代理（proxy）”，它将普通的 C++类转换为“类型安全”的类，这些类的实例位于“线程专有存储空间”中。它结合了 operator–>()“委托”方法和其他 C++特性，如模板、*内联（inlining）*、重载。此外，它还运用了常见的同步模式及技术（idioms）[SSRB00]，如 Double-Checked Locking Optimization 和 Scoped Locking，如下面的 operator–>()实现所示：

```
template <class TYPE> TYPE *
ACE_TSS<TYPE>::operator-> () {
  if (once_ == 0) {
    // Ensure that we're serialized.
    ACE_GUARD_RETURN (ACE_Thread_Mutex, guard, keylock_, 0);

    if (once_ == 0) {
      ACE_OS::thr_keycreate (&key_, &ACE_TSS<TYPE>::cleanup);
```

```
    once_ = 1;
  }
}
TYPE *ts_obj = 0;

// Initialize <ts_obj> from thread-specific storage.
ACE_OS::thr_getspecific (key_, (void **) &ts_obj);

// Check if this method's been called in this thread.
if (ts_obj == 0) {
  // Allocate memory off the heap and store it in a pointer.
  ts_obj = new TYPE;

  // Store the dynamically allocated pointer in TSS.
  ACE_OS::thr_setspecific (key_, ts_obj);
}
return ts_obj;
}
```

实现C++“线程专有存储”代理（proxy）时要考虑到很多问题；关于这一点，请参阅[SSRB00]中关于Thread-Specific Storage模式的介绍。

示例

在这个例子中，我们演示如何将ACE_TSS运用到第9.2节的“一个连接一个线程”日志服务器例程中。在这里的实现中，每一个线程都可以得到“线程专有存储空间”中的请求数（request count）。采用这一设计，我们无需互斥体（mutex）就可以消除“请求数”上的“竞态条件（race condition）”。此外，当使用“线程专有存储空间”时，那些“容易出错、不可移植”的原始CAPI所具有的任何“偶发复杂性”在此也不复存在。

首先，定义一个简单的类，用来跟踪“请求数”：

```
class Request_Count {
public:
  Request_Count (): count_ (0) {}
  void increment () { ++count_; }
  int value () const { return count_; }

private:
  int count_;
};
```

请注意，Request_Count对“线程专有存储”一无所知。然后，将这个类用作ACE_TSS模板的类型参数：

```
static ACE_TSS<Request_Count> request_count;
```

现在，每一个线程都各有一个 Request_Count 副本，能通过 request_count 访问。虽然这个对象在“逻辑上”似乎是“全局”的——也就是说，可以像访问文件范围中任何其他对象一样访问它；但是，它的状态（state）在“物理上”是“局部”存在于使用它的线程中的。将 request_count 集成到 handle_data()方法中很简单，如下所示：

```
virtual int handle_data (ACE_SOCK_Stream *) {
  while (logging_handler_.log_record () != -1)
    // Keep track of number of requests.
    request_count->increment ();

  ACE_DEBUG ((LM_DEBUG, "request_count = %d\n",
              request_count->value ()));
}
```

借助 ACE_TSS wrapper facade，我们无需锁定 increment()和 value()调用；因为这些调用已经消除了“争用（contention）”和“竞态条件（race conditions）”。

9.5 小结

不同的操作系统提供的“多线程”API 不仅在语法上不同，而且在语义上也有微妙的差异。为了消除这些复杂性，ACE“线程”wrapper facade 实现了一组可移植的功能，输出了统一的“多线程”语法和语义。本章提供了好几个经过修改的日志服务器示例，展示了如何通过“多线程”（而不是“多进程”）处理多个客户会话、如何控制服务器线程的实时调度属性。

ACE 还提供了 ACE_Task 并发框架（framework），允许开发者以一种强大的、可扩充的“面向对象”方式使用线程。此外，ACE_Task 并发框架不仅提供了在“对象”环境下创建线程的工具，还提供了灵活、强大的“排队（queueing）”机制；这样，即使是身处“面向消息”和“面向任务”这两个不同体系中的任务之间，也可以传输数据和对象。如 [SH]所述，ACE_Task 框架可用来实现以下常见的并发模式 [SSRM00]:

- *Active Object 模式*。它将方法的“执行”同方法的“调用”分离开来。其目的是提供并发性，简化“线程中的对象”的同步访问。

• *Half-Sync/Half-Async 模式*。它将并发系统中的“同步”和“异步”处理分离开来，简化了编程但不至无端降低性能。这个模式引入了两个通信层：一个用于“同步”服务处理，另一个用于“异步”服务处理。“同步”层和“异步”层之间的通信有一个“排队（queueing）”层过渡。

第 10 章

ACE 同步 Wrapper Facade

本章提要

在本章讲述的 ACE wrapper facade 中，我们通过可移植的 C++类，封装了 OS“同步（synchronization）”机制；给出了一系列相关的例子，演示如何将这些类运用到实践中；此外，还讲述如何将核心的 ACE 同步 wrapper facade 结合起来，模拟出原本没有的同步机制。

10.1 概述

硬件特性（hardware characteristics）显著影响 OS 线程机制的行为，例如，多个线程可以“并行（in parallel）”地运行在各个处理器上。在单处理器中，任何时间点上都只有一个线程是活动（active）的——虽然线程可以被调入、调出，从而造成“并行”运行的表象。但是，在这两种情况下，开发者都必须使用同步机制，以避免“竞态条件（race condition）”，并保证应用程序对共享资源的协调访问。

不同的操作系统通过不同的 API 提供了具有不同语义的同步机制。在这些 API 中，有一些是符合国际标准的，如 Pthreads [IEE96]。有一些 API 则符合*事实上（de facto）*

的标准，如 Win32 [Sol98]。本章将讲述以下 ACE 类：利用这些类，网络应用程序能够可移植地对线程和（或）进程实施同步控制：

ACE 类	说 明
ACE_Guard ACE_Read_Guard ACE_Write_Guard	运用 Scoped Locking 技术[SSRB00]，确保进入或退出一段 C++代码时，能够自动得到或释放锁（lock）。
ACE_Thread_Mutex ACE_Process_Mutex ACE_Null_Mutex	为并发应用程序提供一种简单高效的机制，用以串行访问共享资源。
ACE_RW_Thread_Mutex ACE_RW_Process_Mutex	可以高效、并发地访问“内容经常被读取但较少被修改”的资源。
ACE_Thread_Semaphore ACE_Process_Semaphore ACE_Null_Semaphore	实现“计数信号量”（counting semaphore），这是一种“对多个控制线程实施同步”的通用机制。
ACE_Condition_Thread_Mutex ACE_Null_Condition	使得线程可以高效地协调和调度它们的处理。

这些 wrapper facade 具有以下优点：

- **提高“类型安全性”：**

——使“同步对象”的初始化工作“自动化”——例如，当它们是作为 C++ 类中的数据成员出现的时候。

——防止不正确的使用——例如，将一个互斥体复制给另一个互斥体。

——确保程序“进入”或“退出”某一执行空间时，可以自动“得到”或“释放”锁（lock），即使有异常或错误发生。

- **保证异种 OS 平台上的可移植性。**例如，在本身不支持“readers/writer 锁”和“条件变量”的平台上，ACE 可移植地实现了这些同步机制。

- **增强抽象性和模块性，但不失效率。**通过使用 C++语言特性，如内联函数和模板，使 ACE 同步 wrapper facade 在时间和空间上的额外开销降低，甚至完全消除。

- **输出统一的同步接口，**使得应用程序可以“得到”和“释放”各种不同

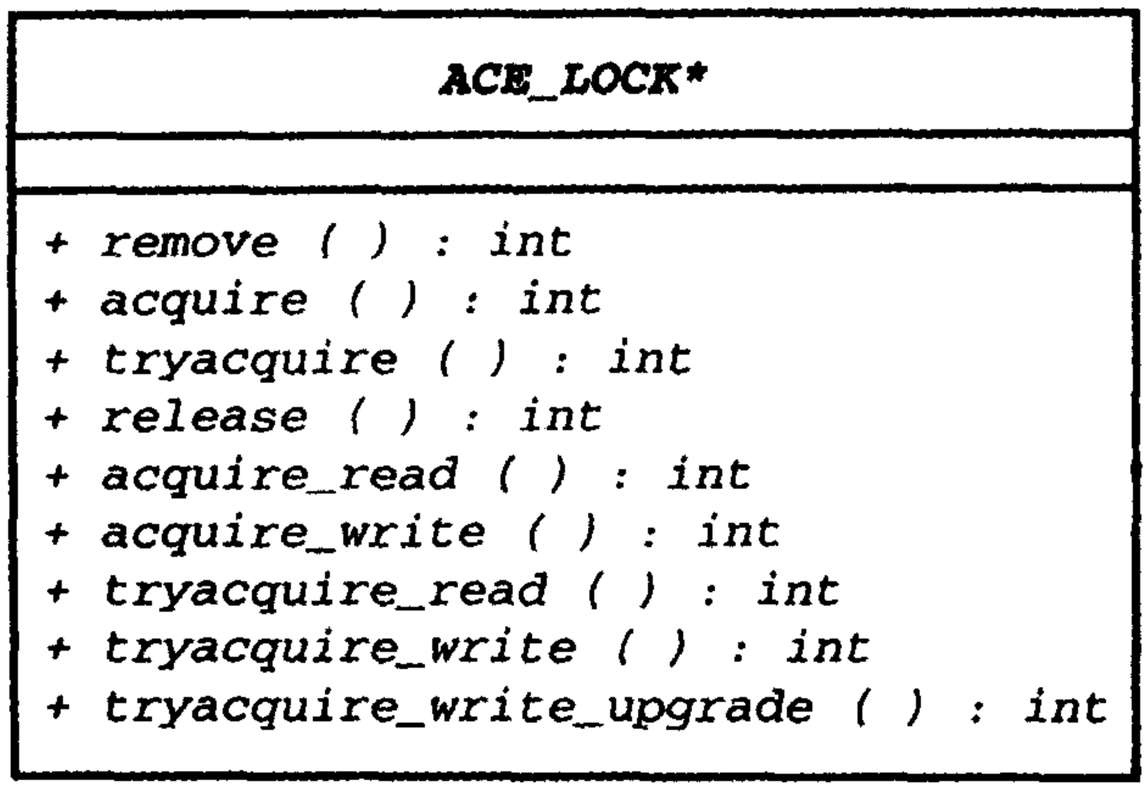


图 10.1：ACE_LOCK* 伪类

类型的锁（lock）。例如，ACE 互斥体、readers/writer、信号量、文件锁（file lock）机制都支持图 10.1 所示的 ACE_LOCK*接口。ACE_LOCK*是“伪类”；也就是说，它不是 ACE 中真正的 C++类。我们只是用它来表示很多 ACE 同步类（如 ACE_Thread_Mutex、ACE_Process_Mutex 和 ACE_Thread_Semaphore）所支持的统一的签名。

• 为代码提供“线程安全”时，**减少对代码的强制修改**。有了 ACE 同步接口的统一性，只需一个 ACE_Guard 模板类（template class）就可以保证：当程序“进入”和“退出”某一运行空间时，大多数类型的锁（lock）都能“得到”和“释放”。

本章提出并讲述了主要的 ACE 同步 wrapper facade 类的功能。我们为每一个类都提供了示例，演示如何通过它们来简化“并发式网络应用程序”中常见的同步问题。

10.2 ACE_Guard 类

设计意图

有了 ACE 同步 wrapper facade 接口的一致性，就能够很容易地正确使用它们，从而有助于克服“跨平台网络编程”中的一些偶发复杂性。这之中的一个常见问题和“锁的范围（lock scoping）”有关。不正确地“获得”和“释放”锁会导致“系统死锁”或意想不到的“竞态条件”。在显式“获得”、“释放” 锁的场合下，我们很难保证在代

码的所有执行路径上都释放了这个锁，特别是在有 C++异常抛出的时候。ACE 提供了 ACE_Guard 类及相关子类，从而有助于保证锁的正确“获得”和“释放”。

功能

ACE_Guard、ACE_Read_Guard 和 ACE_Write_Guard 类实现了 *Scoped Locking 技术（idiom）* [SSRB00]；它们利用了 C++类的构造函数和析构函数语义，从而保证在进入和退出一段 C++代码时，可以自动“获得”、“释放”锁。图 10.2 演示了这些类的接口和它们之间的关系。

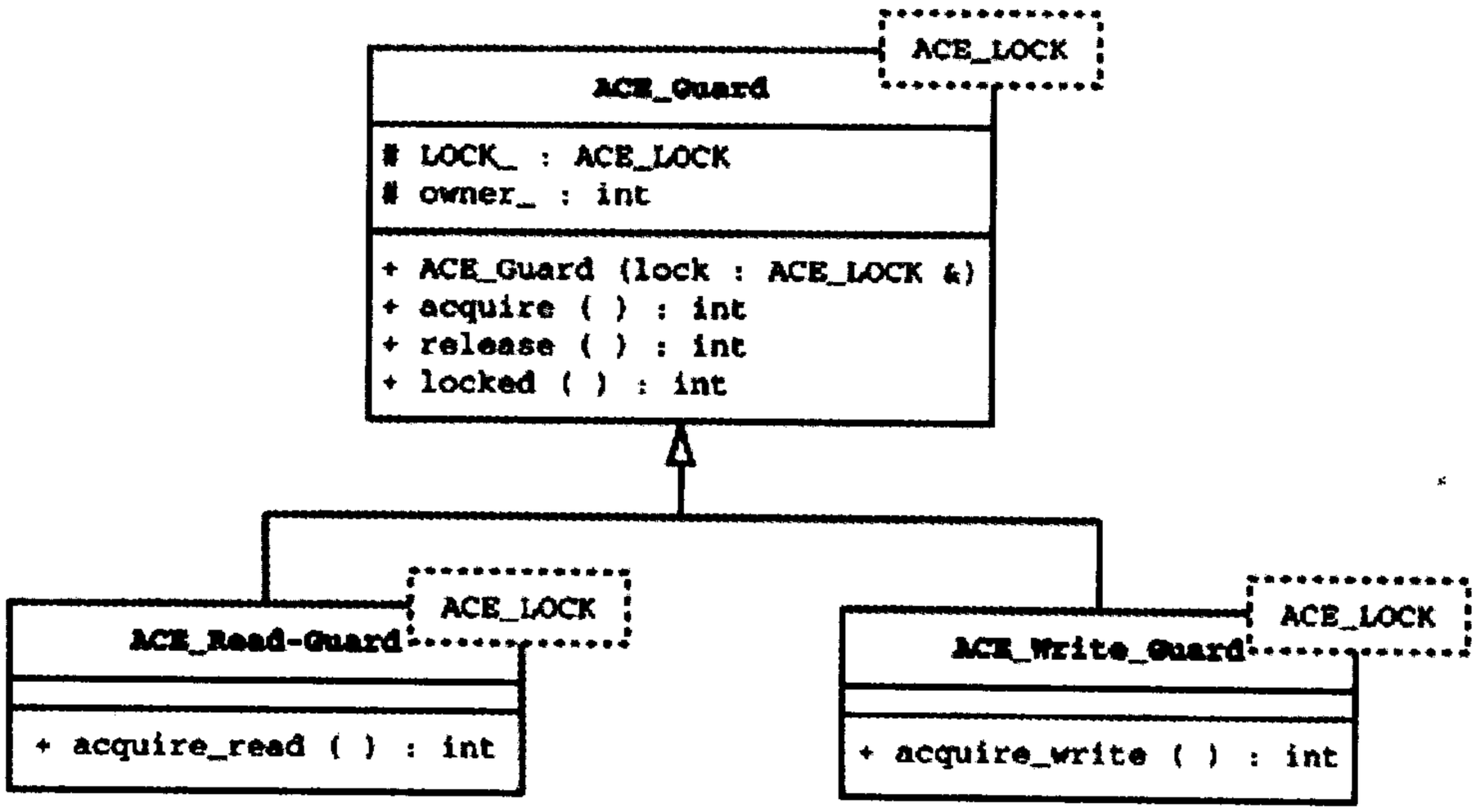


图 10.2：ACE_Guard 系列类图

下面的 C++源代码是 ACE_Guard 类的一部分；它们演示了 ACE_Guard 是如何实现 Scoped Locking 技术的：

```
template <class LOCK>
class ACE_Guard
{
public:
  // Implicitly and automatically acquire the lock.
  ACE_Guard (LOCK &lock): lock_ (&lock) { acquire (); }
```

```
// Implicitly release the lock.
~ACE_Guard () { release (); }

// Explicitly acquire the lock.
int acquire () {
  return owner_ = lock_->acquire ();
}

// Explicitly release the lock.
int release () {
  if (owner_ == -1)
    return -1;
  else {
    owner_ = -1
    return lock_->release ();
  }
}

int locked () const { return owner_ != -1; }
// ... Other methods omitted

protected:
  // Used by subclasses to store pointer.
  ACE_Guard (LOCK *lock): lock_ (lock) {}

  // Pointer to the LOCK we're guarding.
  LOCK *lock_;

  // Keeps track of whether we acquired the lock or failed.
  int owner_;
};
```

一个 ACE_Guard 实例定义了一个代码区域，在这个代码区域中，当这个“守护对象（guard object）”创建和销毁时，一个锁被自动获取和释放。ACE_Guard 是一个类模板（class template），可用于任何 LOCK 类——其中，LOCK 类的公有方法（public method）符合图 10.1 所示的 ACE_LOCK* 伪类的签名。从后文可以看到，本章提供的大多数 ACE 同步 wrapper facade 都属于这种类。

ACE_Read_Guard 和 ACE_Write_Guard 模板类的接口和 ACE_Guard 一样。但是，它们的 acquire()方法是分别通过 LOCK 模板参数上定义的 acquire_read()和 acquire_write()方法得到“读”锁和“写”锁的，如下所示：

```
template <class LOCK>
class ACE_Write_Guard : public ACE_Guard<LOCK>
{
public:
  // Automatically acquire the lock for writing.
  ACE_Write_Guard (LOCK &lock): ACE_Guard (&lock)
  { owner_ = lock_.acquire_write (); }

  // ... Same operations as <ACE_Guard>.
};

template <class LOCK>
class ACE_Read_Guard : public ACE_Guard<LOCK>
{
public:
  // Automatically acquire the lock for reading.
  ACE_Read_Guard (LOCK &lock): ACE_Guard (&lock)
  { owner_ = lock_.acquire_read (); }

  // ... Same operations as <ACE_Guard>.
};
```

在第220页的Atomic_Op实现中，我们演示了ACE_Write_Gurad和ACE_Read_Guard类的用法。

10.3 ACE 互斥体类

设计意图

大多数操作系统都提供了某种形式的互斥体（mutex）机制；利用这些机制，并发应用程序可以串行访问共享资源。和在本书中已经看到的其他大多数“和平台相关”的功能一样，不同的 OS 平台在语法和语义上有着微妙的差异。互斥体还具有不同的“初始化需求”。ACE 互斥体 wrapper facade 的设计，就是为了以一种便利的方式克服这所有问题。

功能

ACE在Wrapper Facade模式的指导下，用ACE_Process_Mutex和ACE_Thread_Mutex类封装了原始的 OS 互斥体同步机制，从而在“系统”范围和“进程”范围中可移植地

实现了“非递归”互斥体语义。[1] 因此，可以运用这些类，让线程串行访问代码关键段（critical section）；这些关键段既可以跨进程，也可以位于一个进程中。ACE_Thread_Mutex 类的接口和图 10.1 给出的 ACE_LOCK* “伪类”的接口完全一样。在下面的 C++类代码中，将演示如何用 Pthreads 来实现 ACE_Thread_Mutex：

```
class ACE_Thread_Mutex
{
public:
  ACE_Thread_Mutex (const char * = 0, ACE_mutexattr_t *attrs = 0)
  { pthread_mutex_init (&lock_, attrs); }
  ~ACE_Thread_Mutex () { pthread_mutex_destroy (&lock_); }
  int acquire () { return pthread_mutex_lock (&lock_); }
  int acquire (ACE_Time_Value *timeout) {
    return pthread_mutex_timedlock
      (&lock_, timeout == 0 ? 0 : *timeout);
  }
  int release () { return pthread_mutex_unlock (&lock_); }
  // ... Other methods omitted ...
private:
  pthread_mutex_t lock_; // Pthreads mutex mechanism.
};
```

ACE_Thread_Mutex 对象中有一个 acquire()方法；对这个方法的所有调用都会阻塞，直至当前拥有这个锁的线程离开了它的关键段为止。一个线程在离开关键段时，必须在它拥有的互斥体对象上调用 release()方法；这同时也使得另一个阻塞在这个互斥体上的线程恢复执行，进入它的关键段。

在 Win32 平台上，ACE_Thread_Mutex 类是用 CRITICAL_SECTION 实现的，这是一种轻量级的 Win32 锁，用来对单个进程内的线程实施串行化控制。相反，Win32 上的 ACE_Process_Mutex 实现（implementation）使用的是基于 HANDLE 的互斥体；它可以工作在一个进程中、或同一机器上的多个进程之间：

```
class ACE_Process_Mutex
{
public:
  ACE_Process_Mutex (const char *name, ACE_mutexattr_t *)
  { lock_ = CreateMutex (0, FALSE, name); }
  ~Thread_Mutex () { CloseHandle (lock_); }
```

[1] “进程”范围递归互斥体在 ACE_Recursive_Thread_Mutex 类中实现，如第 10.6 节所示。

```
  int acquire ()
  { return WaitForSingleObject (lock_, INFINITE); }
  int acquire (ACE_Time_Value *timeout) {
    return WaitForSingleObject
      (lock_, timeout == 0 ? INFINITE : timeout->msec ());
  }
  int release () { return ReleaseMutex (lock_); }
  // ... Other methods omitted ...
private:
  HANDLE lock_; // Win32 serialization mechanism.
};
```

在所有 OS 平台上，ACE_Thread_Mutex 类都是通过原始的“进程范围互斥体”机制实现的。但根据 OS 平台特征的不同，ACE_Process_Mutex 可以通过不同的机制实现，例如：

• Win32 中使用的是原始互斥体，如上所示。

• 一些 UNIX 线程实现，如 Pthreads 和 UI 线程，要求“进程间互斥体（interprocess mutex）”必须在共享内存中分配。在这些平台上，如果可能，ACE 将优先使用 UNIX System V 信号量；因为和原始的“进程间互斥体”相比，前者为恢复系统资源提供了更可靠的机制。

• 一些应用程序需要的信号量超过了 System V IPC 内核参数（kernel parameter）允许的数量，还有一些应用程序需要“执行定时 acquire()操作”的能力。在这些场合下，可以将 ACE 配置为“在共享内存中”使用原始互斥体，从而将“保证正确清理”的责任更多地交给了应用程序开发者。

无论 ACE 使用哪一种底层机制，类的接口完全一样；这使得应用程序开发者可以尝试不同的配置，但无需改动它们的源代码。

ACE_Null_Mutex 是 ACE 支持的另一种互斥体。以下是它的部分接口和实现：

```
class ACE_Null_Mutex
{
public:
  ACE_Null_Mutex (const char * = 0, ACE_mutexattr_t * = 0) {}
  int acquire () { return 0; }
  int release () { return 0; }
  // ...
};
```

在 ACE_Null_Mutex 类中，所有方法都被实现为“空操作”，且为内联函数，因而可以被编译器的优化程序完全处理掉。这个类提供的是伪 ACE_LOCK*接口的“零开销”实现；其他用于处理“线程”和“同步”的 ACE C++ wrapper facade 也共享了这一接口。

ACE_Null_Mutex 是 Null Object 模式[Woo97]的一个例子。它可以和 Strategized Locking 模式[SSRB00]一起使用，使得应用程序可以参数化（parameterize）它们需要的同步类型，但不必修改应用程序代码。这种能力在不需要“互斥（mutual exclusion）”的场合下很有用——例如，当应用程序运行在单线程上，并且（或者）永远不会和其他线程争夺资源访问权的时候。

示例

在下面的代码中，演示如何运用 ACE_Thread_Mutex 来解决第 6.5 节的 UNIX International 互斥体方案存在的问题。

```
#include "ace/Synch.h"

typedef u_long COUNTER;

static COUNTER request_count; // File scope global variable.

// Mutex protecting request_count (constructor initializes).
static ACE_Thread_Mutex m;

// ...

virtual int handle_data (ACE_SOCK_Stream *) {
  while (logging_handler_.log_record () != -1) {
    // Try to acquire the lock.
    if (m.acquire () == -1) return 0;
    ++request_count;   // Count # of requests
    m.release (); // Release lock.
  }

  m.acquire ();
  int count = request_count;
  m.release ();
   ACE_DEBUG ((LM_DEBUG, "request_count = %d\n", count));
   Logging_handler_.close();
   return 0;
}
```

