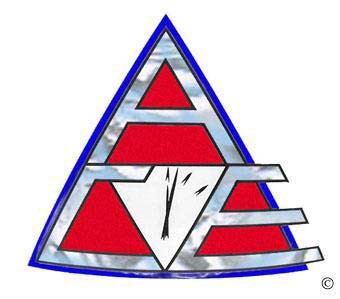
ACE 自适配通信环境中文技术文档

上篇：ACE 技术论文集



作者：Douglas C. Schmidt 等 [http://www.cs.wustl.edu/~schmidt/](http://www.cs.wustl.edu/%7Eschmidt/)

译者：马维达 <http://www.flyingdonkey.com/>

第 **1** 章 **ACE** 自适配通信环境：用于开发通信软件的面向对象网络编

程工具包

Douglas C. Schmidt

摘 要

*ACE* 自适配通信环境（*Adaptive Communication Environment*）是一种面向对象（*OO*）的工具包，它 实现了通信软件的许多基本的设计模式。*ACE* 的目标用户是在 *UNIX* 和 *Win32* 平台上开发高性能通信服 务和应用的开发者。*ACE* 简化了使用进程间通信、事件多路分离、显式动态链接和并发的 *OO* 网络应用 和服务的开发。通过在运行时将服务与应用动态链接进应用，并在一个或多个进程或线程中执行这些服 务，*ACE* 使系统的配置和重配置得以自动化。

本论文描述 *ACE* 的结构和功能，并使用来自像电信、企业级医学成像和 *WWW* 服务这样的领域的例 子阐释核心的 *ACE* 特性。*ACE* 可以自由使用，并正在被用于许多商业项目（比如爱立信、*Bellcore*、西 门子、摩托罗拉、柯达，和 *McDonnell Douglas*），以及许多学院和工业研究项目。*ACE* 已被移植到多种 *OS*（操作系统）平台上，包括 *Win32* 和大多数的 *UNIX/POSIX* 实现。此外，同时有 *C++*和 *Java* 版本的 *A CE* 可用。

**1.1** 介绍

**1.1.1** 问题：分布式软件危机

对健壮的和高性能的分布式计算系统的需求一直在稳定地增长。这些类型的系统的例子包括全球个 人通信系统、网络管理平台，企业级医学成像系统、在线金融分析系统，以及实时航空控制系统。对于 以下方面来说，分布式计算是一种有前途的技术：通过连接性和相互配合促进协作；通过并行处理改善 性能；通过复制改善可靠性和可用性；通过模块性改善可伸缩性和可移植性；通过动态配置和重配置改 善可扩展性；以及通过资源共享和开发系统提高成本效用。

尽管分布式计算提供了许多潜在的好处，开发通信软件仍然是昂贵而易错的。面向对象编程语言、 组件和框架（Framework）是被广泛鼓吹的、用以降低软件成本并提高软件质量的技术。去除那些过分的 宣传，OO 的主要好处源于对模块性和可扩展性的强调，它将易变的实现细节封装在稳定的接口后面，并 增强了软件的复用。

多年来，在某些已被广泛探索的领域中的开发者已经成功地应用了 OO 技术和工具。例如，Microso ft MFC GUI 框架和 OCX 组件是 PC 平台上用于创建图形商业应用的事实上的工业标准。尽管这些工具 有着自身的局限，它们仍然演示了复用通用框架和组件的生产效率优势。

在像电信、医学成像、航空控制和在线事务处理这样的更复杂的领域中，软件开发者历来就缺少标 准的、成型的中间件组件。结果，开发者在很大程度上是从头开始构建、验证和维护软件系统。在一个 政府经济干预减少的艰难的全球竞争时代，这样的作坊式开发过程正在变得难以容忍的昂贵和费时。在 业界，这样的情形导致了一场“分布式软件危机”：计算硬件和网络在变小、变快、变得更为便宜；而分 布式软件的开发和维护在变大、变慢、变得更为昂贵。

构建分布式软件的挑战源于与分布式系统相关联的固有的和非固有的复杂性[1]。固有的复杂性源于 开发分布式软件的基本的挑战，其中主要的有：检测和恢复网络及主机失败、最小化通信响应延迟的影 响，以及确定服务组件和工作负载在网络的处理单元上的最优划分。

非固有的复杂性源于用以开发分布式软件的工具和技术的局限。例如，许多标准的网络机制（比如 s ocket[2]和 TLI[3]）和可复用组件库（比如 X windows 和 Sun RPC）缺乏类型安全的、可移植的、可重入 的和可扩展的应用编程接口（API）。同样地，通用网络编程接口，如 socket 和 TLI，使用弱类型的整型 句柄，可能会导致微妙的运行时错误[4]。

复杂性的另一来源起因于算法分解的普遍使用[5]，它致使软件系统不可扩展和不可复用[6]。尽管图 形用户接口（GUI）普遍采用面向对象技术构建，典型的分布式软件通常仍然使用算法分解进行开发。在 一些流行的网络编程教科书[7, 8, 3]中的例子基于面向算法的设计和实现技术，从而更加恶化了前述问 题。

可扩展性和最大限度复用的缺乏对于复杂的分布式软件是特别成问题的。可扩展性是确保服务和特 性的及时修改和增强的基本要求。复用是有效利用专家开发者的领域知识、以避免重新开发和重新验证 “反复出现的需求和软件挑战的通用解决方案”的基本要求。

**1.1.2** 解决方案：面向对象的设计模式和框架

面向对象的设计模式和框架有助于减少对分布式软件的核心概念和抽象的昂贵的重新发现和发明， 它们因此而备受重视。模式提供了一种封装设计知识的方法，这些设计知识为标准的分布式软件开发问 题提供解决方案[9]。例如，模式对于描述重复出现的“微型结构”（比如反应器（Reactor）[10]和主动对 象（Active Object）[11]）十分有用，这些微型结构是对一些已被证明可用于构建分布式通信软件的通用 对象结构的抽象。但是，被文档化为模式的抽象并不直接产生可复用代码。因此，有必要通过框架的创 建和使用来增加对模式的研究。

通过集成成组的抽象类，并定义这些类的协作的标准途径，框架为应用提供了可复用的软件组件[12]。 框架实例化设计模式族，以帮助开发者避免对通用分布式软件组件的昂贵的重新发明。其成果是“半完 成”的应用骨架，它可以通过继承和实例化框架中的可复用“积木”组件来进行定制。因为框架与关键 的分布式编程任务（比如服务初始化、错误处理、流控制、事件多路分离、并发控制）紧密地集成在一 起，复用的范围可以显著地大于使用传统函数库，甚或是通常的 OO 类库。

本论文被组织如下：1.2 给出 ACE 工具包的结构和功能的综述；1.3 详细描述 ACE C++包装组件和 较高级的 ACE 框架组件及模式；1.4 检查若干使用 ACE 构建的网络应用的实现；还有 1.5 给出结束语

**1.2 ACE** 综述

为阐释 OO 模式和框架是怎样被成功地应用于分布式软件的，本论文考查自适配通信环境（ACE）[6]。

ACE 是可以自由使用的 OO 工具包，其中包含有丰富的、可跨越广泛的 OS 平台执行通用网络编程任务

的可复用包装、类属，以及框架。ACE 提供的任务包括：

z 事件多路分离和事件处理器分派[13, 14, 10, 15]；

z 连接建立和服务初始化[16, 17, 18]； z 进程间通信[19, 4]和共享内存管理； z 分布式通信服务的动态配置[20, 21]； z 并发*/*并行和同步[22, 23, 11, 24]；

z 更高级的分布式服务组件（比如名字服务、事件服务、日志服务、时间服务和令牌服务）。

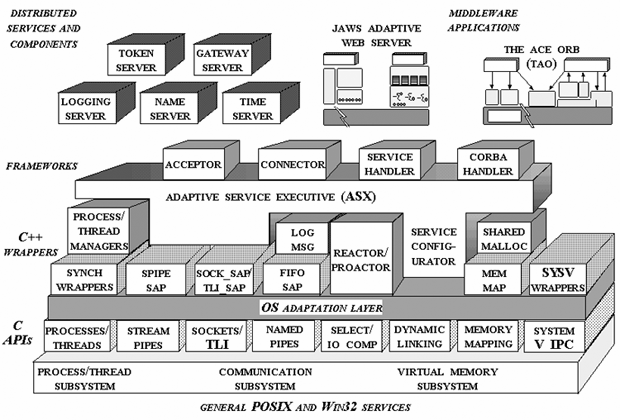


图 1-1 ACE 自适配通信环境中的组件

ACE 工具包的设计采用分层的体系结构。图 1-1 演示了 ACE 组件间的纵向和横向关系。ACE 的较低 层是封装现有的 OS 网络编程机制的 OO 包装（wrapper）。ACE 的高层扩展这些包装，以提供 OO 框架和 组件、覆盖更为广泛的面向应用的网络任务和服务。这一部分的余下部分给出对 ACE 中类属的结构和功 能的综述（如图 1-2 所示）。1.3 提供了对 ACE 的网络编程特性和组件的深入讨论。

贯穿本论文，ACE 组件通过 Booch 表示法[5]来进行图解。实心矩形表示类属，它将一定数量的相关 类合成进一个公共的名字空间。实心云表示对象；如嵌套则表示对象间的合成关系；而无方向的边表示 在两个对象间存在某种类型的链接。虚线云表示类；有向边表示类之间的继承关系；而一端有小圆圈的 无向边表示两个类之间的合成或是使用关系。在三角形内标记的“A”标识一个类为抽象类[25]。抽象类不 能被直接实例化，而必须被子类化。在实例化抽象类子类的任何对象之前，该子类必须提供所有抽象方 法的定义。1

1 通常 C++把抽象方法称为纯虚函数（pure virtual function）

**1.2.1 ACE OS** 适配层

ACE 的源码树含有超过 85,000 行 C++代码。其中大约 9,000 行代码（也就是，大约为总数的 10%） 为 *OS* 适配层所特有。该层将 ACE 的较高层和与下列 OS 机制相关联的平台特有的依赖屏蔽开来：

z 多线程和同步

z 进程间通信

z 事件多路分离

z 显式动态链接

z 内存映射文件和共享内存

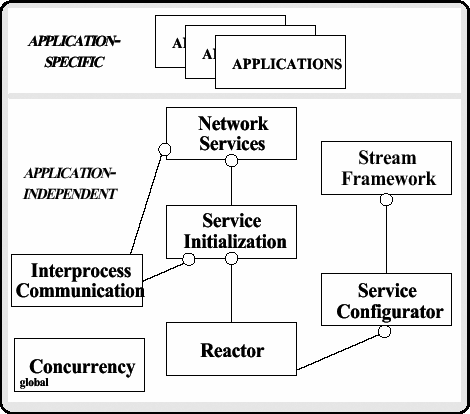


图 1-2 ACE 中的类属

**1.2.2 ACE OO** 包装

在 OS 适配层之上是许多 OO 包装，它们封装并增强在像 Win32 和 UNIX 这样的现代操作系统上可 用的并发、进程间通信（IPC）、以及虚拟内存机制（在图 1-1 底部演示）。应用可以通过有选择地继承、 聚合（aggregating）、和/或实例化下列 ACE 包装类属来合并和编写这些组件：

z **IPC SAP**－它封装本地和/或远地的 IPC 服务访问点（IPC SAP）机制，比如 socket、TLI、UNIX FI FO 和 STREAM 管道，以及 Win32 命名管道[19, 4]；

z 服务初始化－ACE 提供一组连接器（Connector）和接受器（Acceptor）组件[18]，分别使主动和被动 的初始化角色与一旦初始化完成后通信服务执行的任务去耦合（decouple）。

z 并发机制－ACE 抽象较低级的 OS 多线程和多进程机制（比如互斥体和信号量[22]），以创建较高级 的 OO 并发抽象（比如主动对象 Active Object[11]）；

z 内存管理机制－ACE 内存管理组件为管理共享内存和局部内存的动态分配和释放提供了灵活和可扩 展的抽象；

z **CORBA** 集成－ACE 可与 CORBA 实现集成在一起（比如单线程和多线程 Orbix）。

通过采用类型安全的 OO 接口封装 OS 通信、并发和虚拟内存机制，OO 包装的使用提高了应用的健 壮性。这也减少了应用直接访问用弱类型 C 接口编写的底层 OS 库的需求。因此，像 C++和 Java 这样的 OO 语言的编译器可以在编译时、而非运行时检测类型系统违例。ACE 的 C++版本大量使用内联（inlini ng），以消除包装层提供额外的类型安全性和抽象所导致的性能下降。

**1.2.3 ACE** 框架

ACE 含有一个更高层的网络编程框架，集成并增强了较低层的 OS 包装。该框架支持由应用服务组 成的并发网络看守的动态配置。ACE 的框架部分包括以下类属：

z 反应器（**Reactor**）－ACE 反应器[10]提供可扩展的、面向对象的多路分离器，它分派处理器、以响 应多种类型的事件（例如，基于 I/O 的、基于定时器的、基于信号的，以及基于同步的事件）；

z 服务配置器（**Service configurator**）－ACE 服务配置器[21]支持这样的服务构造：其服务可在安装时 和/或运行时动态配置；

z 流（**Stream**）－ACE 流组件[6]简化了由一或多个层次相关的服务（比如协议栈）组成的并发通信应 用的开发。

**1.2.4 ACE** 网络服务组件

除了包装和框架，ACE 还提供了一个网络服务组件的标准库。这些组件在 ACE 中扮演两种角色：

1. 它们演示怎样利用 ACE IPC 包装、反应器、服务配置器、服务初始化、并发、内存管理，以及流组 件；

2. 它们为常见的分布式系统任务提供了可复用组件，比如日志[13]、名字、锁定和时间同步[21]；

当与 OO 语言特性（比如类、继承、动态绑定和参数化类型）和设计模式（比如抽象工厂（Abstract Factory）、构建器（Builder）和服务配置器）相结合时，可复用的 ACE 组件促进了通信服务和应用的开 发，无需修改、重编译、重链接、甚至不用重启运行中的软件，它们就可以被更新和扩展[20]。

**1.3 ACE** 组件详述

**1.3.1 IPC\_SAP**：本地和远地 **IPC** 机制

ACE 提供了植根于 IPC SAP（“进程间通信服务访问点”）基类的一个类属“森林”。标准的基于 I/O 句柄的 OS 本地和远地 IPC 机制提供了面向连接和无连接的协议，IPC SAP 对这些机制进行了封装。如 图 1-3 所示，该类属“森林”包括 SOCK SAP（封装 socket API）、TLI SAP（封装 TLI API）、SPIPE S AP（封装 UNIX SunOS 5.x STREAM 管道 API），以及 FIFO SAP（封装 UNIX 命名管道 API）。

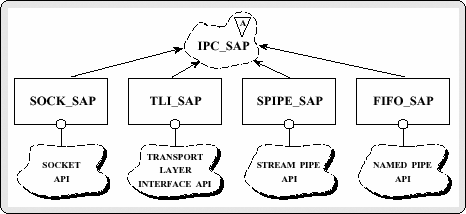


图 1-3 IPC SAP 类属关系

每一类属都被组织成一个继承层次。所有子类都提供定义良好的、本地或远地通信机制的子集的接 口。在一个层次里的子类共同包含了一种特定通信抽象的全部功能（比如 Internet 域或 UNIX 域协议族）。 类的使用（相对于单独的函数）帮助简化了网络编程：

z 使应用与易错的细节相屏蔽：例如，图 1-3 中所示的 ACE\_Addr 类层次通过类型安全的 OO 接口支持 若干不同的网络寻址格式，而非直接使用麻烦而易错的基于 C 的 struct sockaddr 数据结构。

z 合并若干操作、以形成单一操作：例如，SOCK Acceptor 的构造器执行创建被动模式服务器端点所需 的多个 socket 系统调用（比如 socket、bind 和 listen）。

z 将 *IPC* 机制参数化进应用：类构成了通过所需的 IPC 机制的类型来参数化应用的基础。如 1.3.1.2 所 讨论的，这样有助于改善可移植性。

z 增强代码共享：基于继承的层次划分增加了在多种 IPC 机制间共享的通用代码的数量（比如像 fcntl

和 ioctl 这样的低级 OS 设备控制系统调用的 OO 接口）。

下面的部分讨论 IPC SAP 中的每个类属。

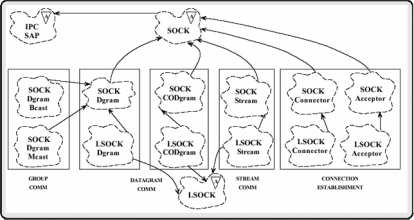


图 1-4 SOCK SAP 类属

**1.3.1.1 SOCK SAP**

SOCK SAP[4]类属为应用提供 Internet 域和 UNIX 域协议族[8]的面向对象的接口。通过继承或实例 化图 1-4 所示的适当的 SOCK SAP 子类，应用可以访问底层的 Internet 域或 UNIX 域的 socket 类型的功 能。ACE\_SOCK \*子类封装 Internet 域的功能，而 ACE\_LSOCK \*子类封装 UNIX 域的功能。如图 1-4 所 示，这些子类可进一步划分为（1）\*Dgram 组件（提供不可靠、无连接、面向消息的功能） vs. \*ACE\_ Stream 组件（提供可靠、面向连接的字节流的功能）和（2）ACE\_\*\_Acceptor 组件（提供通常用于服务 器的连接建立功能） vs. \*Stream 组件（提供同时用于客户和服务器的双向字节流传输功能）。

使用 OO 包装来封装 socket 接口有助于（1）在编译时检测许多微妙的应用类型系统违例，（2）推动 传输层接口的平台无关性，以改善应用的可移植性，以及（3）极大地减少应用代码的数量和花费在较低 级网络编程细节上的开发工作。为演示后面一点，下面的例子程序实现了一个简单的客户应用，使用 AC E\_SOCK\_Dgram\_Bcast 类来向局域网子网中所有在指定的端口号上侦听的服务器广播消息：

int main (int argc, char \*argv[])

{

ACE\_SOCK\_Dgram\_Bcast b\_sap (sap\_any);

char \*msg;

unsigned short b\_port;

msg = argc > 1 ? argv[1] : "hello world\n";

b\_port = argc > 2 ? atoi (argv[2]) : 12345;

if (b\_sap.send (msg, strlen (msg), b\_port) == -1)

perror ("can’t send broadcast"), exit (1);

exit (0);

}

把这个简洁的例子与直接使用 socket 接口实现广播所需的成打的 C 源码相比较很有启发意义。

**1.3.1.2 TLI SAP**

TLI SAP 类属提供系统 V 传输层接口（Transport Layer Interface）的 OO 接口。TLI 的 TLI SAP 继 承层次几乎与 socket 的 SCOK SAP 包装相同。主要的区别是 TLI 和 TLI SAP 并不定义 UNIX 域协议族 的接口。此外，目前 TLI 没有被移植到 Win32 平台。

通过结合 C++特性（比如缺省参数值和模板）和 tirdwr（read/write 兼容性 STREAM 模块），开发可 在编译时参数化、以在基于 socket 或基于 TLI 传输接口上正确操作的应用变得相对直截了当了。例如， 下面的代码演示怎样将 C++模板用于参数化应用所需的 IPC 机制。这些代码是从在 1.4.1 中描述的分布式 日志工具中摘录出来的。在下面的代码中，一个派生自 ACE\_Event\_Handler 的子类被一种特定类型的传 输接口及其相应的协议地址类参数化：

/\* Logging\_Handler header file \*/

template <class PEER\_STREAM, class ADDR>

class Logging\_Handler : public ACE\_Event\_Handler

{

public:

Logging\_Handler (void);

virtual ˜Logging\_Handler (void);

virtual int handle\_input (ACE\_HANDLE);

virtual ACE\_HANDLE get\_handle (void) const

{

return this->peer\_stream\_.get\_handle ();

}

protected:

PEER\_STREAM peer\_stream\_;

}

如下所示，取决于底层 OS 平台的特定属性（比如是基于 BSD 的 SunOS 4.x 还是基于系统 V 的 Sun

OS 5.x），日志应用可以实例化 Client Handler 类、以使用 SOCK SAP 或是 TLI SAP：

/\* Logging application \*/

class Logging\_Handler

#if defined (MT\_SAFE\_SOCKETS)

: public Logging\_Handler<ACE\_SOCK\_Stream, ACE\_INET\_Addr>

#else

: public Logging\_Handler<ACE\_TLI\_Stream, ACE\_INET\_Addr>

#endif /\* MT\_SAFE\_SOCKETS \*/

{

/\* ... \*/

};

在开发必须具有跨平台可移植性的应用时，这种基于模板的方法所带来的更多的灵活性极其有用。

特别地，因为 SunOS 5.2 的 socket 实现不是线程安全的，而 SunOS 4.x 的 TLI 实现含有许多严重缺陷， 在 SunOS 的变种上进行跨平台开发时，必须具有根据传输接口来参数化应用的能力。

TLI SAP 还使应用与许多 TLI 接口的特性相屏蔽。例如，在一个并发服务器中，在 qlen > 1[3]的情 况下，要正确处理 t\_listen 和 t\_accept 的非直观和易错的行为，需要编写一些微妙的应用级代码；而在 T LI Acceptor 类的 accept 方法中这些代码被封装起来。该方法接受来自客户的到来的连接请求。通过使用 C++缺省参数值，基于 TLI SAP 和基于 SOCK SAP 的应用的调用 accept 方法的标准方法在语法上是相同 的。

**1.3.1.3 SPIPE SAP**

SPIPE SAP 类属为高性能本地 IPC 提供 OO 包装。在 Win32 平台上，SPIPE SAP 类属在命名管道之 上实现。Win32 命名管道机制主要用于在同一机器的进程间高效地传输数据。对于本地 IPC，它通常比使 用 socket 更为高效[27]

在 UNIX 平台上，SPIPE SAP 类属通过已安装（mounted）的 STREAM 管道和 connld[28]来实现。S unOS 5.x 提供 fattach 系统调用，将管道句柄安装到 UNIX 文件系统中的指定位置。通过将 connld 流模块 推入管道已安装的一端，可以创建服务器应用。当与服务器运行在同一主机上的客户应用随后打开与已 安装的管道相关联的文件时，客户和服务器都将获得一个 I/O 句柄，标识一个唯一的、非多路服用和双向 的通信信道。

SPIPE SAP 继承层次是 SOCK SAP 及 TLI SAP 所用层次的“镜像“。它提供与 SOCK SAP ACE\_ LSOCK \*类（它们自己封装了 UNIX 域的 socket）相近似的功能。但是，在 SunOS 5.x 平台上，SPIPE SAP 比 ACE\_LSOCK\*接口更为灵活，因为它使得 STREAM 模块可以分别被“推入”和“弹出”SPIPE SAP 终点。SPIPE SAP 还支持在同一主机上执行的进程和/或线程间的字节流和按优先级排序的面向消息 的数据的双向传送[29]。

**1.3.1.4 FIFO SAP**

FIFO SAP 类属封装 UNIX 命名管道机制（也称为 FIFO）。不像 STREAM 管道，命名管道只提供从 一或多个发送者到单一接收者的单向数据信道。而且，来自不同发送者的消息都被放入同一个通信信道。 因而，在每一消息中必须明确地包含某种类型的多路分离标识符（demultiplexing identifier），以使接收 者能够确定是哪一个发送者发送的消息。

SunOS 5.x 中的基于 STREAM 的命名管道同时实现提供面向消息和面向字节流的数据传送机制。相 反，某些平台（比如 SunOS 4.x）只提供面向字节流的命名管道。因此，除非总是使用定长消息，在 Su

nOS 4.x 中通过命名管道发送的消息必须通过某种形式的字节计数、或是特别的结束符来加以区分，以使 接收者能从通信信道字节流中提取消息。为减少这样的局限，ACE FIFO SAP 实现包含有对 SunOS 5.x 的面向消息的机制进行模拟的逻辑。

**1.3.1.5** 其他通信机制

除了封装像 socket 和 TLI 这样的基于句柄的 I/O 通信机制，ACE 还提供了内存映射文件和系统 V U NIX IPC 机制的 OO 封装：

z 内存映射文件：ACE\_Mem\_Map 类提供 Win32 和 UNIX（比如 mmap 系统调用族）上可用的内存映 射文件机制的 OO 接口。这些调用利用底层的 OS 虚拟内存机制[30]来将文件映射到进程的地址空间。 映射文件的内容可直接通过指针访问。指针接口常常比通过标准 read/write I/O 系统调用间接地访问 数据块要更为方便和高效。此外，内存映射文件的内容可以很方便地在两个或多个进程间共享。

现有的 Win32 和 UNIX 的内存映射文件接口的风格有一点巴洛克（baroque，过分雕饰）。例如， 开发者必须手工完成许多“簿记”工作（比如显式地打开文件、确定它的长度、执行多种映射，等等）。 相反，ACE\_Mem\_Map OO 封装提供的接口采用了缺省值和多种有若干类型特征（type signature）变 体的构造器（例如，“从已打开的文件句柄映射”、“从文件名映射”，等等），以简化常见的内存 映射文件的使用模式。

例如，下面的程序使用 ACE\_Mem\_Map OO 封装来映射一个通过命令行指定的文件，并将它的 各行反向打印出来：

static void putline (const char \*s)

{

while (putchar (\*s++) != ’\n’)

continue;

}

int main (int argc, char \*argv[])

{

char \*filename = argv[1];

char \*file\_p;

Mem\_Map mmap (filename);

if (mmap (file\_p) != -1)

{

size\_t size = mmap.size () - 1;

if (file\_p[size] == ’\0’)

file\_p[size] = ’\n’;

while (--size >= 0)

if (file\_p[size] == ’\n’)

putline (file\_p + size + 1);

}

else

}

putline (file\_p);

return 0;

return 1;

把这个 OO 包装接口的使用与直接使用像 read 这样的 I/O 系统调用所需的远为冗长的 C 接口相比较， 很有启发意义。

z 系统 **V IPC** 机制：SunOS UNIX 提供了一套共享内存、同步和消息传递机制，即俗话所说的“系统 V IPC”[29]。这些机制所提供的大多数功能已包含在较新的 SunOS UNIX 机制中（比如，对应地，m map、线程同步[31]、和 STREAM 管道原语）。但是，特定类型的应用（比如数据库引擎）可能会从 系统 V IPC 机制中获益（比如消息队列的对等天性、高效的信号量多操作原子语义，以及系统 V IP C 机制在一系列 UNIX OS 平台上的广泛可用性）。然而，要正确地理解和使用系统 V IPC 机制（特 别是信号量）有一点挑战性，因为这些接口非常通用，而且直到最近，有关它们的行为特性的文档都 非常地少。

ACE 系统 V IPC 包装接口将开发者与无数多余的细节屏蔽开来。例如，对于利用标准 wait 和 si gnal 信号量的应用来说，ACE OO 包装版本的系统 V IPC 信号量使用起来更为直观和简单；如下面 的来自典型的生产者/消费者例子的代码片段所示：

typedef ACE\_SV\_Semaphore\_Simple SEMA; SEMA prod (1, SEMA::CREATE, 1);

SEMA cons (2, SEMA::CREATE, 0);

void producer (void)

{

for (;;)

{

prod.wait ();

// produce resource... cons.signal ();

}

}

void consumer (void)

{

for (;;)

{

cons.wait ();

// consume resource...

prod.signal ();

}

}

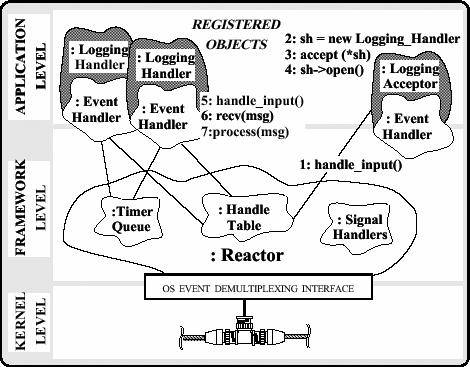
把这个简洁的OO包装接口与直接使用系统V信号量所需的远为冗长的C接口相比较，很有启发意义。

**1.3.2** 反应器（**Reactor**）：事件多路分离和事件处理器分派

通信软件对许多不同种类的事件（比如基于定时器、基于 I/O、基于信号和基于同步的事件）进行多 路分离和处理。例如，WWW 服务器通常在内部使用事件循环、监控一个周知的 Internet 端口（通常是端 口 80）。该端口与一个应用特有的处理器（handler）相关联，此处理器侦听客户在端口 80 上的连接。当 客户连接时，WWW 服务器接受连接，并创建一个事件处理器为此 HTTP 请求服务。例如，如果 Netscap e 浏览器发送一个 GET 请求，WWW 服务器将返回所请求的内容给浏览器。

为统一并自动化事件驱动的处理活动，ACE 提供了一个事件多路分离和事件处理器分派框架，称为 ACE\_Reactor[10]。Reactor 将 UNIX 和 Windows NT 事件多路分离机制（比如 select 和 poll）的功能封装 在一个可移植和扩展的 OO 包装中[10]。这些 OS 事件多路分离系统调用检测在一或更多 I/O 句柄上同时 发生的不同类型的输入和输出事件。

为改善应用的可移植性，不管使用了何种事件多路分离机制，ACE\_Reactor 都提供同样的接口2。另 外，ACE\_Reactor 还封装了在多线程事件处理环境中正确而高效地执行回调方式的分派所必需的互斥机 制。



2 ACE\_Reactor 的扩展版本，称为 ACE\_ReactorEx，用于 Win32 平台，以封装 WaitForMultipleObjects 事件多路分离调用。 因为该功能不能跨平台使用，在本文档中不做讨论。

图 1-5 ACE\_Reactor 类属组件

ACE\_Reactor 中的对象的结构在图 1-5 中演示。这些对象负责（1）事件（比如定时器驱动的呼出队 列生成的临时事件、通信端口上收到的 *I/O* 事件，以及信号事件）的多路分离和（2）分派预登记事件处 理器的适当方法，以处理这些事件。如图 1-5 所示，所有事件处理器对象都派生自 ACE\_Event\_Handler 抽象基类。该类指定一个被用于 ACE\_Reactor 接口，以针对特定事件的到达、分派特定的应用特有的方 法。

ACE\_Reactor 使用在 Event Handler 中声明的虚方法，以集成基于 I/O 句柄、基于定时器，和基于信 号的事件的多路分离。基于 I/O 句柄的事件经由 handle\_input、handle\_output 和 handle\_exceptions 方法分 派；基于定时器的事件经由 handle\_timeout 方法分派；而基于信号的事件经由 handle\_signal 分派。

ACE\_Event\_Handler 的子类（比如 1.4.1 描述的 Logging\_Handler）可以通过定义另外的方法和数据成 员来扩展基类的接口。此外，ACE\_Event\_Handler 接口中的虚方法可以有选择地被子类重载，以实现应用 特有的功能。例如，在 1.4.2 介绍的 PBX 监控服务器中，应用特有的子类定义了 Event Handler 对象，通 过继承和/或实例化 1.3.1 描述的 SOCK SAP 或 TLI SAP 传输接口类的对象，与客户进行通信。在 ACE\_ Event\_Handler 基类中的虚方法被子类定义后，应用可以实例化所得的事件处理器对象。

下面的例子实现一个简单的程序，它使用双向通信信道，在两个进程间持续地前后交换消息。此例 演示服务是怎样从 ACE\_Event\_Handler 继承的。它还描述 ACE\_Reactor 怎样被用于多路分离和分派基于 I/O、基于信号和基于定时器的事件。下面所示的 Ping\_Pong 类从 ACE\_Event\_Handler 继承接口，并实现 它自己的应用特有的功能：

class Ping\_Pong : public ACE\_Event\_Handler

{

public:

Ping\_Pong (char \*b)

: len (min (strlen (b) + 1, BUFSIZ))

{

strncpy (this->buf, b, BUFSIZ);

}

virtual int handle\_input (ACE\_HANDLE handle)

{

return read (handle, this->buf, BUFSIZ);

}

virtual int handle\_output (ACE\_HANDLE handle)

{

return write (handle, this->buf, this->len);

}

virtual int handle\_signal (int signum)

{

this->finished = 1;

}

virtual int handle\_timeout (const Time\_Value &, const void \*)

{

this->finished = 1;

}

bool done (void)

{

return this->finished == 1;

}

private:

sig\_atomic\_t finished; char buf[BUFSIZ]; size\_t len;

};

双向通信信道使用 SVR4 UNIX STREAM 管道创建：

static int init\_handles (ACE\_HANDLE handles[])

{

if (pipe (handles) == -1)

LM\_ERROR ((LOG\_ERROR, "%p\n%a", "pipe", 1));

// Enable message-oriented mode instead of

// bytestream mode. int arg = RMSGN;

if (ioctl (handles[0], I\_SRDOPT, arg) == -1

|| ioctl (handles[1], I\_SRDOPT, arg) == -1)

return -1;

}

主程序通过打开适当的通信信道开始运行。接着，程序派生一个子进程，在两个进程中各实例化一

个名为 callback 的 Ping\_Pong 事件处理器对象，为基于 I/O、基于信号，和基于定时器的事件而将 callbac k 对象登记到 ACE\_Reactor 的一个实例上，然后进入事件循环。如下所示：

int main (int argc, char \*argv[])

{

ACE\_HANDLE handles[2]; ACE\_Reactor reactor;

init\_handles (handles);

pid\_t pid = fork ();

Ping\_Pong callback (argv[1]);

// Register I/O-based event handler reactor.register\_handler (

handles[pid == 0],

&callback, ACE\_Event\_Handler::READ\_MASK

| ACE\_Event\_Handler::WRITE\_MASK);

// Register signal-based event handler reactor.register\_handler (SIGINT, &callback)

// Register timer-based event handler reactor.schedule\_timer (&callback, 0, 10)

/\* Main event loop (run in each process) \*/

while (callback.done () == false)

reactor.handle\_events ();

return 0;

}

基于定时器和基于信号的事件的 callback 事件处理器存储在 ACE\_Reactor 中的相应的表中。同样地，

当调用 register\_handler 方法登记基于 I/O 的事件处理器时，ACE\_Reactor 在一个内部表中存储适当的句柄。 当应用随后通过调用 ACE\_Reactor::handle\_events 方法执行它的主事件循环时，该句柄被作为参数传递给 底层的 OS I/O 多路分离系统调用（例如，select 或 poll）。

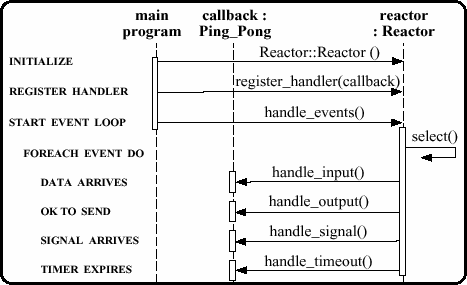


图 1-6 ACE Reactor 交互图

当与预登记的事件处理器 callback 对象相关联的输入、输出，和定时器事件在运行时发生时，ACE\_ Reactor 自动检测这些事件并分派适当的事件处理器对象的方法。被分派的 callback 对象的方法负责完成 应用特有的功能（比如写入消息到通信信道、从通信信道读取消息，或是设置触发程序终止的标志）。 这些组件间的协作通过对象交互图在图 1-6 中描述。

**1.3.3** 并发：多线程和同步机制

ACE 并发类属包含的 OO 包装（例如，ACE\_Mutex、ACE\_Condition、ACE\_Semaphore 和 ACE\_RW

\_Mutex）封装了相应的 Solaris[31]和 POSIX Pthreads[32]多线程和同步机制。这些包装使在类中作为成员 域出现的同步对象的初始化过程得以自动化，并且还简化了线程和同步机制的常见的使用模式。例如， 下面的代码演示了怎样将 SunOS mutex\_t 和 cond\_t 同步机制的 ACE 封装用于典型的共享资源管理类：

class Resource\_Manager

{

public:

Resource\_Manager (u\_int initial\_resources)

: resource\_add\_ (this->lock\_), resources\_ (initial\_resources) {}

int acquire\_resources (u\_int amount\_wanted)

{

this->lock\_.acquire ();

while (this->resources\_ < amount\_wanted)

{

this->waiting\_++;

// Block until resources are released. this->resource\_add\_.wait ();

}

this->resources\_ -= amount\_wanted;

this->lock\_.release ();

}

int release\_resources (u\_int amount\_released)

{

this->lock\_.acquire ();

this->resources\_ += amount\_released;

if (this->waiting\_ == 1)

{

this->waiting\_ = 0;

this->resource\_add\_.signal ()

}

else if (this->waiting\_ > 1)

{

this->waiting\_ = 0;

this->resource\_add\_.broadcast ();

}

}

// ...

this->lock\_.release ();

private:

ACE\_Mutex lock\_; ACE\_Condition<ACE\_Mutex> resource\_add\_; u\_int resources\_;

u\_int waiting\_;

// ...

};

注意 ACE\_Condition 对象 resource\_add 的构造器是怎样将 ACE\_Mutex 对象 lock\_与 Condition 对象绑 定在一起的。与底层的 SunOS cond\_t cond\_wait 接口相比较，这样的方式简化了 ACE\_Condition::wait 调 用接口。

尽管 ACE\_Mutex 包装为同步多线程控制提供了一种相对优雅的方法，它们仍是潜在地易错的，因为 开发者有可能会忘记调用 release 方法（或是由于程序员的疏忽，或是由于 C++异常的发生）。为改善应 用的健壮性，ACE 同步机制有效地利用了 C++类构造器和析构器语义。为确保 ACE\_Mutex 锁被自动获 取和释放，ACE 提供了名为 ACE\_Guard 的助手类，定义如下：

template <class MUTEX>

class ACE\_Guard

{

public:

ACE\_Guard (MUTEX &m): lock\_ (m)

{

this->lock\_.acquire ();

}

˜ACE\_Guard (void)

{

this->lock\_.release ();

}

private:

MUTEX &lock\_;

}

ACE\_Guard 类的对象定义一个代码块，在块开始时获取 ACE\_Mutex，而在块结束时自动释放。 注意 ACE\_Guard 类被定义为模板，由互斥机制参数化。有若干不同类型的互斥语义[33]。每种互斥

共有一个通用接口（也就是，acquire/release），但具有不同的序列化和性能属性。ACE 支持的两种互斥 是非递归和递归锁。

z 非递归锁：非递归锁提供互斥的一种高效的形式，它定义一个临界区，每一时刻只有单个线程可在其 中执行。它们之所以是非递归的，是因为当前拥有锁的线程在将其释放前不可以再次获取它。否则， 就会立即发生死锁。SunOS 5.x 通过它的 mutex\_t、rwlock\_t，和 sema\_t 类型（POSIX Pthreads 不提 供后两种同步机制）为非递归锁提供支持。ASX 框架提供 Mutex、RW\_Mutex，和 Semaphore 包装， 以分别封装这些语义。

z 递归锁：另外一方面，递归锁允许 acquire 方法嵌套调用，只要当前拥有该锁的线程就是试图重新获 取它的线程。递归锁对于回调驱动的事件分派框架（比如 1.3.2 描述的反应器）特别有用，在其中框 架的事件循环执行对预登记的用户定义的对象的回调。因为随后用户定义的对象可能经由它的方法入 口重入分派框架，必须使用递归锁以防止在回调过程中框架持有的锁发生死锁。

下面的 C++模板类为 Solaris 线程和 POSIX Pthreads（它们自己不提供递归锁语义）的同步语义实现 了递归锁语义：

template <class MUTEX>

class ACE\_Recursive\_Thread\_Mutex

{

public:

// Initialize a recursive mutex. ACE\_Recursive\_Thread\_Mutex (void);

// Implicitly release a recursive mutex.

˜ACE\_Recursive\_Thread\_Mutex (void);

// Acquire a recursive mutex. int acquire (void) const;

// Conditionally acquire a recursive mutex. int tryacquire (void) const;

// Releases a recursive mutex. int release (void) const;

private:

ACE\_Mutex nesting\_mutex\_; ACE\_Condition<ACE\_Mutex> mutex\_available\_; thread\_t owner\_id\_;

int nesting\_level\_;

};

注意这个类的接口与 ACE 中可用的其他锁定机制一致[22]。

下面的代码演示怎样将 ACE\_Guard 和 ACE\_Recursive\_Thread\_Mutex 用于回调机制中：

int Callback::dispatch (const Event\_Handler \*eh, Event \*event)

{

// Constructor acquires the lock on entry. ACE\_Guard<ACE\_Recursive\_Thread\_Mutex<ACE\_Mutex> > mon (this->lock\_);

eh->handle\_event (event);

// Destructor of Guard releases the lock on exit.

}

该代码确保 Event\_Handler 对象的登记作为临界区执行。该例子还演示了 C++习语的使用[34]，在其中，

当 ACE\_Guard 对象创建时，类的构造器自动在同步对象上获取锁。同样地，当 mon 对象出了作用域，类 的析构器自动解锁对象。而且，mon 的析构器将被自动调用以释放互斥锁，而不管 if/else 语句的哪一分 支从方法返回。此外，如果在 register\_handler 方法的处理过程中发生了 C++异常，锁也将会被自动释放。 ACE\_Recursive\_Thread\_Mutex 用于确保 dispatch 方法分派的应用特有的 handle\_event 回调不会导致死锁， 如果它们重入 Callback 对象的话。

ACE 还提供一个 ACE\_Thread\_Manager 类，其中包括一组用于管理相互协作、以实现集体行为的线 程组的机制。例如，ACE\_Thread\_Manager 类提供的机制（比如 suspend\_all 和 resume\_all）允许任意数目 的参与线程被原子地挂起和恢复。

**1.3.4** 服务配置器（**Service Configurator**）：显式动态链接机制

静态链接是这样一种技术：在编译时和/或静态链接时将所有对象文件绑定在一起，以组成完整的可 执行程序。相反，动态链接使得对象文件可以在初始调用程序或迟后在运行时的任意时刻被加入地址空 间，和/或被从中删除。SunOS 4.x 和 5.x 同时支持隐式的和显式的动态链接：

z 隐式动态链接被用于实现共享对象文件，也称为共享库[35]。共享对象文件可减少主要和辅助存储的 使用，因为只有一份共享对象的拷贝存在于内存中和磁盘上，而不管执行该代码的进程的数量是多少。 而且，特定的地址解析和重定位操作可以延迟至动态链接的函数被初次引用时。这样的“懒惰计算” 方案将进程启动时的链接编辑开销降到了最小。

z 显式动态链接提供的接口允许应用获取、利用，和/或移除在共享对象文件中定义的符号的运行时地 址绑定[36]。显式动态链接机制显著地增强了通信软件的功能和灵活性，因为服务可以在运行时插入 和/或删除，而不用终止和/或重启整个应用。SunOS 5.x 通过 dlopen/dlsym/dlcose 例程支持显式动态 链接，而 Win32 通过 LoadLibrary/GetProcAddress 调用支持这一特性。

ACE 提供 Service Configurator 类属来在一组类和继承层次中封装 SunOS 的显式链接机制。Service Configurator 有效地利用了其他 ACE 组件来扩展传统的看守配置和控制框架的功能[20]（比如 listen[3]、i netd[2]，和 Windws NT Service Control Manager[37]），这些功能为下列行为提供自动的支持：（1）并

发、多服务通信软件的静态和动态配置，（2）为 I/O 活动监控通信端口组，以及（3）分派在所监控端口 上接收到的消息给适当的应用特有的服务。这一部分的余下部分讨论 Service Configurator 类属的主要组 件。

**1.3.4.1 ACE\_Service\_Object** 继承层次

Service Configurator 中的主要配置单元是服务（service）。服务可以是简单的（比如返回一天中的当 前时间）或高度复杂的（比如 PBX 事件话务的分布式、实时路由器[6,38]）。为了给定义、配置和使用 通信软件提供一致的环境，所有的应用服务都派生自 ACE\_Service\_Object 继承层次（在图 1-7(1)中演示）。

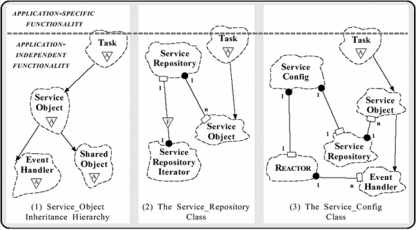


图 1-7 Service Configurator 类属的组件关系

ACE\_Service\_Object 类是一个有着继承关系的多层类型层次的焦点。在此类型层次中的抽象类提供 的标准接口可以有选择地被应用特有的子类实现，以访问特定的、应用无关的 Service Configurator 机制。 这些机制提供透明的动态链接、事件处理器登记、事件多路分离和服务分派。通过使处理器对象的应用 特有部分与底层的应用无关的 Service Configurator 机制去耦合，显著地减少了在运行中的应用中插入和 移除服务所必需的工作。

ACE\_Service\_Object 继承层次由 ACE\_Event\_Handler 和 ACE\_Shared\_Object 抽象基类组成。ACE\_Ev ent\_handler 类已在上面的 1.3.2 中描述。其他类的行为在下面概述。

z **ACE\_Shared\_Object** 抽象基类：该基类指定用于将服务处理器对象动态链接进应用的地址空间的接 口。ACE\_Shared\_Objecti 抽象基类输出三个抽象方法：init、fini，和 info。这些方法在 Service Confi gurator 提供的应用无关的可复用组件和利用这些组件的应用特有功能之间建立了协定。通过使用抽象 方法，Service Configurator 确保服务处理器实现遵从它提供特定的配置相关信息的义务。这些信息随 后被 Service Configurator 用于在运行时自动链接、初始化、标识服务，以及解除服务的链接。

ACE\_Shared\_Object 基类独立于 ACE\_Event\_Handler 类而定义，以清晰地区分它们的两组不相关 的事务。例如，某些应用（比如编译器或文本编辑器）可能会从动态链接中获益，尽管它们可能不需 要通信端口事件多路分离。相反，其他应用（比如 ftp 服务器）可能需要事件多路分离，但可能不需 要动态链接。通过将这些接口分离成两个基类，应用就能够选择服务配置器的一个子集，而不会带来 不必要的存储开销。

z **ACE\_Service\_Object** 抽象派生类：通常，安装和管理复杂的分布式系统需要有动态链接、事件多路 分离和服务分派的支持，以使应用服务动态配置和重配置自动化。因而，Service Configurator 定义了 ACE\_Service\_Object 类，将 ACE\_Event\_Handler 和 ACE\_Shared\_Object 抽象基类的接口结合在一起。 所得到的抽象派生类提供的接口可被开发者用作实现服务、并将其配置进服务配置器的基础。

在开发过程中，ACE\_Service\_Object 的应用特有的子类必须实现由 ACE\_Service\_Object 类接口指 定的 suspend 和 resume 抽象方法。这些方法被 Service Configurator 自动调用，以响应外部的事件。 应用开发者可以控制对象为挂起服务所采取的动作，而无须完全移除它或解除它的链接；唤醒先前挂 起的服务也是一样。

此外，应用特有子类还必须实现 ACE\_Service\_Object 子类所继承（不是定义）的四个抽象方法

（init、fini、info，和 get\_handle）。init 方法用作服务处理器在运行时初始化过程中的入口。当 ACE

\_Service\_Object 的实例被动态链接时，该方法负责执行应用特有的初始化。同样地，当 ACE\_Service

\_Object 在运行时被解除链接并从应用中移除时，Service Configurator 自动调用 fini 方法。该方法通 常执行终止操作，由其释放动态分配的资源（比如内存、I/O 句柄或同步锁）。info 方法格式化用户 可读的一个字符串，简洁地报告服务访问信息及为服务功能建立文档。客户可以查询应用以获取此信 息，并将它用于与应用中运行的某个特定的服务联系。最后，get\_handle 方法被 Reactor 用于从服务 处理器对象中提取底层的 I/O 句柄。此 I/O 句柄标识一个可被用于从客户那里接受连接或接收数据的 传输端点。

z 应用特有的具体派生子类：Service Object 是一个抽象类，因为它的接口包含有继承自 Event Handler 和 Shared Object 抽象基类的抽象方法。因而，开发者必须提供具体的子类（1）定义上面描述的六个 抽象方法，以及（2）实现必需的应用特有的功能。为完成后一任务，子类通常定义由 Service Objec t 接口输出的某些特定的虚方法。例如，常常实现 handle\_input 方法，用以从客户那里接受连接或数 据。

图 1-7（1）中描述的 ACE\_Acceptor 类是应用无关的子类的一个例子；它是分布式日志工具的一 部分，接受连接请求。该类将在 1.4.1 介绍的例子中作进一步描述。

**1.3.4.2 ACE\_Service\_Repository** 类

Service Configurator 类属同时支持单服务和多服务通信软件的配置。因而，为简化运行时管理，常 常有必要个别和/或共同地控制及协调由应用的正在活动的服务组成的各个 ACE\_Service\_Object。ACE\_S ervice\_Repository 是用于协调本地和远地查询的对象管理器，这些查询涉及由基于 Service Configurator 的应用提供的服务。在对象管理器中的搜索结构将服务名（以 ASCII 字符串表示）与 ACE\_Service\_Obje ct（以对象代码表示）绑定在一起。一个服务名唯一地标识在仓库中（repository）存储的一个 ACE\_Serv ice\_Object 的实例。

ACE\_Service\_Repository 中的每一条目都含有一个指向应用特有派生类的 ACE\_Service\_Object 部分 的指针（如图 1-7(2)所示）。这使得 Service Configurator 能够静态或动态地对应用进行 ACE\_Service\_Ob ject 的装载、启用、挂起、恢复，或卸载。对于动态链接的 ACE\_Service\_Object，仓库还维护了底层共享

对象文件的句柄。当 ACE\_Service\_Object 所提供的服务不再被需要时，这个句柄用于从运行中的应用解 除其链接并卸载之。

与 ACE\_Service\_Repository 一起，还提供了一个迭代器（iterator）类。该类用于访问仓库中的每一个

ACE\_Service\_Object，而不会不恰当地危及数据封装。

**1.3.4.3 ACE\_Service\_Config** 类

ACE\_Service\_Config 类是 Service Configurator 框架中的统摄组件。如图 1-7(3)所示，该类集成了其 他的 Service Configurator 框架组件（比如 ACE\_Service\_Repository 和 ACE\_Reactor），以使并发、多服 务通信软件的静态和/或动态配置自动化。

ACE\_Service\_Config 类使用配置文件（称为 svc.conf）来指导它的配置和重配置活动。每一应用都可 与一个独立的 svc.conf 配置文件相关联。同样地，一组应用也可以被单个的 svc.conf 文件描述。图 1-8 使 用扩展的 Backus/Naur 格式（EBNF）描述了 svc.conf 文件中的主要语法成分。文件中的每个服务配置条 目以服务配置指令起头，它指定要执行的配置活动。表 1-1 总结了有效的服务配置指令。

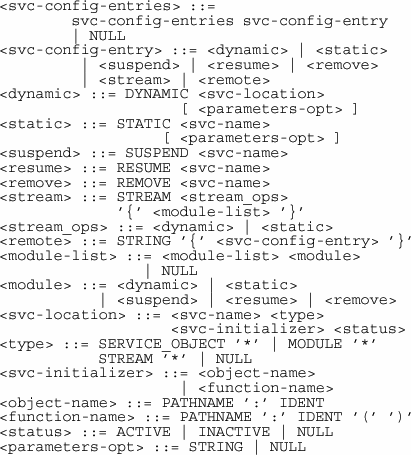


图 1-8 服务配置条目的 EBNF 格式 指令 描述

|  |  |
| --- | --- |
| dynamic | 动态链接和启用服务 |
| static | 启用静态链接的服务 |
| remove | 完全地移除服务 |
| suspend | 挂起服务，而不移除它 |
| resume | 恢复先前挂起的服务 |

表 1-1 服务配置指令

对于每个动态链接的服务，相应的服务配置条目含有指示共享对象文件位置的属性，以及需用于在 运行时初始化服务的参数。通过将服务属性和初始化参数统一进单一配置文件，显著地简化了应用中服 务的安装和管理。svc.conf 文件有助于使应用的结构与它的服务的行为去耦合。基于在 svc.conf 文件中指 定的应用特有的属性和参数，这样的去耦合也允许对框架提供的机制进行“懒惰的”配置和重配置。

图 1-9 描述的状态转移图演示了 Service Configurator 类属中的一些方法，这些方法被调用，以响应 在服务配置、执行和重配置的过程中发生的事件。例如，当 CONFIGURE 和 RECONFIGURE 事件发生时， ACE\_Service\_Config 类的 process\_directives 方法被调用，以查询 svc.conf 文件。在应用的新实例被初始配 置时这个文件第一次被查询。无论何时接收到预先指定的外部事件（比如 UNIX SIGHUP 信号或来自 so cket 的通知）、触发了应用的重配置，这个文件都会被再次查询。

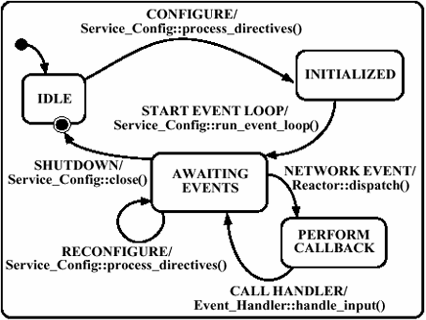


图 1-9 服务配置、执行和重配置的状态转移图

**1.3.5** 流：分层服务的集成

流类属是自适配通信环境的主要焦点。该类属含有自适配服务执行体（ASX）框架，它集成了较低 级的 OO 包装组件（像 IPC SAP）和较高级的类属（像 Reactor 和 Service Configurator）。ASX 框架有

助于简化分层地集成在一起的通信软件的开发，特别是用户级通信协议栈和网络服务器。ASX 框架设计 用于改善应用特有的服务、以及这些服务所依赖的底层 OS 并发、IPC、显式动态链接和多路分离机制的 模块性、可扩展性、复用性和可移植性。

ASX 框架为通信软件的开发者提供下面两种好处：

1. 它嵌入、封装，并实现了通常用于开发通信软件的一些关键设计模式。通过确定大型系统开发中的基 本挑战，设计模式有助于提高软件的质量。这些挑战包括开发者之间的体系结构知识的交流；适应新 的设计范式或体系样式；解决非功能性的压力，比如复用性、可移植性和可扩展性；以及避开那些通 常只能通过经验来学习的开发陷阱和缺陷。

2. 它严格地区分了关键的开发事务。ACE 将通信软件的开发划分为两种不同的范畴：（1）不依赖于应 用的事务，它们对于大多数或所有的通信软件来说都是通用的（比如端口监控；消息缓冲、排队和多 路分理；服务分派；本地/远地进程间通信；并发控制；以及应用配置、安装和运行时服务管理）， 以及（2）应用特有的事务，它们依赖于特定的应用。通过复用 ACE 提供的 OO 包装和框架，开发者 从花费时间重新发明一些经常重复出现的任务的方案中被解放出来。于是，这使得他们可以专注于那 些组成特定应用的、关键的更高级功能需求和设计事务。

**1.3.5.1** 主要的 **ASX** 特性

通过使应用特有的处理策略与下列配置相关的开发活动和机制去耦合，ASX 框架增强了通信软件的 灵活性：

z 与每个应用进程相关联的服务的类型和数目：ASX 框架允许应用将一或多个服务合并进单一的管理 单元。这样的配置通信软件的多服务方法有助于（1）通过自动完成通用的服务初始化活动，简化开 发并复用代码，（2）通过“按需”生成服务处理器，降低 OS 资源的消耗（比如进程表槽），（3） 无须修改已有的源代码或终止正在执行的分派进程（比如 inetd 超级服务器）就能够更新应用服务， 以及（4）通过一组一致的配置管理操作，统一网络服务的管理。

z 服务被配置进应用的时间点：ASX 框架有效地对 Service Configurator 框架（在 1.3.4 中描述）进行利 用，以提供一个可扩展的、面向对象的接口，使显式动态链接的 OS 机制的使用得以自动化。通过允 许内部服务在应用开始执行时或运行时被配置，动态链接增强了通信软件的可扩展性。该特性使得应 用的服务可被动态配置，而无需修改、重编译、重链接，或重启活动的服务。在 ASX 框架中，选择 静态或动态配置可针对每个服务单独进行。而且，选择可延迟至应用开始执行的时候。

z 执行代理的类型：在 ASX 框架中，服务可在运行时通过若干不同类型的进程和线程执行代理来执行。 通过使服务功能与用于调用服务的执行代理去耦合，ASX 框架扩展了开发者可用的应用并发配置方 式的选择范围。

高效的应用并发配置常常依赖于特定的服务需求和平台特性。例如，基于进程的配置对于实现长 持续时间的服务（比如 Internet ftp 和 telnet）来说也许是适当的，这样的配置使它们的安全机制基于 进程所有权。在这种情况下，每个服务（或服务的每个活动实例）可被映射到不同的进程，并在多处 理器平台上并行执行。但是，在其他环境中采用另外的配置可能更为合适。例如，在分离的线程中实 现相互协作的服务（比如在分布式数据库引擎的终端系统中的服务）常常更加简单和高效，因为它们 经常性地访问公用数据结构。在该方法中，每个服务可以在同一进程的不同线程中执行，以降低调度、 上下文切换和同步的开销[6]。

z 层次相关的服务被结合进应用的顺序：复杂服务可使用一系列互连的独立服务对象组成，它们通过传 递消息进行通信。这些对象可以通过本质上任意的配置结合在一起，以满足应用要求和增强组件复用。

z 基于 **I/O** 句柄和基于定时器事件的多路分离机制：这些机制用于将到来的连接请求和数据分派给预登 记的应用特有的处理器。通过一个可扩展的和类型安全的面向对象接口，ASX 框架使用 Reactor 类属 来集成基于 I/O 句柄、基于定时器，和基于信号的事件的多路分离。

z 底层 **IPC** 机制：应用服务可以使用 1.3.1 描述的 IPC SAP 机制来与本地或远地终端系统上的通信实体 交换数据。不像弱类型的、“基于句柄”的 socket 和 TLI 接口，IPC SAP 包装使得应用能够通过类 型安全的、可移植的接口访问底层 OS IPC 机制。

ASX 框架合并了来自若干模块化的通信框架的概念，其中包括系统 V STREAMS[39]、x-kernel[40] 和来自面向对象操作系统 Choices 的 Conduit 框架（对这些和其他通信框架的考察见[42]）。这些框架全 都支持通过将“积木块”协议和服务组件互连来灵活地配置通信子系统。总之，通过使处理功能与周边 框架的基础构造去耦合，这些框架鼓励了标准的与通信相关的组件的开发（比如消息管理器、基于定时 器的事件分派器、多路分离器[40]和各种协议功能[43]）。如下所述，ASX 框架包含的一些额外特性可进 一步使处理功能与底层的处理体系结构去耦合。3

不像 STREAMS，被配置进 ASX 框架的应用服务在用户空间、而不是内核空间中执行。在用户空间、 而不是 OS 内核中开发通用通信软件有着若干优点：

z 对一般 **OS** 特性的访问：在用户空间中运行的应用可以访问全面的 OS 机制（比如动态链接、内存映 射文件、多线程、大虚拟地址空间、进程间通信机制、文件系统和数据库）。相反，驻留内核的组件 常常局限于一组有限的内核特有的机制。尽管有一些习语可以克服其中的一些局限（例如，维护一个 用户级看守、代表驻留内存的组件完成文件 I/O），这些工作方法往往不那么优雅和可移植。

z 增强的开发环境：较高级编程工具（比如符号调试器）可用于开发用户空间中的应用。相反，由于在 内核编程环境中，调试工具的原始和微妙的时序干扰，在 OS 内核中开发网络服务是一项复杂而艰巨 的任务[44]。在这样受限的环境中期待开发者有效地编程是很冒险的。

z 增强的系统健壮性：在用户级进程或线程中产生的异常情况（比如废弃的 NULL 指针、除 0 错误， 等等）只会影响出错进程或线程。但是，OS 内核中的异常情况可能导致整个操作系统的“恐慌”和 崩溃。而且，在每次崩溃后重启 OS 很快会变得冗长乏味。

z 可移植性：在不同 OS 平台间（即使是同一 OS 平台的不同变种）移植内核级驱动程序通常比移植用 户级应用组件要承担更多的复杂性。SunOS 5.x DDI/DKI API 意图减少 UNIX 平台上的一些移植性 问题，但那并未解决在其他平台上（比如 OS/2、Windows NT、VMS 和 Novell Netware）的应用的 移植性问题。

在 OS 内核中实现分布式服务的主要原因是提高性能。例如，一个通信协议栈的驻留内核的实现常常 有助于减少调度、上下文切换和跨越保护域边界的开销，并且因为使用了“线性的”（而非分页的）内存， 还可以表现出更可预测的响应时间。但是，对于许多通信软件应用来说，用户空间开发所带来的更多的 灵活性、简单性、健壮性和可移植性是关键性的需求，可以弥补潜在的性能下降。

**1.3.5.2 Stream** 类属组件

3 处理体系结构表示一或多个 CPU 与在通信系统中实现服务的应用任务和消息之间的“绑定”[6]。

Stream 类属中的组件负责协调一或多个 ACE\_Stream 对象的配置和在运行时的执行。 ACE\_Stream 是用户应用与之协作的对象，用以配置和执行 ASX 框架中的应用特有的服务。如图 1-10 所示，ACE\_Str eam 包含有一系列互连的 ACE\_Module 对象，它们可被（1）开发者在安装时或（2）应用在运行时链接 在一起。ACE\_Module 对象用于将应用的体系结构分解成一系列互连的、功能独立的层次。通常每一层 实现一簇相关的服务功能（比如端对端传输服务或表示层格式化服务）。每个 ACE\_Module 包含的一对 A CE\_Task 对象将该层划分为读端和写端处理功能。

OO 语言特性（比如类、继承、动态绑定和参数化类型）使开发者无须修改不依赖于应用的 ASX 框 架组件，就能够将应用特有的功能合并进流中。例如，将服务层增加进流涉及（1）从缺省 ACE\_Task 接 口继承并有选择地在子类中覆盖若干虚函数，以提供应用特有的功能，（2）分配一个新的 ACE\_Module， 其中包含有应用特有的 ACE\_Task 子类的两个实例（一个用于读端，一个用于写端），以及（3）将 ACE\_ Module 插入到流中。经由消息传递接口，邻近的相互连接的 ACE\_Task 中的服务通过交换类型化的消息 进行协作。

为避免再次发明习惯用语，流类属中的许多类名与系统 V STREAMS 框架中可用的类似组件相对应 [39]。但是，在这两种框架中，用于支持可扩展性和并发的技术有着显著的差异。例如，为 ASX 流类增 加应用特有的功能是通过从已有的框架组件继承若干定义良好的接口和实现来进行的。比起系统 V STR EAMS 中所用的函数指针技术，使用继承来增加新功能提供了强得多的类型安全性。ASX 流类还采用了 不同的并发控制方案来减少死锁的可能性，并简化流中的 Task 之间的流控制。ASX 流类对系统 V STRE AMS 中所用的基于协同程序的、“无重量的”4服务处理机制[45]进行了完全的重新设计和实现。这些 AS X 的变动意图在共享内存的多处理器平台上更为有效地利用多 PE。

4 无重量进程在其上执行的运行时栈也被其他进程所用，这使得编程变得复杂化，并且增加了死锁的潜在可能性。例如， 无重量进程不能挂起执行来等待资源变得可用或等待事件发生。

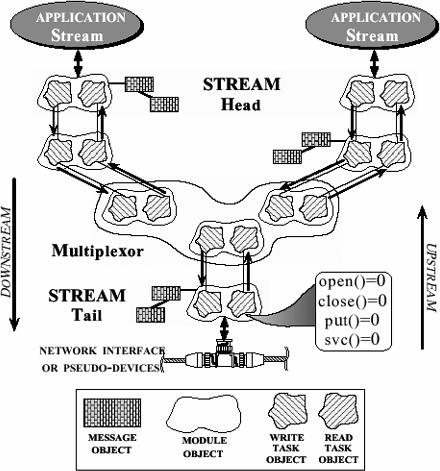


图 1-10 ASX 框架中的组件

余下部分详细讨论流类属的主要组件（例如，ACE\_Stream 类、ACE\_Module 类、ACE\_Task 类）：

z **ACE\_Stream** 类：ACE\_Stream 类定义流的应用接口。ACE\_Stream 对象包含有一个由一或多个层次相 关的服务组成的栈，为应用提供双向的 get/put 风格的接口，用以通过组成特定流的互连 Module 发送 和接收数据及控制消息。ACE\_Stream 还实现了一个 push/pop 风格的接口，允许应用通过插入和移除 下面描述的 ACE\_Module 类的对象、在运行时对流进行配置。

z **ACE\_Module** 类：ACE\_Module 类定义独立的、应用特有的功能层。流通过互连一系列 ACE\_Modul e 对象构成。流中的 ACE\_Module 对象有着松散的耦合，并通过传递类型化的消息来与邻近的 ACE\_ Module 对象协作。每个 ACE\_Module 对象包含有一对指针，指向应用特有的 ACE\_Task 类的子类的 对象。ACE\_Task 类在下面简略描述。

如图 1-10 所示，当流被打开时，两个缺省的 ACE\_Module 对象（ACE\_Stream\_Head 和 ACE\_Str eam\_Tail）自动被安装。这两个 ACE\_Module 解释预定义的 ASX 框架控制消息和数据消息，这些消 息在运行时在流中流动。ACE\_Stream\_Head 类在应用和流之间提供消息缓冲接口。ACE\_Stream\_Tail 类通常将来自网络或伪设备的消息转换为规范的内部消息格式，可被流中更高层的组件进一步处理。 同样地，对于外发的消息，它将其从内部格式转换为网络消息。

z **ACE\_Task** 抽象类：ACE\_Task 类是 ACE 中用于创建用户定义的、处理应用消息的主动对象（Active

Object）[11]和被动对象（Passive Object）的中心机制。ACE 的任务可以进行下面的活动：

 可被动态链接

 可用作 I/O 操作的多路分离端点

 可与多个线程控制相关联（也就是，变成所谓的“主动对象”）

 可将消息存储在队列中，以用于后续处理

 可执行用户定义的服务

ACE\_Task 抽象类定义的接口被派生类继承和实现，以提供应用特有的功能。它之所以是抽象类，是 因为它的接口定义了下面描述的抽象方法（open、colse、put 和 svc）。通过使 ACE\_Stream 类属提供 的不依赖于应用的组件与继承并使用这些组件的、应用特有的子类去耦合，把 ACE\_Task 定义为抽 象类增强了复用。同样地，使用抽象方法允许编译器确保 Task 的子类遵从自己提供下面的功能的义 务：

 初始化和终止方法：派生自 ACE\_Task 的子类必须实现 open 和 close 方法，执行应用特有的 AC E\_Task 初始化和终止活动。这些活动通常分配和释放像连接控制块、I/O 句柄，及同步锁这样的 资源。

ACE\_Task 可与 ACE\_Module 一起或分开定义和使用。当与 ACE\_Module 一起使用时，它 们被成对地存储：一个 ACE\_Task 子类操作读端的、自下而上发送给它的 ACE\_Module 层的消 息的处理；另一个操作写端的、自上而下发送给它的 ACE\_Module 层的消息的处理。当 ACE\_ Module 在流中插入和移除时，Module 的写端和读端的 ACE\_Task 子类的 open 和 close 方法分 别被 ASX 框架自动调用。

 应用特有的处理方法：除了 open 和 close，ACE\_Task 的子类还必须定义 put 和 svc 方法。这些 方法在消息上执行应用特有的处理功能。例如，当消息到达流的头或尾时，作为调用流中的每 一 ACE\_Task 的 put 和/或 svc 方法的结果，它们被“护送”着通过一系列互连的 ACE\_Task。

当在流中某一层的 ACE\_Task 传递消息给另一层中相邻的 ACE\_Task 时，put 方法被调用。 put 方法相对于它的调用者同步地运行，也就是，它从最初调用其 put 方法的 Task 那里借用线 程控制。该线程控制通常源自“上游”应用进程，“下游”处理 I/O 设备中断的进程池[40]，或 流内部的事件分派机制（比如面向连接的传输协议 ACE\_Module 中的用于触发重发的定时器驱 动呼出队列）。

如果 ACE\_Task 作为被动对象执行（也就是，它总是从调用者那里借用线程控制），ACE\_ Task::put 方法就是 ACE\_Task 的入口，并用作 ACE\_Task 在其中执行的上下文。相反，如果 AC E\_Task 作为主动对象执行，ACE\_Task::svc 方法就用于相对于其他 ACE\_Task 异步地执行应用 特有的处理。不像 put，svc 方法不会被相邻的 ACE\_Task 直接调用，而是被与它的 ACE\_Task 相关联的一个单独的线程调用。该线程为 ACE\_Task 的 svc 方法提供了执行上下文和线程控制。 svc 方法运行一个事件循环，持续地等待消息到达 ACE\_Task 的 ACE\_Message\_Queue（见下面 的条目）。

在 put 或 svc 方法的实现中，消息可经由 ACE\_Task 的 put\_next 方法、被转发给流中相邻的 ACE\_Task.。put\_next 调用在相邻层中驻留的下一个 ACE\_Task 的 put 方法。这个对 put 的调用 可以借用调用者的线程控制，并立即处理消息（也就是，图 1-11(1)中所示的同步处理方法）。 相反，put 方法可以将消息入队，并推迟给在单独的线程控制中执行的 svc 方法处理（也就是， 图 1-11(2)中所示的异步处理方法）。如在[6]中所讨论的，选择特定的处理方法对性能和编程的 容易度有着显著的影响。

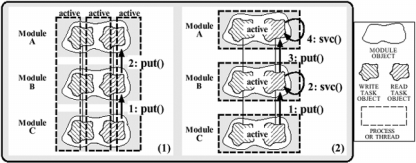


图 1-11 调用 put 和 svc 方法的可选方法

 消息排队机制：除了 open、close、put 和 svc 抽象方法接口，每个 ACE\_Task 还具有一个 ACE\_ Message\_Queue。ACE\_Message\_Queue 是 ACE 中的一种标准组件，用于在 ACE\_Task 之间传递 信息。而且，当 ACE\_Task 作为主动对象执行时，它的 ACE\_Message\_Queue 用于为 svc 方法中 的后续处理缓冲一系列数据消息和控制消息。在消息到达时，svc 方法使消息出队，并执行 AC E\_Task 子类的应用特有的处理任务。

有两种消息可出现在 ACE\_Message\_Queue 中：简单的和复合的。简单消息包含单个 ACE\_ Message\_Block，而复合消息包含多个链接在一起的 ACE\_Message\_Block。复合消息通常由一个 控制块和一或多个跟随的数据块组成。控制块包含有“簿记”信息（比如目的地址和长度域）， 而数据块则包含消息的实际内容。通过传递消息指针而不是拷贝数据，使得在 Task 间传递消息 的开销降到了最低。

ACE\_Message\_Queue 含有一对高低水位标变量，用于在流中相邻的 ACE\_Module 之间实现 层到层的流控制。高水位标指示 ACE\_Message\_Queue 在它进行流控制前所想要缓存的消息的 字节数。低水位标指示表明先前已进行流控制的 ACE\_Message\_Queue 不再被认为是“充满” 时的“水位”。

**1.4 ACE** 实例

ACE 组件目前正在若干研究[46]和商业环境[6, 38, 47]中被用于增强通信软件的配置灵活性、以及可 在多种硬件和软件平台上高效而可移植地运行的通信软件的组件的复用。为演示 ASX 框架是怎样被用于 实践的，这一部分考查目前正在使用 ACE 组件进行开发的两种商业应用的体系结构：分布式日志工具和 用于电信交换设备的分布式监控系统。

**1.4.1** 分布式日志实例

调试分布式软件常常极具挑战性，因为诊断输出出现在不同窗口中和/或不同的远地主机系统上。因

而，ACE 提供了一个分布式日志工具，以简化调试和运行时跟踪。该工具目前被用于一个商业在线事务 处理系统[14]中，为高速网络环境中的群集工作站和多处理器数据库服务器提供日志服务。

如图 1-12 所示，分布式日志工具允许运行在多个客户主机上的应用发送日志记录给运行在指定服务 器主机上的服务器日志看守。这一部分聚焦于日志工具的服务器看守部分的体系结构和配置，它们基于 A SX 框架提供的 Service Configurator 和 ACE\_Reactor 类属来实现。分布式日志工具的完整设计和实现在[1

0]中描述。

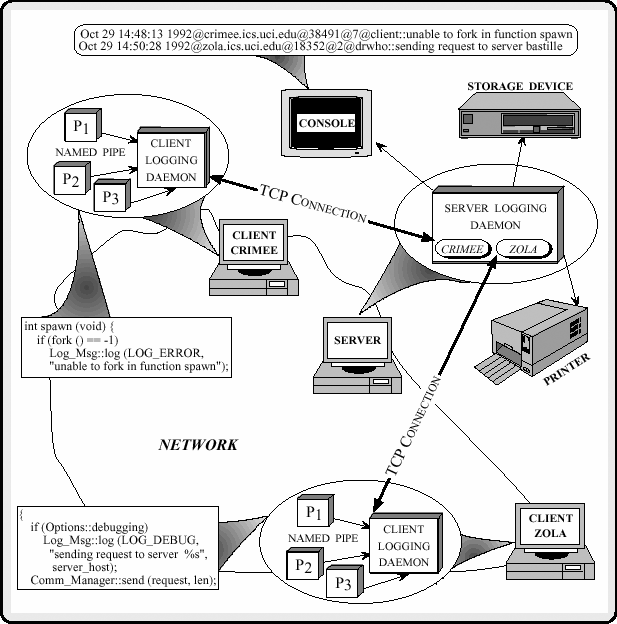


图 1-12 分布式日志工具

**1.4.1.1** 服务器日志看守组件

服务器日志看守是一个并发的多服务看守，它同时处理接收自一或多个客户主机的日志记录。服务 器日志看守的面向对象的设计被分解为若干模块化组件（在图 1-13 中显示）,由它们执行一些良好定义的

任务。应用特有的组件（Logging\_Acceptor 和 Logging Handler）负责处理接收自客户的日志记录。面向 连接的组件（Acceptor 和 Client\_Handler）负责接受来自客户的连接请求和数据。最后，不依赖于应用的 ASX 框架组件（ACE\_Reactor、Service Condigurator 和 IPC SAP 类属）负责完成 IPC、显式动态链接、 事件多路分离、服务分派和并发控制。

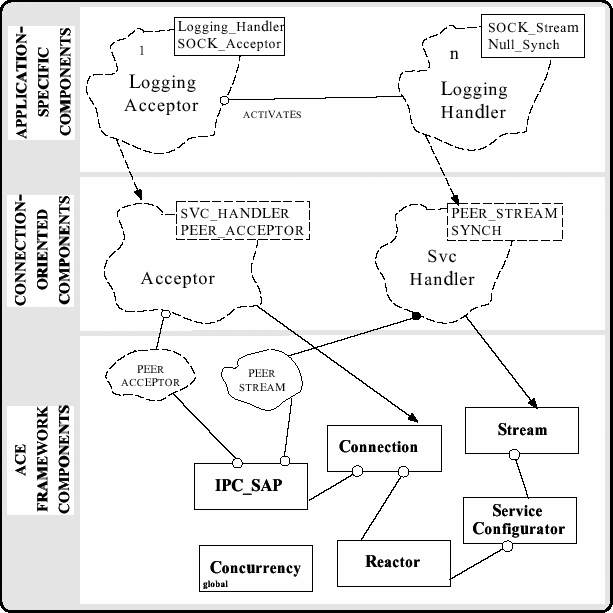


图 1-13 服务器日志看守中的类组件

Logging\_Handler 子类是一种参数化类型，负责处理从客户主机发往服务器日志看守的日志记录。它 的通信机制可以通过 SOCK SAP 或 TLI SAP 包装来实例化如下：

class Logging\_Handler : public Client\_Handler <

#if defined (MT\_SAFE\_SOCKETS) ACE\_SOCK\_Stream,

#else

ACE\_TLI\_Stream,

#endif /\* MT\_SAFE\_SOCKETS \*/

ACE\_INET\_Addr>

{

/\* ... \*/

};

Logging\_Handler 类继承自 Event\_Handler（间接经由 Client\_Handler），而不是 Service Object，因为它不

是被动态链接进服务器日志看守的。

当来自与特定的 Logging\_Handler 相关联的客户主机的日志记录到达时，ACE\_Reactor 自动分派对 象的 handle\_input 方法。该方法格式化记录，并将其显示在一或多个输出设备上（比如图 1-12 中所示的 打印机、持久存储和/或控制台设备）。

Logging\_Acceptor 子类也是一个参数化类型，它负责接受来自参与日志服务的客户主机的连接请求：

class Logging\_Acceptor :

public Client\_Acceptor<Logging\_Handler,

#if defined (MT\_SAFE\_SOCKETS) ACE\_SOCK\_Acceptor,

#else

ACE\_TLI\_Acceptor,

#endif /\* MT\_SAFE\_SOCKETS \*/ ACE\_INET\_Addr>

{

/\* ... \*/

};

因为 Logging\_Acceptor 继承自 ACE\_Service\_Object(间接经由它的 ACE\_Acceptor 基类)，它可以通过服务

器日志看守的 svc.conf 配置文件来在运行时动态地链接进服务器日志看守、并进行相关的操作。同样地， 因为 Logging\_Acceptor 间接继承自 ACE\_Event\_Handler 接口，当来自客户的连接请求到达时，它的 handl e\_input 方法将会被 ACE\_Reactor 自动调用。当连接请求到达时，Logging\_Acceptor 子类分配一个 Loggin g\_Handler 对象，并将这个对象登记到 ACE\_Reactor。

通过使连接建立功能和日志记录接收分离成图 1-13 所示的两种不同的类层次，显著地增强了分布式 日志工具的模块性、可复用性和可配置性。这样的分离允许 ACE\_Acceptor 类被复用于其他类型的面向连 接服务。特别地，为提供完全不同的处理功能，只需要重新实现服务的 ACE\_Client\_Handler 部分的行为。 而且，参数化类型的使用减轻了对某种特定类型的 IPC 机制的依赖。

**1.4.1.2** 服务器日志看守配置

ASX 框架使用 Service Configurator 来将日志服务动态或静态配置进服务器日志看守。动态配置的服 务可在运行时插入、修改或移除，从而改善了服务的灵活性和可扩展性。下面的 svc.conf 文件条目用于 将日志服务动态地配置进服务器日志看守：

dynamic Logger Service\_Object \*

./Logger.so:\_alloc() “-p 7001”

<svc-name>符号 Logger 指定服务名，用于安装和运行时在 ACE\_Service\_Repository 中标识相应的 Service

Object。Service Object \* 是位于共享对象文件中的\_alloc 方法的返回类型，共享对象文件的路径由路径 名./Logger.so 指定。服务配置器框架定位此共享对象文件，并将其动态链接进日志看守的地址空间。服务 路径还指定派生自 Service Object 的应用特有对象的名字。在此例中，\_alloc 函数被用于动态分配新的 L

ogging\_Acceptor 对象。该行剩下的内容（”-p 7001）”

表示一组应用特有的配置参数。这些参数作为 argc/

argv 风格的命令行参数传递给服务的 init 方法。Logging\_Acceptor 类的 init 方法将”-p 7001”解释为端口号， 服务器日志看守将在其上侦听客户的连接请求。

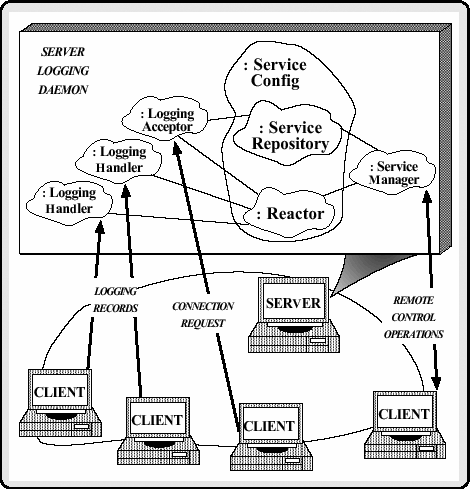


图 1-14 分布式日志工具中的 ACE 组件

当看守一开始执行时，静态配置的服务总是可用的。例如，Service Manager 是标准的 Service Confi gurator 框架组件，客户可用以获取活动看守服务的列表。下面的 svc.conf 文件中的条目用于在初始化过 程中将 Service Manager 服务静态地配置进服务器日志看守中：

static ACE\_Svc\_Manager “-p 911”

为使 static 指令工作，实现 ACE\_Svc\_Manager 服务的对象代码必须与主看守驱动可执行程序静态地 链接在一起。此外，ACE\_Svc\_Manager 对象必须在动态配置发生前插入 Service Repository（ACE\_Servi

ce\_Config 构造器自动完成此工作）。由于这些限制，如果不首先从 Service Repository 中将其移除，静态 配置的服务就不能在运行时重配置。

服务器日志看守的主驱动程序通过下面的代码实现：

int main (int argc, char \*argv[])

{

ACE\_Service\_Config loggerd;

// Configure server logging daemon. if (loggerd.open (argc, argv) == -1 )

return –1;

// Perform logging service. loggerd.run\_reactor\_event\_loop();

return 0;

}

CONFIGURE

Llgger A: L: :Service :Reactor : Service

Daemon Loggi ng Acceptor Llgging\_Handler Con fig Repositor y

' ' ' ' I

I O[!en(} I I 10 1 I

I I I Iprocess directives() I

FOREACtl SVC ENTRY DO I I I

I I I I

I I I load service()

LOAD SERVICE I I I

REGISTER SER VICE I I I

regi ster handler(A) I

START EVENT LOOP

I I I I I I run event loop() 1 I I

I I I handle eventsQ I

! I I

FOREACFI EVENT DO I I

I

I handle input()

**CONNECTION EVE<t T**

L = new Logging\_Handler I

R EGISTER NEW HANDLER L.open (A); I

FOR CLIENT 1/Q register handler(L) I

I I I I handle input() I

DATA EVENT I

I

I write()

PROCESS LOGGING I I

RECORD I I I I I

图 1-15 服务器日志看守的交互图

图 1-15 描述多种框架与相互协作、以提供日志服务的应用特有对象之间的运行时交互。看守配置在 ACE

\_Service\_Config::open 方法中完成。该方法查询下面的 svc.conf 文件，它指定将被配置进看守的服务：

static ACE\_Svc\_manager –p 911 dynamic Logger Service\_Object \*

./Logger.so:\_alloc() “-p 7001”

通过将指定的 ACE\_Service\_Object 插入 ACE\_Service\_Repository，并在 ACE\_Reactor 上登记服务对象处 理器的 ACE\_Event\_Handler 部分，来对 svc.conf 文件中的每个服务配置条目进行处理。

当所有的配置活动完成时，上面所示的主驱动程序调用 ACE\_Service\_Config 的 run\_reactor\_event\_lo op 方法。该方法进入事件循环，持续地调用 ACE\_Reactor::handle\_events 服务分派方法。如图 1-9 所示， 该分派函数阻塞并等待事件的发生（比如来自客户的连接请求或 I/O）。在这些事件发生时，ACE\_Reacto

r 自动分派先前登记的事件处理器，以执行指定的应用特有服务。

ASX 框架还响应触发看守运行时重配置的外部事件。无论何时正在执行的基于 ASX 的看守接收到 预指定的外部事件（比如 UNIX SIGHUP 信号），就会执行上面所述的动态配置步骤。取决于 svc.conf 文 件的已被更新的内容，服务可以被增加、挂起、恢复或从看守中移除。

ASX 框架的动态重配置机制使得开发者无需进行大量的重新开发和安装工作，就可以修改服务器日 志看守的功能或调谐性能。例如，调试日志服务的一个有问题的实现只需要动态地重安装一个功能等价 的服务，在其中包含有额外的手段来帮助隔离错误行为的来源。注意无须修改、重编译、重链接或重启 当前正在执行的服务器日志看守，就可完成此重安装过程。

**1.4.2** 分布式 **PBX** 监控系统

图 1-16 演示专用分组交换机（PBX）电信交换监控系统的客户/服务器体系结构，该系统使用 ASX 框架组件[20]实现。在此分布式通信系统中，服务器经由高速通信链路接收并处理由一或多个与服务器相 连的 PBX 产生的状态信息。服务器转换此状态信息，并将其通过网络转发给客户终端系统，再由后者以 图形方式显示给终端用户。终端用户通常是超级用户，他们使用 PBX 状态信息来监控系统全体成员的性 能，并预测资源的分配、以满足客户的需求。

与服务器相连的 PBX 设备由 Device\_Adapter ACE\_Module 控制。该 ACE\_Module 使服务器的其余部 分与 PBX 特有的通信特性相屏蔽。Device\_Adapter ACE\_Module 的读端维护一组 Device\_Handler 对象（每 个 PBX 一个），它们负责将到来的设备事件解析并转换成规范的、不依赖于 PBX 的消息对象；这种对象 在一种在[6]中描述的灵活的消息管理类之上构建。

在初始化之后，到来的规范消息对象被传递给 Event\_Analyzer ACE\_Module 的读端。该 Module 为服 务器实现应用特有的功能。在 Event\_Analyzer 中维护的一个内部寻址表被用于确定哪些客户应接收消息 对象。在 Event\_Analyzer 确定适当的目的地后，消息对象被转发给 Multicast\_Router ACE\_Module 的读 端。

Multicast\_Router ACE\_Module 是一个可复用组件，它使应用特有的服务器代码的余下部分与客户/

服务器交互的知识，以及对通信协议的特定选择屏蔽开。通过建立连接到 Multicast\_Router ACE\_Module，

客户预订接收服务器发布的事件。Multicast\_Router ACE\_Module 的写端接受来自客户的连接请求，并创 建分离的 Client\_Handler 对象来管理每一个客户连接。该 Client\_Handler 对象处理在服务器和与之相关联 的客户之间所有的后继数据传输和控制操作。一旦客户与服务器连接上，客户就指明它希望监控的 PBX 事件的类型。从这一点开始，当 Multicast\_Router 的读端接收到来自 Event\_Analyzer 的消息对象时，它就 自动将此消息多点发送给所有预订接收该消息对象中封装的特定类型的事件的客户。

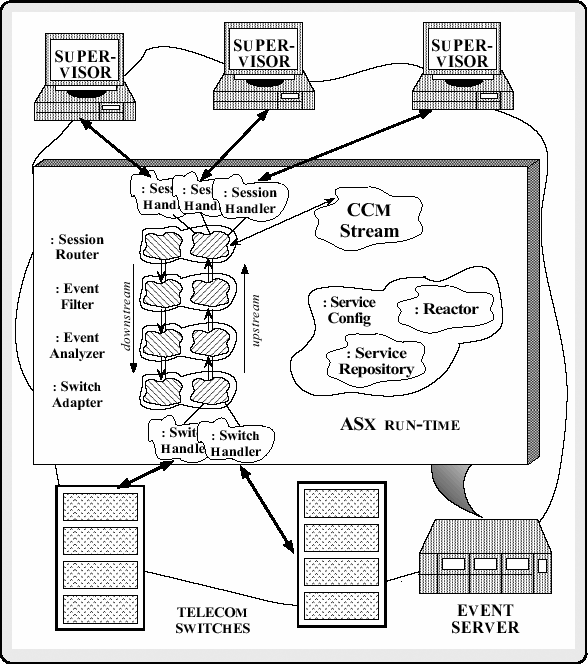


图 1-16 PBX 应用的 ASX 组件

ACE\_Service\_Config 对象被服务器用于控制在安装时静态配置、或在运行过程中动态配置的流模块 组件的初始化和终止。ACE\_Service\_Config 对象包含有 ACE\_Reactor 事件多路分离器的实例，用于将到 来的客户消息分派给适当的 Client\_Handler 或 Device\_Handler 事件处理器。自客户到达的控制消息沿着流 的写端向下发送，开始是 Multicast\_Router，接着是从相互连接的流 ACE\_Module 的写端到 Device\_Adapt

er，后者再将控制消息发送给适当的 PBX 设备。同样地，Reactor 检测来自 PBX 设备的事件，从 Device

\_Adapter ACE\_Module 开始，沿着流向上分派它们。

**1.4.3** 服务器配置

组成 PBX 服务器的 ACE\_Module 可以在任何时候配置进服务器。ASX 框架通过使用 svc.conf 配置脚 本驱动的显式动态链接来提供这样的灵活性。下面的配置脚本指示哪些服务将被动态链接进服务器的地 址空间：

stream Server\_Stream dynamic

STREAM \* /svcs/Server\_Stream.so : \_alloc()

{

dynamic Device\_Adapter

Module \* /svcs/DA.so:\_alloc() "-p 2001" dynamic Event\_Analyzer

Module \* /svcs/EA.so:\_alloc()

dynamic Multicast\_Router

Module \* /svcs/MR.so:\_alloc() "-p 2010"

}

该配置脚本指示各层次相关的服务被动态链接和推入 Server Stream 的顺序。在应用初始化过程中，Servi

ce Config 类解析此配置脚本，并执行每一条目的指令。

Server Stream 由三个动态配置进服务器的 ACE\_Module（Device\_Router、Event\_Analyzer，和 Multi cast\_Router）组成。指定的共享对象文件被动态链接进服务器（如 dynamic 指令所指定的）。随后通过调 用\_alloc 函数，Module 对象的一个实例就被从共享对象库中提取出来。如下面所描述的，如果需要的话， 这些模块可随后被更新和重新链接（例如，安装 ACE\_Module 的更新版本），而无须完全终止执行中的 PBX 服务器。

**1.4.4** 服务器重配置

将服务静态配置进通信软件应用有着许多缺点。例如，如果过多的服务被配置进应用的服务器端， 并且有过多的活动客户同时访问这些服务，就有可能导致性能瓶颈。相反，将过多的服务配置进客户端 也有可能导致性能瓶颈，因为客户常常运行在不那么强大的终端系统上。一般而言，要预先确定适当的 应用服务的划分是困难的，因为处理特性和工作负载可能会随着时间而发生变化。因而，ASX 框架的一 个主要目标就是开发面向对象的服务配置机制，允许开发者将关于哪些服务运行在客户端，哪些运行在 服务器端的决定推迟到开发周期的非常迟后的阶段（也就是，安装时或运行时）。

为了便利灵活的重配置，ASX 框架提供的运行时控制环境使开发者能够在安装时静态地、或在运行 过程中动态地变更他们的应用服务的配置。这样的机制是有用的，因为不同的 OS/硬件平台和不同的网络 特性常常需要不同的服务配置。例如，在一些配置中，服务器执行大多数工作，而在其他配置中，客户

完成更多的工作。而且，在不同的环境下（比如可用的是多处理器服务器平台，还是高速网络），可能 需要不同的终端系统配置。图 1-17 演示了在服务器的处理构成主要瓶颈时，怎样对在图 1-16 中所示的配 置进行变更，以在分布式环境中高效地进行运作。

该重配置过程通过下面的脚本完成：

suspend Server\_Stream stream Server\_Stream

{

remove Event\_Analyzer

}

remote "-h all -p 911"

{

stream Server\_Stream

{

dynamic Event\_Analyzer

Module \* /svcs/EA.so : \_alloc()

}

}

resume Server\_Stream

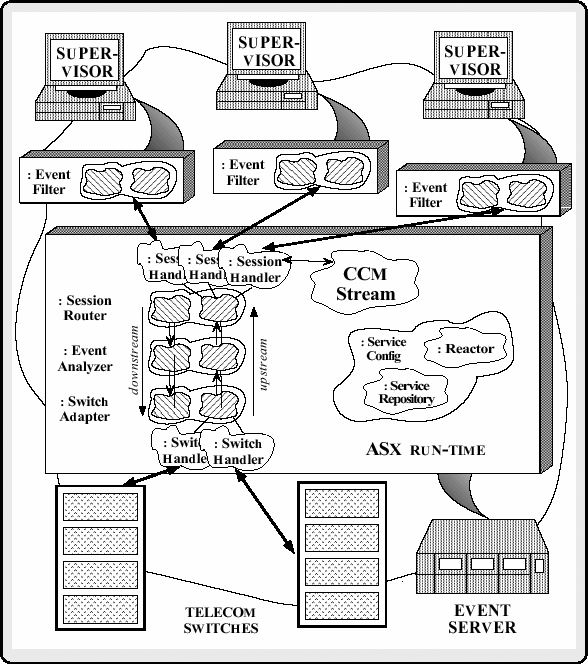


图 1-17 PBX 监控系统的重配置

这个新脚本通过从服务器的流中动态地解除 ACE\_Module 的链接，并动态地将它们链接进每个客户 的流中[20]，将处理功能从服务器移到了客户。ASX 框架替换了以前的使用特别技术的体系结构（比如 参数传递和共享内存），以在服务器中的相关服务间交换消息。与前面的方法相对照，ASX 框架提供的 高度统一的 ACE\_Module 互连机制极大地改善了可移植性和可配置性。

**1.5** 结束语

ACE 自适配通信环境是由许多 OO 组件组成的工具包，它通过实施成功的设计模式和软件体系结构，

帮助降低了分布式软件的复杂性。ACE 将许多通用的与通信相关的功能（比如本地和远地 IPC[4]、事件 多路分离和服务处理器分派[14]、服务初始化[16, 17]、含有整体式和层次化服务的分布式应用[20]的配置 机制、分布式日志[13]，以及服务内部及服务间的并发）统一进可复用的 OO 组件和框架中。

ACE可在[http://www.cs.wustl.edu/~schmidt/ACE.html](http://www.cs.wustl.edu/%7Eschmidt/ACE.html)自由获取。该发布含有源代码、文档，以及在圣路 易斯的华盛顿大学开发的测试实例驱动程序。目前ACE正在许多公司中用于开发通信软件，其中包括Bel lcore、西门子、DEC、摩托罗拉、爱立信、柯达，和McDonnell Douglas。ACE已被移植到Win32（也就 是，Win95、WinNT、Win2K），大多数版本的UNIX（例如，SunOS 4.x和5.x、SGI IRIX、HP-UX、OS F/1、AIX、Linux和SCO），以及POSIX系统（比如VxWorks和MVS OpenEdition）。同时有C++[6]和Jav a[48]版本的ACE可用。

参考文献

[1] F. P. Brooks, *The Mythical Man-Mont*h. Reading, MA: Addison-Wesley, 1975.