ACE_Thread_Mutex C++ wrapper 类的运用，使得最初的代码在一定程度上变得更为整洁，可移植性也得到了提高，而且，当它实例化的时候，底层互斥体对象会被自动初始化。

副栏 22：ACE_GUARD 宏概览

ACE 定义了一组宏，用以简化 ACE_Guard、ACE_Write_Guard 和 ACE_Read_Guard 类的使用。这些宏可以检测死锁，还可以发现“底层的锁（lock）”上的操作何时失败。通过检测，它们可以保证在程序继续执行之前，锁已被真正锁定：

```
# define ACE_GUARD(MUTEX,OBJ,LOCK) \
  ACE_Guard< MUTEX > OBJ (LOCK); \
    if (OBJ.locked () == 0) return;
# define ACE_GUARD_RETURN(MUTEX,OBJ,LOCK,RETURN) \
  ACE_Guard< MUTEX > OBJ (LOCK); \
    if (OBJ.locked () == 0) return RETURN;
# define ACE_WRITE_GUARD(MUTEX,OBJ,LOCK) \
  ACE_Write_Guard< MUTEX > OBJ (LOCK); \
    if (OBJ.locked () == 0) return;
# define ACE_WRITE_GUARD_RETURN(MUTEX,OBJ,LOCK,RETURN) \
  ACE_Write_Guard< MUTEX > OBJ (LOCK); \
    if (OBJ.locked () == 0) return RETURN;
# define ACE_READ_GUARD(MUTEX,OBJ,LOCK) \
  ACE_Read_Guard< MUTEX > OBJ (LOCK); \
    if (OBJ.locked () == 0) return;
# define ACE_READ_GUARD_RETURN(MUTEX,OBJ,LOCK,RETURN) \
  ACE_Read_Guard< MUTEX > OBJ (LOCK); \
    if (OBJ.locked () == 0) return RETURN;
```

但是，ACE“并发”wrapper facade 没有解决第 6.5 节列出的所有问题。例如，程序员还是得手工释放互斥体，从而会因为程序员的疏忽或因为发生 C++异常而导致 bug。此外，需要显式地（explicitly）检查互斥体是否真正得到，这对程序员来说也很麻烦，而且容易出错。例如，在将 request_count 赋给 count 之前，可能忘记去检查 ACE_Thread_Mutex::acquire()方法是否执行成功。我们可以运用 Scoped Locking[SSRB00] 技术，通过副栏 22 中介绍的 ACE_GUARD *宏来简化这个例子。

```
virtual int handle_data (ACE_SOCK_Stream *) {
  while (logging_handler_.log_record () != -1) {
    // Acquire lock in constructor.
    ACE_GUARD_RETURN (ACE_Thread_Mutex, guard, m, -1);
    ++request_count;   // Count # of requests
    // Release lock in destructor.
  }

  int count;
  {
    ACE_GUARD_RETURN (ACE_Thread_Mutex, guard, m, -1);
    count = request_count;
  }
  ACE_DEBUG ((LM_DEBUG, "request_count = %d\n", count));
}
```

对代码稍作改动之后，就能够保证 ACE_Thread_Mutex 会被自动获得和释放；还能保证，将会正确地处理互斥体出错的情况。

至此，已经解决了第 6.5 节提出的很多问题；但以下两个问题仍然存在：

- **这个方案还是具有强制性**。也就是说，我们必须在一个“用花括号隔开”的新的语句空间（statement scope）中，手工插入 ACE_GUARD 宏。第 10.4 节将演示如何消除这种强制性。

- **这个方案还是容易引发错误**。也就是说，在调用 handle_data()方法之前，静态 ACE_Thread_Mutex 全局对象 m 必须被正确地初始化。要保证这一点极为困难，因为 C++无法保证不同文件中的全局对象和静态对象的初始化顺序。副栏 23 讲述了 ACE 是如何通过 ACE_Object_Manager 类来处理这个问题的。

ACE_Object_Manager 类提供了一个全局递归互斥体；我们可以在日志服务器示例中使用它：

```
// ...
while (logging_handler_.log_record () != -1) {
  // Acquire lock in constructor.
  ACE_GUARD_RETURN (ACE_Recursive_Thread_Mutex, guard,
                    ACE_Static_Object_Lock::instance (),
                    -1);
  ++request_count;   // Count # of requests
  // Release lock in destructor.
}
// ...
```

副栏 23：ACE_Object_Manager 类

为保证“全局”对象及“静态”对象初始化顺序的一致性，ACE 提供了 ACE_Object_Manager 类；此类实现了 Object Lifetime Manager[LGS00]模式。此模式管理对象的整个生存期是：从第一次使用之前创建它们，到程序结束时确保正确地销毁它们。ACE_Object_Manager 类提供了以下功能；这些功能有助于将“静态对象的创建/销毁”替代为自动的“动态对象的预分配/释放”：

• 管理程序结束时的“对象清理”工作（一般是 singletons）。除了管理 ACE 库的清理之外，它还提供了一个应用程序接口，用来登记“要被清理的对象”。

• 程序结束时关闭 ACE 库的服务，使得这些服务能够收回它们的存储空间。做法是：创建一个静态实例（static instance），当其他所有静态对象的析构函数被调用时，这个实例的析构函数也会被一起调用。应用程序代码中还提供了 hook 函数，用来登记“程序退出时需要清理的对象和数组”。这些“清理”调用的次序和登记时相反（LIFO）；即，后登记的对象/数组先被清除。

• 预先初始化某些对象，如 ACE_Static_Object_Lock，使得在 main()函数运行之前，它们已被创建。

ACE_Static_Object_Lock 实例由 ACE_Object_Manager 控制。所以，当程序开始执行时，它一定会被初始化；当程序退出时，它也一定会被正确销毁。

10.4 ACE Readers/Writer Lock 类

设计意图

有了 readers/writer 锁（lock），我们就可以对“内容经常被读取但较少被修改”的资源执行高效的并发访问。很多操作系统都在它们的“文件锁定”API 中支持 readers/writer 语义。但是，“同步”行为中如果牵扯到了“文件系统”，不仅会造成不

必要的低效，还会在意想不到的情况下（例如，当“锁文件（lock file）”使用了网络上安装的存储设备的时候）造成阻塞。此外，“文件锁定”机制只在“系统范围”（而非“进程范围”）级（level）可行。

令人惊讶的是，很少有操作系统在“进程范围”级实现“readers/writer 锁”，如 Pthreads（不带 UNIX98 扩展）、Win32 线程，以及很多本身不支持“readers/writer 锁”的实时操作系统。UI 线程通过 rwlock_t 类型支持 readers/writer 锁。如果要在每一个应用程序中处理这些可移植性上的差异，那将很麻烦，而且容易出错；所以 ACE 提供了 readers/writer lock wrapper facade。

功能

ACE 通过 ACE_RW_Thread_Mutex 和 ACE_RW_Process_Mutex 类封装了原始 OS“readers/writer 锁”机制。这些类运用 Wrapper Facade 模式，可移植地实现了“进程”和“系统”范围的“readers/writer 锁”语义。这些类的接口和图 10.1 给出的 ACE_LOCK* “伪类”的签名完全相同。

如果一个平台支持“进程范围”readers/writer 锁，则 ACE_RW_Thread_Mutex 类将只是简单地对原始的同步变量进行封装。但是，在没有提供原始“readers/writer 锁”的 OS 平台上，ACE 会通过已有的低级同步机制（如互斥体、条件变量、信号量）来提供 readers/writer 实现。ACE 的 readers/writer 实现会优先处理 writer。所以，如果有多个 readers 和一个 writer 等待这个锁，writer 会首先得到它。

示例

在第 216 页的例子中，我们基于 ACE_Guard 实现了 handle_data()；这个方案解决了一些问题，但还是需要对代码作强制性的修改——即，在循环中添加 ACE_GUARD_RETURN。一个不容易出错、不具强制性的方案是利用下面的工具：

- C++运算符重载
- ACE_RW_Thread_Mutex

• Scoped Locking 手法（使用了 ACE_WRITE_GUARD 和 ACE_READ_GUARD 宏）

下面，利用这些功能创建一个 Atomic_Op 类；此类支持“线程安全”的算术运算：

```
class Atomic_Op
{
public:
  // Initialize <count_> to <count>.
  Atomic_Op (long count = 0)
    : count_ (count) {}

  // Atomically pre-increment <count_>.
  long operator++ () {
    // Use the <acquire_write> method to acquire a write lock.
    ACE_WRITE_GUARD_RETURN (ACE_RW_Thread_Mutex, guard, lock_,
                            -1);
    return ++count_;
  }

  // Atomically return <count_>.
  operator long () {
    // Use the <acquire_read> method to acquire a read lock.
    ACE_READ_GUARD_RETURN (ACE_RW_Thread_Mutex, guard, lock_,
                           0);
    return count_;
  }

  // ... Other arithmetic operators omitted.

private:
  // Readers/writer lock.
  ACE_RW_Thread_Mutex lock_;

  // Value of the <Atomic_Op> count.
  long count_;
};
```

Atomic_Op 类在 long 数据类型上重载了标准的算术运算符，即++、—、+=等。因为这些方法要修改 count_，所以它们需要获得“写”锁才能互斥地访问 count_。相反，operator long()只需要“读”锁就足够了；因为“读”锁允许多个线程并发读取 count_。

现在，可以用 Atomic_Op 类写出下面的代码；这段代码几乎和最初的“非线程安全”代码一样。只是 COUNTER 的 typedef 作了改动：

```
typedef Atomic_Op COUNTER;

static COUNTER request_count; // File scope global variable.

 virtual int handle_data (ACE_SOCK_Stream *) {
  while (logging_handler_.log_record () != -1)
  // Keep track of number of requests.
  ++request_count; // Actually calls <Atomic_Op::operator++>.

  ACE_DEBUG ((LM_DEBUG, "request_count = %d\n",
             // Actually calls <Atomic_Op::operator long>.
             (long) request_count));
 }
```

在对 operator++()和 operator long()的调用中，我们分别在 ACE_RW_Thread_Mutex 上使用了 acquire_write()和 acquire_read()方法。因而，在多处理器上，Atomic_Op 类的实例化对象上的算术运算如今可以正确地递增/递减计数器（counter），同时，现有的应用程序代码无需作多少修改！

上面给出的 Atomic_Op 类实际上是 ACE_Atomic_Op 类模板的简化版本；后者定义的部分代码如下：

```
template <class TYPE, class LOCK>
class ACE_Atomic_Op {
public:
  // ... Constructors and arithmetic operators omitted.
private:
  LOCK lock_;
  TYPE type_;
};
```

ACE_Atomic_Op 构成了一个简单但极富表达力的“参数化类”抽象体；它结合了以下两类技术：

- Scoped Locking 技术和 Strategized Locking 模式
- C++模板和运算符重载的使用

这个模板类可以正确地、原子地（atomically）在“需要原子操作（atomic operation）”的类型集（family of types）上操作。例如，为了向其他算术类型提供相同的“线程安全”功能，只需像下面这样实例化出 ACE_Atomic_Op 模板类的新对象：

```
ACE_Atomic_Op<double, ACE_Null_Mutex> double_counter;
ACE_Atomic_Op<Complex, ACE_RW_Thread_Mutex> complex_counter;
```

10.5 ACE 信号量类

设计意图

*信号量（semaphore）*是一项功能强大的机制，用于在并发应用程序中“锁定”和（或）“同步访问”共享资源。一个信号量有一个计数（count），表示共享资源的状态。应用程序设计者赋予信号量的计数（count）一定的含义和初值。因此，信号量可用来对“资源池（pool of resource）”的访问施加调节。

释放（releasing）信号量会递增计数值——无论当时是否有等待者（waiter）；所以，信号量可用于跟踪“修改共享程序状态”的事件。线程可以根据这些事件作出决策——即使事件已经发生。虽然大多数操作系统已经提供了某种形式的信号量机制，但是，ACE 信号量 wrapper facade 针对广泛环境下语法和语义的微妙差异，提供了解决方案，消除了它们存在的问题。

功能

ACE_Thread_Semaphore 和 ACE_Process_Semaphore 类根据 Wrapper Facades 模式，分别可移植地封装了“进程范围”及“系统范围”信号量。在很大程度上，这些类的接口和图 10.1 给出的 ACE_LOCK* 伪类的接口是一样的。但是，它的构造函数稍有不同；因为和“互斥体”及“readers/writer 锁”相比，信号量在“初始化”上的开销更昂贵：它允许设置信号量计数的初始值。ACE_Thread_Semaphore API 的部分代码如下：

```
class ACE_Thread_Semaphore
{
public:
  // Initialize the semaphore, with an initial value of <count>,
  // a maximum value of <max>, and unlocked by default.
  ACE_Thread_Semaphore (u_int count = 1,
                        const char *name = 0,
                        void *arg = 0,
                        int max = 0x7FFFFFFF);
  // ... same as pseudo <ACE_LOCK> signatures.
};
```

ACE_Process_Semaphore 具有相同的接口，但它是在“系统范围” 而不是在“进程范围”对线程实施同步。

这两个 ACE 类尽可能地封装了 OS 本身的信号量机制；如果 OS 平台本身不支持信号量，它们会模拟它。这样，应用程序不但可以使用信号量，还可以移植到新平台上——无论那些平台本身是否提供了信号量支持。在第 A.5.2 节，我们为本身不支持信号量的平台提供了一个模拟的 ACE_Thread_Semaphore。

在 ACE_Null_Semaphore 类中，所有方法都被实现为“空操作”，且为内联函数。我们实现了它的两个 acquire()方法，如下所示：

```
class ACE_Null_Semaphore
{
public:
  int acquire () { return 0; }
  int acquire (ACE_Time_Value *) { errno = ETIME; return -1; }
  // ...
};
```

请注意，“定时”版本的 acquire()返回–1，并将 errno 设置为 ETIME，从而保证 ACE_Null_Semaphore 和 Strategized Locking[SSRB00]模式可以结合起来，实现互换性。对于非 null ACE 信号量，acquire()的“阻塞”版本常用来串行访问关键段，而“定时”版本常用来等待另一个线程更新某一条件，或改变某一共享状态。但是，在使用 ACE_Null_Semaphore 的时候，没有其他线程会改变某个状态或条件。否则，使用 null 信号量就不合适。返回“出错值”意味着状态或条件没有（并且无法）改变；这和“状态或条件改变之前发生超时”时的行为是一致的。

示例

虽然信号量可以协调处理多个线程，但它们本身并不在线程间传递任何数据。但是，在线程之间传递数据是一种常用的并发编程技术；所以，如果能提供某种轻量级的进程内消息排队（message queueing）机制，那将非常有用。所以，在此实现了一个 Message_Queue 类；它提供以下功能：

• 允许消息（ACE_Message_Block 对象）在队列的尾部入列（enqueued）或在队列的头部出列（dequeued）。

• 可以调节流量控制，防止快速的“消息生产者线程”吞噬较慢的“消息使用者线程”的资源。

• 必要的时候对“入列”和“出列”操作指定“超时”值，以避免无限期阻塞。

下面，给出 Message_Queue 类实现（implementation）的主要部分，并演示 ACE_Thread_Semaphore 的用法。为简化代码的设计和维护工作，还运用了来自 POSA2 [SSRB00]的模式和技术：

• **Monitor Object**——Message_Queue 的公有方法（public method）遵循 Monitor Object（监控器对象）设计模式 [SSRB00]，像“被同步的方法”那样工作，从而保证一个对象中同时只有一个方法在运行，并且允许对象中的方法可以协同调度它们的执行顺序。

• **Thread-Safe Interface**——Message_Queue 的公有方法得到锁，并将实际的“排队”行为委托给私有（private）的、遵循 *Thread-Safe Interface（“线程安全”接口）模式*的实现方法（implementation method）。这个设计模式将“锁定”操作的开销降至最低，还能保证对象内部的方法调用不会“自死锁”，因为它会尝试重新获得对象已经拥有的锁。

• **Scoped Locking**——通过使用 Scoped Locking（作用域内加锁）技术[SSRB00]，ACE_GUARD*宏（macro）可以保证：签名符合 ACE_LOCK*“伪类”的任何“同步” wrapper facade 都能自动“得到”和“释放”锁。

POSA2 一书中的大多数模式都可以通过 ACE“并发”及“同步”类来实现。

首先，给出 Message_Queue 类的定义：

```
class Message_Queue
{
public:
  // Default high and low water marks.
  enum {
    DEFAULT_LWM = 0,        // 0 is the low water mark.
    DEFAULT_HWM = 16 * 1024 // 16 K is the high water mark.
  };
```

```
  // Initialize.
  Message_Queue (size_t = DEFAULT_HWM, size_t = DEFAULT_LWM);
  // Destroy.
  ~Message_Queue ();
  // Checks if queue is full/empty.
  int is_full () const;
  int is_empty () const;

  // Interface for enqueueing and dequeueing ACE_Message_Blocks.
  int enqueue_tail (ACE_Message_Block *, ACE_Time_Value * = 0);
  int dequeue_head (ACE_Message_Block *&, ACE_Time_Value * = 0);

private:
  // Implementations that enqueue/dequeue ACE_Message_Blocks.
  int enqueue_tail_i (ACE_Message_Block *, ACE_Time_Value * = 0);
  int dequeue_head_i (ACE_Message_Block *&, ACE_Time_Value * = 0);

  // Implement the checks for boundary conditions.
  int is_empty_i () const;
  int is_full_i () const;

  // Lowest number before unblocking occurs.
  int low_water_mark_;
  // Greatest number of bytes before blocking.
  int high_water_mark_;
  // Current number of bytes in the queue.
  int cur_bytes_;
  // Current number of messages in the queue.
  int cur_count_;
  // Number of threads waiting to dequeue a message.
  size_t dequeue_waiters_;
  // Number of threads waiting to enqueue a message.
  size_t enqueue_waiters_;

  // C++ wrapper facades to coordinate concurrent access.
  mutable ACE_Thread_Mutex lock_;
  ACE_Thread_Semaphore notempty_;
  ACE_Thread_Semaphore notfull_;

  // Remaining details of queue implementation omitted...
};
```

在下面给出的 Message_Queue 的构造函数中，创建一个空的消息列表，并初始化 ACE_Thread_Semaphore，使之从 0 开始计数（互斥体 lock_通过它的缺省构造函数被自动初始化）。

```
Message_Queue::Message_Queue (size_t hwm, size_t lwm)
  : low_water_mark_ (lwm),
    high_water_mark (hwm),
    cur_bytes_ (0),
    cur_count_ (0),
    dequeue_waiters_ (0),
    enqueue_waiters_ (0),
    notempty_ (0),
    notfull_ (0)
{ /* Remaining constructor implementation omitted ... */ }
```

下面的方法检查队列是否为“空”(即，不包含消息)，或为“满”(即，包含的消息超过 high_water_mark_字节)。这些方法(像下面其他方法一样)根据 Thread-Safe Interface[SSRB00]模式设计而成。

首先，我们来看看 is_empty()和 is_full()这两个接口方法(interface method)：

```
int Message_Queue::is_empty () const {
  ACE_GUARD_RETURN (ACE_Thread_Mutex, guard, lock_, -1);
  return is_empty_i ();
}

int Message_Queue::is_full () const {
  ACE_GUARD_RETURN (ACE_Thread_Mutex, guard, lock_, -1);
  return is_full_i ();
}
```

这些方法得到 lock_，然后将调用转给下面某个实现方法(implementation method)：

```
int Message_Queue::is_empty_i () const {
  return cur_bytes_ <= 0 && cur_count_ <= 0;
}

int Message_Queue::is_full_i () const {
  return cur_bytes_ >= high_water_mark_;
}
```

这些方法认为已经拥有了 lock_，因而实际执行工作。

enqueue_tail()方法将一个新元素插入到队列的尾部，并返回队列中消息的个数。和 dequeue_head()方法一样，如果 timeout 参数为 0，调用者将被阻塞，直至可以继续动作。否则，调用者只会被阻塞至一定的时间，最长不超过*timeout。当有信号发生时，或者 timeout 指定的时间已到，被阻塞的调用会返回，并将 errno 设置为 EWOULDBLOCK。

```
int Message_Queue::enqueue_tail (ACE_Message_Block *new_mblk,
                                 ACE_Time_Value *timeout)
{
  ACE_GUARD_RETURN (ACE_Thread_Mutex, guard, lock_, -1);
  int result = 0;

  // Wait until the queue is no longer full.
  while (is_full_i () && result != -1) {
    ++enqueue_waiters_;
    guard.release ();
    result = notfull_.acquire (timeout);
    guard.acquire ();
  }

  if (result == -1) {
    if (enqueue_waiters_ > 0)
      --enqueue_waiters_;
    if (errno == ETIME)
      errno = EWOULDBLOCK;
    return -1;
  }

  // Enqueue the message at the tail of the queue.
  int queued_messages = enqueue_tail_i (new_mblk);

  // Tell any blocked threads that the queue has a new item!
  if (dequeue_waiters_ > 0) {
    --dequeue_waiters_;
    notempty_.release ();
  }
  return queued_messages; // guard's destructor releases lock_.
}
```

如果至少有一个线程等待消息出列，enqueue_tail()方法会释放 notempty_信号量。实际“入列”逻辑位于 enqueue_tail_i()中；但在这里，省略了这段代码，因为它属于底层实现细节。请注意，在“调用 not_full_.acquire()”和“重新得到 guard 锁（lock）”这个“时窗（time window）”中，“竞态条件”有可能发生。在那个很小的时间段中，另一个线程有可能调用 dequeue_head()，递减 enqueue_waiters_。因此，在重新得到锁之后，我们会检查计数，防止将 enqueue_waiters_递减至 0 以下。

在 dequeue_head()方法中，删除队列中的第一个元素，将其传回给调用者，并返回队列中剩余元素的数目：

```
int Message_Queue::dequeue_head (ACE_Message_Block *&first_item,
                                 ACE_Time_Value *timeout)
{
  ACE_GUARD_RETURN (ACE_Thread_Mutex, guard, lock_, -1);
  int result = 0;

  // Wait until the queue is no longer empty.
  while (is_empty_i () && result != -1) {
    ++dequeue_waiters_;
    guard.release ();
    result = notempty_.acquire (timeout);
    guard.acquire ();
  }

  if (result == -1) {
    if (dequeue_waiters_ > 0)
      --dequeue_waiters_;
    if (errno == ETIME)
      errno = EWOULDBLOCK;
    return -1;
  }

  // Remove the first message from the queue.
  int queued_messages = dequeue_head_i (first_item);

  // Only signal if we've fallen below the low water mark.
  if (cur_bytes_ <= low_water_mark_ && enqueue_waiters_ > 0) {
    enqueue_waiters_--;
    notfull_.release ();
  }
  return queued_messages; // <guard> destructor releases <lock_>
}
```

上面的 Message_Queue 类实现了 ACE_Message_Queue 和 ACE_Message_Queue_Ex 的功能子集；后两个类请参见 [SH]。和 Message_Queue 实现相比，这两个 ACE“消息队列”类在以下方面有所不同：

- 除了按 FIFO 次序排队之外，它们还允许在队列的头部（即按照 LIFO 次序）插入消息，或按照优先级次序插入消息。

- 它们运用 C++ traits 和 Strategized Locking 模式，将同步机制参数化；这样，在单线程队列结构（configuration）中，程序员可以放弃“严格的线程安全同步”，以换取“更高的效率”。

- 它们提供了分配器（allocator），允许在各种不同的地方（如共享内存、堆内存、

静态内存、线程专用内存）为消息分配内存。

• 可以对它们进行配置，让它们通过 ACE_Thread_Semaphore 或 ACE_Condition_Thread_Mutex 来调度线程的执行顺序，从而用“效率的损失”换取“更多的功能”。

• ACE_Message_Queue_Ex 是 ACE_Message_Queue 的“强类型（strongly-typed）”版本，它的“入列”和“出列”操作中使用的是 MESSAGE_TYPE 模板参数的实例，而不是 ACE_Message_Block。

10.6 ACE 条件变量类

设计意图

借助条件变量（condition variable），线程可以高效地协调（coordinate）和调度（schedule）它们的处理工作。虽然 Pthreads 标准对条件变量提出了要求，但一些应用广泛的 OS 平台，如 Win32 和 VxWorks，本身并没有实现条件变量。虽然条件变量可以通过其他原始同步机制（如信号量、互斥体）模拟出来，但如果需要应用程序开发者来编写这些模拟代码，将会带来不必要的浪费，而且容易引发错误。为解决这些问题，ACE 提供了 ACE_Condition_Thread_Mutex“条件变量”wrapper facade。