[2] S. J. Leffler, M. McKusick, M. Karels, and J. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Op erating Syste*m. Addison-Wesley, 1989.

[3] S. Rago, *UNIX System V Network Programmin*g. Reading, MA: Addison-Wesley, 1993.

[4] D. C. Schmidt, T. H. Harrison, and E. Al-Shaer, “Object-Oriented Components for High-speed Network Programmin g,” in *Proceedings of the* 1st *Conference on Object-Oriented Technologies and System*s, (Monterey, CA), USENIX, June 1

995.

[5] G. Booch, *Object Oriented Analysis and Design with Applications* (2nd *Edition*). Redwood City, California: Benjamin/ Cummings, 1993.

[6] D. C. Schmidt, “ACE: an Object-Oriented Framework for Developing Distributed Applications,” in *Proceedings of the*

6th *USENIX C++ Technical Conferenc*e, (Cambridge, Massachusetts), USENIX Association, April 1994.

[7] D.E.ComerandD.L.Stevens,*Internetworking with TCP/IP Vol III: Client – Server Programming and Application*s.Englew ood Cliffs, NJ: Prentice Hall, 1992.

[8] W. R. Stevens, *UNIX Network Programming, First Editio*n. Englewood Cliffs, NJ: Prentice Hall, 1990.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Softwar*e. Reading, MA: Addison-Wesley, 1995.

[10] D. C. Schmidt, “Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dis patching,” in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-We sley, 1995.

[11] R. G. Lavender and D. C. Schmidt, “Active Object: an Object Behavioral Pattern for Concurrent Programming,” in

*Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley,

1996.

[12] R. Johnson and B. Foote, “Designing Reusable Classes,” *Journal of Object-Oriented Programmin*g, vol. 1, pp. 22–3

5, June/July 1988.

[13] D. C. Schmidt, “The Reactor: An Object-Oriented Interface for Event-Driven UNIX I/O Multiplexing (Part 1 of 2),”

*C++ Repor*t, vol. 5, February 1993.

[14] D. C. Schmidt, “The Object-Oriented Design and Implementation of the Reactor: A C++ Wrapper for UNIX I/O M

ultiplexing (Part 2 of 2),” *C++ Repor*t, vol. 5, September 1993.

[15] T. H. Harrison, D. C. Schmidt, and I. Pyarali, “Asynchronous Completion Token: an Object Behavioral Pattern for

Efficient Asynchronous Event Handling,” in *The* 3rd *Annual Conference on the Pattern Languagesof Programs (Washingto*

*n University technical report#WUCS-97-07*), (Monticello, Illinois), pp. 1–7, February 1997.

[16] D. C. Schmidt, “Design Patterns for Initializing Network Services: Introducing the Acceptor and Connector Pattern s,” *C++ Repor*t, vol. 7, November/December 1995.

[17] D. C. Schmidt, “Connector: a Design Pattern for Actively Initializing Network Services,” *C++ Repor*t, vol. 8, Janua ry 1996.

[18] D. C. Schmidt, “Acceptor and Connector: Design Patterns for Initializing Communication Services,” in *The* 1st *Europ*

*ean Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-07*), July 1997. [19] D. C. Schmidt, “IPC SAP: An Object-Oriented Interface to Interprocess Communication Services,” *C++ Repor*t, vol.

4, November/December 1992.

[20] D. C. Schmidt and T. Suda, “An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Co mmunication Systems,” *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Syst ems*), vol. 2, pp. 280–293, December 1994.

[21] P. Jain and D. C. Schmidt, “Service Configurator: APattern for Dynamic Configuration and Reconfiguration of Com munication Services,” in *The* 3rd *Pattern Languagesof Programming Conference (Washington University technical report # WUCS-97- 07*), February 1997.

[22] D. C. Schmidt, “An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit,” Tech. Re p. WUCS-95-31, Washington University, St. Louis, September 1995.

[23] D. C. Schmidt and C. D. Cranor, “Half-Sync/Half-Async: an Architectural Pattern for Efficient and Well-structured Concurrent I/O,” in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, M A: Addison-Wesley, 1996.

[24] D. C. Schmidt and T. Harrison, “Double-Checked Locking – An Object Behavioral Pattern for Initializing and Acce ssing Thread-safe Objects Efficiently,” in *The* 3rd *Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-07*), February 1997.

[25] Bjarne Stroustrup and Margret Ellis, *The Annotated C++ Reference Manua*l. Addison-Wesley, 1990.

[26] I. Pyarali, T. H. Harrison, and D. C. Schmidt, “Design and Performance of an Object-Oriented Framework for High

-Performance Electronic Medical Imaging,” *USENIX Computing System*s, vol. 9, November/December 1996. [27] R. Davis, *Win32 Network Programmin*g. Reading, MA: Addison-Wesley, 1996.

[28] D. L. Presotto and D. M. Ritchie, “Interprocess Communication in the Ninth Edition UNIX System,” *UNIX Researc h SystemPapers, Tenth Editio*n, vol. 2, no. 8, pp. 523–530,1990.

[29] W. R. Stevens, *Advanced Programming in the UNIX Environmen*t. Reading, Massachusetts: Addison Wesley, 1992. [30] R. Gingell, J. Moran, and W. Shannon, “Virtual Memory Architecture in SunOS,” in *Proceedings of the Summer 19*

*87 USENIX Technical Conferenc*e, (Phoenix, Arizona), 1987.

[31] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. W illiams, “Beyond Multiprocessing... Multithreading the SunOS Kernel,” in *Proceedings of the Summer USENIX Conferenc*e, (San Antonio, Texas), June 1992.

[32] IEEE, *Threads Extension for Portable Operating Systems (Draft 10*), February 1996.

[33] D. C. Schmidt, “Transparently Parameterizing Synchronization Mechanisms into a Concurrent Distributed Applicatio n,” *C++ Repor*t, vol. 6, July/August 1994. 24

[34] G. Booch and M. Vilot, “Simplifying the Booch Components,” *C++ Repor*t, vol. 5, June 1993.

[35] R. Gingell, M. Lee, X. Dang, and M. Weeks, “Shared Libraries in SunOS,” in *Proceedingsof the Summer 1987 US ENIXTechnical Conferenc*e, (Phoenix, Arizona), 1987.

[36] W. W. Ho and R. Olsson, “An Approach to Genuine Dynamic Linking,” *Software: Practice and Experienc*e, vol. 2

1, pp. 375–390, Apr. 1991.

[37] H. Custer, *Inside Windows N*T. Redmond, Washington: Microsoft Press, 1993.

[38] D. C. Schmidt and P. Stephenson, “Experiences Using Design Patterns to Evolve System Software Across Diverse OS Platforms,” in *Proceedings of the* 9th *European Conference on Object-Oriented Programmin*g, (Aarhus, Denmark), AC M, August 1995.

[39] D. Ritchie, “A Stream Input–Output System,” *AT&TBell Labs Technical Journa*l, vol. 63, pp. 311–324, Oct. 1984. [40] N. C. Hutchinson and L. L. Peterson, “The x-kernel: An Architecture for Implementing Network Protocols,” *IEEE T ransactions on Software Engineerin*g, vol. 17, pp. 64–76, January 1991.

[41] J. M. Zweig, “The Conduit: a Communication Abstraction in C++,” in *Proceedings of the* 2nd *USENIX C++ Confer*

*enc*e, pp. 191–203, USENIX Association, April 1990.

[42] D. C. Schmidt and T. Suda, “Transport System Architecture Services for High-Performance Communications System s,” *IEEE Journal on Selected Areas in Communicatio*n, vol. 11, pp. 489–506, May 1993.

[43] D. C. Schmidt, B. Stiller, T. Suda, A. Tantawy, and M. Zitterbart, “Language Support for Flexible, Application-Tai lored Protocol Configuration,” in *Proceedings of the* 18th *Conference on Local Computer Network*s, (Minneapolis, Minnes ota), pp. 369–378, IEEE, Sept. 1993.

[44] A. McRae, “Hardware Profiling of Kernels,” in *USENIX Winter Conferenc*e, (San Diego, CA), USENIX Association, Jan. 1993.

[45] S. Saxena, J. K. Peacock, F. Yang, V. Verma, and M. Krishnan, “Pitfalls in Multithreading SVR4 STREAMS and other Weightless Processes,” in *Proceedings of the Winter USENIX Conferenc*e, (San Diego, CA), pp. 85–106, Jan. 1993. [46] D. C. Schmidt, D. F. Box, and T. Suda, “ADAPTIVE: A Dynamically Assembled Protocol Transformation, Integrati on, and eValuation Environment,” *Journal of Concurrency: Practice and Experienc*e, vol. 5, pp. 269–286, June 1993.

[47] D. C. Schmidt, “A Family of Design Patterns for Application-level Gateways,” *The Theory and Practice of Object*

*Systems (Special Issue on Patterns and Pattern Languages*), vol. 2, no. 1, 1996.

[48] P. Jain and D. Schmidt, “Experiences Converting a C++ Communication Software Framework to Java,” *C++Repor*t, vol. 9, January 1997.

第 **2** 章 包装外观（**Wrapper Façade**）：用于在类中封装函数的结构

型模式

Douglas C. Schmidt

**2.1** 介绍

本论文描述包装外观模式。该模式的意图是通过面向对象（OO）类接口来封装低级函数和数据结构。 常见的包装外观模式的例子是像 MFC、ACE 和 AWT 这样的类库，它们封装本地的 OS C API，比如 so cket、pthreads 或 GUI 函数。

直接对本地 OS C API 进行编程会使网络应用繁琐、不健壮、不可移植，且难以维护，因为应用开 发者需要了解许多低级、易错的细节。本论文阐释包装外观模式怎样使这些类型的应用变得更为简洁、 健壮、可移植和可维护。

本论文被组织如下：2.2 详细描述使用西门子格式[1]的包装外观模式，2.3 给出结束语。

**2.2** 包装外观模式

**2.2.1** 意图

在更为简洁、健壮、可移植和可维护的较高级面向对象类接口中封装低级函数和数据结构。

**2.2.2** 例子

为阐释包装外观模式，考虑图 2-1 中所示的分布式日志服务的服务器。客户应用使用日志服务来记录 关于它们在分布式环境中的执行状态的信息。这些状态信息通常包括错误通知、调试跟踪和性能诊断。 日志记录被发送到中央日志服务器，由它将记录写到各种输出设备，比如网络管理控制台、打印机或数 据库。

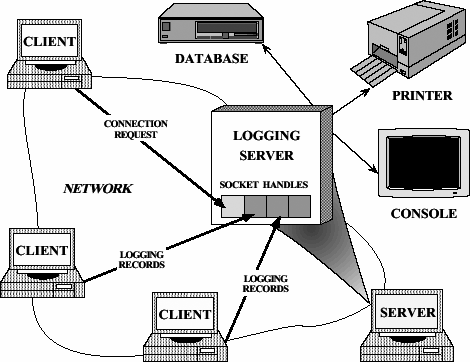


图 2-1 分布式日志服务

图 2-1 所示的日志服务器处理客户发送的连接请求和日志记录。日志记录和连接请求可以并发地在多 个 socket 句柄上到达。各个句柄标识在 OS 中管理的网络通信资源。

客户使用像 TCP[2]这样的面向连接协议与日志服务器通信。因而，当客户想要记录日志数据时，它 必须首先向日志服务器发送连接请求。服务器使用句柄工厂（handle factory）来接受连接请求，句柄工 厂在客户知道的网络地址上进行侦听。当连接请求到达时，OS 句柄工厂接受客户的连接，并创建表示该 客户的连接端点的 socket 句柄。该句柄被返回给日志服务器，后者在这个句柄和其他句柄上等待客户日 志请求到达。一旦客户被连接，它们可以发送日志记录给服务器。服务器通过已连接的 socket 句柄来接 收这些记录，处理记录，并将它们写到它们的输出设备。

开发并发处理多个客户的日志服务器的常见方法是使用低级 C 语言函数和数据结构来完成线程、同 步及网络通信等操作。例如，图 2-2 演示怎样将 Solaris 线程[3]和 socket[4]网络编程 API 用于开发多线程 日志服务器。

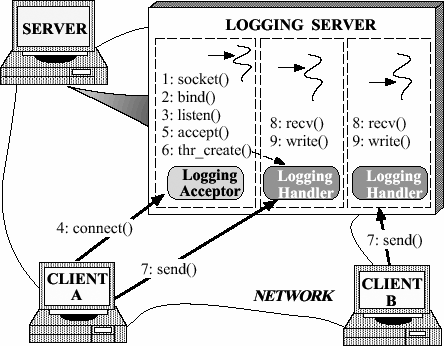


图 2-2 多线程日志服务器

在此设计中，日志服务器的句柄工厂在它的主线程中接受客户网络连接。它随即派生一个新线程， 在单独的连接中运行 logging\_handler 函数、以处理来自每个客户的日志记录。下面的两个 C 函数演示怎 样使用 socket、互斥体和线程的本地 Solaris OS API 来实现此日志服务器设计5。

// At file scope.

// Keep track of number of logging requests. static int request\_count;

// Lock to protect request\_count. static mutex\_t lock;

// Forward declaration.

static void \*logging\_handler (void \*);

// Port number to listen on for requests. static const int logging\_port = 10000;

// Main driver function for the multi-threaded

// logging server. Some error handling has been

// omitted to save space in the example. int main (int argc, char \*argv[])

{

struct sockaddr\_in sock\_addr;

// Handle UNIX/Win32 portability differences.

#if defined (\_WINSOCKAPI\_)

5 对此例的完整代码细节不感兴趣的读者可以跳到 2.2.3。

SOCKET acceptor;

#else

int acceptor;

#endif /\* \_WINSOCKAPI\_ \*/

// Create a local endpoint of communication. acceptor = socket (PF\_INET, SOCK\_STREAM, 0);

// Set up the address to become a server.

memset (reinterpret\_cast <void \*> (&sock\_addr), 0, sizeof sock\_addr);

sock\_addr.sin\_family = AF\_INET; sock\_addr.sin\_port = htons (logging\_port); sock\_addr.sin\_addr.s\_addr = htonl (INADDR\_ANY);

// Associate address with endpoint. bind (acceptor,

reinterpret\_cast <struct sockaddr \*> (&sock\_addr),

sizeof sock\_addr);

// Make endpoint listen for connections. listen (acceptor, 5);

// Main server event loop. for (;;)

{

thread\_t t\_id;

// Handle UNIX/Win32 portability differences.

#if defined (\_WINSOCKAPI\_) SOCKET h;

#else

int h;

#endif /\* \_WINSOCKAPI\_ \*/

// Block waiting for clients to connect. int h = accept (acceptor, 0, 0);

// Spawn a new thread that runs the

// <logging\_handler> entry point and

// processes client logging records on

// socket handle <h>. thr\_create (0, 0,

logging\_handler,

reinterpret\_cast <void \*> (h), THR\_DETACHED,

&t\_id);

}

/\* NOTREACHED \*/

return 0;

}

logging\_handler 函数运行在单独的线程控制中，也就是，每个客户一个线程。它在各个连接上接收并处理

日志记录，如下所示：

// Entry point that processes logging records for

// one client connection.

void \*logging\_handler (void \*arg)

{

// Handle UNIX/Win32 portability differences.

#if defined (\_WINSOCKAPI\_)

SOCKET h = reinterpret\_cast <SOCKET> (arg);

#else

int h = reinterpret\_cast <int> (arg);

#endif /\* \_WINSOCKAPI\_ \*/

for (;;)

{

UINT\_32 len; // Ensure a 32-bit quantity. char log\_record[LOG\_RECORD\_MAX];

// The first <recv> reads the length

// (stored as a 32-bit integer) of

// adjacent logging record. This code

// does not handle "short-<recv>s". ssize\_t n = recv

(h,

reinterpret\_cast <char \*> (&len), sizeof len, 0);

// Bail out if we don’t get the expected len. if (n <= sizeof len) break;

len = ntohl (len); // Convert byte-ordering. if (len > LOG\_RECORD\_MAX) break;

// The second <recv> then reads <len>

// bytes to obtain the actual record.

// This code handles "short-<recv>s".

for (size\_t nread = 0;

nread < len;

nread += n)

{

n = recv (h, log\_record + nread, len - nread, 0);

// Bail out if an error occurs. if (n <= 0) return 0;

}

mutex\_lock (&lock);

// Execute following two statements in a

// critical section to avoid race conditions

// and scrambled output, respectively.

++request\_count; // Count # of requests received.

if (write (1, log\_record, len) == -1)

// Unexpected error occurred, bail out. break;

mutex\_unlock (&lock);

}

close (h);

return 0;

}

注意全部的线程、同步及网络代码是怎样使用 Solaris 操作系统所提供的低级 C 函数和数据结构来编写的。

**2.2.3** 上下文

访问由低级函数和数据结构所提供服务的应用。

**2.2.4** 问题

网络应用常常使用 2.2.2 中所演示的低级函数和数据结构来编写。尽管这是一种惯用方法，由于不能 解决以下问题，它会给应用开发者造成许多问题：

繁琐、不健壮的程序：直接对低级函数和数据结构编程的应用开发者必须反复地重写大量冗长乏味的软 件逻辑。一般而言，编写和维护起来很乏味的代码常常含有微妙而有害的错误。

例如，在 2.2.2 的 main 函数中创建和初始化接受器 socket 的代码是容易出错的，比如没有对 sock\_a

ddr 清零，或没有对 logging\_port 号使用 htons[5]。mutex\_lock 和 mutex\_unlock 也容易被误用。例如，如 果 write 调用返回-1，logging\_handler 代码就会不释放互斥锁而跳出循环。同样地，如果嵌套的 for 循环 在遇到错误时返回，socket 句柄 h 就不会被关闭。

缺乏可移植性：使用低级函数和数据结构编写的软件常常不能在不同的 OS 平台和编译器间移植。而且， 它们甚至常常不能在同一 OS 或编译器的不同版本间移植。不可移植性源于隐藏在基于低级 API 的函数 和数据结构中的信息匮乏。

例如，在 2.2.2 中的日志服务器实现已经硬编码了对若干不可移植的本地 OS 线程和网络编程 C API 的依赖。特别地，对 thr\_create、mutex\_lock 和 mutex\_unlock 的使用不能移植到非 Solaris OS 平台上。同 样地，特定的 socket 特性，比如使用 int 表示 socket 句柄，不能移植到像 Win32 的 WinSock 这样的非 Un ix 平台上；WinSock 将 socket 句柄表示为指针。

高维护开销：C 和 C++开发者通常通过使用#ifdef 在他们的应用源码中显式地增加条件编译指令来获得可 移植性。但是，使用条件编译来处理平台特有的变种在各方面都增加了应用源码的物理设计复杂性[6]。 开发者难以维护和扩展这样的软件，因为平台特有的实现细节分散在应用源文件的各个地方。

例如，处理 socket 数据类型的 Win32 和 UNIX 可移植性（也就是，SOCKET vs. int）的#ifdef 妨碍 了代码的可读性。对像这样的低级 C API 进行编程的开发者必须十分熟悉许多 OS 平台特性，才能编写 和维护该代码。

由于有这些缺点，通过对低级函数和数据结构直接进行编程来开发应用常常并非是有效的设计选择。

**2.2.5** 解决方案

确保应用不直接访问低级函数和数据结构的一种有效途径是使用包装外观模式。对于每一组相关的 函数和数据结构，创建一或多个包装外观类，在包装外观接口所提供的更为简洁、健壮、可移植和可维 护的方法中封装低级函数和数据结构。

**2.2.6** 结构

包装外观模式的参与者的结构在图 2-1 中的 UML 类图中演示：

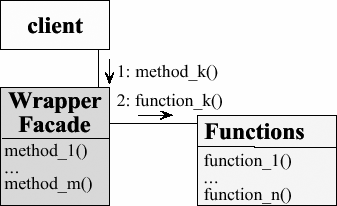


图 2-1 包装外观模式的参与者的结构 包装外观模式中的关键参与者包括： 函数（**Function**）：函数是现有的低级函数和数据结构，它们提供内聚的（cohesive）服务。

包装外观（**Wrapper Façade**）：包装外观是封装函数和与其相关联的数据结构的一个或一组类。包装外 观提供的方法将客户调用转发给一或多个低级函数。

**2.2.7** 动力特性

图 2-2 演示包装外观模式中的各种协作：

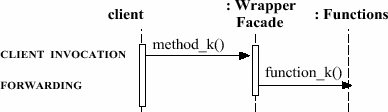


图 2-2 包装外观模式中的协作

如下所述，这些协作是十分简单明了的：

**1.** 客户调用（**Client invocation**）：客户通过包装外观的实例来调用方法。

**2.** 转发（**Forwarding**）：包装外观方法将请求转发给它封装的一或多个底层函数，并传递函数所需的任 何内部数据结构。

**2.2.8** 实现

这一部分解释通过包装外观模式实现组件和应用所涉及的步骤。我们将阐释这些包装外观是怎样克 服繁琐、不健壮的程序、缺乏可移植性，以及高维护开销等问题的；这些问题折磨着使用低级函数和数 据结构的解决方案。

这里介绍的例子基于 2.2.2 描述的日志服务器，图 2-3 演示此例中的结构和参与者。这一部分中的例 子应用了来自 ACE 框架[7]的可复用组件。ACE 提供一组丰富的可复用 C++包装和框架组件，以跨越广 泛的 OS 平台完成常见的通信软件任务。

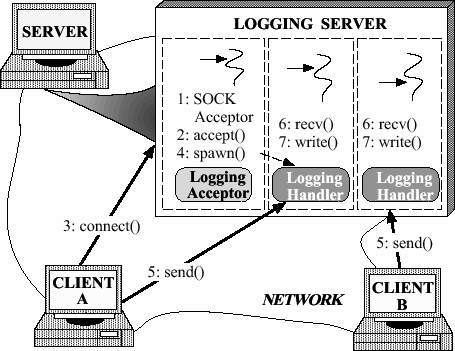


图 2-3 多线程日志服务器

可采取下面的步骤来实现包装外观模式：

**1.** 确定现有函数间的内聚的抽象和关系：像 Win32、POSIX 或 X Windows 这样被实现为独立的函数和 数据结构的传统 API 提供许多内聚的抽象，比如用于网络编程、同步和线程，以及 GUI 管理的机制。 但是，由于在像 C 这样的低级语言中缺乏数据抽象支持，开发者常常并不能马上明了这些现有的函 数和数据结构是怎样互相关联的。因此，应用包装外观的第一步就是确定现有 API 中的较低级函数之 间的内聚的抽象和关系。换句话说，我们通过将现有的低级 API 函数和数据结构聚合进一或多个类中 来定义一种“对象模型”。

在我们的日志例子中，我们从仔细检查我们原来的日志服务器实现开始。该实现使用了许多低级 函数，由它们实际提供若干内聚的服务，比如同步和网络通信。例如，mutex\_lock 和 mutex\_unlock 函数与互斥体同步抽象相关联。同样地，socket、bind、listen 和 accept 函数扮演了网络编程抽象的多 种角色。

**2.** 将内聚的函数组聚合进包装外观类和方法中：该步骤可划分为以下子步骤： 在此步骤中，我们为每组相关于特定抽象的函数和数据结构定义一或多个包装外观类。

**A.** 创建内聚的类：我们从为每组相关于特定抽象的函数和数据结构定义一或多个包装外观类开始。 用于创建内聚的类的若干常用标准包括：

z 将具有高内聚性（cohesion）的函数合并进独立的类中，同时使类之间不必要的耦合最小化。

z 确定在底层函数中什么是通用的什么是可变的，并把函数分组进类中，从而将变化隔离在 统一的接口后面。

一般而言，如果原来的 API 含有广泛的相关函数，就有可能必须创建若干包装外观类来适当地对事务 进行分理。

**B.** 将多个独立函数合并进类方法中：除了将现有函数分组进类中，在每个包装类中将多个独立函数

组合进数目更少的方法中常常也是有益的。例如，为确保一组低级函数以适当的顺序被调用，可能必 须要采用此设计。

**C.** 选择间接层次：大多数包装外观类简单地将它们的方法调用直接转发给底层的低级函数。如果包 装外观方法是内联的，与直接调用低级函数相比，可能并没有额外的间接层次。为增强可扩展性，还 可以通过动态分派包装外观方法实现来增加另外的间接层次。在这种情况下，包装外观类扮演桥接（B ridge）模式[8]中的抽象（Abstraction）角色。

**D.** 确定在哪里处理平台特有的变种：使平台特有的应用代码最少化是使用包装外观模式的重要好处。 因而，尽管包装外观类方法的实现在不同的 OS 平台上可以不同，它们应该提供统一的、平台无关的 接口。

处理平台特有变种的一种策略是在包装外观类方法实现中使用#ifdef。在联合使用#ifdef 和自动配 置工具（比如 GNU autoconf）时，可以通过单一的源码树创建统一的、不依赖于平台的包装外观。 另一种可选策略是将不同的包装外观类实现分解进分离的目录中（例如，每个平台有一个目录），并 配置语言处理工具，以在编译时将适当的包装外观类包含进应用中。

选择特定的策略在很大程度上取决于包装外观方法实现变动的频度。例如，如果它们频繁变动， 为每个平台正确地更新#ifdef 可能是单调乏味的。同样地，所有依赖于该文件的文件可能都需要重编 译，即使变动仅仅对一个平台来说是必需的。

在我们的日志例子中，我们将为互斥体、socket 和线程定义包装外观类，以演示每一子步骤是怎样被 实施的。如下所示：

z 互斥体包装外观：我们首先定义 Thread\_Mutex 抽象，在统一和可移植的类接口中封装 Solaris 互斥体 函数：

class Thread\_Mutex

{

public:

Thread\_Mutex (void)

{

mutex\_init (&mutex\_, 0, 0);

}

˜Thread\_Mutex (void)

{

mutex\_destroy (&mutex\_);

}

int acquire (void)

{

return mutex\_lock (&mutex\_);

}

int release (void)

{

return mutex\_unlock (&mutex\_);

}

private:

// Solaris-specific Mutex mechanism. mutex\_t mutex\_;

// = Disallow copying and assignment. Thread\_Mutex (const Thread\_Mutex &); void operator= (const Thread\_Mutex &);

};

通过定义 Thread\_Mutex 类接口，并随之编写使用它、而不是低级本地 OS C API 的应用，我们可以

很容易地将我们的包装外观移植到其他平台。例如，下面的 Thread\_Mutex 实现在 Win32 上工作：

class Thread\_Mutex

{

public:

Thread\_Mutex (void)

{

InitializeCriticalSection (&mutex\_);

}

˜Thread\_Mutex (void)

{

DeleteCriticalSection (&mutex\_);

}

int acquire (void)

{

EnterCriticalSection (&mutex\_); return 0;

}

int release (void)

{

LeaveCriticalSection (&mutex\_); return 0;

}

private:

// Win32-specific Mutex mechanism. CRITICAL\_SECTION mutex\_;

// = Disallow copying and assignment. Thread\_Mutex (const Thread\_Mutex &); void operator= (const Thread\_Mutex &);

};

如早先所描述的，我们可以通过在 Thread\_Mutex 方法实现中使用#ifdef 以及自动配置工具（比如 G

UN autoconf）来支持多个 OS 平台，以使用单一源码树提供统一的、平台无关的互斥体抽象。相反，我 们也可以将不同的 Thread\_Mutex 实现分解进分离的目录中，并指示我们的语言处理工具在编译时将适当 的版本包含进我们的应用中。

除了改善可移植性，我们的 Thread\_Mutex 包装外观还提供比直接编程低级 Solaris 函数和 mutex\_t 数 据结构更不容易出错的互斥体接口。例如，我们可以使用 C++ private 访问控制指示符来禁止互斥体的拷 贝和赋值；这样的使用是错误的，但却不会被不那么强类型化的 C 编程 API 所阻止。

z **socket** 包装外观：socket API 比 Solaris 互斥体 API 要大得多，也有表现力得多[5]。因此，我们必须 定义一组相关的包装外观类来封装 socket。我们将从定义下面的处理 UNIX/Win32 可移植性差异的 t ypedef 开始：

#if !defined (\_WINSOCKAPI\_)

typedef int SOCKET;

#define INVALID\_HANDLE\_VALUE -1

#endif /\* \_WINSOCKAPI\_ \*/

接下来，我们将定义 INET\_Addr 类，封装 Internet 域地址结构：

class INET\_Addr

{

public:

INET\_Addr (u\_short port, long addr)

{

// Set up the address to become a server.

memset (reinterpret\_cast <void \*> (&addr\_), 0, sizeof addr\_);

addr\_.sin\_family = AF\_INET; addr\_.sin\_port = htons (port); addr\_.sin\_addr.s\_addr = htonl (addr);

}

u\_short get\_port (void) const

{

return addr\_.sin\_port;

}

long get\_ip\_addr (void) const

{

return addr\_.sin\_addr.s\_addr;

}

sockaddr \*addr (void) const

{

return reinterpret\_cast <sockaddr \*>(&addr\_);

}

size\_t size (void) const

{

return sizeof (addr\_);

}

// ...

private:

sockaddr\_in addr\_;

};

注意 INET\_Addr 构造器是怎样通过将 sockaddr\_in 域清零，并确保端口和 IP 地址被转换为网络字节 序，消除若干常见的 socket 编程错误的。

下一个包装外观类，SOCK\_Stream，对应用可在已连接 socket 句柄上调用的 I/O 操作（比如 recv 和 s end）进行封装：

class SOCK\_Stream

{

public:

// = Constructors.

// Default constructor. SOCK\_Stream (void)

: handle\_ (INVALID\_HANDLE\_VALUE) {}

// Initialize from an existing HANDLE. SOCK\_Stream (SOCKET h): handle\_ (h) {}

// Automatically close the handle on destruction.

˜SOCK\_Stream (void) { close (handle\_); }

void set\_handle (SOCKET h) { handle\_ = h; }

SOCKET get\_handle (void) const { return handle\_; }

// = I/O operations.

int recv (char \*buf, size\_t len, int flags = 0);

int send (const char \*buf, size\_t len, int flags = 0);

// ...

private:

// Handle for exchanging socket data. SOCKET handle\_;

};

注意此类是怎样确保 socket 句柄在 SOCK\_Stream 对象出作用域时被自动关闭的。

SOCK\_Stream 对象由连接工厂 SOCK\_Acceptor 创建，后者封装被动的连接建立逻辑[9]。SOCK\_Acc eptor 构造器初始化被动模式接受器 socket，以在 sock\_addr 地址上进行侦听。同样地，accept 工厂方法通 过新接受的连接来初始化 SOCK\_Stream，如下所示：

class SOCK\_Acceptor

{

public:

SOCK\_Acceptor (const INET\_Addr &sock\_addr)

{

// Create a local endpoint of communication. handle\_ = socket (PF\_INET, SOCK\_STREAM, 0);

// Associate address with endpoint.

bind (handle\_, sock\_addr.addr (), sock\_addr.size ());

// Make endpoint listen for connections. listen (handle\_, 5);

};

// Accept a connection and initialize

// the <stream>.

int accept (SOCK\_Stream &stream)

{

stream.set\_handle (accept (handle\_, 0, 0));

if (stream.get\_handle () == INVALID\_HANDLE\_VALUE)

return -1;

else return 0;

}

private:

// Socket handle factory. SOCKET handle\_;

};

注意 SOCK\_Acceptor 的构造器是怎样确保低级的 socket、bind 和 listen 函数总是以正确的次序被调用

的。

完整的 socket 包装外观集还包括 SOCK\_Connector，封装主动的连接建立逻辑[9]。

z 线程外观：在不同的 OS 平台上有许多线程 API 可用，包括 Solaris 线程、POSIX Pthreads 和 Win32

线程。这些 API 显示出微妙的语法和语义差异，例如，Solaris 和 POSIX 线程可以“分离”（detached） 模式被派生，而 Win32 线程则不行。但是，可以提供 Thread\_Manager 包装外观，在统一的 API 中封 装这些差异。如下所示：

class Thread\_Manager

{

public:

int spawn (void \*(\*entry\_point) (void \*), void \*arg,

long flags,

long stack\_size = 0,

void \*stack\_pointer = 0, thread\_t \*t\_id = 0)

{

thread\_t t;

if (t\_id == 0)

t\_id = &t;

return thr\_create (stack\_size, stack\_pointer, entry\_point, arg,

flags, t\_id);

}

// ...

};

Thread\_Manager 还提供联接（join）和取消线程的方法。

**3.** 确定错误处理机制：低级的 C 函数 API 通常使用返回值和整型代码（比如 errno）来将错误通知给它 们的调用者。但是，此技术是容易出错的，因为调用者可能会忘记检查它们的函数调用的返回状态。 更为优雅的报告错误的方式是使用异常处理。许多编程语言，比如 C++和 Java，使用异常处理来

作为错误报告机制。它也被某些操作系统所使用，比如 Win32。 使用异常处理作为包装外观类的错误处理机制有若干好处：

z 它是可扩展的：现代编程语言允许通过对现有接口和使用干扰极少的特性来扩展异常处理策略和机 制。例如，C++和 Java 使用继承来定义异常类的层次。

z 它使错误处理与正常处理得以干净地去耦合：例如，错误处理信息不会显式地传递给操作。而且，应 用不会因为没有检查函数返回值而偶然地忽略异常。

z 它可以是类型安全的：在像 C++和 Java 这样的语言中，异常以一种强类型化的方式被扔出和捕捉， 以增强错误处理代码的组织和正确性。相对于显式地检查线程专有的错误值，编译器会确保对于每种

类型的异常，将执行正确的处理器。 但是，为包装外观类使用异常处理也有若干缺点：

z 它不是通用的：不是所有语言都提供异常处理。例如，某些 C++编译器没有实现异常。同样地，当 O S 提供异常服务时，它们必须被语言扩展所支持，从而降低了代码的可移植性。

z 它使多种语言的使用变得复杂化：因为语言以不同的方式实现异常，或根本不实现异常，如果以不同 语言编写的组件扔出异常，可能很难把它们集成在一起。相反，使用整型值或结构来报告错误信息提 供了更为通用的解决方案。

z 它使资源管理变得复杂化：如果在 C++或 Java 代码块中有多个退出路径，资源管理可能会变得复杂 化[10]。因而，如果语言或编程环境不支持垃圾收集，必须注意确保在有异常扔出时删除动态分配的 对象。

z 它有着潜在的时间和**/**或空间低效的可能性：即使没有异常扔出，异常处理的糟糕实现也会带来时间 和/或空间的过度开销[10]。对于必须具有高效和低内存占用特性的嵌入式系统来说，这样的开销可能 会特别地成问题。

对于封装内核级设备驱动程序或低级的本地 OS API（它们必须被移植到许多平台上）的包装外观来 说，异常处理的缺点也是特别成问题的。对于这些类型的包装外观，更为可移植、高效和线程安全的处 理错误的方式是定义错误处理器抽象，显式地维护关于操作的成功或失败的信息。使用线程专有存储（T hread-Specific Storage）模式[11]是被广泛用于这些系统级包装外观的解决方案。

**4.** 定义相关助手类（可选）：一旦低级函数和数据结构被封装在内聚的包装外观类中，常常有可能创建 其他助手类来进一步简化应用开发。通常要在包装外观模式已被应用于将低级函数和与其关联的数据 聚合进类中之后，这些助手类的效用才变得明显起来。

例如，在我们的日志例子中，我们可以有效地利用下面的实现 C++ *Scoped Locking* 习语的 Guar d 类；该习语确保 Thread\_Mutex 被适当地释放，不管程序的控制流是怎样退出作用域的。

template <class LOCK>

class Guard

{

public:

Guard (LOCK &lock): lock\_ (lock)

{

lock\_.acquire ();

}

˜Guard (void)

{

lock\_.release ();

}

private:

// Hold the lock by reference to avoid

// the use of the copy constructor...

LOCK &lock\_;

}

Guard 类应用了[12]中描述的 C++习语，藉此，在一定作用域中“构造器获取资源而析构器释放它们”。

如下所示：

// ...

{

// Constructor of <mon> automatically

// acquires the <mutex> lock. Guard<Thread\_Mutex> mon (mutex);

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

// Destructor of <mon> automatically

// releases the <mutex> lock.

}

// ...

因为我们使用了像 Thread\_Mutex 包装外观这样的类，我们可以很容易地替换不同类型的锁定机制， 与此同时仍然复用 Guard 的自动锁定/解锁协议。例如，我们可以用 Process\_Mutex 类来取代 Thread\_Mut ex 类，如下所示：

// Acquire a process-wide mutex. Guard<Process\_Mutex> mon (mutex);

如果使用 C 函数和数据结构、而不是 C++类，获得这种程度的“可插性”（pluggability）要困难得多。

**2.2.9** 例子解答

下面的代码演示日志服务器的 main 函数，它已使用 2.2.8 描述的互斥体、socket 和线程的包装外观重

写。

// At file scope.

// Keep track of number of logging requests. static int request\_count;

// Manage threads in this process. static Thread\_Manager thr\_mgr;

// Lock to protect request\_count. static Thread\_Mutex lock;

// Forward declaration.

static void \*logging\_handler (void \*);

// Port number to listen on for requests. static const int logging\_port = 10000;

// Main driver function for the multi-threaded

// logging server. Some error handling has been

// omitted to save space in the example. int main (int argc, char \*argv[])

{

// Internet address of server. INET\_Addr addr (port);

// Passive-mode acceptor object. SOCK\_Acceptor server (addr);

SOCK\_Stream new\_stream;

// Wait for a connection from a client. for (;;)

{

// Accept a connection from a client. server.accept (new\_stream);

// Get the underlying handle.

SOCKET h = new\_stream.get\_handle ();

// Spawn off a thread-per-connection. thr\_mgr.spawn (logging\_handler,

reinterpret\_cast <void \*> (h), THR\_DETACHED);

}

}

logging\_handler 函数运行在单独的线程控制中，也就是，每个相连客户有一个线程。它在各个连接上接收

并处理日志记录，如下所示：

// Entry point that processes logging records for

// one client connection.

void \*logging\_handler (void \*arg)

{

SOCKET h = reinterpret\_cast <SOCKET> (arg);

// Create a <SOCK\_Stream> object from SOCKET <h>. SOCK\_Stream stream (h);

for (;;)

{

UINT\_32 len; // Ensure a 32-bit quantity. char log\_record[LOG\_RECORD\_MAX];

// The first <recv\_n> reads the length

// (stored as a 32-bit integer) of

// adjacent logging record. This code

// handles "short-<recv>s". ssize\_t n = stream.recv\_n

(reinterpret\_cast <char \*> (&len), sizeof len);

// Bail out if we’re shutdown or

// errors occur unexpectedly. if (n <= 0) break;

len = ntohl (len); // Convert byte-ordering. if (len > LOG\_RECORD\_MAX) break;

// The second <recv\_n> then reads <len>

// bytes to obtain the actual record.

// This code handles "short-<recv>s". n = stream.recv\_n (log\_record, len);

// Bail out if we’re shutdown or

// errors occur unexpectedly. if (n <= 0) break;

{

// Constructor of Guard automatically

// acquires the lock. Guard<Thread\_Mutex> mon (lock);

// Execute following two statements in a

// critical section to avoid race conditions

// and scrambled output, respectively.

++request\_count; // Count # of requests

if (write (STDOUT, log\_record, len) == -1)

break;

// Destructor of Guard automatically

// releases the lock, regardless of

// how we exit this block!

}

}

// Destructor of <stream> automatically

// closes down <h>. return 0;

}

注意上面的代码是怎样解决 2.2.2 所示代码的各种问题的。例如，SOCK\_Stream 和 Guard 的析构器会

分别关闭 socket 句柄和释放 Thread\_Mutex，而不管代码块是怎样退出的。同样地，此代码要容易移植和 维护得多，因为它没有使用平台特有的 API。

**2.2.10** 已知应用

本论文中的例子聚焦于并发网络编程。但是，包装外观模式已被应用到其他的许多领域，比如 GUI

框架和数据库类库。下面是包装外观模式的一些广为人知的应用：

**Microsoft Foundation Class**（**MFC**）：MFC 提供一组封装大多数低级 C Win32 API 的包装外观，主要 集中于提供实现 Microsoft 文档/模板体系结构的 GUI 组件。

**ACE** 框架：2.2.8 描述的互斥体、线程和 socket 的包装外观分别基于 ACE 框架中的组件[7]：ACE\_Threa d\_Mutex、ACE\_Thread\_Manager 和 ACE\_SOCK\*类。

**Rogue Wave** 类库：Rogue Wave 的 Net.h++和 Threads.h++类库在许多 OS 平台上实现了 socket、线程和 同步机制的包装外观。

**ObjectSpace System<Toolkit>**：该工具包也提供了 socket、线程和同步机制的包装外观。

**Java** 虚拟机和 **Java** 基础类库：Java 虚拟机（JVM）和各种 Java 基础类库，比如 AWT 和 Swing，提供了 一组封装大多数低级的本地 OS 系统调用和 GUI API 的包装外观。

**2.2.11** 效果

包装外观模式提供以下好处：

更为简洁和健壮的编程接口：包装外观模式在一组更为简洁的 OO 类方法中封装许多低级函数。这减少 了使用低级函数和数据结构开发应用的枯燥性，从而降低了发生编程错误的潜在可能性。

改善应用可移植性和可维护性：包装外观类的实现可用以使应用开发者与低级函数和数据结构的不可移 植的方面屏蔽开来。而且，通过用基于逻辑设计实体（比如基类、子类，以及它们的关系）的应用配置

策略取代基于物理设计实体（比如文件和#ifdef）的策略[6]，包装外观模式改善了软件结构。一般而言， 根据应用的逻辑设计、而不是物理设计来理解和维护它们要更为容易一些。

改善应用的模块性、可复用性和可配置性：通过使用像继承和参数化类型这样的 OO 语言特性，包装外 观模式创建的可复用类组件可以一种整体方式被“插入”其他组件，或从中“拔出”。相反，不求助于粗 粒度的 OS 工具，比如链接器或文件系统，替换成组的函数要难得多。

包装外观模式有以下缺点： 额外的间接性（**Indirection**）：与直接使用低级的函数和数据结构相比，包装外观模式可能带来额外的间

接。但是，支持内联的语言，比如 C++，可以无需显著的开销而实现该模式，因为编译器可以内联用于

实现包装外观的方法调用。

**2.2.12** 参见

包装外观模式与外观模式是类似的[8]。外观模式的意图是简化子系统的接口。包装外观模式的意图 则更为具体：它提供简洁、健壮、可移植和可维护的类接口，封装低级的函数和数据结构，比如本地 OS 互斥体、socket、线程和 GUI C 语言 API。一般而言，外观将复杂的类关系隐藏在更简单的 API 后面， 而包装外观将复杂的函数和数据结构关系隐藏在更丰富的类 API 后面。

如果动态分派被用于实现包装外观方法，包装外观模式可使用桥接模式[8]来实现；包装外观方法在 桥接模式中扮演抽象（Abstraction）角色。

**2.3** 结束语

本论文描述包装外观模式，并给出了详细的例子演示怎样使用它。在本论文中描述的 ACE 包装外观 组件的实现可在 ACE[7]软件发布中自由获取（URL：[http://www.cs.wustl.edu/~schmidt/ACE.html](http://www.cs.wustl.edu/%7Eschmidt/ACE.html)）。该发 布含有在圣路易斯华盛顿大学开发的完整的 C++源码、文档和测试例子驱动程序。目前 ACE 正在用于许 多公司（像 Bellcore、波音、DEC、爱立信、柯达、朗讯、摩托罗拉、SAIC 和西门子）的通信软件项目 中。

感谢

感谢 Hans Rohnert、Regine Meunier、Michael Stal、Christa Schwanninger、Frank Buschmann 和 Br ad Appleton，他们的大量意见极大地改善了包装外观模式描述的形式和内容。

参考文献

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A Syst em of Pattern*s. Wiley and Sons, 1996.

[2] W.R.Stevens,*UNIX Network Programming, First Editio*n. Englewood Cliffs, NJ: Prentice Hall, 1990.

[3] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalin-giah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Wi lliams, “Beyond Multiprocessing... Multithreading the SunOS Ker-nel,” in *Proceedings of the Summer USENIX Conferenc*e, (San Antonio, Texas), June 1992.

[4] W.R.Stevens,*UNIX Network Programming, Second Editio*n. Englewood Cliffs, NJ: Prentice Hall, 1997.

[5] D. C. Schmidt, “IPC SAP: An Object-Oriented Interface to Interprocess Communication Services,” *C++ Repor*t,vol.4, November/December 1992.

[6] J. Lakos, *Large-scale Software Development with C+*+. Reading, MA: Addison-Wesley, 1995.

[7] D. C. Schmidt, “ACE: an Object-Oriented Framework for Developing Distributed Applications,” in *Proceedings of the*

*6th USENIX C++ Technical Conferenc*e, (Cambridge, Mas-sachusetts), USENIX Association, April 1994.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Pat-terns: Elements of Reusable Object-Oriented Softwar*e. Read-ing, MA: Addison-Wesley, 1995.

[9] D. C. Schmidt, “Acceptor and Connector: Design Patterns for Initializing Communication Services,” in *Pattern Langu ages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[10] H. Mueller, “Patterns for Handling Exception Handling Suc-cessfully,” *C++ Repor*t, vol. 8, Jan. 1996.

[11] D. C. Schmidt, T. Harrison, and N. Pryce, “Thread-Specific Storage – An Object Behavioral Pattern for Accessing per-Thread State Efficiently,” *C++ Repor*t,vol.9,Novem-ber/December 1997.

[12] Bjarne Stroustrup, *The C++ Programming Language, 3rd Editio*n. Addison-Wesley, 1998.

第 **3** 章 **IPC SAP**：用于高效、可移植和灵活的网络编程的 **C++**包装

Douglas C. Schmidt

本论文的一个扩展版本[1]（含有在以太网和ATM网络上的性能评测）可在[http://www.cs.wustl.edu/sc hmidt/COOTS-95.ps.Z](http://www.cs.wustl.edu/schmidt/COOTS-95.ps.Z)处获取。

**3.1** 介绍

本论文描述采用 C++包装类来封装 OS 进程间通信（IPC）机制的面向对象（OO）技术，并聚焦于 A CE 框架[2]中的 IPC SAP 组件所提供的 C++包装。ACE 是一组可复用 C++类库和 OO 框架组件，它们简 化了可移植、高性能和实时通信软件的开发。IPC SAP 是 ACE 中的一种组件，它提供了一个 OO 网络编 程接口族来封装 socket 接口[3]、系统 V 传输层接口（TLI）[4]、SVR4 STREAM 管道[5]、UNIX FIFO[6] 和 Windows NT 命名管道[7]。

IPC SAP 中的 C++包装将开发者及应用与 OS 的本地和远地 IPC 机制的不可移植的细节屏蔽开。IPC SAP 封装的 IPC 机制包括标准的面向连接的和无连接的协议，比如在 UNIX/POSIX、Win32 和实时操作 系统中可用的 TCP、UDP 和 IPX/SPX。IPC SAP 利用 OO 技术和 C++特性来提供一组丰富的组件，简化 了高效、可移植和灵活的通信软件的开发。

本论文被组织如下：3.2 概述用于编写通信软件的抽象层级；3.3 描述现有的网络编程接口；3.4 概述 它们的局限；3.5 介绍 IPC SAP 的 OO 设计和实现，并解释它是怎样克服现有网络编程接口的局限的；3.

6 详细检查 socket、TLI、STREAM 管道和 FIFO 的 C++包装；3.7 演示若干例子，使用 IPC SAP 来实现 客户/服务器流式应用；3.8 讨论指导 IPC SAP 的设计的原则；3.9 则总结使用 C++来为本地 OS 接口开发 OO 包装的优点和缺点。

**3.2** 网络编程接口综述

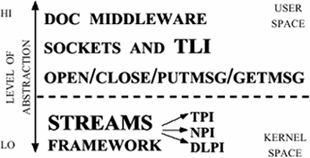


图 3-1 网络编程的抽象层次 编写健壮、可扩展和高效的通信软件是困难的。开发者必须掌握许多复杂的 OS 和通信概念，比如：

－网络寻址和服务标识。

－表示转换（比如加密、压缩和有可选处理器字节序的异种终端系统间的网络字节序转换）。

－进程和线程创建及同步。

－本地和远地进程间通信（IPC）机制的系统调用和库函数接口。

许多编程工具和接口已被创建用来帮助简化通信软件的开发。图 3-1 演示在当代的 OS 平台（比如 U NIX 和 Win32）上可用的 IPC 接口。如图所示，应用可以访问本地和远地 IPC 的网络编程接口的若干层 次。这一部分的余下部分概述每一抽象层，范围从高级分布式对象计算（DOC）中间件到用户级网络编 程接口，再到低级的内核编程接口。

**3.2.1 DOC** 中间件

以一种“请求—响应”方式来与客户交换数据的应用常常使用分布式对象计算（DOC）中间件来开 发。DOC 中间件的宽泛定义包括对象请求代理（ORB），像 CORBA[8]和 Microsoft 的 DCOM[9]；以及面 向消息的中间件，像 Mqseries。DOC 中间件使分布式应用开发的许多麻烦而易错的方面得以自动完成， 包括：

－认证、授权和数据安全；

－服务定位和绑定；

－服务登记和启用；

－事件多路分离和分派；

－在像 TCP 这样的面向字节流的通信协议之上实现消息帧；

－涉及网络字节序和参数整编（marshaling）的表示转换问题。

此外，DOC 中间件还提供一组高级工具，比如 IDL 编译器和名字服务，将开发者与较低级的 OS 系 统调用（它们在网络上传输和接收包）的复杂性屏蔽开。

**3.2.2** 用户级网络编程接口

DOC 中间件通常建构在网络编程接口之上，比如 socket[3]、TLI[4]，或 Windows NT 命名管道。与 较高级的 DOC 中间件相比，通过用户级网络编程接口来开发应用有若干优点：

－最小化不必要功能的时间和空间开销：应用可以忽略不必要的功能，比如 ASCII 数据或内存区域的表 示层转换。

－允许对行为进行细粒度控制：网络编程接口使得对行为的控制粒度更为精细，比如允许多点传输和信

号驱动的异步 I/O。

－增强可移植性：像 socket 这样的网络编程接口可用于广泛的 OS 平台，而 DOC 中间件则不是这样。

对于一类特定的被称为“流式应用”[10]的应用，DOC 中间件提供的请求－响应和“单路”通信机 制并不特别适用。流式应用的特征是高带宽、无类型字节流或相对简单的数据类型的长持续时间的通信， 对通信性能有着严格的要求。交互式电话会议、医学成像和视频点播是流式应用的范例。

流式应用服务质量需求（QoS）常常不能忍受 DOC 中间件所带来的性能开销[11]。这样的开销源于 未优化的表示格式转换、未优化的内存管理、低效的接收者端多路分离、停－等流控制、同步的发送端 方法请求，以及非自适配的重发定时器方案。传统上，满足流式应用的需求涉及到对像 socket[3]或 TLI[4] 这样的网络编程接口的直接访问。

**3.2.3** 内核级网络编程接口

在 OS 内核的通信子系统中有较低级的网络编程接口。例如，SVR4 putmsg 和 getmsg 系统调用可用 于直接访问系统 V STREAMS[14]中的传输供应者接口（TPI）[12]和数据链路供应者接口（DLPI）[13]。

还有可能开发像路由器或网络文件系统这样的网络服务，它们整个地驻留在 OS 内核中[5]。但是， 在这一级进行编程通常不能在不同的 OS 平台间移植。而且，甚至也不能在同一 OS 的不同版本间移植。

**3.2.4** 评估

使用用户级或内核级网络编程接口、而不是 DOC 中间件，通常要更难进行编程。像 socket 和 TLI 这样的传统网络编程库缺少类型安全、可移植、可重入和可扩展的接口。例如，socket 端点通过弱类型的 描述符实现，从而增加了在运行时发生微妙错误的潜在可能性[15]。

本文中描述的 IPC SAP 组件通过封装网络编程接口的大量复杂性，在设计空间中提供了一个“中点”。 IPC SAP 的目标是提高通信软件的正确性、易用性和可移植性*/*可复用性，而又不损害它的性能。IPC S AP 与 ACE 框架[2]一起发布，并被用于许多公司的商业项目中，包括 Bellcore、波音、朗讯、摩托罗拉、 Nortel、SAIC 和西门子，等等。

**3.3** 网络编程接口考察

这一部分考察像 socket 和 TLI 这样的传统网络编程接口的行为和局限。

**3.3.1** 背景

在许多操作系统中，比如 UNIX 和 Win32，通信协议栈驻留在 OS 内核的保护地址空间中。运行在用 户地址空间中的应用程序通过像 socket、TLI 或 Win32 命名管道这样的接口来访问驻留内核的协议栈。这 些接口对本地和远地的通信端点这样来进行管理：允许应用打开到远地主机的连接、磋商和启用/禁用特 定的选项、交换数据，以及在传输完成时关闭全部或部分连接。

socket 和 TLI 松散地建模在 UNIX 文件 I/O 接口之上，后者定义了 open、read、write、close、ioctl、 lseek 和 select 函数[14]。但是，socket 和 TLI 还提供了额外的功能，没有直接被标准的 UNIX 文件 I/O 接 口所支持。这些额外的功能源于文件 I/O 和网络 I/O 之间语法和语义的差异。例如，在分布式环境中，U NIX 系统用于标识文件的路径名并非是全局唯一的。因此，采用了一种不同的命名方案（比如 IP 主机地 址）来唯一地标识网络应用。

socket 和 TLI 接口提供类似的功能。它们支持一种多通信域[3]的通用接口。域指定协议族和地址族。 每个协议族都含有一个协议栈，实现域中特定的通信类型。常用的协议栈提供可靠、双向、面向连接的 消息和流的服务（例如，像 TCP、TP4 和 SPX 这样的协议），以及不可靠、无连接的数据报服务（例如， 像 UDP、CLNP 和 IPX 这样的协议）。

地址族定义地址格式（例如，地址的字节长度、字段的数目和类型和字段顺序）以及一组驻留内核 的对地址格式进行解释的函数（例如，决定一个 IP 数据报要发到哪个子网）。

3.3.2 给出了 socket 综述，3.3.3 简要描述了 TLI，3.3.4 讨论 STREAM 管道，而 3.3.5 讨论 UNIX FIF O。对这些接口的完整讨论超出了本论文的范围（更多详情参见[5, 3, 7, 6, 16]）。

**3.3.2 socket** 接口

socket 接口最初是在 BSD UNIX 中开发的，用以提供 TCP/IP 协议组[3]的接口。从应用的视点来看， socket 是本地的通信端点，与驻留在本地或远地的地址绑定在一起。socket 可通过句柄（也称为描述符） 来访问。

在 UNIX 中，socket 句柄与其他句柄共享同一个名字空间，例如，文件、管道和终端设备句柄。句柄 提供一种封装机制，将应用与内部的 OS 数据结构的知识屏蔽开。句柄标识特定的由 OS 维护的通信端点。

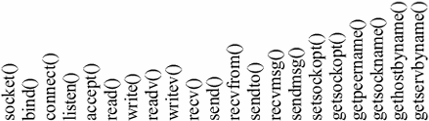


图 3-2 socket 接口中的函数

socket 接口如图 3-2 所示。该接口含有大约两打的系统调用，可分为以下类型： 本地管理：socket 接口为管理本地上下文信息提供以下函数：

－socket：分配最小的未用 socket 句柄；

－bind：将 socket 句柄与本地或远地地址相关联；

－getsockname 和 getpeername：分别确定 socket 所连接的本地或远地地址；

－close：释放 socket 句柄，使它可用于后面的复用。

连接建立和连接终止：socket 接口为建立和终止连接提供以下函数：

－connect：客户通常使用 connect 来主动地与服务器建立连接；

－listen：服务器使用 listen 来指示它想要被动地侦听进入的客户连接请求；

－accept：服务器使用 accept 来创建新的通信端点，以为客户服务；

－shutdown：有选择地终止一个双向连接的读端和/或写端流。

数据传输机制：socket 接口提供以下函数来发送和接收数据：

－read/write：通过特定句柄接收和传输数据缓冲区；

－send/recv：与 read/write 类似，但它们提供一个额外的参数来控制特定的 socket 特有操作（比如交换“紧 急”数据，或“偷看”接收队列中的数据，而又不把它从队列中移除）；

－sendto/recvfrom：交换无连接数据报；

－readv/writev：分别支持“分散读”和“集中写”语义（这些操作优化用户/内核模式切换并简化内存管 理）；

－sendmsg/recvmsg：通用函数，包含了所有其他数据传输函数的行为。对于 UNIX 域的 socket，sendmsg

和 recvmsg 函数还提供在同一主机的任意进程间传递“访问权限”（比如打开文件句柄）的能力。

注意这些接口也可被用于其他类型的 I/O，比如文件和终端。 选项（**option**）管理：socket 接口定义以下函数，允许用户改变 socket 行为的缺省语义：

－setsockopt 和 getsockopt：修改或查询在协议栈不同层次中的选项。选项包括多点传送、广播，以及设 置/获取发送和接收传输缓冲区的大小；

－fcntl 和 ioctl：是 UNIX 系统调用，使在 socket 上能够进行异步 I/O、非阻塞 I/O，以及紧急消息递送。

除了上面描述的 socket 函数，通信软件还可使用以下标准库函数和系统调用：

－gethostbyname 和 gethostbyaddr：处理网络寻址的多种情况，比如映射主机名到 IP 地址；

－getservbyname：通过服务的端口号或人类可读的名字来对它们进行标识；

－ntohl、ntohs、htonl、htons：执行网络字节序转换；

－select：在成组的打开的句柄上执行基于 I/O 和基于定时器的事件多路分离。

**3.3.3 TLI** 接口

TLI 是访问通信协议栈的一种可选接口。基本上，TLI 提供一组和 socket 一样的服务。但是，它更加 强调使应用与底层传输供应者的细节屏蔽开来。[5]详细地讨论 TLI。

**3.3.4 STREAM** 管道

STREAM 管道是对原始的 UNIX 管道机制的增强。早先的 UNIX 管道提供单一的从作者端点到读者 端点的单向字节流。STREAM 管道支持在执行在同一主机上的进程和/或线程间进行双向的字节流和按优 先级排序的消息的递送[16]。尽管 pipe 系统调用接口保持不变，STREAM 管道还提供了额外的功能，大 致等价于 UNIX 域的 SOCK\_STREAM socket。但是它们比 UNIX 域的 socket 要更灵活一些，因为它们使 STREAM 模块可被“压入”或是“弹出”管道端点。

缺省地，流管道仅在它的两个端点间提供单一数据通道。因此，如果多个发送者向管道写入，所有 的消息都被放置到同一个通信通道中。这常常太过受限，因为多路分离单个通道上来自多个客户的数据 必须进行人工编程。例如，每个消息都必须包含一个标识符，使接收者能够确定是哪一个发送者传输的 消息。通过使用已安装的（mounted）STREAM 管道和 connld 模块[17]，应用可以将一个单独的非多路复 用的 I/O 通道专用于服务器和客户的每一实例之间。

STREAM 管道和 connld 的工作方式如下：服务器调用 pipe 系统调用，创建双向通信端点。Fattach 系统调用可以将管道句柄安装（mount）到 UNIX 文件系统中的指定位置。通过将 connld STREAM 模块 压入 STREAM 管道的已安装的一端，就可以创建服务器应用。在运行服务器的同一主机上运行的客户应 用随即打开与已安装管道相关联的文件。在这一点，connld 模块确保客户和服务器分别收到一个唯一的 I

/O 句柄，标识一个非多路复用、双向的通信信道。

**3.3.5 FIFO** 接口

UNIX FIFO（也称为命名管道[6]）是 STREAM 管道的受限形式。不像 STREAM 管道，FIFO 仅提 供单向的、从一或多个发送者到单个接收者的数据通道。而且，来自不同发送者的消息都被放入同一个 通信通道中。因此，必须在每个消息中明确地包括某种类型的多路分离标识符，以使接收者能够确定是 哪一个发送者传输的消息。

SVR4 UNIX 中基于 STREAM 的 FIFO 实现同时提供消息和字节流递送语义。相反，一些早期版本 的 UNIX（比如 SVR3 和 SunOS 4.x）仅提供面向字节流的 FIFO。因此，除非总是使用定长消息，每个 经由 FIFO 发送的消息必须通过某种形式的字节计数或特殊结束符来进行区分，从而使接收者能够从 FIF O 字节流中提取消息。FIFO 在[5, 6, 16]中进一步描述。

**3.4** 问题：现有 **IPC** 接口的局限

socket、TLI、STREAM 管道和 FIFO 为访问本地和远地 IPC 机制提供了广泛的接口。但是这些接口 都有若干局限。下面的讨论聚焦于 socket 接口的局限，但是其中的大多数也适用于其他网络编程接口。

高错误可能性：在 UNIX 和 Win32 中，socket、文件、管道、终端和其他设备的句柄是用“弱类型”的整

数或指针值来标识的。这样的弱类型检查会导致微妙的运行时错误。例如，socket 接口无法确保用于不同 通信角色（比如主动 vs. 被动连接建立，或数据报 vs. 流通信）的 socket 函数的正确使用。而且，编译 器无法检测或阻止句柄的错误使用，因为句柄是弱类型的。因而，可能会不正确地对句柄进行操作，例 如，在为建立连接而设置的句柄上调用数据传输操作。

#include <sys/types.h>

#include <sys/socket.h>

const int PORT\_NUM = 10000;

int buggy\_echo\_server (void)

{

sockaddr s\_addr;

int length; // (1) uninitialized variable. char buf[BUFSIZ];

int s\_fd, n\_fd;

// Create a local endpoint of communication.

if (s\_fd = socket (PF\_UNIX, SOCK\_DGRAM, 0) == -1)

return -1;

// Set up the address information to become a server.

// (2) forgot to "zero out" structure first... s\_addr.sin\_family = AF\_INET;

// (3) used the wrong address family ... s\_addr.sin\_port = PORT\_NUM;

// (4) forgot to use htons() on PORT\_NUM... s\_addr.sin\_addr.s\_addr = INADDR\_ANY;

if (bind (s\_fd, (sockaddr \*) &s\_addr, sizeof s\_addr) == -1)

perror ("bind"), exit (1);

// (5) forgot to call listen()

// Create a new endpoint of communication.

// (6) doesn’t make sense to accept a SOCK\_DGRAM!

if (n\_fd = accept (s\_fd, &s\_addr, &length) == -1)

{

// (7) Omitted a crucial set of parens... int n;

// (8) doesn’t make sense to read from the s\_fd!

while ((n = read (s\_fd, buf, sizeof buf)) > 0)

// (9) forgot to check for "short-writes" write (n\_fd, buf, n);

// Remainder omitted...

}

}

图 3-3 臭虫成灾的 Echo 服务器

图 3-3 描述下列在使用 socket 接口时发生的微妙和“过于常见”的错误：

1. 忘记将 accept 的 len 参数初始化为 struct sockaddr\_in 的大小；

2. 忘记将 socket 地址结构中的所有字节初始化为“0”；

3. 使用了与 socket 的协议族相矛盾的地址族类型；

4. 忽略了使用 htons 库函数来将端口号从主机字节序转换到网络字节序，反之亦然。

5. 创建被动模式的 SOCK\_STREAM socket 时遗漏了 listen 系统调用；

6. 对 SOCK\_DGRAM socket 使用了 accept 函数；

7. 在赋值表达式中错误地遗漏了一组关键的括号；

8. 试图从被动模式 socket 中读，而这样的 socket 只能用于接受连接；

9. 没有能适当地检测和处理由于缓冲而发生的“短写”（short-writes）。

上面所列问题中的一些是 C 的经典问题。例如，如果遗漏了下面这个表达式中的括号

if (n\_fd = accept (s\_fd, &s\_addr, &length) == -1)

n\_fd 的值将总是被设为 0 或者 1（取决于 accept()是否等于-1）。

一个更深的问题是 C 数据结构缺乏足够的抽象。例如，通用的 sockaddr 地址结构使得开发者必须使 用强制类型转换来提供 Internet 域和 UNIX 域地址的一种继承形式。这些“子类”地址结构，sockaddr\_i n 和 sockaddr\_un，分别对 sockaddr“基类”进行重定义。

一般而言，强制类型转换的使用，与弱类型的、基于句柄的 socket 接口一起，使得编译器很难在编 译时检测错误。相反，错误检查被推延到运行时，这使得错误处理变得更为复杂，并且降低了应用的健 壮性。

复杂的接口：socket 提供了单一接口来支持多种协议族，像 TCP/IP、IPX/SPX、ISO OSI 和 UNIX 域的 s ocket。socket 接口含有许多函数，支持不同的通信角色（比如主动 vs. 被动连接建立）、通信优化（比如 在单个系统调用中发送多个缓冲区的 writev），以及用于不常使用的操作的选项，比如广播、多点传送、 异步 I/O 和紧急数据递送。

尽管 socket 将这些功能组合进一个通用的接口，所得到的机制仍然是复杂而又难以掌握的。这样的 复杂性源于 socket 接口过于宽泛的和一维的（one-dimensional）设计。例如，如图 3-2 所示，所有函数都 出现在单一的抽象层中。这样的设计增加了正确学习和使用 socket 所需的努力。这样，程序员必须理解 整个 socket 接口，即使他们只使用其中一部分。

但是，如果仔细地检查 socket，很清楚该接口可以被分解为下面三个函数簇：

1. 通信服务类型：也就是，流 vs. 数据报 vs. 有连接的数据报；

2. 通信角色：也就是，主动的 vs. 被动的（客户通常是主动的，而服务器通常是被动的）

3. 通信域：也就是，本地 vs. 本地/远地。

图 3-4 根据这三个标准来对相关的 socket 函数进行分类：

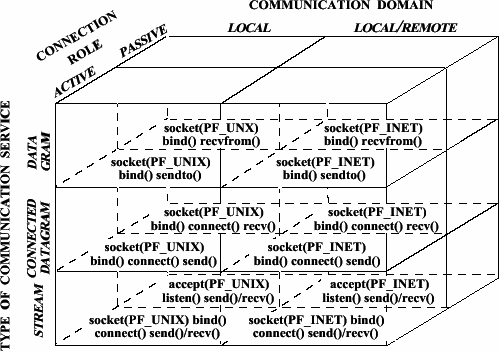


图 3-4 socket 的各个维度

但是，因为接口是一维的，这样自然的分类被弄得含混不清。3.6 演示了怎样将此分类重新构造为一 个类层次，以简化 socket 接口并增强通信软件的类型安全性。

不统一：socket 接口的另一问题是它的若干打函数缺乏统一的命名习惯。不统一的命名使得开发者很难确 定 socket 接口的范围。例如，socket、bind、accept 和 connect 之间的相关并不显而易见。其他网络编程接 口通过在每个函数前面添加公共前缀来解决这一问题。例如，在 TLI 库的每个函数前都有 t\_前缀。

但是，TLI 接口也含有有着过于复杂的语义的操作。例如，不像 socket，TLI 选项处理接口没有以一 种标准的方式来规定。这使得开发者很难编写可移植的应用来访问标准的 TCP/IP 选项。同样地，在 qlen

> 1 的并发服务器中，需要使用微妙的应用级代码来处理 t\_listen 和 t\_accept 的非直观和易错的行为[5]。

**3.5** 解决方案：**IPC SAP C++**包装

**3.5.1** 综述

IPC SAP 封装常用的基于句柄的 IPC 接口，比如 socket、TLI、STREAM 管道和 FIFO。如图 3-5 所 示，IPC SAP 被设计为类属的一座“森林”，包括 SOCK SAP（封装 socket）、TLI SAP（封装 TLI 接口）、 SPIPE SAP（封装 UNIX SVR4 STREAM 管道接口），以及 FIFO SAP（封装 UNIX FIFO 接口）。

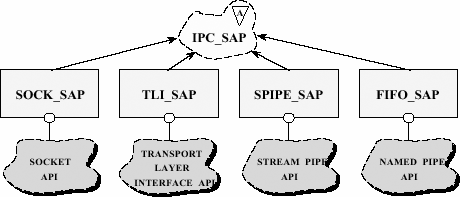


图 3-5 IPC SAP 类属关系

每个类属都被组织为继承层次。所有子类都给现有 IPC 机制的子集提供定义良好的接口。在一个层 次中的所有子类共同地包含了一种特定通信抽象（比如 Internet 域或 UNIX 域的协议族）的全部功能。这 一部分描述 IPC SAP 的设计目标，概述它的类属，并讨论在其 OO 设计之下的法则。

**3.5.2 IPC SAP** 设计目标

IPC SAP 被设计用于改善通信软件的正确性、易学性和易用性、可移植性，以及可复用性，同时维 持高水平的性能和功能。这一部分讨论 IPC SAP 是怎样实现这些目标的。

**3.5.2.1** 提高正确性

socket 的若干问题都与它的弱类型检查有关。通过只允许对类的实例进行“类型安全”的操作，IPC SAP 提高了网络应用代码的正确性。为强制实施类型安全性，IPC SAP 确保它的所有对象都通过构造器 来适当地初始化。此外，对 IPC SAP 对象只能进行良好定义的操作。

IPC SAP 还被设计用于防止偶然的类型安全性违例。例如。SOCK SAP 类属中的组件可防止偶然地 对数据报对象进行面向连接的操作。因此，不可能在数据报对象上调用 accept 方法，在连接器和接受器 工厂对象上接收（recv）或发送（send）数据，或是在面向连接的对象上调用 sendto 方法。

因为 IPC SAP 类是强类型的，任何执行非法操作的企图都会在编译时、而非运行时被拒绝。图 3-14 所示的 buggy\_echo\_server 的 SOCK SAP 修正版对这一点进行了演示。此例更正了图 3-3 中所标识出的所 有 socket 问题。

**3.5.2.2** 增强易学性和易用性

简化常用 IPC 操作的使用是一个与正确性有关的目标。通过提供更简单的接口，开发者能够把注意 力集中在编写应用上，而不是与低级网络代码搅在一起。一般而言，IPC SAP 这样来简化它的网络编程 接口：

提供辅助类，使应用与易错细节相屏蔽：例如，IPC SAP 含有如图 3-6 所示的 Addr 类层次6。该层次通 过类型安全的 C++接口来支持若干不同的网络寻址格式。Addr 层次消除了若干常见的编程错误，这些错 误都与直接使用基于 C 的 struct sockaddr 数据结构有关系。例如，不再有可能忘记把 sockaddr 地址结构 清零。

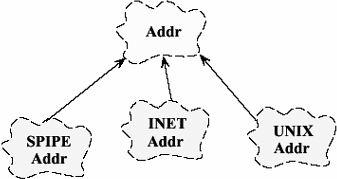


图 3-6 IPC SAP 地址类层次

组合若干操作，以形成单一操作：例如，ACE\_SOCK\_Acceptor 是用于被动连接建立的工厂。它的构造器 执行创建被动模式服务器端点所需的多个 socket 系统调用（比如 socket、bind 和 listen）。

为典型的方法参数值提供缺省参数：例如，accept 的寻址参数常为 NULL 指针。为简化编程，这些值在 S OCK\_Acceptor::accept 中作为 C++缺省参数被给出，以使程序员不必显式地提供它们。

利用 **traits**（特性）来传达“元类”信息：例如，所有 IPC SAP 类都含有一组统一的 traits。这些 traits 进行类型定义，以指定与各自的 IPC SAP 类型相关联的地址类（例如，ACE\_INET\_Addr）和/或流类（例 如，ACE\_TLI\_Stream）。如下所示：

class ACE\_SOCK\_Connector

{

public:

// Traits

typedef ACE\_INET\_Addr PEER\_ADDR;

typedef ACE\_SOCK\_Stream PEER\_STREAM;

// ...

};

class ACE\_TLI\_Connector : public ACE\_SOCK

{

public:

6 为减少混乱，图 1 所有的 ACE IPC SAP 类名的 ACE\_前缀都被省略了。

// Traits

typedef ACE\_INET\_Addr PEER\_ADDR;

typedef ACE\_TLI\_Stream PEER\_STREAM;

//...

};

如 3.7 所示，traits 与 C++参数化类型的联合使用支持一种强大的称为“泛型编程”（generic program

ming）的设计范式 [18]。

**3.5.2.3** 提高可复用性

在 IPC SAP 中使用了基于继承的层次分解，以增加多种 IPC 机制所共享的通用代码的数量。例如， IPC SAP 给像 fcntl 和 ioctl 这样的较低级的 OS 设备控制系统调用提供了一种 C++接口。通过在不同的子 类间共享代码，继承增强了在 IPC SAP 实现中的复用。

例如，IPC SAP 根基类提供的标准方法和数据被其他的派生类所共享。这些共享组件提供句柄和与 其相关的 set/get 方法。此外，还提供了一些方法来在句柄上启用和禁止异步 I/O、非阻塞 I/O，以及紧急 消息递送。

**3.5.2.4** 可移植性

若干 C++特性有助于增强 IPC SAP 的可移植性。例如，IPC SAP 提供一种不依赖于平台的接口，通 过使用 C++模板来改善通信软件的可移植性。如图 3-7 所示，SOCK SAP 和 TLI SAP 类的一个子集提供 了同样的 OO 接口。每个平台可能会拥有不同的用于本地和远地网络编程（例如，socket vs. TLI）的底 层接口。但是，有可能编写出应用，使用两个类中的任何一个来透明地进行参数化。这增强了应用的跨 平台（这些平台可能不同时支持 socket 和 TLI 的平台）可移植性。

通过允许应用被它们所需 IPC 机制的类型参数化，类的使用（相对于独立的函数）有助于简化网络 编程。如 3.8 所讨论的，参数化有助于改善平台间（这些平台支持不同的网络编程接口，比如 socket 或 T LI）的可移植性。

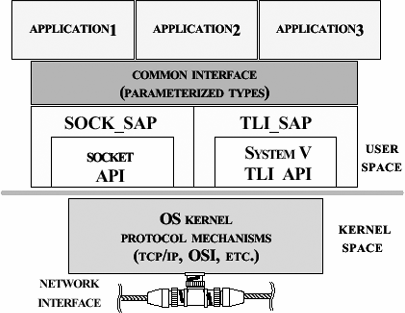


图 3-7 使用模板增强可移植性

**3.5.2.5** 性能

为鼓励开发者用 IPC SAP 替换现有接口，IPC SAP 被设计为能高效地运作。下列技术帮助改善了性 能，而又没有牺牲清晰性和模块性：

使用内联函数：许多 IPC SAP 方法都被指定为 C++内联函数，从而消除了调用 IPC SAP 方法的额外运 行时开销。内联是一种合理的方法，因为每个方法都非常短（平均每个方法大约 3 行）。

避用虚函数：在 IPC SAP 继承层次中没有使用虚函数，从而改善了性能，因为（1）消除了间接的 vtabl e 函数指针分派，以及（2）便利了确实很短而又经常访问的方法（比如发送和接收用户数据）的直接内 联。

**3.6 IPC SAP** 的面向对象设计

这一部分描述组成 IPC SAP 的 C++类属的 OO 设计，并特别强调了 socket 的 SOCK SAP C++包装 的设计。SOCK SAP 已被移植到许多 UNIX 平台、以及 WinSock 网络编程接口上。对这一层面的细节不 感兴趣的读者可能会想跳到 3.8，在其中讨论的是 SOCK SAP 包装类的设计之下的一般法则。

**3.6.1 SOCK SAP** 综述

SOCK SAP 被设计用于克服 3.4 描述的 socket 的局限。使用 C++包装来封装 socket 接口的主要好处 是：

－增强类型安全性：SOCK SAP 在编译时检测许多微妙的应用类型系统违例。

－可移植性：SOCK SAP 提供了可移植的、平台无关的网络编程接口。

－易用性：SOCK SAP 极大地减少了花费在较低级网络编程细节上的应用代码数量和开发工作。

－高效：SOCK SAP 增强了上面所列的软件质量，而又没有牺牲性能[1]。

SOCK SAP 类属为应用提供 Internet 域和 UNIX 域协议族[6]的 OO 接口。SOCK SAP 由大约 12 个 C

++类组成。其总体结构对应于如图 3-8 所示的通信服务、连接角色和通信域的分类。将图 3-4 和图 3-8 进 行比较富有启发意义。图 3-8 中的组件更为简洁，因为它们使用 C++包装在依据继承关联的类中封装了 多种 socket 机制的行为。

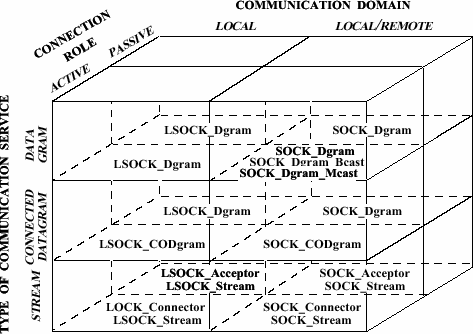


图 3-8 SOCK SAP 类和通信维度的分类

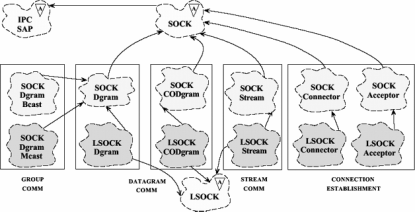


图 3-9 SOCK SAP 类属

SOCK SAP 中的每个类都为组成全部类属的机制的一个子集提供一种抽象接口。多种类型的 Interne t 域和 UNIX 域 socket 的功能是通过继承机制从下面描述的适当的类那里获得的。这些类以及它们的关系 在图 3-9 中通过 Booch 表示法[19]显示7。

应用通过继承或实例化图 3-9 中所示的适当的 SOCK SAP 子类来访问底层的 Internet 域或 UNIX 域 s ocket 类型的功能。如下所述，ACE\_SOCK\* 子类封装 Internet 域的功能，而 ACE\_LSOCK\* 子类封装 U NIX 域的功能。

**3.6.1.1** 基类

IPC SAP、ACE\_SOCK 和 ACE\_LSOCK 类锚定继承层次，并使应用能够进行后续的派生和代码共享。 这些类的对象不能被实例化，因为它们的构造器被声明在类定义的 protected 区域中。

**IPC SAP**：该类是进程间通信机制 C++包装的 IPC SAP 层次的根。它提供所有 IPC SAP（也就是，SOC K SAP、TLI SAP、SPIPE SAP 和 FIFO SAP）组件共有的机制。例如，它提供了方法，可将句柄设置 为非阻塞模式，或者启用异步的、信号驱动的 I/O。

**SOCK**：该类是 SOCK SAP 层次的根。它提供所有其他类共有的机制，比如打开和关闭本地通信端点， 以及处理选项（像选择 socket 队列大小及启用组通信）。

**LSOCK**：该类提供的机制允许应用在本地主机（因而有前缀’L）’

上的不相关进程间发送和接收已打开的

文件句柄。注意系统 V 和 BSD UNIX 都支持这一特性，而 Windows NT 则不支持。其他类从 ACE\_LSO CK 继承以获得这一功能。

7 虚线云表示类，而有向边表示这些类之间的继承关系，例如，ACE\_SOCK\_Stream 继承自 ACE\_SOCK。

SOCK SAP 在网络地址格式和通信语义的基础上区分 ACE\_LSOCK\* 和 ACE\_SOCK\*。特别地，AC E\_LSOCK\* 类使用 UNIX 路径名作为地址，并且仅允许机器内的 IPC。而另一方面，ACE\_SOCK\* 类使 用 Internet 协议（IP）地址和端口号，并同时允许机器内和机器间的 IPC。

**3.6.1.2** 连接建立

客户和服务器间的不对称连接角色是通信软件的典型情况。通常，服务器被动地侦听客户主动发起 的连接[20]。下面的面向连接的 SOCK SAP 类捕捉了被动/主动的连接建立的结构和数据传输关系：

**ACE\_SOCK\_Acceptor** 和 **ACE\_LSOCK\_Acceptor**：这两个类是被动地建立新通信端点、以响应主动连接 请求的工厂[21]。两者分别生成 ACE\_SOCK\_Stream 和 ACE\_LSOCK\_Stream 连接端点对象。

**ACE\_SOCK\_Connector** 和 **ACE\_LSOCK\_Connector**：这两个类是主动地建立新通信端点的工厂。它们 建立与远地端点的连接，并在连接建立时生成适当的\*Stream 对象。连接可以被同步地或异步地发起。两 个工厂分别生成 ACE\_SOCK\_Stream 和 ACE\_LSOCK\_Stream 连接端点对象。

注意\*Acceptor 和 Connector 类不提供发送和接收数据的方法。相反，它们是生成下面描述的\*Stream 数据传输对象的工厂。使用强类型的工厂接口可以在编译时检测和防止本地和非本地\*Stream 对象的偶然 误用。相反，socket 接口仅能在运行时检测这些类型不匹配。

**3.6.1.3** 流通信

尽管建立连接需要区分主动和被动角色，一旦连接建立，数据就可以根据端点所用的协议以任意的 顺序来进行交换。SOCK SAP 在下面的类中隔离了数据传输行为：

**ACE\_SOCK\_Stream** 和 **ACE\_LSOCK\_Stream**：这些类由上面描述的\*Acceptor 或\*Connector 工厂创建。

\*Stream 类为在两个进程间传输数据提供机制。ACE\_LSOCK\_Stream 对象在同一主机上的进程间交换数 据；ACE\_SOCK\_Stream 对象在可驻留在不同主机上的进程间交换数据。

被重载的 send 和 recv \*Stream 方法提供标准的 UNIX write 和 read 语义。因而，send 或 recv 分别读 或写的字节数可能会少于所请求的字节数。这些“短写”（short-writes）或“短读”（short-reads）之所 以发生，是由于 OS 中的缓冲和传输协议中的流控制。为减少编程工作，\*Stream 类提供 send\_n 和 recv\_ n 方法，允许传输和接收正好 n 个字节。另外还提供了“分散读”和“集中写”方法，以高效地同时发送 和接收多个数据缓冲区。

**3.6.1.4** 数据报通信

本论文聚焦于面向连接的流通信，但是，socket 接口也提供无连接的服务，它使用 Internet 协议组中 的 IP 和 UDP 协议。IP 和 UDP 是不可靠的数据报服务，不保证特定的消息会到达它的目的地。无连接服 务被用于那些可容忍一定程度的丢失的应用（比如 rwho 看守[6]）。此外，IP 和 UDP 还提供像 TCP 和 Su n RPC 这样的较高级可靠协议的基础。

SOCK SAP socket 包装通过下面的类来封装 socket 数据报通信：

**ACE\_SOCK\_Dgram** 和 **ACE\_LSOCK\_Dgram**：这两个类为在运行在本地和/或远地主机上的进程间交换 数据报提供机制。不像下面描述的有连接数据报，每个 send 和 recv 操作都必须为发送或接收数据报提供 服务地址。ACE\_LSOCK\_Dgram 同时继承 ACE\_SOCK\_Dgram 和 ACE\_LSOCK 的所有操作。它仅在同一 主机上的进程间交换数据报。而 ACE\_SOCK\_Dgram 类可以在本地和/或远地主机上的进程间交换数据报。

**ACE\_SOCK\_CODgram** 和 **ACE\_LSOCK\_CODgram**：这两个类提供一种“有连接数据报”机制。不像 上面所描述的无连接类，这两个类允许 send 和 recv 操作在交换数据报时省略服务地址。注意有连接数据 报机制只是一种语法上的方便，因为没有其他的语义与数据传输相关联（也就是，数据递送还是不可靠 的）。ACE\_SOCK\_CODgram 的机制从 ACE\_SOCK 基类继承。ACE\_LSOCK\_CODgram 同时继承 ACE\_S OCK\_CODgram 和 ACE\_LSOCK（它提供传递文件句柄的能力）的机制。

**3.6.1.5** 组通信（**Group Communication**）

标准的 TCP 和 UDP 通信是点对点的。但是，有些应用可从提供组通信的更为灵活的递送机制中获益。 因此，下面的类封装了 Internet 协议组提供的多点传送和广播协议：

**ACE\_SOCK\_Dgram\_Mcast**：该类提供的机制用于将 UDP 数据报多点传送给运行在本地子网中的本地和

/或远地主机上的进程。该类的接口支持将数据报多点传送给特定的多点传送组。该类还将开发者与有效 利用多点传送所需的低级细节屏蔽开来。

**ACE\_SOCK\_Dgram\_Bcast**：该类提供的机制用于将 UDP 数据报广播给本地子网中的本地和/或远地主机。 该类的接口支持将数据报广播给（1）所有与主机相连的网络接口，或是（2）一个特定的网络接口。该 类还将开发者与有效利用广播所需的低级细节屏蔽开来。

ACE\_SOCK\_Dgram\_Bcast 类在下面用于将一个消息广播给 LAN 子网中在指定端口号上侦听的所有 服务器：

int main (int argc, char \*argv[])

{

ACE\_SOCK\_Dgram\_Bcast b\_sap (ACE\_Addr::sap\_any);

char \*msg;

u\_short b\_port;

msg = argc > 1 ? argv[1] : "hello world\n";

b\_port = argc > 2 ? atoi (argv[2]) : 12345;

if (b\_sap.send (msg, strlen (msg), b\_port) == -1)

perror ("can’t send broadcast");

return 0;

}

将这个简洁的例子与直接使用 socket 接口实现广播所需的成打的 C 源码行相比较富有启发意义。

**3.6.2** 网络寻址

设计一种高效而通用的网络寻址接口是困难的。困难源于用一种节省空间且统一的接口来表示不同 的网络寻址格式的企图。不同的地址格式要存储以不同大小表示的不同类型的信息。

例如，Internet 域的服务（比如 ftp 或 telnet）用两个字段来标识：（1）一个四字节的 IP 地址（唯一地 标识遍及 Internet 的远地主机），以及（2）一个两字节的端口号（用于将到来的协议数据单元多路分离给 适当的客户或是在远地主机上的服务器进程）。相反，UNIX 域的 socket 通过 UNIX 路径名（长度最多可 以到 108 字节，并只在单个本地主机上有意义）来会合。

现有的由 socket 接口提供的基于 sockaddr 的网络寻址结构是麻烦而易错的。它要求开发者明确地把 地址结构中的所有字节清零。相反，图 3-6 所示的 SOCK SAP 寻址类含有用于操作网络地址的机制。

Addr 基类的构造器确保所有的字段被自动地正确初始化。而且，在不同的地址族间存在的不同的大 小、格式和功能被封装在派生的地址子类中。这使得开发者更容易对网络寻址方案进行扩展、以包括新 的通信域。例如，UNIX Addr 子类与 ACE\_LSOCK\*类相关联，ACE\_INET\_Addr 子类与 ACE\_SOCK\*和 ACE\_TLI\*类相关联，还有 SPIPE Addr 子类与 SPIPE SAP 中的 STREAM 管道包装相关联。

**3.6.3 TLI SAP**

TLI SAP 类属提供系统 V 传输层接口（TLI）的 C++接口。TLI 的 TLI SAP 继承层次几乎与 socket 的 SOCK SAP C++包装类相同。主要的差异是 TLI 和 TLI SAP 没有定义 UNIX 域协议族的接口。通过 使 C++特性（比如缺省参数值和模板）与 tirdwr（read/write 兼容性 STREAM 模块）相联合，开发可在编 译时参数化、以在 socket 或 TLI 网络编程接口上正确运行的应用变得相对直截了当了。

下面的代码演示怎样应用模板来参数化应用所使用的 IPC 机制。该代码是从[22]描述的分布式日志工 具中提取的。在下面的代码中，用一种特定类型的网络编程接口和相应的协议地址类对一个派生自 Event Handler 的子类进行了参数化：

// Logging\_Handler header file. template <class PEER\_STREAM>

class Logging\_Handler : public Event\_Handler

{

public:

Logging\_Handler (void);

virtual ˜Logging\_Handler (void);

virtual int handle\_close (int); virtual int handle\_input (int); virtual int get\_handle (void) const

{

return this->xport\_sap.get\_handle ();

}

protected:

PEER\_STREAM xport\_sap;

};

取决于底层 OS 平台（比如说是基于 BSD 的 SunOS 4.x，还是基于系统 V 的 SunOS 5.x）的特定属 性，日志应用可以实例化 Client Handler 类，以使用 SOCK SAP 或 TLI SAP。如下所示：

#if defined (MT\_SAFE\_SOCKETS)

typedef ACE\_SOCK\_Stream PEER\_STREAM;

#else

typedef ACE\_TLI\_Stream PEER\_STREAM;

// Logging application.

#endif // MT\_SAFE\_SOCKETS.

class Logging\_Handler :

public Logging\_Handler<PEER\_STREAM>

{

// ...

};

在开发运行在多种 OS 平台上的可移植应用时，模板所提供的增强的灵活性是有用的。例如，在跨越

SunOS 平台的多种变种时，能使用网络编程接口来对应用进行参数化的能力是必需的。特别地，SunOS

5.2 中的 socket 实现不是线程安全的，而 SunOS 4.x 中的 TLI 实现含有许多严重的缺陷。

TLI SAP 还将应用与 TLI 接口的许多特性屏蔽开来。例如，在 qlen > 1 的并发服务器中，[5]，ACE

\_TLI\_Acceptor 类的 accept 方法封装了处理 t\_listen 和 t\_accept 的非直观而又易错的行为所需的微妙的应用 级代码。accept 方法被动地建立客户连接请求。通过使用 C++缺省参数值，对于基于 TLI SAP 和基于 S OCK SAP 的应用来说，调用 accept 方法的标准方法在语法上都是一样的。

**3.6.4 SPIPE SAP** 和 **FIFO SAP**

SPIPE SAP 类属为已安装 STREAM 管道和 connld[17]提供一种 C++包装接口。SPIPE SAP 继承层次 是 SOCK SAP 和 TLI SAP 所用的层次的镜像。它提供与 SOCK SAP ACE\_LSOCK\* 类（它们封装的是

UNIX 域的 socket）相类似的功能。但是，SPIPE SAP 比 ACE\_LSOCK\* 接口更灵活，因为它使 STREA M 模块可以分别被“压入”或“弹出”SPIPE SAP 端点。SPIPE SAP 还支持在运行在同一主机上的进程 和/或线程间的字节流和按优先级排序的消息数据的双向递送[16]。

FIFO SAP 类属封装 UNIX FIFO 机制。

**3.7 SOCK SAP C++**包装类编程

这一部分通过使用 ACE SOCK SAP C++包装开发一个客户/服务器流式应用来对它们进行演示。该 应用是[1]中描述的 ttcp 程序的简化版本。为了比较，该应用还用 socket 进行了编写。为保持简短，例子 中的大多数错误检查都被省略了。自然，健壮的程序应该检查库和系统调用的返回值。

图 3-10 和图 3-11 介绍一个用 C 编写的客户/服务器程序，它使用 Internet 域的 socket 和 select 来实现 流应用。图 3-11 所示的服务器创建一个被动模式的侦听者 socket，并等待客户与它连接。一旦连接，服 务器接收来自客户的数据，并将其显示在它的标准输出流上。图 3-10 所示的客户端建立一个到服务器的 TCP 连接，并将它的标准输入流通过连接进行传输。客户使用非阻塞连接来限制它等待连接被接受或拒 绝的时间数量。

大多数的返回值错误检查被省略了，以节省空间。但是，即使是要使这个简单的例子正确工作，所 有 socket 初始化、网络寻址和流控制细节都必须被显式地编写；注意到这一点富有启发意义。而且，图 3

-10 和 3-11 中的代码对于不同时支持 socket 和 select 的平台来说是不可移植的。

图 3-12 和 3-13 使用 SOCK SAP 来重新实现 C 版本的客户/服务器程序。该 SOCK SAP 程序实现了 与图 3-10 和图 3-11 所介绍的相同的功能。与基于 socket 的 C 实现相比，SOCK SAP C++程序展示了下 列好处：

增强的清晰性：例如，网络寻址和主机定位由图 3-6 所示的 Addr 类来处理，它隐藏了在图 3-10 和 3-11 中必须显式编写的微妙而又易错的细节。而且，非阻塞连接建立的低级细节是由 SOCK Connector 工厂来 完成的。此外，模板 *traits* 的使用使在对参数化函数进行实例化时必须指定的类型参数的数目减到了最少。

增强的类型安全性：例如，ACE\_SOCK\_Acceptor 和 ACE\_SOCK\_Connector 连接工厂创建 ACE\_SOCK\_S

tream 对象，从而防止了在运行时发生图 3-3 所示的类型错误。

更小的程序大小：使主动和被动连接建立局限在 ACE\_SOCK\_Acceptor 和 ACE\_SOCK\_Connector 连接工 厂中大量地减少了代码的行数。此外，为构造器和方法参数提供的缺省值减少了常见的使用模式所需的 参数数目。

增强的可移植性：例如，由于使用了模板 traits，在 socket 和 TLI 之间切换只需要将客户中的

send\_data <ACE\_TLI\_Connector> (s\_addr);

改变为

send\_data <ACE\_SOCK\_Connector> (s\_addr);

以及将服务器中的

recv\_data<ACE\_SOCK\_Acceptor> (s\_addr);

改变为

recv\_data<ACE\_TLI\_Acceptor> (s\_addr);

如 3.8 所示，可用条件编译指令来进一步使通信软件与对特定类型的网络编程接口的依赖去耦合。

#define PORT\_NUM 10000

#define TIMEOUT 5

/\* Socket client. \*/

void send\_data (const char host[], u\_short port\_num)

{

struct sockaddr\_in peer\_addr;

struct hostent \*hp;

char buf[BUFSIZ];

int s\_sd, w\_bytes, r\_bytes, n;

/\* Create a local endpoint of communication \*/

s\_sd = socket (PF\_INET, SOCK\_STREAM, 0);

/\* Set s\_sd to non-blocking mode. \*/

n = fcntl (s\_sd, F\_GETFL, 0);

fcntl (s\_sd, F\_SETFL, n | O\_NONBLOCK);

/\* Determine IP address of the server \*/

hp = gethostbyname (host);

/\* Set up address information to contact server \*/ memset ((void \*) &peer\_addr, 0, sizeof peer\_addr); peer\_addr.sin\_family = AF\_INET;

peer\_addr.sin\_port = port\_num;

memcpy (&peer\_addr.sin\_addr, hp->h\_addr, hp->h\_length);

/\* Establish non-blocking connection server. \*/

if (connect (s\_sd, (struct sockaddr \*) &peer\_addr, sizeof peer\_addr) == -1)

{

if (errno == EINPROGRESS)

{

struct timeval tv = {TIMEOUT, 0};

fd\_set rd\_sds, wr\_sds; FD\_ZERO (&rd\_sds); FD\_ZERO (&wr\_sds); FD\_SET (s\_sd, &wr\_sds); FD\_SET (s\_sd, &rd\_sds);

/\* Wait up to TIMEOUT seconds to connect. \*/

if (select (s\_sd + 1, &rd\_sds, &wr\_sds, 0, &tv) <= 0)

perror ("connection timedout"), exit (1);

// Recheck if connection is established.

if (connect (s\_sd, (struct sockaddr \*) &peer\_addr, sizeof peer\_addr) == -1 && errno != EISCONN)

perror ("connect failed"), exit (1);

}

}

/\* Send data to server (correctly handles

"short writes" due to flow control) \*/

while ((r\_bytes = read (0, buf, sizeof buf)) > 0)

for (w\_bytes = 0; w\_bytes < r\_bytes; w\_bytes += n)

n = write (s\_sd, buf + w\_bytes, r\_bytes - w\_bytes);

/\* Close down the connection. \*/

close (s\_sd);

}

int main (int argc, char \*argv[])

{

char \*host = argc > 1 ? argv[1] : "ics.uci.edu";

u\_short port\_num = htons (argc > 2 ? atoi (argv[2]) : PORT\_NUM);

/\* Send data to the server. \*/

send\_data (host, port\_num);

return 0;

}

图 3-10 基于 socket 的客户例子

#define PORT\_NUM 10000

/\* Socket server. \*/

void recv\_data (u\_short port\_num)

{

struct sockaddr\_in s\_addr;

int s\_sd;

/\* Create a local endpoint of communication \*/

s\_sd = socket (PF\_INET, SOCK\_STREAM, 0);

/\* Set up the address information for a server \*/ memset ((void \*) &s\_addr, 0, sizeof s\_addr); s\_addr.sin\_family = AF\_INET;

s\_addr.sin\_port = port\_num;

s\_addr.sin\_addr.s\_addr = INADDR\_ANY;

/\* Associate address with endpoint \*/

bind (s\_sd, (struct sockaddr \*) &s\_addr, sizeof s\_addr);

/\* Make endpoint listen for service requests \*/

listen (s\_sd, 5);

/\* Performs the iterative server activities \*/

for (;;)

{

char buf[BUFSIZ];

int r\_bytes, n\_sd;

struct sockaddr\_in peer\_addr;

int peer\_addr\_len = sizeof peer\_addr;

struct hostent \*hp;

/\* Create a new endpoint of communication \*/

while ((n\_sd = accept (s\_sd, &peer\_addr,

&peer\_addr\_len)) == -1 && errno == EINTR)

continue;

hp = gethostbyaddr (&peer\_addr.sin\_addr, peer\_addr\_len, AF\_INET);

printf ("client %s\n", hp->h\_name);

/\* Read data from client (terminate on error) \*/

while ((r\_bytes = read (n\_sd, buf, sizeof buf)) > 0)

write (1, buf, r\_bytes);

/\* Close the new endpoint (listening endpoint remains open) \*/

close (n\_sd);

}

/\* NOTREACHED \*/

}

int main (int argc, char \*argv[])

{

u\_short port\_num = htons (argc > 1 ? atoi (argv[1]) : PORT\_NUM);

// Receive data from clients. recv\_data (port\_num);

return 0;

}

图 3-11 基于 socket 的服务器例子

static const int PORT\_NUM = 10000;

static const int TIMEOUT = 5;

// SOCK\_SAP Client. template <class CONNECTOR>

void send\_data (CONNECTOR::PEER\_ADDR peer\_addr)

{

// Data transfer object. CONNECTOR::PEER\_STREAM peer\_stream;

// Establish connection without blocking.

CONNECTOR connector (peer\_stream, peer\_addr, ACE\_NONBLOCK);

if (peer\_stream.get\_handle () == -1)

{

// If non-blocking connection is in progress,

// wait up to TIMEOUT seconds to complete. Time\_Value timeout (TIMEOUT);

if (errno != EWOULDBLOCK ||

connector.complete (peer\_stream, peer\_addr, &timeout) == -1)

perror ("connector"), exit (1);

}

// Send data to server (send\_n() handles

// "short writes" correctly). char buf[BUFSIZ];

for (int r\_bytes; (r\_bytes = read (0, buf, sizeof buf)) > 0;)

peer\_stream.send\_n (buf, r\_bytes);

// Explicitly close the connection. peer\_stream.close ();

}

int main (int argc, char \*argv[])

{

char \*host = argc > 1 ? argv[1] : "ics.uci.edu";

u\_short port\_num = htons (argc > 2 ? atoi (argv[2]) : PORT\_NUM);

// Address of the server. ACE\_INET\_Addr s\_addr (port\_num, host)

// Use SOCK SAP wrappers on client’s side. send\_data <ACE\_SOCK\_Connector> (s\_addr);

return 0;

}

图 3-12 基于 SOCK SAP 的客户例子

static const int PORT\_NUM = 10000;

// SOCK\_SAP Server. template <class ACCEPTOR>

void recv\_data (ACCEPTOR::PEER\_ADDR s\_addr)

{

// Factory for passive connection establishment. ACCEPTOR acceptor (s\_addr);

// Data transfer object. ACCEPTOR::PEER\_STREAM peer\_stream;

// Remote peer address. ACCEPTOR::PEER\_ADDR peer\_addr;

// Performs iterative server activities.

for (;;)

{

// Create a new STREAM endpoint

// (automatically restarted if errno == EINTR). acceptor.accept (peer\_stream, &peer\_addr);

printf ("client %s\n", peer\_addr.get\_host\_name ());

// Read data from client (terminate on error). char buf[BUFSIZ];

for (int r\_bytes = 0;;)

{

r\_bytes = peer\_stream.recv (buf, sizeof buf);

if (r\_bytes > 0)

write (1, buf, r\_bytes);

else

}

break;

// Close peer\_stream endpoint

// (acceptor endpoint stays open). peer\_stream.close ();

}

/\* NOTREACHED \*/

}

int main (int argc, char \*argv[])

{

u\_short port\_num = argc == 1 ? PORT\_NUM : atoi (argv[1]);

// Port for the server. ACE\_INET\_Addr s\_addr (port\_num);

// Use Socket wrappers on server’s side. recv\_data<ACE\_SOCK\_Acceptor> (s\_addr);

return 0;

}

图 3-13 基于 SOCK SAP 的服务器例子

**3.8 socket** 包装设计原则

这一部分描述下列贯穿 SOCK SAP 类属所应用的设计原则：

－在编译时强制实现类型安全性

－允许受控的类型安全性违例

－为常见情况进行简化

－用层次类属替代一维的接口

－通过参数化类型增强可移植性

－内联性能关键的方法

－定义辅助类隐藏易错细节

尽管这些原则已广为人知，并被广泛应用于像图形用户接口这样的领域中，但在通信软件领域中它 们还没有被那么广泛地应用。

**3.8.1** 在编译时强制实现类型安全性

在 3.4 讨论的 socket 的若干局限源于在其接口中缺乏类型安全性。为强制实现类型安全性，SOCK S AP 确保它所有的对象都通过构造器适当地初始化。此外，为防止偶然的类型安全性违例，只允许对 SO CK SAP 对象进行合法的操作。后一点已在图 3-14 所示的 echo\_server 的 SOCK SAP 修订版进行演示。 该版本更正了图 3-3 中所标识出的的 socket 和 C 的问题。因为 SOCK SAP 类是强类型的，非法操作在编 译时、而不是运行时被拒绝。例如，不可能在 ACE\_SOCK\_Acceptor 连接工厂上调用 recv 或 send 方法， 因为这些方法不是其接口的一部分。同样地，返回值只用于传达操作的成功或失败，从而减少了在赋值 表达式中误用的潜在可能性。

int echo\_server (ACE\_INET\_Addr s\_addr)

{

// Initialize the passive mode server. ACE\_SOCK\_Acceptor acceptor (s\_addr);

// Data transfer object. ACE\_SOCK\_Stream peer\_stream;

// Client remote address object. ACE\_INET\_Addr peer\_addr;

// Accept a new connection.

if (acceptor.accept (peer\_stream, &peer\_addr) != -1)

{

char buf[BUFSIZ];

for (size\_t n; peer\_stream.recv (buf, sizeof buf, n) > 0;)

// Handles "short-writes."

if (peer\_stream.send\_n (buf, n) != n)

// Remainder omitted.

}

}

图 3-14 Echo 服务器的 SOCK SAP 修订版

**3.8.2** 允许受控的类型安全性违例

该原则通过 IPC SAP 根类所提供的 get\_handle 和 set\_handle 方法进行例示。这两个方法分别提取和 指派底层的句柄。通过提供 get\_handle 和 set\_handle，IPC SAP 允许应用在必须与需要句柄的 UNIX 系统 调用（比如 select）协作时直接绕过 IPC SAP 的类型检查机制。陈述此原则的另一方式是“让 SOCK SA P 的正确使用更容易，不正确使用更困难，但不是不可能以类设计者没有预见到的方式来使用它。”

**3.8.3** 为常见情况进行简化

此原则以下列途径应用于 ACE C++ socket 包装类中： 为常用方法参数提供缺省值：例如，ACE\_SOCK\_Connector 构造器有六个参数：

ACE\_SOCK\_Connector(ACE\_SOCK\_Stream &new\_stream, const ACE\_SOCK\_Addr &remote\_sap, ACE\_Time\_Value \*timeout = 0,

const ACE\_SOCK\_Addr &local\_sap = (ACE\_SOCK\_Addr &) Addr::sap\_any, int protocol\_family = PF\_INET,

int protocol = 0);

但是，在调用与调用间通常只有前两个是变化的：

ACE\_SOCK\_Stream stream;

// Compiler supplies default values.

ACE\_SOCK\_Connector con (stream, ACE\_INET\_Addr (port, host));

// ...

因此，为简化编程，在 ACE\_SOCK\_Connector 中给出了其他参数的缺省值，以使程序员无需每次都 提供它们。

定义节俭的接口：此原则让使用一种特定抽象的代价局部化。IPC SAP 限定应用开发者所必须记忆的细

节数量。它为开发者提供群集的类，执行不同类型的通信（比如面向连接的 vs. 无连接的）和不同的连 接角色（比如主动的 vs. 被动的）。为减少犯错的机会，ACE\_SOCK\_Acceptor 类只允许为程序而应用的 操作扮演被动角色，而 ACE\_SOCK\_Connector 只允许为程序而应用的操作扮演主动角色。此外，与使用 高度通用的 UNIX sendmsg/recvmsg 函数相比，使用 ACE\_SOCK\_SAP 来发送和接收打开的文件句柄有着 一个简单得多的接口。例如，使用 ACE\_LSOCK\* 类来传递 socket 句柄是非常简洁的：

ACE\_LSOCK\_Stream stream; ACE\_LSOCK\_Acceptor acceptor ("/tmp/foo");

// Accept connection. acceptor.accept (stream);

// Pass the Socket handle back to caller. stream.send\_handle (stream.get\_handle ());

与此相比较，使用 socket 接口来实现所需的代码：

int n\_sd; int u\_sd; sockaddr\_un addr; u\_char a[2];

iovec iov;

msghdr send\_msg;

u\_sd = socket (PF\_UNIX, SOCK\_STREAM, 0);

memset ((void \*) &addr, 0, sizeof addr);

addr.sun\_family = AF\_UNIX;

strcpy (addr.sun\_path, "/tmp/foo");

bind (u\_sd, &addr, sizeof addr.sun\_family + strlen ("/tmp/foo"));

listen (u\_sd, 5);

// Accept connection.

n\_sd = accept (u\_sd, 0, 0);

// Sanity check.

a[0] = 0xab; a[1] = 0xcd;

iov.iov\_base = (char \*) a;

iov.iov\_len = sizeof a;

send\_msg.msg\_iov = &iov; send\_msg.msg\_iovlen = 1; send\_msg.msg\_name = (char \*) 0; send\_msg.msg\_namelen = 0; send\_msg.msg\_accrights = (char \*) &n\_sd; send\_msg.msg\_accrightslen = sizeof n\_sd;

// Pass the Socket handle back to caller. sendmsg (n\_sd, &send\_msg, 0);

将多个操作组合进单一操作：创建一个传统的被动模式socket需要多个调用：

int s\_sd = socket (PF\_INET, SOCK\_STREAM, 0);

sockaddr\_in addr;

memset (&addr, 0, sizeof addr); addr.sin\_family = AF\_INET; addr.sin\_port = htons (port); addr.sin\_addr.s\_addr = INADDR\_ANY; bind (s\_sd, &addr, addr\_len); listen (s\_sd);

// ...

相反，ACE\_SOCK\_Acceptor 是一个用于被动连接建立的工厂，它的构造器执行创建被动模式侦听者端点 所需的 socket 调用：socket、bind 和 listen。因此，应用只需简单地如下编写：

ACE\_INET\_Addr addr (port); ACE\_SOCK\_Acceptor acceptor (addr);

**3.8.4** 用层次类属替代一维接口

此原则涉及使用层次相关的类属来重构现有的一维 socket 接口（如图 3-9 所示）。用于构造 SOCK S AP 类属的准则涉及标识、群集和封装相关的 socket 函数，以最大化复用和类组件的共享。

继承支持 SOCK SAP 类属的不同功能子集。例如，不是所有的操作系统都支持传递打开的文件句柄

（例如，Windows NT）。因而，可以在继承层次中省略 ACE\_LSOCK 类（在 3.6.1 中描述），而不会影响

SOCK SAP 设计中的其他类的接口。 继承还增强代码复用和改善模块性。基类表示类属组件间的相似性，而派生类表示差异性。例如，I

PC SAP 设计向着 IPC SAP 和 SOCK SAP 基类中的继承层次的“根”放置了共享的机制，包括用于打开

/关闭和设置/取回底层 socket 句柄的操作，以及对所有派生的 SOCK SAP 类来说都通用的特定的选项管 理函数。而位于继承层次“底部”的子类实现专门的操作，为所提供的通信类型（比如流 vs. 数据报通 信，或本地 vs. 远地通信）而进行定制。这样的方法避免了不必要的代码重复，因为更为专门的派生类 会复用继承层次的根上所提供的更为通用的机制。

**3.8.5** 通过参数化类型增强可移植性

template <class ACCEPTOR>

int echo\_server (ACCEPTOR::PEER\_ADDR s\_addr)

{

// Initialize the passive mode server. ACCEPTOR acceptor (s\_addr);

// Data transfer object. ACCEPTOR::PEER\_STREAM peer\_stream;

// Remote address object. ACCEPTOR::PEER\_ADDR peer\_addr;

// Accept a new connection.

if (acceptor.accept (peer\_stream, &peer\_addr) != -1)

{

char buf[BUFSIZ];

for (size\_t n; peer\_stream.recv (buf, sizeof buf, n) > 0;)

if (peer\_stream.send\_n (buf, n) != n)

// Remainder omitted.

}

}

图 3-15 Echo 服务器的模板版本

通过允许经由参数化类型来整个地替换网络编程接口，用 C++类（而不是独立的 C 函数）包装 sock et 有助于改善可移植性。参数化类型使应用与对特定的网络编程接口的依赖去耦合。图 3-15 通过将 echo

\_server 修改成为 C++函数模板来演示这一技术。取决于底层 OS 平台的特定属性（比如它更为高效地实 现了 TLI 还是 socket），echo\_server 可以通过 SOCK SAP 或 TLI SAP 类来实例化。如下所示：

// Conditionally select IPC mechanism.

#if defined (USE\_SOCKETS)

typedef ACE\_SOCK\_Acceptor ACCEPTOR;

#else // USE\_TLI

typedef ACE\_TLI\_Acceptor ACCEPTOR;

#endif // USE\_SOCKETS. const int PORT\_NUM = 10000;

int main (void)

{

// ...

// Invoke the echo\_server with appropriate

// network programming interfaces. Note the

// use of template traits for addr class. ACCEPTOR::PEER\_ADDR addr (PORT\_NUM); echo\_server<ACCEPTOR> (addr);

}

一般而言，比起其他一些传统方法，比如实现多个版本或在源码中到处使用杂乱的条件编译指令，

使用参数化类型要更没有侵入性并且更可扩展一点。

例如，SOCK SAP 和 TLI SAP 类提供同样的 OO 接口（在图 3-7 中描述）。特定的 OS 平台可能拥有 不同的底层网络编程接口，比如有 socket，但没有 TLI，或反之亦然。使用 IPC SAP，应用可被透明地编 写，通过 SOCK SAP 或 TLI SAP 来进行参数化。C++模板支持“类型一致”（type conformance）的一种 松散形式，并不强迫接口包含所有可能的功能。相反，模板被用于参数化仔细设计的应用代码，用以调 用不同通信抽象的通用方法子集（例如，open、close、send、recv，等等）。

模板提供的类型抽象改善了支持不同网络编程接口（比如 socket 或 TLI）的平台间的可移植性。例 如，对于开发跨越多种 SunOS 平台的应用来说，对网络编程接口的参数化是有用的。SunOS 5.2 中的 so cket 实现不是线程安全的，而 SunOS 4.x 中的 TLI 实现含有许多严重的缺陷。

**3.8.6** 内联性能关键的方法

为鼓励开发者用 C++包装替换现有的低级网络编程接口，SOCK SAP 实现必须高效地运作。为确保 这一点，在关键的性能路径上的方法（比如 ACE\_SOCK\_Stream recv 和 send 方法）被指定为 C++内联函 数，以消除运行时函数调用开销。内联在时间和空间上都是高效的，因为这些方法非常短小（每个方法 大约 2 或 3 行）。内联的使用意味着应该保守地使用虚函数，因为大多数当代的 C++编译器不能充分地把 虚函数开销优化掉。

**3.8.7** 定义辅助类隐藏易错的编程细节

socket 寻址的 C 接口是难以使用和易错的。很容易忽略 sockaddr\_in 的清零或把端口号转换到网络字 节序。为使应用与这些低级细节相屏蔽，IPC SAP 定义了 Addr 类层次（如图 3-6 所示）。该层次通过类 型安全的 C++接口支持若干不同的网络寻址格式。Addr 层次消除了与直接使用基于 C 的 struct sockaddr 数据结构族相关联的常见编程错误。例如，ACE\_INET\_Addr 的构造器自动将 sockaddr 寻址结构清零，并 将端口号转换为网络字节序。如下所示：

class ACE\_INET\_Addr : public ACE\_Addr

{

public:

ACE\_INET\_Addr::ACE\_INET\_Addr (u\_short port, long ip\_addr = 0)

{

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);

}

private:

sockaddr\_in inet\_addr\_;

};

**3.9** 结束语

IPC SAP 提供一个 OO C++包装族，封装在当代的操作系统上可用的标准本地和远地 IPC 机制。通 过使编写正确、紧凑、可移植和高效的代码变得更为容易，这些封装的接口简化了通信软件的开发。此 外，包装方法还便利了向 C++的有组织的迁移，通过（1）逐步对开发者进行 OO 设计原理教学，以及（2） 有效利用现有的非 C++语言的代码库。本论文通过描述使用 C++实现 IPC SAP 的若干优点和缺点、以及 概述将来进一步对 IPC SAP 的使用进行探究的论文来作为结束。

使用 **C++**的优点和缺点：用 C++开发包装的主要优点包括：

－封装变种：类隐藏寻址格式中的差异，比如 Internet vs. UNIX 域寻址。此外，它们还在不同的类中封 装不同的接口行为。例如，ACE\_SOCK\_Acceptor 对象的接口为服务器操作而特别作了剪裁。

－增强功能子集划分：继承使定义功能子集变得更为容易。例如，ACE\_LSOCK 类可在不支持文件句柄 传递的操作系统上被忽略。

－更高的可移植性：模板使得不同的 IPC 机制可被参数化进应用，从而改善了跨平台可移植性。

C++的一个缺点是它缺少可移植的异常处理。在适当使用时，C++异常处理有助于简化错误恢复，并 改善类型安全性。例如，如果 ACE\_INET\_Addr 构造器因为远地地址没有对应到有效的主机而失败的话， 就可以扔出一个异常。但是，如果没有 C++异常处理，就有可能在没有对 IPC SAP 对象进行正常初始化 的情况下开始使用它。该问题将在 ANSI/ISO C++异常处理机制可用于大多数 OS 平台时得以解决。

当前状况和未来主题：IPC SAP 可在 ACE 框架[2]中找到。ACE 支持的 OS 平台包括 Win32（使用 MSV C++和 Borland C++的 Win2000、WinNT 3.5.x、4.x、Win95 和 WinCE）、大多数版本的 UNIX（SunOS 4. x 和 5.x；SGI IRIX 5.x 和 6.x；HP-UX 9.x、10.x 和 11.x；DEC UNIX 3.x 和 4.x、AIX 3.x 和 4.x、DG/ UX、Linux、SCO、UnixWare、NetBSD 和 FreeBSD）、实时操作系统（VxWorks、Chorus、LynxOS 和 pS oS），以及 MVS OpenEdition。

ACE 已被用于许多大学和公司的研究和开发项目。例如，ACE 已在波音被用于构建实时航空控制系 统[23]；在 Bellcore[22]、爱立信[24]、摩托罗拉[25]和朗讯被用于电信系统；在西门子[26]和柯达[27]被用 于医学成像系统；以及在 SAIC/DARPA 的分布式模拟系统。它还被广泛地用于研究项目和课堂教学。

本论文中描述的所有源代码都可以在 [http://www.cs.wustl.edu/~schmidt/ACE.html](http://www.cs.wustl.edu/%7Eschmidt/ACE.html) 找到。[http://www.cs. wustl.edu/~schmidt/ACE-users.html](http://www.cs.wustl.edu/%7Eschmidt/ACE-users.html) 描述使用 ACE 的许多项目。此外，comp.soft-sys.ace 是专用于 ACE 相

关主题的 USENET 新闻组。

参考文献

[1] D. C. Schmidt, T. H. Harrison, and E. Al-Shaer, “Object-Oriented Components for High-speed Network Programmin g,” in *Proceedings of the 1st Conference on Object-Oriented Technologies and system*s, (Monterey,CA), USENIX, June 19

95.

[2] D. C. Schmidt, “ACE: an Object-Oriented Framework for Developing Distributed Applications,” in *Proceedings of the*

*6th USENIX C++ Technical Conferenc*e, (Cambridge, Massachusetts), USENIX Association, April 1994.

[3] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4BSD Ope rating Syste*m. Addison Wesley, 1996.

[4] Sun Microsystems, *Network Interfaces Programmer’s Guid*e, Chapter 6 (TLI Interface) ed., 1992. [5] S. Rago, *UNIX System V Network Programmin*g. Reading, MA: Addison-Wesley, 1993.

[6] W.R.Stevens,*UNIX Network Programming, Second Editio*n. Englewood Cliffs, NJ: Prentice Hall, 1997. [7] H. Custer, *Inside Windows N*T. Redmond, Washington: Microsoft Press, 1993.

[8] Object Management Group, *The Common Object Request Broker: Architecture and Specificatio*n, 2.2 ed., Feb. 1998. [9] D. Box, *Essential CO*M. Addison-Wesley, Reading, MA, 1997.

[10] S. Mungee, N. Surendran, and D. C. Schmidt, “The Design and Performance of a CORBA Audio/Video Streaming

Service,” in *submitted to the Hawaiian International Conference on System Science*s, Jan. 1999.

[11] A. Gokhale and D. C. Schmidt, “Measuring the Performance of Communication Middleware on High-Speed Network s,” in *Proceedings of SIGCOMM ’9*6, (Stanford, CA), pp. 306–317, ACM, August 1996.

[12] OSI Special Interest Group, *Transport Provider Interface Specificatio*n, December 1992. [13] OSI Special Interest Group, *Data Link Provider Interface Specificatio*n, December 1992.

[14] D. Ritchie, “A Stream Input–Output System,” *AT&T Bell Labs Technical Journa*l, vol. 63, pp. 311–324, Oct. 1984. [15] D. C. Schmidt, “IPC SAP: An Object-Oriented Interface to Interprocess Communication Services,” *C++ Repor*t,vol.4, November/December 1992.

[16] W. R. Stevens, *Advanced Programming in the UNIX Environmen*t. Reading, Massachusetts: Addison Wesley, 1992. [17] D. L. Presotto and D. M. Ritchie, “Interprocess Communication in the Ninth Edition UNIX System,” *UNIX Researc h System Papers, Tenth Editio*n, vol. 2, no. 8, pp. 523–530, 1990.

[18] A. Stepanov and M. Lee, “The Standard Template Library,” Tech. Rep. HPL-94-34, Hewlett-Packard Laboratories, A

pril 1994.

[19] G. Booch, *Object Oriented Analysis and Design with Applications (2nd Edition*). Redwood City, California: Benjamin

/Cummings, 1993.

[20] D. C. Schmidt, “Acceptor and Connector: Design Patterns for Actively and Passively Initializing Network Services,”

in *Workshop on Pattern Languages of Object-Oriented Programs at ECOOP ’9*5, (Aarhus, Denmark), August 1995.

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Softwar*e. Reading, MA: Addison-Wesley, 1995.

[22] D. C. Schmidt, “Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dis patching,” in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, M A: Addison-Wesley, 1995.

[23] T. H. Harrison, D. L. Levine, and D. C. Schmidt, “The Design and Performance of a Real-time CORBA Event Ser

vice,” in *Proceedings of OOPSLA ’9*7, (Atlanta, GA), ACM, October 1997.

[24] D. C. Schmidt and P. Stephenson, “Experiences Using Design Patterns to Evolve System Software Across Diverse OS Platforms,” in *Proceedings of the 9th European Conference on Object-Oriented Programmin*g, (Aarhus, Denmark), AC M, August 1995.

[25] D. C. Schmidt, “A Family of Design Patterns for Application-level Gateways,” *The Theory and Practice of Object*

*Systems (Special Issue on Patterns and Pattern Languages*),vol.2, no. 1, 1996.

[26] P. Jain and D. C. Schmidt, “Service Configurator: A Pattern for Dynamic Configuration of Services,” in *Proceeding s of the 3rd Conference on Object-Oriented Technologies and System*s, USENIX, June 1997.

[27] I. Pyarali, T. H. Harrison, and D. C. Schmidt, “Design and Performance of an Object-Oriented Framework for High

-Performance Electronic Medical Imaging,” *USENIX Computing System*s, vol. 9, November/December 1996.

第 **4** 章 **ACE** 轻量级 **OS** 并发机制的 **OO** 封装

Douglas C. Schmidt

摘 要

本论文描述 *ACE* 面向对象的线程封装 *C++*类库的设计，该类库使程序员与 *Solaris* 线程、*POSIX pt hreads* 及 *Win32* 线程之间的差异相屏蔽；并从最终用户和内部设计的视角来展现其体系结构，并讨论了 关键的设计和实现问题。读者将获得对总体设计方法，以及在多种软件质量因素，如质量、可移植性及 可扩展性之间所做的权衡的理解。

**4.1** 介绍

某些类型的分布式应用通过使用并发模式执行任务来从中获益。对于多处理器平台上的网络服务器， 并发特别有助于改善性能和简化编程。对于服务器应用，使用线程来并发地处理多客户请求常常比下面 的设计方法要更为方便和更不易出错：

z 在传输层接口对请求进行人工的序列化；

z 在内部对请求进行排队，并依次处理它们；

z 为每一客户请求派生一个重量级的进程。

本论文描述 ACE 自适配通信环境[1]中包含的 C++类库。ACE 封装并增强了由 Solaris 2.x 线程[2]、P OSIX Pthreads[3]及 Win32 线程[4]所提供的轻量级并发机制。

在本论文中介绍的材料面对的是那些有兴趣了解线程的面向对象（OO）并发编程的战略和战术的技 术人员。读者被假定熟悉一般的 OO 设计和编程技术（比如设计模式[5]、应用框架[6]、模块性、信息隐 藏和对象建模[7]）、OO 表示法（比如 OMT[8]），基本的的 C++编程语言特性（比如类、继承、动态绑定 和参数化类型[9]）、基本的 UNIX 系统编程概念（比如进程管理、虚拟内存和进程间通信[10]），以及网络 术语（比如客户/服务器体系结构[11]、RPC[12]、CORBA[13]和 TCP/IP[14, 15]）。

一般而言，理解本论文并不需要对并发有深入的了解；特别地，也不需要对 Solaris/POSIX/Win32 多 线程和同步机制有深入的了解。对并发编程和多线程的综述在 4.3 介绍，在其中定义了关键的术语，并概 述了多种用于在 Solaris 2.x、POSIX pthreads 和 Win32 线程上进行并发编程的可选机制。

本论文被组织如下：4.2 给出对 ACE OS 线程封装库的目标的综述，并概述该库的组件的面向对象体 系结构。4.3 一般性地介绍并发编程的相关背景材料，并特别介绍了 Solaris 多线程模型。4.4 介绍一种激 发了 ACE 线程封装库的设计的最终用户视点，并聚焦于从并发客户/服务器应用中精选的一个使用实例。

4.5 详细描述 ACE 线程封装库的公共接口和内部设计。4.6 介绍了若干例子，演示在 4.5 中定义的 OO 组

件。最后，4.7 给出结束语。

**4.2 ACE OO** 并发机制综述

**4.2.1** 总体目标

与前几代 SunOS 相比，现代操作系统（比如 Solaris、OSF/1、Windows NT 和 OS/2）的一种显著特 性是其集成的对内核级和用户级多线程及同步的支持8。但是，现有的与这些操作系统一起发布的多线程 和同步机制都是用 C 写成的相对低级的 API。混合使用 C++类和低级 C API 来开发应用给开发者造成了 不可接受的负担。在单个应用中混合这两种风格将导致面向对象和过程编程之间的“阻抗”失配。这样 一种混合的编程风格让人迷惑，并会带来慢性的维护问题。

为避免让每个开发者实现他们自己特别的 OS 线程机制的 C++包装，ACE 提供了一组在此论文中描 述的面向对象的并发组件。这些 ACE 组件为并发编程提供了可移植和可扩展的接口。该接口简化了用于 开发客户和服务器的线程管理和同步机制。它已被移植到 POSIX pthreads 标准的许多试验版本[3]、Solar is 线程[2]、Microsoft Win32 线程[4]，以及 VxWorks tasks。

**4.2.1.1** 总体要求

与封装和简化 OS 线程机制的并发底层的目的相结合，ACE OO 线程封装类库正在被开发以响应下 列常见的应用要求：

z 简化程序设计：通过允许多个应用任务使用传统的同步编程抽象独立地执行（比如 CORBA 远地方法 请求）来实现；

z 透明地改善性能：通过使用像 SPARCcenter 1000 和 2000 共享内存对称多处理器这样的硬件平台的并 行处理能力来实现；

z 显式地改善性能：通过减少数据拷贝，以及通过重叠的通信计算来实现；

z 改善可感知的响应时间：对于交互式应用（比如用户接口或是网络管理应用），通过将分离的线程与 应用中的不同任务或服务相关联来实现。

**4.2.1.2** 设计目标

8 为了更具体，这一部分聚焦于 Solaris 2.x 的线程和同步机制。但是，这些机制、设计原理和接口中的大多数对于 POSIX Pthreads 和 Win32 threads 也有着同样的意义。

ACE OO 线程类库被开发用以实现下列设计目标：

z 通过使开发者能够在他们的并发应用中始终一致地使用 C++和 OO，促进编程风格的一致性。

z 改善底层并发机制的可移植性和可复用性。

z 减少为使应用线程安全化所需做出的强制性变动。

z 排除或减少发生那些微妙的同步错误的潜在可能性。

z 增强抽象和模块性，而又不牺牲性能。

**4.2.2 ACE OO** 线程封装组件的体系化综述

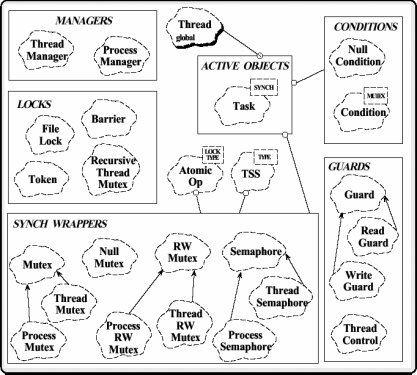


图 4-1 ACE OO 线程封装组件的对象模型

图 4-1 中的 Booch 对象模型演示了 ACE 线程封装类库中的组件。这些组件包括下面描述的 C++类和 类属9。

9 所有的 ACE 类都冠有 ACE\_前缀，以避免弄乱程序的全局名字空间。为了简洁，在下面所有的代码中都省略了该前缀。

**4.2.2.1 ACE** 锁（**Lock**）类属

z **Mutex**、**Thread\_Mutex** 和 **Process\_Mutex**：这些类提供简单而高效的机制来序列化对共享资源（比 如共享内存中的文件或对象）的访问。它们封装了 Solaris、POSIX 和 Win32 同步变量（分别是 mute x\_t、pthread\_mutex\_t 和 HANDLE）；在 4.5.1.1 中描述。

z **RW\_Mutex**、**RM\_Thread\_Mutex**、**RW\_Process\_Mutex**：这些类序列化对共享资源的访问，其中的 内容较少变动，而是更多地用于搜索。它们封装了 Solaris rwlock\_t 同步变量（POSIX pthreads 和 Wi n32 线程实现使用其他机制）；在 4.5.1.3 中描述。

z **Semaphore**、**Thread\_Semaphore**、**Process\_Semaphore**：这些类实现了 Dijkstra 的“计数信号量”抽 象（一种用于序列化多线程控制的通用机制）。它们封装了 Solaris sema\_t 同步变量（POSIX pthreads 和 Win32 线程实现使用其他机制）；在 4.5.1.2 中描述。

z **Null\_Mutex**：Null\_Mutex 类提供一种零开销的锁接口实现，被其他 C++封装用于同步。此类在 4.5.1.

5 中描述。

z **Token**（令牌）：Token 类提供一种比 Mutex 更为通用的同步机制。例如，它实现了“递归互斥体”语 义，拥有令牌的线程可以重新获取它，而不会导致死锁。此外，当其他线程释放令牌时，阻塞在该令 牌上的线程以严格的 FIFO（先进先出）的顺序被服务（相反，Mutex 并不严格地强制实行一种获取 顺序）。该类在 4.5.1.6 中描述。

z **Recursive\_Thread\_Mutex**：通过允许嵌套调用 acquire 方法（只要拥有该锁的线程也是重新获取它的 线程）， Recursive\_Thread\_Mutex 扩展了缺省的 Solaris 线程互斥体语义。它与上面概述的 Thread\_Mu tex 类一起工作；在 4.5.1.4 中描述。

**4.2.2.2 ACE** 守卫（**Guard**）类属

z **Guard**、**Write\_Guard** 和 **Read\_Guard**：这些类确保在进入和退出一个 C++代码块时分别被自动获取 和释放锁。它们在 4.5.2.1 中描述。

z **Thread\_Control**：Thread\_Control 类与 Thread\_Manager 类相结合，用于在线程的发起函数中自动进行 优雅的终止和清扫活动。该类在 4.5.2.2 中描述。

**4.2.2.3 ACE** 条件（**Condition**）类属

z **Condition**：Condition 类用于在涉及共享数据的条件表达式的状态发生变化时进行阻塞。它封装了 So laris 和 POSIX pthreads cond\_t 同步变量（Win32 线程使用其他机制实现）；在 4.5.3.1 中描述。

z **Null\_Condition**：Null\_Condition 类提供零开销的 Condition 接口实现，用于单线程应用。它在 4.5.3.2

中描述。

**4.2.2.4 ACE** 管理器（**Manager**）类属

z **Thread\_Manager**：Thread\_Manager 类含有一套机制来对相互协作以实现集体行为的成组线程进行管 理。该类在 4.5.4.1 中描述。

z **Thread\_Spawn**：Thread\_Spawn 提供一种标准工具，对为并发地处理来自客户的请求所进行的线程创 建进行管理。该类在 4.5.4.2 中描述。

**4.2.2.5 ACE** 主动对象（**Active Object**）类属

z **Task**（任务）：Task 类是 ACE 中用于定义主动对象[16, 17]的中心机制。这些主动对象将输入输出消 息排队，并在分离的线程控制中执行用户定义的消息处理服务。该类在 4.5.5.1 中描述。

**4.2.2.6** 杂项 **ACE** 并发类

z **Thread**：Thread 类封装 Solaris 线程、POSIX Pthreads 和 Win32 线程族的线程创建、终止和管理例程。 该类在 4.5.6.1 中描述

z **Atomic\_Op**：Atomic\_Op 类将同步特性透明地参数化进基本的算术操作。该类在 4.5.6.2 中描述。

z **Barrier**（栅栏）：Barrier 类实现“栅栏同步”，对于许多类型的并行科学应用特别有用。该类在 4.5.6.

3 中描述。

z **TSS**：TSS 类允许“物理上”线程专有的对象被“逻辑地”当作程序的全局对象进行访问。该类在 4.

5.6.4 中描述。

**4.3** 并发编程和多线程的背景

大多数 UNIX 系统程序员都熟悉传统的进程管理系统调用（比如 fork、exec、wait 和 exit）。但是， 他们关于正在形成中的 UNIX 多线程和同步机制（比如 Solaris 线程[2]、POSIX pthreads[3]，或是 Win32 线程[4]）的经验却较少。这一部分将给出对并发编程和 Solaris 线程的相关背景材料的综述。对并发编程 和 Solaris/POSIX/Win32 线程的更为详细的讨论见[2, 18, 19, 3, 4]。

**4.3.1** 进程和线程

进程是使程序指令得以执行的一组资源。这些资源包括虚拟内存、I/O 描述符、运行时栈、信号处理 器、用户和组 id，以及访问控制令牌。在早期的 UNIX 系统上（比如 SunOS 4.x），进程是“单线程”的。 在 UNIX 中，单线程程序中的操作通常是同步的，因为控制总是在程序（也就是，用户代码）中，或是 在操作系统中（经由系统调用）。在某种程度上，传统 UNIX 进程的单线程特性简化了编程，因为没有程 序员显式地进行干预，进程不会与其他进程相互干扰。

但是，使用单线程进程，有许多应用很难开发（特别是网络服务器）。例如，单线程网络文件服务器 不能长期阻塞以处理一个客户请求，因为其他客户的响应性会受到损害。有若干常用方法可用以避免阻 塞在单线程服务器中：

z 事件多路分离器*/*分派器：方法之一是开发一种事件多路分离器/分派器（比如面向对象的反应器框架[2

0]）。该技术被广泛用于在单线程用户接口框架中管理多个输入设备。主事件多路分离器/分派器检测 到来的事件，将其分离到适当的事件处理器，然后分派与该事件处理器相关联的应用特有的回调方法。

该方法的主要缺点是持续时间长的会话必须被开发为有限状态机。当状态的数目增加时该方法将 变得相当笨拙。此外，因为只能使用非阻塞的操作，很难通过像“I/O 流” 这样的技术、或是数据和 指令缓存中的本地引用方案来改善性能。

z 用户级协同例程（User-level co-routines）：另一种方法是开发一个非占先的（non-preemptive）用户级 协同例程包，显式地存储和恢复上下文信息。这使得任务能够挂起它们的执行，直到另一个协同例程 在后面将它们唤醒。Windows 3.1 和 Mac System 7 OS 上的多任务机制是广泛可用的使用这一方法 的系统。

一般而言，要正确地使用协同例程很复杂，因为开发者必须通过周期性地显式派生线程控制来人 工地进行任务占先。而且，每个任务必须只执行相对较短的时间。否则，客户将会觉察请求正在被顺 序地、而非并发地处理。协同例程的另一局限是，如果 OS 在任务引发页错误时阻塞进程中所有的任 务，应用的性能可能会下降。此外，单个任务（例如，进入了一个无限循环）的失败可能会挂起整个 进程。

z 多进程：降低单线程 UNIX 进程的复杂性的另一种方法是使用 fork 和 exec 系统调用提供的粗粒度多 进程能力。fork 派生一个分离的子进程，与父进程并发地执行任务。相互分离的进行可以通过使用像 共享内存和内存映射文件这样的机制来进行直接的协作。在本地主机上，共享内存是一种比消息传递 更为迅捷的 IPC 方法，因为它避免了显式的数据拷贝。

但是，fork 和 exec 的开销和不灵活使得动态的进程请求对于许多应用来说都极为昂贵和复杂。 例如，对于持续时间短的服务（比如解析 IP 地址的以太网号 、从网络文件服务器获取磁盘块，或是 在 SNMP MIB 中设置属性）来说，进程管理开销就太过度了。而且，使用 fork 和 exec 很难对调度和 进程优先级进行细粒度的控制。此外，在共享内存段中共享 C++对象的进程必须对虚表指针的位置做 出不可移植的假定。

多线程机制提供了更为优雅，有时也更为高效的方法来克服上述的传统并发进程技术的局限。线程 是在进程的上下文中执行的单序列的指令步骤。除了指令指针，线程还包括其他的一些资源，比如函数 启用记录的运行时栈、一组通用寄存器，以及线程专有的数据。

传统的工作站操作系统（比如 UNIX 的一些变种[2, 21, 22]和 Windows NT[4]）支持多进程（每一个 进程包含 1 或多个线程）的并发执行，每个进程可包含 1 或多个线程。进程充当被保护的单元、和在单 独的硬件保护地址空间中进行资源分配的单元。线程充当在进程地址空间中运行的执行单元，该线程与 0 或多个线程共享此地址空间。

**4.3.2** 基于线程的并发编程的好处

因为如下原因，在相互分离的线程、而不是进程中实现执行多任务的并发应用常常是有益的：

z 线程创建：不像生成新进程，派生一个新线程不需要（1）复制父地址空间内存，（2）设置新的内核 数据结构，以及（3）消耗一个额外的进程槽，以在大的应用中执行子任务。

z 上下文切换：线程维护最小限度的状态信息，因而降低了上下文切换的开销，因为只须存储和取回少 量状态信息。特别地，线程间的上下文切换消耗的时间比 UNIX 重量级进程间的上下文切换要少。这 是由于在同一进程中的线程间切换时，无需改变 TLB 虚地址映射。而且，严格地在用户级运行的线 程不会带来任何上下文切换开销。

z 同步：当调度和执行应用的线程时，可能不需要在内核模式和用户模式之间进行切换。进程同步比起 线程同步要更为昂贵一些。例如，进行同步的实体常常不是全局的、而是局部的实体。全局同步总是 涉及到内核，而应用线程所使用的局部（或“进程内”）同步则可能无需内核的干预。

z 数据拷贝：分离的线程间通过共享内存进行的通信常常比在分离的进程间使用 IPC 消息传递要快得 多，因为前者避免了显式的数据拷贝的开销。例如，相互协作的数据库服务经常引用驻留内存的公用 数据结构，通过线程来实现这些服务可能要更为简单和高效。一般而言，在线程间使用进程的共享地 址空间进行通信通常比在进程间使用共享内存机制（比如系统 V 共享内存或内存映射文件）要更为 容易和高效。

**4.3.3 Solaris** 上的多进程和多线程综述

这一部分总结 Solaris 2.x 提供的多进程（MP）和多线程（MT）机制的相关背景材料。其他的线程 模型和实现（比如 SGI、Sequent、OSF/1 和 Windows NT）的细节有所不同，但基本的概念都是非常类似 的。

相对来说，传统的 UNIX 进程是一种“重量级”的实体，其中含有一个单线程控制。相反，Solaris 上可用的基于线程的并发机制要更为成熟、灵活和高效（在适当使用时）。如图 4-2 所示，Solaris MP/M T 体系结构在两个层面上运作（内核空间和用户空间），并含有以下 4 种组件：

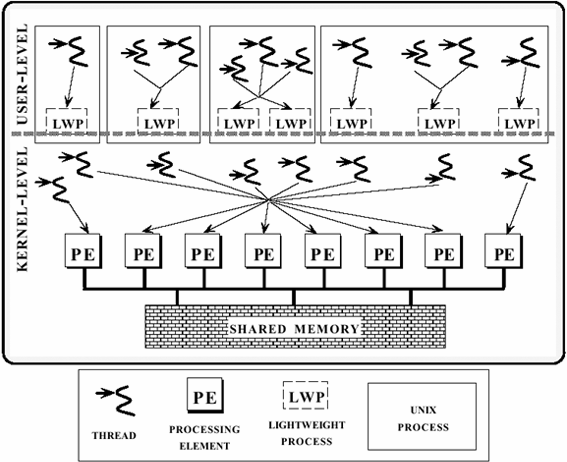


图 4-2 Solairs 2.x 多进程和多线程体系结构

z 处理单元（*Processing element*）：这是执行用户级和内核级指令的 CPU。Sun MP/MT 模型的语义意 图能同时用于单处理器和共享内存的硬件上的对称多处理器。

z 内核线程（*Kernel thread*）:这是处理单元(PE)在内核空间中调度和执行的基本实体。OS 内核为每一 内核线程维护一个小数据结构和栈。内核线程间的上下文切换相对较快，因为它不要求改变虚拟内存 信息。

z 轻量级进程（*Lightweight process*，或 *LWP*）：它们与内核线程相关联。在 Solaris 2.x 中，一个 UNIX 进程不再是一个线程控制，而是在其中含有一或多个 LWP。在 LWP 和它的内核线程间存在着 1 对 1 的映射10。Solaris 中的内核级调度器使用 LWP（因而也包括内核进程）来调度应用任务。LWP 含有 数量相对较多的状态（比如寄存器数据、账务和特征信息、虚拟内存地址范围和定时器）。因而，L WP 间的上下文切换相对较慢。

对于分时调度器类别（缺省），调度器通过“占先”（preemption）将可用的 PE 在多个活动的 L WP 间进行划分。通过这种技术，每个 LWP 运行一段有限的时间（通常为 10 毫秒）。当前 LWP 的时 间片到期后，OS 调度器选择另一个可用的 LWP，执行一次上下文切换，并将被占先的 LWP 放置到 一个队列中。内核使用若干标准（比如优先级、资源可用性、调度类别，等等）来调度 LWP。在分 时调度器类别中，没有固定的 LWP 执行顺序。

z 应用线程：每个 LWP 可被认为是一个“虚拟 PE”，在其上应用线程被一个用户级的线程库调度和多 路复用。每个应用线程与其他线程一起共享它的进程地址空间，尽管它拥有唯一的栈和寄存器组。应

10 另一方面，不是每个内核进程都有 LWP。例如，存在着拥有一个内核线程，并整个地在内核空间中操作的系统线程（像 页看守、NFS 看守，以及呼出线程）。

用线程还可以派生出其他应用线程。在进程中，每个这样的应用线程都独立地执行（尽管取决于硬件、 并非必然地并发执行）

Solaris 2.x 提供一种多层的并发模型，允许使用下面的两种模式来派生和调度应用线程：

1. 绑定线程（Bound thread）：它被 1 对 1 地映射到 LWP 和内核线程。绑定线程允许独立任务在多 PE 上并行执行。因而，如果两个应用线程运行在分离的 LWP 上（因而也是分离的内核线程上）， 它们也就可以并行地执行（假定它们运行在多处理器上，或使用异步 I/O）。而且，应用线程可以 执行阻塞的系统调用并处理页错误，而不会阻止其他应用线程的运行。

重新调度绑定线程需要一次内核级上下文切换。同样地，绑定线程上的同步操作也需要 OS 内核的干预。当应用被设计利用在硬件平台上可用的并行性优点时，绑定线程最为有用。因为每 个绑定线程都要求分配内核资源，分配大量绑定线程可能导致效率低下。

2. 非绑定线程：它们被一个线程运行时库以多对多的方式在一或多个 LWP 和内核线程上多路复用。 该用户级库实现了一个非占先的协作式多任务并发模型。在使内核干预最少化的同时，它调度、 分派，并挂起非绑定线程。与使用绑定到 LWP 的应用线程相比较，非绑定应用线程的派生、上 下文切换和同步所需的开销较小。

取决于应用和/或库与一个进程相关联的内核线程的数目，可以在多 PE 上并行执行一或多个 非绑定线程。因为每个非绑定线程并不分配内核资源，有可能分配数量相当大的非绑定线程，而 不会显著地降低性能。

**4.3.4** 并发编程的挑战

在多处理器上，可以在分离的多个 PE 上并行地运行多于一个的 LWP。在单处理器上，在任何时刻 只能有一个活动的 LWP。不管是怎样的硬件平台，程序员必须确保对共享资源（比如文件、数据库记录、 网络设备、终端，或共享内存）的访问是依次进行的，以防止“竞争状态”（race condition）。竞争状态 在两个或多个并发 LWP 的执行顺序会导致不可预测和错误的结果时发生（比如数据库记录被留在了不一 致状态）。竞争状态可使用 4.3.5 描述的 Solaris 2.x 同步机制来加以排除。这些机制序列化对共享资源的 代码临界区的访问。

除了并发控制的挑战，当使用多线程（而不是多进程，或单线程的反应式事件循环）实现并发应用 时还会出现以下的限制：

z 健壮性：在单进程地址空间中通过线程执行所有任务可能会降低应用的健壮性。该问题之所以发生， 是因为在同一进程地址空间中的分离线程彼此间并没有受到保护。为了降低上下文切换和同步开销， 硬件内存管理单元（MMU）11对线程的保护很少，或者根本没有。

因为线程不受保护，进程中一个有缺陷的服务可能破坏它与在进程的其他线程中运行的服务共享 的全局数据结构。于是这就可能导致不正确的结果，毁坏整个进程，致使网络服务器无限期地挂起， 等等。一个相关的问题是在某个线程中调用的某些 UNIX 系统调用可能对整个进程产生不希望产生的 副作用。例如，exit 系统调用具有销毁进程中所有线程的副作用（应使用 thr\_exit 来终止当前线程）。

z 访问特权：多线程的另一局限是一个进程中的所有线程共享同样的用户 id 和访问文件和其他受保护 资源的特权。因此，为防止对未授权资源意外或故意的访问，那些将安全机制建立在进程所有权之上 的网络服务（比如 Internet ftp 和 telnet 服务）通常都在分离的进程中实现。

11 MMU 保护分离的进程地址空间免于遭受系统中其他活动进程所带来的意外或恶意的破坏。

z 性能：一种常见的误解是认为使应用多线程化将自动提高性能。其实在许多环境中，多线程并不会提 高性能。例如，单处理器[19]上专事计算的应用并不会从多线程中获益，因为计算不会涉及通信。另 外，细粒度的锁定会导致高昂的同步开销[23, 24]。这也阻止了应用对并行处理的优点进行全面利用。

在有些环境中多线程可以显著地提高性能。例如，通过在多处理器平台上运行，一个多线程的面 向连接的应用网关可以从中获益。同样地，在单处理器上，专事 I/O 的应用也可以从多线程中获益， 因为计算涉及到通信和磁盘操作。

**4.3.5 Solaris 2.x** 同步和线程机制综述

这一部分概述并演示在 Solaris 2.x、POSIX pthreads 和 Win32 线程中可用的同步和线程机制。在这 些系统中，线程在单进程地址空间中共享若干资源（比如打开的文件、信号处理器，以及全局内存）。因 此，它们必须利用同步机制来协调对共享数据的访问，以避免发生 4.3.4 所讨论的竞争状态。为演示对同 步机制的需要，考虑下面的 C++代码段：

typedef u\_long COUNTER;

COUNTER request\_count; // At file scope

void \*run\_svc (Queue<Message> \*q)

{

Message \*mb; // Message buffer

while (q->dequeue (mb)) > 0)

{

// Keep track of number of requests

++request\_count;

// Identify request and

// perform service processing here...

}

return 0;

}

该代码形成了一个网络看守（比如用于医学成像的分布式数据库，或分布式文件服务器）的主事件

循环部分。在代码中，主事件循环等待消息从客户到达。当消息到达时，主线程通过 dequeue 方法将它从 消息队列中移除。然后取决于接收到的消息的类型，线程执行某种处理（例如，图像数据库查询、文件 更新，等等）。request\_count 变量追踪到来的客户请求的数量。该信息可用于更新 SNMP MIB 中的属性。

只要 run\_svc 在单线程控制中执行，上面所示的代码工作良好。但是，当 run\_svc 由运行在不同的 P E 上的多线程控制同时执行时，在许多多处理器平台上将会产生不正确的结果。这里的问题是这些代码 并非是“线程安全”的，因为针对全局变量 request\_count 的自增操作含有一个竞争状态。因而，不同的 线程可能会增加存储在它们自己的 PE 数据缓存中的 request\_count 变量的陈旧版本。

这一现象可通过执行下面的例 4-1 中的 C++代码来演示，运行环境为一台运行 Solaris 2.x 操作系统 的共享内存多处理器。Solaris 2.x 允许多线程控制在共享内存多处理器上并行执行。下面所示的例子是上 面演示的网络看守的简化版本：

例 4-1

typedef u\_long COUNTER;

static COUNTER request\_count; // At file scope

void \*run\_svc (int iterations)

{

for (int i = 0; i < iterations; i++)

++request\_count; // Count # of requests

return (void \*) iterations;

}

typedef void \*(\*THR\_FUNC)(void \*);

// Main driver function for the

// multi-threaded server.

int main (int argc, char \*argv[])

{

int n\_threads = argc > 1 ? atoi (argv[1]) : 4;

int n\_iterations = argc > 2 ? atoi (argv[2]) : 1000000;

thread\_t t\_id;

// Divide iterations evenly among threads. int iterations = n\_iterations / n\_threads;

// Spawn off N threads to run in parallel. for (int i = 0; i < n\_threads; i++)

thr\_create (0, 0, THR\_FUNC (&run\_svc), (void \*) iterations,

THR\_BOUND | THR\_SUSPENDED,

&t\_id);

// Resume all suspended threads

// (threads id’s are contiguous...)

for (i = 0; i < n\_threads; i++)

thr\_continue (t\_id--);

// Wait for all threads to exit. int status;

while (thr\_join (0, &t\_id, (void \*\*) &status) == 0)

cout << "thread id = " << t\_id

<< ", status = " << status << endl;

cout << n\_iterations << " = iterations\n"

<< request\_count << " = request count"

<< endl;

return 0;

}

Solaris thr\_create 线程库例程被调用 n\_thread 次，以派生 n 个新线程控制。在此例中，每个新创建的

线程都传递 iterations 的值给 run\_svc 函数，作为它唯一的参数，并执行之。该值使得 run\_svc 例程循环 n

\_iterations/n\_threads 次。

每个线程都使用 THR\_BOUND 和 THR\_SUSPENDED 标志来派生。THR\_BOUND 通知 Solaris 线程 运行时库将该线程绑定到专用的 LWP。每个 LWP 可以在一个多处理器系统中的单独的 PE 上并行运行。 THR\_SUSPENDED 标志创建“挂起”状态的线程，确保在调用 thr\_continue 恢复（resume）测试之前， 所有线程被完全地初始化。thr\_continue 函数是一个 Solaris 线程库例程，可恢复挂起线程的执行。注意此 例利用了 Solaris 是以升序连续分配线程 id 的事实。

一旦所有线程都已被恢复，thr\_join 例程就阻塞主线程的执行。thr\_join 与 UNIX wait 系统调用相类 似——它获取退出线程的状态。thr\_join 将“收割”线程，并返回 0，直到运行 run\_svc 的线程全部退出为 止。当所有其他的线程退出后，主线程打印出 iterations 的总数，以及 request\_count 的最后值，然后退出 程序。

将此代码编译成可执行的 a.out 文件，并以 1 个线程循环 10,000,000 次的方式运行它，得到如下结果：

% a.out 1 10000000

thread id = 4, status = 1000000

10000000 = iterations

10000000 = request count

该结果正如所愿。但是，当以 4 个线程循环 10,000,000 次的方式在 4 个 PE 的机器上运行时，程序打 印出：

% a.out 4 10000000

thread id = 5, status = 1000000 thread id = 7, status = 1000000 thread id = 6, status = 1000000 thread id = 4, status = 1000000

10000000 = iterations

5000000 = request count

显然，有什么出错了，因为全局变量 request\_count 的值只是循环总数的一半。这里的问题是变量 re quest\_count 的自增没有被正确地序列化。

一般而言，在不提供“强有序缓存一致性模型”（strong sequential order cache consistency model） 的共享内存多处理器平台上并行执行时，run\_svc 会产生不正确的结果。为增强性能，许多共享内存多处 理器采用“弱有序缓存一致性语义”（weakly-ordered cache consistency semantics）。例如，SPARC 多处 理器的 V.8 和 V.9 家族同时提供“总存储序”（total store order）和“部分存储序”（partial store order） 内存缓存一致性语义。对于总存储序语义，对正在被其他 PE 上的线程访问的变量的读取和同时进行的对 同一变量的写也许不会被序列化。同样地，对于部分存储序语义，写操作与写操作也可能不会被序列化。 在任一情形中，由于多 PE 间的缓存延迟，需要对内存位置进行不止一次装载和存储的表达式（比如 foo

++或 i = i – 10）可能会产生不一致的结果。为确保线程间共享的变量的读写被正确更新，程序员必须 人工地保证对这些变量的变动成为全局可见的。

在“总存储序”或“部分存储序”共享内存多处理器上强制实现强有序的一种常用技术是使用同步 机制来保护 request\_count 变量的增长。Solaris 2.x 提供若干种同步机制。本论文描述 Solaris 2.x 提供的 四种主要同步机制的 C++包装：互斥体、读者*/*作者锁、计数信号量，以及条件变量[19]。ACE 含有封装 这四种 Solaris 2.x 同步机制（分别为 mutex\_t、rwlock\_t、sema\_t，以及 cond\_t）的 C++包装。在 4.3 的余 下部分我们将概述 Solaris 同步机制的行为。4.4 演示 C++包装的使用，用以简化常见的同步变量的使用， 并改善程序的可靠性。

**4.3.5.1** 互斥锁

互斥锁（通常称为“互斥体”或“二元信号量”）用于保护多线程控制并发访问的共享资源的完整性。 互斥体通过定义临界区来序列化多线程控制的执行，在临界区中每一时刻只有一个线程在执行它的代码。 互斥体简单（例如，只有拥有该互斥体的线程可以释放它）而高效（时间和空间）。

像 Solaris 2.x 这样的操作系统中的互斥体变量的操作通过可适配回旋锁（spin-locks）来实现。回旋 锁通过使用一条原子的硬件指令来确保互斥。对于特定类型的短暂资源争用（像上面的例 4-1 演示的全局 request\_count 变量的自增），它是简单而高效的同步机制。可适配回旋锁使用原子的硬件指令来轮询指定 的内存位置，直到下面的情况之一发生[2]：

z 该位置的值被当前拥有该锁的线程改变。这表示锁已被释放，并可以被回旋的线程获取。

z 持有该锁的线程进入睡眠。此时，回旋线程也让自己进入睡眠，以避免不必要的轮询。

在多处理器上，回旋锁带来的开销相对较小。基于硬件的轮询不会导致系统总线上的争用，因为它仅仅 影响在互斥体上回旋的线程的本地 PE 缓存。

互斥体的一种简单而高效的类型是“非递归”互斥体。非递归互斥体不允许当前拥有互斥体的线程 在释放它之前重新获取它。否则，将会立即发生死锁。Solaris 2.x 和 POSIX pthreads 通过 mutex\_t 数据类 型和相关的 mutex\_lock 和 mutex\_unlock 函数来实现非递归互斥体。

POSIX pthreads 和 Win32 线程都实现了递归和非递归互斥体（其他类型的互斥体在 4.4.4 讨论）。如

4.5.1.4 所描述的，ACE OO 线程封装库提供互斥体 C++封装，以可移植地实现非递归互斥体语义。非递 归互斥体在 ACE Recursive\_Thread\_Mutex 类中可移植地实现。

**4.3.5.2** 读者**/**作者锁

读者/作者锁与互斥体相类似。例如，获取读者/作者锁的线程也必须释放它。多个线程可同时获取一 个读者/作者锁用于读，但只有一个线程可以获取该锁用于写。当互斥体保护的资源用于读远比用于写要 频繁时，读者/作者互斥体有助于改善并发的执行。

Solaris 2.x 通过它的 rwlock\_t 类型来支持读者/作者互斥体。无论是 POSIX pthreads，还是 Win32 线 程都没有本地的读者/作者锁支持。如 4.5.1.3 所描述的，ACE 线程库提供了一个叫作 RW\_Mutex 的类， 在 C++封装类中可移植地实现了读者/作者锁的语义。读者/作者锁的 Solaris 和 ACE 实现都将优先选择权 给作者。因而，如果有多个读者和一个作者在锁上等待，作者将会首先获取它。

**4.3.5.3** 计数信号量

在概念上，计数信号量是可以原子地增减的整数。如果线程试图减少一个值为零的信号量的值，它 就会阻塞，直到另一个线程增加该信号量的值。

计数信号量用于追踪共享程序状态的变化。它们记录某种特定事件的发生。因为信号量维护状态， 它们允许线程根据该状态来作决定，即使事件是发生在过去。

信号量比互斥体效率要低，因为它们保持额外的状态，并使用休眠锁，而不是回旋锁。但是，它们 要更为通用，因为它们无需被最初获取它们的同一线程获取和释放。这使得它们能够用于异步的执行上 下文中（比如信号处理器）。

Solaris 2.x 通过它的 sema\_t 类型支持信号量。Win32 将信号量作为 HANDLE 来支持。POSIX pthrea ds 没有本地的信号量支持。如 4.5.1.2 所描述的，ACE 线程库提供一个叫作 Semaphore 的类来可移植地在 C++包装类中实现信号量语义。

**4.3.5.4** 条件变量

条件变量提供风格与互斥体、读者/作者锁和计数信号量不同的锁定机制。当持有锁的线程在临界区 执行代码时，这三种机制让协作线程进行等待。相反，条件变量通常被一个线程用于使自己等待，直到 一个涉及共享数据的条件表达式到达特定的状态。当另外的协作线程指示共享数据的状态已发生变化， 调度器就唤醒一个在该条件变量上挂起的线程。于是新唤醒的线程重新对它的条件表达式进行求值，如 果共享数据已到达合适状态，就恢复处理。

被条件变量等待的条件表达式可以任意地复杂。一般而言，与其他同步机制相比，条件变量允许更 为复杂的调度决策。条件变量同步使用休眠锁来实现，休眠锁触发上下文切换，并允许另一线程执行， 直到锁被获取。如 4.3.5.1 所描述的，互斥体使用可适配回旋锁来实现。如果线程必须长时间地等待某特 定条件实现，回旋锁就会消耗过多的资源。

对于涉及条件表达式语义的情况，条件变量比信号量或互斥体要更为有用。在这种情况下，等待线 程必须阻塞到特定的涉及共享状态的条件表达式变为真（例如，表不再为空和网络流控制减轻）。在这种 情况下，不需要维护事件历史。因而，条件表达式不记录它们什么时候被“置位”。如果没有正确地使用，

这可能会导致“丢失的苏醒”（lost wakeup）问题[19]。

Solaris 2.x 和 POSIX pthreads 通过 cond\_t 类型支持条件变量。本地的 Win32 API 不支持条件变量。 如 4.5.3.1 所描述的，ACE 线程库提供一个叫作 Condition 的类来可移植地在 C++包装类中实现条件变量 语义。

**4.3.6** 进程 **vs.** 线程同步语义

为增加灵活性和改善性能，Solaris 2.x 提供两种风格的同步语义，并为下列两种情况而优化：（1）在 同一进程中执行的线程（也就是，进程内序列化）以及（2）在相互分离的进程中执行的线程（也就是， 进程间序列化）。在 Solaris 2.x 中，同步机制的\*\_init 函数的 USYNC\_THREAD 标志创建为单个进程中的 线程而优化的变量。同样地，USYNC\_PROCESS 标志创建跨进程有效的同步变量。后一种类型的同步机 制更为通用，尽管略为低效，如果所有线程是在单一进程中运行的话。

**4.3.7** 互斥体例子

下面的代码演示怎样将 Solaris 互斥体变量用于解决我们先前考察的 request\_count 的自增序列化问 题：

例 4-2

typedef u\_long COUNTER;

// At file scope

static COUNTER request\_count;

// mutex protecting request\_count (initialized to zero). static mutex\_t m;

void \*run\_svc (void \*)

{

for (int i = 0; i < iterations; i++)

{

mutex\_lock (&m); // Acquire lock

++request\_count; // Count # of requests mutex\_unlock (&m); // Release lock

}

return (void \*) iterations;

}

在上面的代码中，m 是 mutex\_t 类型的全局变量。在 Solaris 中，凡是被置零的同步变量都使用它的缺省

语义来初始化。例如，mutex\_t 变量 m 总是被初始化成从未锁定状态开始。因此，当 mutex\_lock 第一次 被调用时，它将获得该锁的所有权。任何其他想要获取该锁的线程都必须等待（例如，通过“回旋”）， 直到锁的属主释放 m。

上面所示的例 4-2 解决了原来的同步问题，但它仍具有以下缺陷：

z 不优雅和不一致：代码混合了 C 函数和 C++代码，以及不同的标识符命名习惯。使用混合的编程风 格让人困惑，还可能带来维护问题。

z 强制性：该方案要求改变源代码。在开发大型软件系统时，人工地进行这样的变动会导致维护问题， 如果所有变动没有一致地进行的话。

z 不可移植：该代码只能与 Solaris 2.x 同步机制一起工作。特别地，移植该代码以使用 POSIX pthread s 和 Windows NT 线程将需要改变锁定代码。

z 易错：程序员很容易忘记调用 mutex\_unlock。这会使其他正试图获取该锁的线程饿死。而且，如果锁 的属主试图重新获取它已经拥有的互斥体，就会发生死锁。

程序员还可能忘记初始化互斥体变量。如上面所提到的，在 Solaris 2.x 中，被置零的 mutex\_t 变 量被隐含地初始化。但是，并不能保证在动态分配的结构或类中作为域被分配的 mutex\_t 变量也被隐 含地初始化。而且，其他线程机制（比如 POSIX pthreads 和 Windows NT 线程）并没有这样的初始 化策略，所有同步对象都必须显式地初始化。

在 4.4 我们将检查通过改善 Solaris 同步机制的功能、可移植性和健壮性，C++包装的使用是怎样帮助 克服这些问题的。

**4.4** 通过 **OO** 和 **C++**简化并发编程

这一部分检查一个使用实例，以演示在 C++包装中封装 Solaris 并发机制的优点。该使用实例描述了 一个基于生产系统的有代表性的情景[25]。紧跟 4.5 对库接口的介绍，在 4.6 中还有 ACE OO 线程封装类 库的其他例子。

通过归纳系统开发中发生的实际问题的解决方案，许多有用的 C++类已逐渐发展起来。但是在类的 接口和实现稳定后，随着时间的过去，这样的反复对类进行归纳的过程已不再被强调。这让人遗憾，因 为要进入面向对象设计和 C++的主要障碍是（1）学习并使怎样识别和描述类和对象的过程内在化，以及

（2）理解何时以及怎样应用（或不应用）像模板、继承、动态绑定和重载这样的 C++特性来简化和归纳 他们的程序。

为努力把握 C++类设计演变的动力特性，下面的部分演示逐步应用面向对象技术和 C++习语、以解 决一个惊人地微妙的问题的过程。该问题发生在开发并发分布式应用族的过程中，该应用族在单处理器 和多处理器平台上都能高效地执行。通过使用模板和重载来透明地将同步机制参数化进并发应用，这一 部分集中考察那些涉及对已有代码进行归纳的步骤。其基础代码基于在[1, 26, 20]中描述的自适配通信环 境（ACE）框架中的组件。

此例子检查若干 C++的语言特性，它们可以更为优雅地解决 4.3.5.1 提出的序列化问题。如在那里所 描述的，原来的方案既不优雅、不可移植、易错，并且还要求强制性地改变源代码。这一部分演示一种 进化的、在先前反复的设计演变中所产生的洞见之上构建的 C++方案。

**4.4.1** 初始 **C++**方案

解决原来问题的一种更为优雅一点的方案是通过 C++ Thread\_Mutex 包装来封装现有的 Solaris mute x\_t 操作，如下所示：12

class Thread\_Mutex

{

public:

Thread\_Mutex (void)

{

mutex\_init (&lock\_, USYNC\_THREAD, 0);

}

˜Thread\_Mutex (void)

{

mutex\_destroy (&lock\_);

}

int acquire (void)

{

return mutex\_lock (&lock\_);

}

int release (void)

{

return mutex\_unlock (&lock\_);

}

private:

// Solaris 2.x serialization mechanism. mutex\_t lock\_;

};

给互斥机制定义 C++包装接口的一个优点是应用代码现在变得更为可移植了。例如，下面是 Thread\_Mut ex 类接口的实现，它基于 Windows NT WIN32 API[4]中的机制之上：13

class Thread\_Mutex

{

public:

12 在本论文中，所演示的许多 C++类的例子都在类的定义中实现其方法。这样的风格只是用于说明，不应该在开发应用时 使用。

13 注意该实现仅仅支持单进程中的互斥体。ACE 还使用其他 Win32 机制实现了进程级互斥体。

Thread\_Mutex (void)

{

InitializeCriticalSection (&lock\_);

}

˜Thread\_Mutex (void)

{

DeleteCriticalSection (&lock\_);

}

int acquire (void)

{

EnterCriticalSection (&lock\_); return 0;

}

int release (void)

{

LeaveCriticalSection (&lock\_); return 0;

}

private:

// Win32 serialization mechanism. CRITICAL\_SECTION lock\_;

};

使用 Thread\_Mutex C++包装类使原来的代码变得更为清晰，改善了可移植性，并且确保了 Thread\_ Mutex 对象被定义时，初始化工作自动地进行。如下面的代码段所示：

例 4-3

typedef u\_long COUNTER;

// At file scope.

static COUNTER request\_count;

// Thread\_Mutex protecting request\_count. static Thread\_Mutex m;

void \*run\_svc (void \*)

{

for (int i = 0; i < iterations; i++)

{

m.acquire ();

// Count # of requests.