功能

ACE_Condition_Thread_Mutex 以 Wrapper Facade 模式为指导，对“进程范围”条件变量语义进行了封装。ACE_Condition_Thread_Mutex 类的接口如图 10.3 所示，其主要方法见下表：

方　法	说　明
wait()	原子地（atomically）释放“和条件变量相关联”的互斥体，并处于休眠（sleep）状态，等待超时，或等待另一个线程通过 signal()或 broadcast()方法发出通知信息。
signal()	向一个正在等待条件变量的线程发出通知。
broadcast()	向所有正在等待条件变量的线程发出通知。

如果传递给 ACE_Condition_Thread_Mutex::wait()的 time 参数为非 0 值，这个方法

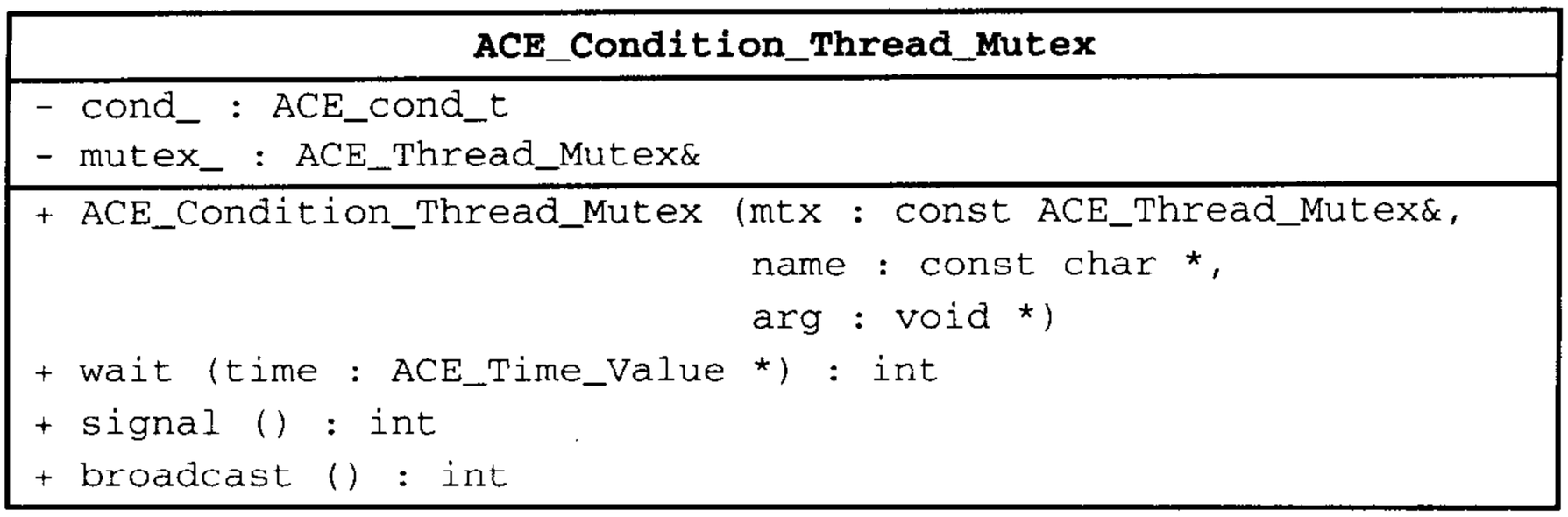


图 10.3：ACE_Condition_Thread_Mutex 类图

会被阻塞，直至超过那个时间点为止（此时间被指定为 ACE_Time_Value）。如果条件变量在接到通知之前已经超时，这个方法会将–1 值返回给调用者，并将 errno 设置为 ETIME，表示发生了超时。无论 wait()如何返回，和条件变量关联的互斥体总会被锁定。

在需要复杂的条件表达式或调度行为的场合下，使用“条件变量”比使用“互斥体”或“信号量”更合适。例如，条件变量常用来实现“同步消息队列”，提供“生产者/消费者”通信，在线程间传递消息。如果消息队列已满，条件变量会阻塞生产者线程；如果队列为空，则会阻塞消费者线程。因为无需维护历史事件，条件变量在接到通知信号时不必作记录。

ACE_Null_Condition 类是一个“遵循 ACE_Condition_Thread_Mutex 接口”的“零开销”实现。这个类和第 10.3 节介绍的 ACE_Null_Mutex 类类似；即，它所有的方法都遵循 Null Object[Woo97]模式，被实现为“空操作”。以下给出的是 ACE_NULL_Condition 类的实现：

```
class ACE_Null_Condition
{
public:
  ACE_Null_Condition (const ACE_Null_Mutex&, int =0, void* =0)
  {}
  ~ACE_Null_Condition () {}
  int remove () { return 0; }
  int wait (ACE_Time_Value* =0) const {errno = ETIME; return -1;}
  int signal () const { return 0; }
  int broadcast () const { return 0; }
};
```

wait()方法返回–1，并将 errno 置为 ETIME；这样，在将 ACE_Condition 和 Strategized Locking 模式结合起来用于像第 10.5 节的 Message_Queue 那样的类时，ACE_Condition 可以正常工作。

示例

缺省情况下，UI 线程的互斥体实现是“非递归”的；即，拥有一个互斥体的线程不能再次得到同一互斥体，否则会在自己身上造成死锁。[2] 虽然非递归互斥体很有效率，但在某些场合下也有很大的局限性。例如，在“回调驱动（callback-driven）”型 C++程序框架（framework）中（在这种程序框架中，程序框架的事件循环会执行用户自定义的*回调（callback）*代码），递归互斥体会特别有用。因为用户自定义代码会通过一个“方法入口点”再次进入程序框架代码；所以，如果使用递归互斥体，则在回调期间，程序框架中的同一线程拥有的互斥体上不会发生死锁。

为处理这类情况，ACE 通过 ACE_Recursive_Thread_Mutex 类为“进程范围”递归互斥体提供了支持。这个类的接口如图 10.4 所示。其中，它的公有接口和图 10.1 中的 ACE_LOCK* 伪类的接口是一致的。在下面的代码中，将演示如何在本身不支持“递归互斥体”的 OS 平台上，用 ACE_Condition_Thread_Mutex 来实现“递归互斥体”。

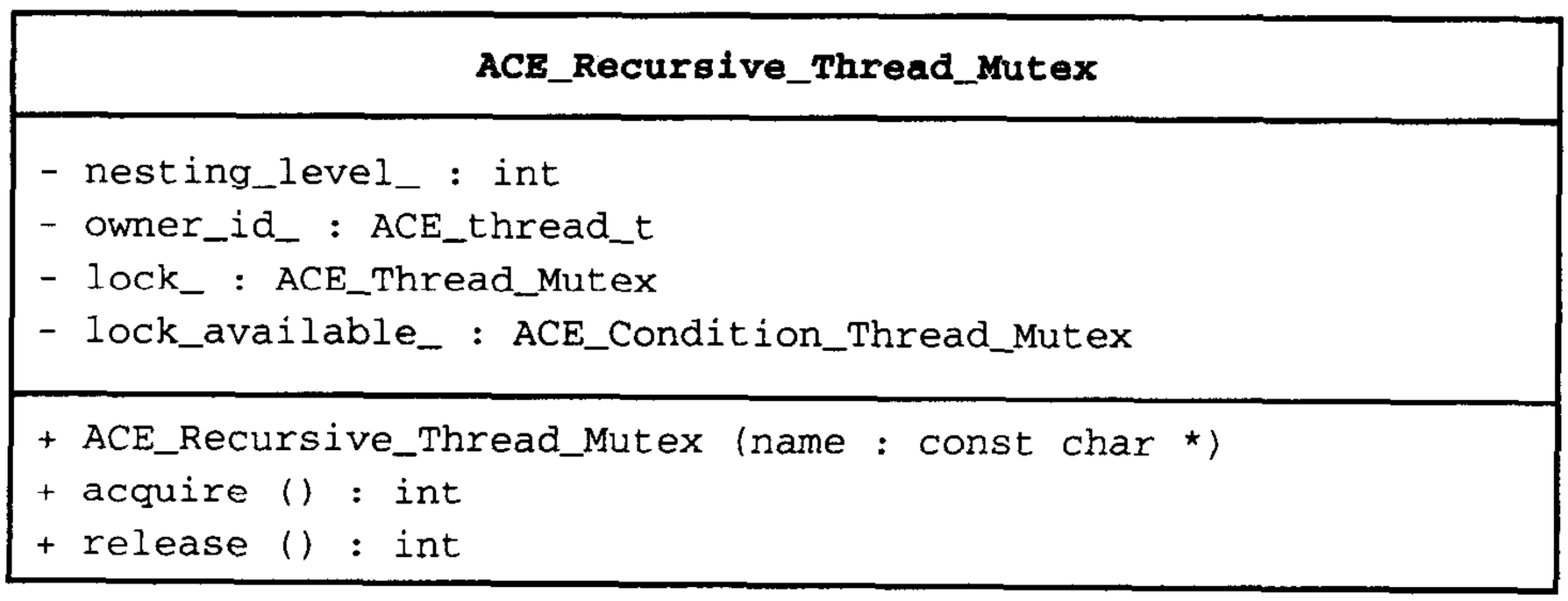


图 10.4：ACE_Recursive_Thread_Mutex 类图

[2] 请注意，Pthreads 和 Win32 本身提供了递归互斥体。

nesting_level_的作用是：对 owner_id_这个线程已经得到“递归互斥体”的次数进行跟踪。lock_串行访问 nesting_level_和 owner_id_。lock_available_条件变量用来挂起（suspend）“非所有者（nonowner）”线程——这些线程等待嵌套层降至 0 且互斥体被释放。ACE_Recursive_Thread_Mutex 的构造函数对这些数据成员进行初始化：

```
ACE_Recursive_Thread_Mutex::ACE_Recursive_Thread_Mutex
  (const char *name, void *arg)
  : nesting_level_ (0), owner_id_ (0), lock_ (name, arg),
    // Initialize the condition variable.
    lock_available_ (lock_, name, arg) { }
```

接着，来看看如何得到一个递归互斥体。

```
 1 int ACE_Recursive_Thread_Mutex::acquire ()
 2 {
 3   ACE_thread_t t_id = ACE_OS::thr_self ();
 4
 5   ACE_GUARD_RETURN (ACE_Thread_Mutex, guard, lock_, -1);
 6
 7   if (nesting_level_ == 0) {
 8     owner_id_ = t_id;
 9     nesting_level_ = 1;
10   }
11   else if (t_id == owner_id_)
12     nesting_level_++;
13   else {
14     while (nesting_level_ > 0)
15       lock_available_.wait ();
16
17     owner_id_ = t_id;
18     nesting_level_ = 1;
19   }
20   return 0;
21 }
```

第 3~5 行 首先，要确定是被哪一个线程调用的；然后，通过 Scoped Locking 技术得到 lock_互斥体。

第 7~12 行 如果没有竞争，则可立即得到互斥体的所有权。如果已经拥有这个互斥体，只是递增嵌套层，让程序继续运行，以避免“自死锁”。

第 13~20 行 如果这个互斥体被另一个线程占有，则使用条件变量 lock_available_，等待嵌套层降为 0。在 lock_available_上调用 wait()方法时，lock_ 互斥体被“原子地

（atomically）”释放，调用者线程进入“睡眠”状态。当嵌套层最终降为 0 时，可以得到 lock_，并获得这个递归互斥体的所有权。guard 的析构函数通过 Scoped Locking 技术释放 lock_。

下面给出 ACE_Recursive_Thread_Mutex::release ()方法：

```
int ACE_Recursive_Thread_Mutex::release ()
{
  owner_id_ = 0;

  // Automatically acquire mutex.
  ACE_GUARD_RETURN (ACE_Thread_Mutex, guard, lock_, -1);

  nesting_level_--;
  if (nesting_level_ == 0)
    lock_available_.signal (); // Inform waiters the lock is
                               // free.

  return 0; // Destructor of <guard> releases the <lock_>.
}
```

当在 lock_available_上调用 signal()方法时，某一“正在等待这一条件”的线程会被唤醒。

10.7 小结

OS 平台提供了多种同步机制，在设计“并发应用程序”时，网络应用程序开发者可以使用这些机制。但是，同步机制除了具有前面各章演示的“与可移植性和 API 的使用有关”的偶发复杂性之外，还存在另一个问题：这些同步机制并不在今天所有的操作系统上存在。在本章，我们演示如何解决这些难题与如何设计“同步”类，从而帮助开发者少犯错误，并通过现有的同步机制来模拟本来没有的东西。最终成果是，我们得到了一组 ACE“同步”wrapper facade；应用程序开发者可以可移植地、可靠地运用它们，以满足大量的同步需要。

ACE“同步”接口改装（adapt）了“语法上不兼容”的原始 OS 同步机制，使得它们在其他 C++类看来具有一致性。利用这些 wrapper facade，可以对网络应用程序中的进程和线程实施同步。本章介绍的是最常用的 ACE“面向对象”同步 wrapper facade。ACE 网站（http://ace.ece.uci.edu）上还介绍了 ACE 提供的其他同步 wrapper facade。

附录 A

ACE C++ Wrapper Facade 的设计原则

本章提要

前面，我们讨论了使用原始 OS API 时存在的偶发复杂性，以及 ACE 如何通过 Wrapper Facade 模式来消除这些复杂性问题，本附录是前面讨论的延续。在本章中，将归纳出蕴涵在 ACE IPC 和并发 wrapper façade 类中的设计原则；这样一来，就可以遵循和改进这些原则，将它们运用到你自己的项目中。我希望，你将来还能够运用这些原则来设计和开发 ACE 中的类！

A.1 概述

在本附录中，将介绍以下设计原则；这些原则贯穿在 ACE wrapper facade 的整个设计之中：

- 通过 wrapper facade 增强类型安全性（type safety）。
- 简化常见情形（common case）。
- 通过继承结构提高设计的清晰性和可扩充性。
- 尽可能地隐藏平台间的差异。
- 实施优化以提高效率。

这里，我们以提纲挈领的形式列出这些原则；这样，你就可以理解这些构成 ACE 的要素，并将它们运用到你自己的框架（framework）和应用程序中。所有这些原则都基于一个背景，即，针对系统级 IPC 及并发机制开发 wrapper facade。可移植性和效率

是 ACE 中的主要考虑因素；所以，我们作出的选择在更高级的“面向对象”应用程序框架中不一定必要，或不一定合适。

A.2 通过 Wrapper Facade 增强类型安全性

高级编程语言，如 C++和 Java，都提供了很强的类型安全性（type safety），从而保证操作（operation）不会施加到不支持它们的对象上。理想情况下，类型错误应该在编译时捕获。但是，OS 库，例如那些实现 C 运行时、Socket、Pthreads 的程序库，很少会展示内部数据结构，因为那样做具有以下弊端：

- 影响“向上兼容”方面的工作；
- 在有些平台上很难支持多种编程语言；
- 使得应用程序可以访问到“它不该使用”的一些私有细节。

因此，OS API 的内部数据结构往往采用的是低级、不透明的表示方式，如 I/O 句柄、socket 句柄，从而展示出“像 C 一样”的接口。这些“不透明”的类型使得编译器很难发现类型错误（type error）。所以，这些错误只能在运行时发现，从而增加了开发和调试的难度，使得“出错处理”工作复杂化，还会降低应用程序的健壮性。本节阐述了“在编译时”增强类型安全性的原则，以及在不需要“过于严格的类型安全性”的情况下，提供*非常手段（escape hatch）*的原则。

A.2.1 设计出“一定会被正确使用”的 C++类

问题：在第 2.3 节我们讨论过，Socket API 具有很多局限性。这些局限性源于以下“由于缺乏类型安全性”所带来的问题：

1. **不具可移植性**——在各个平台上，I/O 句柄的“类型名称”和“底层表示方式”各不相同，这使得它们缺乏可移植性。例如，socket 句柄：在 UNIX 平台上是整数，但在 Win32 上是指针。

2. **容易让人误用**——I/O 句柄的低级表示方式很容易被误用，而且这种误用只能在运行时发现。例如，Socket API 很容易被人这样使用：在一个准备通过 recv()和 send()传输数据的“数据模式”socket 句柄上，不正确地调用了 accept()函数。

方案 => 设计出“一定会被正确使用”的 C++类。“弱类型（weakly typed）”的 I/O 句柄可以被封装在 C++类中；这些类为它们的用户输出的是“强类型（strongly typed）”方法。当应用程序代码使用这些新类时，C++编译器可以保证：只有合法的方法才会被调用。

在下面的 C++代码中，我们演示如何将这条原则应用到第 2.3.1 节的 echo_server() 函数中：

```
int echo_server (const ACE_INET_Addr &addr)
{
  ACE_SOCK_Acceptor acceptor;  // Connection factory.
  ACE_SOCK_Stream peer_stream; // Data transfer object.
  ACE_INET_Addr peer_addr;     // Peer address object.

  // Initialize the passive acceptor and accept a new connection.
  if (acceptor.open (addr) != -1
      && acceptor.accept (peer_stream, &peer_addr) != -1) {
      char buf[BUFSIZ];

    for (size_t n; (n = peer_stream.recv (buf, sizeof buf)) > 0;)
      // send_n() handles "short writes."
      if (peer_stream.send_n (buf, n) != n)
        // Error handling omitted...
  }
}
```

这个版本解决了“使用 Socket API 和 C”时存在的很多问题。例如，ACE_SOCK_Acceptor类只展示了适合“被动建立连接”的方法。因为这些 wrapper façade 类是“强类型”的，不合法的操作在“编译时”而不是“运行时”就会被发现。因而，在 ACE_SOCK_Acceptor 上调用 recv()/send()或在 ACE_SOCK_Stream 上调用 accept() 是不可能的，因为这些方法不是那些 wrapper facade 接口的一部分。

A.2.2 允许在一定范围内破坏“类型安全性”

问题：如上所述，wrapper facade 可以让网络应用程序避免错误，避免“和平台相关”的实现细节（如 socket 句柄表示为整数还是指针）。但是，如果这种额外的“抽象”和“类型安全”实际上会给开发者带来困难，让他们无法“通过原设计者所期望的有用方式”使用这些 wrapper facade，问题就出来了。这种恼人的经历会打击开发者的积极性，从而阻碍他们去利用 wrapper facade 的其他重要优点。

方案 => 允许通过“非常手段”在一定范围内破坏“类型安全性”。这条原则的意图是：使得 wrapper facade 更容易被正确地使用、更难以被错误地使用，同时不可能以“类的设计者最初未曾预料”的某种方式而使用。这条原则在 ACE_IPC_SAP 的 get_handle()和 set_handle()方法中得到了体现；ACE_IPC_SAP 是第 3.3 节介绍的 ACE IPC wrapper facade 的根基类（root base class）。由于 ACE_IPC_SAP 展示了这两个方法来“获取”和“设置”I/O 句柄，当应用程序必须使用“基于句柄”的系统函数时，它可以避开 ACE IPC wrapper facade 的类型检查。

例如，可能需要得到一个 socket 句柄，用于 ACE_Handle_Set 类和 select()函数，如下所示：

```
ACE_SOCK_Acceptor acceptor;
ACE_Handle_Set ready_handles;
// ...

if (ready_handles.is_set (acceptor.get_handle ())
  ACE::select ((int)acceptor.get_handle () + 1, ready_handles ());
```

当然，应该少用这些“非常手段”之类的机制，因为它们会降低可移植性，增加出错几率，从而使得 Wrapper Facade 模式丧失其主要优势。如果有必要将“面向对象”和“非面向对象”这两大抽象（abstraction）混在一起（例如，ACE 自身的某层实现可能就有这样的情况），那么，这些情况应该出现在类实现的内部，很少会被展示给类的用户。如果类的用户需要经常使用“非常手段”，你就得重新审视这个类的接口，看看是否有机会对其实施重构（refactor）或重新设计，以恢复类型安全性。

A.3 简化常见情形

API 开发者负责设计出机制（devising mechanism），使得应用程序开发者能够运用 API 提供的全部功能。现代操作系统、文件系统、协议栈、线程设施提供了广泛的功能，因而今天的 API 一般都庞大而复杂；它们提供了无数的函数，这些函数往往需要很多参数，用来让用户选择他们所需要的行为。

幸运的是，Pareto 法则（也称“80: 20”法则）也适用于软件 API；也就是说，“常用的东西”大都可以通过使用“现有功能中的一个小的子集”而获得[PS90]。此外，相同的函数集（sets of functions）在完成相同或类似任务时，往往会以相同的次序使用。

设计良好的软件工具包会利用这一法则，确定常见情形，并针对这些情形进行简化。ACE wrapper facade 通过以下方式简化常见情形。

A.3.1 将多个函数合并为一个方法

问题：C 语言级的 OS 系统函数 API 所包含的很多函数，往往用来支持相对不大常见的使用场合。例如，我们在第 2.3.2 节讨论过，Socket API 支持很多协议族，如 TCP/IP、IPX/SPX、X.25、ISO OSI、“Unix 领域” sockets。为了支持如此广泛的协议族，Socket API 的原设计者在 Socket API 中分别定义了以下 C 函数：

1. 创建一个 socket 句柄；

2. 将一个 socket 句柄绑定（bind）至一个通信端点上；

3. 标记（mark）一个通信端点，使其成为一个“被动模式”的工厂；

4. 接受一个连接，并返回一个“数据模式”型句柄。

因此，创建和初始化一个“被动模式”的“Internet 领域” socket，将需要多个调用，如下所示：

```
sockaddr_in addr;
int addr_len = sizeof addr;
int n_handle, s_handle = socket (PF_INET, SOCK_STREAM, 0);

memset (&addr, 0, sizeof addr);
addr.sin_family = AF_INET;
addr.sin_port = htons (port);
addr.sin_addr.s_addr = INADDR_ANY;

bind (s_handle, &addr, addr_len);
listen (s_handle);
// ...
n_handle = accept (s_handle, &addr, &addr_len);
```

很多 TCP/IP 服务器都有这么一组(或相应的一组)函数序列,用以主动连接 socket。对于每一个“基于 TCP/IP”的新程序来说,端口的选择可能有微小的变化,但基本上,需要重写的代码是相同的。如第 2.3.1 节所述,这种重复性的工作是引发编程错误、增加调试时间的重要原因,而这一切在所有项目中都可以避免。如果能找到一种简单而且“类型安全”的方式,使得这组操作能够复用(reusable),那么,每个网络应用程序就都可以节省时间、免除麻烦。

方案 => 将多个函数合并为一个方法。有了这一简化,我们就不必在每一个项目中(重新)编写枯燥、容易出错的代码——例如,前文演示的“被动模式”连接初始化代码。被合并的代码行数越多,这个方案带来的好处也越大。因为一旦不需要进行具体的调用,出现“参数使用错误”的可能性就会降低。但这个方案最主要的优点是:要被调用的那组函数的信息和它们的调用次序被封装了起来,因而避免了出错的一大来源。

例如,ACE_SOCK_Acceptor 是一个“工厂”,用来被动地建立连接。它的 open()方法会调用 socket()、bind()和 listen()函数,用来创建“被动模式”的通信端点。因此,为了完成这些功能,应用程序只用提供下面的代码:

```
ACE_SOCK_Acceptor acceptor;
ACE_SOCK_Stream stream;

acceptor.open (ACE_INET_Addr (port));
acceptor.accept (stream);
// ...
```

同样,如果直接使用“基于 C”的 struct sockaddr 数据结构,也容易引发相关编程错误;ACE_INET_Addr 的构造函数将这些常见错误减至最低。例如,它会自动清除 sockaddr_in 地址结构(inet_addr_),并将端口号转换为“网络字节”顺序:

```
ACE_INET_Addr::ACE_INET_Addr (u_short port, long ip_addr) {
  memset (&this->inet_addr_, 0, sizeof this->inet_addr_);
  this->inet_addr_.sin_family = AF_INET;
  this->inet_addr_.sin_port = htons (port);
  memcpy (&this->inet_addr_.sin_addr, &ip_addr, sizeof
  ip_addr);
}
```

总的来说，这个方案产生的代码更简明、更精练，而且不容易出错；因为它运用 Wrapper Facade 模式，避免了“类型安全”问题。

A.3.2 将函数合并起来，置于统一的 Wrapper Facade 之下

问题：今天的计算平台都提供了常见的功能；但对于有些常见类型的功能，在各个平台上，访问它们的方式却大不相同。这有可能只是函数名称不同，但本质上是在做同样的事；也有可能完全是不同的函数，而且在不同的平台上必须以不同的顺序调用。这种变化的“核心 API”和“语义”使得应用程序难以被移植到新的平台上。