++request\_count;

m.release ();

}

return (void \*) iterations;

}

但是，C++封装方法并没有解决所有在 4.3.5.1 所标出的问题。特别地，它没有解决忘记释放互斥体的问

题（它还是需要程序员的人工干预）。此外，使用 Thread\_Mutex 也还是需要对原来的非线程安全的代码 进行强制性的改变。

**4.4.2** 另一种 **C++**方案

确保锁被自动释放的一种直截了当的方法是使用 C++类构造器和析构器语义。下面的工具类使用这 些语言构造来自动进行互斥体的获取和释放：

class Guard

{

public:

Guard (const Thread\_Mutex &m): lock\_ (m)

{

lock\_.acquire ();

}

˜Guard (void)

{

lock\_.release ();

}

private:

const Thread\_Mutex &lock\_;

}

Guard 定义了一“块”代码，在其上一个 Thread\_Mutex 被自动获取，并在退出代码块时自动释放。 它采用了一种通常称为“作为资源获取的构造器—作为资源释放的析构器”[9, 27, 7]的 C++习语。

如上面的代码所示，当 Guard 类的对象被创建时，它的构造器自动获取 Thread\_Mutex 对象上的锁。 同样地，Guard 类的析构器在对象出作用域时自动解锁 Thread\_Mutex 对象。

注意 Guard 类的数据成员 lock\_是 Thread\_Mutex 对象的一个引用。这避免了在 Guard 对象的构造器 和析构器每次执行时创建和销毁底层 Solaris mutex\_t 变量的开销。

通过对代码作出轻微的变动，我们现在保证了 Thread\_Mutex 被自动获取和释放：

例 4-4

typedef u\_long COUNTER;

// At file scope.

static COUNTER request\_count;

// Thread\_Mutex protecting request\_count. static Thread\_Mutex m;

void \*run\_svc (void \*)

{

for (int i = 0; i < iterations; i++)

{

{

// Automatically acquire the mutex. Guard monitor (m);

++request\_count;

// Automatically release the mutex.

}

// Remainder of service processing omitted.

}

}

但是，该方案还是没有解决强制性改变代码的问题。而且，在 Guard 周围增加额外的‘{’和‘}’花括

号分隔符块既不优雅又容易出错。进行维护的程序员可能会误认为花括号并不重要并将它们去除，产生 出下面的代码：

for (int i = 0; i < iterations; i++)

{

Guard monitor (m);

++request\_count;

// Remainder of service processing omitted.

}

遗憾的是，这样的“花括号省略”有副作用：它通过序列化主事件循环、消除了应用中所有并发执行。

因此，所有应该在那段代码区中并行执行的计算都会被不必要地序列化。

**4.4.3** 改良的 **C++**方案

要以一种透明的、非强制的和高效的方式解决现存的问题，需要使用两种另外的 C++特性：参数化 类型和操作符重载。我们使用这些特性来提供一个称为 Atomic\_Op 的模板类，其部分代码显示如下（完 整的接口见 4.5.6.2）：

template <class TYPE>

class Atomic\_Op

{

public:

Atomic\_Op (void) { count\_ = 0; } Atomic\_Op (TYPE c) { count\_ = c; } TYPE operator++ (void)

{

Guard monitor (lock\_);

return ++count\_;

}

operator TYPE ()

{

Guard monitor\_ (lock\_);

return count\_;

}

// Other arithmetic operations omitted...

private:

Thread\_Mutex lock\_; TYPE count\_;

};

Atomic\_Op 类重新定义了普通的针对内建数据类型的算术操作符（比如++、--、+=，等等），以使 这些操作符原子地工作。一般而言，由于 C++模板的“延期实例化”语义，任何定义了基本算术操作符 的类都将与 Atomic\_Op 类一起工作。

因为 Atomic\_Op 类使用了 Thread\_Mutex 的互斥特性，针对 Atomic\_Op 的实例化对象的算术运算现 在在多处理器上工作正常。而且，C++特性（比如模板和操作符重载）还允许这样的技术在多处理器上透 明地工作。此外，Atomic\_Op 中的所有方法操作都被定义为内联函数。因此，一个 C++优化编译器将生 成代码确保 Atomic\_Op 的运行时性能不会低于直接调用 mutex\_lock 和 mutex\_unlock 函数。

使用 Atomic\_Op 类，我们现在可以编写下面的代码，几乎等同于原来的非线程安全代码（实际上， 只是改变了 COUNTER 的类型定义）：

例 4-5

typedef Atomic\_Op<u\_long> COUNTER;

// At file scope

static COUNTER request\_count;

void \*run\_svc (void \*)

{

for (int i = 0; i < iterations; i++)

{

// Actually calls Atomic\_Op::operator++()

++request\_count;

}

}

通过结合 C++构造器/析构器习语（以自动获取和释放 Thread\_Mutex）和模板及重载的使用，我们生

成了一种既简单又非常有表现力的参数化类抽象。该抽象可在无数需要原子操作的类型族上正确而原子 地运作。例如，要为其他算术类型提供同样的线程安全功能，我们只需简单地实例化 Atomic\_Op 模板类 的新对象：

Atomic\_Op<double> atomic\_double; Atomic\_Op<Complex> atomic\_complex;

**4.4.4** 通过参数化互斥机制类型扩展 **Atomic\_Op**

尽管上面描述的 Atomic\_op 和 Guard 类的设计产生了正确和透明的线程安全程序，还是存在着足资 改进的空间。特别地，注意 Thread\_Mutex 数据成员的类型是被硬编码进 Atomic\_Op 类的。既然在 C++ 中可以使用模板，这样的设计决策就是一种不必要的、可以轻易克服的限制。解决方案就是参数化 Guar d，并增加另一种类型参数给模板类 Atomic\_Op，如下所示：

template <class LOCK>

class Guard

{

// Basically the same as before... private:

// new data member change.

const LOCK &lock\_;

};

template <class LOCK, class TYPE>

class Atomic\_Op

{

TYPE operator++ (void)

{

Guard<LOCK> monitor (lock\_);

return ++count\_;

}

// ...

private:

LOCK lock\_; // new data member

TYPE count\_;

};

使用这个新类，我们可以在源代码的开始处作出下面的简单变动：

typedef Atomic\_Op <Thread\_Mutex, u\_long> COUNTER;

// At file scope. COUNTER request\_count;

// ... same as before

**4.4.5** 设计原理和性能问题

在作出上面描述的变动之前，很值得分析一下使用模板来参数化程序所用的互斥机制类型的动机是 否是有益的。毕竟，仅仅因为 C++支持模板，并不意味着它们在所有的环境中就都有用。事实上，通过 模板来参数化和归纳问题空间，而又缺少清晰而足够的理由，可能会增加理解和复用 C++类的难度。

Atomic\_Op 类中模板的使用引发了若干问题。第一个是“所有增加的抽象的运行时性能代价是什么？” 第二个问题是“除了模板，为什么不使用继承和动态绑定来强调统一的互斥体接口并共享通用的代码？” 第三是“程序的同步属性不是因为使用模板和重载而变得不明晰了吗？”若干这样的问题是相关的，本 节将讨论涉及不同设计选择的折衷。

**4.4.5.1** 性能

为 Atomic\_Op 类选用模板的主要原因涉及运行时效率。在模板实例化的过程中，一旦优化 C++编译 器将其展开，额外的运行时开销很少，或根本不存在。相反，继承和动态绑定会带来用以分派虚方法调 用的运行时开销。

图 4-3 演示例 4-2 到例 4-5 中使用的互斥技术所展现的性能14。该图描述处理 1 千万次迭代所需的秒 数，迭代次数被划分为 2.5 百万次迭代/线程。测试例子使用 Sun C++ 3.0.1 编译器的-O4 优化级编译。每 个测试在另外一台闲置的 4 PE Sun SPARCserver 690MP 上执行 10 次。结果被平均，以减少虚假的偏差 量（此数量被证明是微不足道的）。

14 例 4-1 是最初的错误实现，没有使用任何互斥操作。尽管它执行起来极其高效（大约 0.09 秒处理 10,000,000 次迭代）， 它产生的结果完全是不正确的！

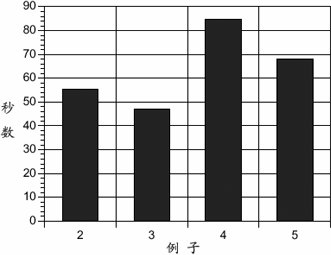


图 4-3 处理 10,000,000 次迭代所需秒数

例 4-2 直接使用 Solaris mutex\_t 函数。例 4-3 使用 C++ Thread\_Mutex 类包装接口。令人惊讶的是， 该实现始终都优于例 4-1，而例 4-1 是直接调用底层 Solaris 互斥体函数。例 4-4 在一对花括号块中使用 G uard 助手类，以确保 Thread\_Mutex 被自动释放。该版本所需的执行时间最多。最后，例 4-5 使用 Atomi c\_Op 模板类，比直接使用 Solaris 互斥体函数的只稍稍低效一点。更为强势的优化 C++编译器可能还会减 少这些结果的偏差量。

|  |  |  |
| --- | --- | --- |
| 例子 | 微秒**/**操作 | 比率 |
| 例 4-2 | 2.76 | 1 |
| 例 4-3 | 2.35 | 0.85 |
| 例 4-4 | 4.24 | 1.54 |
| 例 4-5 | 3.39 | 1.29 |

表 4-1 不同例子的序列化时间

表 4-1 显示从例 4-2 到例 4-5 每次互斥操作所花费的微秒数。注意每次循环迭代需要 2 次互斥体操作

（一次获取锁，一次释放锁）。例 4-2 被用作基准值，因为它直接使用底层 Solaris 原语。例 4-3 到例 4-

5 的第三列通过将它们的值除以例 4-2 的值来进行规格化。

**4.4.5.2** 可移植性

参数化互斥机制类型的一个动机是增强跨平台可移植性。模板使形式的参数类名“Thread\_Mutex” 与实际用于提供互斥的类的名字去耦合。这对于已经使用符号 Thread\_Mutex 来表示现有类型或功能的平 台来说是有用的。在移植到这样的平台时，通过使用模板，不需要对 Atomic\_Op 类源代码进行任何改变。

但是，更为有趣的动机来自于对下述事实的观察：一个人可能实际上想使用若干不同风格的互斥体 语义（在同一程序中，或跨越一个相关程序族）。每个这样的互斥风格共享同一基本的获取和释放协议， 但它们持有拥有不同的序列化和性能属性。4.5.1.1 介绍许多已在实践中被证明为有用的互斥机制。

**4.4.5.3** 透明性

一种反对使用模板来参数化同步的意见认为透明度隐藏了程序的互斥语义。这被认为是“臭虫”还 是“特性”取决于开发者是否认为并发和同步应被集成进程序中。对于包含基本的“积木”组件（比如 4.

5.1.1 描述的 Map\_Manager）的类库，常常希望允许同步语义被参数化，因为这使得开发者能够精确地控 制和指定他们想要的并发语义。这一策略的替换方案有：（1）如果使用多线程，就不使用类库（这显然 限制了功能），（2）在库的外面完成锁定（这可能是低效或不安全的），或者（3）将锁定策略硬编码 进库的实现（这也是不灵活和潜在地低效的）。所有这些方案都是违背面向对象软件系统中的复用原则 的。

**4.4.5.4** 折衷评估

选择适当的设计策略来开发支持并发的类库依赖于若干因素。例如，某些库用户可能会欢迎简单的、 看上去隐藏了并发控制机制的接口。相反，其他库用户可能愿意接受更复杂的接口，以获取额外的控制 和更高的效率。一种分层的类库设计方法也许可以同时满足这两种用户。使用这样的设计方法，类库的 最底层输出大多数或全部的参数化类型作为模板参数。较高层则提供合理的缺省类型值，并提供易于使 用的应用开发者编程接口。

最近被 ANSI C++委员会采纳的新的“缺省模板参数”特性便利了同时满足两种类型库用户的类库 的开发。该特性允许库开发者指定常用的缺省类型作为模板类和函数定义的参数。例如，下面对模板类 A tomic\_Op 的修改为之提供了常用的缺省模板参数：

template <class LOCK = Thread\_Mutex, class TYPE = u\_long>

class Atomic\_Op

{

// Same as before

};

// ...

#if defined (MT\_SAFE)

// Default is Thread\_Mutex and u\_long. typedef Atomic\_Op<> COUNTER;

#else

// Don’t serialize.

typedef Atomic\_Op<Null\_Mutex> COUNTER;

#endif /\* MT\_SAFE \*/ COUNTER request\_count;

由于将并发结合进应用所带来的复杂性，C++模板对于减少多余的开发工作相当有用。但是，就像任

何其他的语言特性一样，也有可能误用模板，不必要地使系统的设计和实现变得复杂化（不用说还会增 加编译和链接时间）。在决定是否使用参数化类型时，可以使用一种启发式方法：留意何时现有代码将 要被复制，而所做修改仅涉及数据类型。如果有另一种合理的情况，需要只是类型不同的第三种版本， 这就说明可能值得对原始代码进行归纳来使用模板。

**4.5** 公共接口和内部设计

这一部分描述 ACE OO 线程封装库中组件的公共接口和相关的内部设计情况。ACE 组件被划分为下 面的组：

z 低级 *C++*线程 *API*：为底层的 OS 线程和同步 API 提供低级 C++封装。低级 C++线程 API 包含在被 称为 OS 的类中。该类封装了各种版本的 UNIX 以及 WIN32 之间的所有差异。ACE 中的其他组件被 编写为只使用该类中的方法，从而使得将 ACE 移植到新的平台上变得更为容易。

z 高级 *C++*线程库：允许多线程应用被编写为使用更高级的 C++特性，像构造器/析构器和模板。高级

C++线程 API 根据低级的 ACE OS 类来编写。它们被划分为三个组：

－锁定机制（在 4.5.1 描述）的 C++包装。

－本地线程函数（在 4.5.6.1 描述）的 C++包装。

－更高级的线程管理类（在 4.5.4.1 描述）。

ACE OO 线程组件的一个使用实例已在 4.4 中介绍。这一部分的余下部分对 ACE 的公共接口、功能 和内部设计进行更为全面的讨论。在适当的地方，这一部分描述该 C++包装类的私有部分，以演示包装 是如何映射到 Solaris 2.x 线程和同步机制上的。POSIX pthreads 和 Win32 包装的实现是类似的。

**4.5.1 ACE** 锁类属

ACE C++包装给 4.3.5 描述的 Solaris、POSIX 和 Win32 OS 同步机制提供了一种可移植、线程安全 和面向对象的接口。下面的条目概述了这些 ACE C++封装的主要好处：

z 提高正确性：通过使作为 C++类和结构中的域出现的同步对象的初始化自动化，以及通过保证锁被自 动获取和释放来实现。

z 统一的同步接口：所有线程和同步的 C++包装都为获取和释放多种类型的锁提供了统一的接口。特别 地，ACE 锁类属中的所有组件都支持四种通用方法：acquire、try\_acquire、release 和 remove。这种统 一性使得开发者有可能使用锁类作为类型参数，来与其他的 ACE 同步组件（比如在 4.5.2.1、4.5.6.2 和 4.5.1.4 定义的那些组件）联合使用。

z 更为直观的错误报告：Solaris 2.x 和 POSIX pthreads 同步函数使用一种不那么标准的机制来将错误返 回给调用者。相反，ACE 包装使用一种更为标准的方法：如果发生失败就返回-1，并设置 errno 来指

示失败的原因。

z 简化常见使用模式：包装简化了低级线程和同步机制的常见使用模式。所示代码通过使用 mutex\_t 和 cond\_t 的 ACE C++包装实现一个简单版本的 Dijkstra 计数信号量（也就是，P 和 V 分别等价于 acqui re 和 release）演示了这一点。

class Semaphore

{

public:

Semaphore (int initial\_value)

: count\_nonzero\_ (lock\_)

{

// Automatically acquire lock. Guard<Thread\_Mutex> monitor (lock\_);

count\_ = initial\_value;

// Automatically release the lock

}

// Block the thread until the semaphore

// count becomes greater than 0,

// then decrement it. void acquire (void)

{

// Automatically acquire lock

Guard<Thread\_Mutex> monitor (lock\_);

// Wait until semaphore is available. while (count\_ == 0)

count\_nonzero\_.wait ();

count\_ = count\_ - 1;

// Automatically release the lock

}

// Increment the semaphore, potentially

// unblocking a waiting thread. void release (void)

{

// Automatically acquire lock

Guard<Thread\_Mutex> monitor (lock\_);

// Allow waiter to continue. if (count\_ == 0)

count\_nonzero\_.signal ();

count\_ = count\_ + 1;

// Automatically release the lock

}

private:

Thread\_Mutex lock\_; Condition<Thread\_Mutex> count\_nonzero\_; u\_int count\_;

};

注意 Condition 对象 count\_nonzero\_的构造器是怎样将 Thread\_Mutex 对象 lock\_和 Condition 对象绑定 在一起的。这简化了 Condition::wait 调用接口。相反，本地的 Solaris 和 pthreads cond\_t cond\_wait 接口要 求每次调用 wait 时都传递一个互斥体作为参数。

Solaris 2.x 和 Win32 提供一种内建的计数信号量实现（见 4.3.5.3 的讨论）。但是，POSIX pthreads[3] 线程库没有包含信号量。因此，上面所示的类既演示了 ACE C++包装的使用，又为 ACE 线程封装库中 P OSIX pthreads 的可移植 Semaphore 实现提供了文档。

**4.5.1.1** 互斥体类

ACE 互斥体包装提供了一种简单而高效的机制来序列化对共享资源的访问。它们封装 Solaris 和 POS IX pthreads mutex\_t 同步变量，以及 Win32 的基于 HANDLE 的互斥体实现。Mutex 的类定义如下所示：

class Mutex

{

public:

// Initialize the mutex.

Mutex (int type = USYNC\_THREAD);

// Implicitly destroy the mutex.

˜Mutex (void);

// Explicitly destroy the mutex. int remove (void);

// Acquire lock ownership (wait

// for lock to be released). int acquire (void) const;

// Conditionally acquire lock

// (i.e., don’t wait for lock

// to be released).

int try\_acquire (void) const;

// Release lock and unblock

// the next waiting thread. int release (void) const;

private:

mutex\_t lock\_;

// Type of synchronization lock.

};

在 ACE 中，线程可以通过调用 Mutex 对象的 acquire 方法来进入临界区。任何对该方法的调用都将 会阻塞，直到当前拥有该锁的线程离开它的临界区。要离开临界区，线程调用它当前拥有的 Mutex 对象 的 release 方法。调用 release 使得另一个阻塞在该互斥体上的线程能够进入它的临界区。

Thread\_Mutex 和 Process\_Mutex 类继承自 Mutex，并使用它的构造器来创建适当类型的互斥体，如下：

class Thread\_Mutex : public Mutex

{

public:

Thread\_Mutex (void): Mutex (USYNC\_THREAD);

};

class Process\_Mutex : public Mutex

{

public:

Thread\_Mutex (void): Mutex (USYNC\_PROCESS);

};

这些调用被映射到适当的底层 API 上，以分别创建线程和进程专用的互斥体。特别地，Thread\_Mutex 的 Win32 实现使用更为高效、但却不那么强大的 CRITICAL\_SECTION 实现，而 Process\_Mutex 实现则使用 较为低效、但却更为强大的 Win32 互斥体 HANDLE。

**4.5.1.2** 信号量类

ACE 信号量包装类实现 Dijkstra 的“计数信号量”抽象，这是一种用于序列化多个线程控制的通用 机制。它们封装 Solaris sema\_t 同步变量。Semaphore 类接口如下所示：

class Semaphore

{

public:

// Initialize the semaphore,

// with default value of "count".

Semaphore (u\_int count,

int type = USYNC\_THREAD, void \* = 0);

// Implicitly destroy the semaphore.

˜Semaphore (void);

// Explicitly destroy the semaphore. int remove (void);

// Block the thread until the semaphore count

// becomes greater than 0, then decrement it. int acquire (void) const;

// Conditionally decrement the semaphore if

// count greater than 0 (i.e., won’t block). int try\_acquire (void) const;

// Increment the semaphore, potentially

// unblocking a waiting thread. int release (void) const;

private:

sema\_t semaphore\_;

};

Thread\_Semaphore 和 Process\_Semaphore 类继承自 Semaphore，并使用它的构造器来创建适当类型的 信号量，如下所示：

class Thread\_Semaphore : public Semaphore

{

public:

Thread\_Semaphore (void): Semaphore (USYNC\_THREAD);

};

class Process\_Semaphore : public Semaphore

{

public:

Thread\_Semaphore (void): Semaphore (USYNC\_PROCESS);

};

**4.5.1.3 RW\_Mutex** 类

ACE 读者/作者包装序列化对这样一种资源的访问：其内容被搜索要多于被变动。它们封装 rwlock\_t 同步变量，这种变量在 Solaris 上在本地实现，而在 Win32 和 Pthreads 上则由 ACE 模拟。RW\_Mutex 接 口如下所示：

class RW\_Mutex

{

public:

// Initialize a readers/writer lock. RW\_Mutex (int type = USYNC\_THREAD,

void \*arg = 0);

// Implicitly destroy a readers/writer lock.

˜RW\_Mutex (void);

// Explicitly destroy a readers/writer lock. int remove (void);

// Acquire a read lock, but

// block if a writer hold the lock. int acquire\_read (void) const;

// Acquire a write lock, but

// block if any readers or a

// writer hold the lock.

int acquire\_write (void) const;

// Conditionally acquire a read lock

// (i.e., won’t block).

int try\_acquire\_read (void) const;

// Conditionally acquire a write lock

// (i.e., won’t block).

int try\_acquire\_write (void) const;

// Unlock a readers/writer lock. int release (void) const;

private:

rwlock\_t lock\_;

};

注意 POSIX Pthreads 和 Win32 线程并不提供 rwlock\_t 类型。为确保代码的可移植性，ACE 提供了一种 基于现有低级同步机制（比如互斥体和条件变量）的 RW\_Mutex 实现。另外，ACE 还提供 RW\_Thread\_ Mutex 和 RW\_Process\_Mutex 实现。

**4.5.1.4 Recursive\_Thread\_Mutex** 类

Recursive\_Thread\_Mutex 扩展缺省的 Solaris 非递归锁定语义。它允许嵌套调用 acquire 方法，只要拥 有该锁的线程也是重获取它的线程。它与 Thread\_Mutex 类一起工作。

缺省地，Solaris 提供非递归互斥体。这些语义在某些环境中太过受限。因此，ACE 通过 Recursive\_ Thread\_Mutex 类为递归锁提供支持。递归锁对于回调驱动的 C++框架[28, 20]特别有用，在其中框架的事 件循环执行对用户定义的代码的回调。因为用户定义的代码有可能随后经由一个方法入口重入框架代码， 递归锁对于防止在回调过程中，在框架所持有的锁上发生死锁十分有用。

下面的 C++类为 Solaris 2.x 同步机制实现递归锁语义（注意 POSIX Pthreads 和 Win32 在它们的本地 线程库中提供递归锁）：

class Recursive\_Thread\_Mutex

{

public:

// Initialize a recursive mutex. Recursive\_Thread\_Mutex (const char \*name = 0

void \*arg = 0);

// Implicitly release a recursive mutex.

˜Recursive\_Thread\_Mutex (void);

// Explicitly release a recursive mutex. int remove (void);

// Acquire a recursive mutex (will increment

// the nesting level and not deadmutex if

// owner of the mutex calls this method more

// than once).

int acquire (void) const;

// Conditionally acquire a recursive mutex

// (i.e., won’t block).

int try\_acquire (void) const;

// Releases a recursive mutex (will not

// release mutex until nesting level == 0). int release (void) const;

thread\_t get\_thread\_id (void);

// Return the id of the thread that currently

// owns the mutex.

int get\_nesting\_level (void);

// Return the nesting level of the recursion.

// When a thread has acquired the mutex for the

// first time, the nesting level == 1. The nesting

// level is incremented every time the thread

// acquires the mutex recursively.

private:

void set\_nesting\_level (int d);

void set\_thread\_id (thread\_t t);

Thread\_Mutex nesting\_mutex\_;

// Guards the state of the nesting level

// and thread id.

Condition<Thread\_Mutex> lock\_available\_;

// This is the condition variable that actually

// suspends other waiting threads until the

// mutex is available.

int nesting\_level\_;

// Current nesting level of the recursion.

thread\_t owner\_id\_;

// Current owner of the lock.

};

下面的代码演示 Recursive\_Thread\_Mutex 类中的方法的实现：

Recursive\_Thread\_Mutex::Recursive\_Thread\_Mutex

(const char \*name, void \*arg)

: nesting\_level\_ (0), owner\_id\_ (0), nesting\_mutex (name, arg),

lock\_available\_ (nesting\_mutex\_, name, arg)

{

}

// Acquire a recursive lock (will increment

// the nesting level and not deadlock if

// owner of lock calls method more than once).

int Recursive\_Thread\_Mutex::acquire (void) const

{

thread\_t t\_id = Thread::self ();

Thread\_Mutex\_Guard mon (nesting\_mutex\_);

// If there’s no contention, just

// grab the lock immediately. if (nesting\_level\_ == 0)

{

set\_thread\_id (t\_id);

nesting\_level\_ = 1;

}

// If we already own the lock,

// then increment the nesting level

// and proceed.

else if (t\_id == owner\_id\_)

nesting\_level\_++;

else

{

}

// Wait until the nesting level has dropped to

// zero, at which point we can acquire the lock. while (nesting\_level\_ > 0)

lock\_available\_.wait ();

set\_thread\_id (t\_id);

nesting\_level\_ = 1;

return 0;

}

// Releases a recursive lock.

int Recursive\_Thread\_Mutex::release (void) const

{

thread\_t t\_id = Thread::self ();

// Automatically acquire mutex. Thread\_Mutex\_Guard mon (nesting\_mutex\_);

nesting\_level\_--;

if (nesting\_level\_ == 0)

// Inform waiters that the lock is free. lock\_available\_.signal ();

return 0;

}

下面是基于 4.4 介绍的 Atomic\_Op COUNTER 的变种的一个 Recursive\_Thread\_Mutex 的例子。在例

中，Atomic\_Op 在单线程中被多次递归的函数调用：

// Counter is a recursive lock.

typedef Atomic\_Op<Recursive\_Thread\_Mutex> COUNTER;

// Keep track of the recursion depth. static COUNTER recursion\_depth;

int factorial (int n)

{

if (n <= 1)

{

}

else

{

}

}

cout << "recursion depth = "

<< recursion\_depth << endl;

return n;

// First call acquires lock, subsequent

// calls increment nesting level. recursion\_depth++;

return factorial (n - 1) \* n;

当 recursion\_depth 计数器增长时，Recursive\_Thread\_Mutex 的使用防止了死锁的发生。尽管这演示了递归

锁的行为，它并非是一个非常令人信服的例子。在多线程中执行 factorial 的程序可能会产生不可预知的结 果，因为 recursion\_depth 是一个全局变量，可能会被多个线程控制连续地修改！在这种情况下，一种更 为适当的（并且更低廉的）锁定策略将使用 4.5.6.4 描述的线程专有存储模式[29]。

**4.5.1.5 Null\_Mutex** 类

Null\_Mutex 类提供一种零开销的通用锁定接口的实现，该接口与其他用于线程和同步的 C++包装共 享。Null\_Mutex 的接口和极其简单的实现如下所示：

class Null\_Mutex

{

public:

Null\_Mutex (void) {}

˜Null\_Mutex (void) {}

int remove (void) { return 0; }

int acquire (void) const { return 0; }

int try\_acquire (void) const { return 0; }

int release (void) const { return 0; }

};

如上面的代码所示，Null\_Mutex 类将 acquire 和 release 方法实现为“空操作”内联函数，编译优化器将

把它们完全清除掉。4.6 演示 Null\_Mutex 的使用。

**4.5.1.6** 令牌（**Token**）类

该类提供了一种比 Mutex 更为通用的同步机制。例如，它实现了“递归互斥体”语义，拥有该令牌 的线程可以重新获取它，而不会死锁。此外，当其他线程释放 Token 时，阻塞并等待该 Token 的线程严 格地按照 FIFO（先进先出）的顺序被服务。相反，Mutex 并不严格地实施一种获取顺序。

Token 类的接口如下所示：

class Token

{

public:

// Initialization and termination.

Token (const char \*name = 0, void \* = 0);

˜Token (void);

// Acquire the token, sleeping until it is

// obtained or until <timeout> expires.

// If some other thread currently holds

// the token then <sleep\_hook> is called

// before our thread goes to sleep.

int acquire (void (\*sleep\_hook)(void \*), void \*arg = 0,

Time\_Value \*timeout = 0);

// This behaves just like the previous

// <acquire> method, except that it

// invokes the virtual function called

// <sleep\_hook> that can be overridden

// by a subclass of Token.

int acquire (Time\_Value \*timeout = 0);

// This should be overridden by a subclass

// to define the appropriate behavior before

// <acquire> goes to sleep. By default,

// this is a no-op...

virtual void sleep\_hook (void);

// An optimized method that efficiently

// reacquires the token if no other threads

// are waiting. This is useful for if you

// don’t want to degrad the quality of

// service if there are other threads

// waiting to get the token.

int renew (int requeue\_position = 0, Time\_Value \*timeout = 0);

// Become interface-compliant with other

// lock mechanisms (implements a

// non-blocking <acquire>). int tryacquire (void);

// Shuts down the Token instance. int remove (void);

// Relinquish the token. If there are any

// waiters then the next one in line gets it. int release (void);

// Return the number of threads that are

// currently waiting to get the token. int waiters (void);

// Return the id of the current thread that

// owns the token.

thread\_t current\_owner (void);

};

**4.5.2 ACE** 守卫（**Guard**）类属

**4.5.2.1 Guard** 类

与 C 一级的互斥体 API 相比较，4.5.1.1 描述的 Mutex 包装为同步多线程控制提供了一种优雅的接口。 但是，Mutex 潜在地容易出错，因为程序员有可能忘记调用 release 方法（如 4.3.7 所示）。这可能由于程 序员的疏忽或是 C++异常的发生而发生。

因此，为改善应用的健壮性，ACE 同步机制有效地利用 C++类构造器和析构器的语义来确保 Mutex 锁被自动获取和释放。ACE 提供了一个称为 Guard、Write\_Guard 和 Read\_Guard 的类族，确保在进入和 退出 C++代码块时分别自动获取和释放锁。

Guard 类是最基本的守卫机制，定义如下：

template <class LOCK>

class Guard

{

public:

// Implicitly and automatically acquire (or try

// to acquire) the lock.

Guard (LOCK &l, int block = 1): lock\_ (&l)

{

result\_ = block ? acquire () : tryacquire ();

}

// Implicitly release the lock.

˜Guard (void)

{

if (result\_ != -1)

lock\_.release ();

}

// 1 if locked, 0 if can’t acquire lock

// (errno will contain the reason for this). int locked (void)

{

return result\_ != -1;

}

// Explicitly release the lock. int remove (void)

{

return lock\_->remove ();

}

// Explicitly acquire the lock. int acquire (void)

{

return lock\_->acquire ();

}

// Conditionally acquire the lock (i.e., won’t block). int tryacquire (void)

{

return lock\_->tryacquire ();

}

// Explicitly release the lock. int release (void)

{

return lock\_->release ();

}

private:

// Pointer to the LOCK we’re guarding. LOCK \*lock\_;

// Tracks if acquired the lock or failed. int result\_;

};

Guard 类的对象定义一“块”代码，在其上锁被自动获取，并在退出块时自动释放。注意这种机制也能为

Mutex、RW\_Mutex 和 Semaphore 同步封装工作。这演示了使用 C++包装的另一个好处：通过改编不必要 地互不兼容的接口（比如 Solaris 2.x 信号量和互斥体），这些封装促进了接口的一致性。

缺省地，上面所示的 Guard 类构造器将会阻塞，直到锁被获取。会有这样的情况，程序必须使用非 阻塞的 acquire 调用（例如，防止死锁）。因此，可以传给 ACE Guard 的构造器第二个参数，指示它使 用锁的 try\_acquire 方法，而不是 acquire。随后调用者可以使用 Guard 的 locked 方法来原子地测试实际上 锁是否已被获取。

Read\_Guard 和 Write\_Guard 类有着与 Guard 类相同的接口。但是，它们的 acquire 方法分别对锁进行 读和写。

**4.5.2.2 Thread\_Control** 类

Thread\_Control 类用于与 Thread\_Manager 类相结合，以使在线程的发起函数中的优雅的线程终止和 清扫活动自动化。例如，Thread\_Control 的构造器存储状态信息。当最初用于调用线程的函数终止时，该 信息自动地从相关联的 Thread\_Manager 中移除该线程。该技术能正确地工作，不管（1）函数经由哪一 条路径被执行，以及（2）是否有异常被扔出。就这一点来说，Thread\_Control 类的行为与 4.5.2.1 介绍的 Guard 类相类似。

Thread\_Control 类的接口给出如下：

class Thread\_Control

{

public:

// Initialize the thread control object.

// If INSERT != 0, then register the thread

// with the Thread\_Manager.

Thread\_Control (Thread\_Manager \*, int add = 0);

// Implicitly kill the thread on exit and

// remove it from its associated Thread\_Manager.

˜Thread\_Control (void);

// Explicitly kill the thread on exit and

// remove it from its associated Thread\_Manager. void \*exit (void \*status);

// Set the exit status (and return status). void \*set\_status (void \*status);

// Get the current exit status. void \*get\_status (void);

};

**4.5.3 ACE** 条件（**Condition**）类属

**4.5.3.1 Condition** 类

Condition 类用于在涉及共享数据的条件表达式的状态发生变化时进行阻塞。它封装 Solaris 线程和 P OSIX pthreads cond\_t 同步变量。Condition 类接口给出如下：

template <class MUTEX>

class Condition

{

public:

// Initialize the condition variable. Condition (const MUTEX &m,

int type = USYNC\_THREAD,

void \*arg = 0);

// Implicitly destroy the condition variable.

˜Condition (void);

// Explicitly destroy the condition variable. int remove (void);

// Block on condition, or until absolute

// time-of-day has elapsed. If abstime

// == 0 use blocking wait().

int wait (Time\_Value \*abstime = 0) const;

// Signal one waiting thread. int signal (void) const;

// Signal \*all\* waiting threads. int broadcast (void) const;

private:

cond\_t cond\_;

// Reference to mutex lock. const MUTEX &mutex\_;

};

注意 Win32 不提供条件变量抽象。因此，ACE 线程库使用其他像信号量和互斥体这样的 ACE 组件 来实现条件变量。

**4.5.3.2 Null\_Condition** 类

Null\_Condition 类是上面描述的 Condition 接口的一种零开销实现。它的方法全都实现为空操作。这 对于根本不需要互斥的情况很有用（例如，一个特定的程序或服务将总是在单线程控制中运行，并且/或 者不与其他线程争夺对共享资源的访问）。使用 Null \*类的原因是允许应用参数化它们所需的同步类型， 而不需要改动应用代码。Null\_Condition 类接口给出如下：

template <class MUTEX>

class Null\_Condition

{

public:

Null\_Condition (const MUTEX &m, int type = 0,

void \*arg = 0) {}

˜Null\_Condition (void) {}

int remove (void) { return 0; }

int wait (Time\_Value \*abstime = 0) const

{

errno = ETIME; return -1;

}

int signal (void) const

{

errno = ETIME; return -1;

}

int broadcast (void) const

{

errno = ETIME; return -1;

}

};

Null\_Condition 类在“精神”上与 4.5.1.5 描述的 Null\_Mutex 类是一样的。

**4.5.4 ACE** 线程管理器类属

**4.5.4.1 Thread\_Manager** 类

Thread\_Manager 类含有一组机制来对进行协作、以实现集体动作的成组线程进行管理。例如，Threa d\_Manager 类提供的机制（比如 suspend\_all 和 resume\_all）允许任意数量的参与线程被原子地挂起或恢复。 Thread\_Manager 类还将应用与不同风格的多线程机制（比如 Solaris、POSIX 和 Win32）间的许多不兼容 特性屏蔽开来。

Thread\_Manager 类的接口演示如下：

class Thread\_Manager

{

public:

// Initialize the thread manager. Thread\_Manager (int size);

// Implicitly destroy thread manager.

˜Thread\_Manager (void);

// Initialize the manager with room

// for SIZE threads.

int open (int size = DEFAULT\_SIZE);

// Release all resources. int close (void);

// Create a new thread. int spawn (THR\_FUNC,

long, thread\_t \* = 0, void \*stack = 0,

size\_t stack\_size = 0);

// Create N new threads.

int spawn\_n (int n, THR\_FUNC,

void \*args, long flags);

// Clean up when a thread exits. void \*exit (void \*status);

// Blocks until there are no

// more threads running. void wait (void);

// Resume all stopped threads. int resume\_all (void);

// Suspend all threads. int suspend\_all (void);

// Send signum to all stopped threads. int kill\_all (int signal);

private:

// ...

};

**4.5.4.2 Thread\_Spawn** 类

Thread\_Spawn 类提供的一种标准机制管理线程的创建，以并发地处理来自客户的请求。该类作为“线 程工厂”，接受来自客户的连接，并“按需”派生线程，以运行由用户提供的服务处理器（SVC\_HANDL ER）指定的服务。

Thread\_Spawn 类的接口给出如下：

template <class SVC\_HANDLER, class PEER\_ACCEPTOR, class PEER\_ADDR>

class Thread\_Spawn

: public Acceptor <SVC\_HANDLER, PEER\_ACCEPTOR,

PEER\_ADDR>

{

public:

// = Initialization methods.

Thread\_Spawn (Thread\_Manager \*tm, Reactor \*);

virtual int open (const PEER\_ADDR &sia, Reactor \*);

protected:

virtual int handle\_input (int fd);

// Template method that accepts connection

// and spawns a thread.

virtual int handle\_close (int fd, Reactor\_Mask);

// Called when this factory is closed down.

virtual SVC\_HANDLER \*make\_svc\_handler (void);

// Factory method that creates an appropriate

// SVC\_HANDLER \*.

virtual int thr\_flags (void);

// Returns the flags used to spawn a thread.

};

注意此类是怎样从 ACE Acceptor 类继承的，后者是一种通用工厂，用于被动地连接客户和创建服务处理

器[30]。

**4.5.5 ACE** 主动对象（**Active Object**）类属

**4.5.5.1 Task**（任务）类

Task 类是 ACE 中用于创建用户定义的主动对象[16]和被动对象（它们处理应用消息）的中心机制。

ACE Task 可进行以下活动：

z 可被动态链接；

z 可用作 I/O 操作的端点；

z 可与多个线程控制相关联（也就是，成为所谓的“主动对象”）；

z 可在队列中存储消息，用于后续处理；

z 可执行用户定义的服务。

Task 抽象类定义的接口被派生类继承和实现，以提供应用特有的功能。它之所以是一个抽象类，是 因为它的接口定义了一些在下面描述的纯虚方法（open、close、put 和 svc）。通过使 Stream 类属提供的 与应用无关的组件与继承并使用这些组件的应用特有的子类去耦合，将任务定义为抽象类增强了复用。 同样地，纯虚函数的使用允许 C++编译器确保任务的子类遵从它提供以下功能的义务：

z 初始化和终止方法：派生自 Task 的子类必须实现 open 和 close 方法，它们执行应用特有的 Task 初始 化和终止活动。这些活动通常分配和释放资源，比如连接控制块、I/O 句柄和同步锁。

Task 可与 Module（模块）一起或是分开定义和使用。当与模块一起使用时，任务被成对地存储： 一个 Task 子类处理读端的自下游发到该模块层的消息，另一个处理写端的自上游发到该模块层的消 息。

当一个模块被插入流，或从流中被移除时，它的读端和写端的任务子类的 open 和 close 方法分别 自动地被 ASX 框架调用。

z 应用特有的处理方法：除了 open 和 close 方法，Task 的子类还必须定义 put 和 svc 方法。这些方法在 消息上执行应用特有的处理功能。例如，当消息到达流的头或尾时，作为调用流中的每个任务的 put 和/或 svc 方法的结果，它们被“护送”通过一系列互连的任务。

put 方法在流中某层的一个 Task 传递消息给另一层中相邻的 Task 时被调用。put 方法相对于它的调用 者同步地运行，也就是，它从起先调用它的 put 方法的 Task 那里借用线程控制。该线程控制通常从 应用进程“自下而上”、从处理 I/O 设备中断[31]的线程池“自上而下”发起，或是由事件分派机制（比 如在面向连接的传输协议模块中、用于触发重发的定时器驱动的呼出队列）在流内部发起。

如果一个 ACE Task 作为被动对象执行（也就是，它总是从调用者那里借用线程控制），Task::pu

t 方法就是进入该 Task 的入口，并用作 Task 在其中执行它的工作的上下文。相反，如果一个 ACE T ask 作为主动对象执行，Task::svc 方法就被用于相对于其他 Task 异步地执行应用特有的处理。不像 p ut，svc 方法并不直接从相邻的 Task 那里调用。相反，它被与它从属的 Task 相关联的一个分离的线程 调用。该线程为 Task 的 svc 方法提供执行上下文和线程控制。svc 方法运行一个事件循环，持续地等 待消息到达 Task 的 Message Queue（消息队列，见下一条目）。

在 put 或 svc 方法的实现中，消息可以通过任务的 put\_next 实用方法传递给流中相邻的 Task。pu t\_next 调用驻留在相邻层中的下一个 Task 的 put 方法。这个对 put 的调用可以从调用者那里借用线程 控制，并立即处理消息（也就是，图 4-4（1）演示的同步处理方法）。相反地，put 方法可以将消息入 队，并将处理推迟给它的在分离的线程控制中执行的 svc 方法（也就是，图 4-4（2）演示的异步处理 方法）。如在[1]中所讨论的，选择特定的处理方法对性能和编程的容易程度有着显著的影响。

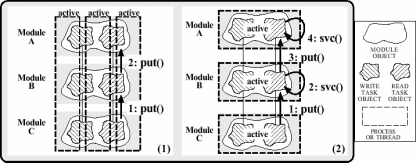


图 4-4 调用 put 和 svc 方法的可选方案

z 消息排队机制：除了 open、close、put 和 svc 纯虚方法接口，每个任务还含有一个 Message Queue（消 息队列）。Message Queue 是 ACE 中的标准组件，用于在 Task 间传递消息。而且，当 Task 作为主动 对象执行时，它的 Message Queue 用于缓存一系列数据消息和控制消息，以在 svc 方法中作后续处理。 在消息到达时，svc 方法使消息出队，并执行 Task 子类的应用特有的处理工作。

Message Queue 中可以出现两种类型的消息：简单的和复合的。简单消息含有单个 Message Blo ck，而复合消息含有多个链接在一起的 Message Block。复合消息通常由一个控制块和紧随的一或多 个数据块组成。控制块含有“簿记”信息（比如目的地址和长度域），而数据块含有消息的实际内容。 通过传递消息指针而不是拷贝数据，在 Task 间传递 Message Block 的开销被降到了最低。

Message Queue 含有一对高低水位标变量，用于在流中的相邻 Module 间实现层到层的流控制。 高水位标指示消息队列在进行流控制之前所愿意缓存的消息字节数。低水位标指示在先前已进行流控 制的 Message Queue 不再被视为满的“水位”。

Task 类的接口提供如下：

template <class SYNCH>

class Task : public Service\_Object

{

public:

// Initialization/termination methods. Task (Thread\_Manager \*thr\_mgr = 0, Message\_Queue<SYNCH> \*mp = 0);

virtual int open (void \*flags = 0) = 0;

virtual int close (u\_long = 0) = 0;

// Transfer msg into the queue to handle

// immediate processing.

virtual int put (Message\_Block \*, Time\_Value \*tv = 0) = 0;

// Run by a daemon thread to handle

// deferred processing.

virtual int svc (void) = 0;

protected:

// Turn the task into an active object.. int activate (long flags);

// Routine that runs the service routine

// as a daemon thread.

static void \*svc\_run (Task<SYNCH> \*);

// Tests whether a message can be enqueue

// without blocking.

int can\_put (Message\_Block \*);

// Insert message into the message list.

int putq (Message\_Block \*, Time\_Value \* = 0);

// Extract the first message from the list.

int getq (Message\_Block \*&, Time\_Value \* = 0);

// Return a message to the queue.

int ungetq (Message\_Block \*, Time\_Value \* = 0);

// Transfer message to the adjacent Task

// in a Stream.

int put\_next (Message\_Block \*, Time\_Value \* = 0);

// Turn the message back around.

int reply (Message\_Block \*, Time\_Value \* = 0);

// Task utility routines to identify names. const char \*name (void) const;

Task<SYNCH> \*sibling (void); Module<SYNCH> \*module (void) const;

// Check if queue is a reader. int is\_reader (void);

// Check if queue is a writer. int is\_writer (void);

// Special routines corresponding to

// certain message types. int flush (u\_long flag);

// Manipulate watermarks.

void water\_marks (IO\_Cntl\_Msg::IO\_Cntl\_Cmds, size\_t);

};

**4.5.6** 杂项 **ACE** 并发类

**4.5.6.1 Thread** 类工具

Thread 类工具在 C++包装中封装 Solaris、POSIX 和 Win32 族的线程创建、终止和管理例程族。该类 提供了一种映射到 Solaris 线程、POSIX Pthreads 和 Win32 线程的通用接口。

Thread 类的接口提供如下：

typedef void \*(\*THR\_FUNC)(void \*);

class Thread

{

public:

// Spawn N new threads, which execute

// "func" with argument "arg".

static int spawn\_n (size\_t n, THR\_FUNC func, void \*arg, long flags, void \*stack = 0,

size\_t stack\_size = 0);

// Spawn a new thread, which executes

// "func" with argument "arg". static int spawn (THR\_FUNC,

void \*arg, long, thread\_t \* = 0, void \*stack = 0,

size\_t stack\_size = 0, hthread\_t \*t\_handle = 0);

// Wait for one or more threads to exit.

static int join (hthread\_t, hthread\_t \*, void \*\*);

// Suspend the execution of a thread. static int suspend (hthread\_t);

// Continue the execution of a

// previously suspended thread. static int resume (hthread\_t);

// Send signal signum to the thread. static int kill (thread\_t, int signum);

// Return the unique ID of the thread. static thread\_t self (void);

// Yield the thread to another. static void yield (void);

// Exit current thread, returning "status". static void exit (void \*status);

// Set LWP concurrency level of the process. static int setconcurrency (int new\_level);

// Get LWP concurrency level of the process. static int getconcurrency (void);

static int sigsetmask (int how,

const sigset\_t \*set, sigset\_t \*oset = 0);

// Change and/or examine calling thread’s

// signal mask.

static int keycreate (thread\_key\_t \*keyp,

void (\*)(void \*value));

// Allocates a <keyp> that is used to

// identify data that is specific to each

// thread in the process. The key is global

// to all threads in the process.

static int setspecific (thread\_key\_t key, void \*value);

// Bind value to the thread-specific data

// key, <key>, for the calling thread.

static int getspecific (thread\_key\_t key, void \*\*valuep);

// Stores the current value bound to <key>

// for the calling thread into the location

// pointed to by <valuep>.

};

**4.5.6.2 Atomic\_Op** 类

Atomic\_Op 类将同步透明地参数化进基本的算术运算中。

template <class LOCK, class TYPE>

class Atomic\_Op

{

public:

// Initialize count\_ to 0. Atomic\_Op (void);

// Initialize count\_ to c. Atomic\_Op (TYPE c);

// Atomically increment count\_. TYPE operator++ (void);

// Atomically increment count\_ by inc. TYPE operator+= (const TYPE inc);

// Atomically decrement count\_. TYPE operator-- (void);

// Atomically decrement count\_ by dec. TYPE operator-= (const TYPE dec);

// Atomically compare count\_ with rhs. TYPE operator== (const TYPE rhs);

// Atomically check if count\_ >= rhs. TYPE operator>= (const TYPE rhs);

// Atomically check if count\_ > rhs. TYPE operator> (const TYPE rhs);

// Atomically check if count\_ <= rhs. TYPE operator<= (const TYPE rhs);

// Atomically check if count\_ < rhs. TYPE operator< (const TYPE rhs);

// Atomically assign rhs to count\_. void operator= (const TYPE rhs);

// Atomically return count\_. operator TYPE ();

private:

LOCK lock\_; TYPE count\_;

};

**4.5.6.3 Barrier**（栅栏）类

Barrier 类实现“栅栏同步”，它对于许多类型的并行科学应用特别有用。该类允许 count 数量的线程 同步它们的完成（所谓的“栅栏同步”）。它的实现使用了“子栅栏生成编号”方案来避免过度的开销， 并确保所有线程都正确地离开栅栏。

class Barrier

{

public:

// Initialize the barrier to

// synchronize count threads. Barrier (u\_int count,

int type = USYNC\_THREAD,

void \*arg = 0);

// Block the caller until all count threads

// have called wait() and then allow all

// the caller threads to continue in parallel. int wait (void);

};

**4.5.6.4 TSS** 类

TSS 类允许“物理上”线程专有的（也就是，线程私有的）对象被“逻辑地”当作程序的全局对象 进行访问。该类所基于的底层“线程专有存储”模式在[29]中描述。

下面是 ACE TSS 类的公共接口：

template <class TYPE>

class TSS

{

public:

// If caller passes a non-NULL ts\_obj \*

// this is used to initialize the

// thread-specific value. Thus, calls

// to operator->() will return this value. TSS (TYPE \*ts\_obj = 0);

// Get the thread-specific object for the key

// associated with this object. Returns 0

// if the data has never been initialized,

// otherwise returns a pointer to the data. TYPE \*ts\_object (void);

// Use a "smart pointer" to obtain the

// thread-specific object associated with

// the key.

TYPE \*operator-> ();

};

**4.6** 使用 **ACE OO** 线程封装库

这一部分介绍若干例子，演示 ACE 线程库中的关键特性的使用。请查阅 4.5 描述的接口，以确定 A CE 并发组件的行为。

**4.6.1** 用于消息多路分离的映射管理器

选择一种有着适当语义的互斥机制常常取决于类被使用的“上下文”。下面的例子演示通用 ACE 工 具包[32]中的“映射管理器”的接口和实现。此组件通常用于在网络服务器中将外部的标识符（比如端口 号或连接 id）映射到内部的标识符（比如指向卫星输出链路开始流控制后用于存储消息的队列的指针）。 Map\_Manager 的部分接口和实现显示如下：

template <class EXT\_ID, class INT\_ID, class LOCK>

class Map\_Manager

{

public:

// Associate EXT\_ID with the INT\_ID.

int bind (EXT\_ID ext\_id, const INT\_ID \*int\_id)

{

Write\_Guard<LOCK> monitor (lock\_);

// ...

}

// Break any association of EXT\_ID. int unbind (EXT\_ID ext\_id)

{

Write\_Guard<LOCK> monitor (lock\_);

// ...

}

// Locate INT\_ID associated with EXT\_ID and

// pass out parameter via INT\_ID.

// If found return 0, else -1.

int find (EXT\_ID ext\_id, INT\_ID &int\_id)

{

Read\_Guard<LOCK> monitor (lock\_);

if (locate\_entry (ext\_id, int\_id)

// ext\_id is successfully located. return 0;

else

}

return -1;

private:

LOCK lock\_;

// ...

};

该方法的一个好处是无论哪条执行路径退出方法，lock\_都会被释放。例如，if/else 语句的任何一条分支 从 find 方法中返回，lock\_都会被正确释放。此外，如果在 locateEntry 助手方法的处理过程中有异常发生， “作为资源获取/释放的构造器/析构器”习语也会正确地释放 lock\_。这是有用的，因为 C++异常处理机 制被设计为在从异常被扔出的块中退出时、调用所有必需的析构器。注意我们使用了获取和释放 lock\_的 显式调用来编写 find 的定义，也就是：

int find (EXT\_ID ext\_id, INT\_ID &int\_id)

{

lock\_.acquire ();

if (locateEntry (ext\_id, int\_id)

{

}

else

{

}

}

// ext\_id is successfully located. lock\_.release ();

return 0;

lock\_.release ();

return -1;

find 方法的逻辑更不自然，空间的使用效率也低下。此外，也不能保证有异常从 locateEntry 方法中扔出

时，lock\_会被释放。

Map\_Manager 模板类用于实例化的 LOCK 类型取决于锁被使用时程序代码中的并行的特定结构。例 如，在某些情况下，这样声明是有用的：

typedef Map\_Manager <Addr, TCB, Mutex>MAP\_MANAGER;

并使得所有对 find、bind 和 unbind 的调用自动地被序列化。在其他的一些情况下，可以通过使用 Null\_M

utex 类来关闭同步，而不用改变已有的库代码：

typedef Map\_Manager <Addr, TCB, Null\_Mutex>MAP\_MANAGER;

还有一种情况，对 find 的调用可能远比对 bind 或 unbind 的调用频繁。在这样的情况下，使用 RW\_Mute x 读者/作者锁可能是有意义的：

typedef Map\_Manager <Addr, TCB, RW\_Mutex>MAP\_MANAGER;

通过使用 C++包装和模板，我们可以创建一个高度可移植、平台无关的互斥类接口，而无需对我们 所用的不同同步机制强加任何语法约束。通过使用模板来参数化锁定类型，无需改动应用代码、或只需 很少一点改动就可以适应新的同步语义。但是，像往常一样，选择适当的同步机制需要通过剖面和经验 的衡量进行指导。

**4.6.2** 线程安全的消息排队机制

该例子演示 ACE Condition 包装（4.5.1.1）和 ACE Mutex 包装（4.5.3.1）的使用。代码从 Message\_ Queue 类中摘录，该类包含在 4.5.5.1 描述的 Task 类中。Message\_Queue 可被同步策略类型参数化，以获 取所期望的并发控制级。缺省地，并发控制级是“线程安全”，如 ACE Synch.h 文件中的 MT\_Synch 类 所定义的：

class MT\_Synch

{

public:

typedef Condition<Mutex> CONDITION;

typedef Mutex MUTEX;

};、

如果 MT\_Synch 被用于实例化 Message\_Queue，所有的公共方法都将是线程安全的，但同时也带来 相应的开销。相反，如果 Null\_Synch 类用于实例化 Message\_Queue，所有公共方法都不是线程安全的， 同时也就没有额外的开销。Null\_Synch 也在 Synch.h 中定义，如下所示：

class Null\_Synch

{

public:

typedef Null\_Condition<Null\_Mutex> CONDITION;

typedef Null\_Mutex MUTEX;

};

在 4.3.5 开始处的 run\_svc 函数中有 Message\_Queue 的使用实例。Message\_Queue 在系统 V STREAM [33]和 BSD UNIX[34]提供的消息排队和缓冲区管理机制的基础上构建。

ACE Message\_Queue 由一或多个通过 prev\_和 next\_指针链接在一起的 Message\_Block 组成。这样的 结构可以高效地操作任意大的消息，而不会导致巨大的内存拷贝开销。

// The contents of a message are represented

// internally by a Message\_Block. class Message\_Block

{

public:

Message\_Block (size\_t size,

Message\_Type type = MB\_DATA, Message\_Block \*cont = 0,

char \*data = 0);

// ...

};

Message\_Queue 是一种线程安全的消息排队机制。注意 C++“traits”习语的使用将 Condition 和 Mut

ex 类型合并进了单个模板参数。

template <class SYNCH = MT\_Synch>

class Message\_Queue

{

public:

// Default high and low water marks. enum

{

// 0 is the low water mark. DEFAULT\_LWM = 0,

// 1 K is the high water mark. DEFAULT\_HWM = 4096,

// Message queue was active

// before activate() or deactivate(). WAS\_ACTIVE = 1,

// Message queue was inactive

// before activate() or deactivate(). WAS\_INACTIVE = 2

};

// Initialize a Message\_Queue. Message\_Queue (size\_t hwm = DEFAULT\_HWM,

size\_t lwm = DEFAULT\_LWM);

// Destroy a Message\_Queue.

˜Message\_Queue (void);

/\* Checks if queue is full/empty. \*/

int is\_full (void) const;

int is\_empty (void) const;

// Enqueue and dequeue a Message\_Block \*. int enqueue\_tail (Message\_Block \*new\_item,

Time\_Value \*tv = 0);

int enqueue\_head (Message\_Block \*new\_item, Time\_Value \*tv = 0);

int dequeue\_head (Message\_Block \*&first\_item, Time\_Value \*tv = 0);

// Deactivate the queue and wakeup all threads

// waiting on the queue so they can continue. int deactivate (void);

// Reactivate the queue so that threads can

// enqueue and dequeue messages again. int activate (void);

private:

// Routines that actually do the enqueueing

// and dequeueing (assumes locks are held). int enqueue\_tail\_i (Message\_Block \*);

int enqueue\_head\_i (Message\_Block \*);

int enqueue\_head\_i (Message\_Block \*&);

// Check the boundary conditions. int is\_empty\_i (void) const;

int is\_full\_i (void) const;

// Implement activate() and deactivate()

// methods (assumes locks are held). int deactivate\_i (void);

int activate\_i (void);

// Pointer to head of Message\_Block list. Message\_Block \*head\_;

// Pointer to tail of Message\_Block list. Message\_Block \*tail\_;

// 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\_;

// Indicates that the queue is inactive. int deactivated\_;

// C++ wrapper synchronization primitives

// for controlling concurrent access. SYNCH::MUTEX lock\_;

SYNCH::CONDITION notempty\_; SYNCH::CONDITION notfull\_;

};

Message\_Queue 类的实现显示如下。Message\_Queue 的构造器创建一个空的消息列表，并初始化 Co

ndition 对象。注意 Mutex 的 lock\_被它的缺省构造器自动创建。

template <class SYNCH> Message\_Queue::Message\_Queue (size\_t hwm,

size\_t lwm)

: notfull\_ (lock\_), notempty\_ (lock\_)

{

// ...

}

下面的代码检查队列是否为“空”（也就是，没有消息在其中）或“满”（也就是，在其中的字节

的数目多于 high\_water\_mark）。注意这些方法（像下面的其他方法一样）怎样利用一种模式，公共方法 藉此来获取锁，而私有方法假定锁已被持有。

template <class SYNCH> int

Message\_Queue<SYNCH>::is\_empty\_i (void) const

{

return cur\_bytes\_ <= 0 && cur\_count\_ <= 0;

}

template <class SYNCH> int

Message\_Queue<SYNCH>::is\_full\_i (void) const

{

return cur\_bytes\_ > high\_water\_mark\_;

}

template <class SYNCH> int

Message\_Queue<SYNCH>::is\_empty (void) const

{

Guard<SYNCH::MUTEX> monitor (lock\_);

return is\_empty\_i ();

}

template <class SYNCH> int

Message\_Queue<SYNCH>::is\_full (void) const

{

Guard<SYNCH::MUTEX> monitor (lock\_);

return full ();

}

下面的方法用于启用和停用 Message\_Queue。deactivate 方法停用队列，并唤醒所有等待在该队列上

的线程，以使它们继续。没有消息被从队列中移除。其他任何被调用的方法都立即返回-1，且 errno == ESHUTDOWN，直到队列被再次激活。这些信息允许调用者检测状态的变化。

template <class SYNCH> int

Message\_Queue<SYNCH>::deactivate (void)

{

Guard<SYNCH::MUTEX> m (lock\_);

return deactivate\_i ();

}

template <class SYNCH> int

Message\_Queue<SYNCH>::deactivate\_i (void)

{

int current\_status = deactivated\_ ? WAS\_INACTIVE : WAS\_ACTIVE;

// Wake up all the waiters. notempty\_.broadcast (); notfull\_.broadcast ();

deactivated\_ = 1;

return current\_status;

}

activate 方法重新启用队列，以使线程能够再次让消息入队或出队。如果队列在调用前是不活动的，

该方法返回 WAS\_INACTIVE；如果队列在调用前是活动的，就返回 WAS\_ACTIVE。

template <class SYNCH> int

Message\_Queue<SYNCH>::activate (void)

{

Guard<SYNCH::MUTEX> m (lock\_);

return activate\_i ();

}

template <class SYNCH> int

Message\_Queue<SYNCH>::activate\_i (void)

{

int current\_status =

deactivated\_ ? WAS\_INACTIVE : WAS\_ACTIVE;

deactivated\_ = 0;

return current\_status;

}

enqueue\_head 方法在队列的前面插入一个新条目。像其他的入队和出队方法一样，如果 tv 参数为 N

ULL，调用者将阻塞直到可以继续执行。否则，调用者将阻塞，等待\*tv 所指定的数量的时间。但是当队 列被关闭、有信号发生，或 tv 中指定的时间过去了，阻塞的调用都会返回，且 errno == EWOULDBLO CK。

template <class SYNCH> int

Message\_Queue<SYNCH>::enqueue\_head

(Message\_Block \*new\_item, Time\_Value \*tv)

{

Guard<SYNCH::MUTEX> monitor (lock\_);

if (deactivated\_)

{

errno = ESHUTDOWN;

return -1;

}

// Wait while the queue is full. while (is\_full\_i ())

{

// Release the lock\_ and wait for

// timeout, signal, or space becoming

// available in the list.

if (notfull\_.wait (tv) == -1)

{

if (errno == ETIME)

errno = EWOULDBLOCK;

return -1;

}

if (deactivated\_)

{

errno = ESHUTDOWN;

return -1;

}

}

// Actually enqueue the message at the

// head of the list. enqueue\_head\_i (new\_item);

// Tell any blocked threads that the

// queue has a new item!

notempty\_.signal ();

return 0;

}

注意当队列先前为空时，这个方法是怎样只发送信号给 notempty\_条件对象的。通过减少不必要的信号发

送所导致的上下文切换的数量，这样的优化使性能得到了提高。其他两个入队和出队方法也执行类似的

优化。

enqueue\_tail 方法在队列的末尾插入新条目。它返回的是队列中条目的数量。

template <class SYNCH> int

Message\_Queue<SYNCH>::enqueue\_tail

(Message\_Block \*new\_item, Time\_Value \*tv)

{

Guard<SYNCH::MUTEX> monitor (lock\_);

if (deactivated\_)

{

errno = ESHUTDOWN;

return -1;

}

// Wait while the queue is full. while (is\_full\_i ())

{

// Release the lock\_ and wait for

// timeout, signal, or space becoming

// available in the list.

if (notfull\_.wait (tv) == -1)

{

if (errno == ETIME)

errno = EWOULDBLOCK;

return -1;

}

if (deactivated\_)

{

errno = ESHUTDOWN;

return -1;

}

}

// Actually enqueue the message at

// the end of the list. enqueue\_tail\_i (new\_item);

// Tell any blocked threads that

// the queue has a new item!

notempty\_.signal ();

return 0;

}

dequeue\_head 方法移除队列中最前面的条目，并将它传回给调用者。该方法返回队列中所余条目的

数目。

template <class SYNCH> int

Message\_Queue<SYNCH>::dequeue\_head

(Message\_Block \*&first\_item, Time\_Value \*tv)

{

Guard<SYNCH::MUTEX> monitor (lock);

// Wait while the queue is empty. while (is\_empty\_i ())

{

// Release the lock\_ and wait for

// timeout, signal, or a new message

// being placed in the list.

if (notempty\_.wait (tv) == -1)

{

if (errno == ETIME)

errno = EWOULDBLOCK;

return -1;

}

if (deactivated\_)

{

errno = ESHUTDOWN;

return -1;

}

}

// Actually dequeue the first message. dequeue\_head\_i (first\_item);

// Tell any blocked threads that

// the queue is no longer full. notfull\_.signal ();

return 0;

}

下面的代码演示一个使用 Message\_Queue 的经典的“有界缓冲区”的 ACE 实现。程序使用两个线程

来并发地将 stdin 拷贝到 stdout。图 4-5 演示并发运行的 ACE 组件间的关系。生产者线程从 stdin 流中读 取数据，创建消息，然后将消息放入 Message\_Queue。消费者线程使消息出队并将它写到 stdout。为节省 空间，省略了大多数错误检查。

#include "Message\_Queue.h"

#include "Thread\_Manager.h"

typedef Message\_Queue<MT\_Synch> MT\_Message\_Queue;

// Global thread manager. static Thread\_Manager thr\_mgr;

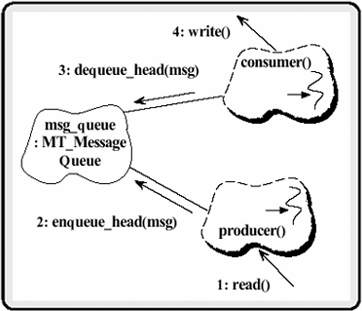


图 4-5 并发生产者和消费者

生产者从 stdin 中将数据读取到消息中，然后为消费者排队此消息。没有更多数据可读时，一个 NU LL 指针就被放到队列中。该指针用于通知消费者何时退出。

static void \*producer (MT\_Message\_Queue \*msg\_queue)

{

// Insert thread into thr\_mgr. Thread\_Control tc (&thr\_mgr); char buf[BUFSIZ];

for (int n; ; )

{

// Allocate a new message.

Message\_Block \*mb = new Message\_Block (BUFSIZ);

n = read (0, mb->rd\_ptr (), mb->size ());

if (n <= 0)

{

// Shutdown message to the other

// thread and exit. mb->length (0);

msg\_queue->enqueue\_tail (mb);

}

// Send the message to the other thread. else

{

mb->wr\_ptr (n);

msg\_queue->enqueue\_tail (mb);

}

}

// The destructor of Thread\_Control removes

// the exiting thread from the

// Thread\_Manager automatically. return 0;

}

consumer 从 Message\_Queue 中取出一个消息，将它写到 stderr 流中，并删除此消息。producer 发送一

个 NULL 指针，通知消费者停止读取并退出。

static void \*consumer(MT\_Message\_Queue \*msg\_queue)

{

Message\_Block \*mb = 0;

// Insert thread into thr\_mgr. Thread\_Control tc (&thr\_mgr); int result = 0;

// Keep looping, reading a message out

// of the queue, until we timeout or get a

// message with a length == 0, which signals

// us to quit. for (;;)

{

result = msg\_queue->dequeue\_head (mb);

if (result == -1)

return -1;

int length = mb->length ();

if (length > 0)

::write (1, mb->rd\_ptr (), length);

delete mb;

if (length == 0)

break;

}

// The destructor of Thread\_Control removes

// the exiting thread from the

// Thread\_Manager automatically. return 0;

}

main 函数派生两个线程来运行 producer 和 consumer 函数，以并行地拷贝 stdin 到 stdout。

int main (int argc, char \*argv[])

{

// Use the thread-safe instantiation

// of Message\_Queue. Message\_Queue msg\_queue;

thr\_mgr.spawn (THR\_FUNC (producer), (void \*) &msg\_queue, THR\_NEW\_LWP | THR\_DETACHED);

thr\_mgr.spawn (THR\_FUNC (consumer), (void \*) &msg\_queue,

THR\_NEW\_LWP | THR\_DETACHED);

// Wait for producer/consumer threads to exit. thr\_mgr.wait ();

return 0;

}

**4.6.3** 并发网络数据库服务器

下面的例子演示一个使用 ACE 线程管理组件开发的并发网络数据库服务器。客户请求触发服务器根 据它们唯一的数字 ID 查找“雇员”。如果找到匹配项，就将名字返回给客户。

每个给服务器的客户请求都并行地运行。这个例子演示了 Thread\_Manager 和 Thread\_Control 类的使 用。此外，它还演示了 socket 的 ACE C++包装类的使用[35]。

下面所示的代码意在简化例子，并没有表现一个高度健壮和高效的实现是如何开发的。例如，产品 实现会设置所派生的绑定线程数目的上限，以避免消耗大量的内核资源。此外，产品实现还可能会明确

地使用一个更为成熟的数据库方案。

#include "SOCK\_Acceptor.h"

#include "Thread\_Manager.h"

// Per-process thread manager. Thread\_Manager thr\_mgr;

// Function called when a new thread is created.

// This function is passed a connected client

// SOCK\_Stream, which it uses to receive a

// database lookup request from a client. static void \*lookup\_name (ACE\_HANDLE handle)

{

// Local thread control object. Thread\_Control tc (&thr\_mgr);

enum

{

};

// Maximum line we’ll read from a client. MAXLINE = 255,

// Maximum size of employee name. EMPNAMELEN = 512

// Simple read-only database. static struct

{

int emp\_id;

const char emp\_name[EMPNAMELEN];

} emp\_db[] =

{

123, "John Wayne Bobbit",

124, "Cindy Crawford",

125, "O. J. Simpson",

126, "Bill Clinton",

127, "Rush Limbaugh",

128, "Michael Jackson",

129, "George Burns",

0, ""

};

SOCK\_Stream new\_stream;

char recvline[MAXLINE];

char sendline[MAXLINE];

new\_stream.set\_handle (handle);

ssize\_t n = new\_stream.recv (recvline, MAXLINE);

int emp\_id = atoi (recvline);

int found = 0;

for (int index = 0; found == 0 && emp\_db[index].emp\_id; index++)

if (emp\_id == emp\_db[index].emp\_id)

{

found = 1;

n = sprintf (sendline, "%s", emp\_db[index].emp\_name);

}

if (found == 0)

n = sprintf (sendline, "%s", "ERROR");

new\_stream.send\_n (sendline, n + 1);

new\_stream.close ();

// The destructor of Thread\_Control removes the

// exiting thread from the Thread\_Manager

// automatically. return 0;

}

// Default port number.

static const int default\_port = 5000;

int main (int argc, char \*argv[])

{

// Port number of server.

u\_short port = argc > 1 ? atoi (argv[1]) : default\_port;

// Internet address of server. INET\_Addr addr (port);

// Passive-mode listener object. SOCK\_Acceptor server (addr);

SOCK\_Stream new\_stream;

// Wait for a connection from a client

// (this illustrates a concurrent server). for (;;)

{

// Accept a connection from a client. server.accept (new\_stream);

// Spawn off a thread-per client request. thr\_mgr.spawn (THR\_FUNC (lookup\_name),

(void \*) new\_stream.get\_handle (), THR\_BOUND | THR\_DETACHED);

}

// NOTREACHED

return 0;

}

**4.7** 结束语

本论文描述了 ACE 中提供的面向对象线程封装库。ACE 线程类库为开发者提供了若干好处：

z 通过使开发者能够在他们的并发应用中始终使用 C++和 OO，改善了编程风格的一致性。

z 减少了为使应用线程安全化所进行的强制性变动的数量。例如，库中的工具类（比如 Atomic\_Op、M

utex、RW\_Mutex、Semaphore 和 Condition）改善了底层 OS 特有的并发机制的可移植性和复用性。

z 消除或使发生微妙的同步错误的可能性降至了最低。若干 ACE 线程库类（比如 Guard 和 Thread\_Con trol）确保即使发生异常，资源也会被适当地分配和释放。

z 增强了抽象和模块性，而不用牺牲性能。使用 C++语言特性（比如内联函数和模板）确保了 ACE O O 线程库所提供的额外功能不会显著地降低效率。

ACE OO 线程封装库已在许多商业项目中被使用。这些产品包括 Ericsson EOS 族电信交换监控应用、 Bellcore ATM 交换管理软件、Motorola Iridium 全球个人通信系统的网络管理子系统和核心基础构造子系 统，以及在柯达和西门子的企业级电子医学成像系统。

参考文献

[1] D. C. Schmidt, “ACE: an Object-Oriented Framework for Developing Distributed Applications,” in *Proceedings of the*

*6th USENIX C++ Technical Conferenc*e, (Cambridge, Massachusetts), USENIX Association, April 1994.

[2] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalin-giah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Wi lliams, “Beyond Multiprocessing... Multithreading the SunOS Kernel,” in *Proceedings of the Summer USENIX Conferenc*e,

(San Antonio, Texas), June 1992.

[3] IEEE, *Threads Extension for Portable Operating Systems (Draft 10*), February 1996. [4] H. Custer, *Inside Windows N*T. Redmond, Washington: Microsoft Press, 1993.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Softwar*e. Reading, MA: Addison-Wesley, 1995.

[6] R. Johnson and B. Foote, “Designing Reusable Classes,” *Journal of Object-Oriented Programmin*g, vol. 1, pp. 22–35, June/July 1988.

[7] G. Booch, *Object Oriented Analysis and Design with Applications (2nd Edition*). Redwood City, California: Benjamin/

Cummings, 1993.

[8] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Desig*n. Englewoo d Cliffs, NJ: Prentice Hall, 1991.

[9] Bjarne Stroustrup, *The C++ Programming Language, 2nd Editio*n. Addison-Wesley, 1991.

[10] W. R. Stevens, *Advanced Programming in the UNIX Environmen*t. Reading, Massachusetts: Addison Wesley, 1992. [11] D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP Vol III: Client – Server Programming and Applicatio n*s.Englewood Cliffs, NJ: Prentice Hall, 1992.

[12] Sun Microsystems, *Open Network Computing: Transport Independent RP*C, June 1995.

[13] Object Management Group, *The Common Object Request Broker: Architecture and Specificatio*n, 1.2 ed., 1993. [14] W. R. Stevens, *UNIX Network Programming, First Editio*n. Englewood Cliffs, NJ: Prentice Hall, 1990.

[15] W. R. Stevens, *TCP/IP Illustrated, Volume* 1. Reading, Massachusetts: Addison Wesley, 1993.

[16] R. G. Lavender and D. C. Schmidt, “Active Object: an Object Behavioral Pattern for Concurrent Programming,” in *Proceedings of the 2nd Annual Conference on the Pattern Languages of Program*s, (Monticello, Illinois), pp. 1–7, Septem ber 1995.

[17] D. C. Schmidt and C. D. Cranor, “Half-Sync/Half-Async: an Architectural Pattern for Efficient and Well-structured Concurrent I/O,” in *Proceedings of the 2nd Annual Conference on the Pattern Languages of Program*s, (Monticello, Illino is), pp. 1–10, September 1995.

[18] A. D. Birrell, “An Introduction to Programming with Threads,” Tech. Rep. SRC-035, Digital Equipment Corporation, January 1989.

[19] Sun Microsystems, Inc., Mountain View, CA, *SunOS 5.3 Guide to Multi-Thread Programmin*g, Part number: 801-31

76-10 ed., May 1993.

[20] D. C. Schmidt, “Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dis patching,” in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, M A: Addison-Wesley, 1995.

[21] A. Garg, “Parallel STREAMS: a Multi-Process Implementation,” in *Proceedings of the Winter USENIX Conferenc*e, (Washington, D.C.), Jan. 1990.

[22] S. Saxena, J. K. Peacock, F. Yang, V. Verma, and M. Krishnan, “Pitfalls in Multithreading SVR4 STREAMS and other Weightless Processes,” in *Proceedings of the Winter USENIX Conferenc*e, (San Diego, CA), pp. 85–106, Jan. 1993. [23] D. C. Schmidt and T. Suda, “Measuring the Performance of Parallel Message-based Process Architectures,” in *Proce edings of the Conference on Computer Communications (INFOCOM*), (Boston, MA), pp. 624–633, IEEE, April 1995.

[24] Mats Bjorkman and Per Gunningberg, “Locking Strategies in Multiprocessor Implementations of Protocols,” in *Proce edings of the Symposium on Communications Architectures and Protocols (SIGCOMM*), (San Francisco, California), ACM,

1993.

[25] D. C. Schmidt, “A Family of Design Patterns for Application-level Gateways,” *The Theory and Practice of Object*

*Systems (Special Issue on Patterns and Pattern Languages*),vol.2, no. 1, 1996.

[26] D. C. Schmidt and P. Stephenson, “An Object-Oriented Framework for Developing Network Server Daemons,” in *P*

*roceedings of the 2nd C++ World Conferenc*e, (Dallas, Texas), SIGS, Oct. 1993.

[27] G. Booch and M. Vilot, “Simplifying the Booch Components,” *C++ Repor*t, vol. 5, June 1993.

[28] M. A. Linton and P. R. Calder, “The Design and Implementation of InterViews,” in *Proceedings of the USENIX C*

*++ Worksho*p, November 1987.

[29] T. Harrison and D. C. Schmidt, “Thread-Specific Storage: A Pattern for Reducing Locking Overhead in Concurrent

Programs,” in *OOPSLA Workshop on Design Patterns for Concurrent, Parallel, and Distributed System*s, ACM, October

1995.