“多线程”就体现了 API 和语义在各种平台上的变化。在 POSIX 线程（Pthreads）、UNIX International（UI）线程、Win32 线程和实时操作系统等各个平台上，要创建具有不同属性的线程，其方式各异。下面列出了它们的一些主要不同：

- **函数名称**——在 Pthreads 中，创建线程的函数被称为 pthread_create()，在 UI 线程中为 thr_create()，在 Win32 中为 CreateThread()。

- **返回值**——一些线程 API 会在调用成功时返回一个线程 ID（或句柄），另外一些线程 API 则返回 0。出错时，一些 API 会简单地返回一个孤立的值（出错代码存储在别处），另一些 API 则会直接返回一个出错代码。

- **函数的数量**——为了指定线程的属性，如栈的大小或优先级，在 Win32 中，所有的属性值都要被传递给 CreateThread()，在 UI 线程中则要传递给 thr_create()。但在 Pthreads 中，我们要分别使用函数来创建、修改线程属性数据结构，然后将其传递给 pthread_create()函数，最后用另外的函数来销毁它。

- **函数调用的次序**——当需要多个函数调用时，它们使用的次序有时会不同。例如，Pthreads draft 4 API 需要在创建线程“之后”将其标记为“可连接”，而其他 Pthreads 实现则需要在此“之前”。

方案 => 将函数合并起来，置于统一的 Wrapper Facade 之下。这样，当需要将其移植至不同的平台上时，就可以正确地进行必要的修改。例如，为了处理“线程创建”的“多平台”需求，ACE_Thread_Manager::spawn()方法通过参数获得它需要的所有信息（请阅读第 A.3.3 节关于“参数”的重要原则），并以正确的顺序调用所有 OS 系统函数。这之中，我们考虑到了各个平台上的不同的返回值约定（convention），并

将其统一为整个 ACE 中使用的返回约定，即 0 表示成功，–1 表示失败，失败原因保存在 errno 中。采用这一方案，ACE 中以及用户程序中所有的线程创建，在所有平台上都可以通过一个方法调用来完成。

在统一（unifying）函数的过程中，面临的课题是：要确定“将要展示的接口”的粒度（granularity），以及应该合并（combine）哪些低级函数。在作决定时，要考虑以下两点：

• **可移植性**——在所有平台上，这个 wrapper facade 方法应能提供相同的语义。但不一定总能做到；例如，在第 5.4 节曾讨论过，不是所有的 OS 平台都允许你创建“系统范围”线程。但是，在很多场合下，可以模拟出“OS 平台本不支持”的特性，或忽略掉“所请求的功能和属性”中那些不重要的部分，从而使这个方法能够屏蔽这些差异。关于这一点，请看第 A.5 节更详细的介绍。

• **易于使用**——应该让调用者能够确定如何使用 wrapper façade，调用之前必须做哪些准备（例如，确定创建属性），返回值是什么，返回值具有什么含义，等等。较之拥有较多的方法，如果只拥有少量方法，用户将更容易使用它们。此外，请当心副作用（如内部内存分配）；调用者必须记得处理这些副作用。

A.3.3 重新组织参数次序，并提供缺省值

问题：OS 库中的系统函数必须面对各种可能的使用方式，还要允许用户访问所有的系统功能。因此，这些函数往往提供了很多“不常使用（seldom used）”的参数，但需要我们显式地为它们提供缺省值。此外，较之函数原型中的参数顺序，应用程序传递非缺省值给函数时，所用的顺序不一定总是与之匹配，这增加了编程中出错的可能性。而且，为了使 OS 库能够被各种语言调用，OS 库往往是用 C 实现的；这样，就难以运用一些消除这些问题的“语言特性”和“设计抽象”。

例如，UI 线程的 thr_create()函数接受六个参数：

1. 栈指针；
2. 栈的大小；
3. 入口点函数；
4. 入口点函数的 void *参数；
5. 用于创建线程的标志（flag）；
6. 被创建线程的标识符。

大多数应用程序需要的是缺省（default）的栈（stack）语义，所以，第 1 个参数和第 2 个参数通常都是 0。但是，开发者必须记得将 0 这个值传递进来，这很麻烦，还容易引发错误。对并发应用程序来说，通过“可连接（joinable）”语义创建线程也很常见——也就是说，当这个线程退出时，另一个线程会访问它并获得它的状态。所以，第 5 个参数需要一个常用值，但根据使用场合，它还会变化。第 3 个和第 4 个参数是最常变化的值。第 6 个参数由 thr_create()设置，所以必须由调用者提供。

方案 => 重新组织参数次序，并提供缺省值，以简化常见情形。在设计 wrapper facade 类时，要考虑以下两个重要因素：

• 要知道有哪些常见情形，并能设计你自己的接口，使得应用程序开发者在常见场合下能够很容易地使用它们。

• 可以指定和使用一种“面向对象”实现语言，例如 C++，并能利用这一语言的特性以及“面向对象”设计所带来的高级抽象。

针对以上介绍的问题，可以重新组织参数，将最常用的参数放在最前面，将最不经常使用的参数放在最后面，并为它们提供缺省值。

例如，可以对 ACE_Thread_Manager::spawn()的参数进行排序，将变化最频繁的参数（如“线程函数”和它的 void *参数）放在最前面。然后，为其他参数提供缺省值（例如，缺省的线程同步行为是 THR_JOINABLE）。如此一来，在常见场合下，大多数应用程序都只用传递最少量的、必要的信息。

C++缺省参数还能用于其他目的。例如，ACE_SOCK_Connector 类中的 connect()方法具有以下签名：

```
int ACE_SOCK_Connector::connect
  (ACE_SOCK_Stream &new_stream,
   const ACE_SOCK_Addr &remote_sap,
   ACE_Time_Value *timeout = 0,
   const ACE_Addr &local_sap = ACE_Addr::sap_any,
   int reuse_addr = 0,
   int flags = 0,
   int perms = 0,
   int protocol_family = PF_INET,
   int protocol = 0);
```

相比之下，ACE_TLI_Connector 的 connect()方法的原型稍有不同：

```
int ACE_TLI_Connector::connect
  (ACE_TLI_Stream &new_stream,
   const ACE_Addr &remote_sap,
   ACE_Time_Value *timeout = 0,
   const ACE_Addr &local_sap = ACE_Addr::sap_any,
   int reuse_addr = 0,
   int flags = O_RDWR,
   int perms = 0,
   const char device[] = ACE_TLI_TCP_DEVICE,
   struct t_info *info = 0,
   int rw_flag = 1,
   struct netbuf *udata = 0,
   struct netbuf *opt = 0);
```

实际上，在各个调用中，只有 connect()的头几个参数会变化。所以，为了简化编程，这些类中的 connect()方法可以使用缺省值；这样，开发者就不必每次提供这些值了。如此一来，这两个类在常见场合下的使用方式几乎是一样的。例如，ACE_SOCK_Connector 大致如下：

```
ACE_SOCK_Stream stream;
ACE_SOCK_Connector connector;

// Compiler supplies default values.
connector.connect (stream, ACE_INET_Addr (port, host));
// ...
```

ACE_TLI_Connector 大致如下：

```
ACE_TLI_Stream stream;
ACE_TLI_Connector connector;

// Compiler supplies default values.
connector.connect (stream, ACE_INET_Addr (port, host));
// ...
```

缺省参数带来的常见函数签名可以同[SH]中讨论的“参数化类型（parameterized type）”结合起来使用，以增强 *generative programming（生成式编程）*[CE00]，并通过“参数化类型”来处理变化，如第 A.5.3 节所述。

A.3.4 将“有内在关系的对象”显式关联起来

问题：由于“OS 级”程序库具有“通用（general-purpose）”特征，所以多个函数和数据结构之间往往存在依赖性。但是，因为这些依赖性是隐式（implicit）的，很难做到自动识别和实施这种依赖性。在第 2.3.1 节，我们通过 socket()、bind()、listen()和 accept()函数之间的关系说明了这一点。如果有多个数据对象，它们有时一起使用，有时又单独使用，这个问题就会加剧。

例如，在使用“线程”工具（facility）时，互斥体（mutex）是常用的同步机制。互斥体还会同条件变量一起使用；也就是说，在条件变量的生存期内，条件变量和互斥体通常关联在一起。在这方面，Pthreads API 很容易出错，因为它没有将这种关联明确化。pthread_cond_*()函数的签名接受一个 pthread_mutex_t*参数。因而，这一语法本身没能表示 pthread_cond_t 和 pthread_mutex_t 之间的紧密耦合。此外，Pthreads 条件变量 API 使用起来很麻烦，因为每次调用 pthread_cond_wait()函数的时候，互斥体都得作为一个参数传递进来。

方案 => 将“有内在关系的对象”显式关联起来，使应用程序开发者只用记住最少的细节。作为 wrapper facade 的设计者，你的任务是使类用户的工作更容易正确完成，且更难以（或不可能）出错。为达到这一目标，设计一个“将多个相关对象封装起来”的类是一种不错的技术，因为它将“有内在关系的对象”之间的关联体现在代码中。

在第 A.2 节，我们讲述了“运用 wrapper facade 增强类型安全性”的原则：

这里介绍的原则较之前又进了一步。在 A.2 节，我们的目标是运用 C++类型系统来保证“强类型检查（strong type-checking）”；在这里，我们的目标是运用 C++特性实施“强类型对象”之间的依赖性。例如，为了实施上面例子中的条件变量和它的互斥体之间的关联，在第 10.6 节讲述的 ACE_Condition_Thread_Mutex 类中，我们提供了一个条件变量，还有一个互斥体引用：

```
class ACE_Condition_Thread_Mutex {
  // ....
private:
  ACE_cond_t cond_; // Condition variable instance.
  ACE_Thread_Mutex &mutex_; // Reference to a mutex lock.
};
```

这个类的构造函数需要一个“ACE_Thread_Mutex 对象的引用”，强制用户正确地关联这个互斥体，如下所示：

```
ACE_Condition_Thread_Mutex::ACE_Condition_Thread_Mutex
  (const ACE_Thread_Mutex &m): mutex_ (m) { /* ... */ }
```

其他任何用法都会被 C++编译器捕获，并被拒绝。

A.4 通过继承结构提高设计的清晰性和可扩充性

继承（inheritance）是“面向对象”设计技术及编程语言的一个基本特征 [Mey97]。尽管经验表明，层次过多的继承结构会带来麻烦 [GHJV95]；但是，如果能明智地使用继承，并通过继承方式构造软件抽象，还是很有好处的。

A.4.1 将一维的 API 用继承结构来代替

问题：一定程度上，Socket API 的复杂性源于其“过于广泛”的、“一维（one-dimensional）”的设计方式 [Mar64]。例如，所有的函数都出现在同一个抽象级，如图 A.1 所示。采用这种“一维”设计，学习使用 Socket API 时将需要花费更多的精力。例如，为了编写一个网络应用程序，程序员必须理解 Socket API 中的大多数知识，即使只使用其中的一部分。此外，功能的分散会导致代码的重复。

```
socket ( )
bind ( )
connect ( )
listen ( )
accept ( )
read ( )
write ( )
readv ( )
writev ( )
recv ( )
send ( )
recvfrom ( )
sendto ( )
recvmsg ( )
sendmsg ( )
setsockopt ( )
getsockopt ( )
getpeername ( )
getsockname ( )
gethostbyname ( )
getservbyname ( )
```

图 A.1：Socket API 中的函数

方案 => 将一维的 API 用继承结构来代替，从而最大程度地提供“可复用性”和“代码共享”。这条原则的中心是：用“结构上有关系的类”重新构造（restructure）已有的一维 API。例如，ACE Socket wrapper facade 将图 A.1 所示的 Socket API 函数聚合在一起，构成了图 3.1 所示的类继承结构。“继承”提高了代码的可复用性，增强了 ACE 的模块化：

- **通过基类（base class）表达相似性（similarity）**。分层设计可以避免不必要的代码重复；因为基类可以表达“继承结构根部提供的公共机制”。例如，ACE IPC wrapper facade 共享了继承结构根部（例如，ACE_IPC_SAP 和 ACE_SOCK 基类中）的机制。这些机制包括“打开/关闭”及“设置/获取”底层 I/O 句柄的方法，以及某些“为所有派生的 wrapper facade 类所公有”的“选项管理”函数。

- **通过派生类（derived class）表达差异性（difference）**。位于继承结构叶节点的子类实现的是专有方法；这些方法根据所提供的通信类型，对功能作了定制（customized）。例如，ACE_SOCK 的子类提供了“流”通信与“数据报”通信、“本地”通信与“远程”通信。

A.4.2 将“伪继承”用“C++继承”来代替

问题：由于继承结构上的抽象性不足，很多“C 语言级”系统编程 API 很容易引发错误。例如，Socket API 的“网络地址”机制使用的是 C 语言的结构和类型转换，

它们为“Internet 领域”及“UNIX 领域”地址提供了笨拙的“继承”形式。最通用的 socket 地址结构是 sockaddr，它定义了一个“伪基类（pseudo base class）”，用以指定地址信息。它的第一个字段表示的是指定的地址簇，第二个字段则是一个让人费解（quasi-opaque）的字节数组。于是，各个特定的地址簇结构必须通过覆盖这个字节数组来指定地址信息。在第 2.3.1 节和第 3.2 节我们曾作过介绍，这种“伪继承”设计会产生很多微妙的 bug。特别是，类型转换的使用——结合“弱类型”的、“基于句柄”的 Socket API——会使得编译器很难在编译时发现程序员的错误。

方案 => 将“伪继承”用“C++继承”来代替。这条原则并不是在强制应用程序开发者去和“类型转换”与“类型覆盖”作斗争，而是要利用 C++中的“面向对象”语言特性。例如，ACE 定义了第 3.2 节讲述的 ACE_Addr 类继承结构，为各种网络地址格式（例如，图 A.2 所示的 Internet 及 UNIX 领域地址、STREAM 管道、文件和设备）提供了“类型安全”的 wrapper facade。每一个子类都在一个公共 API 之后隐藏了地址簇特有的细节。

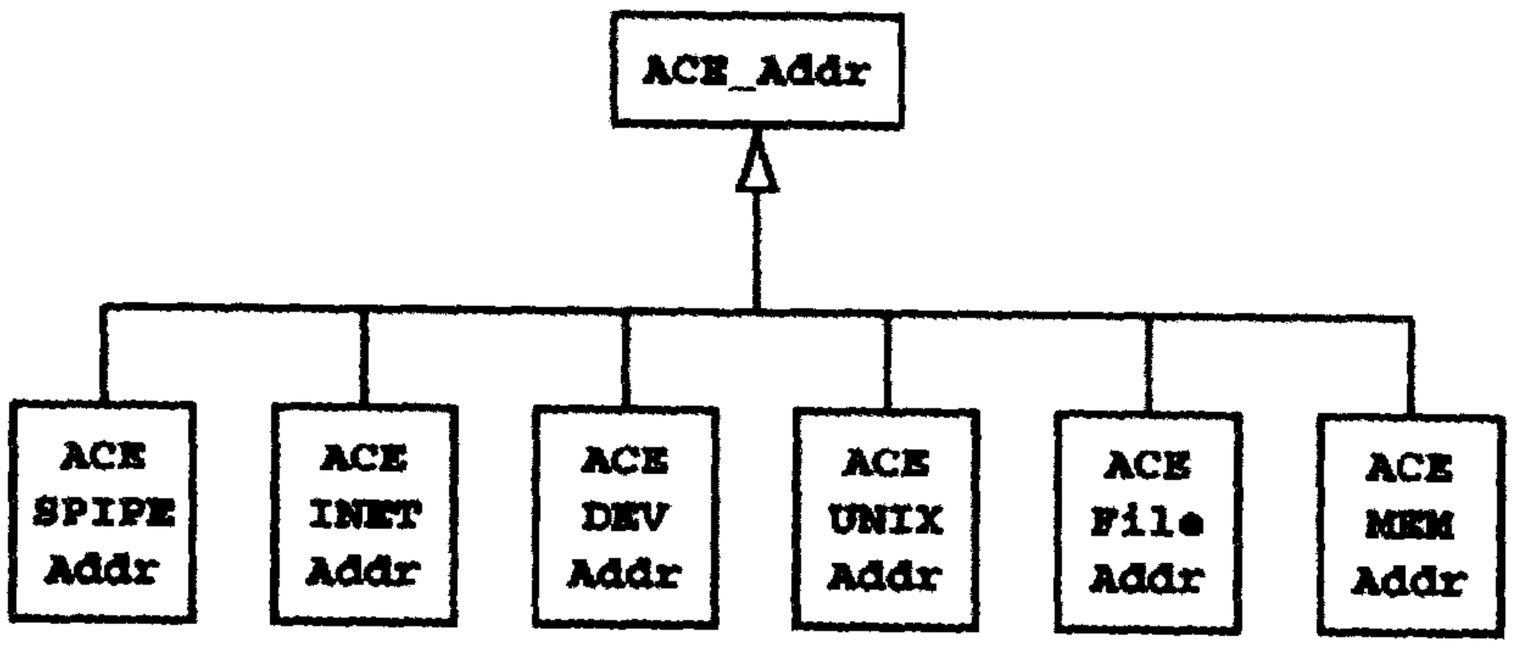


图 A.2：ACE IPC 地址的继承结构

A.5 尽可能地隐藏平台间的差异

“多平台”软件开发对类的设计者提出了很多挑战，因为各个平台提供的功能相差很大。这些差异往往表现为以下某一形式：

- **“缺乏”并且“不可能”实现某些功能**——有些平台完全没有提供其他

平台所提供的功能。例如，一些平台没有“僵死进程（zombie）”的概念（参阅第 8 章）；因而，在这些平台上，没有办法也没有必要提供某种方法来避免“僵死进程”。

- **“缺乏”但可以“模拟”某些功能**——有些平台没有提供其他平台所提供的能力，但却提供了其他功能，足以模拟出缺少的功能。例如，在某些 OS 平台上，第 10 章介绍的“条件变量”和“readers/writer 锁”可以通过模拟的方法来实现。

- **“缺乏”某些功能，但存在同样不错的“类似”特性**——有些平台没有提供某项能力，如 Socket API，但提供了等价的功能，如 TLI。有些情况下，两项功能可能都存在，但在这个平台上，其中一项功能的实现会比另一个更出色。例如，其中一个可能具有更佳的性能、更少的缺陷。

本节讲述如何处理这些情况，从而得到一组 C++类，使得我们可以更方便地开发可移植的网络应用程序。

A.5.1 允许源代码支持任何有益的功能

问题：有时候，各个平台的功能集（feature set）所存在的差异，使得 wrapper facade 无法隐藏它们。例如，在一个平台的“功能集”中，如果根本不存在核心线程（kernel-thread）语义，wrapper facade 就无法神奇地将其发明创造出来。因而，类的设计者面临一个问题：如果 OS 平台上缺乏某些功能，则设计类时是否也应该缺少相应的方法或方法集（set of methods）？

在将软件向多个平台移植时，最终会碰到这么一个平台：某项重要的功能在这个平台上没有提供支持。对于这个问题，我们的第一反应可能是：避免提供“对应于那项功能”的类或类的方法，强制编译器在编译时发现问题。如果缺少的功能对应用程序来说极为重要，这个选择就是合适的。例如，对很多网络应用程序来说，“创建新进程”是一项核心功能。如果某个平台没有提供“多进程”能力，在编译时知道这种情况将是有利的。但是，在很多情况下，应用程序应能对运行时任何可能的问题作出处理和反应；此时，应能允许尝试其他的方案。

方案 => 允许源代码支持任何有益的功能。有些情况下，即使源代码中使用的功能在某一平台上没有被提供，并且无法模拟，此时最好还是要允许源代码正确通过编

译。这可能是因为：

• **希望能够在运行时发现问题，并允许应用程序选择其他应对方案。**这种情形通常发生在类库的设计中，因为设计者无法预计类的所有使用场合。这种情况下，较好的做法是实现一个方法，让它向调用者返回一个“没有实现”的标志。这样一来，调用者就有机会选择另一种技术和功能，用来报告错误并继续执行，或终止。基于这一目的，ACE 定义了一个名为 ACE_NOTSUP_RETURN 的宏（macro）。

•**“忽略缺少的功能”产生的效果和“拥有这项功能”的效果一样。**对于这种情况，请考虑这样一个 UNIX 应用程序，它使用了 ACE_Process_Options::avoid_zombies()方法，用来避免处理“僵死进程”（参见第 168 页）。如果将这个应用程序移植到 Win32，而 Win32 上没有“僵死进程”的概念。这种情况下，调用 avoid_zombies()方法并让它返回成功是完全可取的，因为这个方法不做任何事情，就已经实现了它的本意。

A.5.2 模拟出缺少的功能

问题：由于今天的计算平台实现了广泛的功能和标准，因而往往会造成功能集（feature set）的差异。在将应用程序移植到新的平台上时，这种差异会导致某块功能区的缺乏，而在前一个平台上，某个项目（project）还依赖于该功能区中这些功能。但是，在对“隐藏这些功能之后”的概念作仔细分析之后，往往可以发现另外某些功能；如果将这些功能结合起来，就可以模拟出缺少的功能。

和普遍认识相反：标准（standard）并不是编写可移植软件的灵丹妙药。标准有很多，各个平台实现的是不同的标准集，而且在不同的 OS 版本之间还会经常变化。即使在同一个标准之中，某些特性往往也是可选的。例如，在 Pthreads 中，“信号量”和“实时调度级别”功能是可选部分。

如果某一软件是针对某一平台或某一组平台开发的，则在将它移植到新的平台上时，往往会出现“软件中用到的功能在新的平台上不存在”的情况。此时，如果为了避开这一不存在的功能而对程序作修改，将会给整个项目带来*连锁反应（ripple effect）*。但是，在对新平台的“功能集”作仔细分析之后，可能会发现这么一组功能：如果将它们结合起来，就可以模拟出不存在的功能。因此，如果在一个项目中能够聪明地使用 C++ wrapper facade class，则一系列代价昂贵的设计和代码修改就有可能避免。

方案 => 模拟出缺少的功能。根据 Wrapper Facade 模式设计的类能以方便、“类型安全”的方式封装 OS 平台的原始功能。这些类也可以包含一些原始代码；这些代码可以运用平台现有的能力，模拟出一些没有提供的功能。例如，下面的 C++代码演示的是，在原本没有提供“进程范围”信号量 wrapper facade 的 OS 平台上，如何利用 ACE“条件变量”和“互斥体”类去实现它。

```
class ACE_Thread_Semaphore
{
private:
  ACE_Thread_Mutex mutex_; // Serialize access.
  // Wait for <count_> to become non-zero.
  ACE_Condition_Thread_Mutex count_nonzero_;
  u_long count_; // Keep track of the semaphore count.
  u_long waiters_; // Keeps track of the number of waiters.
public:
  ACE_Thread_Semaphore (u_int count = 1)
    : count_nonzero_ (mutex_), // Associate mutex and condition.
      count_ (count),
      waiters_ (0) {}
```

请注意，在 count_nonzero_的初始化表达式中，mutex_对象如何被绑定到它自身上。这遵循了第 A.3.4 节讲述的一条原则，即：将“有内在联系的对象”显式地关联起来。

acquire()方法阻塞“调用者线程”，直至信号量计数大于 0，如下所示：

```
  int acquire () {
    ACE_GUARD_RETURN (ACE_Thread_Mutex, guard, mutex_, -1);
    int result = 0;

    // Count # of waiters so we can signal them in <release()>.
    waiters_++;

    // Put calling thread to sleep waiting on semaphore.
    while (count_ == 0 && result == 0)
      // Release/reacquire <mutex_>
      result = count_nonzero_.wait ();
```

```
    --waiters_;
    if (result == 0) --count_;
    return result;
  }
```

出于完整性考虑，在下面实现 ACE_Thread_Semaphore 的 release()方法；它递增信号量计数，潜在地解除了“等待在 count_nonzero_条件变量上”的阻塞线程。

```
  int release () {
    ACE_GUARD_RETURN (ACE_Thread_Mutex, guard, mutex_, -1);

    // Notify waiters that the semaphore has been released.
    if (waiters_ > 0) count_nonzero_.signal ();

    ++count_;
    return 0;
  }
  // ... Other methods omitted ...
};
```

请注意，ACE_Thread_Semaphore 类的 acquire()和 release()方法都使用了 ACE_GUARD_RETURN 宏（如副栏 22 所述）；这个宏封装了 Scoped Locking 技术[SSRB00]，从而保证了 mutex_的自动加锁和解锁。这个设计是“简化常见情形”原则（如第 A.3 节所述）的又一个例子。