[30] D. C. Schmidt, “Acceptor and Connector: Design Patterns for Actively and Passively Initializing Network Services,”

in *Workshop on Pattern Languages of Object-Oriented Programs at ECOOP ’9*5, (Aarhus, Denmark), August 1995.

[31] N. C. Hutchinson and L. L. Peterson, “The x-kernel: An Architecture for Implementing Network Protocols,” *IEEE T*

*ransactions on Software Engineerin*g, vol. 17, pp. 64–76, January 1991.

[32] D. C. Schmidt, “The ADAPTIVE Communication Environment: Object-Oriented Network Programming Components for Developing Client/Server Applications,” in *Proceedings of the 12th Annual Sun Users Group Conferenc*e,(San Francisc o, CA), pp. 214–225, SUG, June 1994.

[33] UNIX Software Operations, *UNIX System V Release 4 Programmer’s Guide: STREAM*S. Prentice Hall, 1990. [34] W. R. Stevens, *TCP/IP Illustrated, Volume* 2. Reading, Massachusetts: Addison Wesley, 1993.

[35] D. C. Schmidt, T. H. Harrison, and E. Al-Shaer, “Object-Oriented Components for High-speed Network Programmin g,” in *Proceedings of the 1st Conference on Object-Oriented Technologies and System*s,(Monterey,CA), USENIX, June 19

95.

第 **5** 章 **C/C++**线程专有存储（**Thread-Specific Storage**）：用于访

问“**per-Thread**”状态的对象行为模式

Douglas C. Schmidt Nat Pryce15 Timothy H. Harrison16

摘 要

在理论上，使应用多线程化可以改善性能（通过同时执行多个指令流），并简化程序结构（通过允许 每个线程同步地、而不是反应式地或异步地执行）。而在实践中，由于获取和释放锁的开销，多线程应用 常常并不比单线程应用执行得更好，甚至还会更糟。此外，由于避免条件竞争和死锁所需的复杂的并发 控制协议，多线程应用常常难以编程。

本论文描述线程专有存储（*Thread-Specific Storage*）模式，该模式可减轻多线程性能和编程复杂性 的若干问题。通过允许多个线程使用一个逻辑上的全局访问点来获取线程专有数据，而又不给每次访问 带来锁定开销，线程专有存储模式可改善性能，并简化多线程应用。

**5.1** 意图

允许多个线程使用一个逻辑上的全局访问点来获取线程专有数据，而又不给每次访问带来锁定开销。

**5.2** 动机

**5.2.1** 上下文和压力

线程专有存储应被用于这样的多线程应用：它们经常访问那些逻辑上是全局的、而物理上是专有于 每个线程的对象。例如，像 UNIX 和 Win32 这样的操作系统使用 errno 来向应用报告错误信息。当错误在 系统调用中发生时，OS 设置 errno 来报告问题、并返回文档化的失败状态。当应用检测到失败时，它检 查 errno 来确定发生了何种类型的错误。

15 本研究部分地得到了来自西门子 AG 的授权支持。

16 本研究由 British Telecom, plc 资助。

例如，考虑下面典型的 C 代码片段，它接收来自非阻塞 TCP socket 的缓冲区：

// One global errno per-process. extern int errno;

void \*worker (SOCKET socket)

{

// Read from the network connection

// and process the data until the connection

// is closed. for (;;)

{

char buffer[BUFSIZ];

int result = recv (socket, buffer, BUFSIZ, 0);

// Check to see if the recv() call failed. if (result == -1)

{

}

else

}

}

if (errno != EWOULDBLOCK)

// Record error result in thread-specific data. printf ("recv failed, errno = %d", errno);

// Perform the work on success. process\_buffer (buffer);

如果 recv 返回-1，代码检查 errno 是否等于 EWOULDBLOCK，如果不是（例如，如果 errno = EINTR）

就打印出错误消息；返回的不是-1 代码就处理它接收到的缓冲区。

**5.2.2** 常见陷阱和缺陷

尽管上面所示的“全局错误变量”方法17对于单线程应用工作得相当好，在多线程应用中却会发生微 妙的问题。特别地，占先式多线程系统中的条件竞争会导致一个线程中的方法所设置的 errno 值被另一线 程中的应用错误地解释。因而，如果多个线程同时执行 worker 函数，全局版本的 errno 就有可能会由于 条件竞争而被不正确地设置。

例如，在图 5-1 中两个线程（T1 和 T2）可以在 socket 上执行 recv 调用。在此例中，T1 的 recv 返回

-1，并设置 errno 为 EWOULDBLOCK，指示目前没有数据在 socket 上排队。但是在它检查这种情况之前，

T1 被占先，T2 运行。假设 T2 被中断，它设置 errno 为 EINTR。如果 T2 随之又立即被占先，T1 将会错

17 附录讨论了使用其他可选技术来报告错误的权衡（比如异常和传递一个显式的错误参数给每一调用）。

误地假定它的 recv 调用被中断，并执行错误的动作。因而，这个程序是错误的和不可移植的，因为它的 行为依赖于线程执行的顺序。

在这之下的问题是对全局 errno 值的设置和测试发生在两个步骤中：（1）recv 调用设置该值和（2） 应用测试该值。因此，“显而易见”的解决方案，即用互斥体包装 errno 并不能解决竞争状态，因为设置/ 测试涉及到多个操作（也就是，它不是原子的）。

解决此问题的一种途径是创建更为成熟的锁定协议。例如，recv 调用可以在内部获取 errno\_mutex， 并且必须由应用在 recv 返回、errno 的值被测试后来释放它。但是，此方案并不合乎需要，因为应用可能 会忘记释放锁，从而导致饥饿和死锁。而且，如果应用必须在每次库调用后检查错误状态，额外的锁定 开销将会显著地降低性能，即使在没有使用多个线程的情况下也是如此。

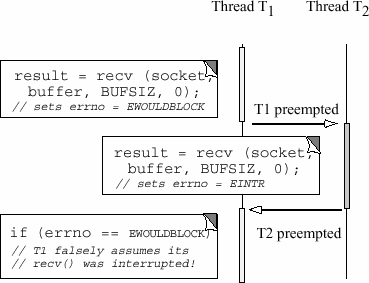


图 5-1 多线程程序中的条件竞争

**5.2.3** 解决方案：线程专有存储

上面描述的陷阱和缺陷的一种通用解决方案是使用线程专有存储模式。该模式消除以下压力：

z 效率：线程专有存储允许线程中的一系列方法原子地访问线程专有的对象，而又不会给每次访问带来 锁定开销。

z 简化应用编程：对于应用程序员来说，线程专有存储易于使用，因为系统开发者可以通过数据抽象或 宏来使线程专有存储的使用在源码级完全透明化。

z 高度可移植：线程专有存储在大多数多线程 OS 平台上都可用，并且可以在缺乏它的平台上（比如 V

xWorks）方便地实现。

因此，不管应用是运行在单线程还是多线程中，使用线程专有存储模式都不会带来额外开销，并且 无需改动代码。例如，下面的代码演示 errno 是如何在 Solaris 2.x 上定义的：

// defined in <sys/errno.h>).

#if defined (\_REENTRANT)

// The \_errno() function returns the

// thread-specific value of errno.

#define errno (\*\_errno())

#else

// Non-MT behavior is unchanged. extern int errno;

#endif /\* REENTRANT \*/

void \*worker (SOCKET socket)

{

// Exactly the same implementation shown above.

}

如果\_REENTRANT 标志被设置，errno 符号就被定义为调用名为\_errno 的助手函数的宏，此函数返回一

个指向 errno 的线程专有值的指针。该指针被宏去除引用，以使它能够任意出现在赋值运算的左边或右边。

**5.3** 适用性

应用有以下特性时可使用线程专有存储：

z 应用最初的编写假定了单线程控制，并正在被移植到多线程环境，而又不能改变现有 API；或是

z 应用含有多个占先式线程控制，可以任意的调度顺序并发执行；以及

z 每个线程控制调用一系列方法，这些方法共享只对该线程来说是公用的数据；以及

z 在每个线程中被对象共享的数据必须通过一个全局可见的访问点来访问；该访问点“逻辑地”与其他 线程共享，但在“物理上” 对于每个线程却是唯一的；以及

z 数据在方法间隐式地传递，而不是经由参数显式地传递。18

理解上面描述的特性对于使用（或不使用）线程专有存储模式来说是至关紧要的。例如，UNIX errno 变 量是一个数据例子：（1）逻辑上全局，但是物理上线程专有，以及（2）在方法间隐式地传递。

当应用有以下特性时，不要使用线程专有存储模式：

z 多个线程为单个任务协同工作，该任务需要并发访问共享数据。例如，多线程应用可以对在内存中的 数据库并发地进行读写。在这样的情况下，线程必须共享不是线程专有的记录和表。如果使用线程专 有存储来存储此数据库，线程就不能共享这些数据。因而，对数据库记录的访问必须通过同步原语（例 如，互斥体）来控制，以使线程能在共享数据上协作。

z 维护物理和逻辑上都分离的数据要更为直观和高效。例如，通过将数据作为参数显式地传递给所有方

18 在将单线程 API 移植到多线程系统时，这样的情形很常见。

法，有可能使线程访问仅在每个线程中可见的数据。在这样的情况下，线程专有存储模式有可能是不 必要的。

**5.4** 结构和参与者

图 5-2 演示线程专有存储中的以下参与者的结构： 应用线程（**Application Thread**）

z 应用线程使用 TS Object Proxy（TS 对象代理）来访问驻留在线程专有存储中的 TS Object。如 5.9 所示，线程专有存储模式的实现可以使用“灵巧指针”（smart pointer）来隐藏 TS Object Proxy，以 使应用看起来像是在直接访问 TS Object。

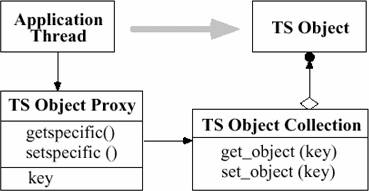


图 5-2 线程专有存储模式中的参与者的结构

线程专有对象代理（**TS Object Proxy**）（**errno** 宏）

z TS Object Proxy 定义 TS Object 的接口。它负责通过 getspecific 和 setspecific 方法来为每个应用线程 提供对一个唯一的对象的访问。例如，在 5.2 的错误处理例子中，errno TS Object 是一个 int。

一个 TS Object Proxy 的实例负责一种类型的对象，也就是，它为所有访问代理的线程而充当对线程 专有 TS Object 进行访问的中介。例如，多个线程可以使用同一个 TS Object Proxy 来访问线程专有 的 errno 值。代理所存储的 key（专有钥）值是由 TS Object Collection（TS 对象集合）在代理被创建 时分配的，并由 getspecific 和 setspecific 方法传递给集合。

TS Object Proxy 的目的是隐藏 key 和 TS Object Collection。没有代理，Application Thread 必须显式 地获取集合和使用专有钥。如 5.9 所示，线程专有存储的大多数细节可以通过 TS Object Proxy 的灵 巧指针来完全隐藏。

线程专有对象（**TS Object**）（**\* \_errno**（）**value**）

z TS Object 是特定线程的线程专有对象的实例。例如，线程专有的 errno 是 int 类型的对象。它由 TS Object Collection 管理，并且只能通过 TS Object Proxy 来访问。

线程专有对象集合（**TS Object Collection**）

z 在复杂的多线程应用中，线程的 errno 值可以是驻留在线程专有存储中的许多类型的数据中的一种。 因而，要获取线程专有错误数据的线程，必须使用一个 key。这个 key 必须与 errno 相关联，以允许 线程访问 TS Object Collection 中的正确条目。

TS Object Collection 含有一组与特定线程相关联的所有线程专有对象，也就是，每个线程都有唯一 的 TS Object Collection。TS Object Collection 将 key 映射到线程专有的 TS Object。通过 get\_object(k ey)和 set\_object(key)方法，TS Object Proxy 使用 key 来从 TS 对象集合中获取一个特定的 TS 对象。

**5.5** 协作

图 5-3 中的交互图演示线程专有存储模式中的参与者之间的以下协作：

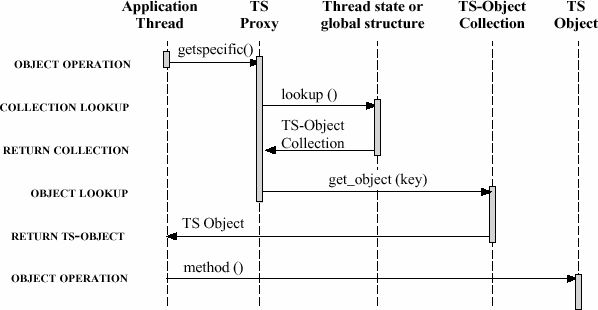


图 5-3 线程专有存储模式中的参与者之间的交互

z 定位 **TS Object Collection**：每个 Application Thread 中的方法在 TS Object Proxy 上调用 getspecific 和 setspecific 方法来访问 TS Object Collection，后者存储在线程中、或在通过线程 ID19进行索引的全 局结构中。

z 从线程专有存储中获取 **TS Object**：一旦 TS Object Collection 被定位，TS Object Proxy 就使用它的

key 来从集合中获取恰当的 TS Object。

z 设置**/**获取 **TS Object** 的状态：在这里，应用线程使用普通的 C++方法调用来在 TS Object 上进行操 作。无需进行锁定，因为对象通过一个指针来引用，而该指针仅能在调用线程中进行访问。

19 进程中的每个线程都含有一个唯一的标识值，称为“线程 ID”；与进程 ID 的概念相类似。

**5.6** 效果

**5.6.1** 好处

使用线程专有存储有若干好处，包括： 效率：线程专有存储可实现成无需对线程专有数据进行锁定。例如，通过将 errno 放入线程专有存储中，

每个线程都可以可靠地设置和测试该线程中的方法的完成状态，而无需使用复杂的同步协议。这排除了 线程中共享数据的锁定开销，比起获取和释放互斥体要更为迅捷[1]。

易于使用：对于应用程序员来说，线程专有存储使用起来很简单，因为系统开发者可以通过数据抽象或 宏来使线程专有存储的使用在源码级完全透明化。

**5.6.2** 缺点

使用线程专有存储还有着以下缺点： 它鼓励了（线程安全的）全局变量的使用：许多应用不要求多个线程通过公用访问点来访问线程专有的

数据。如果是这样，数据的存储应使只有拥有该数据的线程可对它进行访问。例如，考虑一个网络服务 器，它使用工作者线程池来处理来自客户的请求。这些线程可能会记录所执行服务的数量和类型。这个 日志机制可以作为使用线程专有存储的全局 Logger 对象来访问。但是，更简单的方法是将每个工作者线 程表示为主动对象[2]，并在其内部存储 Logger 的实例。在这样的情况下，只要将 Logger 作为参数传递 给主动对象中的所有函数，对 Logger 的访问就不会产生额外开销。

它隐藏了系统的结构：线程专有存储的使用隐藏了应用中的对象之间的关系，可能会导致应用更难被理 解。如附录 A.2 所描述的，在某些情况下，显式地表现对象间的关系可以消除对线程专有存储的需要。

**5.7** 实现

线程专有存储模式可以通过多种途径来实现。这一部分解释实现该模式所需的每一步骤。这些步骤 被总结如下：

**1.** 实现 **TS Object Collection**：如果 OS 不提供线程专有存储的实现，它可以使用任何一种可用以维护

TS Object Collection 中数据结构的一致性的机制来实现。

**2.** 封装线程专有存储的细节：如 5.8 所示，线程专有存储的接口通常是弱类型和易错的。因而，一旦有 线程专有存储的实现允许，就使用 C++编程语言特性（比如模板和重载）来将线程专有存储的低级细 节隐藏在 OO API 后面。

这一部分的余下部分描述怎样实现低级的线程专有存储 API。5.8 提供了完整的示例代码，5.9 检查 通过 C++包装来封装低级的线程专有存储 API 的若干方法。

**5.7.1** 实现 **TS Object Collection**

图 5-2 中所示的 TS Object Collection 含有所有属于某个特定线程的 TS Object。该集合可以使用指 向 TS Object 的指针表来实现，这些对象通过 key 进行索引。线程必须通过 key 来在访问线程专有对象之 前定位 TS Object Collection。因此，第一个设计挑战就是决定怎样定位和存储 TS Object Collection。

TS 对象集合可通过以下两种方式存储：（1）外在于所有线程，或（2）内在于每一线程。对每种方 法的描述和评估如下：

**1.** 外在于所有线程：该方法定义每个线程的 ID 到它的 TS 对象集合表（如图 5-4 所示）的一个全局映射。 定位适当的集合可能需要使用读者/作者锁来防止竞争状态。但是一旦集合被定位，就不再需要另外 的锁定了，因为只有一个线程可在其 TS Object Collection 中活动。

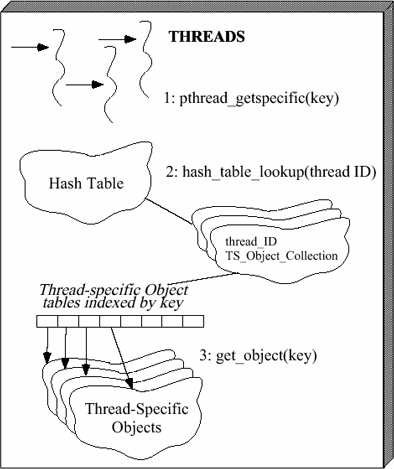


图 5-4 线程专有存储的外部实现

**2.** 内在于每个线程：该方法要求进程中的每个线程与它的其他内部状态（比如运行时线程栈、程序计数 器、通用寄存器和线程 ID）一起存储一个 TS 对象集合。当线程访问线程专有对象时，通过把相应的 key 作为线程内部的 TS Object Collection 的索引来获取该对象（如图 5-5 所示）。该方法不需要额外 的锁定。

在外部和内部实现方案之间选择需要开发者进行以下权衡：

定长 **vs.** 变长 **TS Object Collection**：对于外部和内部两种实现，如果线程专有钥的范围相对较小，TS Object Collection 可作为定长数组存储。例如，POSIX Pthread 标准定义了专有钥的最小数目\_POSIX\_TH READ\_KEYS\_MAX，遵从标准的实现必须对其加以支持。如图 5-5 所示，如果长度是固定的（例如，12

8 个专有钥，这是 POSIX 的缺省值），通过简单地使用对象的专有钥来在 TS Object Collection 数组中检 索，查找时间可为 O(1)。

但是，线程专有钥的范围也可以很大。例如，Solaris 线程对专有钥的数目没有预定义的限制。因此，

Solaris 使用了一种变长数据结构，这可能会增加管理 TS Object Collection 所需的时间。

定长 **vs.** 变长的线程 **ID** 到 **TS Object Collection** 的映射：线程 ID 的范围可以从很小到很大的值。这对 于内部实现来说没有任何问题，因为线程 ID 隐含地与相应的包含在线程状态中的 TS Object Collection 关联在一起。

但是，对于外部实现，要使每个可能的线程 ID 值都在定长数组中有相应条目可能是不现实的。相反， 让线程使用一个动态数据结构来将线程 ID 映射到 TS Object Collection，在空间上要更为经济。例如，一 种方法是在线程 ID 上使用哈希函数，以获得在哈希表桶中的一个偏移；在哈希表桶中含有一系列二元组， 将线程 ID 映射到它们相应的 TS Object Collection（如图 5-4 所示）。

全局 **vs.** 局部 **TS** 对象集合：内部方法局部地存储 TS Object Collection，而外部方法全局地存储它们。 取决于外部表的实现，全局定位可允许线程访问其他线程的 TS Object Collection。尽管这看起来废止了 线程专有存储的整个出发点，如果线程专有存储实现通过回收使用无用的专有钥来提供自动的垃圾收集 的话，这还是有用的。该特性对于限制专有钥数目为较小值的实现特别重要（例如，Windows NT 有着 每个进程 64 个专有钥的限制）。

但是，使用外部表增加了每个线程专有对象的访问时间，因为如果可全局访问的表被修改的话（例 如，创建新专有钥时），需要使用同步机制（比如读者/作者锁）来避免竞争状态。在另一方面，将 TS O bject Collection 局部地保持在每个线程的状态中需要每个线程有更多的存储，而总的内存消耗也不会减 少。

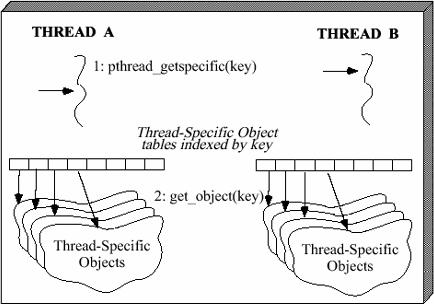


图 5-5 线程专有存储的内部实现

**5.8** 示例代码

**5.8.1** 实现 **POSIX Pthreads** 线程专有存储 **API**

下面的代码演示在使用专有钥定长数组将 TS Object“内部地”存储在线程中时，线程专有存储是怎 样实现的。这个例子是从 POSIX Pthreads[4]的一个公开可用的实现[3]改编而来。

下面所示的 thread\_state 结构含有线程的状态：

struct thread\_state

{

// The thread-specific error number. int errno\_;

// Thread-specific data values.

void \*key\_[\_POSIX\_THREAD\_KEYS\_MAX];

// ... Other thread state.

};

除了 errno 和线程专有存储指针的数组，该结构还包括了一个指向线程的栈和空间的指针；这些栈和空间

用于存储在上下文切换过程中存储/恢复的数据（例如，程序计数器）。 对于一个特定的线程专有对象，所有线程用同一个专有钥值来设置或取回线程专有的值。例如，如

果 Logger 对象正在被登记以跟踪线程专有的日志属性，线程专有的 Logger 代理将分配得到某个专有钥值 N。所有线程都将使用这个值 N 来访问它们的线程专有日志对象。目前正在使用的专有钥的总数目可相 对于所有线程全局地存储。如下所示：

typedef int pthread\_key\_t;

// All threads share the same key counter. static pthread\_key\_t total\_keys\_ = 0;

每次有新的线程专有钥被请求时，total\_keys\_count 都会自动增长，如下面的 pthread\_key\_create 函数 所示：

// Create a new global key and specify

// a "destructor" function callback.

Int pthread\_key\_create (pthread\_key\_t \*key, void (\*thread\_exit\_hook) (void \*))

{

if (total\_keys\_ >= \_POSIX\_THREAD\_KEYS\_MAX)

{

// pthread\_self() refers to the context of the

// currently active thread. pthread\_self ()->errno\_ = ENOMEM; return -1;

}

thread\_exit\_hook\_[total\_keys\_] = thread\_exit\_hook;

\*key = total\_keys\_++;

return 0;

}

pthread\_key\_create 函数分配一个新的专有钥值，唯一地标识一个线程专有数据对象。此外，它还允许应 用将一个 thread\_exit\_hook 与一个专有钥关联。该挂钩（hook）是一个函数指针，会被自动调用，当（1） 线程退出时，以及（2）有线程专有对象登记专有钥时。指向“线程退出挂钩”的函数指针的数组可被全 局地存储。如下所示：

// Exit hooks to cleanup thread-specific keys.

static void (\*thread\_exit\_hook\_[\_POSIX\_THREAD\_KEYS\_MAX]) (void);

下面的 pthread\_exit 函数演示线程退出挂钩函数是怎样在 pthread\_exit 的实现中被调用的：

// Terminate the thread and call thread exit hooks. void pthread\_exit (void \*status)

{

// ...

for (i = 0; i < total\_keys; i++)

if (pthread\_self ()->key\_[i] && thread\_exit\_hook\_[i])

// Indirect pointer to function call. (\*thread\_exit\_hook\_[i]) (pthread\_self ()->key\_[i]);

// ...

}

应用可为各个线程专有数据对象登记不同的函数，但是对于每个对象、会为每个线程调用同一个函

数。登记动态分配的线程专有存储对象是一种常见的使用方法。因此，线程退出挂钩通常看起来是这样 的：

static void cleanup\_tss\_Logger (void \*ptr)

{

// This cast is necessary to invoke

// the destructor (if it exists). delete (Logger \*) ptr;

}

该函数释放一个动态分配的 Logger 对象。

pthread\_setspecific 函数为调用线程将 value 绑定到给定的 key：

// Associate a value with a data key

// for the calling thread.

int pthread\_setspecific (int key, void \*value)

{

if (key < 0 || key >= total\_keys) {

pthread\_self ()->errno\_ = EINVAL;

return -1;

}

pthread\_self ()->key\_[key] = value;

return 0;

}

同样地，pthread\_getspecific 为调用线程把绑定到给定 Key 的数据存储到 value 中：

// Retrieve a value from a data key

// for the calling thread.

int pthread\_getspecific (int key, void \*\*value)

{

if (key < 0 || key >= total\_keys)

{

pthread\_self ()->errno\_ = EINVAL;

return -1;

}

\*value = pthread\_self ()->key\_[key];

return 0;

}

因为数据存储在每个线程的内部状态中，这些函数不需要任何额外的锁来访问线程专有数据。

**5.8.2** 在应用中使用线程专有存储

下面的例子演示怎样在可从多于一个线程调用的一个 C 函数中，使用来自 POSIX Pthread 规范的线 程专有存储 API，而又无须显式地调用初始化函数：

// Local to the implementation.

static pthread\_mutex\_t keylock = PTHREAD\_MUTEX\_INITIALIZER;

static pthread\_key\_t key;

static int once = 0;

void \*func (void)

{

void \*ptr = 0;

// Use the Double-Checked Locking pattern

// (described further below) to serialize

// key creation without forcing each access

// to be locked.

if (once == 0)

{

pthread\_mutex\_lock (&keylock);

if (once == 0)

{

// Register the free(3C) function

// to deallocation TSS memory when

// the thread goes out of scope. pthread\_key\_create (&key, free); once = 1;

}

pthread\_mutex\_unlock (&keylock);

}

pthread\_getspecific (key, (void \*\*) &ptr);

if (ptr == 0)

{

ptr = malloc (SIZE);

pthread\_setspecific (key, ptr);

}

return ptr;

}

**5.8.3** 评估

上面的解决方案直接在应用代码中调用线程专有库函数（比如 pthread\_getspecific 和 pthread\_setspeci fic）。但是这些直接用 C 编写的 API 有以下局限：

z 不可移植：POSIX Pthreads、Solaris 线程和 Win32 线程非常类似。但是，Win32 线程的语义有着微妙 的不同，因为它们不提供一种可靠的方法来在线程退出时清理在线程专有存储中分配的对象。而且， 在 Solaris 线程中没有 API 用于删除专有钥。这使得开发者难以在 UNIX 和 Win32 平台间编写可移植 代码。

z 难以使用：尽管省略了错误检查，在 5.8.2 中的 func 例子所演示的锁定操作仍然是复杂而又不直观的。 该代码是“双重检查锁定模式”（Double-Checked Locking pattern）[5]的 C 实现。将这个 C 实现与 5.

9.2.1 中的 C++版本相比较，以观察使用 C++包装所带来的更多的简单性、清晰性和类型安全性是富 有启发意义的。

z 非类型安全的：POSIX Pthreads、Solaris 和 Win32 线程专有存储接口将指向线程专有对象的指针作为

void \*存储。尽管这种方法很灵活，它很容易造成错误，因为 void \*消除了类型安全性。

**5.9** 变种

5.8 演示了怎样通过 POSIX pthread 接口实现和使用线程专有存储模式。但是，所得到的解决方案不 可移植、难以使用，且不是类型安全的。为了克服这些局限，可以开发另外的类和 C++包装来健壮地以 一种类型安全的方式编写线程专有存储。

这一部分演示怎样使用 C++包装来封装 POSIX Pthreads、Solaris 或 Win32 线程提供的低级线程专有 存储机制。5.9.1 描述怎样通过硬编码的 C++包装来封装 POSIX Pthread 库接口，5.9.2 描述一种使用 C++ 模板包装的更为通用的解决方案。用于每种可选方法的例子是 5.6.2 描述的 Logger 抽象的一个变种。

**5.9.1** 硬编码的 **C++**包装

使一个类的所有实例成为线程专有的一种方法是直接使用线程专有库例程。实现该方法所需的步骤 描述如下。错误检查已被省略到最少以节约空间。

**5.9.1.1** 定义线程专有状态信息

第一步是决定必须在线程专有存储中存取的对象状态信息。例如，Logger 可能有以下状态：

class Logger\_State

{

public:

int errno\_;

// Error number.

int line\_num\_;

// Line where the error occurred.

// ...

};

每个线程都将拥有它自己的一份这些状态信息的拷贝。

**5.9.1.2** 定义外部类接口

下一步是定义被所有应用线程使用的外部类接口。下面的 Logger 外部类接口看起来就像是一个平常 的非线程专有的 C++类：

class Logger

{

public:

// Set/get the error number. int errno (void);

void errno (int);

// Set/get the line number. int line\_num (void);

void line\_num (int);

// ...

};

**5.9.1.3** 定义线程专有的助手函数

该步骤使用线程库提供的线程专有存储函数来定义一个助手函数，它返回一个指向适当的线程专有 存储的指针。这个助手函数通常执行以下步骤：

**1.** 专有钥初始化：为每个线程专有对象初始化一个专有钥，并使用此专有钥来存/取一个指向动态分配 内存的线程专有指针；此内存含有内部结构的实例。该代码可以被实现如下：

class Logger

{

public:

// ... Same as above ...

protected:

Logger\_State \*get\_tss\_state (void);

// Key for the thread-specific error data. pthread\_key\_t key\_;

// "First time in" flag. int once\_;

};

Logger\_State \*Logger::get\_tss\_state (void)

{

// Check to see if this is the first time in

// and if so, allocate the key (this code

// doesn’t protect against multi-threaded

// race conditions...). if (once\_ == 0)

{

pthread\_key\_create (this->key\_, free);

once\_ = 1;

}

Logger\_State \*state\_ptr;

// Get the state data from thread-specific

// storage. Note that no locks are required... pthread\_getspecific (this->key\_, (void \*\*) &state\_ptr);

if (state\_ptr == 0)

{

state\_ptr = new Logger\_State;

pthread\_setspecific (this->key\_, (void \*) state\_ptr);

}

// Return the pointer to thread-specific storage. return state\_ptr;

};

**2.** 获取指向线程专有存储对象的指针：外部接口中的每个方法都调用 get\_tss\_state 助手函数来获取指向

驻留在线程专有存储中的 Logger\_State 对象的指针。如下所示：

int Logger::errno (void)

{

return this->get\_tss\_state ()->errno\_;

}

**3.** 执行正常操作：一旦外部接口方法有了该指针，应用就可以对线程专有对象进行操作，就如同它是平 常的（也就是，非线程专有的）C++对象：

Logger logger;

int recv\_msg (HANDLE socket, char \*buffer, size\_t bufsiz)

{

if (recv (socket, buffer, bufsiz, 0) == -1)

{

logger->errno () = errno;

return -1;

}

// ...

}

int main (void)

{

// ...

if (recv\_msg (socket, buffer, BUFSIZ) == -1

&& logger->errno () == EWOULDBLOCK)

// ...

}

**5.9.1.4** 对硬编码包装的评价

使用硬编码包装的优点是它将应用与线程专有库函数的知识屏蔽开来。该方法的缺点是它不能促进 复用性、可移植性，或是灵活性。特别地，对于每个线程专有类，开发者都需要在类中重新实现线程专 有助手方法。

而且，如果应用被移植到有着不同的线程专有存储 API 的平台上，就必须改变在每个线程专有类中 的代码，以使用新的线程库。此外，直接改变线程专有类使得程序难以变更线程策略。例如，将一个线 程专有类变为全局类需要对代码进行侵入性的变动，从而降低了灵活性和复用性。特别地，每次对对象 内部的状态信息的访问都将要求改变用于从线程专有存储取回该状态的助手方法。

**5.9.2 C++**模板包装

一种更为可复用、可移植和灵活的方法是实现 TS Object Proxy 模板，负责所有的线程专有方法。该 方法允许类与线程专有存储怎样实现的知识去耦合。通过定义称为 TSS 的代理类，该解决方案改善了代 码的可复用性、可移植性和灵活性。如下所示，该类是一个模板，通过其对象驻留在线程专有存储中的 类来参数化：

// TS Proxy template template <class TYPE> class TSS

{

public:

// Constructor. TSS (void);

// Destructor

˜TSS (void);

// Use the C++ "smart pointer" operator to

// access the thread-specific TYPE object. TYPE \*operator-> ();

private:

// Key for the thread-specific error data. pthread\_key\_t key\_;

// "First time in" flag. int once\_;

// Avoid race conditions during initialization. Thread\_Mutex keylock\_;

// Cleanup hook that deletes dynamically

// allocated memory.

static void cleanup\_hook (void \*ptr);

};

该类中的方法描述如下。和前面一样，错误检查已被省略到最少以节约空间。

**5.9.2.1 C++**委托操作符（**Delegation Operator**）

通过重载 C++委托操作符（操作符->），应用可以调用 TSS 代理上的方法，就好像是在调用目标类一 样。在此实现中使用的 C++委托操作符控制所有对类 TYPE 的线程专有对象的访问。操作符->方法受到 了来自 C++编译器的特殊对待。如 5.9.2.3 所述，它先从线程专有存储那里获取一个指向适当的 TYPE 的 指针，随后就重新委托原来在其上调用的方法。

TSS 类中的大多数工作都在下面所示的操作符->方法中执行：

template <class TYPE> TYPE \* TSS<TYPE>::operator-> ()

{

TYPE \*tss\_data = 0;

// Use the Double-Checked Locking pattern to

// avoid locking except during initialization.

// First check.

if (this->once\_ == 0)

{

// Ensure that we are serialized (constructor

// of Guard acquires the lock).

Guard <Thread\_Mutex> guard (this->keylock\_);

// Double check

if (this->once\_ == 0)

{

pthread\_key\_create (&this->key\_, &this->cleanup\_hook);

// \*Must\* come last so that other threads

// don’t use the key until it’s created. this->once\_ = 1;

}

// Guard destructor releases the lock.

}

// Get the data from thread-specific storage.

// Note that no locks are required here... pthread\_getspecific (this->key\_, (void \*\*) &tss\_data);

// Check to see if this is the first time in

// for this thread. if (tss\_data == 0)

{

// Allocate memory off the heap and store

// it in a pointer in thread-specific

// storage (on the stack...). tss\_data = new TYPE;

// Store the dynamically allocated pointer in

// thread-specific storage.

pthread\_setspecific (this->key\_, (void \*) tss\_data);

}

return tss\_data;

}

TSS 模板是一个代理，它透明地将普通 C++类转换为类型安全、线程专有的类。它结合了操作符->

和其他一些 C++特性，像模板、内联和重载。它还利用了像双重检查锁定优化[5]和代理[6, 7]这样的模式。 在代码中，双重检查锁定优化模式用于在操作符->中两次测试 once\_标志。尽管多个线程可以同时访

问 TSS 的同一实例，仅有一个线程可以合法地创建一个专有钥（也就是，通过 pthread\_key\_create）。随 后所有线程将使用这一专有钥来访问参数化类 TYPE 的线程专有对象。因此，操作符->使用 Thread\_Mut ex keylock\_来确保仅有一个线程执行 pthread\_key\_create。

第一个获取 keylock\_的线程设置 once\_为 1，所有调用操作符->的后续线程将发现 once != 0，于是跳

过初始化步骤。对 once\_的第二次测试处理这样的情况：在第一个线程设置 once\_为 1 之前，多个并行执 行的线程在 keylock\_处排队等候。在这种情况下，当其他排队等待的线程最终获得互斥体 keylock\_，它们 会发现 once\_等于 1，就不会执行 pthread\_key\_create。

一旦 key\_被创建，不再需要有进一步的锁定来访问线程专有数据。这是因为 pthread\_{getspecific, se tspecific}函数从调用线程的状态处获取类 TYPE 的 TS Object，而此线程状态是独立于其他线程的。

除了减少锁定开销，上面所示的类 TSS 的实现将应用代码与对象是专有于调用线程的这一事实屏蔽 开来。为达成这一点，该实现使用了像模板、操作符重载和委托操作符（也就是，操作符->）这样的 C+

+特性。

**5.9.2.2** 构造器和析构器

TSS 类的构造器是很小的，它只是简单地初始化局部实例变量：

template <class TYPE>

TSS<TYPE>::TSS (void): once\_ (0), key\_ (0) {}

注意我们没有在构造器中分配 TSS 专有钥或是一个新的 TYPE 实例。这样设计有若干原因：

z 线程专有存储语义：最初创建 TSS 对象的线程（例如，主线程）常常不是使用该对象的线程（例如， 工作者线程）。因此，在构造器中预先初始化一个新的 TYPE 并没有好处，因为此实例只能被主线程 访问。

z 延期的初始化：在某些 OS 平台上，TSS 专有钥是有限的资源。例如，Windows NT 仅允许每个进程 总共有 64 个 TSS 专有钥。因此，不到绝对需要的时候，不应分配专有钥。相反，初始化被延期到操 作符->第一次被调用时。

TSS 析构器给我们带来若干棘手的设计问题。显而易见的解决方案是在操作符->中释放所分配的

TSS 专有钥。但是，这一方法有若干问题：

■ 特性缺乏：Win32 和 POSIX pthreads 定义了函数来释放 TSS 专有钥。但是，Solaris 没有。 因此，很难编写一个可移植的包装。

■ 竞争状态：Solaris 线程不提供函数来释放 TSS 专有钥的主要原因是实现起来很昂贵。问题 在于每个线程都分别维护通过同一个专有钥引用的对象。只有在所有这些线程退出、且内存 被回收后才能安全地释放该专有钥。

作为上面所提到的问题的结果，我们的析构器是一个空操作。

template <class TYPE> TSS<TYPE>::˜TSS (void)

{

}

cleanup\_hook 是一个静态方法，它在删除其 ptr 参数之前，将其强制转换到适当的 TYPE \*。

template <class TYPE> void

TSS<TYPE>::cleanup\_hook (void \*ptr)

{

// This cast is necessary to invoke

// the destructor (if it exists). delete (TYPE \*) ptr;

}

这确保了在线程退出时，每个线程专有对象的析构器都会被调用。

**5.9.2.3** 用例

下面的方案基于 C++模板包装来解决我们一直在讨论的例子：被多个工作者线程访问的线程专有存 储 Logger。

// This is the "logically" global, but

// "physically" thread-specific logger object,

// using the TSS template wrapper. static TSS<Logger> logger;

// A typical worker function. static void \*worker (void \*arg)

{

// Network connection stream.

SOCK\_Stream \*stream = static\_cast <SOCK\_Stream \*> arg;

// Read from the network connection

// and process the data until the connection

// is closed. for (;;)

{

char buffer[BUFSIZ];

int result = stream->recv (buffer, BUFSIZ);

// Check to see if the recv() call failed. if (result == -1)

{

}

else

if (logger->errno () != EWOULDBLOCK)

// Record error result.

logger->log ("recv failed, errno = %d", logger->errno ());

// Perform the work on success. process\_buffer (buffer);

}

}

考虑上面对 logger->errno 的调用。C++编译器用两个方法调用来替换这一调用。第一个是对 TSS::操

作符->的调用，它返回一个驻留在线程专有存储中的 Logger 实例。随后编译器生成第二个方法，调用前 面的调用返回的 Logger 对象的 errno 方法。在这种情况下，TSS 作为一个代理，允许应用访问和操作线 程专有的错误值，就如同它们是平常的 C++对象一样。20

上面的 Logger 例子是一个好范例，在其中使用逻辑上的全局访问点是有益的。因为 worker 函数是全 局的，线程要同时管理 Logger 对象的物理和逻辑的分离并不那么简明。相反，线程专有的 Logger 允许多 个线程使用单个逻辑访问点来操作物理上分离的 TSS 对象。

**5.9.2.4** 评估

基于 C++操作符->的 TSS 代理设计有以下好处：

z 最大化代码复用：通过使线程专有方法与特定的应用类（也就是，形式参数类 TYPE）去耦合，不再 需要重写微妙的线程专有钥的创建和分配逻辑。

z 提高可移植性：将应用移植到其他线程库（比如 Win32 中的 TLS 接口）只需要改变 TSS 类，而不是 所有使用此类的应用。

z 更大的灵活性和透明性：将一个类变为线程专有类（或相反），仅需要改变此类对象的定义方式。这 可以在编译时被决定，如下所示：

#if defined (\_REENTRANT)

static TSS<Logger> logger;

#else

// Non-MT behavior is unchanged. Logger logger;

#endif /\* REENTRANT \*/

注意不管使用的是线程专有还是非线程专有形式的 Logger，Logger 的使用方式都保持不变。

**5.10** 已知应用

下面是线程专有存储模式的已知应用：

20 注意 C++操作符->不能为像 int 这样的内建类型工作，因为它们没有方法可以被委托，这也是为什么我们不能使用 int

来代替上面所用的 Logger 类的原因。

z 在支持 POSIX 和 Solaris 线程 API 的 OS 平台上实现的 errno 机制是被广泛使用的线程专有存储模式

的例子[1]。此外，与 Win32 一起提供的 C 运行时库支持线程专有的 errno。Win32 GetLastError/SetLa stError 也实现了线程专有存储模式。

z 在 Win32 操作系统中，窗口属于线程[8]。每个拥有窗口的线程都有一个私有的消息队列，OS 会在其 中放入用户接口事件。获取等待处理的下一消息的 API 调用使下一个消息从调用线程的消息队列中出 队，而该消息队列就驻留在线程专有存储中。

z OpenGL[9]是一个用于渲染三维图形的 C API。程序根据多边形来渲染图形；多边形通过反复调用 gl Vertex 函数、以传递多边形的每一顶点给库来描述。在顶点被传递给库之前设置的状态变量精确地决 定 OpenGL 在接收到顶点时如何进行绘制。该状态在 OpenGL 库中、或是在图形卡上作为封装的全局 变量存储。在 Win32 平台上，OpenGL 库为每个使用该库的线程在线程专有存储中维护一组唯一的状 态变量。

z 线程专有存储被用于在 ACE 网络编程工具包[10]中实现它的错误处理方案，该方案与 5.9.2.3 描述的

Logger 方法相类似。此外，ACE 还实现了 5.9.2 描述的线程安全的线程专有存储模板包装。

**5.11** 相关模式

用线程专有存储实现的对象常常被用作“per-thread”的单体（Singleton）[7]，例如，errno 就是一个 “per-thread”单体。但是，并非所有线程专有存储的使用都是单体，因为线程可以拥有从线程专有存储 中分配的一种类型的多个实例。例如，在 ACE[10]中实现的每一个 Task 对象都在线程专有存储中存储一 个清理挂钩。

5.8 所示的 TSS 模板类被用作代理，将库、框架和应用与 OS 线程库提供的线程专有存储的实现屏蔽 开来。

双重检查锁定优化模式[5]通常被用于这样的应用：它们利用线程专有存储模式来避免约束线程专有 存储钥的初始化顺序。

**5.12** 结束语

由于防止竞争状态和死锁所需的额外的并发控制协议，使现有应用多线程化常常会显著地增加软件 的复杂性[11]。通过允许多线程使用一个逻辑上的全局访问点来获取线程专有数据，而又不给每次访问带 来锁定代价，线程专有存储模式减轻了一些同步开销和编程复杂性。

应用线程使用 TS Object Proxy 来访问 TS Object。代理委托 TS Object Collection 来获取相应于每个 应用线程的对象。这确保了不同的应用线程不会共享同一个 TS Object。

5.9.2 显示怎样实现线程专有存储模式的 TS Object Proxy，以确保线程通过强类型的 C++类接口来访 问只属于它们自己的数据。与其他模式（比如代理、单体和双重检查锁定）和 C++语言特性（比如模板 和操作符重载）相结合，可实现 TS Proxy，使得使用线程专有存储模式的对象可像传统对象一样被处理。

感谢

感谢 Peter Sommerlad 和 Hans Rohnert 对本论文早期版本的富有洞察力的意见。

参考文献

[1] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Wil liams, “Beyond Multiprocessing... Multithreading the SunOS Kernel,” in *Proceedings of the Summer USENIX Conferenc*e, (San Antonio, Texas), June 1992.

[2] R. G. Lavender and D. C. Schmidt, “Active Object: an Object Behavioral Pattern for Concurrent Programming,” in *P*

*attern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1

996.

[3] F. Mueller, “A Library Implementation of POSIX Threads Under UNIX,” in *Proceedings of the Winter USENIX Conf erenc*e, (San Diego, CA), pp. 29–42, Jan. 1993.

[4] IEEE, *Threads Extension for Portable Operating Systems (Draft 10*), February 1996.

[5] D. C. Schmidt and T. Harrison, “Double-Checked Locking – An Object Behavioral Pattern for Initializing and Acces sing Thread-safe Objects Efficiently,” in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A Sys tem of Pattern*s. Wiley and Sons, 1996.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Softwar*e. Reading, MA: Addison-Wesley, 1995.

[8] C. Petzold, *Programming Windows 9*5. Microsoft Press, 1995.

[9] J. Neider, T. Davis, and M. Woo, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release* 1. Addison-Wesley, Reading, MA, 1993.

[10] D. C. Schmidt, “ACE: an Object-Oriented Framework for Developing Distributed Applications,” in *Proceedings of th e 6th USENIX C++ Technical Conferenc*e, (Cambridge, Massachusetts), USENIX Association, April 1994.

[11] J. Ousterhout, “Why Threads Are A Bad Idea (for most purposes),” in *USENIX Winter Technical Conferenc*e,(San

Diego, CA), USENIX, Jan. 1996.

[12] H. Mueller, “Patterns for Handling Exception Handling Successfully,” *C++ Repor*t, vol. 8, Jan. 1996.

[13] N. Pryce, “Type-Safe Session: An Object-Structural Pattern,” in *Submitted to the 2nd European Pattern Languages of*

*Programming Conferenc*e, July 1997.

[14] J. Gosling and F. Yellin, *The Java Application Programming Interface Volume 2: Window Toolkit and Applet*s. Add ison-Wesley, Reading, MA, 1996.

附录 **A** 可选方案

在实践中，线程专有存储通常用于为面向对象软件解决下面两种使用情况：

1. 在模块间隐式地传递信息（例如，错误信息）。

2. 将遗留的以过程风格编写的单线程软件改编到现代的多线程操作系统和编程语言。

但是，对于#1 使用情况，避免使用线程专有存储常常是一个好主意，因为它可能会增强模块间的耦合并 降低可复用性。例如，对于错误处理的情况，常常可以使用 A.1 描述的异常来避开线程专有存储。

除非重新设计，对于#2 使用情况，不能不使用线程专有存储。但是在设计新软件时，常常可以使用 如下所描述的异常处理、显式的组件间通信上下文或对象化线程来避开线程专有存储。

**A.1** 异常处理

在模块间报告错误的一种优雅方法是使用异常处理。许多现代语言，比如 C++和 Java，使用异常处 理来作为错误报告的机制。它还被用于一些操作系统中，比如 Win32。例如，下面的代码演示一种假想 的 OS，其系统调用会扔出异常。

void \*worker (SOCKET socket)

{

// Read from the network connection

// and process the data until the connection

// is closed. for (;;)

{

char buffer[BUFSIZ];

try

{

}

// Assume that recv() throws exceptions. recv (socket, buffer, BUFSIZ, 0);

// Perform the work on success. process\_buffer (buffer);

catch (EWOULDBLOCK)

{

continue;

}

catch (OS\_Exception error)

{

// Record error result in thread-specific data. printf ("recv failed, error = %s", error.reason);

}

}

}

使用异常处理有若干好处：

z 它是可扩展的：现代的 OO 语言通过一些尽量不侵犯现有接口和使用的特性（比如使用继承来定义异 常类层次）来便利异常处理策略和机制的扩展。

z 它干净地使错误处理和正常处理得以去耦合：例如，错误处理信息不是被显式地传递给操作的。而且， 应用不会由于没有完成对函数返回值的检查而偶然地“忽略”了一个异常。

z 它可以是类型安全的：在强类型的语言中，比如 C++和 Java，异常以一种强类型的方式被扔出和捕捉， 以增强错误处理代码的组织和正确性。与显式地检查线程专有错误值相反，编译器会确保对于每种类 型的异常，执行正确的处理方法。

但是，使用异常处理也有若干缺点：

z 它并非普遍可用的：并非所有语言都提供异常处理，而许多 C++编译器也并没有实现异常。同样地， 如果一种 OS 提供异常处理服务，它们必须被语言扩展支持，从而也就降低了代码的可移植性。

z 它使多种语言的使用复杂化：因为语言以不同的方式实现异常，或者根本不实现异常，在以不同语言 编写的组件扔出异常时，可能会很难将它们集成在一起。使用整数值或结构来报告错误信息提供了一 种普遍可用的解决方案。

z 它使资源管理复杂化：例如，由于增加了 C++代码块中退出路径的数目[12]，而导致这样的问题。如 果语言或编程环境不支持垃圾回收，就必须付出努力来确保动态分配的对象在异常扔出时被删除。

z 它有时间和**/**或空间效率低下的可能性：异常处理的糟糕实现会带来时间和空间的过度开销，即使没 有异常被扔出也是如此[12]。对于必须保持小而高效的嵌入系统来说，此开销可能会特别地成问题。

对于必须在多种平台上可移植地运行的系统级框架（比如内核级设备驱动器，或低级通信子系统） 来说，异常处理的缺点会特别地成问题。对于这些类型的系统，一种更为可移植、高效和线程安全的处 理错误的方法是定义一个错误处理器抽象，显式地维护关于操作的成功或失败的信息。

**A.2** 组件间通信的显式上下文

线程专有存储通常用于存储“per-thread”状态，以允许库和框架中的软件组件高效地进行通信。例 如，errno 被用于将错误值从被调用组件传递给调用者。同样地，对 OpenGL API 函数的调用传递信息给 OpenGL 库，并被存储在线程专有状态中。通过显式地将在组件间传递的信息表示为对象，可以避免使用 线程专有存储。

如果预先已经知道组件必须为它的用户存储的信息的类型，调用线程可以创建对象，并将其作为操 作的一个额外参数传递给组件。另外，组件必须创建一个对象来保持上下文信息，以响应来自调用线程 的请求；并在线程对组件加以使用之前将一个标识符传给线程。这些类型的对象常被称为“上下文对象”

（context object）；软件组件按需创建的上下文对象常被称为“会话”（session）。 下面的错误处理方案是一个简单的例子，演示调用线程怎样创建上下文对象；该方案传递一个显式

的参数给所有的操作：

void \*worker (SOCKET socket)

{

// Read from the network connection and

// process the data until the connection

// is closed. for (;;)

{

char buffer[BUFSIZ];

int result;

int errno;

// Pass the errno context object explicitly.

result = recv (socket, buffer, BUFSIZ, 0, &errno);

// Check to see if the recv() call failed. if (result == -1)

{

}

else

}

}

if (errno != EWOULDBLOCK)

printf ("recv failed, errno = %d", errno);

// Perform the work on success. process\_buffer (buffer);

组件创建的上下文对象可以使用类型安全的会话模式（Type-Safe Session Pattern）[13]来实现。在此

模式中，上下文对象存储组件所需的状态，并提供一个可被多态地调用的抽象接口。组件返回一个指向 该抽象接口的指针给调用线程，后者随后调用接口的操作来使用组件。

OpenGL 和 Java AWT 库[14]（用于在诸如窗口、打印机或位图这样的设备上渲染图形）所提供接口 之间的差异演示了怎样使用类型安全的会话。在 AWT 中，程序通过从设备请求一个 GraphicsContext 来 在设备上进行绘制。GraphicsContext 封装在设备上进行渲染所需的状态，并提供了一个接口，通过它程 序可以设置状态变量，并调用绘制操作。可以动态地创建多个 GraphicsContext 对象，从而消除了任何保 持线程专有状态的需要。

与线程局部存储和异常处理相比较，使用上下文对象有以下这些好处：

z 它更加可移植：它不需要有可能不被普遍支持的语言特性；

z 它更为高效：线程可以直接存储和访问上下文对象，而不必在线程专有存储表中进行查找。也不需要 编译器构建额外的数据结构来处理异常；

z 它是线程安全的：上下文对象或会话句柄可以存储在线程的栈中，这自然是线程安全的。

但是使用由调用线程创建的上下文对象也有若干缺点：

z 它是强制性的：上下文对象必须传给所有操作，并且必须在每次操作后显式地检查。这搅乱了程序逻

辑，并有可能需要改变已有的组件接口，以增加错误处理器参数。

z 增加了每次调用的开销：每次调用都会有额外开销，因为必须给每个方法调用增加一个额外的参数， 而不管是否需要该对象。尽管在某些情况下，这是可以接受的，但对于执行非常频繁的方法，开销可 能会是相当显著的。相反，除非发生错误，基于线程专有存储的错误处理方案就不需要被使用。

与在调用线程中创建上下文对象相比较，使用由组件创建的会话有以下好处：

z 它较少强制性：线程不必显式地将上下文对象作为操作的参数传给组件。编译器会安排将指向上下文 对象的指针作为隐藏的 this 指针传给它的操作。

z 它使初始化和关闭自动化：在线程从组件那里获取会话之前，它不能开始使用会话。于是如果多个操 作在矛盾的状态中，组件就可以确保它们不会被调用。相反，如果组件使用隐藏的状态，调用者必须 在调用操作之前显式地初始化库，并在组件结束后将它关闭。忘记这样做会导致含混的错误或资源浪 费。

z 结构是显式的：不同代码模块之间的关系被显式地表示为对象，这使得要理解系统的行为变得更容易。

与在调用者的栈上创建上下文对象相比较，在组件中创建它们有以下缺点：

z 分配开销：组件必须在堆上、或是从某种封装的缓存那里分配会话对象。比起在栈上分配对象，这样 做的效率常常要更为低下。

**A.3** 对象化线程

在面向对象语言中，应用可以显式地将线程表示为对象。线程类可以通过从一个抽象基类派生来定 义，在此基类中封装了作为并发线程运行所需的状态；并调用一个实例方法来作为线程的入口。线程的 入口方法可以在基类中被定义为纯虚函数，并在派生类中定义。任何所需的线程专有状态（比如会话上 下文）可被定义为对象实例变量，并可为线程类的任何方法所用。对这些变量的同时访问可以通过使用 语言级访问控制机制、而不是显式的同步对象来防止。

下面使用 ACE Task（任务）[10]的一种变种来演示这一方法；任务可用于将线程控制和对象相关联：

class Task

{

public:

// Create a thread that calls the svc() hook. int activate (void);

// The thread entry point. virtual void svc (void) = 0;

private:

// ...

};

class Animation\_Thread : public Task

{

public:

Animation\_Thread (Graphics\_Context \*gc)

: device\_ (gc) {}

virtual void svc (void)

{

device\_->clear ();

// ... perform animation loop...

}

private:

Graphics\_Context \*device\_;

};

使用对象化线程有以下好处：

z 它更为高效：线程无需在隐藏的数据结构中进行查找、以访问线程专有状态。

z 它不具强制性：使用对象化线程时，指向当前对象的指针作为每个函数调用的额外参数被传递。不像 显式的会话上下文，在源码中该参数是隐藏的，并由编译器自动管理，从而可使源码保持整洁。

使用对象化线程有以下缺点：

z 不容易访问线程专有存储：只有类方法可以访问实例变量。这使得使用实例变量来在可复用库和线程 间进行通信变得不直观。无论如何，以这种方式使用线程专有存储增强了组件间的耦合。一般而言， 异常提供了在模块间报告错误的一种更为弱耦合的方法，尽管在像 C++这样的语言里它们有着自身的 陷阱和缺陷[12]。

z 额外开销：传给每个操作的额外、隐藏的参数会带来一些开销。在执行非常频繁的函数中，这些开销 可能会非常地显著。

第 **6** 章 主动对象（**Active Object**）：用于并发编程的对象行为模式

R. Greg Lavender Douglas C. Schmidt

摘 要

本论文描述主动对象（*Active Object*）模式。该模式使方法执行与方法调用去耦合，以简化对驻留在 它自己的线程控制中的对象的同步访问。主动对象模式允许一或多个交错访问数据的独立执行的线程被 建模为单个对象。这一并发模式能良好地适用于广泛的生产者*/*消费者和读者*/*作者应用类。该模式通常用 于需要多线程服务器的分布式系统中。此外，客户应用，比如窗口系统和网络浏览器，采用主动对象来 简化并发和异步的网络操作。

**6.1** 意图

主动对象设计模式使方法执行与方法调用去耦合，以增强并发、并简化对驻留在它自己的线程控制 中的对象的同步访问。

**6.2** 别名

并发对象和 Actor。

**6.3** 例子

为演示主动对象模式，考虑一个通信网关[1]的设计。网关使协作的组件去耦合，并允许它们进行交 互，而无需彼此直接依赖[2]。图 6-1 中所示的网关在分布式系统中将来自一或多个供应者进程的消息路 由到一或多个消费者进程[3]。

在我们的例子中，网关、供应者和消费者在面向连接的协议 TCP[4]之上进行通信。因此，当网关软 件尝试向远地消费者发送数据时，可能会遇到来自 TCP 传输层的流控制。TCP 使用流控制来确保快速的 生产者或网关不会过快地生产出数据，以致慢速消费者或拥挤的网络不能缓冲和处理这些数据。

为了改善所有供应者和消费者的端到端服务质量（QoS），整个网关进程不能在任何到消费者的连接

上阻塞以等待流控制缓解。此外，当供应者和消费者的数目增加时，网关还必须能高效地扩展， 防止阻塞并提高性能的一种有效的方法是在网关设计中引入并发。并发应用允许执行对象 O 的方法

的线程控制与调用 O 的方法的线程控制去耦合。而且，在网关中使用并发还使 TCP 连接被流控制的线程 的阻塞不会阻碍 TCP 连接未被流控制的线程的执行。

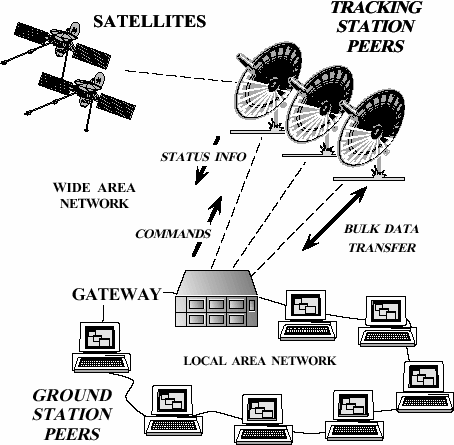


图 6-1 通信网关

**6.4** 上下文

对运行在相互分离的线程控制中的对象进行访问的客户。

**6.5** 问题

许多应用受益于使用并发对象来改善它们的 QoS，例如，通过允许应用并行地处理多个客户请求。 并发对象驻留在它们自己的线程控制中，而不是使用单线程被动对象——这些对象在调用其方法的客户

的线程控制中执行它们的方法。但是，如果对象并发执行，且这些对象被多个客户线程共享，我们必须 同步对它们的方法和数据的访问。在存在这样的问题时，会产生三种压力：

1. 对对象方法的并发调用不应阻塞整个进程，以免降低其他方法的 **QoS**：例如，如果在我们的网关例 子中，一个外出的到消费者的 TCP 连接因为流控制而阻塞，网关进程仍应该能在等待流控制缓解的 同时，排队新的消息。同样地，如果其他外出的 TCP 连接没有被流控制，它们应该能独立于任何阻 塞连接发送消息给它们的消费者。

2. 对共享对象的同步访问应该很简单：如果开发者必须显式地使用低级同步机制，比如像获取和释放互 斥锁（mutex），常常难于对网关这样的应用进行编程。一般而言，当对象被多个客户线程访问时，需 进行同步约束的方法应该被透明地序列化。

3. 应用应设计为能透明地利用硬件**/**软件平台上可用的并行机制：在我们的网关例子中，发往不同消费 者的消息应该被网关并行地在不同的 TCP 连接上发送。但是，如果整个网关被编写为仅在单个线程 控制中运行，性能瓶颈就不可能通过在多处理器上运行网关而被透明地克服。

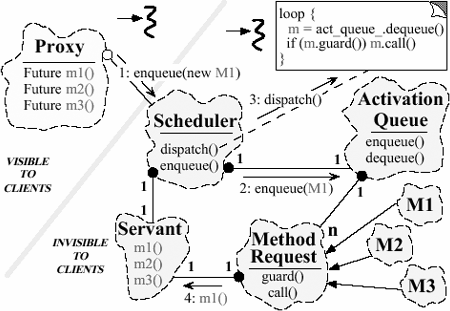
**6.6** 解决方案

对于每个需要并发执行的对象，使对对象方法的请求与方法执行去耦合。这样的去耦合被设计用于 使客户线程看起来像是调用一个平常的方法。该方法被自动转换为方法请求对象，并传递给另一个线程 控制，在其中它又被转换回方法，并在对象实现上被执行。

主动对象由以下组件组成：代理（Proxy）[5, 2]表示对象的接口，仆人（Servant）提供对象的实现。 代理和仆人运行在分离的线程中，以使方法调用和方法执行能并发运行：代理在客户线程中运行，而仆 人在不同的线程中运行。在运行时，代理将客户的方法调用（Method Invocation）转换为方法请求（Met hod Request），并由调度者（Scheduler）将其存储在启用队列（Activation Queue）中。调度者持续地运 行在与仆人相同的线程中，当启用队列中的方法请求变得可运行时，就将它们出队，并分派给实现主动 对象的仆人。客户可通过代理返回的“期货”（future）获取方法执行的结果。

**6.7** 结构

主动对象模式的结构在下面的 Booch 类图中演示：



在主动对象模式中有六个关键的参与者：

代理（**Proxy**）

z 代理提供一个接口，允许客户使用标准的强类型程序语言特性，而不是在线程间传递松散类型的消息， 来调用主动对象的可公共访问的方法。当客户调用代理定义的方法时，就会在调度者的启用队列上触 发方法请求对象的构造和排队；所有这些都发生在客户的线程控制中。

方法请求（**Method Request**）

z 方法请求用于将代理上的特定方法调用的上下文信息，比如方法参数和代码，从代理传递给运行在分 离线程中的调度者。抽象方法请求类为执行主动对象方法定义接口。该接口还包含守卫（guard）方 法，可用于确定何时方法请求的同步约束已被满足。对于代理提供的每个主动对象方法（它们在其仆 人中需要同步的访问），抽象方法请求类被子类化，以创建具体的方法请求类。这些类的实例在其方 法被调用时由代理创建，并包含了执行这些方法调用和返回任何结果给客户所需的特定的上下文信 息。

启用队列（**Activation Queue**）

z 启用队列维护一个有界缓冲区，内有代理创建的待处理的方法请求。该队列跟踪哪些方法请求将要执 行。它还使客户线程与仆人线程去耦合，以使两个线程能并发运行。

调度者（**Scheduler**）

z 调度者运行在与其客户不同的线程中，它管理待处理的方法请求的启用队列。调度者决定下一个出队 的方法请求，并在实现该方法的仆人上执行。这样的调度决策基于各种标准，比如像方法被插入到启 用队列中的顺序；以及同步约束，例如特定属性的满足或特定事件的发生，比如在有界数据结构中有 新的条目空间变得可用。调度者通常使用方法请求守卫来对同步约束进行求值。

仆人（**Servant**）

z 仆人定义被建模为主动对象的行为和状态。它实现在代理中定义的方法及相应的方法请求。仆人方法 在调度者执行其相应的方法请求时被调用；因而，仆人在调度者的线程控制中执行。仆人还可提供其 他方法，由方法请求用于实现它们的守卫。

期货（**Future**）

z 期货[7, 8]允许客户在仆人结束方法的执行后获取方法调用的结果。当客户通过代理调用方法时，期 货被立即返回给客户。期货为被调用的方法保留空间，以存储它的结果。当客户想要获取这些结果时， 它可以阻塞或者轮询，直到结果被求值和存储到期货中，然后与期货“会合”。

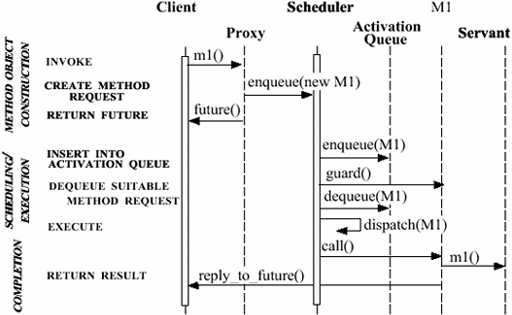
**6.8** 动力特性

下图演示主动对象模式中的协作的三个阶段：

**1.** 方法请求构造和调度：在此阶段，客户调用代理上的方法，从而触发方法请求的创建；方法请求维护 方法的参数绑定，以及其他任何执行方法和返回结果所需的绑定。代理随后将方法请求传递给调度者， 后者将其放入启用队列中。如果方法被定义为“两路”（two way）[6]的，一个期货的绑定被返回给 调用该方法的客户。如果方法被定义为“单路”（oneway）的，就没有期货被返回，也就是，它没有 返回值。

**2.** 方法执行：在此阶段，调度者在与其客户不同的线程中持续运行。在此线程中，调度者监控启用队列， 并确定哪些方法请求已成为可运行的，例如，当它们的同步约束已被满足时。当方法请求成为可运行 的，调度者就使其出队，绑定到仆人，并分派仆人上的适当方法。当此方法被调用时，它可以访问/ 更新它的仆人的状态并创建它的结果。

**3.** 完成：在最后的阶段中，结果（如果有的话）被存储在期货中，而调度者持续地监控启用队列中，看 是否有可运行的方法请求。在一个两路方法完成后，客户可以通过与期货会合来获取它的结果。一般 而言，任何与期货会合的客户都可以获取它的结果。当方法请求和期货不再被引用时，它们就被删除 或被垃圾回收。



**6.9** 实现

这一部分解释使用主动对象模式构建并发应用所涉及的步骤。使用主动对象模式实现的应用是 6.3 网关的一部分。图 6-2 演示该例子的结构和参与者。这一部分中的例子使用 ACE 框架[9]的可复用组件。 ACE 提供了一组丰富的可复用 C++包装和框架组件，可跨越广泛的 OS 平台执行常见的通信软件任务。

**1.** 实现仆人：仆人定义被建模为主动对象的行为和状态。客户可通过代理来访问仆人所实现的方法。此 外，仆人还可包含其他方法，方法请求可以用这些方法来实现守卫，以允许调度者对运行时同步约束 进行求值。这些约束决定调度者分派方法请求的顺序。

在我们的网关例子中，仆人是一个消息队列，缓冲待处理的递送给消费者的消息。对于每一个远 地消费者，都有一个Consumer Handler（消费者处理器），其中含有一个到消费者进程的TCP连接。 此外，Consumer Handler含有一个被建模为主动对象的消息队列，并通过MQ\_Servant来实现。当从供 应者传递到网关的消息在等待被发送到它们的远地消费者时，每个Consumer Handler的主动对象消息 队列就存储这些消息。下面的类提供了这个仆人的接口：

class MQ\_Servant

{

public:

MQ\_Servant (size\_t mq\_size);

// Message queue implementation operations. void put\_i (const Message &msg);

Message get\_i (void);

// Predicates.

bool empty\_i (void) const;

bool full\_i (void) const;

private:

// Internal Queue representation, e.g., a

// circular array or a linked list, etc.

};

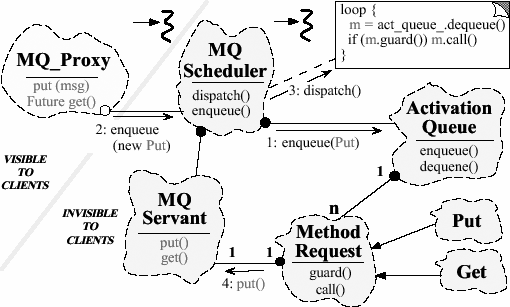


图 6-2 将消费者处理器的消息队列实现为主动对象

put\_i 和 get\_i 方法分别实现队列的插入和移除操作。此外，仆人还定义了两个断言（Predicate）： empty\_i 和 full\_i，可区分三种内部状态（1）空，（2）满，以及（3）既不为空也不为满。这些断言 用于方法请求的看守方法的实现，后者允许调度者强制实施运行时同步约束；这些同步约束规定仆 人的 put\_i 和 get\_i 的调用顺序。

注意 MQ\_Servant 类是怎样设计，以使同步机制始终外在于仆人。例如，在我们的网关例子中， MQ\_Servant 类中的方法并不包括任何实现同步的代码。该类仅仅提供实现仆人功能和检查它的内部 状态的方法。这样的设计避免了“继承异常”[10, 11, 12, 13]问题；如果子类需要不同的同步策略， 该问题将会妨碍仆人实现的复用。因而，对主动对象的同步约束的改变不需要影响它的仆人实现。

**2.** 实现代理和方法请求：代理为客户提供仆人方法的接口。对于客户的每一次方法调用，代理都会创建 一个方法请求。方法请求是方法上下文21的抽象。该上下文通常包括方法参数、到将要应用此方法的 仆人的绑定、结果期货，以及方法请求的 call 方法的代码。

在我们的网关例子中，MQ\_Proxy 提供步骤 1 中定义的 MQ\_Servant 的抽象接口。该消息队列被 C onsumer Handler 用于排队递送给消费者的消息，如图 6-2 所示。此外，MQ\_Proxy 还是一个工厂，它 构造方法请求的实例，并将它们传递给调度者，后者将它们排队，用于后面在分离的线程中的执行。 MQ\_Proxy 的 C++实现如下所示：

21 此上下文常被称为“罩子”（closure）。

class MQ\_Proxy

{

public:

// Bound the message queue size. enum { MAX\_SIZE = 100 };

MQ\_Proxy (size\_t size = MAX\_SIZE)

: scheduler\_ (new MQ\_Scheduler (size)), servant\_ (new MQ\_Servant (size)) {}

// Schedule <put> to execute on the active object. void put (const Message &m)

{

Method\_Request \*method\_request = new Put (servant\_, m);

scheduler\_->enqueue (method\_request);

}

// Return a Message\_Future as the ‘‘future’’

// result of an asynchronous <get>

// method on the active object. Message\_Future get (void)

{

Message\_Future result;

Method\_Request \*method\_request = new Get (servant\_, result);

scheduler\_->enqueue (method\_request);

return result;

}

// ... empty() and full() predicate implementations ...

protected:

// The Servant that implements the

// Active Object methods. MQ\_Servant \*servant\_;

// A scheduler for the Message Queue. MQ\_Scheduler \*scheduler\_;

};

MQ\_Proxy的每个方法都将它的调用转换为方法请求，并将其传递给它的MQ\_Scheduler，后者

将请求入队，用于后续的启用。Method\_Request基类定义虚guard和call方法，分别被它的调度者用于 决定方法请求是否可被执行和在它的仆人上执行方法请求。如下所示：

class Method\_Request

{

public:

// Evaluate the synchronization constraint. virtual bool guard (void) const = 0;

// Implement the method. virtual void call (void) = 0;

};

该类中的方法必须被子类定义，代理中定义的每个方法都有一个相应的Method\_Request子类。定

义这两个方法的原因是为调度者提供一个统一接口来计算和执行具体Method\_Request。因而，调度者 就得以与怎样计算同步约束、或是触发具体Method\_Request执行的特定知识去耦合。

例如，在我们的网关例子中，当客户调用代理上的put方法时，该方法被转换为Put子类的实例； 该子类继承自Method\_Request，并含有指向MQ\_Servant的指针。如下所示：

class Put : public Method\_Request

{

public:

Put (MQ\_Servant \*rep, Message arg)

: servant\_ (rep), arg\_ (arg) {}

virtual bool guard (void) const

{

// Synchronization constraint: only allow

// <put\_i> calls when the queue is not full. return !servant\_->full\_i ();

}

virtual void call (void)

{

}

private:

// Insert message into the servant. servant\_->put\_i (arg\_);

MQ\_Servant \*servant\_; Message arg\_;

};

注意 guard 方法怎样使用 MQ\_Servant 的 full\_I 断言来实现同步约束，以允许调度者确定 Put 方法

请求何时可以执行。当 Put 方法请求可被执行时，调度者调用它的 call 挂钩方法。该方法使用它的 M Q\_Servant 运行时绑定来调用仆人的 put\_i 方法。put\_i 方法在仆人的上下文中执行，并且不需要任何 显式的序列化机制，因为调度者通过方法请求 guard 来强制实施所有必要的同步约束。

代理还将 get 方法转换为 Get 类的实例；Get 类定义如下：

class Get : public Method\_Request

{

public:

Get (MQ\_Servant \*rep, const Message\_Future &f)

: servant\_ (rep), result\_ (f) {}

bool guard (void) const

{

// Synchronization constraint:

// cannot call a <get\_i> method until

// the queue is not empty. return !servant\_->empty\_i ();

}

virtual void call (void)

{

// Bind the dequeued message to the

// future result object. result\_ = servant\_->get\_i ();

}

private:

MQ\_Servant \*servant\_;

// Message\_Future result value. Message\_Future result\_;

};

对于代理中所有返回值的两路方法，比如在我们的网关例子中的 get\_i 方法，Message\_Future 被返

回给调用该方法的客户线程，如下面的实现步骤 4 所示。客户可以选择立即对 Message\_Future 的值进 行求值，在这样的情况下，客户会阻塞、直到方法请求被调度者执行为止。相反，对主动对象方法调 用的返回结果的求值也可被延期，在这样的情况下，客户线程和执行该方法的线程可以异步地执行。

**3.** 实现启用队列：每个方法请求都被放入启用队列中。启用队列通常实现为线程安全的有界缓冲区，由 客户线程与调度者及仆人的线程共享。启用队列还提供一个迭代器，允许调度者能依照迭代器模式[5] 遍历它的元素。

下面的 C++代码演示 Activation\_Queue 是怎样被用于网关中的：

class Activation\_Queue

{

public:

// Block for an "infinite" amount of time

// waiting for <enqueue> and <dequeue> methods

// to complete.

const int INFINITE = -1;

// Define a "trait".

typedef Activation\_Queue\_Iterator iterator;

// Constructor creates the queue with the

// specified high water mark that determines

// its capacity.

Activation\_Queue (size\_t high\_water\_mark);

// Insert <method\_request> into the queue, waiting

// up to <msec\_timeout> amount of time for space

// to become available in the queue.

void enqueue (Method\_Request \*method\_request, long msec\_timeout = INFINITE);

// Remove <method\_request> from the queue, waiting

// up to <msec\_timeout> amount of time for a

// <method\_request> to appear in the queue. void dequeue (Method\_Request \*method\_request,

long msec\_timeout = INFINITE);

private:

// Synchronization mechanisms, e.g., condition

// variables and mutexes, and the queue

// implementation, e.g., an array or a linked

// list, go here.

// ...

};

enqueue 和 dequeue 方法提供一种“有界缓冲区生产者/消费者”并发模式，允许多个线程同时插入和

移除 Method\_Request，而不会破坏 Activation\_Queue 的内部状态。一或多个客户线程扮演生产者角色， 通过代理将 Method\_Request 入队。调度者线程扮演消费者角色，当 Method\_Request 的 guard 方法求 值为“真”时，将它们出队，并调用它们的 call 挂钩来执行仆人方法。

Activation\_Queue 被设计为使用条件变量和互斥体[14]的有界缓冲区。因此，当试图从空的 Activ ation\_Queue 中移除 Method\_Request 时，调度者将会阻塞 msec\_timeout 长度的时间。同样地，当试图 在满的 Activation\_Queue 中（也就是，当前 Method\_Request 数目等于其高水位标的队列）插入时， 客户线程将会阻塞最多 msec\_timeout 长度的时间。如果 enqueue 方法超时了，控制会返回给客户线程， 而方法没有被执行。

**4.** 实现调度者：调度者维护启用队列，并执行同步约束已满足的待处理方法请求。调度者的公共接口通 常为代理提供一个方法，用于将方法请求放入启用队列中；还有另一个方法，用于在仆人上分派方法 请求。这些方法运行在分离的线程中，也就是，代理运行在与调度者和仆人不同的线程中，而后两者

运行在同一线程中。

在我们的网关例子中，我们定义 MQ\_Scheduler 类如下：

class MQ\_Scheduler

{

public:

// Initialize the Activation\_Queue to have the

// specified capacity and make the Scheduler

// run in its own thread of control. MQ\_Scheduler (size\_t high\_water\_mark);

// ... Other constructors/destructors, etc.,

// Insert the Method Request into

// the Activation\_Queue. This method

// runs in the thread of its client, i.e.,

// in the Proxy’s thread.

void enqueue (Method\_Request \*method\_request)

{

act\_queue\_->enqueue (method\_request);

}

// Dispatch the Method Requests on their Servant

// in the Scheduler’s thread. virtual void dispatch (void);

protected:

// Queue of pending Method\_Requests. Activation\_Queue \*act\_queue\_;

// Entry point into the new thread. static void \*svc\_run (void \*arg);

};

调度者在与它的客户线程不同的线程控制中执行它的 dispatch 方法。这些客户线程驱使代理将方

法请求放入调度者的 Activation\_Queue 中。调度者在它自己的线程中监控它的 Activation\_Queue，选 择其 guard 求值所得为“真”（也就是，同步约束已被满足）的 Method\_Request。于是这个 Method\_ Request 就通过调用它的 call 挂钩方法被执行。注意多个客户线程可以共享同一个代理。代理方法无 需是线程安全的，因为调度者和启用队列会处理并发控制。

例如，在我们的网关例子中，MQ\_Scheduler 的构造器初始化 Activation\_Queue，并派生一个新线 程来运行 MQ\_Scheduler 的 dispatch 方法。如下所示：

MQ\_Scheduler (size\_t high\_water\_mark)

: act\_queue\_ (new Activation\_Queue (high\_water\_mark))

{

// Spawn a separate thread to dispatch

// method requests.

Thread\_Manager::instance ()->spawn (svc\_run, this);

}

这个新线程执行 svc\_run 静态方法，后者仅仅是一个调用 dispatch 方法的适配器。如下所示：

void \*MQ\_Scheduler::svc\_run (void \*args)

{

MQ\_Scheduler \*this\_obj = reinterpret\_cast<MQ\_Scheduler \*> (args);

this\_obj->dispatch ();

}

dispatch 方法基于底层的 MQ\_Servant 断言 empty\_i 和 full\_i 来决定 Put 和 Get 方法请求的处理顺序。

这些断言反映仆人的状态，比如消息队列是否为空、满，或都不是。通过经由方法请求的 guard 方法 对这些断言约束进行求值，调度者可以确保对 MQ\_Servant 的公平的共享访问。如下所示：

virtual void MQ\_Scheduler::dispatch (void)

{

// Iterate continuously in a

// separate thread. for (;;)

{

Activation\_Queue::iterator i;

// The iterator’s <begin> call blocks

// when the <Activation\_Queue> is empty.

for (i = act\_queue\_->begin (); i != act\_queue\_->end (); i++)

{

// Select a Method Request ‘mr’

// whose guard evaluates to true. Method\_Request \*mr = \*i;

if (mr->guard ())

{

// Remove <mr> from the queue first

// in case <call> throws an exception. act\_queue\_->dequeue (mr);

mr->call ();

delete mr;

}

}

}

}

在我们的网关例子中，MQ\_Scheduler 类的 dispatch 的实现持续地执行下一个其 guard 求值为真的 Met

hod\_Request。但是，调度者实现还可以更为成熟，并且可以含有表示仆人同步状态的变量。例如， 要实现一个多读者/单作者的同步策略，可在调度者中存储若干计数器变量，以跟踪读和写请求的数 目。调度者可使用这些计数器来确定一个单个的作者何时可以继续执行，也就是，当目前的读者数目 为 0，而目前又没有其他作者在运行时。注意计数器的值独立于仆人的状态，因为后者仅仅被调度者 用来代表仆人强制实施正确的同步策略。

**5.** 决定会合和返回值策略：会合策略决定客户怎样从在主动对象上调用的方法那里获取返回值。之所以 需要会合策略，是因为主动对象仆人并不在调用它们的方法的客户的线程里执行。主动对象的实现通 常从以下的会合和返回值策略中进行选择：

1. 同步的等待：在代理中同步地阻塞客户线程，直到方法请求被调度者分派、结果被计算并存储在 期货中。

2. 同步的超时：仅阻塞有限数量的时间，如果在指定的时间段里两路调用的结果没有返回，方法调 用就会失败。如果 timeout 为零，客户线程就进行“轮询”，也就是，如果调度者不能立即分派 方法请求，客户线程就返回调用者、而不使方法请求入队。

3. 异步的：排队方法请求，将控制立即返回给客户线程。如果该方法是要产生结果的两路调用，就 必须使用某种形式的期货机制，以提供对结果值的同步的访问（或是错误状态，如果方法调用失 败的话）。

期货构造允许进行两路异步调用，它们返回值给客户。当仆人完成方法执行时，它获取期货上的 写锁，用结果值更新期货。任何正阻塞等待该结果值的线程都被唤醒，并可以并发地访问结果值。期 货对象可在作者和所有读者都不再引用它后被垃圾回收。在像 C++这样不直接支持垃圾回收的语言 里，期货对象可以在它们不再被使用时通过像计数器指针[2]这样的习语来进行回收。

在我们的网关例子中，在 MQ\_Proxy 上调用 get 方法最终导致 Get::call 方法被 MQ\_Scheduler 分派， 如上面的步骤 2 所示。因为 MQ\_Proxy get 方法返回一个值，当客户调用它时会返回一个 Message\_F uture。Message\_Future 被定义如下：

class Message\_Future

{

public:

// Copy constructor binds <this> and <f> to the

// same <Message\_Future\_Rep>, which is created if

// necessary.

Message\_Future (const Message\_Future &f);

// Constructor that initializes <Message\_Future> to

// point to <Message> <m> immediately. Message\_Future (const Message &m);

// Assignment operator that binds <this> and <f>

// to the same <Message\_Future\_Rep>, which is

// created if necessary.

void operator= (const Message\_Future &f);

// ... other constructors/destructors, etc.,

// Type conversion, which blocks

// waiting to obtain the result of the

// asynchronous method invocation. operator Message ();

};

Message\_Future 使用计数指针习语[2]来实现。通过使用引用计数的 Message\_Future\_Rep 体（通过 Me ssage\_Future 句柄单独进行访问），这种习语简化了动态分配的 C++对象的内存管理。

一般而言，客户可通过下面两种方法中的一种来从 Message\_Future 对象那里获取消息的结果值：

z 立即求值：客户可以选择立即对 Message\_Future 进行求值。例如，运行在分离线程中的网关的 C

onsumer Handler 可以选择进行阻塞，直到来自供应者的新消息到达。如下所示：

MQ\_Proxy mq;

// ...

// Conversion of Message\_Future from the

// get() method into a Message causes the

// thread to block until a message is

// available.

Message msg = mq.get ();

// Transmit message to the consumer. send (msg);

z 延期求值：对主动对象的方法请求的返回值的求值可以被延期。例如，如果消息不是立即可用， Consumer Handler 可存储来自 mq 的 Message\_Future 返回值，并执行其他的“簿记”任务，比如 交换 *keep-alive* 消息，以确保它的消费者仍然是活动的。当 Consumer Handler 完成这些任务后， 它可以进行阻塞，直到来自供应者的消息到达。如下所示：

// Obtain a future (does not block the client). Message\_Future future = mq.get ();

// Do something else here...

// Evaluate future in the conversion operator;

// may block if the result is not available yet. Message msg = Message (future);

**6.10** 例子解答

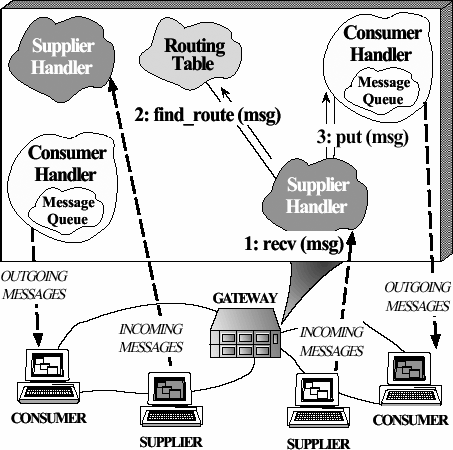


图 6-3 通信网关

网关软件在内部包含有 Supplier 和 Consumer Handler，分别用作远地供应者和消费者的本地代理[2,

5]。如图 6-3 所示，Supplier Handler 接收来自远地供应者的消息，检查消息中的地址域，并将该地址用 作 Routing Table 中的关键字；该关键字标识哪一个远地消费者应该接收该消息。Routing Table 维护 Con sumer Handler 的一个映射表；每一个 Consumer Handler 负责在各自的 TCP 连接上递送消息给它的远地 消费者。

为在多个 TCP 连接上处理流控制，每个 Consumer Handler 都含有一个使用 6.9 中描述的主动对象来 实现的消息队列。Consumer\_Handler 类被定义如下：

class Consumer\_Handler

{

public:

Consumer\_Handler (void);

// Put the message into the queue. void put (const Message &msg)

{

message\_queue\_.put (msg);

}

private:

// Proxy to the Active Object. MQ\_Proxy message\_queue\_;

// Connection to the remote consumer. SOCK\_Stream connection\_;

// Entry point into the new thread. static void \*svc\_run (void \*arg);

};

在它们自己的线程中运行的 Supplier Handler 将消息放进适当的 Consumer Handler 的消息队列中。

如下所示：

Supplier\_Handler::route\_message (const Message &msg)

{

// Locate the appropriate consumer based on the

// address information in the Message.

Consumer\_Handler \*ch = routing\_table\_.find (msg.address ());

// Put the Message into the Consumer Handler’s queue. ch->put (msg);

};

为处理放置在消息队列中的消息，每个 Consumer\_Handler 在它的构造器中都派生出一个单独的线程

控制。如下所示：

Consumer\_Handler::Consumer\_Handler (void)

{

// Spawn a separate thread to get messages

// from the message queue and send them to

// the consumer.

Thread\_Manager::instance ()->spawn (svc\_run, this);

}

这个新线程执行 svc\_run 方法，取得由 Supplier Handler 线程放置在队列中的消息，并在 TCP 连接上将它

们发送给消费者。如下所示：

void \*Consumer\_Handler::svc\_run (void \*args)

{

Consumer\_Handler \*this\_obj =

reinterpret\_cast<Consumer\_Handler \*> (args);

for (;;)

{

// Conversion of Message\_Future from the

// get() method into a Message causes the

// thread to block until a message is

// available.

Message msg = this\_obj->message\_queue\_.get ();

// Transmit message to the consumer. this\_obj->connection\_.send (msg);

}

}

因为消息队列被实现成主动对象，send 操作可以在任何给定的 Consumer\_Handler 对象中阻塞，而不会影

响其他 Consumer\_Handler 的服务质量。

**6.11** 变种

下面是主动对象模式的一些变种： 集成调度者：为减少实现主动对象模式所需组件的数目，代理和仆人的角色常常被集成进调度者组件中，

尽管仆人仍然在与代理不同的线程中执行。而且，将方法调用转换为方法请求也可被集成进调度者。例 如，下面是使用集成调度者来实现消息队列的另一种方法：

class MQ\_Scheduler

{

public:

MQ\_Scheduler (size\_t size)

: act\_queue\_ (new Activation\_Queue (size))

{}

// ... other constructors/destructors, etc., void put (const Message &msg)

{

Method\_Request \*method\_request =

// The <MQ\_Scheduler> is the servant. new Put (this, msg);

act\_queue\_->enqueue (method\_request);

}

Message\_Future get (void)

{

Message\_Future result;

Method\_Request \*method\_request =

// The <MQ\_Scheduler> is the servant. new Get (this, result);

act\_queue\_->enqueue (method\_request);

}

// ...

return result;

private:

// Message queue servant operations. void put\_i (const Message &msg); Message get\_i (void);

// Predicates.

bool empty\_i (void) const;

bool full\_i (void) const;

Activation\_Queue \*act\_queue\_;

// ...

};

通过使生成方法请求的地方集中化，可以简化模式的实现，因为组件变少了。当然，缺点是调度者必须

知道仆人和代理的类型，这使得开发者难以将调度者复用于不同类型的主动对象。

消息传递：一种更为精致的集成调度者变种是将代理和仆人一起去除，并在客户线程和调度者线程间使 用直接的消息传递。如下所示：

class Scheduler

{

public:

Scheduler (size\_t size)

: act\_queue\_ (new Activation\_Queue (size))

{}

// ... other constructors/destructors, etc.,

// Enqueue a Message Request in the thread of

// the client.

void enqueue (Message\_Request \*message\_request)

{

act\_queue\_->enqueue (message\_request);

}

// Dispatch Message Requests in the thread of

// the Scheduler.

virtual void dispatch (void)

{

Message\_Request \*mr;

// Block waiting for next request to arrive. while (act\_queue\_->dequeue (mr))

{

// Process the message request <mr>.

}

}

protected:

Activation\_Queue \*act\_queue\_;

// ...

};

在此设计中没有代理，于是客户简单地创建适当类型的 Message\_Request，并调用 enqueue，将请求插入 到 Activation\_Queue。同样地，也没有仆人，于是运行在调度者线程中的 dispatch 方法简单地使下一个 M essage\_Request 出队，并根据它的类型来进行处理。

一般而言，开发一种消息传递机制要比开发主动对象容易，因为需要开发的组件较少。但是，消息 传递通常更为麻烦而易错，因为应用开发者要负责对代理和仆人逻辑进行编程，而不是让主动对象开发 者来编写此代码。

多态期货：多态期货[15]允许对期货所代表的最终结果类型进行参数化，并会采取必要的同步。特别地， 多态期货结果值提供一次写、多次读的同步。客户是否会阻塞在期货上取决于结果值是否已被计算。因 此，多态期货部分地是一种读者/作者条件同步模式，部分地是生产者/消费者同步模式。

下面的类演示一种 C++多态期货模板：

template <class T>

class Future

{

// This class implements a ‘single write, multiple

// read’ pattern that can be used to return results

// from asynchronous method invocations.

public:

// Constructor. Future (void);

// Copy constructor that binds <this> and <r> to

// the same <Future> representation

Future (const Future<T> &r);

// Destructor.

˜Future (void);

// Assignment operator that binds <this> and

// <r> to the same <Future>.

void operator = (const Future<T> &r);

// Cancel a <Future>. Put the future into its

// initial state. Returns 0 on success and -1

// on failure.

int cancel (void);

// Type conversion, which obtains the result

// of the asynchronous method invocation.

// Will block forever until the result is

// obtained. operator T ();

// Check if the result is available. int ready (void);

private:

Future\_Rep<T> \*future\_rep\_;

// Future representation implemented using

// the Counted Pointer idiom.

};

客户可以这样来使用多态期货：

// Obtain a future (does not block the client). Future<Message> future = mq.get ();

// Do something else here...

// Evaluate future in the conversion operator;

// may block if the result is not available yet.

Message msg = Message (future);

分布式主动对象：在此变种中，在代理和调度者之间存在着分布式的边界，而不是传统的主动对象模式 中的线程边界。因此，客户代理扮演 *stub* 的角色，负责将方法参数整编（marshal）为方法请求的参数格 式；后者可跨越网络进行传输，并由在单独的地址空间中的仆人执行之。此外，该变种通常还引入服务 器端的 *skeleton* 概念，在将方法请求的参数传递给服务器中的仆人方法之前对它们进行去整编。

线程池：线程池是一种泛化的主动对象，支持每个主动对象有多个仆人。这些仆人可以提供相同的服务， 以提高吞吐量和响应能力。所有仆人都运行在自己的线程中，并在为当前作业作好准备时主动要求调度 者分配新的请求。于是一旦有新的作业可做，调度者就会分配一个新作业。

**6.12** 已知应用

下面是已知的主动对象模式的特定使用：

**CORBA ORB**：主动对象模式已被用于实现并发 ORB 中间件框架，比如 CORBA[6]和 DCOM[16]。例如， TAO ORB[17]为它的缺省并发模型[18]实现了主动对象模式。在此设计中，CORBA stub 对应主动对象模 式的代理，前者将远地的操作调用转换为 CORBA 请求。TAO ORB 核心的 Reactor（反应器）是调度者， 而 ORB 核心中的 socket 队列对应着启用队列。开发者创建仆人，它在服务器的上下文中执行方法。客户 可以进行同步的两路调用，它们会阻塞调用线程，直到操作返回；或是进行异步的方法调用，它们会返 回一个 Poller 期货对象，可随后进行求值[19]。

**ACE** 框架：ACE 框架[9]提供了主动对象模式中的 Method Request、Activation Queue 和 Future 组件的可 复用实现。这些组件已被用于实现许多分布式系统产品。

**Siemens MedCom**：主动对象模式还被用于 Siemens MedCom 框架，它为电子医学系统[20]提供了一种面 向组件的黑盒子框架。MedCom 采用主动对象模式和命令处理器模式（Command Processor Pattern）来 简化在多个医学服务器上访问病人信息的客户窗口应用。

西门子呼叫中心管理系统：该系统使用主动对象模式的线程池变种。

**Actor**（行动者）：主动对象模式已被用于实现 Actor[21]。一个 Actor 含有一组实例变量和 behavior（行为）， 响应其他 Actor 发送的消息。发送给 Actor 的消息被放置在它的消息队列中。在 Actor 模型里，消息以到 达的顺序被“当前” behavior 执行。每个 behavior 提名一个替换 behavior 来执行下一个消息，该执行有 可能是发生在进行提名的 behavior 完成执行之前。基本的 Actor 模型的变种允许消息队列中的消息基于不 同于到达顺序的其他标准来被执行[22]。当主动对象模式用于实现 Actor 时，调度者对应于 Actor 调度机 制，方法请求对应于为 Actor 定义的 behavior，而仆人是共同表示 Actor 状态的一组实例变量[23]。代理 仅仅是一种强类型的用于传递消息给 Actor 的机制。

**6.13** 效果

主动对象模式提供以下好处： 增强应用并发性并简化同步复杂性：并发性是通过允许客户线程和异步方法同时运行来增强的。同步复

杂性则通过调度者来简化；后者对同步约束进行求值，以依照它们的状态来保证对仆人的序列化访问。

透明而有效地利用可用的并发：如果硬件和软件平台能高效地支持多 CPU，该模式可以允许多个主动对 象依据它们的同步约束并行地执行。

方法执行序可与方法调用序不同：异步调用的方法基于它们的同步约束来执行，其顺序可与它们被调用 的顺序不同。

但是，主动对象模式也有以下缺点： 性能开销：取决于调度者怎样实现（例如，在用户空间还是内核空间），当调度和执行主动对象方法调用

时，可能会产生上下文切换、同步和数据移动等开销。一般而言，主动对象最适用于粒度相对较粗的对

象。相反，如果对象的粒度非常细，与其他像管程（Monitor）这样的并发模式相比，主动对象的性能开 销可能会很过分。

复杂的调试：由于调度者的并发性和非决定论，有可能会难以调试含有主动对象的程序。而且，许多调 试器都不能充分地支持并发应用。

**6.14** 参见

管程模式确保某一时刻在被动对象中只有一个方法在执行，而不管并发调用此对象的方法的数目有 多少。管程通常比主动对象要更为高效，因为它们带来的上下文切换和数据移动的开销较少。但是，使 用管程模式，在客户和服务器之间增加分布式边界要更难。

反应器模式[24]负责处理多个事件处理器的多路分离和分派；处理器在可以开始操作而不会阻塞时被 触发。该模式常常用于替代主动对象模式，以调度对被动对象的回调操作。它还可以与主动对象模式结 合使用，以构成在下一段中描述的半－同步/半－异步（Half-Sync/Half-Async）模式。

半－同步／半异步模式[25]是一种体系结构模式，用于使系统中的同步 I/O 和异步 I/O 去耦合，以简 化并发编程工作，而又不影响执行效率。该模式通常使用主动对象模式来实现同步任务层，使用反应器 模式[24]来实现异步任务层，以及使用生产者/消费者模式来实现排队层。

命令处理器（Command Processor）模式[2]与主动对象模式类似。它的意图是分离请求的发出与执行。 命令处理器对应于调度者，它维护待处理的服务请求，后者被实现为命令（Command）[5]。它们在对应 于仆人的供应者那里执行。但是，命令处理器模式并不特别关心并发，而且客户、命令处理器和供应者 驻留在同一线程控制中。因而，没有代理来向客户代表仆人。客户创建命令，并将它们直接传递给命令 处理器。

Broker 模式[2]有着许多与主动对象模式一样的组件。特别地，客户通过代理来访问 Broker，而服务

器通过仆人来实现远地对象。Broker 模式和主动对象模式之间的主要差异是：在 Broker 模式中的代理和 仆人之间是分布式的边界，而在主动对象模式中的代理和仆人之间是线程边界。

互斥模式[26]是一种简单的锁定机制，常常用于在实现并发被动对象（也称为管程）时代替主动对象。 互斥模式可以略微不同的形式出现，比如回旋锁或信号量。互斥模式可有多种语义，比如递归互斥体和 优先级继承互斥体。

感谢

主动对象模式由 Greg Lavender 首创。感谢 Frank Buschmann、 Hans Rohnert、 Martin Botzler、 Michael Stal、 Christa Schwanninger 和 Greg Gallant 提出大量意见，极大地改进了这一版本的模式描述 的形式和内容。

参考文献

[1] D. C. Schmidt, “Acceptor and Connector: Design Patterns for Initializing Communication Services,” in *Pattern Langu ages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A Syst em of Pattern*s. Wiley and Sons, 1996.

[3] D. C. Schmidt, “A Family of Design Patterns for Application-level Gateways,” *The Theory and Practice of Object S*

*ystems (Special Issue on Patterns and Pattern Languages*),vol.2, no. 1, 1996.

[4] W. R. Stevens, *TCP/IP Illustrated, Volume* 1. Reading, Massachusetts: Addison Wesley, 1993.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Softwar*e. Reading, MA: Addison-Wesley, 1995.

[6] Object Management Group, *The Common Object Request Broker: Architecture and Specificatio*n, 2.2 ed., Feb. 1998. [7] R. H. Halstead, Jr., “Multilisp: A Language for Concurrent Symbolic Computation,” *ACM Trans. Programming Langu ages and System*s, vol. 7, pp. 501–538, Oct. 1985.

[8] B. Liskov and L. Shrira, “Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Sys tems,” in *Proceedings of the SIGPLAN’88 Conference on Programming Language Design and Implementatio*n, pp. 260–2

67, June 1988.

[9] D. C. Schmidt, “ACE: an Object-Oriented Framework for Developing Distributed Applications,” in *Proceedings of the*

*6th USENIX C++ Technical Conferenc*e, (Cambridge, Massachusetts), USENIX Association, April 1994.

[10] P. America, “Inheritance and Subtyping in a Parallel Object-Oriented Language,” in *ECOOP’87 Conference Proceedi ng*s, pp. 234–242, Springer-Verlag, 1987.

[11] D. G. Kafura and K. H. Lee, “Inheritance in Actor-Based Concurrent Object-Oriented Languages,” in *ECOOP’89 C*

*onference Proceeding*s, pp. 131–145, Cambridge University Press, 1989.

[12] S. Matsuoka, K. Wakita, and A. Yonezawa, “Analysis of Inheritance Anomaly in Concurrent Object-Oriented Langu ages,” *OOPS Messenge*r, 1991.

[13] M. Papathomas, “Concurrency Issues in Object-Oriented Languages,” in *Object Oriented Development* (D. Tsichritzis, ed.), pp. 207–245, Centre Universitaire D’Informatique, University of Geneva, 1989.

[14] W. R. Stevens, *UNIX Network Programming, Second Editio*n. Englewood Cliffs, NJ: Prentice Hall, 1997.

[15] R. G. Lavender and D. G. Kafura, “A Polymorphic Future and First-Class Function Type for Concurrent Object-Ori ented Programming in C++,” in *Forthcomin*g, [1995. http://www.cs.utexas.edu/users/lavender/papers/futures.ps.](http://www.cs.utexas.edu/users/lavender/papers/futures.ps)

[16] D. Box, *Essential CO*M. Addison-Wesley, Reading, MA, 1997.

[17] D. C. Schmidt, D. L. Levine, and S. Mungee, “The Design and Performance of Real-Time Object Request Broker s,” *Computer Communication*s, vol. 21, pp. 294–324, Apr. 1998.

[18] D. C. Schmidt, “Evaluating Architectures for Multi-threaded CORBA Object Request Brokers,” *Communications of t he ACM special issue on CORB*A, vol. 41, Oct. 1998.

[19] Object Management Group, *CORBA Messaging Specificatio*n, OMG Document orbos/98-05-05 ed., May 1998.

[20] P. Jain, S. Widoff, and D. C. Schmidt, “The Design and Performance of MedJava – Experience Developing Perfor mance-Sensitive Distributed Applications with Java,” *IEE/BCS Distributed Systems Engineering Journa*l, 1998.

[21] G. Agha, *A Model of Concurrent Computation in Distributed System*s. MIT Press, 1986.

[22] C. Tomlinson and V. Singh, “Inheritance and Synchronization with Enabled-Sets,” in *OOPSLA’89 Conference Procee ding*s, pp. 103–112, Oct. 1989.

[23] D. Kafura, M. Mukherji, and G. Lavender, “ACT++: A Class Library for Concurrent Programming in C++ using A

ctors,” *Journal of Object-Oriented Programmin*g, pp. 47–56, October 1992.

[24] D. C. Schmidt, “Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dis patching,” in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, M A: Addison-Wesley, 1995.

[25] D. C. Schmidt and C. D. Cranor, “Half-Sync/Half-Async: an Architectural Pattern for Efficient and Well-structured Concurrent I/O,” in *Proceedings of the 2nd Annual Conference on the Pattern Languages of Program*s, (Monticello, Illino is), pp. 1–10, September 1995.

[26] Paul E. McKinney, “A Pattern Language for Parallelizing Existing Programs on Shared Memory Multiprocessors,” in

*Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley,

1996.

第 **7** 章 **ACE** 反应器（**Reactor**）的设计和使用：用于事件多路分离

的面向对象框架

Douglas C. Schmidt Irfan Pyarali22

**7.1** 介绍

本文关注 ACE 框架[2]中的反应器（Reactor）模式[1]的设计和实现。**反应器模式处理由一个或多个客户并发地投递给应用程序的服务请求**。应用程序的每个服务可由一个或多个方法组成，并由一个单独的事件处理器代表；事件处理器负责分派服务特有的请求。

在本论文描述的反应器模式的实现中，事件处理器分派由 ACE\_Reactor对象来完成。ACE\_Reactor 结合了 I/O 事件，以及其他类型的事件，比如定时器和信号的多路分离。在此实现的核心是一个同步的事 件多路分离器，例如，select[3]或 WaitForMultipleObjects[4]。**当事件发生时，ACE\_Reactor 自动分派预登记的事件处理器的方法，后者随即执行应用指定的服务。**

本论文被组织如下：7.2 描述 ACE\_Reactor 框架中的主要特性；7.3 概述 ACE\_Reactor 实现的 OO 设 计[2]；7.4 检查若干例子、演示 ACE\_Reactor 怎样简化并发的事件驱动网络应用的开发；7.5 描述使用 A CE\_Reactor 开发事件驱动应用时所应遵循的设计规则；7.6 给出结束语。

**7.2** 反应器的特性

ACE\_Reactor 提供 OO 多路分离和分派框架，它简化了事件驱动应用的开发。下面的段落描述反应器 框架提供给应用开发者的特性：

统一的 **OO** 多路分离和分派接口：使用 ACE\_Reacotr 的应用并不直接访问诸如 select 或 WaitForMultiple Objects 这样的低级事件多路分离系统调用，而是通过从 ACE\_Event\_Handler 基类继承来创建具体的事件 处理器。该类指定处理多种类型事件的虚方法，比如 I/O 事件、定时器事件、信号和同步事件。图 7-1 演 示 ACE\_Reactor 中的关键组件。它演示了实现 7.4 描述的日志服务器的具体事件处理器。使用反应器框 架的应用创建具体事件处理器，并将它们登记到 ACE\_Reactor。

使事件处理器分派自动化：当 ACE\_Reactor 管理的处理器上有活动发生时，反应器自动调用适当的预登 记的具体事件处理器的虚方法。C++对象被登记到 ACE\_Reactor。对象、而不是单独的函数的使用，允许 在对具体事件处理器的挂钩方法的调用之间，状态可以方便地保持。这种风格的 OO 编程对于开发在多 次客户调用间保持状态的事件处理器来说很有用。

22 本工作部分地得到了西门子 SCR 的支持。

支持透明的可扩展性：ACE\_Reactor 的功能和它的已登记的事件处理器可被透明地扩展，而无须修改或 重编译现有代码。为实现这样的扩展性，反应器框架采用继承和动态绑定来去耦（1）它的较低级的事件 多路分离和分派机制与（2）应用定义的处理事件的较高级的策略。

ACE\_Reactor 管理的低级机制包括检测多个 I/O 句柄上的事件、使定时器到期，以及分派适当的事件 处理器方法来处理这些事件。应用特有的具体事件处理器执行的较高级策略包括连接建立策略、数据编 码和解码，以及处理来自客户的服务请求。例如，TAO[5]实时 CORBA ORB 使用反应器框架来分离它的 低级事件多路分离机制和它的 GIOP 连接管理和协议处理[5]的较高级策略。

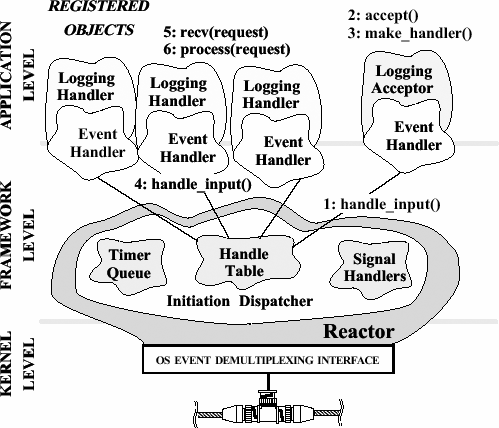


图 7-1 反应器组件

增加复用：ACE\_Reactor 的多路分离和分派机制可被许多网络应用复用。通过复用，而不是重新发明， 开发者可以专注于较高级的应用特有的策略，而不是反复与较低级事件多路分离和分派机制相纠缠。

相反，直接使用像 select 和 WaitForMultipleObject 这样的低级事件多路分离操作的程序员必须为每一 个应用重新实现、调试和调谐同样的多路分离和分派代码。而所有利用 ACE\_Reactor 的应用自动地复用 它的特性，以及将来的增强和优化。

增强类型安全性：ACE\_Reactor 将应用开发者和易错的细节（它们与使用像 select 这样的低级事件多路分 离系统调用来编程是相关联的）屏蔽开来。这些细节涉及设置和清除位掩码、处理超时和中断，以及分 派挂钩方法。特别地，ACE\_Reacotr 消除了导致 select 出错的若干微妙原因，这些错误涉及 fd\_set 位掩码 的误用。

改善可移植性：ACE\_Reactor 在若干事件多路分离机制上运行，其中包括 Win32 WaitForMultipleObjects 和 select（它在 UNIX 和 Win32 上都可用）。ACE\_Reactor 将应用和底层事件多路分离机制的移植性差异 屏蔽开来。如图 7-6 所示，ACE\_Reactor 向应用输出同样的接口，而不管底层系统调用是什么。而且，A CE\_Reactor 使用像桥接（Bridge）[6]这样的设计模式来增强它内部的可移植性。因而，将 ACE\_Reactor 从 select 移植到 WaitForMultipleObjects 仅需要对框架进行一些局部变动[7]。

线程安全性：反应器框架是完全多线程的。所以多个线程可安全地共享单个 ACE\_Reactor。同样地，多 个 ACE\_Reactor 也可在一个进程的不同线程中运行。反应器框架提供必要的同步机制来防止条件竞争和 类中方法的死锁[8]。

高效的多路分离：ACE\_Reactor 非常高效地执行它的事件多路分离逻辑。例如，基于 select 的 ACE\_Reac tor 使用 7.3.2 中描述的 ACE\_Handle\_Set 类来避免每次只查看 fd\_set 位掩码的一位。这一优化基于一种成 熟的算法，使用异或操作符来将运行时复杂度从 O（总位数）降低到 O（已置位的位数）。这充分地减少 了运行时开销。

**7.3** 反应器框架的 **OO** 设计

这一部分描述反应器框架的 OO 设计。主要焦点在于它的组件的结构和关键的设计决策。在适当的 地方还讨论了实现细节。7.3.1 概述 OS 平台无关的组件，而 7.3.2 覆盖平台相关的组件。

**7.3.1** 平台无关的类组件

这一小部分总结反应器框架中的平台无关的类，其中包括 ACE\_Reactor、ACE\_Time\_Value、ACE\_T

imer\_Queue 和 ACE\_Event\_Handler。

**7.3.1.1 ACE\_Reactor** 类

ACE\_Reactor 定义反应器框架的公共接口。图 7-2 演示 ACE\_Reactor 类中关键的公共方法。此类中的 方法大致被分组为以下几个种类：

管理器方法：构造器和 open 方法通过动态地分配多个实现对象来创建和初始化 ACE\_Reactor 对象；这些 实现对象在下面的 7.3.2.1 和 7.3.2.2 中描述。析构器和 close 方法释放这些对象。

基于 **I/O** 的方法：派生自 ACE\_Event\_Handler 的具体事件处理器通过反应器的 register\_handler 方法登记 到 ACE\_Reactor；同样地，具体事件处理器可以通过它的 remove\_handler 方法移除。

基于定时器的方法：ACE\_Reactor 的定时器策略使用最终期限来评估被调度的定时器的优先级。ACE\_Re actor 的定时器策略提供的操作包括（1）登记将在用户指定时间执行的具体事件处理器和（2）取消先前 登记的事件处理器。

事件循环方法：在登记初始的具体事件处理器之后，应用最终进入一个事件循环，重复调用 ACE\_React or 的 handle\_events 方法中的一个。这些方法阻塞应用指定长度的时间，等待各种事件的发生，比如 I/O 句柄上的同步 I/O 事件或基于定时器的事件。在事件发生时，ACE\_Reactor 分派具体事件处理器的适当方 法；这些方法已被应用登记以处理这些事件。

class ACE\_Reactor

{

public:

// = Initialization and termination methods. enum { DEFAULT\_SIZE = FD\_SETSIZE };

// Initialize a Reactor instance that may

// contain SIZE entries (<restart> indicates

// to restart system calls after interrupts). ACE\_Reactor (int size, int restart = 0);

virtual int open (int size = DEFAULT\_SIZE, int restart = 0);

// Perform cleanup activities to close down

// an instance of an <ACE\_Reactor>. void close (void);

˜ACE\_Reactor (void);

// = I/O-based event handler methods.

// Register an <ACE\_Event\_Handler> object according

// to the <ACE\_Reactor\_Mask>(s) (e.g., READ\_MASK,

// WRITE\_MASK, etc.).

virtual int register\_handler (ACE\_Event\_Handler \*, ACE\_Reactor\_Mask);

// Remove the handler associated with the

// appropriate <ACE\_Reactor\_Mask>(s).

virtual int remove\_handler (ACE\_Event\_Handler \*, ACE\_Reactor\_Mask);

// = Timer-based event handler methods.

// Register a handler to expire at time <delta>.

// When <delta> expires the <handle\_timeout>

// method will be called with the current time

// and <act> as parameters. If <interval> is > 0

// then the handler is reinvoked periodically

// at that <interval>. The <delta> is interpreted

// "relative" to the current time of day.

virtual void schedule\_timer

(ACE\_Event\_Handler \*, const void \*act,

const ACE\_Time\_Value &delta, const ACE\_Time\_Value &interval);

// Locate and cancel timer.

virtual void cancel\_timer (ACE\_Event\_Handler \*);

// = Event-loop methods

// Block process until I/O events occur or timer

// expires, then dispatch activated handler(s). virtual int handle\_events (void);

// Perform a timed event-loop that waits up to TV

// time units for events to occur; if no events

// occur then 0 is returned, otherwise return

// TV - (actual\_time\_waited).

virtual int handle\_events (ACE\_Time\_Value &tv);

private:

// Pointer to the implementation class,

// e.g., <ACE\_Select\_Reactor> or

// <ACE\_WFMO\_Reactor>. Reactor\_Impl \*reactor\_impl\_;

};

图 7-2 ACE 反应器接口

**7.3.1.2 ACE\_Event\_Handler** 类

该基类指定 ACE\_Reactor 用以控制和协调具体事件处理器的多路分离和分派接口。ACE\_Event\_Han dler 接口中的虚方法如图 7-3 所示。

// Handle portability issues.

#if defined (UNIX)

typedef int ACE\_HANDLE;

#elif defined (WIN32)

typedef HANDLE ACE\_HANDLE;

#endif /\* UNIX \*/

class ACE\_Event\_Handler

{

public:

// These values can be bitwise "or’d" together to

// instruct the <ACE\_Reactor> to check for

// multiple I/O activities on a single handle. enum {

READ\_MASK = 01, WRITE\_MASK = 02, EXCEPT\_MASK = 04,

RWE\_MASK = READ\_MASK | WRITE\_MASK | EXCEPT\_MASK,

DONT\_CALL // Don’t callback to handle\_close().

};

// Returns the I/O handle associated with the

// derived object (must be supplied by a subclass). virtual ACE\_HANDLE get\_handle (void) const = 0;

// Called when event handler is removed from

// an <ACE\_Reactor>.

virtual int handle\_close (ACE\_HANDLE, ACE\_Reactor\_Mask);

// Called when input becomes available. virtual int handle\_input (ACE\_HANDLE);

// Called when output is possible. virtual int handle\_output (ACE\_HANDLE);

// Called when urgent data is available. virtual int handle\_except (ACE\_HANDLE);

// Called when timer expires (<tv> stores the

// current time and <act> is the argument given

// when the handler was originally scheduled).

virtual int handle\_timeout (const ACE\_Time\_Value &tv, const void \*act = 0);

// Called when signal is triggered by OS. virtual int handle\_signal (int signum);

};

图 7-3 ACE 事件处理器接口

ACE\_Reactor 通过派生自 ACE\_Event\_Handler 的具体事件处理器来实现它的事件驱动回调机制。具

体事件处理器可以有选择地重定义 ACE\_Event\_Handler 中的虚方法，以执行应用定义的处理来响应多种 类型的事件。这些事件包括（1）同步 I/O，例如，读、写和异常；（2）定时器；（3）信号；以及（4） 同步，例如，将 Win32 互斥体从复位切换到置位[4]。

7.3.1.4 中描述的 ACE\_Timer\_Queue 使用具体事件处理器来处理基于定时器的事件。当此队列管理的 定时器到期时，先前排定的（scheduled）事件处理器的 handle\_timeout 方法被调用。有两个参数被传递给

该方法：（1）当前时间和（2）void \* 异步完成令牌（ACT）[9]，当事件处理器最初被排定时，此令牌 也作为参数被传递给 schedule\_timer。

当具体事件处理器中的任何方法的返回值<０时，反应器都自动调用处理器的 handle\_close 清扫方法。 应用可对该方法进行编程以执行终止活动，比如关闭日志文件或删除对象分配的动态内存。当 handle\_cl ose 方法返回时，ACE\_Reactor 从它的内部表中将与之相关联的具体事件处理器移除掉。

具体事件处理器通常要提供一个 I/O 句柄23。当应用调用 ACE\_Reactor 的 register\_handler 方法时，该 方法回调具体事件处理器的 get\_handle 方法，以获取底层的 I/O 句柄。

当应用调用 ACE\_Reactor 的 handle\_events 方法时，所有已登记的具体事件处理器的句柄都被传递给 底层的 OS 事件多路分离调用，例如，select 或 WaitForMultipleObjects。随后，当 I/O 事件变为“就绪” 时，OS 就激活这些句柄。在此时，ACE\_Reactor 通过调用被定义用来处理该事件的方法，通知适当的具 体事件处理器。

**7.3.1.3 ACE\_Time\_Value** 类

该 C++包装封装底层 OS 平台的日期和时间结构，例如，在大多数 UNIX 平台上定义的 struct timev al 类型。timeval 结构包括两个域，根据秒和微秒来表示时间。其他 OS 平台，比如 POSIX 和 Win32，使 用稍稍不同的时间表示法。为此，ACE\_Time\_Value 类封装这些细节以提供可移植的 C++接口。

ACE\_Time\_Value 类的主要方法在图 7-4 中演示。ACE\_Time\_Value 包装使用操作符重载来简化基于 时间的比较。这样的重载允许标准的算术语法被用于涉及时间比较的关系表达式。

// Time value structure from /usr/include/sys/time.h

// struct timeval { long secs; long usecs; };

class ACE\_Time\_Value

{

public:

ACE\_Time\_Value (long sec = 0, long usec = 0); ACE\_Time\_Value (timeval t);

// Returns sum of two <ACE\_Time\_Value>s. friend ACE\_Time\_Value operator +

(const ACE\_Time\_Value &lhs, const ACE\_Time\_Value &rhs);

// Returns difference between two

// <ACE\_Time\_Value>s.

friend ACE\_Time\_Value operator -( const ACE\_Time\_Value &lhs, const ACE\_Time\_Value &rhs);

// Relational and comparison operators for

23 完全基于定时器的具体事件处理器中的 I/O 句柄可被忽略。

// normalized <ACE\_Time\_Value>s. friend int operator <

(const ACE\_Time\_Value &lhs, const ACE\_Time\_Value &rhs);

// Other relation operators...

private:

// ...

};

图 7-4 ACE 时间值接口

ACE\_Time\_Value 的方法被实现用来“规格化”时间数量。规格化调整 timeval 结构中的两个域，使

之使用能确保精确比较的规范的编码方式。例如，在规格化之后，数量 ACE\_Time\_Value(1, 1000000)将 与 ACE\_Time\_Value(2)相等。相反，直接按位比较这些非规格化的类值将检测不到它们的相等。

下面的代码创建两个 ACE\_Time\_Value 对象，它们由用户提供的命令行参数加上当前时间来构造。 随后显示两个对象之间的正确顺序关系：

int main (int argc, char \*argv[])

{

if (argc != 3)

ACE\_ERROR\_RETURN ((LM\_ERROR, "usage: %d" "time1 time2\n"), 1);

ACE\_Time\_Value time = ACE\_OS::gettimeofday ();

ACE\_Time\_Value timer1 = time + ACE\_Time\_Value (ACE\_OS::atoi (argv[1])); ACE\_Time\_Value timer2 = time + ACE\_Time\_Value (ACE\_OS::atoi (argv[2]));

if (timer1 > timer2)

ACE\_DEBUG ((LM\_DEBUG, "timer 1 is greater\n"));

else if (timer2 > timer1)

ACE\_DEBUG ((LM\_DEBUG, "timer 2 is greater\n"));

else

ACE\_DEBUG ((LM\_DEBUG, "timers are equal\n"));

return 0;

}

上面所示的代码对所有 OS 平台都是完全可移植的。注意像操作符重载和类这样的 C++特性的使用是

怎样简化与时间相关操作的使用的。

**7.3.1.4 ACE\_Timer\_Queue** 类

ACE\_Reactor 的基于定时器的机制对于需要定时器支持的应用来说是很有用的。例如，WWW 服务 器需要看门狗定时器来释放资源，如果客户在它们连接后不在指定的时间间隔内发送 HTTP 请求的话。 同样地，像 Windows NT Service Control Manager[4]这样的看守配置框架要求在它们控制之下的服务周 期性地报告它们的当前状态。这些“心跳”消息用于确认服务没有异常地终止。

ACE\_Timer\_Queue 类提供的机制允许应用登记派生自 ACE\_Event\_Handler 的、基于定时器的具体事 件处理器。ACE\_Timer\_Queue 确保这些事件处理器中的 handle\_timout 方法将来在应用指定的时间被调用。 图 7-5 中所示的 ACE\_Timer\_Queue 类的方法使得应用可以调度、取消，和调用定时器对象。

class ACE\_Timer\_Queue

{

public:

ACE\_Timer\_Queue (void);

// True if queue is empty, else false. int is\_empty (void) const;

// Returns earliest time in queue.

const ACE\_Time\_Value &earliest\_time (void) const;

// Schedule a <handler> to be dispatched at

// the <future\_time> and at subsequent

// <interval>s.

int virtual schedule

(ACE\_Event\_Handler \*handler, const void \*act,

const ACE\_Time\_Value &future\_time, const ACE\_Time\_Value &interval);

// Cancel all registered <ACE\_Event\_Handlers>

// that match the address of <handler>, which

// can be registered multiple times.

int virtual cancel(ACE\_Event\_Handler \*handler);

// Cancel the single <ACE\_Event\_Handler>

// matching the <timer\_id> value returned

// from <schedule>.

int virtual cancel (int timer\_id, const void \*\*act = 0);

// Expire all timers <= <expire\_time>

// (note, this method must be called manually

// since it is not invoked asynchronously).

void virtual expire(const ACE\_Time\_Value &expire\_time);

private:

// ...

};

图 7-5 ACE\_Timer\_Queue 接口

应用调度具体事件处理器，在延迟一定数量的时间之后到期。如果它到期了，act 作为值被传递给事

件处理器的 handle\_timeout 挂钩方法。如果 interval 不等于 ACE\_Time\_Value::zero，该值就被用于自动重 调度事件处理器。

Schedule 方法返回一个定时器 id，唯一地标识每个事件处理器在定时器队列的内部表中的登记。定 时器 id 被 cancel 方法用于在事件处理器到期之前将其移除。如果一个非 NULL act 被传给 cancel，这个 a ct 就被设置为应用在定时器最初被排定时所传入的异步完成令牌（ACT）[9]。 这使得程序可以释放动态 分配的 ACT，以避免内存泄漏。

缺省地，ACE\_Reactor 所用的 ACE\_Timer\_Queue 被作为堆来实现。堆是一种“部分有序的、几乎完 全的二进制树”，它确保插入和删除一个具体事件处理器的平均和最坏情况的时间复杂度为 O(lg n)。堆 表示法对大多数应用实例、特别是实时应用来说都是很适用的。

ACE\_Timer\_Queue 堆由含有 ACE\_Time\_Value、ACE\_Event\_Handler \*，和 void \*的三元组组成。A CE\_Event\_Handler \* 域指向被排定的定时器对象，该对象的运行时间由 ACE\_Time\_Value 域指定。void

\* 域是在具体事件处理器被最初被排定时所提供的参数。当定时器到期时，这个参数被自动传给 7.3.1.2 中描述的 handle\_timeout 方法。堆中的每个 ACE\_Time\_Value 都以绝对时间单元来存储，例如，按照 UN IX gettimeofday 系统调用所生成的时间。

在 ACE\_Timer\_Queue 接口中使用了虚方法。因而，应用可以扩展缺省的 ACE 实现来支持其他可选 的数据结构，如 *delta* 表[11]和定时轮（timing wheel）[12]。delta 表将时间存储为“相对的”单元，即相 对于表的最前面的 ACE\_Time\_Value 的偏移或“delta”。定时轮使用循环缓冲区，使得程序有可能在 O(1) 时间内启动、停止和维护定时器。ACE 框架提供了若干可选的定时器队列的实现。

**7.3.2** 平台相关的类组件

ACE\_Reactor 类是应用用以访问 ACE 反应器框架的公共接口。ACE\_Reactor 接口由一些虚方法组成。 因此，它可以通过继承来扩展。

但是，扩展 ACE\_Reactor 最常用的方法并不是派生它的子类。相反，如图 7-6 所示，桥接模式[6]被 用来使 ACE\_Reactor 接口与它的 ACE\_Reactor\_Impl 子类实现去耦合。ACE 框架提供的两个子类包括 AC E\_Select\_Reactor 和 ACE\_WMFO\_Reactor，它们分别封装了 select 和 WaitForMultipleObjects OS 事件多路 分离调用。

在不同 OS 平台上，ACE\_Reactor\_Impl 子类的实现也不同。但是，ACE\_Reactor 接口提供的方法的 名字和总的功能保持不变。这种统一性源于 ACE\_Reactor 的设计的模块性，该设计还增强了反应器的复 用、可移植性和可维护性。ACE\_Reactor 的 WaitForMultipleObjects 和 select 版本概述如下。

**7.3.2.1 ACE\_Select\_Reactor** 类

如图 7-6(1)所示，基于 select 的 ACE\_Reactor 包含有三个 ACE\_Event\_Handler \* 数组。这些数组存

储的指针指向已登记用来处理各种类型事件的具体事件处理器。

ACE\_Handle\_Set 类为底层的 fd\_set 位掩码数据类型提供高效的 C++包装。fd\_set 将 I/O 句柄名字空 间映射到紧凑的位向量表示，并提供若干操作来置位、复位和测试与 I/O 句柄相对应的位。可传给 select 调用一或多个 fd\_set。

ACE\_Handle\_Set 类通过以下方法来优化若干常用的 fd\_set 操作：（1）使用“全字”（full-word）比 较，以使不必要的位操作最少化， （2）缓存某些值，以避免每次调用都计算位偏移，以及（3）使用一 种异或算法，它与 fd\_set 中的活动句柄的数目、而非潜在的活动句柄的数目线性相关。

**7.3.2.2 ACE\_WMFO\_Reactor** 类

WaitForMultipleObjects 接口比 select 更为通用，它允许应用等待更广泛的事件，比如同步事件。因 此，基于 WaitForMultipleObjects 的 ACE\_Reactor 既不需要三个 ACE\_Event\_Handler \* 数组，也不需要 A CE\_Handle\_Set 类，而是在内部分配和使用单个的 ACE\_Event\_Handler 指针数组和句柄数组，以存储已登 记的具体事件处理器。

..J

..J

u

-

..J

*REGISTERED*

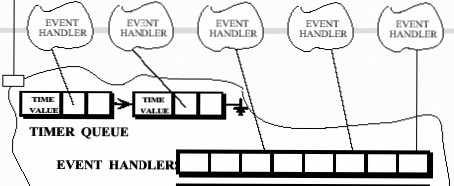
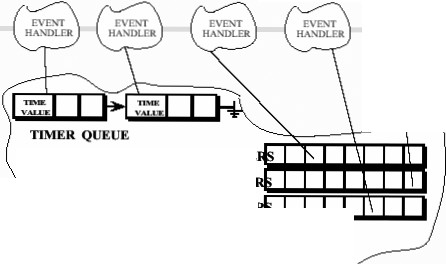
Reactor

Reactor-Imp\*reactor-impl\_



*REGISTERED*

< *OBJECTS*



*IO'*

*I NTS TO ONE IMPL*

*OBJECTS*

*(*

"l;('l

..J

..J

;:..

EXCEPT ON EVENT HANDLE' WRITE EVENT HANDLE

READ EVENT HANDLE 1 1 1 1 1 • 1 1 1

t Select Reactor

u

1 select() I -

HANDLE ARRA1 I I I I I I I I

WMFO Reador



WaliForMuJtipleObjects()

OPERATING SYSTEM LEVEL

图 7-6 将桥接模式用于反应器实现

**7.4** 分布式日志服务例子

反应器框架意在简化事件驱动应用的开发，比如 Web 服务器[13, 14]和 CORBA 对象请求代理[15]。 这一部分描述一个分布式日志服务（distributed logging service）的设计和实现，以演示 ACE\_Reactor 在 实践中的应用。

**7.4.1** 综述

日志提供一种“只追加”的存储服务，记录由一或多个应用发来的诊断信息。日志的主要单元是记 录。到来的记录被追加到日志的末尾，而所有其它类型的写访问都是被禁止的。

如图 7-7 所示，下面检查的日志服务使用客户/服务器体系结构，以使通过 TCP/IP 网络连接的工作站 和服务器能够记录事件。日志服务将 ACE\_Reactor 的多路分离与分派特性和[16]中描述的 C++ IPC 包装 库提供的 BSD sockets 与系统 V 传输层接口（TLI）的 OO 接口结合在了一起。

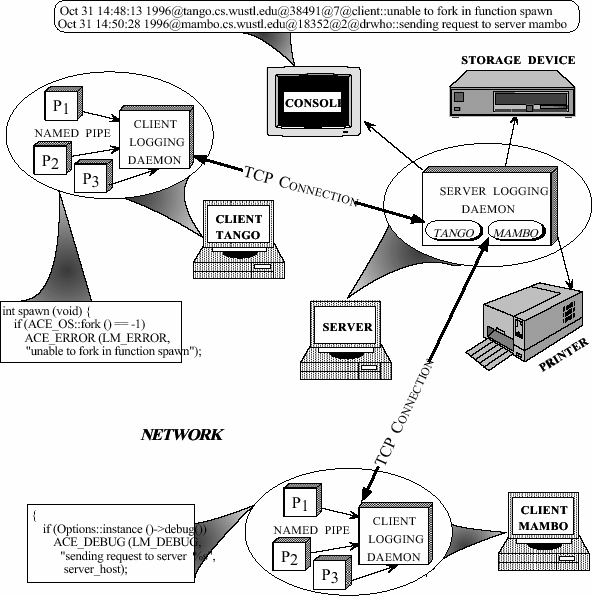


图 7-7 分布式日志服务中的组件

日志服务中的关键组件描述如下：

应用日志接口（**Application logging interface**）：在客户主机上运行的应用进程，例如，P1、P2、P3， 使用 ACE\_Log\_Msg C++类来生成日志记录，比如 LM\_ERROR 和 LM\_DEBUG。ACE\_Log\_Msg::log 方 法提供一种 printf 风格的接口。图 7-8 概述了在应用接口、客户和服务器日志看守之间交换的记录的若干 优先级和数据格式。当被应用调用时，日志接口格式化日志记录，打上时间戳，将它们写到周知的 STR EAM 管道[17]中去。如下所述，客户日志看守负责处理这些记录。

// The following enum indicates the relative

// priorities of the logging messages. enum Log\_Priority

{

// Messages that contain information normally

// used only when debugging a program.

LM\_DEBUG,

// Critical conditions, e.g., hard device errors

LM\_ERROR

// ...

};

struct Log\_Record

{

enum {

// Maximum number of bytes in logging record. MAXLOGMSGLEN = 1024

};

// Type of logging record. Log\_Priority type\_;

// length of the logging record. long length\_;

// Time logging record generated. long time\_stamp\_;

// Id of process that generated the record. long pid\_;

// Logging record data.

char rec\_data\_[MAXLOGMSGLEN];

};

图 7-8 日志记录格式

客户日志看守（**Client logging daemon**）：客户日志看守运行在所有参与分布式日志服务的主机上。每

个客户日志看守与 STREAM 管道的读端相连，后者用于从这台机器上的应用那里接收日志记录。使用 S TREAM 管道是因为它们是只用于本地主机的 IPC 的高效形式。此外，STREAM 管道的语义允许“结合 优先级”的消息，可按照“重要性顺序”，以及“到达顺序”接收[18]。

客户日志看守从应用进程那里持续地接收日志记录。它随后将多字节的记录头字段转换为网络字节 顺序。最后，它使用 TCP 将记录转发给服务器日志看守。服务器通常运行在远地主机上。

服务器日志看守（**Server logging daemon**）：服务器日志看守持续地收集、重格式化和输出到来的日志 记录。这一部分的余下部分专注于服务器日志看守。在此例中演示和描述了多种 ACE\_Reactor 和 ACE C

++ socket 包装机制。

**7.4.2** 服务器日志看守

下面描述用来构造服务器日志看守的类的接口和实现。日志服务器在单独的进程中运行，并发地处 理来自客户的日志记录。并发是由 ACE\_Reactor 提供的，它将它的注意力以循环方式“分时”给每个活 动的客户。

每次应用调用 ACE\_Reactor 的 handle\_events 方法，就从每个 I/O 句柄变为活动的客户那里读取一条 日志记录。日志记录被写到服务器日志看守的标准输出。该输出可被重定向到多种设备，比如打印机、 持久存储仓库或日志管理控制台。

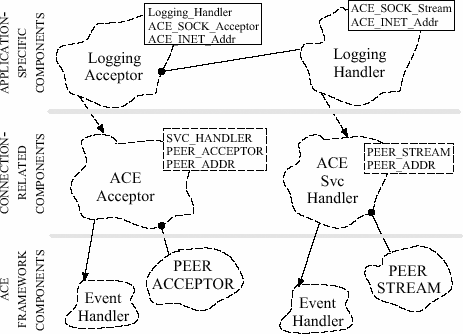


图 7-9 服务器日志看守中的组件

有若干 C++类组件出现在日志服务体系中。在图 7-9 中使用 Booch 表示法[19]演示了多种组件间的继 承和模板参数化关系。为增强复用和可扩展性，图中所示的组件被设计用以使服务器日志看守体系结构 的以下方面去耦合：

反应器框架组件（**Reactor framework component**）：反应器框架中的组件封装执行 I/O 多路分离和具体 事件处理器分派的最底层机制。这些组件已在 7.3 中讨论。

连接相关组件（**Connection-related component**）：这些通用模板实现接受器（Acceptor）模式[20]，提供 可复用的连接工厂组件。ACE\_Acceptor 是一个接受来自远地客户的网络连接、创建 ACE\_Svc\_Handler 的 模板。ACE\_Svc\_Handler 是与相连的客户交换数据的模板。这些组件在 7.4.2.1 中讨论。

应用特有组件（**Application-specific component**）：这些组件实现分布式日志服务的应用特有的部分。L

ogging\_Acceptor 类给 ACE\_Acceptor 提供具体的参数化类型，后者创建专用于日志应用的连接处理实例。

同样地，Logging\_Handler 类也通过具体类型来实例化，这种具体类型提供必需的应用特有的功能，以接 收和处理来自远地客户的日志记录。这些组件在 7.4.2.2 中讨论。

使用这样的高度去耦合的 OO 分解极大地增强了服务器日志看守的开发和可扩展性。这些组件的每 一个被描述如下。

**7.4.2.1** 连接相关组件

下面的类用于实现 Acceptor 模式[20]。该模式用于使（1）被动连接建立与（2）一旦服务的两端连接 和初始化后、服务所进行的处理去耦合。

**ACE\_Acceptor** 类：该类为一族类提供一种通用模板，使从客户接受网络连接请求所必需的步骤标准化和 自动化。图 7-10 演示了 ACE\_Acceptor 类的接口。

// A template class that handles connection

// requests from a remote client.

template <class SVC\_HANDLER, class PEER\_ACCEPTOR>

class ACE\_Acceptor : public ACE\_Event\_Handler

{

public:

ACE\_Acceptor (ACE\_Reactor \*r, const PEER\_ACCEPTOR::PEER\_ADDR &a);

～ACE\_Acceptor (void);

protected:

virtual ACE\_HANDLE get\_handle (void) const; virtual int handle\_input (ACE\_HANDLE); virtual int handle\_close

(ACE\_HANDLE = ACE\_INVALID\_HANDLE,

ACE\_Reactor\_Mask = ACE\_Event\_Handler::READ\_MASK);

private:

// Accept connections. PEER\_ACCEPTOR acceptor\_;

};

图 7-10 Acceptor 类接口

ACE\_Acceptor 模板类继承自 ACE\_Event\_Handler。这一派生使得 ACE\_Acceptor 可以与反应器框架

无缝地交互。此外，该模板类通过具体的 SVC\_HANDLER（它知道怎样执行与客户的 I/O）和 PEER\_AC CEPTOR 类来参数化（它知道怎样接受客户连接）。

从 ACE\_Acceptor 实例化的类有能力完成以下工作：

1. 接受远地客户发送的连接请求。

2. 动态分配 SVC\_HANDLER 子类的对象。

3. 将此对象登记到 ACE\_Reactor 的一个实例。随后，SVC\_HANDLER 类必须知道怎样处理与客户交换 的数据。

ACE\_Acceptor 类的实现如图 7-1124所示。当一或多个连接请求到达时，handle\_input 方法被 Reactor 自动分派。该方法的行为如下：首先，它动态创建新的 SVC\_HANDLER 对象，负责处理发送数据和接收 来自新客户的数据。其次，它将一个到达的连接接受进 SVC\_HANDLER。最后，它调用新 SVC\_HANDL ER 的 open 挂钩。如下所示，该挂钩可以将新创建的 SVC\_HANDLER 登记到 ACE\_Reactor，或是生成一 个独立的线程控制，等等。

// Shorthand names

#define SH SVC\_HANDLER

#define PA PEER\_ACCEPTOR

template <class SH, class PA> ACE\_Acceptor<SH, PA>::ACE\_Acceptor

(ACE\_Reactor \*reactor,

const PA::PEER\_ADDR &addr)

: acceptor\_ (addr)

{

// Register to accept connections. reactor->register\_handler

(this, ACE\_Event\_Handler::ACCEPT\_MASK);

}

template <class SH, class PA> ACE\_HANDLE ACE\_Acceptor<SH, PA>::get\_handle (void) const

{

// Return the underlying I/O handle

// when called by Reactor during

// registration.

return this->acceptor\_.get\_handle ();

}

template <class SH, class PA> int

ACE\_Acceptor<SH, PA>::handle\_close

(ACE\_HANDLE, ACE\_Reactor\_Mask)

{

// Close down the Acceptor and

// release the handle resources. return this->acceptor\_.close ();

24 这是 ACE\_Acceptor 的简化版本。完整实现见[20]。

}

template <class SH, class PA> ACE\_Acceptor<SH, PA>::˜ACE\_Acceptor (void)

{

this->handle\_close ();

}

// Template Method that accepts connections

// from client hosts, creates and activates

// a service handler.

template <class SH, class PA> int

ACE\_Acceptor<SH, PA>::handle\_input

(ACE\_HANDLE)

{

// Create a new Svc\_Handler. SH \*svc\_handler = new SH;

// Accept connection into the handler.

this->acceptor\_.accept (svc\_handler->peer ());

// Activate the handler. svc\_handler->open (0);

}

图 7-11 Aceptor 类实现

**ACE\_Svc\_Handler** 类：该参数化类型为处理与客户交换的数据提供一种通用模板。例如，在分布式日志

服务中，I/O 格式牵涉到日志记录。但是，也可以很容易地为其他应用换用不同的格式。ACE\_Svc\_Handl er 类的接口在图 7-12 中描述。和 ACE\_Acceptor 类一样，该类继承 ACE\_Event\_Handler 基类的功能。这 使得从 ACE\_Svc\_Handler 实例化的具体事件处理器可被动态创建并登记到 ACE\_Reactor。ACE\_Acceptor 类中的 handle\_input 方法自动完成这一行为。

// Receive client message from the remote clients. template <class PEER\_STREAM>

class ACE\_Svc\_Handler : public ACE\_Event\_Handler

{

public:

ACE\_Svc\_Handler (void);

// Must be filled in by subclass virtual int open (void \*) = 0; PEER\_STREAM &peer (void);

// Demultiplexing hooks.

virtual ACE\_HANDLE get\_handle (void) const;

protected:

// Connection open to the client. PEER\_STREAM peer\_stream\_;

};

图 7-12 Svc\_Handler 类接口

图 7-13 演示 ACE\_Svc\_Handler 类实现。注意继承、动态绑定和参数化类型的结合是怎样使框架的通

用部分（例如，连接建立）与应用特有的功能（例如，接收日志记录）去耦合的。

#define PS PEER\_STREAM

// Extract the underlying PS (e.g., for

// use by accept()).

template <class PS> PS & ACE\_Svc\_Handler<PS>::peer (void)

{

return this->peer\_stream\_;

}

template <class PS> ACE\_HANDLE ACE\_Svc\_Handler<PS>::get\_handle (void) const

{

// Return the underlying I/O handle

// when called by Reactor during

// registration.

return this->peer\_stream\_.get\_handle ();

}

图 7-13 Svc\_Handler 类实现

当 ACE\_Reactor 被指示从它的内部表中移除 ACE\_Svc\_Handler 时，它自动调用具体事件处理器的 ha

ndle\_close 方法。缺省地，该方法释放处理器的内存；该内存原来是由 ACE\_Acceptor 类中的 handle\_inpu t 方法分配的。具体事件处理器通常在客户进程关闭或发生严重传输错误时被移除。

**7.4.2.2** 应用特有的服务

下面的类实现服务的应用特有部分，在此例中也就是日志服务器看守。

**Logging\_Acceptor** 类：为实现分布式日志应用的服务器看守部分，Logging\_Acceptor 类从通用的 ACE\_A

cceptor 模板实例化，如下所示：

typedef ACE\_Acceptor <Logging\_Handler, ACE\_SOCK\_Acceptor>Logging\_Acceptor;

SVC\_HANDLER 模板参数通过下面描述的 Logging\_Handler 类实例化。同样地，PEER\_ACCEPTOR 模板 参数被 ACE\_SOCK\_Acceptor 类替换。ACE\_SOCK\_\* 实例化类型是称为 SOCK\_SAP 的 C++包装的一部 分[16]。SOCK\_SAP 封装 socket 接口，以在客户和服务器上的进程间可靠地传输数据。

通过使用参数化类型，用于 IPC 的类可以是任何遵从参数化类中所用 API 的网络编程接口。例如， 取决于底层 OS 平台的特定属性（比如是 UNIX 的 BSD 还是系统 V 变种），日志应用可以使用 SOCK\_S AP 或 TLI\_SAP（后者是系统 V 传输层接口（TLI）的 ACE C++包装）来实例化 ACE\_Svc\_Handler 类。 这种技术在下面演示：

// Logging application.

#if defined (USE\_SOCKETS)

typedef ACE\_SOCK\_Stream PEER\_STREAM;

#elif defined (USE\_TLI)

typedef ACE\_TLI\_Stream PEER\_STREAM;

#endif /\* USE\_SOCKETS \*/

class Logging\_Handler

: public ACE\_Svc\_Handler<PEER\_STREAM>

{

// ...

};

在开发必须跨越多种 OS 平台运行的应用时，基于模板的可扩展性所提供的灵活性十分有用。事实上，

通过传输接口来参数化应用的能力对于跨越 OS 平台的变种也很有用。例如，Solaris 的某些版本不提供线 程安全的 socket 实现。

**Logging\_Handler** 类：该类通过实例化 ACE\_Svc\_Handler 类创建：

class Logging\_Handler :

public ACE\_Svc\_Handler<ACE\_SOCK\_Stream>

{

public:

// Initialization hook called by

// the <ACE\_Acceptor>. virtual int open (void \*)

{

ACE\_SOCK\_Stream::PEER\_ADDR addr;

// Cache remove host name.

peer ().get\_remote\_addr (addr); ACE\_OS::strcpy (host\_name\_,

addr.get\_host\_name ());

// Register ourselves with the Reactor so

// we can be dispatched automatically when

// I/O arrives from clients. ACE\_Reactor::instance ()->register\_handler

(this, ACE\_Event\_Handler::READ\_MASK);

}

// Demultiplexing hook called by

// the <ACE\_Reactor>.

virtual int handle\_input (ACE\_HANDLE);

private:

char host\_name\_[MAXHOSTNAME];

};

当此类的对象被动态分配时，open 挂钩缓存相关联的客户的主机地址。如图 7-7 中的“控制台”窗口所 示，该主机的名字和从客户日志看守那里接收到的日志记录一起被打印出来。

PEER\_STREAM 参数被 ACE\_SOCK\_Stream 类替换。当输入到达底层的 ACE\_SOCK\_Stream 时，ha ndle\_input 方法被 ACE\_Reactor 自动调用。该方法可实现如下：

// Hook method for handling the reception of

// remote logging transmissions from clients. int Logging\_Handler::handle\_input (ACE\_HANDLE)

{

ssize\_t n = peer\_stream\_.recv (&len, sizeof len);

if (n != sizeof len)

// Trigger handle\_close(). return -1;

else

{

ACE\_Log\_Record lr;

size\_t len = ntohl (len);

n = this->peer\_stream\_.recv\_n (&lr, len));

if (n != len)

ACE\_ERROR\_RETURN ((LM\_ERROR,

"%p at host %s\n", "client logger",

lr.decode ();

this->host\_name\_),

-1);

if (lr.len == n)

lr.print (this->host\_name\_, 0, stderr);

else

ACE\_ERROR\_RETURN ((LM\_DEBUG,

"lr.len = %d, n = %d\n", lr.len,

n),

-1);

return 0

}

}

该方法执行两个 recv，以模拟经由底层 TCP 连接的面向消息的服务。这种行为是必需的，因为 TCP 提供

面向字节流、而不是面向记录的服务25。第一个 recv 读取跟随的日志记录的长度，此长度存储为定长整 数。随后第二个 recv 读取该长度那么多的字节，以获取实际的记录。自然，发送此消息的客户必须遵从 同样的消息帧协议。

**7.4.2.3 main()**驱动程序

下面的事件循环驱动基于 ACE\_Reactor 的日志服务器：

int main (int argc, char \*argv[])

{

// 1. Set the program name with the logger. ACE\_LOG\_MSG->open (argv[0]);

// Ensure correct usage. if (argc != 2)

ACE\_ERROR\_RETURN ((LM\_ERROR,

"usage: %n port-number"),

-1);

// 2. Create an addr and an acceptor. ACE\_INET\_Addr port (ACE\_OS::atoi (argv[1]));

Logging\_Acceptor acceptor

(ACE\_Reactor::instance (), port);

25 注意这一实现在处理“短读”时并不是完全健壮的。

// 3. Loop forever, handling client requests. for (;;)

ACE\_Reactor::instance ()->handle\_events ();

/\* NOTREACHED \*/

return 0;

}

在步骤 1 中，一个 ACE\_Log\_Msg 被创建，以将服务器生成的任何日志记录定向到它自己的标准错

误流。图 7-11 和图 7-13 中的代码演示服务器怎样使用应用日志接口来在本地记录它自己的诊断消息。因 为此本地配置不使用服务器日志看守，也就不会有导致“无限日志循环”的危险。

在步骤 2 中，服务器创建 Logging\_Acceptor，它的构造器将它自己登记到 ACE\_Reactor 单体。在步 骤 3 中，服务器进入一个无穷循环，它阻塞在 handle\_events 方法中，直到接收到来自客户日志看守的事 件。

图 7-14 演示在两个客户被 ACE\_Reactor 分派、并开始参与分布式日志服务之后，日志服务器看守的 状态。如图所示，为每个客户，动态地实例化和登记了一个 Logging\_Handler。

当事件到达服务器，ACE\_Reactor 自动分派 Logging\_Acceptor 和 Logging\_Handler 的 handle\_input 方 法。例如，当连接请求从客户日志看守到达时，ACE\_Reactor 调用 Logging\_Acceptor 的 handle\_input 方法。 该方法接受新连接，并创建 Logging\_Handler，由它读取客户发送的所有数据，并将其显示在标准输出流 上。同样地，当日志记录或关闭消息从相连的客户日志看守到达时，ACE\_Reactor 调用相应的 Logging\_ Handler 的 handle\_input 方法。

图 7-7 描绘整个运行中的系统。日志记录通过应用日志接口的 ACE\_Log\_Msg::log 方法生成。该方法 将日志记录转发给本地客户日志看守。客户日志看守随即通过网络将记录传送给服务器日志看守，在这 里它被显示在服务器的日志控制台上。

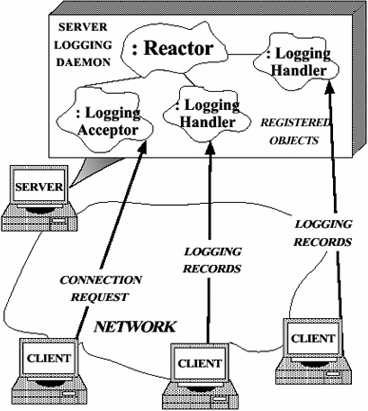


图 7-14 服务器日志看守的运行时配置

服务器显示的日志信息指示（1）应用接口生成日志记录的时间，（2）应用所运行在其上的主机，

（3）应用的进程标识符，（4）日志记录的优先级，（5）应用的命令行名字（也就是，“argv[0]”）， 以及（6）含有日志消息文本的任意文本串。

**7.4.3** 评估可选的日志工具实现

7.4.2 中描述的分布式日志服务最初用 C 编写，并被用于一种商业的在线事务处理产品。这一部分根 据若干软件质量因素，比如模块性、可扩展性、可复用性和可移植性，来比较该分布式日志服务的 C++ 和 C 版本。

**7.4.3.1** 基于 **C** 的非 **OO** 日志服务

基于 ACE\_Reactor 的分布式日志服务是对一种早期的、功能等价的非 OO 日志服务的面向对象的重 新实现。原版本是为一种基于 BSD UNIX 的商业在线事务处理产品开发的。当时它用 C 编写，并直接使 用 BSD socket 和 select。随后，它被移植到其他操作平台上，比如系统 V UNIX 和 Win32。

由于下面的一些问题，原来的 C 实现很难修改、扩展和移植：

紧耦合的功能：在原来的 C 日志服务中，事件多路分离、服务分派和事件处理操作都与接受客户连接请

求及接收客户日志记录的代码紧耦合在一起。

全局变量的过多使用：若干全局数据结构被用于维持下面两者之间的关系：（1）每个客户的上下文信息

（比如客户主机名和并发处理状态）和（2）标识相应的上下文记录的 I/O 句柄。因此，对程序的任何增 强和修改都将直接影响现有源代码。

**7.4.3.2** 基于 **C++**的 **OO** 日志服务

在本论文中描述的基于 ACE\_Reactor 的日志服务的 OO 版本使用数据抽象、继承、动态绑定和模板 来提供下面好处：

最小化对全局变量的依赖：基于 ACE\_Reactor 的日志服务没有包含全局变量。相反，每个登记到 ACE\_R

eactor 的 Logging\_Handler 对象都封装了客户地址和用于与客户通信的底层 I/O 句柄。

去耦策略和机制：使处理到来连接和数据的应用策略与执行多路分离和分派的底层机制去耦合。使策略 与机制去耦合增强了下列软件质量因子：

z 可复用性：ACE 反应器框架提供可复用的组件，执行所有较低级的事件多路分离和服务分派。因 而，如 7.4 所示，要实现服务服务器日志看守，仅需要少量应用特有的代码。这些代码主要用于 应用处理活动，比如接受新连接和接收客户日志记录。

z 可扩展性：ACE\_Reactor 体系结构中策略和机制的分离同时增强了它的公共接口之上和之下的可 扩展性。例如，要扩展服务器日志看守的功能，给它增加“认证日志”特性，相当地直接：只是 简单地从 ACE\_Event\_Handler 基类继承，并有选择地实现必需的虚方法。同样地，通过实例化 A CE\_Acceptor 和 ACE\_Svc\_Handler 模板，可以无需重新开发已有的基础构造，就制造出后续的应 用。相反，要对原来的非 OO 的 C 版本完成同样的修改，需要直接改动已有的代码。

z 可移植性：有可能修改 ACE\_Reactor 底层的事件多路分离机制，而又不影响现有的应用代码。例 如，从 BSD 平台将基于 ACE\_Reactor 的分布式日志服务移植到系统 V 或 Win32 平台，无需对应 用代码进行明显的修改。相反，将原有的 C 版本分布式日志服务从 select 移植到 WaitForMultiple Objects 是麻烦而易错的。例如，有若干被引入源代码的的微妙错误直到运行时才被发现。

z 效率：在某些实时应用中，数据会立即在一或多个句柄上可用。因此，通过非阻塞的 I/O 轮询这 些句柄可能比使用像 select 或 WaitForMultipleObjects 这样的 OS 事件多路分离器更为高效。扩展 ACE\_Reactor 以支持这种可选的多路分离实现无需修改它的公共接口。

在反应器框架中使用 OO 的一个逻辑后果是它大量地使用动态绑定。[16]讨论了为什么在为 socke t 设计“瘦”C++包装时，避免使用动态绑定常常是可取的。在有些编译器上，间接的虚表分派所 带来的开销可能会相当地高。在这样的情况下，开发者可能需要避免大量使用动态绑定。 但是，一般而言，ACE\_Reactor 所带来的清晰性、可扩展性和模块性的显著增强已足够补偿效率 的轻微下降。而且，ACE\_Reactor 通常用于开发分布式应用。分布式系统中开销的主要来源是像 缓存、延迟、网络/主机接口硬件、表示层格式化、内存到内存的拷贝，以及进程管理这样的活动 [21]。因而，动态绑定所导致的额外的间接性通常是微不足道的[22]。此外，通过使用“adjustor

thunks，”

好的 C++编译器可以将虚方法的额外开销完全优化掉[23]。

**7.5** 有效使用反应器的设计准则

ACE\_Reactor 是事件多路分离和事件处理器分派的强大框架。但是，像其他框架一样，学习使用 AC E\_Reactor 需要时间和努力。缩短学习曲线的的一种途径是去理解有效使用反应器所必须遵从的设计准则。 下面描述的设计准则基于帮助 ACE 用户正确进行反应器框架编程所获得的大量经验。

**7.5.1** 理解具体事件处理器的返回值语义

具体事件处理器定义的各种 handle\_\* 挂钩方法的返回值致使 ACE\_Reactor 以不同的方式工作。使用 返回值来触发不同行为意在降低 ACE\_Reactor 的 API 的复杂度。但是，返回值常常使得程序员莫明其妙。 因而，理解从 handle\_\* 方法返回的值的效应非常重要；这些值分为三种情况：

零：handle\_\* 方法返回零（0）通知 ACE\_Reactor、事件处理器希望继续像前面一样被处理，也就是，它 应该保持在 ACE\_Reactor 的实现的一张表中。这样，当下一次 ACE\_Reactor 的事件多路分离器系统调用 经由 handle\_events 被调用时，它还会继续包括该事件处理器的句柄。对于那些生存期超出一次 handle\_\* 方法分派的事件处理器，这是一种“正常的”行为。

大于零：handle\_\* 方法返回大于０（> 0）的值通知 ACE\_Reactor、事件处理器希望在 ACE\_Reactor 阻塞 在它的事件多路分离器系统调用上面之前，再一次被分派。对协作的事件处理器来说，这种特性有助于 增强全面的系统“公正性”。特别地，这种特性使得一个事件处理器在再次持有控制之前，允许其他事 件处理器被分派。

小于零：handle\_\* 方法返回小于０（< 0 ）的值通知 ACE\_Reactor、事件处理器想要被关闭、并从 ACE

\_Reactor 的内部表中移除。为完成此工作、ACE\_Reactor 调用事件处理器的 handle\_close 清扫方法。该方 法可以执行用户定义的终止活动，比如删除对象分配的动态内存或关闭日志文件。handle\_close 方法返回 后，ACE\_Reactor 将相关联的具体事件处理器从它的内部表中移除。

为减少 handle\_\* 返回值所带来的问题，在实现具体事件处理器时，遵守下面的设计准则：

设计准则 **0**：不要手工删除事件处理器对象或显式调用 *handle\_close*——相反，确保 ACE\_Reactor 自动调 用 handle\_close 清扫方法。因而，应用必须遵从适当的协议来移除事件处理器，也就是，或者通过（1） 从 handle\_\* 挂钩方法中返回负值，或者通过（2）调用 remove\_handler。

该设计准则确保 ACE\_Reactor 能够适当地清扫它的内部表。如果不服从这一准则，当 ACE\_Reactor 试图移除已经在外部被删除的具体事件处理器时，就会带来不可预测的内存管理问题。后面的设计准则 详细说明怎样确保 ACE\_Reactor 调用 handle\_close 清扫方法。

设计准则 **1**：从继承自 *ACE\_Event\_Handler* 的类的 *handle\_\** 方法中返回的表达式必须是常量（*constant*）。

这一设计准则有助于静态地检查是否 handle\_\* 方法返回了恰当的值。如果必须违反此准则，开发者必须 在 return 语句之前加一注释，解释为何要使用变量，而不是常量。

设计准则 **2**：如果从继承自 *ACE\_Event\_Handler* 的类的 *handle\_\** 方法中返回的值不为 *0*，必须在 *return*

语句之前加一注释，说明该返回值的含义。这一设计准则确保所有非 0 的返回值都是开发者有意使用的。

**7.5.2** 理解 **handle\_close()**清扫挂钩的语义

必须记住 handle\_close 清扫挂钩方法只能由 ACE\_Reactor（1）隐式地调用，也就是，当 handle\_\* 方 法返回-1 这样的负值时，或是（2）显式地调用，也就是，如果应用调用 remove\_handler 方法来解除具体 事件处理器的登记。特别地，ACE\_Reactor 不会在本地应用或是远地应用关闭 I/O 句柄时自动调用 handl e\_close。因此，应用必须确定何时 I/O 句柄已被关闭，并采取适当的步骤，以使 ACE\_Reactor 触发 handl e\_close 清扫方法。

下面的来自 7.4.2.2 的 Logging\_Handler 代码片段演示怎样正确地触发清扫挂钩：

// Hook method for handling the reception of

// remote logging transmissions from clients. int Logging\_Handler::handle\_input (ACE\_HANDLE)

{

ssize\_t n = peer\_stream\_.recv (&len, sizeof len);

if (n == 0)

// Trigger handle\_close(). return -1;

// ...

// Keep handler registered for ‘‘normal’’ case. return 0;

}

当 handle\_input 方法从 recv 那里收到 0，它就返回-1。该值触发 ACE\_Reactor 调用 handle\_close 清扫挂钩。

为最少化 handle\_\* 返回值所带来的问题，在实现具体事件处理器时，应遵守下面的设计准则：

设计准则 **3**：当你想要触发具体事件处理器的相应 *handle\_close* 清扫方法时，从 *handle\_\** 方法中返回一 个负值。值-1 通常用于触发清扫挂钩，因为它是 ACE\_OS 系统调用包装中一个常用的错误代码。但是， 任何来自 handle\_\* 方法的负数都将触发 handle\_close。

设计准则 **4**：将所有 *Event\_Handler* 清扫活动限制在 *handle\_close* 清扫方法中。一般而言，将所有的清扫 活动合并到 handle\_close 方法中，而不是分散在事件处理器的各个 handle\_\* 方法中要更为容易。在处理 动态分配的、必须用 delete this 来清除的事件处理器时，特别需要遵从此设计准则（见准则 9）。

**7.5.3** 记住 **ACE\_Time\_Value** 参数是相对的

传递给 ACE\_Reactor 的 schedule\_timer 方法的两个 ACE\_Time\_Value 参数必须相对于当前时间指定。 例如，下面的代码调度一个对象，延迟 delay 秒后开始，每 interval 秒打印一次可执行程序的名字（也就 是，argv[0]）：

class Hello\_World : public ACE\_Event\_Handler

{

public:

virtual int handle\_timeout (const ACE\_Time\_Value &tv, const void \*act)

{

ACE\_DEBUG ((LM\_DEBUG, "%[s] %d, %d\n", act,

tv.sec (), tv.usec ()));

return 0;

}

};

int main (int argc, char \*argv[])

{

if (argc != 3)

ACE\_ERROR\_RETURN ((LM\_ERROR,

"usage: %s delay interval\n", argv[0]), -1);

Hello\_World handler; // timer object.

ACE\_Time\_Value delay = ACE\_OS::atoi (argv[1]); ACE\_Time\_Value interval = ACE\_OS::atoi (argv[2]);

// Schedule the timer. ACE\_Reactor::instance ()->schedule\_timer

(&handler,

(const void \*) argv[0], delay,

interval);

// Run the event loop. for (;;)

ACE\_Reactor::instance ()->handle\_events ();

/\* NOTREACHED \*/

}

一种常见的错误是误将绝对的时间值传递给 schedule\_timer。例如，考虑一个不同的例子：

ACE\_Time\_Value delay = ACE\_OS::atoi (argv[1]);

delay += ACE\_OS::gettimeofday ();

// Callback every following 10 seconds. ACE\_Time\_Value interval = delay + 10;

ACE\_Reactor::instance ()->schedule\_timer

(&handler,

0, delay, interval);

但是，该定时器在将来很长的时间内都不会到期，因为它将本日的当前时间加到了用户所要求的 del ay 和 interval 上。

下面是实现具体事件处理器时，为最小化与绝对的 ACE\_Time\_Value 有关的问题，所应遵从的设计 准则：

设计准则 **5**：不要将绝对时间用作 *ACE\_Reactor::schedule\_timer* 的第三或第四参数。一般而言，这些参数 应该小于一个极长的延迟，更远小于当前时间。

**7.5.4** 小心追踪 **ACE\_Event\_Handler** 的生存期

对登记到 ACE\_Reactor 上的 ACE\_Event\_Handler 的跟踪失败会导致各种问题。不使用像 Purify[24] 这样的内存错误检测工具，很难去追踪这些问题；但这样的工具也只能捕捉下面的一些、而不是全部的 与生存期相关的问题：

**7.5.4.1** 保守地使用非动态分配的事件处理器

考虑下面的具体事件处理器的定义：

class My\_Event\_Handler : public ACE\_Event\_Handler

{

public:

My\_Event\_Handler (const char \*str = "hello")

: str\_ (ACE\_OS::strnew (str)) {}

virtual int handle\_close

(ACE\_HANDLE = ACE\_INVALID\_HANDLE, ACE\_Reactor\_Mask = ACE\_Event\_Handler::READ\_MASK)

{

// Commit suicide. delete this;

}

~My\_Event\_Handler (void)

{

delete [] this->str\_;

}

// ...

private:

char \*str\_;

};

该类在从 ACE\_Reactor 上移除时，通过它的 handle\_close 清扫方法删除它自己。尽管这看起来有一点不太 传统，它却是完全有效的 C++习语。但是，它仅在正被删除的对象是动态分配的的情况下才能够工作。 相反，如果正被删除的对象不是动态分配的，全局动态内存堆将会被破坏。原因是 delete 操作符将

把 this 解释为堆中有效的地址。当 delete 操作符试图将非堆的内存插入它的内部空闲表时，就会造成微妙 的内存管理问题。

下面的例子演示一个常见的可导致堆崩溃的使用实例：

int main (void)

{

// Non-dynamically allocated. My\_Event\_Handler my\_event\_handler;

ACE\_Reactor::instance ()->register\_handler (&my\_event\_handler, ACE\_Event\_Handler::READ\_MASK);

// ...

// Run event-loop.

while (/\* ...event loop not finished... \*/) ACE\_Reactor::instance ()->handle\_events ();

// The <handle\_close> method deletes an

// object that wasn’t allocated dynamically...

ACE\_Reactor::instance ()->remove\_handler

(&my\_event\_handler, ACE\_Event\_Handler::READ\_MASK);

return 0;

}

上面代码的问题是 remove\_handler 被调用时，ACE\_Reactor 将会调用 My\_Event\_Handler 的 handle\_

close 方法。遗憾的是，handle\_close 方法会对 my\_event\_handler 对象执行 delete this 操作，而此对象并非 是动态分配的。

防止发生此问题的一种方法是将析构器放置在 My\_Event\_Handler 的私有区域，也就是：

class My\_Event\_Handler : public ACE\_Event\_Handler

{

public:

My\_Event\_Handler (const char \*str);

// ...

private:

// Place destructor into the private section

// to ensure dynamic allocation.

˜My\_Event\_Handler (void);

// ...

};

在此类中，My\_Event\_Handler 的析构器被放置在类的私有访问控制区中。这种 C++习语确保该类的所有 实例都必须是动态分配的。如果实例被偶然地定义为 static 或 auto，它在编译时就会被作为错误标记出来。

下面是实现具体事件处理器时，为最小化与具体事件处理器的生存期相关的问题，所应遵从的设计 准则：

设计准则 **6**：不要 *delete* 不是动态分配的事件处理器。任何含有 delete this、而其类又没有私有析构器的 handle\_close 方法，都有可能违反这一设计准则。在缺乏一种能够静态地识别这一情况的规约检查器时， 应该在 delete this 的紧前面加上注释，解释为何要使用这一习语。

**7.5.4.2** 适当地解除具体事件处理器的登记

下面的程序演示与具体事件处理器的生存期相关的另一种常见错误：

ACE\_Reactor reactor;

int main (void)

{

My\_Event\_Handler my\_event\_handler;

ACE\_Reactor::instance ()->register\_handler (&my\_event\_handler, ACE\_Event\_Handler::READ\_MASK);

while (/\* ...event loop not finished... \*/) ACE\_Reactor::instance ()->handle\_events ();

// The destructor of the ACE\_Reactor singleton

// will be called when the process exits. It

// removes all registered event handlers. return 0;

}

my\_event\_handler 的生存期由 main 函数的生存期决定。相反，ACE\_Reactor 单体的生存期由进程的生存

期决定。因而，当进程退出时，反应器的析构器将会被调用26。通过调用所有仍然登记在册的事件处理器 的 handle\_close 方法，ACE\_Reactor 的析构器将这些处理器全部移除掉。但是，如果 my\_event\_handler 仍 然登记在 Reactor 上，它的 handle\_close 方法将会在该对象出了作用域、并被销毁之后调用。

下面是实现具体事件处理器时，为最小化与具体事件处理器的生存期相关的问题，所应遵从的其他 三条设计准则：

设计准则 **7**：总是从堆中动态分配具体事件处理器。这是解决许多与具体处理器的生存期有关的问题的相 对直接的方法。如果不可能遵从此准则，必须在具体事件处理器登记到 ACE\_Reactor 时给出注释，解释 为什么不使用动态分配。该注释应该在将静态分配的具体处理器登记到 ACE\_Reactor 的 register\_handler 语句的紧前面出现。

设计准则 **8**：在 *ACE\_Event\_Handler* 退出它们“生活”的作用域之前，从与它们相关联的 *ACE\_Reactor*

中将它们移除掉。该准则应在未遵从准则 7 的情况下使用。

设计准则 **9**：只允许在 *handle\_close* 方法中使用 *delete this* 习语，也就是，不允许在其他 handle\_\* 方法中 使用 delete this。该准则有助于检查是否有与删除非动态分配的内存有关的潜在错误。自然，与 ACE\_Re actor 无关的组件可以拥有不同的对自删除进行管辖的准则。

设计准则 **10**：仅在为具体事件处理器所登记的最后一个事件已从 *ACE\_Reactor* 中移除时执行 *delete this* 操作。过早删除在 ACE\_Reactor 上登记了多个事件的具体处理器会导致“晃荡的指针”，遵从此准则可 以避免发生这样的情况。

例如，my\_event\_handler 可以登记 READ 和 WRITE 事件，如下所示：

ACE\_Reactor::instance ()->register\_handler (&my\_event\_handler, ACE\_Event\_Handler::READ\_MASK

| ACE\_Event\_Handler::WRITE\_MASK);

在此情形下，当 handle\_input 返回-1 时，ACE\_Reactor 将调用 handle\_close 清扫挂钩方法。在具体事

件处理器登记的 WRITE\_MASK 也被移除之前（例如，让它返回一个负值，或是通过下面的语句显式地 将它移除），该方法不能执行 delete this 操作。

ACE\_Reactor::instance ()->remove\_handler (&my\_event\_handler, ACE\_Event\_Handler::WRITE\_MASK);

下面的方法演示追踪此信息的一种途径：

class My\_Event\_Handler : public ACE\_Event\_Handler

{

public:

My\_Event\_Handler (void)

{

// Keep track of which bits are enabled. ACE\_SET\_BITS (this->mask\_,

ACE\_Event\_Handler::READ\_MASK

| ACE\_Event\_Handler::WRITE\_MASK);

// Register ourselves with the Reactor for

// both READ and WRITE events. ACE\_Reactor::instance ()->register\_handler (this, this->mask\_);

}

virtual int handle\_close (ACE\_HANDLE h, ACE\_Reactor\_Mask mask)

{

if (mask == ACE\_Event\_Handler::READ\_MASK)

{

ACE\_CLR\_BITS (this->mask\_, ACE\_Event\_Handler::READ\_MASK);

// Perform READ\_MASK cleanup logic.