A.5.3 通过“参数化类型”处理变化性

问题：网络应用程序和中间件常常需要运行在广泛的平台上，但在这些平台上，OS 能力的可用性和效率相差很大。例如，某些 OS 平台可能有不同的底层网络 API，如 Socket 而非 TLI，或相反。此外，在不同的 OS 平台上，实现这些 API 时的效率多少也会有所不同。在这样的异种平台上撰写可复用软件，必须解决以下问题：

• 不同的应用程序可能需要配置不同的中间件策略，如不同的同步或 IPC 机制。添加“新策略”或“改进策略”应当简单直接。理想情况下，应用程序的每一个函数或类都应该只有一种副本，以避免版本差异。

• 为了应付变化而使用的机制不应过度影响运行时性能。特别是，由于虚方法具有间接性 [HLS97]，继承和动态绑定会给运行时带来额外开销。

方案 => 通过“参数化类型”处理变化性，而不是通过“继承”和“动态绑定”。参数化类型（parameterized type）消除了应用程序对特定策略（如同步或 IPC API）的依赖，同时不会带来运行时开销。虽然参数化类型会招致编译和链接时开销，但它们一般可以被编译为高效的代码 [Bja00]。

例如，通过 C++类（而不是孤立的 C 函数）来封装 Socket API，有助于提高可移植性，因为它允许通过“参数化类型”批量替换网络编程机制。下面的代码演示了这一原则。它运用 generative（生成式）[CE00]及 generic（泛型）[Ale01]编程技术来修改 echo_server()，使其成为一个 C++函数模板。

```
template <class ACCEPTOR>
int echo_server (const typename ACCEPTOR::PEER_ADDR &addr)
{
  // Connection factory.
  ACCEPTOR acceptor;
  // Data transfer object.
  typename ACCEPTOR::PEER_STREAM peer_stream;
  // Peer address object.
  typename ACCEPTOR::PEER_ADDR peer_addr;
  int result = 0;

  // Initialize passive mode server and accept new connection.
  if (acceptor.open (addr) != -1
      && acceptor.accept (peer_stream, &peer_addr) != -1) {
    char buf[BUFSIZ];
    for (size_t n; (n = peer_stream.recv (buf, sizeof buf)) > 0;)
      if (peer_stream.send_n (buf, n) != n) {
        result = -1;
        break;
      }

    peer_stream.close ();
  }
  return result;
}
```

通过使用 ACE 和 C++模板，应用程序可以被参数化：根据底层 OS 平台的特性，它可以透明地使用 C++ Socket 或 TLI wrapper facade：

```
// Conditionally select IPC mechanism.
#if defined (USE_SOCKETS)
typedef ACE_SOCK_Acceptor ACCEPTOR;
#elif defined (USE_TLI)
typedef ACE_TLI_Acceptor ACCEPTOR;
#endif /* USE_SOCKETS. */

int driver_function (u_short port_num)
{
  // ...

  // Invoke the <echo_server()> with appropriate network
  // programming APIs. Note use of template traits for <addr>.
  typename ACCEPTOR::PEER_ADDR addr (port_num);
  echo_server<ACCEPTOR> (addr);
}
```

这一技术之所以可行，基于以下原因：

• ACE C++ Socket 及 TLI wrapper facade 类通过一个共同的签名（signature）展示“面向对象”接口。如果接口的设计最初没有做到一致性（consistency），可以运用 Adapter 模式 [GHJV95]，使得它们保持一致，如第 A.3.3 节所述。

• C++模板支持“基于签名”的“类型一致性（type conformance）”，不需要类型参数（type parameter）包容所有可能的功能。相反，模板将应用程序代码参数化；这些应用程序代码本来只是调用一个“方法子集（subset of methods）”，而这些方法对各种网络编程方法来说是公共的，例如，open()、close()、send()和 recv()。

总体来说，较之其他手段（例如，实现多个版本的 echo_server()函数，或在应用程序代码中到处使用条件编译指令），“参数化类型”的破坏性（intrusive）更弱，但更具可扩充性（extensible）。

A.6 针对效率实施优化

针对较高级的应用程序和 GUI 框架（framework）使用的设计原则不一定适用于较低级的网络编程工具包。特别是，性能上的因素经常会排除某些语言特性、技术和模式的使用。本节就这方面问题进行了讨论。

A.6.1 针对效率设计 Wrapper Facade

问题：尽管“面向对象”技术和 C++已经成功应用多年，但在一些应用领域一直存在这样一个认识：它们不适合应用到“具有严格性能要求”的系统中。例如，在需要“可预测的运行时行为”、低延迟、小存储空间的实时嵌入式应用程序中，面向对象编程语言的“动态绑定”和原始的“异常处理”特性会带来问题。但是，很多实时应用领域（如航空、呼叫处理、过程控制、分布式交互仿真）可以受益于灵活、可移植的“主机设施中间件”。因此，针对这些领域开发面向对象软件，应避免使用“会增加无谓开销”的语言特性和模式。

方案 => 针对效率设计 Wrapper Facade。ACE 运用下面的技术来保证 wrapper facade 的高效：

• 很多 ACE wrapper facade 是“具体类型（concrete type）”，也就是说，它们的方法是非虚方法（nonvirtual）。这种设计消除了分发（dispatch）“动态绑定方法”的开销，提高了方法（method）内联（inlining）的机会，还可以将对象放在合适的共享内存中。

• 所有 ACE IPC wrapper facade 都直接包含了 open()和 close()方法，而且，它们的析构函数不关闭句柄。这一设计可以防止 I/O 句柄作为“值（value）”从一个对象传递给另一个对象时，句柄会被过早关闭，从而导致发生微妙的错误。ACE 的 wrapper facade 有意避免了 Bridge 模式 [GHJV95]的使用，因为那会招致动态内存分配，并会降低而不是提高效率。相反，我们在 ACE 中使用的原则是：不在类继承结构的根部（root）执行某些类型的隐式操作（implicit operation）；相反，只是在需要的时候，由较高层抽象体提供这些功能。例如，在第 4.4.1 节的 Logging_Server 类中，“接受者”socket 句柄是在这个类的析构函数中关闭的；在那个抽象层上，这种用法很常见。

A.6.2 内联影响性能的方法

问题：通过提供额外一层抽象性，wrapper facade 提高了原始的“C 语言级”函数 API 的“可移植性”和“类型安全性”。但是，如果这些 wrapper facade 缺乏高效的实现，很多网络应用程序开发者将不会用它们取代现有的 C 网络 API。网络应用程序往往对时间很敏感，开发者必须小心地降低延迟、提高响应时间，因为网络本身已经加剧了这些问题。对 OS 平台 API 的调用往往是执行路径中对时间影响很大的环节，所以，网络应用程序开发者必须提防任何“有可能降低效率”的开销。

方案 => 内联影响性能的方法，从而将“由于类型安全性的提高”所导致的性能开销降至最低程度，甚至消除。ACE 广泛使用了 C++内联，用来消除 OS 适应层和 C++ wrapper facade 中的额外抽象所导致的“方法调用”开销。在“影响性能的执行路径”中使用的方法，如 ACE_SOCK_Stream 的 recv()和 send()方法，被指定为 C++内联函数。内联带来了时间及空间效率，因为这些方法代码很少（例如，一个方法只有 1~2 行代码）。此外，在 ACE 中影响性能的地方，也很少使用虚方法，因为很多 C++编译器的优化程序不能完全消除虚方法的开销 [HLS97]。

A.6.3 在系统级工具包中避免使用异常处理

问题：有经验的 C++程序员会注意到，在本书及 ACE 中，类的定义都没有包含异常规范（exception specification）。初看上去，这似乎是个缺陷，因为在调用链（call chain）中处理错误标志（error indication）很困难，而这正是使用 C++“异常处理”机制的基本动机。但是，在系统级工具包中使用原始的 C++异常处理机制存在两个问题：

1. **可移植性**——ACE 始于 1991 年，也就是说，在异常（exception）加入到 C++ 标准之前。时至今日，编译器对异常的支持仍不统一，而且，由于对旧技术的依赖性，不是所有的最终用户都能移植到新的编译器上。ACE 在广泛的编译器和平台上都要能够运行；所以，对原始“异常”的任何依赖将妨碍它“可移植地”在某些环境下使用。

2. **性能**——很多编译器会带来时间和空间开销，这对某些类型的应用程序来说是无法接受的。例如，对于嵌入式系统中的实时应用程序来说，原始的“异常处理”所

导致的内存空间和执行时间上的开销是绝不允许的。因此，即使某个平台支持原始的“异常处理”机制，但在那个平台上，实时及嵌入式程序还是不能使用它。

方案 => 避免在系统级工具包中使用异常处理。基于以上原因，ACE 不会抛出异常。在内部，它通过使用 Thread-Specific Storage 模式 [SSRB00]，在调用者和被调用者之间传递错误状态（error status）信息。采用这一模式，线程中的连续操作（sequential operation）就能够原子地（atomically）访问公共数据，而且每次访问都不会招致锁定开销（locking overhead）。通过将出错信息放在“线程专有存储空间”，每个线程都能可靠地设置、检测线程中的操作状态，并且无需使用额外的锁（lock）或复杂的同步协议。

有些系统希望在应用软件中使用异常；对于这样的系统，ACE 可以和原始的“C++异常支持”一同参与编译。在那些“使用 make 命令生成（build）ACE”的平台中，只用在命令行或 platform_macros.GNU 文件中添加一条语句 exceptions=1。

A.7 小结

本附录介绍了一些设计原则，整个 ACE 都遵循这些原则，以实现 ACE 的目标——提供主机基础设施中间件（host infrastructure middleware）所具有的好处（参阅第 0.3.1 节）。这些原则有一些在 generative programming 社区 [CE00]已经为人所熟知，但对网络程序员来说还很陌生。至于其他原则，有关 Wrapper Facade 模式的讲述中，我们提供了更抽象的介绍，见[SSRB00]。ACE 运用了很多精心构思、久经考验的设计原理，这使得 ACE 成为了一个灵活、强大、可移植的工具包。在数以千计的项目中，ACE 帮助它们避免了本书讲述的很多“和偶发复杂性相关”的缺陷。

附录 B

ACE 的过去、现在和将来

本章提要

本附录提供了 ACE 的简史，介绍了它的起源，以及过去十年中作为开放源码（open-source）的演变过程；还介绍了基于 ACE 并和 ACE 捆绑在一起的、“符合标准”的中间件；最后，展望了我们所能看到的 ACE 的未来。

B.1 ACE 的演变

Eric Raymond——开放源码运动 [O'R98]的先驱者——喜欢这么说：“每一个好的软件作品都始于开发者需要解决自身的问题”[Ray01]。虽然并不是所有情况都是如此，但这句话对 ACE 来说十分恰当。本节将讲述 ACE 的演变过程——从最初作为一位研究者为简化自己的工作而开发的工具，到成为世界上最具可移植性、最为广泛运用的C++网络编程工具包之一。

B.1.1 起因

1990 年，Doug Schmidt 中断了他在 University of California, Irvine (UCI)的博士学位的学习，加盟了一家在硅谷成立的公司，Independence Technologies Inc. (ITI)（后被 BEA Systems, Inc.收购）。ITI 致力于基于 UNIX 的联机事务处理（OLTP, online transaction processing），这在 1990 年还是新鲜事物。正是在 ITI，Doug 受到了 Richard Stevens 的经典著作《UNIX Network Programming》的影响，他吸收并运用书中的知识，用 C 和 C++开发了一个 OLTP 中间件平台。

1991 年，为了完成有关“并发式网络协议处理”的论文 [SS95]，Doug 回到了 UCI。他的博士学位课题专注了 A *Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment* (ADAPTIVE) [SBS93]的设计和优化。ADAPTIVE 提供了可定制的轻量级自适应协议机（protocol machine），有助于改善点对点（end-to-end）应用的性能，并降低传输系统的开销 [SS93]。

20 世纪 90 年代初，在编写诸如 ADAPTIVE 之类的系统软件时，必须面对两个讨厌的“偶发复杂性”问题：

1. ADAPTIVE 中的协议代码是用“面向对象”设计技术和 C++编写的。但是，在访问 OS 资源，如进程、线程、锁、socket、共享内存、DLL、文件时，还是需要回过头来求助于 C 函数 API 和算法设计。

2. 为了处理常见的网络编程任务，如连接的建立、同步事件多路分离和发送、同步、并发体系，相同的“样本”代码就得编写多次。

创建 ADAPTIVE 通信环境（ACE）就是为了攻克这两个难题。ACE 的第一个版本公开发布于 1992 年，它运行在 SunOS 4.x 和 5.x 上。这个版本包含大约 10,000 行代码，综合运用了 C++特性、高级 OS 机制和模式，提供了一个可扩展的、高效的“面向对象”网络软件开发工具包。

如果 Doug 没有以开放源码的模式发布 ACE，那么在他完成博士学位之后，ACE 之花很可能已经凋零。幸运的是，Doug 曾投身于自由软件社区多年，编写过 GNU GPERF perfect hash function generator [Sch90]，并同 Doug Lea 一道开发了 GNU LIBG++ 库的一部分 [Lea88]。因而，他十分清楚地知道，建立一个以“开放源码”手段和工具为中心的社区 [SP01]，会带来极大的好处。开放源码开发模式的主要优势是，能够拓展出庞大的用户社区。在社区中，应用程序开发者和最终用户可以在质量保证、文档、技术支持等方面相互协助。

B.1.2 转折点

到了 1994 年，ACE 已经日趋成熟，被应用到数十个商业项目中，包括 Ericsson [SS94] 和 Iridium [Sch00]的网络管理系统。那时，Doug 已经成为了 Washington University, St. Louis 的助教。在这段过渡时期，商业应用领域掀起的一个浪潮是：如何利用中间件来提高网络应用软件的可移植性和灵活性。

ACE 就处于中间件浪潮的前端。于是，Doug 和他在 Washington University 的研究小组收到了很多有远见的赞助商提供的基金，这些赞助商包括：ATDesk、BBN、Boeing、Cisco、DARPA、Ericsson、Hughes、Icomverse、Irdium、Kodak、Krones、Lockheed Martin、Lucent、Motorola、Nokia、Nortel、NSF、Raytheon、SAIC、Siemens MED、Siemens SCR、Siemens ZT、Sprint、Telcordia。这些赞助商认识到，为了克服核心网络应用软件开发中的困难，重新独立地去发现和创造一些专门的方案，代价将十分昂贵，而且耗时。幸运的是，ACE 项目的参与者已经发现、整理、具体化了很多重要的模式和框架（framework），这个模式和框架可用来解决这些困难。这样，Doug 的赞助商在开发、维护他们的网络应用程序时可以运用这些模式和框架，减少了很多麻烦和开销。

接下来的六年里，大约 700 万美元的赞助资金使得 Doug 可以雇请到数十名研究生和全职人员。这些雇员有很多都在 Washington University, St. Louis 的 Distributed Object Computing (DOC) Groups 工作多年；后来，随着 Doug 返回 UCI 担任 Electrical and Computer Engineering Department 副教授，DOC 也迁至 UCI。总体来说，DOC Groups 做出了以下贡献：

1. 大大扩展了 ACE 的能力，比最初的规模扩大了约 25 倍，增加了数百个类和框架（framework）。

2. 将 ACE 移植到了数十个新的 OS/编译器平台上；请看副栏 1 中关于这些平台的清单。

3. 撰写了大量杂志文章和技术论文，阐述了 ACE 工具包中的 wrapper facade 和框架（framework）的模式（pattern）和性能（performance）。因为可以在 http://www.cs.wustl.edu/~schmidt/publications.html 下载到这些论文，全世界的开发者都可以评估 ACE 的应用领域和品质，从而确定 ACE 能够在多大程度上满足自己的需要。

DOC Group 成员和 ACE 用户群的贡献在于，他们将 ACE 从“一名开发者的个人爱好”转变为了“世界上运用最广泛的”、“可用于在广泛的软硬件平台上进行并发式面向对象网络编程”的 C++框架（framework）之一。

B.1.3 跨越断层

1996 年，当 Doug 和他在 Washington University 的小组开发 ACE 3.x 版本时，Steve Huston 正在从事咨询和网络应用程序开发工作。在一个项目中，需要在很短的周期内开发一个 C++网络软件。通过 Web 搜索之后，他发现了 ACE。遗憾的是，ACE 当时还不能移植到他那个项目的平台（Unixware）上。但是，如果不使用 ACE 提供的功能，这个项目绝对无法完成；所以，Steve 开始着手将 ACE 移植到 Unixware 上。在这个过程中，他不但将 ACE 移植到了新的平台上，还学习了足够的 ACE 知识来开展自己的工作，最后开发出了自己的系统；而这所有时间加起来还是少于白手起家编写网络软件所花的时间。新系统表现得十分高效。这是 ACE 众多成功案例中的第一个 [Gro]。

认识到 ACE 的巨大优势之后，Steve 继续关注 DOC Group 和 ACE 邮件列表。虽然 ACE 在持续改进，但它大多还只是为研究人员和产业研发团体所使用。但 ACE 具有如此实用性，决不应该留在研究人员的手中；Steve 认识到，为了使 ACE 跨越断层 [Moo91]继而支持主流商业软件开发者，高品质的支持迫在眉睫。车轮在 1996 年底开始转动；到了 1997 年中，Steve 已经将他的公司更名为 Riverace Corporation（http://www.riverace.com）；公司的业务专注在产品支持和服务上，目的是使 ACE 更易于学习和使用。

今天，Riverace 所提供的支持和咨询服务，已经帮助很多公司在项目开发中最有效地利用了 ACE。Riverace 继续将 ACE 作为一种开放源码产品，不断提高它的品质；它所提供的商业级支持，有助于将 ACE 的应用拓展到世界范围的企业软件开发中。今天的 ACE 具有商业软件的“外观和品质”；但无需任何“使用”或“开发”上的许可费用，就可以尽享它的所有好处。副栏 24 概要介绍了 ACE 的开放源码许可协议。

副栏 24：ACE 开放源码许可协议

ACE 开放源码许可协议类似于所谓 BSD UNIX 开放源码许可协议。用户可以自由、永久、最终使用、修改、复制、发布 ACE 源代码和“从源代码编译出的目标代码”，还可以复制、发布此软件的修改版本。尤其是，ACE 可以用于私有软件，用户没有义务再次分发“通过 ACE 构造的”或“派生于 ACE 的”任何源代码。完整的 ACE 版权及许可信息提供在 ACE 发行版本的$ACE_ROOT/COPYING 文件中。

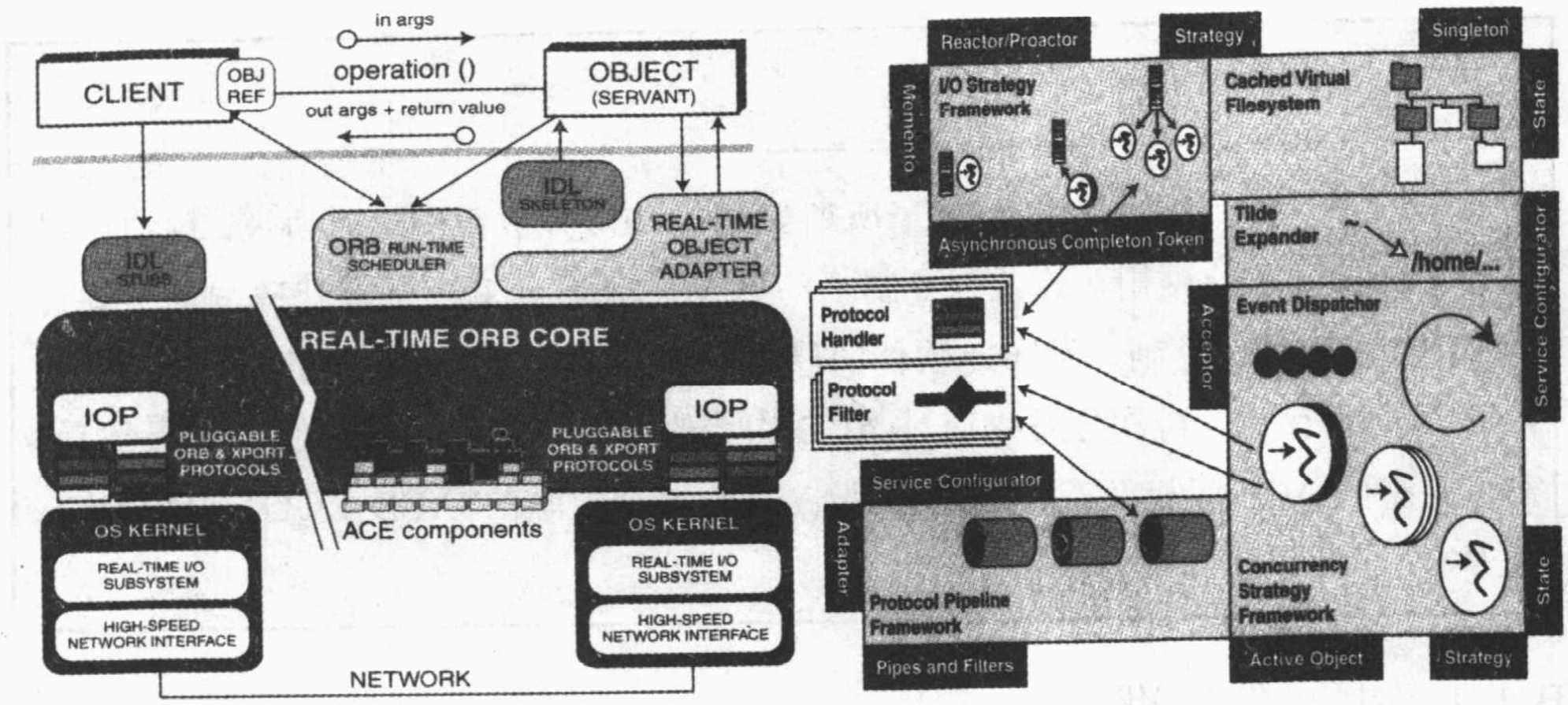
B.1.4 中间件标准

到了 20 世纪 90 年代后期，信息产业日趋商品化；也就是说，主要软硬件产品都变得更快、更便宜、更好，其发展速度让人无法预测。过去十年里，我们都受益于硬件的商品化；这些硬件包括 CPU 和存储设备，以及网络设备，如 IP 路由器。最近一段时期，由于以下标准日益成熟，很多软件层（software layer）和组件（component）也日趋商品化：

- **编程语言**，如 Java 和 C++；
- **操作环境**，如 POSIX 和 Java 虚拟机；
- **中间件**，如 CORBA、Enterprise Java Beans 和.NET Web services [SS01]。

这些软件产品的商品化所带来的结果是，被高壁垒长期封锁的产业，如电信和航空，更容易受破坏性技术（disruptive technology）[Chr97]和全球竞争的冲击，使得它们的价格徘徊在赢利的边缘。

Doug 感到，软件产业的长期成功必须依靠有效的开放标准；所以，他启动了两个副项目，致力于将 ACE 的研究成果转化为开放的中间件标准。在本节，将讲述这两个副项目的成果，即两个基于标准的中间件工具包——TAO 和 JAWS，如图 B.1 所示。TAO 和 JAWS 是用 ACE 提供的 wrapper facade 和框架（framework）开发而成的，如下所述。



(1) THE ACE ORB (TAO) (2) The JAWS Web Server Framework

图 B.1：基于 ACE 且符合标准的中间件

ACE ORB (TAO)

TAO 是 CORBA 规范 [SLM98]的“高性能”、“实时”实现。它包括图 B.1(1)所示的网络接口、OS、网络协议和 CORBA 中间件功能。TAO 中使用的很多模式在 [SC00]中有所介绍。和 ACE 一样，TAO 属于开放源码软件，可以在 http://ace.ece.uic.edu/TAO 下载到它。

在标准化的进程中，DOC Group 在 TAO 工作方面扮演着重要的角色：

- 它影响了 OMG 的实时 CORBA 规范 [Obj01]，特别是它的“显式绑定（explicit binding）”和“可移植同步器（portable synchronizer）”特性。
- 它作为两个 ORB 之一，被选作 DISA 的 Joint Tactical Architecture (JTA)，从而被吸纳到了 DII COE 中。
- 它被用作 DMSO HLA/RTI 分布式交互模拟标准 [OSLN99]的基础。

今天，TAO 被应用在世界上数百个研究和商业项目中，涉及航空、电信、仿真、医疗保健、科学计算、金融服务等领域。它有望成为第一个成功地“飞翔在战斗机中”[Lac98]的 CORBA ORB。TAO 由 OCI（http://www.theaceorb.com）提供支持，它使用的开放源码许可协议和商业模式同 Riverace 的类似。