}

else if (mask == ACE\_Event\_Handler::WRITE\_MASK)

{

ACE\_CLR\_BITS (this->mask\_, ACE\_Event\_Handler::WRITE\_MASK);

// Perform WRITE\_MASK cleanup logic.

}

// Only delete ourselves if we’ve been closed

// down for both READ and WRITE events. if (this->mask\_ == 0)

delete this;

}

// ... handle\_input() and handle\_output() methods.

private:

ACE\_Reactor\_Mask mask\_;

// Keep track of when to delete this.

};

上面的解决方案维护 ACE\_Reactor\_Mask，追踪何时一个具体事件处理器登记的所有事件已被从 ACE\_Re actor 移除。

**7.5.5** 注意 **WRITE\_MASK** 语义

下面的代码指示 ACE\_Reactor，只要可以无阻塞地向一个句柄写，就回调一个 event\_handler。

ACE\_Reactor::instance ()->mask\_ops (event\_handler, ACE\_Event\_Handler::WRITE\_MASK, ACE\_Reactor::ADD\_MASK);

但是，除非连接被流控制，否则总是可以向一个句柄写。因此，反应器会持续地回调 event\_handler 的 handle\_output 方法，直到（1）发生连接流控制或（2）mask\_ops 方法被指示清除 WRITE\_MASK。一 种常见的编程错误是忘记清除此掩码，导致 ACE\_Reactor 不断地调用 handle\_output 方法。应遵从下面的 设计准则来避免这一问题：

设计准则 **11**：当你不再需要具体事件处理器的 *handle\_output* 方法被回调时，清除 *WRITE\_MASK*。 下面的代码演示怎样确保 handle\_output 方法不再被回调：

ACE\_Reactor::instance ()->mask\_ops (event\_handler, ACE\_Event\_Handler::WRITE\_MASK, ACE\_Reactor::CLR\_MASK);

ACE\_Reactor 还定义了完成同样操作的简捷方法：

ACE\_Reactor::instance ()->cancel\_wakeup

(event\_handler,

ACE\_Event\_Handler::WRITE\_MASK);

这些方法通常在已不再有在具体事件处理器上待决的输出消息时被调用。 为帮助自动查验此准则，程序员必须在他们的 handle\_output 方法中插入注释，这些注释指示哪些返

回路径不会清除 WRITE\_MASK，也就是，事件处理器想要在“可以写”时继续被回调。同样地，程序员

还应该注释那些 WRITE\_MASK 被清除的路径。如果在 handle\_output 方法中没有路径清除 WRITE\_MAS K，那就意味着可能违反了此准则。

例如，下面的 handle\_output 方法演示此设计准则的可能的应用：

int My\_Event\_Handler::handle\_output (ACE\_HANDLE)

{

if (/\* output queue is now empty \*/)

{

ACE\_Reactor::instance ()->cancel\_wakeup (event\_handler, ACE\_Event\_Handler::WRITE\_MASK);

/\* Removing WRITE\_MASK \*/

return 0;

} else

{

// ... continue to transmit messages

// from the output queue.

/\* Not removing WRITE\_MASK \*/

return 0;

}

}

如果没有注释指示对 WRITE\_MASK 的清除，就有可能违反了此设计准则。

**7.5.6** 适当地登记具体事件处理器

当为 I/O 操作在 ACE\_Reactor 上登记具体事件处理器时，选择下面的方法中的一种： 显式地传递句柄：该方法使用下面的 ACE\_Reactor 方法：

int register\_handler (ACE\_HANDLE io\_handle, ACE\_Event\_Handler \*event\_handler, ACE\_Reactor\_Mask mask);

并显式地传递 I/O 设备的 ACE\_HANDLE，也就是：

void register\_socket (ACE\_HANDLE socket, ACE\_Event\_Handler \*handler)

{

ACE\_Reactor::instance ()->register\_handler

(socket, handler, ACE\_Event\_Handler::READ\_MASK);

// ...

}

注意此 register\_handler 方法允许同一个具体事件处理器与多个 ACE\_HANDLE 一起进行登记。反应器的

这一特性使得我们有可能最小化处理许多客户所需的状态的数量；这些客户同时与同一事件处理器相连 接。

隐式地传递句柄：该方法使用 ACE\_Reactor 的另一个 register\_handler 方法：

int register\_handler

(ACE\_Event\_Handler \*event\_handler, ACE\_Reactor\_Mask mask);

在这种情形下，ACE\_Reactor 执行一次“双重分派”（double-dispatch）[6]来通过具体事件处理器的 get\_ handle 方法从处理器中获取底层的 ACE\_HANDLE。该方法在 ACE\_Event\_Handler 基类中定义，具有特征 const：

virtual ACE\_HANDLE get\_handle (void) const;

当使用隐式登记时，常见的一种错误是在从 ACE\_Event\_Handler 派生子类时忽略了 get\_handle 上的 const。 这样的疏忽将导致编译器不能适当地在子类中重定义 get\_handle 方法。相反，它将在子类中隐藏该方法， 从而产生代码、调用基类的 get\_handle 方法；该方法缺省返回-1。

因此，服从下面的设计准则十分重要：

设计准则 **12**：确定 *get\_handle* 方法的特征与 *ACE\_Event\_Handler* 基类中的一致。如果你不遵从此准则， 并且你“隐式地”将 ACE\_HANDLE 传递给 ACE\_Reactor，ACE\_Event\_Handler 基类中的缺省 get\_handle 将返回-1，而这是错误的。

**7.5.7** 从反应器中移除已关闭的句柄**/**处理器

当连接被关闭时，句柄就不再能用于 I/O。在这样的情况下，select 将会持续地报告句柄“读就绪”，

这样你就可以在句柄上调用 close 了。此步骤通常在 handle\_close 清扫方法中完成。

一个常见的错误是对已死句柄及其事件处理器的移除的失败。这样 ACE\_Reactor 将会持续地回调事 件处理器的 handle\_input 方法，直到它被从 ACE\_Reactor 中移除。下面的设计准则有助于避免这一问题：

设计准则 **13**：当连接关闭时（或当连接上发生错误时），从 *handle\_\** 方法中返回一个负值。 遵从此设计准则的代码通常被构造如下：

int handle\_input (ACE\_HANDLE handle)

{

// ...

ssize\_t result = ACE\_OS::read (handle, buf, bufsize);

if (result <= 0)

// Connection has closed down or an

// error has occurred. return -1;

else

// ...

当返回-1 时，ACE\_Reactor 将调用你的 handle\_close 清扫方法。为避免资源泄漏，确定该方法给了事件处

理器以机会来删除它自己，并关闭它的句柄（例如，ACE\_OS::close (handle)）。一旦 handle\_close 返回，

ACE\_Reactor 就有机会从它的内部表中移除句柄/处理器对。

**7.5.8** 使用 **DONT\_CALL** 标志来避免递归的 **handle\_close()**回调

前面的准则描述了在应用显式或隐式地（通过从 handle\_\* 挂钩方法中返回负值）调用它的 remove\_ handler 时，ACE\_Reactor 怎样自动调用 handle\_close 方法。但是，如果应用在 handle\_close 清扫方法中调 用 remove\_handler，就必须特别小心，因为这有可能触发无限递归。下面的准则处理这一问题。

设计准则 **14**：在 *handle\_close* 方法中调用 *remove\_handler* 时，总是传递给它 *DONT\_CALL* 标志。该准则 确保 ACE\_Reactor 不会递归地调用 handle\_close 方法。下面的代码演示怎样应用此准则：

int My\_Event\_Handler::handle\_close

(ACE\_HANDLE, ACE\_Reactor\_Mask)

{

// ...

ACE\_Reactor::instance ()->remove\_handler

(this->get\_handle (),

// Remove all the events for which we’re

// registered. We must pass the DONT\_CALL

// flag here to avoid infinite recursion. ACE\_Event\_Handler::RWE\_MASK |

ACE\_Event\_Handler::DONT\_CALL);

// ...

}

顺便说一下，remove\_handler 通常在下列情况下在 handle\_close 中被调用：（1）为多个事件登记了同一

个具体事件处理器，以及（2）handle\_close 第一次被调用时需要触发事件处理器的完全关闭。因而，han dle\_close 还应该移除在 ACE\_Reactor 中与该事件处理器相关联的其他事件。

**7.6** 结束语

ACE\_Reactor 是设计用于简化并发的、事件驱动的分布式应用的 OO 框架。通过在 OO C++接口中封 装低级的 OS 事件多路分离机制，ACE\_Reactor 使得开发正确、简洁、可移植和高效的应用变得更为容易。 同样地，通过分离策略与机制，ACE\_Reactor 增强了复用、改善了可移植性，并提供了透明的可扩展性。

下面的 C++语言特性对 ACE\_Reactor 的设计和使用它的功能的应用有所帮助：

类：C++类提供的封装改善了可移植性。例如，ACE\_Reactor 类将应用与像 WaitForMultipleObjects 和 sel ect 这样的 OS 事件多路分离器之间的差异屏蔽开来。

对象：将 C++对象、而不是单独的函数登记到 ACE\_Reactor 有助于将应用特有的状态和使用此状态的方 法集成在一起。

继承和动态绑定：通过允许开发者不修改现有代码就增强 ACE\_Reactor 及与其相关联的应用的功能，这 些特性促进了透明的可扩展性。

模板：通过将可变性引入统一的类（它们可被“插入“到通用模板中），C++参数化类型有助于增强可复 用性。例如，除了 Logging\_Handler 和 ACE\_SOCK\_Acceptor，ACE\_Acceptor 还可用 SVC\_HANDLER 和 PEER\_ACCEPTOR 实例化。

使用 ACE\_Reactor 的一个潜在的不利方面是一开始很难理解应用的主线程控制在哪里执行。这是与 事件循环回调分派器（比如 ACE\_Reactor 或 X-windows）相关的一个常见问题。但是，在使用此方法编 写若干应用后，围绕这种“间接事件回调”分派模型的迷惑通常就会消失了。

ACE\_Reactor 和 ACE socket 包装的 C++源代码和文档可在 [http://www.cs.wustl.edu/~schmidt/ACE.ht](http://www.cs.wustl.edu/%7Eschmidt/ACE.ht) ml 找到。这一版本还包括一套测试程序和例子，以及许多其他封装命名管道、流管道、mmap 和系统 V IPC 机制（也就是，消息队列、共享内存和信号量）的 C++包装。

感谢

感谢西门子公司的 Hans Rohnert 和 Bill Landi 对本论文提出的有益意见。

参考文献

[1] D. C. Schmidt, “Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Disp atching,” in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, MA: Addison-Wesley, 1995.

[2] D. C. Schmidt, “ACE: an Object-Oriented Framework for Developing Distributed Applications,” in *Proceedings of the*

*6th USENIX C++ Technical Conferenc*e, (Cambridge, Massachusetts), USENIX Association, April 1994. [3] W.R.Stevens,*UNIX Network Programming, Second Editio*n. Englewood Cliffs, NJ: Prentice Hall, 1997. [4] H. Custer, *Inside Windows N*T. Redmond, Washington: Microsoft Press, 1993.

[5] D. C. Schmidt and C. Cleeland, “Applying Patterns to Develop Extensible ORB Middleware,” *Submitted to the IEEE Communications Magazin*e, 1998.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Softwar*e. Reading, MA: Addison-Wesley, 1995.

[7] D. C. Schmidt and P. Stephenson, “Experiences Using Design Patterns to Evolve System Software Across Diverse O S Platforms,” in *Proceedings of the 9th European Conference on Object-Oriented Programmin*g, (Aarhus, Denmark), AC M, August 1995.

[8] D. C. Schmidt, “Transparently Parameterizing Synchronization Mechanisms into a Concurrent Distributed Application,”

*C++ Repor*t, vol. 6, July/August 1994.

[9] I. Pyarali, T. H. Harrison, and D. C. Schmidt, “Asynchronous Completion Token: an Object Behavioral Pattern for E fficient Asynchronous Event Handling,” in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Rie hle, eds.), Reading, MA: Addison-Wesley, 1997.

[10] R. E. Barkley and T. P. Lee, “A Heap-Based Callout Implementation to Meet Real-Time Needs,” in *Proceedings of the USENIX Summer Conferenc*e, pp. 213–222, USENIX Association, June 1988.

[11] D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP Vol II: Design, Implementation, and Internal*s. Engle wood Cliffs, NJ: Prentice Hall, 1991.

[12] G. Varghese and T. Lauck, “Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementati on of a Timer Facility,” in *The Proceedings of the 11th Symposium on Operating System Principle*s, November 1987.

[13] J. Hu, S. Mungee, and D. C. Schmidt, “Principles for Developing and Measuring High-performance Web Servers ov er ATM,” i n *Proceeedings of INFOCOM ’9*8, March/April 1998.

[14] J. Hu, I. Pyarali, and D. C. Schmidt, “Measuring the Impact of Event Dispatching and Concurrency Models on We b Server Performance Over High-speed Networks,” in *Proceedings of the 2nd Global Internet Conferenc*e, IEEE, Novemb er 1997.

[15] D. C. Schmidt, D. L. Levine, and S. Mungee, “The Design and Performance of Real-Time Object Request Broker s,” *Computer Communication*s, vol. 21, pp. 294–324, Apr. 1998.

[16] D. C. Schmidt, “IPC SAP: An Object-Oriented Interface to Interprocess Communication Services,” *C++ Repor*t,vol.4, November/December 1992.

[17] D. L. Presotto and D. M. Ritchie, “Interprocess Communication in the Ninth Edition UNIX System,” *UNIX Researc h System Papers, Tenth Editio*n, vol. 2, no. 8, pp. 523–530, 1990.

[18] UNIX Software Operations, *UNIX System V Release 4 Programmer’s Guide: STREAM*S. Prentice Hall, 1990.

[19] G. Booch, *Object Oriented Analysis and Design with Applications 2nd Edition*). Redwood City, California: Benjamin/ Cummings, 1993.

[20] D. C. Schmidt, “Acceptor and Connector: Design Patterns for Initializing Communication Services,” in *Pattern Lang uages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[21] D. C. Schmidt and T. Suda, “Transport System Architecture Services for High-Performance Communications System s,” *IEEE Journal on Selected Areas in Communicatio*n, vol. 11, pp. 489–506, May 1993.

[22] A. Koenig, “When Not to Use Virtual Functions,” *C++ Journa*l, vol. 2, no. 2, 1992.

[23] T. H. Harrison, D. L. Levine, and D. C. Schmidt, “The Design and Performance of a Real-time CORBA Event Ser vice,” in *Proceedings of OOPSLA ’9*7, (Atlanta, GA), ACM, October 1997.

[24] P. S. Inc., *Purify User’s Guid*e. PureAtria Software Inc., 1996.

第 **8** 章 前摄器（**Proactor**）：用于为异步事件多路分离和分派处理

器的对象行为模式**27**

Irfan Pyarali Tim Harrison Douglas C. Schmidt Thomas D. Jordan

摘 要

现代操作系统为开发并发应用提供了多种机制。同步多线程是一种流行的机制，用于开发同时执行 多个操作的应用。但是，线程常常有很高的性能开销，并且需要对同步模式和原理有深入的了解。因此， 有越来越多的操作系统支持异步机制，在减少多线程的大量开销和复杂性的同时，提供了并发的好处。

本论文中介绍的前摄器（*Proactor*）模式描述怎样构造应用和系统，以有效地利用操作系统支持的异 步机制。当应用调用异步操作时，*OS* 代表应用执行此操作。这使得应用可以让多个操作同时运行，而又 不需要应用拥有相应数目的线程。因此，通过使用更少的线程和有效利用 *OS* 对异步操作的支持，前摄器 模式简化了并发编程，并改善了性能。

**8.1** 意图

前摄器模式支持多个事件处理器的多路分离和分派，这些处理器由异步事件的完成来触发。通过集 成完成事件（completion event）的多路分离和相应的事件处理器的分派，该模式简化了异步应用的开发。

**8.2** 动机

这一部分提供使用前摄器模式的上下文和动机。

**8.2.1** 上下文和压力

前摄器模式应该被用于应用需要并发执行操作的性能好处、又不想受到同步多线程或反应式编程的

27 此研究部分地得到了一项来自西门子 MED 的授权的支持。

约束时。为说明这些好处，设想一个需要并发执行多个操作的网络应用。例如，一个高性能 Web 服务器 必须并发处理发送自多个客户的 HTTP 请求[1, 2]。图 8-1 显示了 Web 浏览器和 Web 服务器之间的典型 交互。当用户指示浏览器打开一个 URL 时，浏览器发送一个 HTTP GET 请求给 Web 服务器。收到请求， 服务器就解析并校验请求，并将指定的文件发回给浏览器。

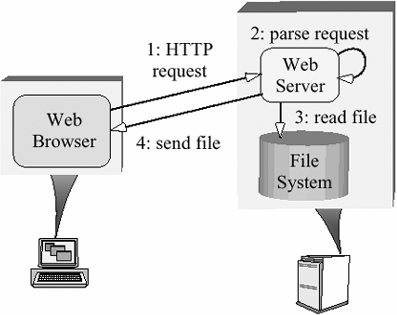


图 8-1 典型的 Web 服务器通信软件体系结构

开发高性能 Web 服务器要求消除以下压力：

z 并发性：服务器必须同时执行多个客户请求；

z 效率：服务器必须最小化响应延迟、最大化吞吐量，并避免不必要地使用 CPU；

z 编程简单性：服务器的设计应该简化高效的并发策略的使用；

z 可适配性：应该使继承新的或改进的传输协议（比如 HTTP 1.1[3]）所带来的维护代价最小化。

Web 服务器可以使用若干并发策略来实现，包括多个同步线程、反应式同步事件分派和前摄式异步 事件分派。下面，我们检查传统方法的缺点，并解释前摄器模式是怎样提供一种强大的技术，为高性能 并发应用而支持高效、灵活的异步事件分派策略的。

**8.2.2** 传统并发模型的常见陷阱和缺陷

同步的多线程和反应式编程是实现并发的常用方法。这一部分描述这些编程模型的缺点。

**8.2.2.1** 通过多个同步线程实现的并发

或许最为直观的实现并发 Web 服务器的途径是使用同步的多线程。在此模型中，多个服务器线程同 时处理来自多个客户的 HTTP GET 请求。每个线程同步地执行连接建立、HTTP 请求读取、请求解析和 文件传输操作。作为结果，每个操作都阻塞直到完成。

同步线程的主要优点是应用代码的简化。特别是，Web 服务器为服务客户 A 的请求所执行的操作在 很大程度上独立于为服务客户 B 的请求所需的操作。因而，很容易在分离的线程中对不同的请求进行服 务，因为在线程之间共享的状态数量很少；这也最小化了对同步的需要。而且，在分离的线程中执行应 用逻辑也使得开发者可以使用直观的顺序命令和阻塞操作。

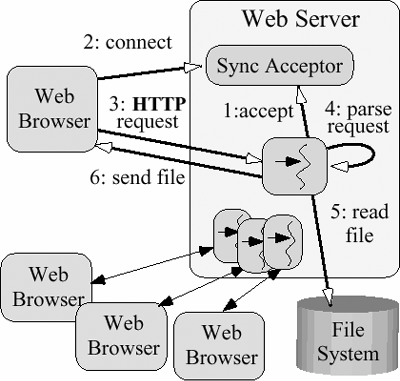


图 8-2 多线程 Web 服务器体系结构

图 8-2 显示使用同步线程来设计的 Web 服务器怎样并发地处理多个客户请求。该图显示的 Sync Acc eptor 对象封装服务器端用于同步接受网络连接的机制。使用“Thread Per Connection”并发模型，各个 线程为服务 HTTP GET 请求所执行的一系列步骤可被总结如下：

1. 每个线程同步地阻塞在 accept socket 调用中，等待客户连接请求；

2. 客户连接到服务器，连接被接受；

3. 新客户的 HTTP 请求被同步地从网络连接中读取；

4. 请求被解析；

5. 所请求的文件被同步地读取；

6. 文件被同步地发送给客户。

附录 A.1 中有一个将同步线程模型应用于 Web 服务器的 C++代码例子。

如上所述，每个并发地连接的客户由一个专用的服务器线程服务。在继续为其他 HTTP 请求服务之 前，该线程同步地完成一个被请求的操作。因此，要在服务多个客户时执行同步 I/O，Web 服务器必须派 生多个线程。尽管这种同步线程模式是直观的，且能够相对高效地映射到多 CPU 平台上，它还是有以下 缺点：

线程策略与并发策略被紧耦合：这种体系结构要求每个相连客户都有一个专用的线程。通过针对可用资 源（比如使用线程池来对应 CPU 的数目）、而不是正被并发服务的客户的数目来调整其线程策略，可能 会更好地优化一个并发应用；

更大的同步复杂性：线程可能会增加序列化对服务器的共享资源（比如缓存文件和 Web 页面点击日志） 的访问所必需的同步机制的复杂性；

更多的性能开销：由于上下文切换、同步和 CPU 间的数据移动[4]，线程的执行可能很低效； 不可移植性：线程有可能在有些平台上不可用。而且，根据对占先式和非占先式线程的支持，OS 平台之

间的差异非常大。因而，很难构建能够跨平台统一运作的多线程服务器。

作为这些缺点的结果，多线程常常不是开发并发 Web 服务器的最为高效的、也不是最不复杂的解决方案。

**8.2.2.2** 通过反应式同步事件分派实现的并发

另一种实现同步 Web 服务器的常用方法是使用反应式事件分派模型。反应器（Reactor）模式描述应 用怎样将 Event Handler 登记到 Initiation Dispatcher。Initiation Dispatcher 通知 Event Handler 何时能发起 一项操作而不阻塞。

单线程并发 Web 服务器可以使用反应式事件分派模型，它在一个事件循环中等待 Reactor 通知它发 起适当的操作。Web 服务器中反应式操作的一个例子是 Acceptor（接受器）[6]到 Initiation Dispatcher 的 登记。当数据在网络连接上到达时，分派器回调 Acceptor，后者接受网络连接，并创建 HTTP Handler。 于是这个 HTTP Handler 就登记到 Reactor，以在 Web 服务器的单线程控制中处理在那个连接上到来的 U RL 请求。

图 8-3 和图 8-4 显示使用反应式事件分派设计的 Web 服务器怎样处理多个客户。图 8-3 显示当客户连 接到 Web 服务器时所采取的步骤。图 8-4 显示 Web 服务器怎样处理客户请求。图 8-3 的一系列步骤可被 总结如下：

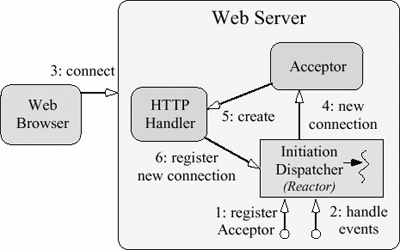


图 8-3 客户连接到反应式 Web 服务器

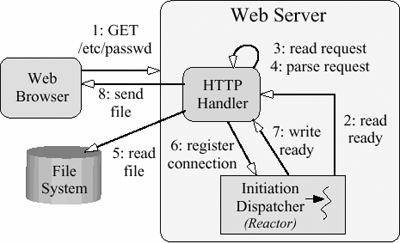


图 8-4 客户发送 HTTP 请求到反应式 Web 服务器

1. Web 服务器将 Acceptor 登记到 Initiation Dispatcher，以接受新连接；

2. Web 服务器调用 Initiation Dispatcher 的事件循环；

3. 客户连接到 Web 服务器；

4. Initiation Dispatcher 将新连接请求通知 Acceptor，后者接受新连接；

5. Acceptor 创建 HTTP Handler，以服务新客户；

6. HTTP Handler 将连接登记到 Initiation Dispatcher，以读取客户请求数据（就是说，在连接变得“读 就绪”时）；

7. HTTP Handler 服务来自新客户的请求。

图 8-4 显示反应式 Web 服务器为服务 HTTP GET 请求所采取的一系列步骤。该过程描述如下：

1. 客户发送 HTTP GET 请求；

2. 当客户请求数据到达服务器时，Initiation Dispatcher 通知 HTTP Handler；

3. 请求以非阻塞方式被读取，于是如果操作会导致调用线程阻塞，读操作就返回 EWOULDBLOCK（步 骤 2 和 3 将重复直到请求被完全读取）；

4. HTTP Handler 解析 HTTP 请求；

5. 所请求的文件从文件系统中被同步读取；

6. 为发送文件数据（就是说，当连接变得“写就绪”时），HTTP Handler 将连接登记到 Initiation Dispa tcher；

7. 当 TCP 连接变得写就绪时，Initiation Dispatcher 通知 HTTP Handler；

8. HTTP Handler 以非阻塞方式将所请求文件发送给客户，于是如果操作会导致调用线程阻塞，写操作 就返回 EWOULDBLOCK（步骤 7 和 8 将重复直到数据被完全递送）。

附录 A.2 中有一个将反应式事件分派模型应用于 Web 服务器的 C++代码例子。

因为 Initiation Dispatcher 运行在单线程中，网络 I/O 操作以非阻塞方式运行在 Reactor 的控制之下。 如果当前操作的进度停止了，操作就被转手给 Initiation Dispatcher，由它监控系统操作的状态。当操作可 以再度前进时，适当的 Event Handler 会被通知。

反应式模式的主要优点是可移植性，粗粒度并发控制带来的低开销（就是说，单线程不需要同步或 上下文切换），以及通过使应用逻辑与分派机制去耦合所获得的模块性。但是，该方法有以下缺点：

复杂的编程：如从前面的列表所看到的，程序员必须编写复杂的逻辑，以保证服务器不会在服务一个特 定客户时阻塞。

缺乏多线程的 **OS** 支持：大多数操作系统通过 select 系统调用[7]来实现反应式分派模型。但是，select 不 允许多于一个的线程在同一个描述符集上等待。这使得反应式模型不适用于高性能应用，因为它没有有 效地利用硬件的并行性。

可运行任务的调度：在支持占先式线程的同步多线程体系结构中，将可运行线程调度并时分（time-slice） 到可用 CPU 上是操作系统的责任。这样的调度支持在反应式体系结构中不可用，因为在应用中只有一个 线程。因此，系统的开发者必须小心地在所有连接到 Web 服务器的客户之间将线程分时。这只能通过执 行短持续时间、非阻塞的操作来完成。

作为这些缺点的结果，当硬件并行可用时，反应式事件分派不是最为高效的模型。由于需要避免使 用阻塞 I/O，该模式还有着相对较高的编程复杂度。

**8.2.3** 解决方案：通过前摄式操作实现的并发

当 OS 平台支持异步操作时，一种高效而方便的实现高性能 Web 服务器的方法是使用前摄式事件分 派。使用前摄式事件分派模型设计的 Web 服务器通过一或多个线程控制来处理异步操作的完成。这样， 通过集成完成事件多路分离（*completion event demultiplexing*）和事件处理器分派，前摄器模式简化了异 步的 *Web* 服务器。

异步的 Web 服务器将这样来利用前摄器模式：首先让 Web 服务器向 OS 发出异步操作，并将回调方 法登记到 Completion Dispatcher（完成分派器），后者将在操作完成时通知 Web 服务器。于是 OS 代表 W eb 服务器执行操作，并随即在一个周知的地方将结果排队。Completion Dispatcher 负责使完成通知出队，

并执行适当的、含有应用特有的 Web 服务器代码的回调。

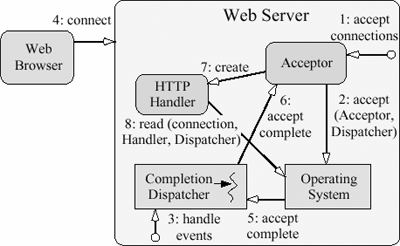


图 8-5 客户连接到基于前摄器的 Web 服务器

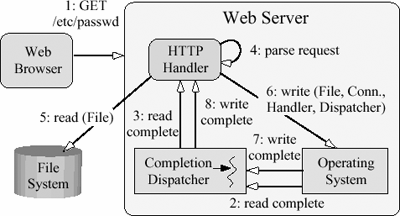


图 8-6 客户发送请求给基于前摄器的 Web 服务器

图 8-5 和图 8-6 显示使用前摄式事件分派设计的 Web 服务器怎样在一或多个线程中并发地处理多个 客户。图 8-5 显示当客户连接到 Web 服务器时所采取的一系列步骤。

1. Web 服务器指示 Acceptor 发起异步接受；

2. 接受器通过 OS 发起异步接受，将其自身作为 Completion Handler 和 Completion Dispatcher 的引用传 递；并将用于在异步接受完成时通知 Acceptor；

3. Web 服务器调用 Completion Dispatcher 的事件循环；

4. 客户连接到 Web 服务器；

5. 当异步接受操作完成时，操作系统通知 Completion Dispatcher；

6. Completion Dispatcher 通知接受器；

7. Acceptor 创建 HTTP Handler；

8. HTTP Handler 发起异步操作，以读取来自客户的请求数据，并将其自身作为 Completion Handler 和

Completion Dispatcher 的引用传递；并将用于在异步读取完成时通知 HTTP Handler。

图 8-6 显示前摄式 Web 服务器为服务 HTTP GET 请求所采取的步骤。这些步骤解释如下：

1. 客户发送 HTTP GET 请求；

2. 读取操作完成，操作系统通知 Completion Dispatcher；

3. Completion Dispatcher 通知 HTTP Handler（步骤 2 和 3 将重复直到整个请求被接收）；

4. HTTP Handler 解析请求；

5. HTTP Handler 同步地读取所请求的文件；

6. HTTP Handler 发起异步操作，以把文件数据写到客户连接，并将其自身作为 Completion Handler 和

Completion Dispatcher 的引用传递；并将用于在异步写入完成时通知 HTTP Handler。

7. 当写操作完成时，操作系统通知 Completion Dispatcher；

8. 随后 Completion Dispatcher 通知 Completion Handler（步骤 6-8 将重复直到文件被完全递送）。

8.8 中有一个将前摄式事件分派模型应用于 Web 服务器的 C++代码例子。 使用前摄器模式的主要优点是可以启动多个并发操作，并可并行运行，而不要求应用必须拥有多个

线程。操作被应用异步地启动，它们在 OS 的 I/O 子系统中运行直到完成。发起操作的线程现在可以服务 另外的请求了。

例如，在上面的例子中，Completion Dispatcher 可以是单线程的。当 HTTP 请求到达时，单个 Comp letion Dispatcher 线程解析请求，读取文件，并发送响应给客户。因为响应是被异步发送的，多个响应就 有可能同时被发送。而且，同步的文件读取可以被异步的文件读取取代，以进一步增加并发的潜力。如 果文件读取是被异步完成的，HTTP Handler 所执行的唯一的同步操作就只剩下了 HTTP 协议请求解析。

前摄式模型的主要缺点是编程逻辑至少和反应式模型一样复杂。而且，前摄器模式可能会难以调试， 因为异步操作常常有着不可预测和不可重复的执行序列，这就使分析和调试复杂化了。8.7 描述怎样应用 其他模式（比如异步完成令牌[8]）来简化异步应用编程模型。

**8.3** 适用性

当具有以下一项或多项条件时使用前摄器模式：

z 应用需要执行一个或多个不阻塞调用线程的异步操作；

z 当异步操作完成时应用必须被通知；

z 应用需要独立于它的 I/O 模型改变它的并发策略；

z 通过使依赖于应用的逻辑与应用无关的底层构造去耦合，应用将从中获益；

z 当使用多线程方法或反应式分派方法时，应用的执行将很低效，或是不能满足性能需求。

**8.4** 结构和参与者

在图 8-7 中使用 OMT 表示法演示了前摄器模式的结构。

前摄器模式中的关键参与者包括：

前摄发起器（Proactive Initiator。Web 服务器应用的主线程）：

z Proactive Initiator 是应用中任何发起 Asynchronous Operation（异步操作）的实体。它将 Completion Handler 和 Completion Dispatcher 登记到 Asynchronous Operation Processor（异步操作处理器），此处 理器在操作完成时通知前摄发起器。

完成处理器（Completion Handler。Acceptor 和 HTTP Handler）：

z 前摄器模式将应用所实现的 Completion Handler 接口用于 Asynchronous Operation 完成通知。 异步操作（Asynchronous Operation。Async\_Read、Async\_Write 和 Async\_Accept 方法）：

z Asynchronous Operation 被用于代表应用执行请求（比如 I/O 和定时器操作）。当应用调用 Asynchron ous Operation 时，操作的执行没有借用应用的线程控制28。因此，从应用的角度来看，操作是被异步 地执行的。当 Asynchronous Operation 完成时，Asynchronous Operation Processor 将应用通知委托给 Completion Dispatcher。

异步操作处理器（Asynchronous Operation Processor。操作系统）：

z Asynchronous Operation 是由 Asynchronous Operation Processor 来运行直至完成的。该组件通常由 O S 实现。

完成分派器（Completion Dispatcher。Notification Queue）：

z Completion Dispatcher 负责在 Asynchronous Operation 完成时回调应用的 Completion Handler。当 As ynchronous Operation Processor 完成异步发起的操作时，Completion Dispatcher 代表应用执行应用回 调。

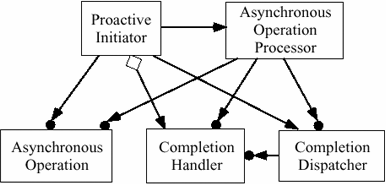


图 8-7 前摄器模式中的参与者

28 相反，反应式事件分派模型[5]偷取应用的线程控制来同步地执行操作。

**8.5** 协作

有若干良好定义的步骤被用于所有 Asynchronous Operation。在高水平的抽象上，应用异步地发起操 作，并在操作完成时被通知。图 8-8 显示在模式参与者之间必定发生的下列交互：

**1.** 前摄发起器发起操作：为执行异步操作，应用在 Asynchronous Operation Processor 上发起操作。例 如，Web 服务器可能要求 OS 在网络上使用特定的 socket 连接传输文件。要请求这样的操作，Web 服 务器必须指定要使用哪一个文件和网络连接。而且，Web 服务器必须指定（1）当操作完成时通知哪 一个 Completion Handler，以及（2）一旦文件被传输，哪一个 Completion Dispatcher 应该执行回调。

**2.** 异步操作处理器执行操作：当应用在 Asynchronous Operation Processor 上调用操作时，它相对于其 他应用操作异步地运行这些操作。现代操作系统（比如 Solaris 和 Windows NT）在内核中提供异步 的 I/O 子系统。

**3.** 异步操作处理器通知完成分派器：当操作完成时，Asynchronous Operation Processor 取得在操作被发 起时指定的 Completion Handler 和 Completion Dispatcher。随后 Asynchronous Operation Processor 将 Asynchronous Operation 的结果和 Completion Handler 传递给 Completion Dispatcher，以用于回调。 例如，如果文件已被异步传输，Asynchronous Operation Processor 可以报告完成状态（比如成功或失 败），以及写入网络连接的字节数。

**4.** 完成分派器通知应用：Completion Dispatcher 在 Completion Handler 上调用完成挂钩，将由应用指定 的任何完成数据传递给它。例如，如果异步读取完成，通常一个指向新到达数据的指针将会被传递给 Completion Handler。

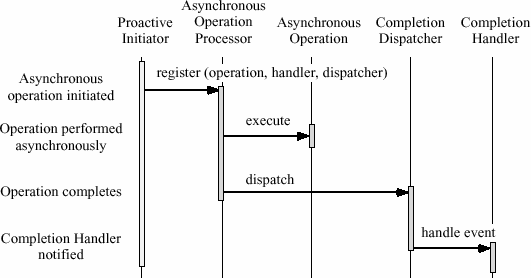


图 8-8 前摄器模式的交互图

**8.6** 效果

这一部分详述使用前摄器模式的效果。

**8.6.1** 好处

前摄器模式提供以下好处： 增强事务分离：前摄器模式使应用无关的异步机制与应用特有的功能去耦合。应用无关的机制成为可复

用组件，知道怎样多路分离与 Asynchronous Operation 相关联的完成事件，并分派适当的由 Completion

Handler 定义的回调方法。同样地，应用特有的功能知道怎样执行特定类型的服务（比如 HTTP 处理）。

改善应用逻辑可移植性：通过允许接口独立于执行事件多路分离的底层 OS 调用而复用，它改善了应用的 可移植性。这些系统调用检测并报告可能同时发生在多个事件源之上的事件。事件源可以是 I/O 端口、定 时器、同步对象、信号，等等。在实时 POSIX 平台上，异步 I/O 函数由 aio API 族[9]提供。在 Windows NT 中，I/O 完成端口和重叠式（overlapped）I/O 被用于实现异步 I/O[10]。

完成分派器封装了并发机制：使 Completion Dispatcher 与 Asynchronous Operation Processor 去耦合的一 个好处是应用可以通过多种并发策略来配置 Completion Dispatcher，而不会影响其他参与者。如 8.7 所讨 论的，Completion Dispatcher 可被配置使用包括单线程和线程池方案在内的若干并发策略。

线程策略被与并发策略去耦合：因为 Asynchronous Operation Processor 代表 Proactive Initiator 完成可能 长时间运行的操作，应用不会被迫派生线程来增加并发。这使得应用可以独立于它的线程策略改变它的 并发策略。例如，Web 服务器可能只想每个 CPU 有一个线程，但又想同时服务更多数目的客户。

提高性能：多线程操作系统执行上下文切换，以在多个线程控制中轮换。虽然执行一次上下文切换的时 间保持相当的恒定，如果 OS 上下文要切换到空闲线程的话，在大量线程间轮换的总时间可以显著地降低 应用性能**29**。例如，线程可以轮询 OS 以查看完成状态，而这是低效率的。通过只激活那些有事件要处理 的合理的线程控制，前摄器模式能够避免上下文切换的代价。例如，如果没有待处理的 GET 请求，Web 服务器不需要启用 HTTP Handler。

应用同步的简化：只要 Completion Handler 不派生另外的线程控制，可以不考虑、或只考虑少许同步问 题而编写应用逻辑。Completion Handler 可被编写为就好像它们存在于一个传统的单线程环境中一样。例 如，Web 服务器的 HTTP GET 处理器可以通过 Async Read 操作（比如 Windows NT TransmitFile 函数[1]） 来访问磁盘。

**8.6.2** 缺点

前摄器模式有以下缺点：

29 某些老操作系统显示出这样的行为，大多数现代操作系统则没有。

难以调试：以前摄器模式编写的应用可能难以调试，因为反向的控制流在框架基础结构和应用特有的处

理器上的回调方法之间来回振荡。这增加了在调试器中对框架的运行时行为的“单步跟踪”的困难度， 因为应用开发者可能不了解或不能获得框架的代码。这与试图调试使用 LEX 和 YACC 编写的编译器的词 法分析器和解析器时所遇到的问题是类似的。在这些应用中，当线程控制是在用户定义的动作例程中时， 调试是相当直接的。但是一旦线程控制返回到所生成的有限确定自动机（Deterministic Finite Automate， DFA）骨架时，就很难跟住程序逻辑了。

调度和控制未完成操作：Proactive Initiator 可能没有对 Asynchronous Operation 的执行顺序的控制。因此，

Asynchronous Operation Processor 必须被小心设计，以支持 Asynchronous Operation 的优先级和取消处理。

**8.7** 实现

前摄器模式可以通过许多方式实现。这一部分讨论实现前摄器模式所涉及的步骤。

**8.7.1** 实现异步操作处理器

实现前摄器模式的第一步是构建 Asynchronous Operation Processor。该组件负责代表应用异步地执 行操作。因此，它的两项主要责任是输出 Asynchronous Operation API 和实现 Asynchronous Operation E ngine 以完成工作。

**8.7.1.1** 定义异步操作 **API**

Asynchronous Operation Processor 必须提供 API、允许应用请求 Asynchronous Operation。在设计这 些 API 时有若干压力需要考虑：

可移植性：此 API 不应约束应用或它的 Proactive Initiator 使用特定的平台。

灵活性：常常，异步 API 可以为许多类型的操作共享。例如，异步 I/O 操作常常被用于在多种介质（比 如网络和文件）上执行 I/O。设计支持这样的复用的 API 可能是有益的。

回调：当操作被调用时，Proactive Initiator 必须登记回调。实现回调的一种常用方法是让调用对象（客户） 输出接口、让调用者知道（服务器）。因此，Proactive Initiator 必须通知 Asynchronous Operation Process or，当操作完成时，哪一个 Completion Handler 应被回调。

完成分派器：因为应用可以使用多个 Completion Dispatcher，Proactive Initiator 还必须指示由哪一个 Co

mpletion Dispatcher 来执行回调。

给定所有这些问题，考虑下面的用于异步读写的 API。Asynch\_Stream 类是用于发起异步读写的工厂。 一旦构造，可以使用此类来启动多个异步读写。当异步读取完成时，Asynch\_Stream::Read\_Result 将通过 Completion\_Handler 上的 handler\_read 回调方法被回传给 handler。类似地，当异步写入完成时，Asynch\_ Stream::Write\_Result 将通过 Completion\_Handler 上的 handler\_write 回调方法被回传给 handler。

class Asynch\_Stream

// = TITLE

// A Factory for initiating reads

// and writes asynchronously.

{

// Initializes the factory with information

// which will be used with each asynchronous

// call. <handler> is notified when the

// operation completes. The asynchronous

// operations are performed on the <handle>

// and the results of the operations are

// sent to the <Completion\_Dispatcher>. Asynch\_Stream (Completion\_Handler &handler,

HANDLE handle, Completion\_Dispatcher \*);

// This starts off an asynchronous read.

// Upto <bytes\_to\_read> will be read and

// stored in the <message\_block>.

int read (Message\_Block &message\_block, u\_long bytes\_to\_read,

const void \*act = 0);

// This starts off an asynchronous write.

// Upto <bytes\_to\_write> will be written

// from the <message\_block>.

int write (Message\_Block &message\_block, u\_long bytes\_to\_write,

const void \*act = 0);

...

};

**8.7.1.2** 实现异步操作引擎

Asynchronous Operation Processor 必须含有异步执行操作的机制。换句话说，当应用线程调用 Asyn chronous Operation 时，必须不借用应用的线程控制而执行此操作。幸好，现代操作系统提供了用于 Asy nchronous Operation 的机制（例如，POSIX 异步 I/O 和 WinNT 重叠式 I/O）。在这样的情况下，实现模式 的这一部分只需要简单地将平台 API 映射到上面描述的 Asynchronous Operation API。

如果 OS 平台不提供对 Asynchronous Operation 的支持，有若干实现技术可用于构建 Asynchronous Operation Engine。或许最为直观的解决方案是使用专用线程来为应用执行 Asynchronous Operation。要 实现线程化的 Asynchronous Operation Engine，有三个主要步骤：

**1.** 操作调用：因为操作将在与进行调用的应用线程不同的线程控制中执行，必定会发生某种类型的线程 同步。一种方法是为每个操作派生一个线程。更为常用的方法是为 Asynchronous Operation Processo r 而管理一个专用线程池。该方法可能需要应用线程在继续进行其他应用计算之前将操作请求排队。

**2.** 操作执行：既然操作将在专用线程中执行，所以它可以执行“阻塞”操作，而不会直接阻碍应用的进 展。例如，在提供异步 I/O 读取机制时，专用线程可以在从 socket 或文件句柄中读时阻塞。

**3.** 操作完成：当操作完成时，应用必须被通知到。特别是，专用线程必须将应用特有的通知委托给 Co mpletion Dispatcher。这要求在线程间进行另外的同步。

**8.7.2** 实现完成分派器

当 Completion Dispatcher 从 Asynchronous Operation Processor 接收到操作完成通知时，它会回调与 应用对象相关联的 Completion Handler。实现 Completion Dispatcher 涉及两个问题：（1）实现回调以及（2） 定义用于执行回调的并发策略。

**8.7.2.1** 实现回调

Completion Dispatcher 必须实现一种机制，Completion Handler 通过它被调用。这要求 Proactive Init iator 在发起操作时指定一个回调。下面是常用的回调可选方案：

回调类：Completion Handler 输出接口、让 Completion Dispatcher 知道。当操作完成时，Completion Dis patcher 回调此接口中的方法，并将已完成操作的有关信息传递给它（比如从网络连接中读取的字节数）。

函数指针：Completion Dispatcher 通过回调函数指针来调用 Completion Handler。该方法有效地打破了 C

ompletion Dispatcher 和 Completion Handler 之间的知识依赖。这有两个好处：

1. Completion Handler 不会被迫输出特定的接口；以及

2. 在 Completion Dispatcher 和 Completion Handler 之间不需要有编译时依赖。

会合点：Proactive Initiator 可以设立事件对象或条件变量，用作 Completion Dispatcher 和 Completion Ha

ndler 之间的会合点。这在 Completion Handler 是 Proactive Initiator 时最为常见。在 Asynchronous Operat ion 运行至完成的同时，Completion Handler 处理其他的活动。Completion Handler 将在会合点周期性地

检查完成状态。

**8.7.2.2** 定义完成分派器并发策略

当操作完成时，Asynchronous Operation Processor 将会通知 Completion Dispatcher。在这时，Compl etion Dispatcher 可以利用下面的并发策略中的一种来执行应用回调：

动态线程分派：Completion Dispatcher 可为每个 Completion Handler 动态分配一个线程。动态线程分派可 通过大多数多线程操作系统来实现。在有些平台上，由于创建和销毁线程资源的开销，这可能是所列出 的 Completion Dispatcher 实现技术中最为低效的一种，

后反应式分派（**Post-reactive dispatching**）：Completion Dispatcher 可以发信号给 Proactive Initiation 所设 立的事件对象或条件变量。尽管轮询和派生阻塞在事件对象上的子线程都是可选的方案，最为高效的后 反应式分派方法是将事件登记到 Reactor。后反应式分派可以通过 POSIX 实时环境中的 aio\_suspend 和 Wi n32 环境中的 WaitForMultipleObjects 来实现。

**Call-through** 分派：来自 Asynchronous Operation Processor 的线程控制可被 Completion Dispatcher 借用， 以执行 Completion Handler。这种“周期偷取”策略可以通过减少空闲线程的影响范围来提高性能。在一 些老操作系统会将上下文切换到空闲线程、又只是从它们切换出去的情况下，这种方法有着收回“失去 的”时间的巨大潜力。

Call-through 分派在 Windows NT 中可以使用 ReadFileEx 和 WriteFileEx Win32 函数来实现。例如， 线程控制可以使用这些调用来等待信号量被置位。当它等待时，线程通知 OS 它进入了一种称为“可报警 等待状态”（alterable wait state）的特殊状态。在这时，OS 可以占有对等待中的线程控制的栈和相关资 源的控制，以执行 Completion Handler。

线程池分派：由 Completion Dispatcher 拥有的线程池可被用于 Completion Handler 的执行。在池中的每 个线程控制已被动态地分配到可用的 CPU。线程池分派可通过 Windows NT 的 I/O 完成端口来实现。

在考虑上面描述的 Completion Dispatcher 技术的适用性时，考虑表 8-1 中所示的 OS 环境和物理硬件 的可能组合：

|  |  |  |
| --- | --- | --- |
| 线程模型 | 系统类型 | |
| 单处理器 | 多处理器 |
| 单线程 | A | B |
| 多线程 | C | D |

表 8-1 Completion Dispatcher 并发策略

如果你的 OS 只支持同步 I/O，那就参见反应器模式[5]。但是，大多数现代操作系统都支持某种类型的异 步 I/O。

在表 8-1 的 A 和 B 组合中，假定你不等待任何信号量或互斥体，后反应方式的异步 I/O 很可能是最 好的。否则，Call-through 实现或许更能回应你的问题。在 C 组合中，使用 Call-through 方法。在 D 组合

中，使用线程池方法。在实践中，系统化的经验测量对于选择最为合适的可选方案来说是必需的。

**8.7.3** 实现完成处理器

Completion Handler 的实现带来以下考虑。

**8.7.3.1** 状态完整性

Completion Handler 可能需要维护关于特定请求的状态信息。例如，OS 可以通知 Web 服务器，只有 一部分文件已被写到网络通信端口。作为结果，Completion Handler 可能需要重新发出请求，直到文件被 完全写出，或连接变得无效。因此，它必须知道原先指定的文件，还剩多少字节要写，以及在前一个请 求开始时文件指针的位置。

没有隐含的限制来阻止 Proactive Initiator 将多个 Asynchronous Operation 请求分配给单个 Completio n Handler。因此，Completion Handler 必须在完成通知链中一一“系上”请求特有的状态信息。为完成 此工作，Completion Handler 可以利用异步完成令牌（Asynchronous Completion Token）模式[8]。

**8.7.3.2** 资源管理

与在任何多线程环境中一样，使用前摄器模式的 Completion Handler 还是要由它自己来确保对共享 资源的访问是线程安全的。但是，Completion Handler 不能跨越多个完成通知持有共享资源。否则，就有 发生“用餐哲学家问题”的危险[11]。

该问题在于一个合理的线程控制永久等待一个信号量被置位时所产生的死锁。通过设想一个由一群 哲学家出席的宴会可以演示这一问题。用餐者围绕一个圆桌就座，在每个哲学家之间只有一支筷子。当 哲学家觉得饥饿时，他必须获取在他左边和在他右边的筷子才能用餐。一旦哲学家获得一支筷子，不到 吃饱他们就不会放下它。如果所有哲学家都拿起在他们右边的筷子，就会发生死锁，因为他们将永远也 不可能拿到左边的筷子。

**8.7.3.3** 占先式策略（**Preemptive Policy**）

Completion Dispatcher 类型决定在执行时一个 Completion Handler 是否可占先。当与动态线程和线程 池分派器相连时，Completion Handler 自然可占先。但是，当与后反应式 Completion Dispatcher 相连时， Completion Handler 并没有对其他 Completion Handler 的占先权。当由 Call-through 分派器驱动时，Comp letion Handler 相对于在可报警等待状态的线程控制也没有占先权。

一般而言，处理器不应该执行持续时间长的同步操作，除非使用了多个完成线程，因为应用的总体 响应性将会被显著地降低。这样的危险可以通过增强的编程训练来降低。例如，所有 Completion Handle r 被要求用作 Proactive Initiator，而不是去执行同步操作。

**8.8** 示例代码

这一部分显示怎样使用前摄器模式来开发 Web 服务器。该例子基于 ACE 框架[4]中的前摄器实现。 当客户连接到 Web 服务器时，HTTP\_Handler 的 open 方法被调用。于是服务器就通过在 Asynchrono

us Operation 完成时回调的对象（在此例中是 this 指针）、用于传输数据的网络连接，以及一旦操作完成 时使用的 Completion Dispatcher（proactor\_）来初始化异步 I/O 对象。随后读操作异步地启动，而服务器 返回事件循环。

当 Async read 操作完成时，分派器回调 HTTP\_Handler::handle\_read\_stream。如果有足够的数据，客 户请求就被解析。如果整个客户请求还未完全到达，另一个读操作就会被异步地发起。

在对 GET 请求的响应中，服务器对所请求文件进行内存映射，并将文件数据异步地写往客户。当写 操作完成时，分派器回调 HTTP\_Handler::handle\_write\_stream，从而释放动态分配的资源。

附录中含有两个其他的代码实例，使用同步的线程模型和同步的（非阻塞）反应式模型实现 Web 服 务器。

class HTTP\_Handler

: public Proactor::Event\_Handler

// = TITLE

// Implements the HTTP protocol

// (asynchronous version).

//

// = PATTERN PARTICIPANTS

// Proactive Initiator = HTTP\_Handler

// Asynch Op = Network I/O

// Asynch Op Processor = OS

// Completion Dispatcher = Proactor

// Completion Handler = HTPP\_Handler

{

public:

void open (Socket\_Stream \*client)

{

// Initialize state for request request\_.state\_ = INCOMPLETE;

// Store reference to client. client\_ = client;

// Initialize asynch read stream

stream\_.open (\*this, client\_->handle (), proactor\_);

// Start read asynchronously. stream\_.read (request\_.buffer (), request\_.buffer\_size ());

}

// This is called by the Proactor

// when the asynch read completes

void handle\_read\_stream(u\_long bytes\_transferred)

{

if (request\_.enough\_data(bytes\_transferred))

parse\_request ();

else

// Start reading asynchronously. stream\_.read (request\_.buffer (),

request\_.buffer\_size ());

}

void parse\_request (void)

{

// Switch on the HTTP command type. switch (request\_.command ())

{

// Client is requesting a file. case HTTP\_Request::GET:

// Memory map the requested file. file\_.map (request\_.filename ());

// Start writing asynchronously.

stream\_.write (file\_.buffer (), file\_.buffer\_size ());

break;

// Client is storing a file

// at the server.

case HTTP\_Request::PUT:

// ...

}

}

void handle\_write\_stream(u\_long bytes\_transferred)

{

if (file\_.enough\_data(bytes\_transferred))

// Success....

else

// Start another asynchronous write

stream\_.write (file\_.buffer (), file\_.buffer\_size ());

}

private:

// Set at initialization. Proactor \*proactor\_;

// Memory-mapped file\_; Mem\_Map file\_;

// Socket endpoint. Socket\_Stream \*client\_;

// HTTP Request holder

HTTP\_Request request\_;

// Used for Asynch I/O Asynch\_Stream stream\_;

};

**8.9** 已知应用

下面是一些被广泛记载的前摄器的使用：

**Windows NT** 中的 **I/O** 完成端口：Windows NT 操作系统实现了前摄器模式。Windows NT 支持多种 As ynchronous Operation，比如接受新网络连接、读写文件和 socket，以及通过网络连接传输文件。操作系 统就是 Asynchronous Operation Processor。操作结果在 I/O 完成端口（它扮演 Completion Dispatcher 的角 色）上排队。

异步 **I/O** 操作的 **UNIX AIO** 族：在有些实时 POSIX 平台上，前摄器模式是由 aio API 族[9]来实现的。这 些 OS 特性非常类似于上面描述的 Windows NT 的特性。一个区别是 UNIX 信号可用于实现真正异步的 C ompletion Dispatcher（Windows NT API 不是真正异步的）。

**ACE Proactor**：ACE 自适配通信环境 [4]实现了前摄器组件，它封装 Windows NT 上的 I/O 完成端口， 以及 POSIX 平台上的 aio API。ACE 前摄器抽象提供 Windows NT 所支持的标准 C API 的 OO 接口。这 一实现的源码可从 ACE 网站 [http://www.cs.wustl.edu/~schmidt/ACE.html](http://www.cs.wustl.edu/%7Eschmidt/ACE.html) 获取。

**Windows NT** 中的异步过程调用（**Asynchronous Procedure Call**）：有些系统（比如 Windows NT）支持

异步过程调用（APC）。APC 是在特定线程的上下文中异步执行的函数。当 APC 被排队到线程时，系统 发出软件中断。下一次线程被调度时，它将运行该 APC。操作系统所发出的 APC 被称为内核模式 APC。

应用所发出的 APC 被称为用户模式 APC。

**8.10** 相关模式

图 8-9 演示与前摄器相关的模式。

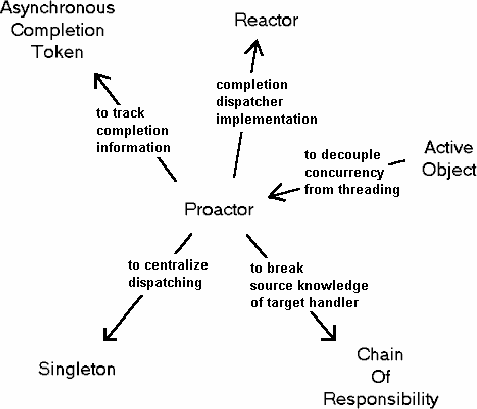


图 8-9 前摄器模式的相关模式

异步完成令牌（ACT）模式[8]通常与前摄器模式结合使用。当 Asynchronous Operation 完成时，应 用可能需要比简单的通知更多的信息来适当地处理事件。异步完成令牌模式允许应用将状态高效地与 As ynchronous Operation 的完成相关联。

前摄器模式还与观察者（Observer）模式[12]（在其中，当单个主题变动时，相关对象也会自动更新） 有关。在前摄器模式中，当来自多个来源的事件发生时，处理器被自动地通知。一般而言，前摄器模式 被用于异步地将多个输入源多路分离给与它们相关联的事件处理器，而观察者通常仅与单个事件源相关 联。

前摄器模式可被认为是同步反应器模式[5]的一种异步的变体。反应器模式负责多个事件处理器的多 路分离和分派；它们在可以同步地发起操作而不会阻塞时被触发。相反，前摄器模式也支持多个事件处 理器的多路分离和分派，但它们是被异步事件的完成触发的。

主动对象（Active Object）模式[13]使方法执行与方法调用去耦合。前摄器模式也是类似的，因为 A synchronous Operation Processor 代表应用的 Proactive Initiator 来执行操作。就是说，两种模式都可用于 实现 Asynchronous Operation。前摄器模式常常用于替代主动对象模式，以使系统并发策略与线程模型去

耦合。 前摄器可被实现为单体（Singleton）[12]。这对于在异步应用中，将事件多路分离和完成分派集中到

单一的地方来说是有用的。

责任链（Chain of Responsibility，COR）模式[12]使事件处理器与事件源去耦合。在 Proactive Initia tor 与 Completion Handler 的隔离上，前摄器模式也是类似的。但是，在 COR 中，事件源预先不知道哪 一个处理器将被执行（如果有的话）。在前摄器中，Proactive Initiator 完全知道目标处理器。但是，通过

建立一个 Completion Handle（r

它是由外部工厂动态配置的责任链的入口），这两种模式可被结合在一起：。

**8.11** 结束语

前摄器模式包含了一种强大的设计范式，支持高性能并发应用的高效而灵活的事件分派策略。前摄 器模式提供并发执行操作的性能助益，而又不强迫开发者使用同步多线程或反应式编程。

参考文献

[1] J. Hu, I. Pyarali, and D. C. Schmidt, “Measuring the Impact of Event Dispatching and Concurrency Models on Web

Server Performance Over High-speed Networks,” in *Proceedings of the 2nd Global Internet Conferenc*e, IEEE, November

1997.

[2] J. Hu, I. Pyarali, and D. C. Schmidt, “Applying the Proactor Pattern to High-Performance Web Servers,” in *Proceedi ngs of the 10th International Conference on Parallel and Distributed Computing and System*s, IASTED, Oct. 1998.

[3] J. C. Mogul, “The Case for Persistent-connection HTTP,” in *Proceedings of ACMSIGCOMM ’95 Conference in Com puter Communication Revie*w, (Boston, MA, USA), pp. 299–314, ACM Press, August 1995.

[4] D. C. Schmidt, “ACE: an Object-Oriented Framework for Developing Distributed Applications,” in *Proceedings of the*

*6th USENIX C++ Technical Conferenc*e, (Cambridge, Massachusetts), USENIX Association, April 1994.

[5] D. C. Schmidt, “Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Disp atching,” in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, MA: Addison-Wesley, 1995.

[6] D. C. Schmidt, “Acceptor and Connector: Design Patterns for Initializing Communication Services,” in *Pattern Langu ages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[7] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4BSD Ope rating Syste*m. Addison Wesley, 1996.

[8] I. Pyarali, T. H. Harrison, and D. C. Schmidt, “Asynchronous Completion Token: an Object Behavioral Pattern for E fficient Asynchronous Event Handling,” in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Rie hle, eds.), Reading, MA: Addison-Wesley, 1997.

[9] “Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application: Program Inter face (API) [C Language],” 1995.

[10] *Microsoft Developers Studio, Version 4.2 - Software Development Ki*t, 1996.

[11] E. W. Dijkstra, “Hierarchical Ordering of Sequential Processes,” *Acta Informatic*a, vol. 1, no. 2, pp. 115–138, 1971. [12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Softwar*e.

Reading, MA: Addison-Wesley, 1995.

[13] R. G. Lavender and D. C. Schmidt, “Active Object: an Object Behavioral Pattern for Concurrent Programming,” in *Proceedings of the 2nd Annual Conference on the Pattern Languages of Program*s, (Monticello, Illinois), pp. 1–7, Septem ber 1995.

附录 **A** 可选实现

本附录概述用于开发前摄器模式的可选实现的代码。下面，我们检查使用多线程的同步 I/O 和使用单 线程的反应式 I/O。

**A.1** 多个同步线程

下面的代码显示怎样使用线程池同步 I/O 来开发 Web 服务器。当客户连接到服务器时，池中的一个 线程接受连接，并调用 HTTP\_Handler 中的 open 方法。随后服务器同步地从网络连接读取请求。当读操 作完成时，客户请求随之被解析。在对 GET 请求的响应中，服务器对所请求文件进行内存映射，并将文 件数据同步地写往客户。注意阻塞 I/O 是怎样使 Web 服务器能够遵循 2.2.1 中所概述的步骤的。

class HTTP\_Handler

// = TITLE

// Implements the HTTP protocol

// (synchronous threaded version).

//

// = DESCRIPTION

// This class is called by a

// thread in the Thread Pool.

{

public:

void open (Socket\_Stream \*client)

{

HTTP\_Request request;

// Store reference to client. client\_ = client;

// Synchronously read the HTTP request

// from the network connection and

// parse it.

client\_->recv (request);

parse\_request (request);

}

void parse\_request (HTTP\_Request &request)

{

// Switch on the HTTP command type. switch (request.command ())

{

// Client is requesting a file. case HTTP\_Request::GET:

// Memory map the requested file. Mem\_Map input\_file;

input\_file.map (request.filename());

// Synchronously send the file

// to the client. Block until the

// file is transferred.

client\_->send (input\_file.data (), input\_file.size ());

break;

// Client is storing a file at

// the server.

case HTTP\_Request::PUT:

// ...

}

}

private:

// Socket endpoint. Socket\_Stream \*client\_;

// ...

};

**A.2** 单线程反应式事件分派

下面的代码显示怎样将反应器模式用于开发 Web 服务器。当客户连接到服务器时，HTTP\_Handler:: open 方法被调用。服务器登记 I/O 句柄和在网络句柄“读就绪“时回调的对象（在此例中是 this 指针）。 然后服务器返回事件循环。

当请求数据到达服务器时，reactor\_回调 HTTP\_Handler::handle\_input 方法。客户数据以非阻塞方式被 读取。如果有足够的数据，客户请求就被解析。如果整个客户请求还没有到达，应用就返回反应器事件

循环。

在对 GET 请求的响应中，服务器对所请求的文件进行内存映射；并在反应器上登记，以在网络连接 变为“写就绪”时被通知。当向连接写入数据不会阻塞调用线程时，reactor\_就回调 HTTP\_Handler::handl er\_output 方法。当所有数据都已发送给客户时，网络连接被关闭。

class HTTP\_Handler :

public Reactor::Event\_Handler

// = TITLE

// Implements the HTTP protocol

// (synchronous reactive version).

//

// = DESCRIPTION

// The Event\_Handler base class

// defines the hooks for

// handle\_input()/handle\_output().

//

// = PATTERN PARTICIPANTS

// Reactor = Reactor

// Event Handler = HTTP\_Handler

{

public:

void open (Socket\_Stream \*client)

{

// Initialize state for request request\_.state\_ = INCOMPLETE;

// Store reference to client. client\_ = client;

// Register with the reactor for reading. reactor\_->register\_handler

(client\_->handle (), this, Reactor::READ\_MASK);

}

// This is called by the Reactor when

// we can read from the client handle. void handle\_input (void)

{

int result = 0;

// Non-blocking read from the network

// connection.

do

result = request\_.recv (client\_->handle ());

while (result != SOCKET\_ERROR && request\_.state\_ == INCOMPLETE);

// No more progress possible,

// blocking will occur

if (request\_.state\_ == INCOMPLETE && errno == EWOULDBLOCK)

reactor\_->register\_handler (client\_->handle (), this, Reactor::READ\_MASK);

else

// We now have the entire request parse\_request ();

}

void parse\_request (void)

{

// Switch on the HTTP command type. switch (request\_.command ())

{

// Client is requesting a file. case HTTP\_Request::GET:

// Memory map the requested file. file\_.map (request\_.filename ());

// Transfer the file using Reactive I/O. handle\_output ();

break;

// Client is storing a file at

// the server.

case HTTP\_Request::PUT:

// ...

}

}

void handle\_output (void)

{

// Asynchronously send the file

// to the client.

if (client\_->send (file\_.data (), file\_.size ())

== SOCKET\_ERROR

&& errno == EWOULDBLOCK)

// Register with reactor... else

// Close down and release resources. handle\_close ();

}

private:

// Set at initialization. Reactor \*reactor\_;

// Memory-mapped file\_; Mem\_Map file\_;

// Socket endpoint. Socket\_Stream \*client\_;

// HTTP Request holder. HTTP\_Request request\_;

};

第 **9** 章 接受器－连接器（**Acceptor-Connector**）：用于连接和初始

化通信服务的对象创建模式

Douglas C. Schmidt

**9.1** 意图

接受器－连接器设计模式（Acceptor-Connector）使分布式系统中的连接建立及服务初始化与一旦服 务初始化后所执行的处理去耦合。这样的去耦合通过三种组件来完成：*acceptor*、*connector* 和 *service ha ndler*（服务处理器）。连接器主动地建立到远地接受器组件的连接，并初始化服务处理器来处理在连接上 交换的数据。同样地，接受器被动地等待来自远地连接器的连接请求，在这样的请求到达时建立连接， 并初始化服务处理器来处理在连接上交换的数据。随后已初始化的服务处理器执行应用特有的处理，并 通过连接器和接受器组件建立的连接来进行通信。

**9.2** 例子

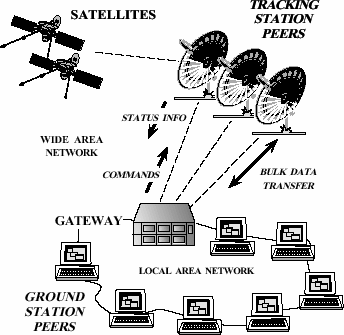


图 9-1 面向连接的应用级网关的物理体系

为演示接受器－连接器模式，考虑图 9-1 中所示的多服务、应用级 Gateway（网关）。通常，Gatewa y 使分布式系统中的协作组件去耦合，并允许它们在无需互相直接依赖的情况下进行交互。图 9-1 中的这 个 Gateway 在不同的服务端点间路由数据，这些端点运行在用于监视和控制人造卫星群的远地 Peer（对 端）上。每个 Peer 中的服务经由 Gateway 发送和接收若干类型的数据,比如状态信息、大块数据和命令。 一般而言，Peer 可以分布在局域网和广域网中。

该 Gateway 是一个路由器，负责协调它的 Peer 之间的通信。从 Gateway 的角度来看，它为之路由数 据的 Peer 服务仅由其应用级通信协议来进行区分，这些协议可能会使用不同的帧格式和有效负载类型。 Gateway 在它的 Peer 之间使用面向连接的 TCP/IP 协议[11]来传输数据。在我们的示例网络配置中， 每个服务都与一个连接端点绑定在一起；该端点由 IP 主机地址和 TCP 端口号指定。端口号唯一地标识服 务的类型。为每种类型的服务/端口维护单独的连接增强了路由策略的灵活性，并提供了更为健壮的错误

处理，如果网络连接意外关闭的话。

在我们的分布式应用中，Gateway 和 Peer 必须能改变它们的连接角色，以支持不同的使用情况。特 别地，或是可以主动地发起连接，或是可以被动地等待连接请求。例如，在一种配置中，Gateway 可以主 动地发起到远地 Peer 的连接，以便路由数据给它们。在另一种配置中，Gateway 可以被动地接收来自 Pe er 的连接请求，后者随即经由 Gateway 路由数据给另外的 Peer。同样地，在一种使用情况下，Peer 可以 是主动的连接发起者，而在另一种使用情况下则是被动的连接接受者。

由于我们的分布式应用的天性，预先指定连接建立和服务初始化角色、并将它们硬编码进 Gateway 和 Peer 组件的传统设计太不灵活。这样的设计过度地将连接建立、服务初始化和服务处理组件耦合在一 起。这样的紧耦合使得独立于通信角色改变连接角色变得很困难。

**9.3** 上下文

分布式系统中利用面向连接协议来在服务端点间进行通信的客户/服务器应用。

**9.4** 问题

分布式应用常常含有复杂的代码，执行连接建立和服务初始化。一般而言，分布式应用中在服务端 点间交换的数据的处理极大地独立于配置问题，比如（1）是哪一个端点发起连接，也就是，连接角色 v s. 通信角色，以及（2）连接管理协议 vs. 网络编程 API。下面对这些问题进行概述：

z 连接角色 **vs.** 通信角色：连接建立角色天然地不对称，也就是，被动的服务端点进行等待，而主动 的服务端点发起连接。但是，一旦连接建立，通信角色和连接角色可以是互不相关的。因而，数据可 以任意的服从服务通信协议的方式在服务端点间进行传输。常用的通信协议包括点对点、请求－响应 和单路流。

z 连接管理协议 **vs.** 网络编程 **API**：不同的网络编程接口，比如 socket 或 TLI，提供不同的 API 来使用 多种连接管理协议建立连接。但是，不管用于建立连接的协议是什么，数据可以使用统一的消息传递 操作来在端点间进行传输，例如，send/recv 调用。

一般而言，用于连接建立和服务初始化的策略变动的频度要远小于应用服务实现和通信协议。因而，

使这些方面去耦合、以使它们能独立地变化，对于开发和维护分布式应用来说是必要的。对于将连接及 初始化协议与通信协议分离的问题，下面一些压力会对解决方案产生影响：

z 应该容易增加新类型的服务、新的服务实现和新的通信协议，而又不影响现有的连接建立和服务初始 化软件。例如，可能有必要扩展 Gateway，以与运行在 IPX/SPX 通信协议、而不是 TCP/IP 之上的目 录服务进行互操作。

z 应该有可能使下面的两种角色去耦合：（1）连接角色，也就是，哪一个进程发起连接 vs. 接受连接， 以及（2）通信角色，也就是，哪一个服务端点是客户或服务器。通常，“客户”和“服务器”之间的 区分指的是通信角色，它们可以与连接角色不相关。例如，在发起到被动服务器的连接时，客户常常 扮演主动角色。但是，这些连接角色可以反转过来。例如，扮演主动通信角色的客户可以被动地等待 另一个进程对其进行连接。9.2 中的例子演示了后一种使用情况。

z 应该有可能编写可以移植到许多 OS 平台上的通信软件，以最大化可用性和市场占有率。许多低级网 络编程 API 的语义只是有着表面的不同，而语法却互不兼容，因而难以使用低级 API，比如 socket 和 TLI，来编写可移植应用。

z 应该有可能将程序员与低级网络编程 API（像 socket 或 TLI）类型安全性的缺乏屏蔽开来。例如，连 接建立代码应完全地与后续的数据传输代码去耦合，以确保端点被正确地使用。没有这种强去耦，服 务可能会错误地在被动模式的传输端点工厂上读写数据，而后者仅应被用于接受连接。

z 应该有可能通过使用像异步连接建立这样的 OS 特性来降低连接响应延迟。例如，有大量对端的应用 可能需要异步、并发地建立许多连接。高效和可伸缩的连接建立对于运行在高响应延迟的 WAN 上的 应用来说特别重要。

z 应该可以尽可能多地复用通用的连接建立和服务初始化软件，以有效利用先前的开发成果。

**9.5** 解决方案

对于分布式应用提供的每个服务，使用接受器－连接器模式来使连接建立及服务初始化与由服务的 两个端点在连接和初始化之后执行的后续处理去耦合。

引入两个工厂，生成已连接和初始化的服务处理器，用于实现应用的服务。第一个工厂，称为接受 器，创建并初始化传输端点，被动地在特定地址上侦听来自远地连接器的连接请求。第二个工厂，连接 器，主动地发起到远地接受器的连接。接受器和连接器都初始化相应的服务处理器，处理在连接上交换 的数据。一旦服务处理器被连接和初始化，它们就执行应用特有的处理，一般不再与接受器和连接器进 行交互。

**9.6** 结构

在图 9-2 中30通过 Booch 类图[2]演示了接受器－连接器模式中的参与者的结构。

30 在此图中，虚线云表示类；云中的虚线框表示模板参数；在一端有空心圆的实线无向边表示两个类之间的使用关系。

服务处理器（**Service Handler**）：Service Handler 实现应用服务，通常扮演客户角色、服务器角色，或同

时扮演这两种角色。它提供挂钩方法，由 Acceptor 或 Connector 调用，以在连接建立时启用应用服务。此 外，Service Handler 还提供数据模式传输端点，其中封装了一个 I/O 句柄，比如 socket。一旦连接和初始 化后，该端点被 Service Handler 用于与和其相连的对端交换数据。

接受器（**Acceptor**）：Acceptor 是一个工厂，实现用于被动地建立连接并初始化与其相关联的 Service Ha ndler 的策略。此外，Acceptor 包含有被动模式的传输端点工厂，它创建新的数据模式端点，由 Service Handler 用于在相连的对端间传输数据。通过将传输端点工厂绑定到网络地址，比如 Acceptor 在其上侦听 的 TCP 端口号，Acceptor 的 open 方法对该工厂进行初始化。

一旦初始化后，被动模式的传输端点工厂侦听来自对端的连接请求。当连接请求到达时，Acceptor

创建 Service Handler，并使用它的传输端点工厂来将新连接接受进 Service Handler 中。

连接器（**Connector**）：Connector 是一个工厂，实现用于主动地建立连接并初始化与其相关联的 Service Handler 的策略。它提供方法，由其发起到远地 Acceptor 的连接。同样地，它还提供另一个方法，完成对 Service Handler 的启用；该处理器的连接是被同步或异步地发起的。Connector 使用两个分开的方法来透 明地支持异步连接建立。

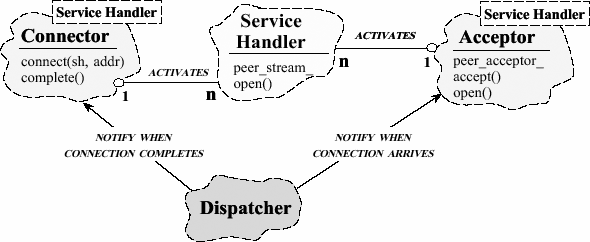


图 9-2 Acceptor-Connector 模式的参与者的结构

当连接建立时，Acceptor 和 Connector 都通过调用 Service Handler 的启用挂钩方法来将其启用。一旦

Service Handler 被 Acceptor 或 Connector 工厂完全初始化，它通常就不再与这些组件进行交互了。

分派器（**Dispatcher**）：为 Acceptor，Dispatcher 将在一或多个传输端点上接收到的连接请求多路分离给适 当的 Acceptor。Dispatcher 允许多个 Acceptor 向其登记，以侦听同时在不同端口上从不同对端而来的连接。

为 Connector，Dispatcher 处理异步发起的连接的完成。在这种情况下，当异步连接被建立时，Dispa tcher 回调 Connector。Dispatcher 允许多个 Service Handler 通过一个 Connector 来异步地发起和完成它们 的连接。注意对于同步连接建立，Dispatcher 并不是必需的，因为发起连接的线程控制也完成服务服务处 理器的启用。

Dispatcher 通常使用事件多路分离模式来实现，这些模式由反应器（Reactor）[3]或前摄器（Proactor） [4]来提供，它们分别处理同步和异步的多路分离。同样地，Dispatcher 也可以使用主动对象（Active Obj ect）模式[5]来实现为单独的线程或进程。

**9.7** 动力特性

下面的部分描述接受器－连接器模式中 Acceptor 和 Connector 组件所执行的协作。我们检查三种规范 的情况：Acceptor、异步的 Connector 和同步的 Connector。

**9.7.1 Acceptor** 组件协作

图 9-3 演示 Acceptor 和 Service Handler 之间的协作。这些协作被划分为三个阶段：

**1.** 端点初始化阶段：为被动地初始化连接，应用调用 Acceptor 的 open 方法。该方法创建被动模式的传 输端点，将其绑定到网络地址，例如，本地主机的 IP 地址和 TCP 端口号，并随后侦听来自对端 Con nector 的连接请求。其次，open 方法将 Acceptor 对象登记到 Dispatcher，以使分派器能够在连接事件 到达时回调 Acceptor。最后，应用发起 Dispatcher 的事件循环，等待连接请求从对端 Connector 到来。

**2.** 服务初始化阶段：当连接请求到达时，Dispatcher 回调 Acceptor 的 accept 方法。该方法装配以下活动 所必需的资源：（1）创建新的 Service Handler，（2）使用它的被动模式传输端点工厂来将连接接受进 该处理器的数据模式传输端点中，以及（3）通过调用 Service Handler 的 open 挂钩将其启用。Servic e Handler 的 open 挂钩可以执行服务特有的初始化，比如分配锁、派生线程、打开日志文件，和/或将 该 Service Handler 登记到 Dispatcher。

**3.** 服务处理阶段：在连接被动地建立和 Service Handler 被初始化后，服务处理阶段开始了。在此阶段， 应用级通信协议，比如 HTTP 或 IIOP，被用于在本地 Service Handler 和与其相连的远地 Peer 之间、 经由前者的 peer\_stream\_端点交换数据。当交换完成，可关闭连接和 Service Handler，并释放资源。

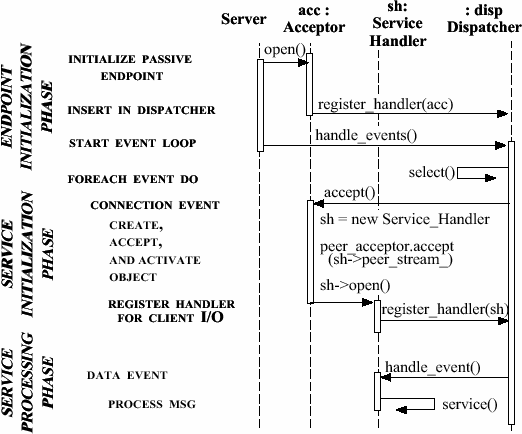


图 9-3 Acceptor 参与者之间的协作

**9.7.2 Connector** 组件协作

Connector 组件可以使用两种常用方案来初始化它的 Service Handler：同步的和异步的。同步的服务 初始化对于以下情形来说是有用的：

z 如果建立连接的延迟非常低，例如，经由回路设备与在同一主机上的服务器建立连接；或是

z 如果有多个线程控制可用，并且使用不同的线程来同步地连接每个 Service Handler 有足够的效率； 或是

z 如果服务必须以固定顺序初始化，而客户不到连接建立不能执行其他有用的工作。

同样地，异步服务初始化在相反的情形中是有用的：

z 如果连接延迟很高，并且有许多对端需要连接，例如，在高延迟 WAN 之上建立大量连接；或是

z 如果仅有单个线程控制可用，例如，如果 OS 平台不提供应用级线程；或是

z 如果服务被初始化的顺序不重要，或如果客户应用必须在建立连接的同时执行另外的工作，比如刷新

GUI。

同步的 Connector 情况中的参与者之间的协作可被划分为以下三个阶段：

**1.** 连接发起阶段：为在 Service Handler 和它的远地 Peer 之间发起连接，应用调用 Connector 的 connect

方法。该方法阻塞调用线程的线程控制、直到连接同步完成，以主动地建立连接。

**2.** 服务初始化阶段：在连接完成后，Connector 的 connect 方法调用 complete 方法来启用 Service Handl er。complete 方法通过调用 Service\_Handler 的 open 挂钩方法来完成启用；open 方法执行服务特有的 初始化。

**3.** 服务处理阶段：此阶段与 Service Handler 被 Acceptor 创建后所执行的服务处理阶段相类似。特别地， 一旦 Service Handler 被启用，它使用与和其相连接的远地 Service Handler 交换的数据来执行应用特 有的服务处理。

同步服务初始化的协作如图 9-4 所示。在此方案中，Connector 将连接发起和服务初始化阶段结合进 单一的阻塞操作中。在此情况中，只为每个线程控制中的每次 connect 调用建立一个连接。

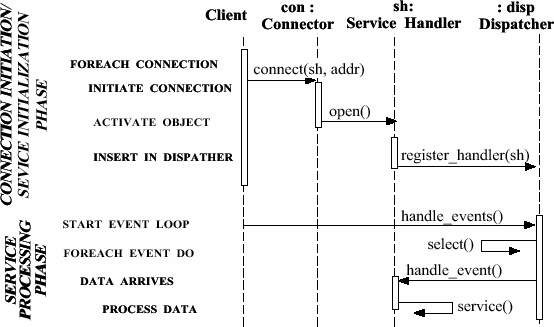


图 9-4 用于同步连接的 Connector 参与者之间的协作

异步的 Connector 中的参与者之间的协作可被划分为以下三个阶段：

**1.** 连接发起阶段：为在 Service Handler 和其远地 Peer 之间发起一个连接，应用调用 Connector 的 conne ct 方法。就如同同步方案，Connector 主动地建立连接。但是，在连接异步完成的同时，它不会阻塞 调用者的线程控制。相反，它将 Service Handler 的传输端点（我们在此例中将其称为 peer\_stream\_） 登记到 Dispatcher，并将控制返回给它的调用者。

**2.** 服务初始化阶段：在连接异步完成后，Dispatcher 回调 Connector 的 complete 方法。该方法通过调用

Service Handler 的 open 挂钩来将其启用。这个 open 挂钩执行服务特有的初始化。

**3.** 服务处理阶段：此阶段与前面描述的其他服务处理阶段相类似。一旦 Service Handler 被启用，它使 用与和其相连接的远地 Service Handler 交换的数据来执行应用特有的服务处理。

图 9-5 演示这三个阶段的使用异步连接建立的协作。在异步方案中，注意连接发起阶段被暂时与服务 初始化阶段分离开来。这样的去耦合使得多个连接发起（经由 connect）和完成（经由 complete）能够在 各自的线程控制中并行地进行。

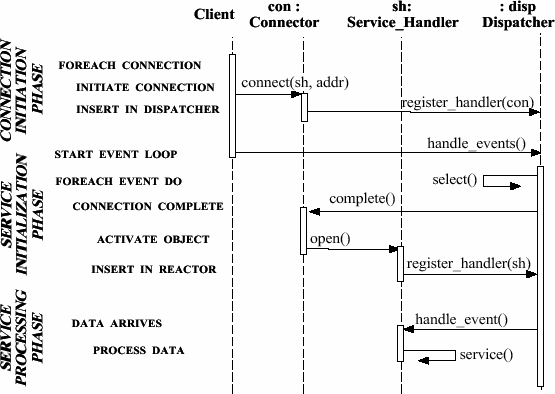


图 9-5 用于异步连接的 Connector 参与者之间的协作

**9.8** 实现

这一部分解释使用接受器－连接器模式来构建通信软件应用所涉及的步骤。这里的实现基于 ACE O O 网络编程工具包[6]中的可复用组件和应用。ACE 提供一组丰富的可复用 C++包装和框架组件，它们可 在一系列 OS 平台上执行常用的通信软件任务。

接受器－连接器模式中的参与者被划分为反应、连接和应用层，如图 9-6 所示。31

反应和连接层分别为分派事件和初始化服务执行通用的、与应用无关的策略。应用层通过提供建立 连接和执行服务处理的具体类来实例化这些通用策略。这样的事务分离增强了接受器－连接器模式实现 中的可复用性、可移植性和可扩展性。

31 该图演示了另外的 Booch 表示法：有向边表示类间的继承关系；虚线有向边表示模板实例化；还有实心圆表示两个类之 间的组合关系。

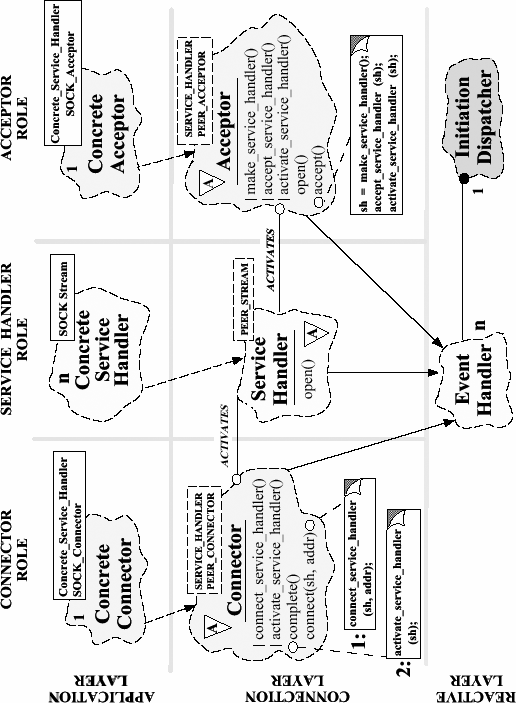


图 9-6 Acceptor-Connector 模式实现中的参与者的分层和划分

下面对接受器－连接器模式实现的讨论从底部的反应层开始，并向上通过连接层和应用层。

**9.8.1** 反应层

反应层处理发生在由 I/O 句柄表示的传输端点（比如 socket 端点）上的事件。该层的两个参与者，I nitiation Dispatcher（发起分派器）和 Event Handler（事件处理器），是由反应器（Reactor）模式[3]定义 的。该模式使得程序在单线程控制中就能够高效地完成来自多个来源的多种类型的事件的多路分离。

反应层中的两个主要角色是：

事件处理器：它规定由挂钩方法[7]组成的接口，抽象地表示应用可提供的事件处理操作。例如，这些挂 钩方法表示这样一些事件：新连接请求、异步开始的连接请求的完成，或是来自相连对端的数据的到达， 等等。Acceptor 和 Connector 组件是从 Event Handler 派生的具体的事件处理器。

发起分派器：为登记、移除和分派 Event Handler 定义接口。Synchronous Event Demultiplexer（同步的 事件多路分离器），比如 select[8]或 WaitForMultipleObjects[9]，通知 Initiation Dispatcher 何时回调应用特 有的事件处理器，以响应特定类型的事件。常用事件包括连接接受事件、数据输入和输出事件，以及超 时事件。

注意 Initiation Dispatcher 是 9.6 描述的 Dispatcher 的实现。一般而言，接受器－连接器 Dispatcher 可 以是反应式、前摄式（Proactive）或多线程的。在这一实现中的特定的 Initiation Dispatcher 使用反应式模 型来在单线程控制中多路分离和分派具体的事件处理器。在我们的例子中，Initiation Dispatcher 是单体（S ingleton）[10]，因为我们只需要它的一个实例用于整个进程。

**9.8.2** 连接层

连接层：

1. 创建 Service Handler；

2. 被动地或主动地将 Service Handler 连接到它们的远地对端；以及

3. 一旦连接，启用 Service Handler。

在此层中的所有行为都是完全通用的。特别地，注意下面描述的实现中的类是怎样委托具体的 IPC

机制和 Concrete Service Handler 的；后者是由在 9.8.3 中描述的应用层实例化的。 应用层委托连接层的方式与连接层委托反应层的方式相类似。例如，反应层中的 Initiation Dispatche

r 代表连接层处理与初始化有关的事件，比如异步的建立连接。 在连接层中有三个主要角色：Service Handler（服务处理器）、Acceptor 和 Connector。

z 服务处理器：该抽象类继承自 Event\_Handler，并为客户、服务器或同时扮演两种角色的组件所提供 的服务处理提供通用接口。应用必须通过继承来定制此类，以执行特定类型的服务。Service Handler 接口如下所示：

// PEER\_STREAM is the type of the

// Concrete IPC mechanism. template <class PEER\_STREAM>

class Service\_Handler : public Event\_Handler

{

public:

// Pure virtual method (defined by a subclass). virtual int open (void) = 0;

// Accessor method used by Acceptor and

// Connector to obtain the underlying stream. PEER\_STREAM &peer (void)

{

return peer\_stream\_;

}

// Return the address that we’re connected to. PEER\_STREAM::PEER\_ADDR &remote\_addr (void)

{

return peer\_stream\_.remote\_addr ();

}

protected:

// Concrete IPC mechanism instance. PEER\_STREAM peer\_stream\_;

};

一旦 Acceptor 或 Connector 建立了连接，它们调用 Service Handler 的 open 挂钩。该纯虚方法必须被

Concrete Service Handler 子类定义；后者执行服务特有的初始化和后续处理。

连接器：该抽象类实现主动连接建立和初始化 Service Handler 的通用策略。它的接口如下所示：

// The SERVICE\_HANDLER is the type of service.