JAWS Adaptive Web Server (JAWS)

JAWS 是用 ACE 框架（framework）构造而成的高性能、自适应式 Web 服务器[HS99]。它实现了 HTTP 1.0 规范。图 B.1(2)展示了 JAWS 中主要的结构化组件和模式。JAWS 被构造为“框架的框架”。整个 JAWS 框架包含以下组件和框架：

- **并发策略（concurrency strategy）**，实现了“一个请求一个线程”、“线程池”这样的并发机制；这些机制可以在运行时自适应（adaptively）选择，或在初始化期间预先选择；
- **I/O 策略（I/O strategy）**，实现了各种 I/O 机制，如异步、同步和反应式 I/O；
- **事件分发器（event dispatcher）**，协调 JAWS 的 I/O 和并发策略之间的事件处理；
- **协议处理器（protocol handler）**，实现“HTTP 请求”的解析和处理；
- **协议管道（protocol pipeline）**，可以过滤（filter）操作，使之方便地同“协议处理器”正在处理的数据结合（incorporated）；
- **缓冲虚拟文件系统（cached virtual filesystem）**，通过缓冲策略（如 *“最近最少使用（LRU, least recently used）”*、*“最少使用（LFU, least frequently used）”*）降低文件系统的访问开销，从而提高 Web 服务器的性能。

这些框架（framework）中的每一个都被构造为一组“相互协作”的对象，这些对象是通过结合和扩充 ACE 中的组件实现的。构成（constituting）JAWS 的模式在[SSRB00]中作了论述。和 ACE、TAO 一样，JAWS 被应用在很多商业项目中。此外，它也是开放源码软件——发布在 ACE 版本的$ACE_ROOT/apps/目录中。

ACE 是 TAO 和 JAWS 成功的基础。技术的发展是相互促进的。例如，在实现和重构 [FBB+99]TAO/JAWS 的进程中，一方面，得到了很多新类，另一方面，ACE 中很多类和框架也得到了提炼和优化。

B.1.5 开放源码的作用

ACE 开放源码社区有数以千计的开发者和最终用户。没有他们的支持，ACE 不可能获得今天的成功，原因是：

缺乏足够的资金投入。开发 ACE+TAO 消耗的工作量超过了 100 个“人年（person-years）”。假设在一个传统的公司中，一个“人年”在薪金和福利上花费 20 万美元；那么，如果使用传统的“封闭源码”模式开发 ACE+TAO，将需要近 2000 万美元的资金。从第 261 页可以知道，开发 ACE+TAO 时，DOC Group 花费赞助商的实际资金大约只有这个数目的三分之一。这些差额大多是由 ACE+TAO 开发源码社区在时间和精力上的无偿付出换来的。

缺乏短期投资回报。ACE 经历了数年的发展，才可以应用于产品系统。例如，在很多 ACE 框架中，需要多次反复工作，才能确定合适的 API、模式和关系。很少有公司或风险投资商愿意花这么长的时间来投资中间件项目——特别是，如果最终结果会免费付送！

缺乏足够的时间、兴趣和可用的平台。若要开发“具有高度可移植性”的中间件，就需要在相对“平凡（mundane）”的工作（例如，确保软件能够顺畅地在大量编译器和 OS 平台上构建和运行）上投入大量的精力。即使某个核心开发小组不乏足够的时间和兴趣（在长期资金不足的高级研发环境下，这是不可能的），但为了创建和维护移植环境，要获得所有必需的硬件/软件平台，代价将极为昂贵。幸运的是，采用开放源码模式，DOC 工作组就可以利用 Web 上庞大的人力和计算资源，从而能够以省时、经济的方式完成大量“平凡”的工作[SP01]。

缺乏广泛的技能。ACE 极大地受益于全世界中间件专家的贡献和指导。Doug 曾经在 20 世纪 80 年代后期参与 GNU C 和 C++项目，并和多位自由软件先驱共事，其中包括 Richard Stallman、Doug Lea、Michael Tiemann。因而，在着手开发 ACE 时，Doug 清楚地知道，创立技术专家组对于扩展和支持一个软件工具包的功能极为重要。

1 2 3 4 5 6 7 8 甲乙丙甲乙

1991 年，ACE 的第一个工作版本一经完成，就立即以匿名 ftp 的形式公开发布了出来——我们知道，那时还没有 HTTP 和 Web。很快，一个小的用户群形成了；他们帮助修改错误，并将 ACE 移植到新的平台上，从而增加了“自由软件”的价值——那时，“开放源码”这个词还没有出现！

B.2 未来之路

将来，我们会在 ACE 上集中开展以下工作：

加强文档。ACE 缺乏全面、易于使用的文档；这在某种程度上妨碍了它获得更大的成功。为了弥补这一缺陷，一项大型、包括多个方面的工作正在展开，并且会在将来成为工作的中心。2000 年，Carlos O'Ryan 和 Darrell Brunsch 首先在 UCI 开始行动，将 ACE 参考资料的制作方式转化为 Doxygen [Dim01]。Doxygen 更好地表达了 ACE 的用法，生成的参考资料也更容易使用。经过完善的参考资料和即将出版的《The ACE Programmer's Guide》[HJS]和《C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks》[SH]将有助于更多的 C++开发者最大程度地利用 ACE 的强大功能和灵活性。

支持新平台和现有平台的新版本。我们将继续增强和扩充 ACE，以支持现有操作系统的新版本。我们还会拓展新的平台，这主要看 ACE 用户群和 Riverace 客户的兴趣和需求。在 ACE 中，C++特性的运用也会扩展。ACE 在大多数 C++编译器上都可以工作，包括一些来自 Cfront 年代的编译器。随着时间的推移和更多的编译器支持 C++标准，ACE 的“编译器支持之窗（window of compiler support）”将会移动（slide），而不是扩大(enlarged)。这种发展使得 ACE 可以利用 C++标准中的新特性，如原始的(native)异常、模板部分特殊化、模板成员函数和 C++标准库等。

提高健壮性。ACE 需要进行广泛的回归测试（regression test），这种测试每天不断地在数十个平台上进行。这些测试会自动检测编译错误、配置错误、运行时错误和内存错误。任何时候，都可以跟踪 ACE 的状态，在 http://ace.ece.uci.edu/scoreboard/上得到这些回归测试的输出结果。因为 ACE 是一套大型软件，我们将继续扩大它的回归测试能力，以保证它的品质和健壮性能继续得到提高。

减少内存占用。在普通 OS 平台上，ACE 核心库的体积从 700KB 到 1.5MB 不等；如果编译中包含了全部调试信息，体积还会大得多。在某些环境下，特别是在内存受限的嵌入式系统 [NW0]中，ACE 的内存占用（memory footprint）会成为障碍。因此，我们已经在开始一项工作，努力使 ACE 的子集可以分开生成（built）；这样，应用程序就只用承担它需要的那部分子集的内存开销，不用承担它不需要的那部分开销。关于 ACE 子集的划分情况，请参看 ACE 软件包中的$ACE_ROOT/docs/ACE-subsets.html。

请让我们知道，你是否有兴趣参与这些项目！

B.3 结束语

过去 10 年里，我们的工作涉及网络应用程序研发的许多方面——从图纸上的学术研究到亲手参与产品开发和咨询。根据我们的经验，要想将高级软件研发技术的巨大威力带给主流商品软件的开发者，最有效的方式是：

1. 基于最前沿的系统研究，开发出高品质的可复用软件；
2. 和软件的早期用户合作，将其运用到现实世界的应用程序中；
3. 向主流软件开发者提供商业支持和服务，帮助他们转化和吸收新技术。

除非技术上的优势已经展现在现实世界的运行系统中，否则，很少会有研究者能够将他们的技术有效地注入商品市场。但是，技术和研究的完全脱离会制约前沿技术的研发工作，从而妨碍新技术的进步和扩充。研究团队和商业团体的紧密合作，有助于维护技术的进步和新技术向主流市场的顺利转化。相互联系的商业和学术团体（如 Riverace 和 DOC Group）惟有通过耐心和合作，才能在这两方面都做到最好。对团队中的所有成员，以及受益于这些合作的用户群来说，其回报是丰厚的。

术语表

Abstract Class – 抽象类

一种类，这种类没有实现接口中定义的所有方法。一个抽象类为它的子类（subclasses）定义了一个公共接口。

Acceptor-Connector Pattern – 接受器-连接器模式

一种设计模式，用于将网络系统中相互合作的对等服务之间的“连接和初始化”同“连接和初始化之后，这些服务所执行的处理”分离开来。

Accidental complexity – 偶发复杂性

在应用领域中，用来开发软件的工具和技术（如“非类型安全”的 API、过程设计）存在局限性，作为“这些局限性的副作用”出现的复杂性被称为偶发复杂性。

Active Connection Establishment – 主动连接建立

对等应用程序（peer application）扮演的一种连接角色，用于发起一个“和远程对等端的连接”（和 *Passive Connection Establishment* 对应）。

Active Object – 主动对象

一种实现了 Active Object 模式的对象（和 *Passive Object* 对应）。

Active Object Pattern – 主动对象模式

一种设计模式。它将“方法的执行”同“方法的调用”分离开来，以提高并发性，并简化对“自己的控制线程中的对象”的同步访问。

Algorithmic Decomposition – 算法分解

一种设计范式（paradigm）。它根据功能将程序进行分解。

API – 应用程序编程接口

即 Application Programming Interface。软件平台（如操作系统）的外部接口；构筑在这些软件平台之上的系统或应用程序将使用这一接口。

Aspects – 方面，特征

程序的一种属性（property），如内存管理、同步、容错。这种属性不受模块限制（module boundary）。

Asynchronous I/O – 异步 I/O

一种“发送”或“接收”数据的机制。I/O 操作启动后，调用者不必等待操作结束，因而不会造成阻塞。

Bandwidth – 带宽

通信介质（如网络或总线）的容量（capacity）。

Barrier Synchronization – 栅栏同步

一种线程同步机制。它用来指定一组线程，使得这组线程中的每一个线程到达某一状态（如结束了某一集体操作或任务）时，可以继续同步执行。一个栅栏代表执行路径中的一个特定点。每一个线程到达这个栅栏时都将等待，直至其他线程也到达这一点。当这组线程全部到达这个栅栏时，栅栏将“落下”，使所有线程一起继续执行。

Broadcast – 广播

Broadcast（广播）是 *multicast（多播）*的一个特殊形式：消息从一个发送方传输给某个域（domain）中的所有接收方。

Bus – 总线

一种将计算设备（如 CPU、磁盘和网络接口）连接在一起的高速通信信道。

Busy Wait – 忙等待

线程使用的一种技术，用于等待一个锁（lock）。它执行一个密集循环并不断轮询，检查这个锁在每次循环中是否可用。较之通过“睡眠（sleeping）”、“允许其他线程执行”的方式等待锁的释放，“忙等待”与之大相径庭。

Callback – 回调

注册在分发器（dispatcher）中的对象。当特定事件发生时，分发器会回过头来针对这个对象调用一个方法。

Client – 客户

在本书中，客户表示一个会调用或使用“其他组件提供的功能”的角色、组件或子系统。

Collocation – 并置，配置

将一个对象置于和“访问这个对象”的客户相同的进程或主机中时，相关的行为被称为并置。“并置”常用来提高“引用局部性（locality of reference）”（和 *Distribution* 比较）。

Common Data Representation (CDR) – 公共数据表示

CORBA 定义的一种标准格式，用于数据的整编/解编。它使用了半规范（bicanonical）的“接收方保证正确性”的表示法；只有当发送方的字节顺序和接收方的字节顺序不同时，它才会导致开销。

Common Middleware Services – 公共中间件服务

这一层中间件定义的是“独立于领域（domain-independent）”的服务，如事件通知、日志、多媒体流、持久化、安全、全局时间同步、实时调度和分布式资源管理、容错、并发控制、可恢复式交易等，用来分配、调度、协调分布式系统中的各种资源。

Component – 组件

软件系统中用来实现某一（组）特定服务的一个被封装的部分。一个组件有一个或多个接口，用来对它的服务提供访问。组件可以用作系统结构中的构件。在编程语言级，组件可以体现为模块、类、对象或一组相关的函数。没有实现接口中所有元素的组件被称为抽象组件（abstract component）。

Component Configurator Pattern – 组件配置器模式

一种设计模式，允许应用程序在运行时（run time）连接（link）和反连接（unlink）其组件的实现，不必修改、重新编译或重新静态连接应用程序。

Concrete Class – 具体类

可实例化（instantiate）对象的类。和抽象类相反，具体类中的所有方法都提供了实现。这个术语用于将具体的子类（subclass）同它们的抽象超类（superclass）区分开来。

Concurrency – 并发

对象、组件或系统具有的一种能力，用来执行“逻辑上具有同时性”的操作（和 *Parallelism* 对应）。

Condition Variable – 条件变量

一种同步机制。利用这一机制，一组相互协作的线程可以将自己暂时挂起（suspend），直至“和这些线程之间的共享数据有关”的条件表达式达到预定状态为止。条件变量总是和互斥体一起使用；线程在计算条件表达式之前必须获得互斥体。如果条件表达式为 false，则线程原子地（atomically）将自己挂起在条件变量上，并释放互斥体，使得其他线程可以修改共享数据。当一个合作线程（cooperating thread）修改这个数据时，它会通知条件变量，从而原子地恢复一个已经挂起在这个条件变量上的线程，并再次获得它的互斥体。

Connection – 连接

一种“完全的关联（full association）”，对等端（peers）利用它来在网络应用程序端点之间交换数据。

Contention Scope – 竞争范围，竞争空间

一个区域。线程在这个区域中竞争资源（如 CPU 时间）。参见 *Process Scope* 和 *System Scope*。

Cooperative Cancelation – 协作式取消、协作式删除

ACE 中实现的一种线程删除机制；利用这一机制，删除方线程可以请求一个或多个线程删除自己。那些线程有可能在方便的时候接受这个请求，但并不一定会那样做。

CORBA – 公共对象请求代理体系

即 Common Object Request Broker Architecture（公共对象请求代理体系），一种有关分布式对象计算的中间件标准，由 Object Management Group（OMG）制定。

Critical Section – 关键段，临界区

在一个对象或子系统中，不该并发执行的代码可以通过关键段做到同步。关键段是一组指令序列，它具有这样的不变性（invariant）：当一个线程或进程在一个关键段中执行时，其他线程或进程就都不能在这个关键段中执行。

Daemon – 监控器，后台服务程序

一种在后台连续运行的服务器进程，为客户提供各种服务。

Data and Instruction Caches – 数据及指令高速缓存

专门为 CPU 配置的高速内存，可以提高整个系统的性能。

Data-Mode Socket – 数据模式 socket

见 Scoket。

Datagram – 数据报

一条携带了足够信息、完整、独立的消息，无需在源主机和目标主机或网络之间作前期信息交换，就可以从源主机发送到目标主机。

Deadlock – 死锁

死锁是并发（concurrency）中存在的一种危险。当多个线程试图获得多个锁，并且无限期阻塞在一个循环等待状态上时，死锁便会发生。

Deferred Cancelation – 延迟取消，延迟删除

一种线程删除机制。线程删除被延迟，直至“将被删除的线程”要求执行任何尚未完成的删除。

Demarshalling – 解编，解除整编

将“被*整编（marshaled）*”的消息从“和主机无关”的格式转换为“和主机相关”的格式。

Demultiplexing – 多路分离

将从一个输入端口传输进来的数据发送到多个目标接收方的机制。输入端口和接收方之间具有 1:N 的关系。多路分离常被应用到传入的（incoming）事件和数据流中。其反操作被称为 multiplexing（多路复用）。

Design Pattern – 设计模式

设计模式为提炼“软件系统中的组件”或“这些组件之间的关系”提供了方案。它描述一种常见的通信组件结构，以解决某一特定环境下普遍存在的设计问题。

Distribution – 分布

将一个对象置于一个“和访问它的客户不同”的进程或主机上的相关行为。分布常用来提高容错性，或用来访问远程资源（与 *Collocation* 比较）。

Distribution Middleware – 分布式中间件

这个层次的中间件可以使普通的网络编程任务（例如，连接和内存管理、整编和解编、端点和请求的多路分离、同步、多线程）自动化；这样一来，开发者几乎可以像编写*独立应用程序（stand-alone application）*那样编写分布式应用程序——即，在目标对象上调用操作时，不必关心它们的位置、语言、OS 或硬件。

Domain – 域，领域

和某一问题领域（problem area）相关的概念、知识和其他信息。常被用在“应用领域”，表示一个应用程序所涉及的问题领域。在 Internet 上，域是一个逻辑地址实体，如 uci.edu 或 riverace.com。

Domain Analysis – 领域分析

一个归纳、反馈驱动的过程，用来系统地分析一个应用领域，以确定这个领域的核心问题和设计空间，从而为它们找到有效的方案。

Domain-Crossing Penalty – 跨领域代价

系统函数调用从用户空间（user-space）跨越至内核空间（kernel-space）时产生的性能开销。除了切换至“高特权访问模式”时涉及的中断之外，I/O 调用一般还得向（从）用户空间的内存复制数据，因而增加了延迟。

Domain-specific Middleware services – 特定领域中间件服务

这个层次的中间件定义的服务针对的是特定领域（如电信或电子商务）的需求。“特定领域”中间件服务针对的是特定的纵向市场，而不是“底层”中间件提供的具有更

广泛可复用性的服务。

Double-Checked Locking Optimization Pattern – 双检测锁定优化模式

一种设计模式。运用这一模式，在程序执行期间，如果代码的关键段必须以“线程安全”的方式获得锁，竞争和同步的开销能够得以降低。

Dynamic Binding – 动态绑定

一种将操作名称（消息）同对应代码（方法）之间的关联（association）延迟到运行时的机制。用于实现“面向对象”语言中的多态性（polymorphism）。

Endpoint – 端点

连接的端头。

Escape Hatch – 非常手段，应急手段

一种预防措施，使得程序能够以“设计者未预见到”的方式使用系统功能。

Event – 事件

一种消息，传达某一重要行为的发生以及和这一行为相关的任何数据。

Event Loop – 事件循环

一种程序结构，用来连续等待和处理事件。

Exception Safe – 异常安全

所谓一个组件是“异常安全”的，是指这个组件中发生的异常或从“这个组件调用的组件”中传出的异常不会造成资源泄漏，也不会造成不稳定状态。

Factory – 工厂

一个方法或函数。在实例化或初始化一个对象或组件实例时，工厂可用来创建、组装所需的资源。

Flow Control – 流量控制

一种网络协议机制，用来防止“快速发送方”耗尽“慢速接收方”的缓冲区和计算资源。

Framework – 框架

见 *Object-Oriented Framework*。

Gather-Write – 集中写入

一种输出操作，将多个非连续数据缓冲区中的内容在一个单独的操作中传输。

Generative Programming – 生成式程序设计

一种专注于设计和实现“软件组件”的编程技术；这些组件可以结合起来，生成专门的、高度优化的系统，以满足特定的需求。

Generic Programming – 泛型程序设计

一种结合使用设计模式和C++参数化类型的编程技术，使得开发者可以编写出具有表达性、灵活、高效、高度可复用性的代码。

Half-Sync/Half-Async Pattern – 半同步/半异步模式

一种结构化模式，用于在并发系统中分离异步和同步处理，从而简化编程，同时不会过度降低性能。这个模式引入了两个通信层，一个用于异步服务处理，另一个用于同步服务处理。一个排队层介于异步层和同步层之间，担负中间调节作用。

Handle – 句柄

“句柄”标识操作系统内核所管理的资源。通常，这些资源包括网络连接、打开的文件、计时器、同步对象，等等。

Host – 主机

连接在网络上的、可寻址的计算机。

Host Infrastructure Middleware – 主机基础设施中间件

这个层次的中间件封装了主机提供的并发及IPC机制，用以创建OO网络编程能力，从而消除“用原始OS API（如Socket或Pthreads）开发网络应用程序时”存在的大量繁琐、容易出错、不可移植的问题。

Hot Spot – 热区

执行频繁（通常很短）的代码段。

Idiom – 惯用手法，常用技术

所谓idiom，指的是“和编程语言相关”的低级（low-level）模式。idiom描述如何使用特定语言的特性来实现组件的特定方面，或组件之间的关系。

Inherent Complexity – 内在复杂性，固有复杂性

一个领域（domain）的基本性质所导致的复杂性。它会使程序的开发复杂化（见Accidental Complexity）。

Inheritance – 继承

面向对象语言的一个特性，用于从已有的类派生出新类。继承定义了实现（implementation）的复用、子类型的关系，或以上二者。不同的编程语言有可能提供单继承或多重继承。

Inlining – 内联

一种编译时优化技术，可以将“对函数或方法的调用”替换为“那个函数或方法的实际代码体”。如果内联的函数/方法的代码很多，将会导致代码“膨胀”，并会给存储器的使用和分页（paging）带来负面影响。

Interface – 接口

类、组件或子系统中可公共访问的部分。

Internet – 互联网

基于网际协议（IP）的、世界范围的、“由网络组成”的网络。被普遍认为是自人类创造“火”和“MTV”以来最重要的发明。

Internet Protocol (IP) – 网际协议

一种网络层协议，对数据包进行“最大可能（best-effort）”的分段、重新组装和发送。

Interprocess Communication (IPC) – 进程间通信

位于独立地址空间的进程之间的通信。IPC 机制的例子有：共享内存、UNIX 管道、消息队列、socket 通信。

Intranet – 内部网

公司或其他组织内部的计算机网络。这种网络对来自外部的访问具有防护措施，它利用 Internet 技术进行通信，为公司范围的信息交换、协同工作和工作流程提供了平台。

Jitter – 抖动

所谓 jitter，是指一系列操作中所造成的延迟的标准偏差。

Latency – 延迟

操作中造成的时间滞后。

Layer – 层，级

一种抽象级别，用来定义一个体系结构中的一组特殊服务。第 n 层是第 n–1 层服务的消费者，又是第 n+1 层服务的提供者。

Leader/Followers Pattern – 领导者/跟随者模式

一种提供高效“并发”模型的结构化（architectural）模式。多个线程轮流享有一组事件源，以对事件源上发生的服务请求执行检查、多路分离、分发和处理。

Least Frequently Used (LFU) – 最少使用

一种缓冲策略，可使最不经常使用的元素被最先选出。

Least Recently Used (LRU) – 最近最少使用

一种缓冲策略，可使最早使用的元素被最先选出。

Linearization – 线性化

将“丰富类型（richly typed）”数据（如数组、表或图）*整编（marshaling）*为线性内存缓冲区的过程。

Lock – 锁

一种机制，用来实现某种类型的*关键段（critical section）*。可将“可连续获得和释放”的锁（如一个静态的互斥体）添加到一个类中。如果同时有多个线程试图获得一个锁，则只有一个线程会成功，其他线程会阻塞，直至那个锁可用。其他锁定机制（如信号量或 readers/writer 锁）定义了不同的同步语义。

Lock-Step – 锁步

一种协议设计范式（paradigm），在后续请求发出之前，需要先回答一个请求。

Marshaling – 整编

将一组数据从“和主机相关”的格式转换为“和主机无关”的格式。

Memory Management Unit (MMU) – 内存管理单元

MMU 用来保护各个进程地址，防止它们遭受系统中其他活动进程的无意或恶意破坏。

Message – 消息

“消息”用于对象、线程或进程之间的通信。在面向对象系统中，消息这个术语用来描述“一个操作或一个对象中的方法”的选择（selection）和激活（activation）。这类消息是同步的，即发送方等待接收方执行完“被激活的操作”。线程和进程通常采用异步通信，即发送方继续执行，不必等待接收方的应答。

Message Passing – 消息传递

一种 IPC 机制，用来在线程或进程之间交换消息（和 *Shared Memory* 对应）。

Middleware – 中间件

一组抽象层和组件，提供可复用的公共服务和网络编程机制。中间件位于操作系统及其协议栈之上，但位于任何特定的应用程序结构和功能之下。

Monitor Object Pattern – 监控器对象模式

一种设计模式。它可以对并发方法的执行实施同步，确保一个对象内部同时只有一个方法在执行。它还允许一个对象中的方法可以协同调度它们的执行顺序。

Multicast – 多播

一种网络协议，允许发送方向多个接收方高效地传输消息（和 *Unicast* 对应）。

Mutex – 互斥体

互斥体是一种“互斥（mutual exclusion）”锁定机制，它保证一个关键段中同时只有一个线程处于并发活动状态，从而防止“竞态条件”。

Networked Application Architecture – 网络应用体系

一种体系结构，有助于软件组件和资源的使用；这些组件和资源可以放在分布式系

统的所有地方，用来执行网络应用程序。

Nonrecursive Mutex – 非递归互斥体

一种互斥体。任何线程要想重新获得它，首先得将它释放。与 *Recursive Mutex* 对应。

Object-Oriented Framework – 面向对象框架

一组集成在一起的类，共同为一系列相关应用提供可复用的软件架构。在面向对象环境中，一个框架由抽象类和具体类组成。这种框架的实例化方式包括：组合（composing）和继承现有的类。

Object Request Broker – 对象请求代理

一种中间件层。允许客户在分布式对象上调用方法，而不必关心对象的位置、编程语言、操作系统平台、网络协议或硬件。

One-Way Method Invocation – 单向方法调用

一种方法调用。方法向服务器对象传递参数，但不从服务器接收任何结果（和 *Two-Way Method Invocation* 对应）。

Out-of-Band Data – 带外数据

正常字节流传输之外发送的数据，也称紧急数据。

Parallelism – 并行

对象、组件或系统具有的一种能力，用以执行“物理上具有同时性”的操作（和 *Concurrency* 对应）。

Parameterized Type – 参数化类型

编程语言的一种特性，允许类被其他各种类型“参数化”（与 *Template* 对应）。

Passive Connection Establishment – 被动连接建立

对等应用程序扮演的一种连接角色，用于接受一个来自远程对等方的连接（与 *Active Connection Establishment* 对应）。

Passive-Mode Socket – 被动模式 Socket

见 Socket。

Passive Object – 被动对象

一种对象。这种对象借助调用者线程来执行它的方法（与 *Active Object* 对应）。

Pattern – 模式

“模式”描述特定设计环境下经常出现的某一特定的设计问题，并为这个问题提供可靠的方案。在说明方案时要描述它的组成部分，这些组成部分的职责和关系以及它

们的协作方式。

Pattern Language – 模式语言

一系列相互关联的模式，用来定义一个过程，以系统地解决软件开发中的问题。

Peer-to-Peer – 端对端

在分布式系统中，对等端（peers）指的是相互通信的进程。和 client-server（客户-服务器）体系中的组件不同，对等端既可以扮演客户或服务器的角色，也可以身兼二职，还能动态地改变这些角色。

Pipes and Filters Pattern – 管道与过滤器模式

一种结构化模式，为处理数据流的系统提供一个结构。

Platform – 平台

一个系统在它的实现（implementation）中用到的硬件和（或）软件的总称。软件平台包括操作系统、库和框架。一个平台实现了一个虚拟机，应用程序就运行在这个虚拟机之上。

Port Number – 端口号

一个 16 位数，用于在 TCP 协议中标识一个通信端点。

Priority Inversion – 优先级倒置

调度上的一种危险性，发生在“低优先级”线程或请求阻塞了“高优先级”线程或请求的时候。

Proactor Pattern – 前摄器模式

一种结构化模式，允许“事件驱动型”应用程序高效地多路分离和发送“异步操作结束时触发的服务请求”，以获得并发上的性能优势，但毋须承担相应的职责。

Process – 进程

进程提供了资源（如虚拟内存）和保护能力（如“用户/组”标识符和“硬件保护”地址空间）；进程中的一个或多个线程可以使用这些资源和能力。较之线程，进程维护更多的状态信息，创建、同步和调度中需要更多的开销，而且常常是通过“消息传递”或“共享内存”机制同其他进程通信。

Process-Scope Contention – 进程范围竞争，进程空间竞争

一种并发策略。线程处理或同步竞争发生在一台主机的一个进程中（与 *System-Scope Contention* 对应）。

Protocol – 协议

一组规则，描述消息如何在通信端之间交换，以及这些消息的语法和语义。

Protocol Stack – 协议栈

一组具有结构化层次的协议。

Quality of Service (QoS) – 服务质量

一组策略和机制，用来控制和增强通信的特性，如带宽、延迟、波动。

Race Condition – 竞态条件

竞态条件是一种并发上的危险，它发生在多个线程同时在一个关键段中执行，但这个关键段没有被正确串行化（serialized）的时候。

Reactor Pattern – 反应器模式

一种结构化模式，允许“事件驱动型”应用程序多路分离和发送服务请求；这些服务请求由一个或多个客户发送给一个应用程序。

Readers/Writer Lock –“多读取者/单写入者”锁

一种锁，允许多个线程并发访问某一资源，但同一时间只允许一个线程修改这个资源，进而防止并发访问和修改。

Recursive Mutex – 递归互斥体

一种锁，拥有这个互斥体的线程可以再次得到它，但不会在这个线程上造成“自死锁”。与 *Nonrecursive Mutex* 对应。

Refactoring – 重构

一种渐近式行为。从现有软件中提取通用行为，以改善组件和框架（framework）的结构（structure）和可复用性（reusability）。

Reify – 具体化

一种行为，为抽象体创建具体的实例。例如，一个具体的反应器实现（implementation）“具体化”了 Reactor 模式，一个对象“具体化”了一个类。

Ripple Effect – 连锁反应，晕轮效应

前一次变化的副作用必然引发必要变化的现象，这里借用一颗石子投入平静的湖水，产生波纹的情景。

Scatter-Read – 分散读取

一种输入操作。将数据存储至调用者提供的多个缓冲区，而非单个连续的缓冲区。

Scheduler – 调度器，调度程序

一种机制，用来确定线程或事件执行的次序。

Scoped Locking Idiom – “范围锁定”技术，“作用域锁定”技术

C++常用技术。当控制（control）进入一个范围时，一定能得到锁；当控制离开这

个范围时，一定能自动释放锁——无论从这个范围的哪条路径返回。

Semaphore – 信号量

一种锁定机制。它需要维护一个计数值，只要计数值大于零，某个线程就可以得到这个信号量而不会阻塞。但如果计数值变为零，则线程将在这个信号量上阻塞，直至另一个线程释放这个信号量，递增计数并使得计数值大于零。

Serialization – 串行化，序列化

一种机制。保证一个关键段中同时只有一个线程在执行，从而防止竞态条件。

Service – 服务，服务程序

在网络编程环境下，“服务”指的是：（1）服务器提供的某项定义良好的功能，如 Inetd super-server 提供的 ECHO 服务；（2）服务器监控程序提供的一组功能，如 INETD super-server 本身；（3）一组用来协作完成一项共同任务的服务器进程，如一个局域网（LAN）子网中的一组 RWHO 监控程序——它们周期性地广播和接收状态信息，将用户行为报告给其他主机。

Shared Library – 共享库

一种程序库，可以被多个进程共享，可以被动态链接至某一进程地址空间之内或之外，以提高应用程序在运行时的灵活性和可扩充性（也称 *Dynamically Linked Library*，或 DLL）。

Shared Memory – 共享内存

操作系统的一种机制，允许一台计算机上的多个进程共享某一公共内存段（与 *Message Passing* 对应）。

Sleep-Lock – 休止锁，睡眠锁

一种“同步”实现。通过“睡眠（sleep）”和“允许其他线程运行”的方式等待某个锁释放（比较 *Spin-Lock*）。

Smart Pointer – 智能指针

智能指针是一种 C++对象，其外观和行为和内建指针类似，但能够实现内建指针无法提供的很多功能，如缓存、持久性或线程专有存储访问等。

Socket – 套接字

和网络编程有关的一组术语。一个 socket 是一个“标识了某一网络地址和端口号”的通信端点。Socket API 是大多数操作系统支持的一组函数调用，利用 Socket API，网络应用程序可以建立连接，并通过 socket 端点进行通信。“数据模式”socket 用来在已经连接好的对等端之间交换数据。“被动模式”socket 是一个工厂，它返回一个句柄，

指向一个已经连接好的“数据模式”socket。

Spin-Lock – 自旋锁

一种“同步”实现。在等待一个锁时，它会执行密集循环，在每一次循环中查询这个锁是否可用（比较 *Sleep-Lock*）。

Stand-Alone Application Architecture – 独立应用体系

一种体系结构。它要求应用程序的所有组成部分和资源都位于一台计算机上，这样在执行应用程序时，就不必依赖于另一台计算机。

Starvation – 饥饿

调度上的一种危害。一个或多个线程长期被高优先级线程抢占，因而永远不会被执行。

Strategized Locking Pattern – “策略化锁定”模式

一种设计模式，将那些“防止组件的关键段被并发访问”的同步机制参数化。

Synchronization Mechanism – 同步机制

一种锁定机制，用于协调线程的执行顺序。

Synchronous I/O – 同步 I/O

一种发送或接收数据的机制。I/O 操作启动后，调用者被阻塞，等待操作结束。

System – 系统

一组软件和（或）硬件的集合，用来执行一个或数个服务。一个系统可以是一个平台或一个应用程序，也可以同时包括二者。

System-Scope contention – 系统范围竞争，系统空间竞争

一种并发策略。“线程处理”或“同步竞争”发生在一台主机上的各个进程之间（与 *Process-Scope Contention* 对应）。

Template – 模板

C++语言的一个特性。通过各种类型、常量或函数指针，类和函数可以被参数化。模板常被称为“泛型”或“参数化类型”。

Thread – 线程

一组独立的指令序列，执行在一个“可被其他线程共享”的地址空间中。每个线程有它自己的运行时栈（stack）和寄存器，使得它可以执行同步 I/O，不至阻塞其他正在并发执行的线程。和进程相比，线程维护极少的状态信息，创建、同步和调度过程中需要的开销也相对较少；此外，线程通常是通过“所在进程的内存空间”中的对象和其他线程通信的，而不是通过共享内存。

Thread-Per-Connection – 一个连接一个线程

一种并发模型，它将每个网络连接同一个独立的线程关联在一起。在连接期间，这个模型会在“独立”的线程中处理每一个“连接至服务器”的客户。对那些“必须和多个客户长期会话”的服务器来说，这很有用。对那些“将每一个请求同一个线程关联在一起”的客户（如 HTTP 1.0 Web 浏览器）来说，这没有什么用处。后者实际上是一个 *thread-per-request* 模型。

Thread-Per-Request – 一个请求一个线程

一种并发模型，它为每一个请求创建一个新线程。如果服务器必须处理“来自多个客户”且“持续时间长”的请求事件（如数据库查询），这个模型将很有用。对于持续时间短的请求，它的作用会打折扣，因为为每一个请求创建新线程都需要开销。如果有很多客户同时发送请求，则还会消耗大量的操作系统资源。

Thread Pool – 线程池

一种并发模型，用来分配一组“能够同时处理请求”的线程。这个模型是 *thread-per-request* 的一个变型，它预先创建一个线程池，以分期承担“线程创建”的成本。如果服务器希望限制对操作系统资源的消耗，这个模型将很有用。客户请求可以并发执行，直至同一时刻的请求数超过池中的线程数——此时，新增的请求必须排队，直至某个线程变为可用。

Thread-Safe – 线程安全

多个线程并发执行同一段代码时，不会造成任何不利的副作用（竞态条件、数据冲突等）。

Thread-Safe Interface Pattern – “线程安全”接口模式

一种设计模式。这种模式能降低锁定开销，还能保证组件之间的方法调用不会产生“自死锁”——方法是：重新得到一个已被组件占有的锁。

Thread-Specific Storage (TSS) Pattern – “线程专有存储”模式

一种设计模式。允许多个线程通过一个“逻辑上的”全局访问点来获得某个线程中的局部对象，但不会在每一个对象访问上引发锁定开销。

Traits – 特征类

一种传递信息的类型（type）；这些信息被另一个类或算法使用，以在编译时确定策略（policy）和实现细节。

Transmission Control Protocol (TCP) – 传输控制协议

一种“面向连接”的传输协议，用于在本地和远程端点之间可靠、有序、不重复地

交换数据的字节流。

Transport Layer – 传输层

协议栈中的一层，负责“点到点”数据传输和连接管理。

Transport Layer Interface (TLI) – 传输层接口

TLI 是 System V UNIX 提供的一组函数调用，网络应用程序用它来建立连接，并通过已连接的传输节点进行通信。

Type Safe – 类型安全

编程语言的类型系统（type system）提供的一种性质，确保只有合法的操作才能在类型实例（instances of types）上调用。

Unicast – 单播

一种网络协议，允许一个发送方向单个接收方传输消息（与 *Multicast* 对应）。

Unicode – 通用编码

一种标准，规定字符的表示方法，包括大多数书面语言中的字符，以及标点符号、数学符号和其他符号的表示方法。

Unreliable Transport Protocol – 不可靠传输协议

一种传输协议。这个协议不对“被传输数据”的处理作任何保证：数据可能永远不会到达，也可能不按次序到达，或者多次到达。

User Datagram Protocol (UDP) – 用户数据报协议

一种不可靠、无连接的传输协议，用于在本地和远程节点之间交换数据报消息。

Virtual Machine – 虚拟机

一个抽象层，可向高级应用程序或其他虚拟机提供一组服务。

Virtual Memory – 虚拟内存

一种操作系统机制，可使开发者编写的应用程序的地址空间大于计算机上的物理空间。

Weakly Typed – 弱类型

一种数据（datum），其声明的数据类型没有完全反映其用途和目的。

Wrapper Facade – 包装器外观

一个或多个类，它在一个类型安全的 OO 接口中封装了函数和数据。

Zombie – 僵死进程

一个“已经退出、但依然占用进程表中的一个位置”的进程。僵死进程会一直维持这种状况，直至父进程得到它的退出状态（exit status）。

参考文献

[ABLL92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activation: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, pages 53—79, February 1992.

[AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, Boston, 2000.

[Ale01] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, Boston, 2001.

[AOS+00] Alexander B. Arulanthu, Carlos O'Ryan, Douglas C. Schmidt, Michael Kircher, and Jeff Parsons. The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging. In *Proceedings of the Middleware 2000 Conference*. ACM/IFIP, April 2000.

[AOSK00] Alexander B. Arulanthu, Carlos O'Ryan, Douglas C. Schmidt, and Michael Kircher. Applying C++, Patterns, and components to Develop an IDL Compiler for CORBA AMI Callbacks. *C++ Report*, 12(3), March 2000.

[Aus98] Matt Austern. *Generic Programming and the STL: Using and Extending the C++ Standard*. Addison-Wesley, Reading, Massachusetts, 1998

[BA90] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice Hall International Series in Computer Science, 1990.

[BC94] Arindam Banerji and David L. Cohn. Shared Objects and vtbl Placement – Revisited. Journal of C Language and Translation, 6(1): 44-60, September 1994.

[Bec00] Kent Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Boston, 2000.

[Bja00] Bjarne Stroustrup. *The C++ Programming Language, 3rd Edition*. Addison-Wesley, Boston, 2000.

[Bla91] U. Black. *OSI: A Model for Computer Communications Standards*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture – A System of Patterns.* Wiley and Sons, New York, 1996.

[Boo94] Grady Booch. *Object Oriented Analysis and Design with Applications (2nd Edition).* Benjamin/Cummings, Redwood City, California, 1994.

[Box97] Donald Box. *Essential COM*. Addison-Wesley, Reading, Massachusetts, 1997.

[Bro87] Frederick P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEI Computer*, 20(4): 10-19, April 1987.

[But97] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, Reading, Massachusetts, 1997.

[CB01] G. Coulson and S. Baichoo. Implementing the CORBA GIOP in a High-Performance Object Request Broker Environment. *ACM Distributed Computing Journal*, 14(2), April 2001.

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, 2000.

[Chr97] Clayton Christensen. *The Innovator's Dilemma: When New Technologies Cause Great Firms to Fail.* Harvard Business School Press, Cambridge, Massachusetts, 1997.

[CS92] Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP Vol III: Client-Server Programming and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[DC90] Stephen E. Deering and David R. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems*, 8(2):85—110, May 1990.

[Dim01] Dimitri van Heesch. Doxygen. http://www.doxygen.org, 2001.

[EKB+92] J.R. Eykholt, S.R. Kleiman, S. Barton, R. Faulkner, A Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams. Beyond Multiprocessing... Multithreading the SunOS Kernel. In *Proceedings of the Summer USENIX Conference*, San Antonio, Texas, June 1992.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring – Improving the Design of Existing Code.* Addison-Wesley, Reading, Massachusetts, 1999.

[FJS99a] Mohamed Fayad, Ralph Johnson, and Douglas C. Schmidt, editors. *Implementing Application Frameworks: Object-Oriented Frameworks at Work.*

[FJS99b] Mohamed Fayad, Ralph Johnson, and Douglas C. Schmidt, editors. *Building Application Framework: Object-Oriented Foundations of Framework Design.*

[Fow97] Martin Fowler. *Analysis Patterns*. Addison-Wesley, Reading, Massachusetts, 1997.

[FS97] Mohamed E. Fayad and Douglas C. Schmidt. Object-Oriented Application Frameworks. *Communications of the ACM*, 40(10), October 1997.

[Gal95] Bill Gallmeister. *POSIX.4 Programming for the Real World.* O'Reilly, Sebastopol, California, 1995.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[Gro] DOC Group. ACE Success Stories. http://www.cs.wustl.edu/~schmidt/ACE-users.html.

[GS99] Aniruddha Gokhale and Douglas C. Schdimt. 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*, 17(9), September 1999.

[HJS] Stephen D. Huston, James C.E. Johnson, and Umar Syyid. *The ACE Programmer's Guide*. Addison-Wesley, Boston (forthcoming).

[HLS97] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt. The Design and Performance of a Real-time CORBA Event Service. In *Proceedings of OOPSLA '97*, pages 184-199, Atlanta, GA, October 1997. ACM.

[HP91] Norman C. Hutchinson and Larry L. Peterson. The *x*-kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1): 64—76, January 1991.

[HS99] James Hu and Douglas C. Schmidt. JAWS: A Framework for High Performance Web Servers. In Mohamed Fayad and Ralph Johnson, editors, *Domain-Specific Application Frameworks: Frameworks Experience by Industry*. Wiley & Sons, New York, 1999.

[HV99] Michi Henning and Steve Vinoski. *Advanced CORBA Programming With C++*. Addison-Wesley, Reading, Massachusetts, 1999.

[IEE96] IEEE. *Threads Extension for Portable Operating Systems (Draft 10)*, February 1996.

[JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *Unified Software Development Process*. Addison-Wesley Object Technology Series. Addison-Wesley, Reading, Massachusetts, 1999.

[JM98] Bruce Jacob and Trevor Mudge. Virtual Memory: Issues of Implementation. *IEEE Computer*, 31(6):33—43, June 1998.

[JO99] Anthony Jones and Jim Ohlund. *Network Programming for Microsoft Windows*. MS Press, 1999.

[Joh97] Ralph Johnson. Frameworks = Patterns + Components. *Communications of the ACM*, 40(10), October 1997.

[Jor91] David Jordan. Instantiation of C++ Objects in Shared Memory. *Journal of Object-Oriented Programming*, March-April 1991.

[Jos99] Nicolai Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley, Reading, Massachusetts, 1999.

[Kha92] Khanna, S., *et al.* Realtime Scheduling in SunOS 5.0. In *Proceedings of the USENIX Winter Conference*, pages 375—390. USENIX Association, 1992.

[Kic97] Gregor Kiczales. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.

[Koe88] Andrew Koenig. *C Traps and Pitfalls*. Addison-Wesley, 1988.

[Kof93] Thomas Kofler. Robust iterators for ET++. *Structured Programming*, 14(2):62—85, 1993.

[KSS96] Steve Kleiman, Devang Shah, and Bart Smaalders. *Programming with Threads*. Prentice Hall, Upper Saddle River, NJ, 1996.

[Lac98] Ralph Lachenmaier. Open Systems Architecture Puts Six Bombs on Target. http://www.cs.wustl.edu/~schmidt/TAO-boeing.html, December 1998.

[Lea88] Doug Lea. libg++, the GNU C++ Library. In *Proceedings of the 1st C++ Conference*, pages 243-256, Denver, CO, October 1988. USENIX.

[Lea99] Doug Lea. Concurrent Java: Design Principles and Patterns, *Second Edition*. Addison-Wesley, Reading, Massachusetts, 1999.

[Lew95] Bil Lewis. *Threads Primer: A Guide to Multithreaded Programming*. Prentice-Hall, Englewood Cliffs. NJ, 1995.

[LGS00] David L. Levine, Christopher D. Gill, and Douglas C. Schmidt. Object Lifetime Manager – A complementary Pattern for Controlling Object Creation and Destruction. *C++ Report*, 12(1), January 2000.

[Mar64] Herbert Marcuse. *One Dimensional Man: Studies in Ideology of Advanced Industrial Society*. Beacon Press, Boston, 1964.

[MBKQ96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System.* Addison Wesley, Reading, Massachusetts, 1996.

[Mey96] Scott Meyers. *More Effective C++.* Addison-Wesley, Reading, Massachusetts, 1996.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction, Second Edition.* Prentice Hall, Englewood Cliffs, NJ, 1997.

[Moo91] Geoffrey Moore. *Crossing the Chasm: Marketing and Selling High-Tech Products to Mainstream Customers.* HarperCollins, New York, 1991.

[MSKS00] Sumedh Mungee, Nagarajan Surendran, Yamuna Krishnamurthy, and Douglas C. Schmidt. The Design and Performance of a CORBA Audio/Video Streaming Service. In Mahbubur Syed, editor, *Design and Management of Multimedia Information Systems: Opportunities and Challenges.* Idea Group Publishing, Hershey, PA, 2000.

[NGSY00] Balachandran Natarajan, Aniruddha Gokhale, Douglas C. Schmidt, and Shalini Yajnik. Applying Patterns to Improve the Performance of Fault-Tolerant CORBA. In *Proceedings of the 7th International Conference on High Performance Computing (HiPC 2000),* Bangalore, India, December 2000. ACM/IEEE.

[NL91] Bill Nitzberg and Virginia Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer,* pages 52—60, August 1991.

[NW01] James Noble and Charles Weir. *Small Memory Software: Patterns for Systems with Limited Memory.* Addison-Wesley, Boston, 2001.

[Obj01] Object Management Group. *The Common Object Request Broker: Architecture and Specification,* 2.5 edition. September 2001.

[O'R98] Tim O'Reilly, The Open-Source Revolution. *Release 1.0,* November 1998. http://www.oreilly.com/catalog/opensources/.

[OSLN99] Carlos O'Ryan, Douglas C. Schmidt, David Levine, and Russell Noseworthy. Applying a Scalable CORBA Events Service to Large-scale Distributed Interactive Simulations. In *Proceedings of the 5th Workshop on Object-oriented Real-time Dependable Systems,* Montery, CA, November 1999. IEEE.

[Par72] David L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM,* 15(12), December 1972.

[POS95] Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application: Program Interface (API) [C Language], 1995.

[POS+00] Irfan Pyarali, Carlos O'Ryan, Douglas C. Schmidt, Nanbor Wang, Vishal Kachroo, and Aniruddha Gokhale. Using Principle Patterns to Optimize Real-time ORBs. *Concurrency Magazine,* 8(1), 2000.

[PS90] Adam Porter and Richard Selby. Empirically Guided Software Development Using Metric-Based Classification Trees. *IEEE Software,* March 1990.

[PTM97] Jelica Protic, Milo Tomaevic, and Veljko Milutinovic. *Distributed Shared Memory Concepts and Systems.* IEEE Computer Society Press, 1997.

[Rag93] Steve Rago. *UNIX System V Network Programming.* Addison-Wesley, Reading, Massachusetts, 1993.

[Ray01] Eric Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* O'Reilly, February 2001. http://www.oreilly.com/catalog/cathbazpaper/.

[Ric97] Jeffery Richter. *Advanced Windows, Third Edition.* Microsoft Press, Redmond, WA, 1997.

[RJB98] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley Object Technology Series. Addison-Wesley, Reading, Massachusetts, 1998.

[SBS93] Douglas C. Schmidt, Donald F. Box, and Tatsuya Suda. ADAPTIVE: A Dynamically Assembled Protocol Transmission, Integration, and eValuation Environment. *Journal of Concurrency: Practice and Experience,* 5(4):269—286, June 1993.

[SC00] Douglas C. Schmidt and Chris Cleeland. Applying a Pattern Language to Develop Extensible ORB Middleware. In Linda Rising, editor, *Design Patterns in Communications.* Cambridge University Press, 2000.

[Sch90] Douglas C. Schmidt. *GPERF:* A Perfect Hash Function Generator. In *Proceedings of the 2nd C++ Conference,* pages 87—102, San Francisco, California, April 1990. USENIX.

[Sch94] Curt Schimmel. *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers.* Addison-Wesley, Reading, Massachusetts, 1994.

[Sch00] Douglas C. Schmidt. Applying a Pattern Language to Develop Application-level Gateways. In Linda Rising, editor, *Design Patterns in Communications.* Cambridge University Press, 2000.

[SH] Douglas C. Schmidt and Stephen D. Huston. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks.* Addison-Wesley, Boston (forthcoming).

[SKKK00] Douglas C. Schmidt, Vishal Kachroo, Yamuna Krishnamurthy, and Fred Kuhns. Appying QoS-enabled Distributed Object Computing Middleware to Next-generation Distributed Applications. *IEEE Communications Magazine,* 38(10):112—123, October 2000.

[SLM98] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The Design and Performance of Real-Time Object Request Brokers. *Computer Communications,* 21(4):294-324, April 1998.

[SMFG01] Douglas C. Schmidt, Sumedh Mungee, Sergio Flores-Gaitan, and Aniruddha 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,* 21(2), 2001.

[Sol98] David A. Solomon. *Inside Windows NT, 2nd Ed.* Microsoft Press, Redmond, Washington, 2nd edition, 1998.

[SP01] Douglas C. Schmidt and Adam Porter. Leveraging Open-Source Communities to Improve the Quality and Performance of Open-Source Software. In *First Workshop on Open-Source Software Engineering, 23rd International Conference on Software Engineering,* May 2001.

[SS93] Douglas C. Schmidt and Tatsuya Suda. Transport System Architecture Services for High-Performance Communications Systems. *IEEE Journal on Selected Areas in Communication*, 11(4):489-506, May 1993.

[SS94] Douglas C. Schmidt and Tatsuya Suda. An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems. *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, 2:280-293, December 1994.

[SS95] Douglas C. Schmidt and Tatsuya Suda. Measuring the Performance of Parallel Message-based Process Architectures. In *Proceedings of the Conference on Computer Communications (INFOCOM)*, pages 624-633, Boston, April 1995. IEEE.

[SS01] Richard E. Schantz and Douglas C. Schmidt. Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications. In John Marciniak and George Telecki, editors, *Encyclopedia of Software Engineering*. Wiley & Sons, New York, 2001.

[SSRB00] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[Ste92] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, Reading, Massachusetts, 1992.

[Ste93] W. Richard Stevens. *TCP/IP Illustrated, Volume 1*. Addison-Wesley, Reading, Massachusetts, 1993.

[Ste96] W. Richard Stevens. *TCP/IP Illustrated, Volume 3*. Addison-Wesley, Reading, Massachusetts, 1996.

[Ste98] W. Richard Stevens. *UNIX Network Programming, Volume 1: Networking APIs: Sockets and XTI, Second Edition*. Prentice-Hall, Englewood Cliffs, NJ, 1998.

[Ste99] W. Richard Stevens. *UNIX Network Programming, Volume 2: Interprocess Communications, Second Edition*. Prentice-Hall, Englewood Cliffs, NJ, 1999.

[Sun98] Sun Microsystems, Inc. *Java Remote Method Invocation Specification (RMI)*, October 1998.

[Szy98] Clemens Szyperski. *Component Software—Beyond Object-Oriented Programming*. Addison-Wesley, Reading, Massachusetts, 1998.

[Tan96] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, Upper Saddle River, New Jersey, 3rd edition, 1996.

[wKS00] Martin Fowlers with Kendall Scott. *UML Distilled – A Brief Guide to the Standard Object Modeling Language (2nd Edition)*. Addison-Wesley, Boston, 2000.

[Woo97] Bobby Woolf. The Null Object Pattern. In Robert Martin, Frank Buschmann, and Dirk Riehle, editors, *Pattern Languages of Program Design*. Addison-Wesley, Reading, Massachusetts, 1997.

索引

A

Abstract types, 30
accept(), 65, 66
Acceptor-Connector pattern, 15
Acceptors, nonblocking, 145
ACE (ADAPTIVE Communication Environment), 8, 12-17
  building, and programs that use, 19
  C++ wrapper facade layer, 14-15
  displaying classes, 50
  downloading, 12
  evolution of, 259-267
  framework layer, 15-16
  future for, 267-268
  layered architecture of, 12, 13
  network services, 16-17
  OS adaptation layer, 13-14
  web site for, 14
ACE::read_n(), 76
ACE::select(), 141, 151-152
ACE::write_n(), 76
ACE_Addr, 49-52
  hash(), 51
  operator != (), 51
  operator == (), 51
ACE_Condition_Thread_Mutex. 208, 229-230
  broadcast(), 229
  signal(), 229
  wait(), 229
ACE_const_cast, 176
ACE_Data_Block, 72
ACE_DEBUG, 93
ACE_dynamic_cast, 176
ACE_ERROR, 93
ACE_ERROR_RETURN, 92, 93
ACE_FILE_Connector, 85
ACE_FILE_IO, 85
ACE_Guard, 208-212, 216-217
ACE_Handle_Set, 140-147
  clr_bit(), 143
  fdset(), 143
  is_set(), 143
  max_set(), 143
  num_set(), 143
  reset(), 143
  set_bit(), 143
  sync(), 143
ACE_Handle_Set_Iterator, 140, 147-151
  operator(), 148
ACE_Hash_Map_Manager, 154, 155
ACE_INET_Addr, 49-52
  addr_to_string(), 52
  get_host_name(), 52
  get_port_number(), 52
  string_to_addr(), 52
ACE_InputCDR, 76-80

good_bit(), 78
operator >> (), 78
steal_contents(), 78
ACE_IPC_SAP, 52-53
disable(), 53, 62
enable(), 53, 62
get_handle(), 53
set_handle(), 53
ACE_Log_Msg::Log(), 93
ACE_Mem_Map, 66
ACE_Message_Block, 72-76
clone(), 74
clr_flags(), 74
copy(), 74
duplicate(), 74
length(), 74
msg_priority(), 74
msg_type(), 74
next(), 74
prev(), 74
rd_ptr(), 74
release(), 74
set_flags(), 74
size(), 74
total_length(), 74
wr_ptr(), 74
ACE_Message_Queue, 72
ACE_Null_Condition, 208, 230-231
ACE_Null_Mutex, 208, 212-217
ACE_Null_Semaphore, 208, 222-229
ACE_Object_Manager, 217-218
ACE_OutputCDR, 76-80
begin(), 78
end(), 78
good_bit(), 78
operator << (), 78
total_length(), 78
ACE_Process, 160-165
child(), 163
exit_code(), 162
getpid(), 162
kill(), 163
parent(), 163
prepare(), 162
spawn(), 162
terminate(), 162
unmanage(), 162
wait(), 162
ACE_Process_Manager, 160, 182
close(), 170
instance(), 170
open(), 170
spawn(), 170
spawn_n(), 170
wait(), 170
ACE_Process_Manager, 169
ACE_Process_Mutex, 208, 212-217
ACE_Process_Options, 160, 165-169
avoid_zombies(), 168
creation_flags(), 168
set_process_attributes(), 168
command_line(), 167
pass_handle(), 167
set_handles(), 167
setenv(), 167
seteuid(), 168
setruid(), 168
working_directory(), 167
ACE_Process_Semaphore, 208, 222-229
ACE_Read_Guard, 208-212, 219-221
ACE_Recursive_Thread_Mutex, 231-233
ACE_reinterpret_cast, 176
ACE_RW_Process_Mutex, 208, 219-221
ACE_RW_Thread_Mutex, 208, 219-221
ACE_Sched_Params, 186, 198-201
next_priority(), 199
prev_priority(), 199
priority_max(), 199

priority_min(), 199
ACE::select(), 140
ACE_SOCK, 54-55
close(), 55
get_local_addr(), 55
get_option(), 55
get_remote_addr(), 55
open(), 55
set_option(), 55
ACE_SOCK_Acceptor, 64-67
accept(), 66
open(), 66
ACE_SOCK_Connector, 56-60
complete(), 58
connect(), 58
ACE_SOCK_IO, 60-64
ACE_SOCK_Stream, 60-64
recv(), 62
recv_n(), 62
recvv_n(), 62
send(), 62
send_n(), 62
sendv_n(), 62
ACE_static_cast, 176
ACE_Task, 205
ACE_Thread_Manager, 186-198
cancel_all(), 189
close(), 189
exit(), 189
instance(), 189
join(), 189
spawn(), 189
spawn_n(), 189
testcancel(), 189
wait(), 189
ACE_Thread_Mutex, 208, 212-217
ACE_Thread_Semaphore, 208, 222-229
ACE_Time_Value, 58, 62
ACE_TSS, 186, 187, 201-205
cleanup(), 203
operator -> (), 203
ACE_Write_Guard, 208-212, 219-221
Active Object, 205
Active Object pattern, 16, 133
addr_to_string(), 52
Address family, 37
Arrays of primitive types, 77
Asynchronous I/O, 108
Asynchronous message exchange, synchronous versus, 26-28
avoid_zombies(), 168

B

Barrier synchronization, 197
begin(), 78
Blocking, 56, 58, 61, 62, 65
broadcast(), 229

C

C++ code, displaying, 50
C++ iostreams, 76
C++ wrapper facade layer, 14
cancel_all(), 189
Casts, 176
child(), 163
cleanup(), 203
clone(), 74
close(), 55, 170, 189
clr_bit(), 143
clr_flags(), 74
COM+, 9
command_line(), 167
Commercial off-the-shelf(COTS), 9, 10
Common middleware service layer, role of, 9
Communication design
connectionless versus connection-oriented protocols, 23-26
message passing versus shared memory, 28-31
synchronous versus asynchronous message exchange, 26-28

Communication dimension, 6
Communication domain, 36
complete(), 58
Component, 16
Component Configurator Pattern, 16-17
Concrete class, 30
Concurrency design, 6
  concurrent servers, 105-106
  iterative servers, 103-105
  limitations with OS, 135-136
  process/thread spawning strategies, 112-114
  processes versus threads 109-112
  reactive servers, 106-108
  real-time scheduling, 119-121
  task- versus message-based architectures, 121-122
  threading models, 114-119
  time-shared scheduling, 119-121
Concurrency framework, 15
Concurrent servers, 105-106
Condition variables, 133-134
Configuration dimensions, 6
connect(), 56, 58
Connection establishment and termination, Socket API, 35
Connection establishment framework, 15
Connectionless versus connection-oriented protocols, 23-26
const-cast, 176
Container classes, 155
Contention scope, 114
Cooperative cancellation, 190-191, 197
copy(), 74
CORBA, 9, 29
  Common Data Representation (CDR), 77
Cost containment, 11
CreateProcess(), 109, 161
CreateThread(), 110, 129
creation_flags(), 168
CreateProcess(), 128

D

Data framing strategies, 24
Data transfer mechanism, Socket API, 35
Data-mode socket, 56
Deadlocks, 107
Debugging macros, 93
Demarshaling, 9, 11, 76-77
dequeue_head(), 227-228
Descriptor, *see* Handles
disable(), 53, 62
Dispatching framework, 15
Distributed shard memory (DSM), 30-31
Distribution middleware, role of, 8-9
Domain analysis, 5
Domain-crossing penalty, 91
Domain-specific middleware services layer, role of, 9
Double-Checked Locking Optimization pattern, 191, 192, 203
duplicate(), 74
dynamic_cast, 176

E

Eager spawning, 112
echo_server(), 237
echo_server, 37-38
Efficiency issues, 46
enable(), 53, 62
end(), 78
EndPoints, 34
enqueue_tail(), 226-227
errno, 130, 202
Error macros, 93
Error propagation strategies, 137
Escape hatches, 237-238
Event demultiplexing framework, 15

synchronous, 125-127
Event loops, 125
exit(), 111, 128, 189
exit_code(), 162
ExitProcess(), 111, 128
ExitThread(), 129
F
FD_CLR(), 127
FD_ISSET(), 127
FD_SET(), 127
fd_set, 126, 141-143, 148-150
FD_ZERO(), 127
fdset(), 143
First-in, first-out (FIFO), 120, 198
for loop, 147
fork(), 109, 128, 161, 164
Framework layer, 15-16
G
get_handle(), 53, 238
get_host_name(), 52
get_local_addr(), 55
get_option(), 55
get_port_number(), 52
get_remote_addr(), 55
gethostbyname(), 84
gepid(), 162
GetThreadPriority(), 130
good_bit(), 78
H
Half-Sync/Half-Async pattern, 16, 112, 206
handle_connections(), 84, 91, 93, 156
handle_connections, 177-179
handle_data(), 84, 91, 94, 146, 156-157, 196-197
Handles, 34
errors and, 37-40
hash(), 51
Hook methods, Logging_Server, 83-84
Host infrastructure middleware layer, role of, 8, 10-13
Hybrid-threading model, 116-117
I
instance(), 170, 189
Internet Protocol (IP), 24
Interprocess communication (IPC), local and remote, 33
iovec structure, 63
is_set(), 143
Iterative servers, 103-105
Iterative_Logging_Server, 91-95
Iterator pattern, 148
J
Java Package, 8
Java RMI, 9
JAWS, 265
Jitter, 11
join(), 189
K
Kernel-threading model, 115-116
kill(), 161, 163
L
Last-in, first-out (LIFO), 218
Leader/Followers pattern, 112
length(), 74
Lightweight processes (LWPs), 116-117
Linearization, 76
Local context management, Socket API, 35
Local shared memory, 29
Lock-step sequence, 27
Locking, 203, 210-212, 218-224
log_record(), 91
Logging service
asynchronous request/response protocol, 28
client application, 95-98
example of, 17-19
initial, 80-95
message framing protocol, 86
message passing, 31
TCP/IP connection, 25
Logging service, implementing

ACE_InputCDR, 72-80
ACE_OutputCDR, 76-80
Logging_Client::send(), 95-97
Logging_Handler, 86-91
log_record(), 91
recv_log_record(), 87-90
write_log_record(), 90-91
Logging_Process, 180-182
Logging_Server, 81-86
handle_connections(), 84, 91
handle_data(), 84, 91
hook methods, 83-84
make_log_file(), 85-86
open(), 83-84
run(), 83
wait_for_multiple_events(), 84

M

Macros
debugging and error, 93
guard, 216
main(), 94, 157
make_log_file(), 85-86, 92, 156
Marshaling, 9, 11, 76-77
max_set(), 143
Memory management unit (MMU), 109
Memory-mapped files, 29-30
Message exchange, synchronous versus asynchronous, 26-28
Message passing versus shared memory, 28-31
Message(s)
composite, 73
framing protocol, 86
simple, 73
Message-based concurrency architecture, 121-122
Message-oriented middleware (MOM), 29
Message_Queue, 223-229
Microsoft Windows, 34
Middleware standards, 263-264
Monitor Object pattern, 133, 224
msg_priority(), 74
msg_type(), 74
Multiplexing connections, 24-25
Multiprocessing
advantages and disadvantage of, 109-110
mechanisms, 127-128
spawning strategies, 112-114
Multiprocessing wrapper facades
ACE_Process, 161-165
ACE_Process_Manager, 169-182
ACE_Process_Options, 165-169
overview of, 159-161
Multithreading
advantages and disadvantages of, 110-112
mechanisms, 129-130
models, 114-119
spawning strategies, 112-114
Multithreading wrapper facades
ACE_Sched_Params, 186, 198-201
ACE_Thread_Manager, 186-198
ACE_TSS, 186, 187, 201-205
overview of, 185-187
Mutual exclusion (mutex) locks, *see also* ACE_Condition_Thread_Mutex, ACE_Null_Mutex, ACE_Process_Mutex, ACE_RW_Process_Mutex, ACE_RW_Thread_Mutex, and ACE_Thread_Mutex, 105, 132, 134

N

N: 1 user-threading model, 114-115
N:M hybrid-threading model, 116-117
Nagle's algorithm, 55, 63, 64
Network addressing, Socket API, 36
Network services, library of, 16-17
Networked applications
challenges of, 1-4

design dimensions, 5-7
example of, 2-3
next(), 74
next_priority(), 199
Nonblocking, 56, 58, 61, 62, 65
Nonmultiplxing connections, 25
num_set(), 143
O
Object Lifetime Manager pattern, 218
Object Request Brokers (ORBs), 9
Object, differences between a thread And an, 194
Object-oriented middleware
benefits of host infrastructure, 10-13
layers, 7-10
role of, 7, 9-10
On-demand spawning, 113
1:1 kernel-threading model, 115-116
open(), 55, 66, 155, 170
Logging_Server, 83-84, 92
operator!=(), 51
operator(), 148
operator->(), 203
operator<<(), 78-79
operator==(), 51
operator>>(), 78-80
Options management, Socket API, 36
OS adaptation layer, 13-14
P
parent(), 163
pass_handle(), 167
Passive-mode socket, 56
Pattern
Acceptor-Connector, *see* Acceptor-Connector pattern
Active Object, *see* Active Object pattern
Component Configurator, *see* Component Configurator pattern
Double-Checked Locking Optimization, *see* Double-Checked Locking Optimization pattern
Half-Sync/Half-Async, *see* Half-Sync/Half-Async pattern
Iterator, *see* Iterator pattern
Leader/Followers, *see* Leader/Followers pattern
Monitor Object, *see* Monitor Object pattern
Object Lifetime Manager, *see* Object Lifetime Manager Pattern
Pipes and Filters, *see* Pipes and Filters pattern
Proactor, *see* Proactor framework
Reactor, *see* Reactor framework
Singleton, *see* Singleton pattern
Thread-Safe Interface, *see* Thread-Safe Interface pattern
Wrapper Facade, *see* Wrapper Facade pattern
PEER_ADDR, 57
PEER_STREAM, 57
Pipes and Filters pattern, 16
poll(), 126
Portability, 46, 52, 135, 136, 152, 164, 176
lack of, Socket API, 41-43
Ports, ephemeral, 51
POSIX, 161, 163
prepare(), 162
prev(), 74
prev_priority(), 199
Primitive types, 77
arrays of, 77
printf(), 93
Priority inversion, 25
priority_max(), 199
priority_min(), 199
Proactive servers, 108

Proactor framework, 15
Process, *see also* Multiprocessing
  contention scope, 114
  lifetime operations, 128
  pool, 106
  property operations, 128
  synchronization operations, 128
Process-per-connection, 171-180
Protocol stacks, 7
Protocols
  connectionless versus connection-oriented, 23-26
  defined, 23
  family, 36
pthread_cancel(), 129, 187
pthread_create(), 110, 129
pthread_exit(), 129
pthread_getschedparam(), 130
pthread_getspecific(), 130
pthread_join(), 129
pthread_key_create(), 130
pthread_kill(), 188
pthread_setschedparam(), 130
pthread_setspecific(), 130
pthread_testcancel(), 187

Q

Quality of service (QoS) requirements, 10-11

R

Race conditions, 130-132
rd_ptr(), 74
Reactive servers, 106-108
Reactor framework, 15
read_n(), 76
Readers/writer locks, 132-133
Real-time scheduling, 119-121
recv(), 56, 62
recv_log_record(), 87-90
recv_n(), 62
Recvv_(), 62
reinterpret_cast, 176
release(), 74
Request/response protocols, asynchronous and synchronous, 26-28
reset(), 143
Reuse, 10
Round-robin, 120, 198-199
RPC, 29
run(), 83, 173, 174
run_master(), 174-175
run_svc(), 194-196
run_worker(), 175-176

S

Scheduler activations, 116
Scoped Locking, 203, 210-212, 224
select(), 107, 126-127, 140, 151-152
Semantic variations, 187
Semaphores, *see also* ACE_Null_Semaphore, ACE_Process_Semaphore, and ACE_Thread_Semaphore, 105, 133-134
send(), 56, 62
send_n(), 62
sendv_n(), 62
Serialization, 30
Service configurator framework, 16-17
Service dimensions, 6
Service initialization framework, 15
set_bit(), 143
set_flags(), 74
set_handle(), 53, 238
set_handles(), 167
set_option(), 55
set_process_attributes(), 168
setenv(), 167
seteuid(), 168
setruid(), 168
SetThreadPriority(), 130
Shared memory
  C++ objects and, 30
  distributed, 30-31

local, 29-30
message passing versus, 28-31
signal(), 229
Singleton pattern, 191, 192
size(), 74
Sleep locks, 133
sockaddr, 49
Socket API
address family, 37
Connection establishment and termination, 35
Data transfer mechanisms, 35
limitations of, 37-43
local context management, 35
network addressing, 36
nonportable and nonuniform, 41-43
options management, 36
protocol family, 36
role of, 34
Socket wrapper facades
ACE_Addr, 49-52
ACE_INET_Addr, 49-52
ACE_IPC_SAP, 52-53
ACE_SOCK, 54-55
ACE_SOCK_Acceptor, 64
ACE_SOCK_Connector, 56-60
ACE_SOCK_IO, 60-64
ACE_SOCK_Stream, 60-64
benefits of, 46
overview of, 45-49
relationships between, 47
structure of, 46
using traits for, 57
Sockets, 34
spawn(), 162, 170, 189
spawn_n(), 170, 189
Spawning
of threads, 195
of worker processes, 175, 178
strategies for processes and threads, 112-114
Spin locks, 132
Stand-alone applications, example of, 2
Standards, open, 10
static_cast, 176
steal_contents(), 78
Strategic focus, 9
Strategized Locking pattern, 215
Streams framework, 16
string_to_addr(), 52
structtimeval, 127
sync(), 143
Synchronization mechanisms, 130-134
Synchronization wrapper facades
ACE_Condition_Thread_Mutex, 208, 229-230
ACE_Guard, 208-212, 216-217
ACE_Null_Condition, 208, 230-231
ACE_Null_Mutex, 208, 212-217
ACE_Null_Semaphore, 208, 222-229
ACE_Process_Mutex, 208, 212-217
ACE_Process_Semaphore, 208, 222-229
ACE_Read_Guard, 208-212, 219-221
ACE_Recursive_Thread_Mutex, 231-233
ACE_RW_Process_Mutex, 208, 219-221
ACE_RW_Thread_Mutex, 208, 219-221
ACE_Thread_Mutex, 208, 212-217
ACE_Thread_Semaphore, 208, 222-229
ACE_Write_Guard, 208-212, 219-221
overview of, 207-209

Synchronous event demultiplexing, 125-127
Synchronous event demultiplexing, wrapper facades
ACE::select(), 141, 151-152
ACE_Handle_Set, 141-147
ACE_Handle_Set_Iterator, 147-151
ACE_Handle_Set, 140
ACE_Handle_Set_Iterator, 140
overview of, 139-141
Synchronous versus asynchronous message exchange, 26-28
Syntactic variations, 187
System contention scope, 114
System V STREAMS, 73
System V UNIX shared memory, 29
T
TAO, 264-265
Task, 205
Task framework, 15
Task-based concurrency architecture, 121
Template method, Logging_Server::run(), 83
terminate(), 162
TerminateProcess(), 128, 161
TerminateThread(), 129
testcancel(), 189
THR_DETACHED, 190
THR_JOINABLE, 190
thr_kill(), 188
THR_NEW_LWP, 190
THR_SCOPE_PROCESS, 190
THR_SCOPE_SYSTEM, 190
Thread, *see also* Multithreading
difference between an object and a, 194
lifetime operations, 129
pool, 106
property operations, 130
spawning of, 195
specific storage, 130
synchronization operations, 129
Thread-per-connection, 106, 191-198
Thread-per-request concurrent server, 105-106
Thread-Safe Interface pattern, 224, 226
Thread-specific storage (TSS), *see also* ACE_TSS, 130
Timed socket operations, 56, 58, 60-62, 65
Timeouts, 62-64, 152
timeval, 127
TlsAlloc(), 130
TlsGetValue(), 130
TlsSetValue(), 130
total_length(), 74, 78
TP4, 24
Traits, 57
Transmission Control (TCP), 24-26
TSS, *see* Thread-specific storage
Type errors, 46
U
UNIX, 34, 35, 161, 163
unmanage(), 162
User Datagram Protocol (IP), 24
User-threading model, 114-115
V
Virtual methods, 30
W
wait(), 128, 162, 170, 189, 229
wait_for_multiple_events(), 84, 93, 144-145, 155-156
WaitForMultipleObjects(), 107, 126, 128, 129, 161
WaitForSingleObject(), 128, 129, 161
waitpid(), 128, 161
Wildcard, 51
Win32, 161
working_directory(), 167
wr_ptr(), 74

Wrapper Facade pattern, 14, 45, 85
hide platform differences, 248-254
hierarchies to enhance clarity and extensibility, 246-248
optimize for efficiency, 255-257
simplify for common case, 238-246
to enhance type safety, 236-238
write_log_record(), 90-91
write_n(), 76
writev(), 63
WSASend(), 63
X
XTP, 24

[General Information]
书名=C++网络编程 卷1 运用ACE和模式消除复杂性
作者=[美]Douglas C.Schmidt Stephen D.Huston著 於春景译
页数=303
SS号=11074903
出版日期=

书名页
版权页
前言
目录
正文