// The PEER\_CONNECTOR is the type of concrete

// IPC active connection mechanism. template <class SERVICE\_HANDLER,

class PEER\_CONNECTOR>

class Connector : public Event\_Handler

{

public:

enum Connect\_Mode

{

SYNC, // Initiate connection synchronously. ASYNC // Initiate connection asynchronously.

};

// Initialization method. Connector (void);

// Actively connecting and activate a service.

int connect (SERVICE\_HANDLER \*sh,

const PEER\_CONNECTOR::PEER\_ADDR &addr, Connect\_Mode mode);

protected:

// Defines the active connection strategy.

virtual int connect\_service\_handler(SERVICE\_HANDLER \*sh, const PEER\_CONNECTOR::PEER\_ADDR &addr, Connect\_Mode mode);

// Register the SERVICE\_HANDLER so that it can

// be activated when the connection completes.

int register\_handler (SERVICE\_HANDLER \*sh, Connect\_Mode mode);

// Defines the handler’s concurrency strategy.

virtual int activate\_service\_handler(SERVICE\_HANDLER \*sh);

// Activate a SERVICE\_HANDLER whose

// non-blocking connection completed. virtual int complete (HANDLE handle);

private:

// IPC mechanism that establishes

// connections actively. PEER\_CONNECTOR connector\_;

// Collection that maps HANDLEs

// to SERVICE\_HANDLER \*s.

Map\_Manager<HANDLE, SERVICE\_HANDLER \*>handler\_map\_;

// Inherited from the Event\_Handler -- will be

// called back by Eactor when events complete

// asynchronously.

virtual int handle\_event (HANDLE, EVENT\_TYPE);

};

// Useful "short-hand" macros used below.

#define SH SERVICE\_HANDLER

#define PC PEER\_CONNECTOR

Conncetor 通过特定类型的 PEER CONNECTOR 和 SERVICE HANDLER 被参数化。PEER CONNECTO R 提供的传输机制被 Connector 用于主动地建立连接，或是同步地、或是异步地。SERVICE HANDLER 提供的服务对与相连的对端交换的数据进行处理。C++参数化类型被用于使（1）连接建立策略与（2）服 务处理器类型、网络编程接口和传输层连接协议去耦合。

参数化类型是有助于提高可移植性的实现决策。例如，它们允许整体地替换 Connector 所用的 IPC 机 制。这使得 Connector 的连接建立代码可在含有不同网络编程接口（例如，有 socket，但没有 TLI；反之 亦然）的平台间进行移植。例如，取决于平台是支持 socket 还是 TLI[11]，PEER CONNECTOR 模板参数 可以通过 SOCK Connector 或 TLI Connector 来实例化。使用参数化类型的另一个动机是改善运行时效率， 因为模板实例化发生在编译时。

更为动态的去耦合可以经由继承和多态、通过使用[10]中描述的工厂方法（Factory Method）和策略

（Strategy）模式来完成。例如，Connector 可以存储指向 PEER CONNECTOR 基类的指针。根据从工厂 返回的 PEER CONNECTOR 的子类，这个 PEER CONNECTOR 的 connect 方法可在运行时被动态地绑定。 一般而言，在参数化类型和动态绑定之间的权衡是参数化类型可能带来额外的编译/链接时开销，而动态 绑定可能带来额外的运行时开销。

connect 方法是应用用以通过 Connector 来发起连接的入口。它的实现如下所示：32

template <class SH, class PC> int

Connector<SH, PC>::connect(SERVICE\_HANDLER \*service\_handler, const PEER\_CONNECTOR::PEER\_ADDR &addr,

Connect\_Mode mode)

{

connect\_service\_handler (service\_handler, addr, mode);

}

该方法使用桥接（Bridge）模式[10]来使 Concrete Connector 能透明地修改连接策略，而不用改变组件接

口。为此，connect 方法委托 Connector 的连接策略，connect\_service\_handler，来发起连接。如下所示：

template <class SH, class PC> int

Connector<SH, PC>::connect\_service\_handler

(SERVICE\_HANDLER \*service\_handler,

const PEER\_CONNECTOR::PEER\_ADDR &remote\_addr, Connect\_Mode mode)

{

// Delegate to concrete PEER\_CONNECTOR

// to establish the connection.

if (connector\_.connect (\*service\_handler, remote\_addr, mode) == -1)

{

if (mode == ASYNC && errno == EWOULDBLOCK)

{

// If connection doesn’t complete immediately

// and we are using non-blocking semantics

// then register this object with the

// Initiation\_Dispatcher Singleton so it will

// callback when the connection is complete. Initiation\_Dispatcher::instance

32 为节省空间，本论文中的大多数错误处理都被省略了。

()->register\_handler (this, WRITE\_MASK);

// Store the SERVICE\_HANDLER in the map of

// pending connections. handler\_map\_.bind

(connector\_.get\_handle (), service\_handler);

}

}

else if (mode == SYNC)

// Activate if we connect synchronously. activate\_service\_handler (service\_handler);

}

如图 9-7 所示，如果 Connect\_Mode 参数的值是 SYNC，一旦连接同步地完成，SERVICE HANDLE

R 将会被启用。该图与图 9-4 相类似，但是提供了另外的实现细节，比如 get\_handle 和 handle\_event 挂钩 方法的使用。

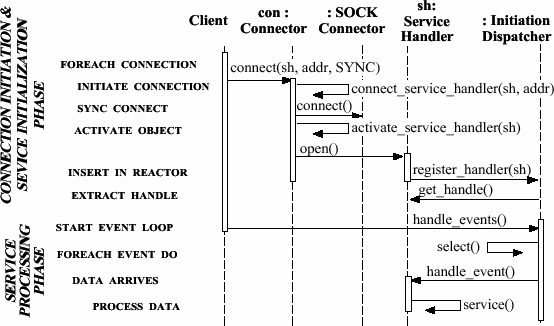


图 9-7 用于同步连接的 Connector 参与者之间的协作

为高效地与多个 Peer 相连，Connector 可能还需要主动、异步地建立连接，也就是，不阻塞调用者。 如图 9-8 所示，异步行为通过将 ASYNC 连接模式传递给 Connector::connect 来指定。该图与图 9-5 相类 似，但是还提供了其他与当前实现相应的细节。

一旦实例化，PEER CONNECTOR 类提供具体的 IPC 机制来同步或异步地发起连接。这里所显示的 Connector 模式的实现使用 OS 和通信协议栈所提供的异步连接机制。例如，在 UNIX 或 Win32 上，Conn ector 可以将 socket 设置进非阻塞模式，并使用像 select 或 WaitForMultipleObject 这样的事件多路分离器 来确定连接何时完成。

为处理还未完成的异步连接，Connector 维护 Service Handler 映射表。因为 Connector 继承自 Event

Handler，当连接完成时，Initiation Dispatcher 可以自动回调 Connector 的 handle\_event 方法。

handle\_event 方法是一个适配器（Adapter）[10]，它将 Initiation Dispatcher 的事件处理接口转换为对

Connector 模式的 complete 方法的调用。

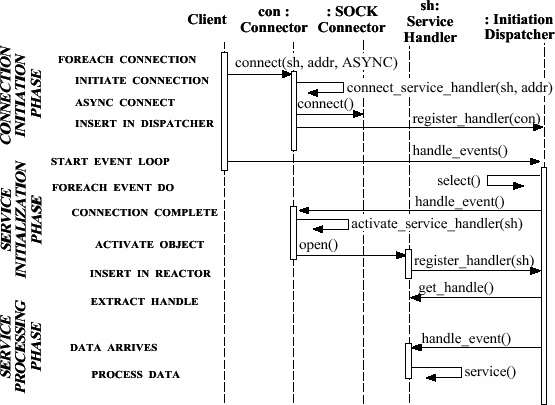


图 9-8 用于异步连接的 Connector 参与者之间的协作

Connector 的 handle\_event 方法如下所示：

template <class SH, class PC> int

Connector<SH, PC>::handle\_event (HANDLE handle, EVENT\_TYPE type)

{

// Adapt the Initiation\_Dispatcher’s event

// handling API to the Connector’s API. complete (handle);

}

complete 方法启用刚刚成功完成非阻塞连接的 SERVICE HANDLER，如下所示：

template <class SH, class PC> int

Connector<SH, PC>::complete (HANDLE handle)

{

SERVICE\_HANDLER \*service\_handler = 0;

// Locate the SERVICE\_HANDLER corresponding

// to the HANDLE.

handler\_map\_.find (handle, service\_handler);

// Transfer I/O handle to SERVICE\_HANDLER \*. service\_handler->set\_handle (handle);

// Remove handle from Initiation\_Dispatcher. Initiation\_Dispatcher::instance

()->remove\_handler (handle, WRITE\_MASK);

// Remove handle from the map. handler\_map\_.unbind (handle);

// Connection is complete, so activate handler. activate\_service\_handler (service\_handler);

}

complete 方法在其内部映射表中查找并移除已连接的 SERVICE HANDLER，并将 I/O 句柄传递给 SERVI

CE HANDLER。最后，它通过调用 activate\_service\_handler 方法初始化 SERVICE HANDLER。该方法委 托由 SERVICE HANDLER 的 open 挂钩指定的并发策略。如下所示：

template <class SH, class PC> int

Connector<SH, PC>::activate\_service\_handler

(SERVICE\_HANDLER \*service\_handler)

{

service\_handler->open ();

}

Service Handler 的 open 挂钩在连接成功建立时被调用。注意该挂钩都将被调用，不管（1）连接是

同步还是异步发起的，或（2）它们是被主动还是被动连接的。这样的统一性使得开发者有可能编写这样 的 Service Handler，其处理可以完全地与它们是怎样被连接和初始化的去耦合。

接受器（**Acceptor**）：该抽象类为被动连接建立和初始化 Service Handler 实现通用的策略。Acceptor 的接 口如下所示：

// The SERVICE\_HANDLER is the type of service.

// The PEER\_ACCEPTOR is the type of concrete

// IPC passive connection mechanism. template <class SERVICE\_HANDLER,

class PEER\_ACCEPTOR>

class Acceptor : public Event\_Handler

{

public:

// Initialize local\_addr transport endpoint factory

// and register with Initiation\_Dispatcher Singleton.

virtual int open(const PEER\_ACCEPTOR::PEER\_ADDR &local\_addr);

// Factory Method that creates, connects, and

// activates SERVICE\_HANDLER’s. virtual int accept (void);

protected:

// Defines the handler’s creation strategy.

virtual SERVICE\_HANDLER \*make\_service\_handler (void);

// Defines the handler’s connection strategy.

virtual int accept\_service\_handler(SERVICE\_HANDLER \*);

// Defines the handler’s concurrency strategy.

virtual int activate\_service\_handler(SERVICE\_HANDLER \*);

// Demultiplexing hooks inherited from Event\_Handler,

// which is used by Initiation\_Dispatcher for

// callbacks.

virtual HANDLE get\_handle (void) const;

virtual int handle\_close (void);

// Invoked when connection requests arrive. virtual int handle\_event (HANDLE, EVENT\_TYPE);

private:

// IPC mechanism that establishes

// connections passively. PEER\_ACCEPTOR peer\_acceptor\_;

};

// Useful "short-hand" macros used below.

#define SH SERVICE\_HANDLER

#define PA PEER\_ACCEPTOR

Acceptor 通过特定类型的 PEER ACCEPTOR 和 SERVICE HANDLER 被参数化。PEER ACCEPTOR 提供的传输机制被 Acceptor 用于被动地建立连接。SERVICE HANDLER 提供的服务对与远地对端交换的 数据进行处理。注意 SERVICE HANDLER 是由应用层提供的具体的服务处理器。

参数化类型使 Acceptor 的连接建立策略与服务处理器的类型、网络编程接口及传输层连接发起协议 去耦合。就如同 Connector 一样，通过允许整体地替换 Acceptor 所用的机制，参数化类型的使用有助于提 高可移植性。这使得连接建立代码可在含有不同网络编程接口（比如有 socket，但没有 TLI；反之亦然） 的平台间移植。例如，取决于平台能够更为高效地支持 socket 还是 TLI，PEER ACCEPTOR 模板参数可 以通过 SOCK Acceptor 或 TLI Acceptor 来实例化。

下面给出 Acceptor 的方法的实现。应用通过调用 Acceptor 的 open 方法来将其初始化。如下所示：

template <class SH, class PA> int

Acceptor<SH, PA>::open

(const PEER\_ACCEPTOR::PEER\_ADDR &local\_addr)

{

// Forward initialization to the PEER\_ACCEPTOR. peer\_acceptor\_.open (local\_addr);

// Register with Initiation\_Dispatcher, which

// ‘‘double-dispatches’’ without get\_handle()

// method to extract the HANDLE. Initiation\_Dispatcher::instance

()->register\_handler (this, READ\_MASK);

}

local\_addr 被传递给 open 方法。该参数含有网络地址，例如，本地主机的 IP 地址和 TCP 端口号，用于侦

听连接。Open 方法将此地址转发给 PEER ACCEPTOR 定义的被动连接接受机制。该机制初始化传输端 点工厂，由后者将地址广告给有兴趣与此 Acceptor 连接的客户。

传输端点工厂的行为由用户所实例化的 PEER ACCEPTOR 的类型来决定。例如，它可以是 socket[1

3]、TLI[14]、STREAM 管道[15]、Win32 命名管道等的 C++包装。

在传输端点工厂被初始化后，open 方法将其自身登记到 Initiation Dispatcher。Initiation Dispatcher

执行“双重分派”，回调 Acceptor 的 get\_handle 方法，以获取底层传输端点工厂的 HANDLE。如下所示：

template <class SH, class PA> HANDLE Acceptor<SH, PA>::get\_handle (void)

{

return peer\_acceptor\_.get\_handle ();

}

Initiation Dispatcher 在内部表中存储此 HANDLE。Synchronous Event Demultipler（同步事件多路分

离器），比如 select，随即被用于检测和多路分离到来的来自客户的连接请求。因为 Acceptor 类继承自 E vent Handler，当连接从对端到达时，Initiation Dispatcher 可以自动回调 Acceptor 的 handle\_event 方法。 该方法是一个适配器（Adapter），它将 Initiation Dispatcher 的事件处理接口转换为对 Acceptor 的 accept 方法的调用。如下所示：

template <class SH, class PA> int

Acceptor<SH, PA>::handle\_event (HANDLE, EVENT\_TYPE)

{

// Adapt the Initiation\_Dispatcher’s event handling

// API to the Acceptor’s API. accept ();

}

如下所示，accept 方法是一个模板方法（Template Method）[10]，它为创建新 SERVICE HANDLER、 将连接接受进其中并启用服务而实现接受器－连接器模式的被动初始化策略：

template <class SH, class PA> int

Acceptor<SH, PA>::accept (void)

{

// Create a new SERVICE\_HANDLER.

SH \*service\_handler = make\_service\_handler ();

// Accept connection from client. accept\_service\_handler (service\_handler);

// Activate SERVICE\_HANDLER by calling

// its open() hook.

activate\_service\_handler (service\_handler);

}

该方法非常简洁，因为它将所有低级细节都分解进具体的 SERVICE HANDLER 和 PEER ACCEPTOR 中，

后二者通过参数化类型被实例化，并可被 Acceptor 的子类定制。特别地，因为 accept 是模板方法，子类 可以扩展 Acceptor 的任意或所有的连接建立和初始化策略。这样的灵活性使得开发者有可能编写这样的 Service Handler，其行为与它们被被动地连接和初始化的方式是相分离的。

make\_service\_handler 工厂方法定义 Acceptor 用于创建 SERVICE HANDLER 的缺省策略。如下所示：

template <class SH, class PA> SH \* Acceptor<SH, PA>::make\_service\_handler (void)

{

return new SH;

}

缺省行为使用了“请求策略”（demand strategy），它为每个新连接创建新的 SERVICE HANDLER。但是，

Acceptor 的子类可以重定义这一策略，以使用其他策略创建 SERVICE HANDLE，比如创建单独的单体

（Singleton）[10]或从共享库中动态链接 SERVICE HANDLER。

accept\_service\_handler 方法在下面定义 Acceptor 所用的 SERVICE HANDLER 连接接受策略：

template <class SH, class PA> int

Acceptor<SH, PA>::accept\_service\_handler(SH \*handler)

{

peer\_acceptor\_->accept (handler->peer ());

}

缺省行为委托 PEER ACCEPTOR 所提供的 accept 方法。子类可以重定义 accept\_service\_handler 方法，以

执行更为复杂的行为，比如验证客户的身份，以决定是接受还是拒绝连接。

Activate\_service\_handler 定义 Acceptor 的 SERVICE HANDLER 并发策略：

template <class SH, class PA> int

Acceptor<SH, PA>::activate\_service\_handler(SH \*handler)

{

handler->open ();

}

该方法的缺省行为是通过调用 SERVICE HANDLER 的 open 挂钩将其启用。这允许 SERVICE HANDLE

R 选择它自己的并发策略。例如，如果 SERVICE HANDLER 继承自 Event Handler，它可以登记到 Initi ation Dispatcher，从而在事件发生在 SERVICE HANDLER 的 PEER STREAM 通信端点上时，使 Initiati on Dispatcher 能够分派其 handle\_event 方法。Concrete Acceptor 可以重定义此策略，以完成更为复杂的 并发启用。例如，子类可以使 SERVICE HANDLER 成为主动对象（Active Object）[5]，使用多线程或 多进程来处理数据。

当 Acceptor 终止时，无论是由于错误还是由于整个应用的关闭，Initiation Dispatcher 都调用 Accept or 的 handle\_close 方法，后者可以释放任何动态获取的资源。在此例中，handle\_close 方法简单地将 clos e 请求转发给 PEER ACCEPTOR 的传输端点工厂。如下所示：

template <class SH, class PA> int

Acceptor<SH, PA>::handle\_close (void)

{

peer\_acceptor\_.close ();

}

**9.8.3** 应用层

应用层提供具体的进程间通信（IPC）机制和具体的 Service Handler。IPC 机制被封装在 C++类中， 以简化编程、增强复用，并使开发者能够整个地替换 IPC 机制。例如，9.9 中使用的 SOCK Acceptor、S OCK Connector，以及 SOCK Stream 类是 ACE C++ socket 包装类库[11]的一部分。它们通过高效、可移 植和类型安全的 C++包装来封装像 TCP 和 SPX 这样的面向连接协议的面向流的语义。

应用层中的三个主要角色描述如下：

具体的服务处理器（**Concrete Service Handler**）：该类定义具体的应用服务，由 Concrete Acceptor 或 C oncrete Connector 启用。Concrete Service Handler 通过特定类型的 C++ IPC 包装（它与和其相连的对端 进行数据交换）来实例化。

具体的连接器（**Concrete Connector**）：该类通过具体的参数化类型参数 SERVICE HANDLER 和 PEER CONNECTOR 来实例化通用的 Connector 工厂。

具体的接受器（**Concrete Acceptor**）：该类通过具体的参数化类型参数 SERVICE HANDLER 和 PEER A CCEPTOR 来实例化通用的 Acceptor 工厂。

Concrete Service Handler 还可以定义服务的并发策略。例如，Service Handler 可以从 Event Handler

继承，并采用反应器（Reactor）[3]模式来在单线程控制中处理来自对端的数据。相反，Service Handler 也可以使用主动对象（Active Object）模式[5]处理到来的数据，而其所在线程控制与 Acceptor 连接它所 用的不相同。下面，我们为我们的 Gateway 例子实现 Concrete Service Handler，演示怎样灵活地配置若 干不同的并发策略，而又不影响接受器－连接器模式的结构或行为。

在 9.9 的示例代码中，SOCK Connector 和 SOCK Acceptor 是分别用于主动和被动地建立连接的 IPC 机制。同样地，SOCK Stream 被用作数据传输递送机制。但是，通过其他机制（比如 TLI Connector 或 N amed Pipe Acceptor）来参数化 Connector 和 Acceptor 也是相当直接的，因为 IPC 机制被封装在 C++包装 类中。同样地，通过使用不同的 PEER STREAM，（比如 SVR4 UNIX TLI Stream 或 Win32 Named Pip e Stream）来参数化 Concrete Service Handler，很容易改变数据传输机制。

9.9 演示怎样实例化 Concrete Service Handler、Concrete Connector 和 Concrete Acceptor，实现 9.2 中描述的 Peer 和 Gateway。这个特定的应用层例子定制连接层中的 Connector 和 Acceptor 组件所提供的 通用初始化策略。

**9.9** 例子解答

下面的代码演示 9.2 中描述的 Peer 和 Gateway 怎样使用接受器－连接器模式来简化连接建立和服务 初始化。9.9.1 演示 Peer 怎样扮演被动角色，9.9.2 演示 Gateway 怎样在与被动的 Peer 的连接建立中扮演 主动角色。

**9.9.1** 用于对端的具体组件

图 9-9 演示 Concrete Acceptor 和 Concrete Service Handler 组件是怎样在 Peer 中构造的。该图中的 A

cceptor 组件与图 9-11 中的 Connector 组件是互补的。

用于与 **Gateway** 通信的服务处理器：如下所示的 Status Handler、Bulk Data Handler 和 Command Handl er 类处理发送到 Gateway 和从 Gateway 接收的路由消息。因为这些 Concrete Service Handler 类继承自 S ervice Handler，它们可以被 Acceptor 被动地初始化。

为演示接受器－连接器模式的灵活性，这些 Service Handler 中的每个 open 例程都可以实现不同的并 发策略。特别地，当 Status Handler 被启用时，它运行在单独的线程中；Bulk Data Handler 作为单独的 进程运行；而 Command Handler 运行在与 Initiation Dispatcher 相同的线程中，后者为 Acceptor 工厂进行 连接请求的多路分离。注意这些并发策略的改变并不影响 Acceptor 的实现，它是通用的，因而也是高度 灵活和可复用的。

我们从定义一个 Service Handler 开始，它为基于 socket 的数据传输使用 SOCK Stream：

typedef Service\_Handler <SOCK\_Stream>PEER\_HANDLER;

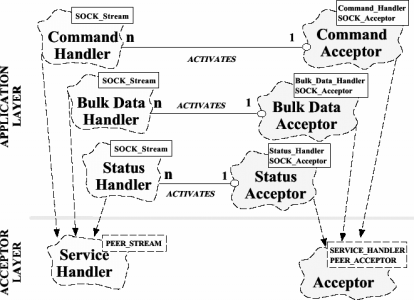


图 9-9 对端的 Acceptor 参与者的结构

PEER HANDLER 的 typedef 构成所有后续服务处理器的基础。例如，Status Handler 类处理发送到 Gate way 和从 Gateway 接收的状态数据：

class Status\_Handler : public PEER\_HANDLER

{

public:

// Performs handler activation. virtual int open (void)

{

// Make handler run in separate thread (note

// that Thread::spawn requires a pointer to

// a static method as the thread entry point). Thread::spawn (&Status\_Handler::service\_run, this);

}

// Static entry point into thread, which blocks

// on the handle\_event () call in its own thread. static void \*service\_run (Status\_Handler \*this\_)

{

// This method can block since it

// runs in its own thread.

while (this\_->handle\_event () != -1)

continue;

}

// Receive and process status data from Gateway. virtual int handle\_event (void)

{

char buf[MAX\_STATUS\_DATA];

stream\_.recv (buf, sizeof buf);

// ...

}

// ...

};

PEER HANDLER 还可被子类化，以生成具体的服务处理器，处理大块数据和命令。例如，Bulk Data H

andler 类处理发送到 Gateway 和从 Gateway 接收的大块数据：

class Bulk\_Data\_Handler : public PEER\_HANDLER

{

public:

// Performs handler activation. virtual int open (void)

{

// Handler runs in separate process. if (fork () == 0) // In child process.

// This method can block since it

// runs in its own process. while (handle\_event () != -1)

continue;

// ...

}

// Receive and process bulk data from Gateway. virtual int handle\_event (void)

{

char buf[MAX\_BULK\_DATA];

stream\_.recv (buf, sizeof buf);

// ...

}

// ...

};

Command Handler 类处理发送到 Gateway 和从 Gateway 接收的命令。

class Command\_Handler : public PEER\_HANDLER

{

public:

// Performs handler activation. virtual int open (void)

{

// Handler runs in same thread as main

// Initiation\_Dispatcher singleton. Initiation\_Dispatcher::instance

()->register\_handler (this, READ\_MASK);

}

// Receive and process command data from Gateway. virtual int handle\_event (void)

{

char buf[MAX\_COMMAND\_DATA];

// This method cannot block since it borrows

// the thread of control from the

// Initiation\_Dispatcher. stream\_.recv (buf, sizeof buf);

// ...

}

//...

};

用于创建 **Peer Service Handler** 的接受器：如下所示的 s\_acceptor、bd\_acceptor 和 c\_acceptor 对象是 Con

crete Acceptor 工厂实例，它们分别创建并启用 Status Handler、Bulk Data Handler 和 Command Handler。

// Accept connection requests from Gateway and

// activate Status\_Handler.

Acceptor<Status\_Handler, SOCK\_Acceptor> s\_acceptor;

// Accept connection requests from Gateway and

// activate Bulk\_Data\_Handler. Acceptor<Bulk\_Data\_Handler, SOCK\_Acceptor> bd\_acceptor;

// Accept connection requests from Gateway and

// activate Command\_Handler.

Acceptor<Command\_Handler, SOCK\_Acceptor> c\_acceptor;

注意模板和动态绑定的使用是怎样允许特定细节灵活地变化的。特别地，在整个这一部分中，当并 发策略被修改时，没有 Acceptor 组件发生变化。这样的灵活性的原因是并发策略已被分解进 Service Ha ndler 中，而不是与 Acceptor 耦合在一起。

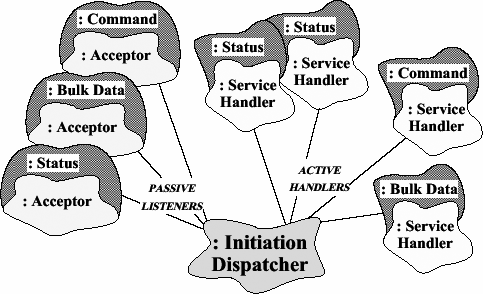


图 9-10 对端中的 Acceptor 组件的对象图

**Peer** 主函数：主程序通过调用具体的 Acceptor 工厂的 open 挂钩（以每个服务的 TCP 端口为参数）来对 它们进行初始化。如 9.8.2 所示，每个 Acceptor 工厂自动地在它的 open 方法中将其自身登记到 Initiation Dispatcher 的实例。

// Main program for the Peer. int main (void)

{

// Initialize acceptors with their

// well-known ports.

s\_acceptor.open (INET\_Addr (STATUS\_PORT)); bd\_acceptor.open (INET\_Addr (BULK\_DATA\_PORT)); c\_acceptor.open (INET\_Addr (COMMAND\_PORT));

// Event loop that handles connection request

// events and processes data from the Gateway. for (;;)

Initiation\_Dispatcher::instance()->handle\_events ();

}

一旦 Acceptor 被初始化，主程序进入事件循环，使用 Initiation Dispatcher 来检测来自 Gateway 的连

接请求。当连接到达时，Initiation Dispatcher 回调适当的 Acceptor，由其创建适当的 PEER HANDLER

来执行服务、将连接接受进处理器、并启用处理器。

图 9-10 演示在与 Gateway（如图 9-12 所示）的四个连接被建立、以及四个 Service Handler 被创建和 启用后，Peer 中的 Concrete Acceptor 组件之间的关系。在 Concrete Service Handler 与 Gateway 交换数据 的同时，三个 Acceptor 也在主线程中持续地侦听新连接。

**9.9.2** 用于 **Gateway** 的具体组件

图 9-11 演示 Concrete Connector 和 Concrete Service Handler 组件是怎样在假想的 Gateway 配置中构 造的。该图中的 Connector 组件与图 9-9 中的 Acceptor 组件是互补的。

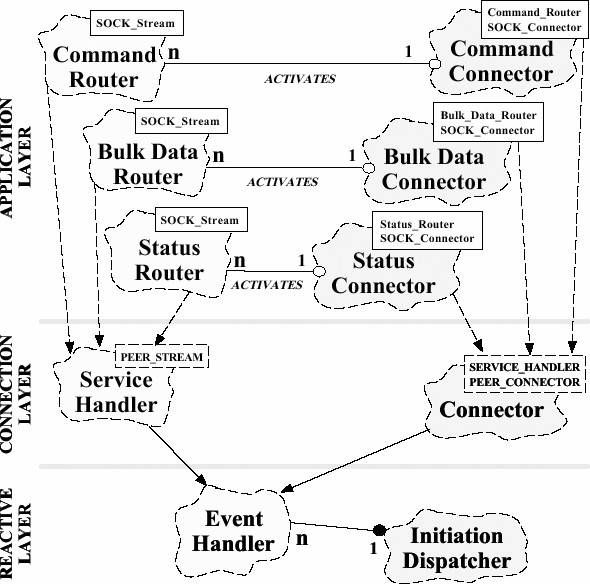


图 9-11 网关的 Connector 参与者的结构

用于 **Gateway** 路由的服务处理器：如下所示的 Status Router、Buld Data Router 和 Command Router 类 将它们从源 Peer 接收到的数据路由到一或多个目的 Peer。因为这些 Concrete Service Handler 类继承自 S ervice Handler，它们可以被 Connector 主动地连接和初始化。

为演示接受器－连接器模式的灵活性，Service Handler 中的每个 open 例程实现不同的并发策略。特 别地，当 Status Router 被启用时，它运行在单独的线程中；Bulk Data Router 作为单独的进程运行；而 C ommand Router 运行在与 Initiation Dispatcher 相同的线程中，后者为 Connector 工厂进行连接完成事件的 多路分离。就如同 Acceptor 一样，注意这些并发策略的变动并不会影响 Connector 的实现，它是通用的， 因而也是高度灵活和可复用的。

我们从定义一个 Service Handler 开始，它为基于 socket 的数据传输而定制：

typedef Service\_Handler <SOCK\_Stream>PEER\_ROUTER;

该类构成所有后续路由服务的基础。例如，Status Router 类路由状态数据到 Peer，或路由来自 Peer 的数 据：

class Status\_Router : public PEER\_ROUTER

{

public:

// Activate router in separate thread. virtual int open (void)

{

// Thread::spawn requires a pointer to a

// static method as the thread entry point). Thread::spawn (&Status\_Router::service\_run, this);

}

// Static entry point into thread, which blocks

// on the handle\_event() call in its own thread. static void \*service\_run (Status\_Router \*this\_)

{

// This method can block since it

// runs in its own thread.

while (this\_->handle\_event () != -1)

continue;

}

// Receive and route status data from/to Peers. virtual int handle\_event (void)

{

char buf[MAX\_STATUS\_DATA];

peer\_stream\_.recv (buf, sizeof buf);

// Routing takes place here...

}

// ...

};

PEER ROUTER 可被子类化，以生成用于路由大块数据和命令具体的服务处理器。例如，Bulk Data Ro

uter 路由数据到 Peer，或路由来自 Peer 的数据：

class Bulk\_Data\_Router : public PEER\_ROUTER

{

public:

// Activates router in separate process. virtual int open (void)

{

if (fork () == 0) // In child process.

// This method can block since it

// runs in its own process. while (handle\_event () != -1)

continue;

// ...

}

// Receive and route bulk data from/to Peers. virtual int handle\_event (void)

{

char buf[MAX\_BULK\_DATA];

peer\_stream\_.recv (buf, sizeof buf);

// Routing takes place here...

}

};

Command Router 类路由命令数据到 Peer，或路由来自 Peer 的命令数据：

class Command\_Router : public PEER\_ROUTER

{

public:

// Activates router in same thread as Connector. virtual int open (void)

{

Initiation\_Dispatcher::instance

()->register\_handler (this, READ\_MASK);

}

// Receive and route command data from/to Peers. virtual int handle\_event (void)

{

char buf[MAX\_COMMAND\_DATA];

// This method cannot block since it borrows the

// thread of control from the Initiation\_Dispatcher. peer\_stream\_.recv (buf, sizeof buf);

// Routing takes place here...

}

};

用于创建 **Peer Service Handler** 的接受器：下面的 typedef 定义为 PEER ROUTER 而定制的 connector 工

厂：

typedef Connector<PEER\_ROUTERS, SOCK\_Connector>PEER\_CONNECTOR;

不像 Concrete Acceptor 组件，我们只需要单个 Concrete Connector。原因是每个 Concrete Acceptor 都被用作创建特定类型的 Concrete Service Handler（比如 Bulk Data Handler 或 Command Handler）的工 厂。因此，必须预先知道全部的类型，从而使多种 Concrete Acceptor 类型成为必要。相反，传递给 Con nector 的 connect 方法的 Concrete Service Handler 是在外部初始化的。因此，它们可以统一地被当作 PE ER ROUTER 处理。

**Gateway**主函数：Gateway的主程序如下所示。get\_peer\_addrs函数创建Status、Bulk Data和Command Rou ter，通过Gateway路由消息。该函数（它的实现没有给出）从配置文件或名字服务中读取Peer地址列表。 每个Peer地址含有IP地址和端口号。一旦Router被初始化，上面定义的Connector工厂通过将ASYNC标志 传递给connect方法来异步地发起所有连接。

// Main program for the Gateway.

// Obtain an STL vector of Status\_Routers,

// Bulk\_Data\_Routers, and Command\_Routers

// from a config file.

void get\_peer\_addrs (vector<PEER\_ROUTERS> &peers);

int main (void)

{

// Connection factory for PEER\_ROUTERS. PEER\_CONNECTOR peer\_connector;

// A vector of PEER\_ROUTERs that perform

// the Gateway’s routing services.

vector<PEER\_ROUTER> peers;

// Get vector of Peers to connect with. get\_peer\_addrs (peers);

// Iterate through all the Routers and

// initiate connections asynchronously.

for (vector<PEER\_ROUTER>::iterator i = peers.begin ();

i != peers.end ();

i++)

{

PEER\_ROUTER &peer = \*i;

peer\_connector.connect (peer, peer.remote\_addr (), PEER\_CONNECTOR::ASYNC);

}

// Loop forever handling connection completion

// events and routing data from Peers. for (;;)

Initiation\_Dispatcher::instance()->handle\_events ();

/\* NOTREACHED \*/

}

所有连接都被异步地调用。它们通过 Connector 的 complete 方法并发地完成，该方法在 Initiation Dispatc

her 的事件循环中被回调。此事件循环还为 Command Router 对象多路分离和分派路由事件；该对象运行 在 Initiation Dispatcher 的线程控制中。Status Router 和 Bulk Data Router 分别执行在单独的线程和进程 中。

图 9-12 演示在与 Peer（如图 9-10 所示）的四个连接被建立、以及四个 Concrete Service Handler 被 创建和启用后，Gateway 中的组件之间的关系。该图演示到另一个 Peer 的四个连接，它们被 Connector“拥 有”，还没有完成。当所有 Peer 连接完全建立时，Gateway 将路由并转发由 Peer 发送给它的消息。

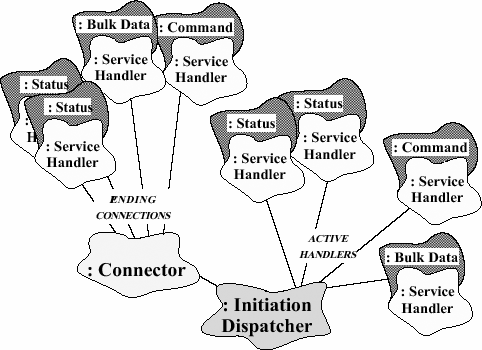


图 9-12 网关中的 Connector 组件的对象图

**9.10** 已知应用

接受器－连接器模式已被用于广泛的框架、工具包和系统中：

**UNIX** 网络超级服务器：比如 inetd[13]、listen[14]以及来自 ASX 框架[6]的 Service Configurator（服务配 置器）看守。这些超级服务器利用主 Acceptor 进程，在一组通信端口上侦听连接。每个端口都和与通信 有关的服务（比如标准的 Internet 服务 ftp、telnet、daytime 和 echo）相关联。接受器进程使 inetd 超级服 务器的功能分解为两个分离的部分：一个用于建立连接，另一个用于接收和处理来自对端的的请求。当 服务请求在被监控的端口上到达时，接受器进程接受请求，并分派适当的预登记的处理器来执行服务。

**CORBA ORB**：许多 CORBA 实现中的 ORB 核心层使用接受器－连接器来在客户请求 ORB 服务时被动 地初始化服务器对象实现。[17]描述接受器－连接器模式怎样被用于实现 The ACE ORB（TAO）[18]；T AO 是 CORBA 一种实时实现。

**WWW** 浏览器：像 Netscape 和 Internet Explorer 这样的 WWW 浏览器中的 HTML 解析组件使用 connect or 组件的异步版本来建立与服务器的连接；这些服务器与 HTML 页面中嵌入的图像相关联。这样的行为 是特别重要的，于是多个 HTTP 连接就可被异步地发起，以避免阻塞浏览器主事件循环。

**Ericsson EOS** 呼叫中心管理系统：该系统使用接受器－连接器模式来使应用级呼叫中心管理器事件服务 器[19]主动地与分布式中央管理系统中的被动的超级用户建立连接。

**Spectrum** 项目：Spectrum 项目的高速医学图像传输子系统[20]使用接受器－连接器模式来为存储大型医 学图像被动地建立连接，并初始化应用服务。一旦连接被建立，应用就发送数兆字节的医学图像给图像

仓库；或从图像仓库中进行接收。

**ACE** 框架：在本论文中描述的 Service Handler、Connector 和 Acceptor 类的实现在 ACE 面向对象网络编 程框架[6]中作为可复用组件提供。

**9.11** 效果

**9.11.1** 好处

接受器－连接器模式提供以下好处： 增强面向对象软件的可复用性、可移植性和可扩展性：通过使服务初始化机制与后续服务处理去耦合来

实现。例如，Acceptor 和 Connector 中的应用无关的机制是可复用的组件，它们知道怎样（1）分别主动 和被动地建立连接，以及（2）一旦连接被建立，初始化相关联的 Service Handler。相反，Service Handl er 知道怎样执行应用特有的服务处理。

这样的事务分离是通过使初始化策略与服务处理策略去耦合来完成的。因而，每种策略都可以独立 地发展。用于主动初始化的策略可以只编写一次，放进类库或框架中，并通过继承、对象合成或模板实 例化来复用。因而，不需要为每个应用都重写同样的主动初始化代码。相反，服务可以根据不同的应用 需求进行变化。通过使用 Service Handler 来参数化 Acceptor 和 Connector，可以使这样的变化的影响被局 限在软件的少量组件中。

改善应用健壮性：应用健壮性是通过彻底地使 Service Handler 和 Acceptor 去耦合来改善的。这样的去耦 合确保了被动模式传输端点工厂 peer\_acceptor\_不会偶然地被用于读写数据。这消除了在使用像 socket 或 TLI[11]这样的弱类型网络编程接口时，可能发生的一类常见错误。

**9.11.2** 缺点

接受器－连接器模式有以下缺点： 额外的间接性：与直接使用底层的网络编程接口相比较，接受器－连接器模式可能带来额外的间接性。

但是，支持参数化类型的语言（比如 C++、Ada 或 Eiffel）可以较小的代价代价实现这些模式，因为编译 器可以内联用于实现这些模式的方法调用。

额外的复杂性：对于简单客户应用（使用单个网络编程接口与单个服务器相连，并执行单项服务）来说， 该模式可能会增加不必要的复杂性。

**9.12** 参见

接受器－连接器模式使用模板方法（Template Method）和工厂方法（Factory Method）模式[10]。A cceptor 的 accept 和 Connector 的 connect 及 complete 函数是模板方法，它们在连接建立时为连接到远地对 端并初始化 Service Handler 而实现通用的服务策略。模板方法的使用使子类可以对创建、连接和启用 Co ncrete Service Handler 的特定细节进行修改。工厂方法被用于使 Service Handler 的创建与它的后续使用 去耦合。

接受器－连接器模式有与客户－分派器－服务器（Client-Dispatcher-Service）模式（在[21]中描述） 类似的意图。它们都关心使主动连接建立与后续服务去耦合。主要的区别是接受器－连接器模式同时致 力于同步和异步连接的被动和主动服务初始化，而客户－分派器－服务器）模式只专注于同步连接建立。

感谢

感谢 Frank Buschmann 和 Hans Rohnert 对本论文提出的有益意见。

参考文献

[1] W. R. Stevens, *TCP/IP Illustrated, Volume* 1. Reading, Massachusetts: Addison Wesley, 1993.

[2] G. Booch, *Object Oriented Analysis and Design with Applications (2nd Edition*). Redwood City, California: Benjamin/ Cummings, 1993.

[3] D. C. Schmidt, “Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Disp atching,” in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, MA: Addison-Wesley, 1995.

[4] T. Harrison, I. Pyarali, D. C. Schmidt, and T. Jordan, “Proactor – An Object Behavioral Pattern for Dispatching Asy nchronous Event Handlers,” in *The 4th Pattern Languages of Programming Conference (Washington University technical r eport #WUCS-97-34*), September 1997.

[5] R. G. Lavender and D. C. Schmidt, “Active Object: an Object Behavioral Pattern for Concurrent Programming,” in *P*

*attern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1

996.

[6] D. C. Schmidt, “ACE: an Object-Oriented Framework for Developing Distributed Applications,” in *Proceedings of the*

*6th USENIX C++ Technical Conferenc*e, (Cambridge, Massachusetts), USENIX Association, April 1994.

[7] W. Pree, *Design Patterns for Object-Oriented Software Developmen*t. Reading, MA: Addison-Wesley, 1994. [8] W.R.Stevens,*UNIX Network Programming, Second Editio*n. Englewood Cliffs, NJ: Prentice Hall, 1997.

[9] H. Custer, *Inside Windows N*T. Redmond, Washington: Microsoft Press, 1993.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Softwar*e. Reading, MA: Addison-Wesley, 1995.

[11] D. C. Schmidt, T. H. Harrison, and E. Al-Shaer, “Object-Oriented Components for High-speed Network Programmin

g,” in *Proceedings of the 1st Conference on Object-Oriented Technologies and System*s,(Monterey,CA), USENIX, June 19

95.

[12] D. C. Schmidt, “IPC SAP: An Object-Oriented Interface to Interprocess Communication Services,” *C++ Repor*t,vol.4, November/December 1992.

[13] W. R. Stevens, *UNIX Network Programming, First Editio*n. Englewood Cliffs, NJ: Prentice Hall, 1990. [14] S. Rago, *UNIX System V Network Programmin*g. Reading, MA: Addison-Wesley, 1993.

[15] D. L. Presotto and D. M. Ritchie, “Interprocess Communication in the Ninth Edition UNIX System,” *UNIX Researc h System Papers, Tenth Editio*n, vol. 2, no. 8, pp. 523–530, 1990.

[16] Object Management Group, *The Common Object Request Broker: Architecture and Specificatio*n, 2.0 ed., July 1995. [17] D. C. Schmidt and C. Cleeland, “Applying Patterns to Develop Extensible ORB Middleware,” *Submitted to the IEE E Communications Magazin*e, 1998.

[18] D. C. Schmidt, D. L. Levine, and S. Mungee, “The Design and Performance of Real-Time Object Request Broker s,” *Computer Communication*s, vol. 21, pp. 294–324, Apr. 1998.

[19] D. C. Schmidt and T. Suda, “An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Co mmunication Systems,” *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Syst ems*), vol. 2, pp. 280–293, December 1994.

[20] G. Blaine, M. Boyd, and S. Crider, “Project Spectrum: Scalable Bandwidth for the BJC Health System,” *HIMSS, H*

*ealth Care Communication*s, pp. 71–81, 1994.

[21] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A Sys tem of Pattern*s. Wiley and Sons, 1996.

第 **10** 章 服务配置器（**Service Configurator**）模式：通过服务配

置器模式动态配置通信服务

Prashant Jain Douglas C. Schmidt

**10.1** 介绍

一系列迅速增长的通信服务正出现在 Internet 上。通信服务是为客户提供功能的服务器中的组件。在 Internet 可用的服务包括：WWW 浏览和内容获取服务（例如，Alta Vista、Apache、Netscape 的 HTTP 服 务器）、软件发布服务（例如，Castinet）、电子邮件和网络新闻传输代理（例如，sendmail 和 nntpd）、远 程文件访问（例如，ftpd）、远程终端访问（例如，rlogind 和 telnetd）、路由表管理（例如，gated 和 route d）、主机和用户活动报告（例如，fingerd 和 rwhod）、网络时间协议（例如，ntpd），以及请求代理服务（例 如，orbixd 和 RPC portmapper），等等。

实现这些服务的常用方法是将每个服务开发成为单独的程序，随后编译、链接，并在单独的进程中 执行每个程序。但是，这样的“静态”配置服务方法生成不灵活、常常也是低效的应用和软件体系结构。 静态配置的主要问题是它相对于应用中的其他服务，将特定服务的实现和服务的配置紧耦合在一起。

本文描述服务配置器（Service Configurator）模式，它通过使服务的行为与服务实现被配置进应用的 时间点去耦合来增强应用的灵活性（常常还有性能）。本文使用用 C++编写的分布式时间服务作为例子来 演示服务配置器模式。但是，服务配置器已经以许多方式被实现，范围从现代操作系统（像 Solaris 和 W indows NT）中的设备驱动程序，到 Internet 超级服务器（像 inetd 和 Windows NT 服务控制管理器），以 及 Java applets。

**10.2** 服务配置器模式

**10.2.1** 意图

使服务的行为与服务实现被配置进应用或系统的时间点去耦合。

**10.2.2** 别名

超级服务器。

**10.2.3** 动机

服务配置器模式使服务的实现与服务被配置进应用或系统的时间点去耦合。这样的去耦改善了服务 的模块性，并允许服务独立于配置问题（比如两个服务是否必须驻留在一起，或是采用何种并发模型执 行服务）而持续发展。

此外，服务配置器模式使它配置的服务的管理集中化。这便利了服务的自动初始化和终止，并且可 以通过将常用的服务初始化和终止模式分解进高效的可复用组件而提高性能。

这一部分使用分布式时间服务作为例子来说明服务配置器模式的动机。

**10.2.3.1** 上下文

当服务需要进行动态发起、挂起、恢复和终止时，应该采用服务配置器模式。此外，如果服务配置 决策必须被推迟至运行时，也应该采用服务配置器模式。

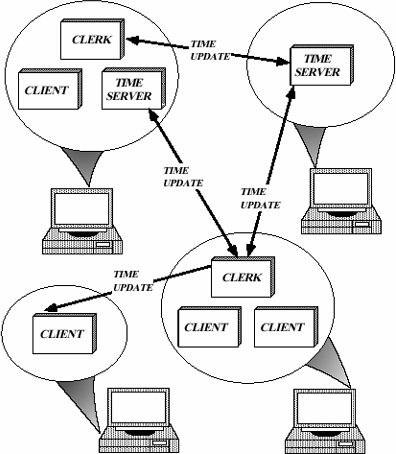


图 10-1 分布式时间服务

为说明该模式的动机，考虑图 10-1 中所示的分布式时间服务。该服务为在局域网和广域网中协作的 计算机提供准确、容错的时钟同步。在要求多个主机维护精确的全局时间的分布式系统中，同步时间服 务是很重要的。例如，大型分布式医学成像系统[1]需要全局同步的时钟，以确保病人的检查被精确地记 录时间，并由放射科医生通过健康保健递送系统迅捷地加以分析。

如图 10-1 所示，该分布式时间服务的体系结构含有下列时间服务器（Time Server）、事务员（Clerk） 和客户（Client）组件：

z 时间服务器回答事务员作出的关于时间的查询。

z 事务员查询一或多个时间服务器，以确定正确的时间、使用若干分布式时间算法[2, 3]中的一种来计 算近似正确的时间，并且更新它自己的本地系统时间。

z 客户使用事务员维护的全局时间信息来提供与其他主机上的客户所用的时间观念的一致性。

**10.2.4** 常用解决方案

一种实现分布式时间服务的方法是将时间服务器、事务员和客户的逻辑功能静态地配置进分离的物 理进程。在该方法中，可有一或多个主机运行时间服务器进程；后者处理来自事务员进程的时间更新请 求。下面的 C++代码段演示静态配置的时间服务器进程的结构：

// The Clerk\_Handler processes time requests

// from Clerks.

class Clerk\_Handler :

public Svc\_Handler <SOCK\_Stream>

{

public:

// This method is called by the Reactor

// when requests arrive from Clerks. virtual int handle\_input (void)

{

// Read request from Clerk and reply

// with the current time.

}

// ...

};

// The Clerk\_Acceptor is a factory that

// accepts connections from Clerks and creates

// Clerk\_Handlers.

typedef Acceptor<Clerk\_Handler, SOCK\_Acceptor> Clerk\_Acceptor;

int main (int argc, char \*argv[])

{

// Parse command-line arguments. Options::instance ()->parse\_args (argc, argv);

// Set up Acceptor to listen for Clerk connections. Clerk\_Acceptor acceptor(Options::instance ()->port ());

// Register with the Reactor Singleton.

Reactor::instance ()->register\_handler(&acceptor, ACCEPT\_MASK);

// Run the event loop waiting for Clerks to

// connect and perform time queries. for (;;)

Reactor::instance ()->handle\_events ();

/\* NOTREACHED \*/

}

该程序使用反应器（Reactor）模式[4]和接受器（Acceptor）模式[5]来实现静态配置的时间服务器进程。

每个需要全局时间同步的主机都可以运行事务员进程。事务员基于接收自一或多个时间服务器的值 周期性地更新它们的本地系统时间。下面的 C++代码段演示静态配置的事务员进程的结构：

// This class communicates with the Time\_Server. class Time\_Server\_Handler { /\* ... \*/ };

// This class establishes connections with the

// Time Servers and periodically queries them for

// their latest time values.

class Clerk : public Svc\_Handler <SOCK\_Stream>

{

public:

// Initialize the Clerk. Clerk (void)

{

Time\_Server\_Handler \*\*handler = 0;

// Use the Iterator pattern and the

// Connector pattern to set up the

// connections to the Time Servers.

for (ITERATOR iterator (handler\_set\_); iterator.next (handler) != 0; iterator.advance ())

{

connector\_.connect (\*handler);

Time\_Value timeout\_interval (60);

// Register a timer that will expire

// every 60 seconds. This will trigger

// a call to the handle\_timeout() method,

// which will query the Time Servers and

// retrieve the current time of day. Reactor::instance ()->schedule\_timer

(this, timeout\_interval);

}

}

// This method implements the Clock Synchronization

// algorithm that computes local system time. It

// is called periodically by the Reactor’s timer

// mechanism.

int handle\_timeout (void)

{

// Periodically query the servers by iterating

// over the handler set and obtaining time

// updates from each Time Server. Time\_Server\_Handler \*\*handler = 0;

// Use the Iterator pattern to query all

// the Time Servers

for (ITERATOR iterator (handler\_set\_); iterator.next (handler) != 0; iterator.advance ())

{

Time\_Value server\_time = (\*handler)->get\_server\_time ();

// Compute the local system time and

// store this in shared memory that

// is accessible to the Client processes.

}

}

private:

typedef Unbounded\_Set <Time\_Server\_Handler \*>HANDLER\_SET;

typedef Unbounded\_Set\_Iterator<Time\_Server\_Handler \*> ITERATOR;

// Set of Clerks and iterator over the set.

HANDLER\_SET handler\_set\_;

// The connector\_ is a factory that

// establishes connections with Time Servers

// and creates Time\_Server\_Handlers. Connector<Time\_Server\_Handler, SOCK\_Connector>connector\_;

};

int main (int argc, char \*argv[])

{

// Parse command-line arguments. Options::instance ()->parse\_args (argc, argv);

// Initialize the Clerk. Clerk clerk;

// Run the event loop, periodically

// querying the Time Servers to determine

// the global time. for (;;)

Reactor::instance ()->handle\_events ();

/\* NOTREACHED \*/

}

该程序使用反应器（Reactor）模式[4]和连接器（Connector）模式[6]来实现静态配置的事务员进程。

客户进程可以使用它们的本地事务员报告的同步的时间。为最少化通信开销，当前时间可被存储在 共享内存中，后者被映射到事务员和同一主机上所有客户的地址空间中。除了时间服务，这些主机提供 的其他通信服务（比如文件传输、远程登录和 HTTP 服务器）也可以在分离的静态配置的进程中执行。

**10.2.5** 常用解决方案的陷阱和缺陷

尽管像反应器、接受器和连接器这样的模式的使用改善了上面所示的分布式时间服务器的模块性和 可移植性，使用静态方法来配置通信服务有以下缺点：

z 在开发周期中必须过早地作出服务配置决策：这是不合需要的，因为开发者可能无法预先知道使服务 组件驻留在一起或分布驻留的最佳方式。例如，在无线计算环境中、内存资源的匮乏可能会迫使客户 和事务员被划分进运行在分离的主机上的两个独立进程。相反，在实时的航空控制环境中，可能必须 使事务员和服务器驻留进一个进程，以降低通信响应延迟33。迫使开发者过早地采用特定的服务配置

33 如果使事务员和服务器驻留在同一进程中，事务员可以通过以下手段来优化通信：（1）消除对到服务器的 socket 连接的 需要，以及（2）经由共享内存直接访问服务器的时间。

会阻碍灵活性，并可能降低性能和功能。

z 修改某服务可能会对其他服务产生不利影响：每个服务组件的实现都与它的初始配置紧耦合在一起。 这导致开发者难以在不影响其他服务的情况下修改某个服务。例如，在上面提到的实时航空控制环境 中，可以静态地配置事务员和时间服务器、使它们在一个进程中执行，以降低响应延迟。但是如果事 务员所实现的分布式时间算法被改变，现有的事务员代码就有可能也需要修改、重编译和静态重链接。 而终止进程以改变事务员代码可能也会终止时间服务器。对于高度可用的系统（比如电信交换机或呼 叫中心[7]），这样的服务中断可能是不能接受的。

z 系统性能不能高效地伸缩：将每个服务与一个进程关联在一起占死了 OS 资源（比如 I/O 描述符、虚 拟内存和进程表槽口）。如果服务常常空闲，这样的设计可能是很浪费的。而且，对于许多短期生存 的通信任务（比如向时间服务器请求当前时间，或在域名服务中解析主机地址请求）来说，进程常常 是错误的抽象。在这些情况下，多线程主动对象（Active Object）（8）或单线程反应式[4]事件循环可 能更为高效。

**10.2.6** 更好的解决方案

更为方便和灵活的实现分布式服务的方法常常是使用服务配置器模式。该模式使通信服务的行为与 这些服务被配置进应用或系统的时间点去耦合。服务配置器模式消除了以下需求的压力：

z 延缓对服务的特定类型或特定实现的选择，直至设计周期中非常迟后的阶段：这允许开发者集中考虑 服务的功能（例如，时间同步算法），而不用过早地卷入特定的服务配置。通过使功能与配置去耦合， 服务配置器模式允许应用独立于系统所用的配置策略和机制而发展。

z 通过编写多个独立开发的、不需要全局知识的服务来构建完整的应用或系统：服务配置器模式要求所 有服务都拥有用于配置和控制的统一接口。这使得服务可被当作积木、很容易地作为组件集成进更大 的应用中。贯穿所有服务的统一接口使得它们在怎样配置上有着同样的“外观和感受”（“look and f eel”）。继而，这样的统一性通过促进“最少惊讶法则”（“principle of least surprise”）而简化了应用 开发。

z 在运行时优化、控制和重配置服务的行为：使服务的实现与它的配置去耦合使得调谐服务的特定实现 或配置参数成为可能。例如，取决于硬件和操作系统上可用的并行性，在分离的线程或进程中运行多 个服务可能更高效，也可能更低效。当有更多的信息可用于帮助优化服务时，服务配置器模式使应用 能够在运行时选择和调整这些行为。此外，在分布式系统中增加新服务或更新服务常常可以无需停止 现有服务就得以完成。

图 10-2 使用 OMT 表示法来演示根据服务配置器模式设计的分布式时间服务的结构。

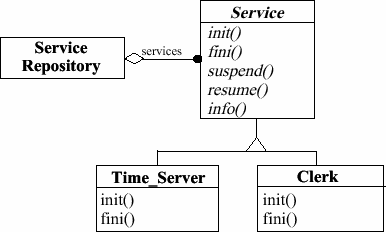


图 10-2 分布式时间服务的结构

Service 基类为配置和控制服务（比如时间服务或事务员）提供标准接口。基于服务配置器的应用使 用该接口来发起、挂起、恢复和终止服务，以及获取关于服务的运行时信息（比如它的 IP 地址和端口号）。 服务自身驻留在 Service Repository（服务仓库）中，可由基于服务配置器的应用来在 Service Repository 中增加或移除。

Service 基类的两个子类出现在分布式时间服务中：Time Server 和 Clerk。每个子类表示一个在分布 式时间服务中有着特定功能的具体 Service。Time Server 服务负责接收和处理来自 Clerk 的时间更新请求。 Clerk 服务是连接器（Connector）[6]工厂，负责（1）为每个服务器创建新连接，（2）动态分配新的处理 器，以向相连的服务器发送时间更新请求，（3）通过处理器接收来自所有服务器的回复，以及（4）随后 更新本地系统时间。

通过管理时间服务中的服务组件的配置，服务配置器模式使分布式时间服务变得更为灵活，并从而 使它与实现问题分离开来。此外，服务配置器提供了框架来将其他通信服务的配置和管理合并在一个管 理单元中。

**10.3** 适用性

当有以下情况时使用服务配置器模式：

z 服务必须被动态地发起、挂起、恢复及终止；以及

z 服务的实现可能会改变，但是它相对于有关服务的配置保持不变；并且/或者一组驻留在一起的服务 可能会改变，但是它们的实现保持不变；或是

z 通过编写多个独立开发和可动态配置的服务，应用或系统可以被简化；或是

z 通过使用单一管理单元进行配置，多个服务的管理可以被简化或优化。

当有以下情况时不要使用服务配置器模式：

z 由于安全限制，动态（重）配置不合需要（在这种情况下，可能必须使用静态配置34）；或是

z 服务的初始化或终止太过复杂，或与上下文的耦合过于紧密，以致于不能以统一的方式来完成；或是

z 因为从不变动，服务不能从动态配置中受益；或是

z 对于动态（重）配置使用的 OS 和语言机制所带来的额外的间接层次，苛刻的性能需求要求使其最小 化。

**10.4** 结构和参与者

在图 10-3 中使用 OMT 表示法演示了服务配置器模式的结构：

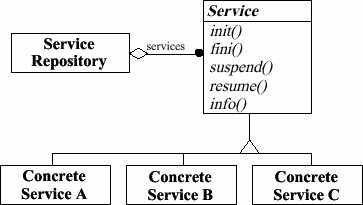


图 10-3 服务配置器模式的结构 服务配置器模式中的关键参与者包括：

z 服务（**Service**）：规定含有挂钩方法[9]（比如初始化和终止）的接口，由基于服务配置器的应用用于 动态配置 Service。

z 具体服务（**Clerk** 和 **Time Server**）：实现服务的挂钩方法及其他的服务特有的功能（比如事件处理和 与客户的通信）。

z 服务仓库（**Service Repository**）：维护基于服务配置器的应用所提供的所有服务的仓库。这使得管理 实体可以对被配置的服务的行为进行集中管理和控制。

**10.5** 协作

图 10-4 描述下面的服务配置器模式三个阶段中的组件间协作：

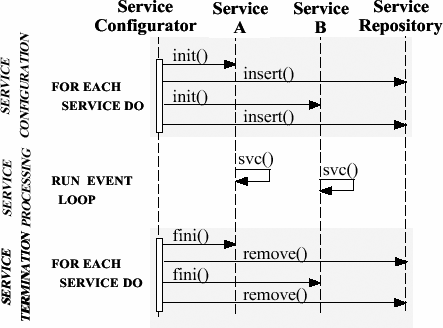


图 10-4 服务配置器模式的交互图

z 服务配置：服务配置器通过调用 Service 的 init 方法来对其进行初始化。一旦 Service 被成功初始化， 服务配置器将其增加到 Service Repository；后者管理和控制所有的 Service。

z 服务处理：在被配置进系统后，Service 执行其处理任务（也就是，为客户请求服务）35。当 Service

的处理在执行时，服务配置器可以挂起和恢复 Service。

z 服务终止：一旦不再需要，服务配置器通过调用 Service 的 fini 挂钩方法来将其终止。该挂钩允许 Se rvice 在终止前进行清理。一旦 Service 终止，服务配置器将其从 Service Repository 中移除。

**10.6** 效果

**10.6.1** 好处

服务配置器模式提供以下好处：

z 集中式管理：该模式将一或多个服务合并进单一的管理单元中。通过自动完成通用的服务初始化和终 止活动（比如打开和关闭文件、获取和释放锁，等等），这有助于开发的简化。此外，它还利用一组 统一的配置管理操作（比如初始化、挂起、恢复和终止）来使通信服务的管理集中化。

z 增强模块性和复用：该模式通过使服务的实现与服务的配置去耦合而改善了通信服务的模块性和可复 用性。此外，所有服务都有统一的接口，并通过它来进行配置，从而鼓励了复用、并简化了后续服务

的开发。

z 增强配置动态性：该模式使服务能够被动态地重配置，而不用修改、重编译，或静态地重新链接已有 代码。此外，服务的重配置常常可以无需重启该服务、或其他与之驻留一处的服务就得以完成。36

z 增加调整和优化的机会：通过使服务功能与用于执行服务的并发策略去耦合，该模式增加了开发者可 用的服务配置选择的范围。通过从一系列并发策略中进行选择，开发者可以适应性地调整看守的并发 水平，以配合客户的需求和可用的 OS 处理资源。可选项包括在客户请求到达时派生线程或进程、或 在服务创建时预先派生线程或进程，等等。

**10.6.2** 缺点

服务配置器模式有以下缺点：

z 缺乏确定性：该模式导致在应用的服务被在运行时配置之前，难以确定该应用的行为。对于实时系统 来说，这可能是成问题的，因为在与其他特定的服务一起运行时，动态配置的服务可能无法可预测地 执行。例如，一个新配置的服务可能会消耗过多的 CPU 周期，从而饿死其他服务，并导致它们错过 它们的最终期限。

z 可靠性降低：使用服务配置器模式的应用比静态配置的应用可能要更不可靠，因为服务的特定配置可 能会有害地影响服务的执行。例如，有毛病的服务可能会崩溃，从而破坏它与其他服务共享的状态信 息。如果多个服务被配置成在同一进程中运行，这就会特别地成问题。

z 增加开销：该模式增加了服务执行的额外的间接层次。例如，服务配置器首先初始化服务，并将其装 载进 Service Repository。这在时间紧急的应用中可能会带来过多的开销。此外，使用动态链接来实现 服务配置器模式增加了方法调用和全局变量访问的额外的间接层次。

**10.7** 实现

服务配置器模式可以通过许多方式实现。这一部分解释实现该模式时所涉及的步骤和可选方案。表 1

0-1 中总结了这些步骤和可选方案。

|  |  |
| --- | --- |
| 步骤 | 常用可选方案 |
| 定义服务控制接口 | z 服务从抽象基类继承  z 服务响应控制消息 |
| 定义服务仓库 | z 维护服务实现表 |
| 选择配置机制 | z 在命令行指定  z 通过用户接口指定  z 通过配置文件指定 |
| 确定服务执行机制 | z 反应式执行 |

36 确保健壮的动态服务重配置超出了服务配置器模式的范围。支持健壮的重配置主要是协议（Protocol）和策略（Policy） 的事情，而服务配置器模式主要致力于（重）配置机制（Mechanism）。

z 多线程主动对象

z 多进程主动对象 表 10-1 实现服务配置器模式所涉及的步骤

z 定义服务控制接口：下面是服务必须支持的的基本接口，用以使服务配置器能够配置和控制服务：

■ 服务初始化：提供服务的入口，并执行服务的初始化。

■ 服务终止：终止服务的执行。

■ 服务挂起：暂时挂起服务的执行。

■ 服务恢复：恢复挂起服务的执行。

■ 服务信息：报告描述服务的信息（例如，端口号或服务名）。

定义服务控制接口有两种基本的方法，基于继承的和基于消息的：

■ 基于继承的：该方法使每个服务都从公共的基类继承。这是 ACE Service Configurator 框架 [7]和 Java applets 所用的方法。它通过定义含有许多纯虚“挂钩”方法的抽象基类来工作， 如下所示：

class Service

{

public:

// = Initialization and termination hooks. virtual int init (int argc, char \*argv[]) = 0; virtual int fini (void) = 0;

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

// = Informational hook.

virtual int info (char \*\*, size\_t) = 0;

};

init 方法用作 Service 的入口。它被服务配置器用于初始化 Service 的执行。fini 方法允许服

务配置器终止 Service 的执行。suspend 和 resume 方法是调度挂钩，由服务配置器用于挂起 和恢复 Service 的执行。info 方法允许服务配置器获取与 Service 相关的信息（比如它的名字 和网络地址）。这些方法一起给出了服务配置器和由它管理的 Service 之间的统一约定。

■ 基于消息的：另一种控制通信服务的方法是编写每个 Service，使之响应一组特定的消息。 这使得开发者有可能将服务配置器集成进缺乏继承的非 OO 编程语言中（比如 C 或 Ada83）。 Windows NT 服务控制管理器（SCM）使用这种方案。每个 Windows NT 主机都拥有主 SC

M 进程，通过向系统服务传递多种控制信息（比如 PAUSE、RESUME 和 TERMINATE）来 自动对其进行初始化和管理。每个 SCM 管理的服务的开发者都要负责编写代码来处理这些 消息。

z 定义服务仓库：Service Repository 被用于维护所有的服务实现，比如对象、可执行程序，或是动态

链接库（DLL）。当服务被配置进系统、或从中移除时，服务配置器使用 Service Repository 来对其进 行访问。每个服务的当前状态（比如它是活动的还是挂起的）也在仓库中维护。服务配置器存储并访 问在主内存、文件系统或内核（例如，它可以使用报告进程和线程状态的操作）中的 Service Reposi tory 信息。

z 选择配置机制：服务需要在执行前被配置。配置服务需要指定一些属性，这些属性指示服务实现（比 如可执行程序或 DLL）的位置，以及在运行时初始化服务所需的参数。这样的配置条件可以通过多 种方式来指定（比如在命令行上、经由环境变量、通过用户接口，或是在配置文件中）。通过将服务 属性和初始化参数合并在一处，集中式配置机制可以简化应用中服务的安装和管理。

z 确定服务执行机制：已由服务配置器动态配置的服务可以使用反应式[4]和主动对象[8]方案的多种组 合来执行。下面简要地检查这些可选方案：

■ 反应式执行：可为服务配置器和它配置的所有服务的执行使用单线程控制。

■ 多线程主动对象：该方法使动态配置的服务运行在服务配置器进程里它们自己的线程控制 中。服务配置器可以“按需”派生新线程，或是在已有线程池中执行这些服务。

■ 多进程主动对象：该方法使动态配置的服务运行在它们自己的进程中。服务配置器可以“按 需”派生新进程，或是在已有进程池中执行这些服务。

**10.8** 示例代码

下面的代码介绍一个用 C++写成的服务配置器模式的例子。该例子聚焦于 10.2.3 中介绍的分布式时 间服务的与配置有关的方面。此外，它还演示了其他模式（比如反应器模式[4]、接受器[5]和连接器[6]模 式）的使用，它们通常被用于开发通信服务和对象请求代理（Object Request Broker）。

在下面的例子中，图 10-3 所示的 OMT 类图中的 Concrete Service 类由 Time Server 类以及 Clerk 类 来表示。这一部分中的 C++代码实现 Time Server 和 Clerk 类37。两个类都继承自 Service，使得它们可被 动态配置进应用中。此外，该方法使用基于显式动态链接[7] 和配置文件的配置机制来动态地配置分布式 时间服务的事务员和服务器部分。服务执行机制则基于在单线程控制中的反应式事件处理模型。

下面的例子显示事务员组件可以怎样改变它用于计算本地系统时间的算法、而又不影响服务配置器 配置的其他组件的执行。一旦算法被修改，事务员组件就由服务配置器进行动态重配置。

下面所示的代码还包括main驱动函数，它提供任何基于服务配置器的应用的通用入口。使用ACE[7]， 该实现可运行在UNIX/POSIX和Win32平台上；ACE可通过WWW在[http://www.cs.wustl.edu/~schmidt/ACE.](http://www.cs.wustl.edu/%7Eschmidt/ACE) html获取。

**10.8.1 Time Server** 类

Time Server使用Acceptor类来接受来自一或多个事务员的连接。Acceptor类使用接受器模式[5]来为所

37 为节省空间，省略了大多数错误处理代码。

有来自事务员的连接创建处理器；这些事务员想要接收时间更新请求。该设计使Time Server的实现与它 的配置去耦合，因此，开发者可以独立于Time Server的配置改变它的实现。这为改进时间服务器的实现 提供了灵活性。

Time Server类继承自在10.7中定义的Service基类。这使得服务配置器能够动态地链接Timer Server， 以及解除其链接。在将Time Server服务装载进Service Repository之前，服务配置器调用它的init挂钩。该 方法执行Time Server特有的初始化代码。同样地，当服务不再被需要时，fini挂钩方法被服务配置器自动 调用，以将其终止。

// The Clerk\_Handler processes time requests

// from Clerks.

class Clerk\_Handler :

public Svc\_Handler <SOCK\_Stream>

{

// This is identical to the Clerk\_Handler

// defined in the second section.

};

class Time\_Server : public Service

{

public:

// Initialize the service when linked dynamically. virtual int init (int argc, char \*argv[])

{

// Parse command line arguments to get

// port number to listen on. parse\_args (argc, argv);

// Set the connection acceptor endpoint into

// listen mode (using the Acceptor pattern). acceptor\_.open (port\_);

// Register with the Reactor Singleton. Reactor::instance ()->register\_handler

(&acceptor\_, ACCEPT\_MASK);

}

// Terminate the service when dynamically unlinked. virtual int fini (void)

{

// Close down the connection. acceptor\_.close ();

}

// Other methods (e.g., info(), suspend(), and

// resume()) omitted.

private:

// Parse command line arguments or those

// specified by the configuration file. int parse\_args (int argc, char \*argv[]);

// Acceptor is a factory that accepts

// connections from Clerks and creates

// Clerk\_Handlers.

Acceptor<Clerk\_Handler, SOCK\_Acceptor>acceptor\_;

// Port the Time Server listens on. int port\_;

};

注意服务配置器也可以分别通过调用 Time Server 的 suspend 和 resume 挂钩来将其挂起和恢复。

**10.8.2 Clerk** 类

Clerk 使用 Connector 类来建立和维护与一或多个 Time Server 的连接。Connector 类使用连接器模式[6]

来为每个到时间服务器的连接创建处理器。处理器接收并处理来自 Time Server 的时间更新。

Clerk 类继承自 Service 基类。因此，像 Time Service 一样，它也可以被服务配置器动态地配置。服 务配置器可以分别通过调用 Clerk 的 init、suspend、resume 和 fini 挂钩来将其初始化、挂起、恢复和终止。

// This class communicates with the Time\_Server. class Time\_Server\_Handler

: public Svc\_Handler <SOCK\_Stream>

{

public:

// Get the current time from a Time Server. Time\_Value get\_server\_time (void);

// ...

};

// This class establishes and maintains connections

// with the Time Servers and also periodically queries

// them to calculate the current time.

class Clerk : public Service

{

public:

// Initialize the service when linked dynamically. virtual int init (int argc, char \*argv[])

{

// Parse command line arguments and for

// every host:port specification of server,

// create a Clerk instance that handles

// connection to the server. parse\_args (argc, argv);

Time\_Server\_Handler \*\*handler = 0;

// Use the Iterator pattern and the

// Connector pattern to set up the

// connections to the Time Servers.

for (ITERATOR iterator (handler\_set\_); iterator.next (handler) != 0; iterator.advance ())

{

connector\_.connect (\*handler);

Time\_Value timeout\_interval (60);

// Register a timer that will expire

// every 60 seconds. This will trigger

// a call to the handle\_timeout() method,

// which will query the Time Servers and

// retrieve the current time of day. Reactor::instance ()->schedule\_timer

(this, timeout\_interval);

}

}

// Terminate the service when dynamically unlinked. virtual int fini (void)

{

Time\_Server\_Handler \*\*handler = 0;

// Disconnect from all the time servers. for (ITERATOR iterator (handler\_set\_);

iterator.next (handler) != 0;

iterator.advance ()) (\*handler)->close ();

// Remove the timer.

Reactor::instance ()->cancel\_timer (this);

}

// info(), suspend(), and resume() methods omitted.

// The handle\_timeout method implements the

// Clock Synchronization algorithm that computes

// local system time. It is called periodically

// by the Reactor’s timer mechanism. int handle\_timeout (void)

{

// Periodically query the servers by iterating

// over the handler set and obtaining time

// updates from each Time Server. Time\_Server\_Handler \*\*handler = 0;

// Use the Iterator pattern to query all

// the Time Servers

for (ITERATOR iterator (handler\_set\_); iterator.next (handler) != 0; iterator.advance ())

{

Time\_Value server\_time = (\*handler)->get\_server\_time ();

// Compute the local system time and

// store this in shared memory that

// is accessible to the Client processes.

}

}

private:

// Parse command line arguments or those

// specified by the configuration file and

// create Clerks for every specified server. int parse\_args (int argc, char \*argv[]);

typedef Unbounded\_Set <Time\_Server\_Handler \*>HANDLER\_SET;

typedef Unbounded\_Set\_Iterator<Time\_Server\_Handler \*> ITERATOR;

// Set of Clerks and iterator over the set. HANDLER\_SET handler\_set\_;

// Connector used to set up connections

// to all servers.

Connector<Time\_Server\_Handler, SOCK\_Connector>connector\_;

};

通过遍历其处理器列表，Clerk 周期性地发送时间更新请求给所有与其相连的 Time Server。一旦 Cl

erk 接收到来自所有与其相连的 Time Server 的响应，它就重新计算它的本地系统时间。因而，当客户向 事务员请求当前时间时，它们会接收到全局同步的时间值。

**10.8.3** 配置应用

**10.8.3.1** 共驻式配置

下面的代码演示应用的动态配置和执行，使用配置文件来使 Time Server 和 Clerk 共同驻留在同一 O S 进程中：

int main (int argc, char \*argv[])

{

// Configure the daemon. Service\_Config daemon (argc, argv);

// Perform daemon services updates. daemon.run\_event\_loop ();

/\* NOTREACHED \*/

}

这个完全通用的 main 程序在 Service Config 对象的构造器中动态地配置通信服务。该方法查询下面

的 svc.conf 配置文件：

# Configure a Time Server. dynamic Time\_Server Service\*

netsvcs.dll:make\_Time\_Server() "-p $TIME\_SERVER\_PORT"

# Configure a Clerk. dynamic Clerk Service\*

netsvcs.dll:make\_Clerk()

"-h tango.cs:$TIME\_SERVER\_PORT"

"-h perdita.wuerl:$TIME\_SERVER\_PORT"

"-h atomic-clock.lanl.gov:$TIME\_SERVER\_PORT"

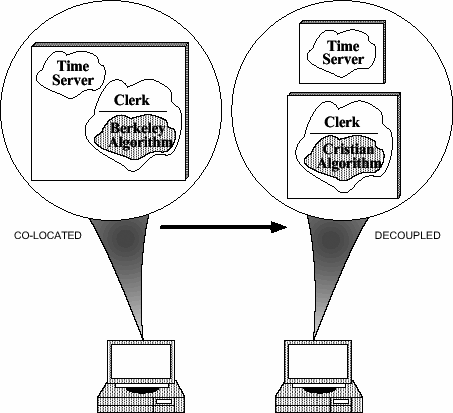
"-P 10" # polling frequency

ACE 服务配置框架处理 svc.conf 配置文件中的每一条目。该框架将 dynamic 指令解释为将指定 Servi ce 动态链接进应用进程的命令。表 10-2 总结了可用的服务配置指令。

|  |  |
| --- | --- |
| 指令 | 描述 |
| **dynamic** | 动态链接和启用服务 |
| **static** | 启用静态链接的服务 |
| **remove** | 完全地移除服务 |
| **suspend** | 挂起服务，而不将其移除 |
| **resume** | 恢复先前挂起的服务 |

表 10-2 服务配置指令

图 10-5 重配置时间服务器和事务员



例如，svc.conf 文件中的第一个条目指定一个服务名（Time\_Server），由 Service Repository 用于标识

动态配置的服务。make\_Time\_Server 是一个工厂函数，位于动态链接库 netsvcs.dll 中。服务配置器框架动 态地将此 DLL 链接进应用的地址空间，然后调用 make\_Time\_Server 工厂函数。该函数动态地分配新的 T ime Server 实例，如下所示：

Service \*make\_Time\_Server(void)

{

return new Time\_Server;

}

在第一个条目中最后的字符串参数指定含有端口号的环境变量，Time Server 将在该端口上侦听 Clerk 连 接。服务配置器将此字符串转换为“argc/argv”风格的向量，并将其传递给 Time Server 的 init 挂钩。如果 init 方法成功执行，Service \* 就被存储在 Service Repository 中的 Time\_Server 名下。

svc.conf 文件中的第二个条目指定怎样动态配置 Clerk。和前面一样，服务配置器将 netsvcs.dll 动态地 链接进应用的地址空间，并调用 make\_Clerk 工厂函数来创建新的 Clerk 实例。三个时间服务器（tanggo. cs、perdita.wuerl 和 atomic-clock.lanl）的名字和端口号被传递给 init 挂钩。此外，-P 10 选项定义事务员 轮询时间服务器的频度。

**10.8.3.2** 分布式配置

假设我们不想将 Time Server 和 Clerk 放在一起，以减少应用的内存占用。因为我们正在使用服务配 置器模式，所有要做的就是将 svc.conf 文件划分为两部分。一部分含有 Time Server 条目，而另一部分含 有 Clerk 条目。因为服务配置器模式有将服务的行为与配置去耦合这一优点，服务自身可以无需改变。图

10-5 显示在 Time Server 和 Clerk 驻留在同一进程时、以及在划分后配置看起来是怎样的。

**10.8.4** 重配置应用

现在假设我们需要改变 Clerk 的算法实现。例如，我们可能决定从 Berkeley 算法[2]的实现切换到 Cr istian 的算法[3]的实现。下面对它们作一概述：

z *Berkeley* 算法：在该方法中，Time Server 是主动的组件，周期性地轮询网络中的全部机器，询问那 里的时间。基于接收到的响应，它计算正确时间的合计值，并告诉所有机器相应地调整它们的时间。

z *Cristian* 算法：在该方法中，Time Server 是响应事务员作出的查询的被动实体。它不会主动地查询其 他机器来确定它自己的时间。

要回应 Time Server 的特性，可能必需改变时间同步算法。例如，如果 Time Server 所驻留的机器上有 W WV 接收器38，Time Server 可被用作被动的实体，而 Cristian 算法也可能就是适合的。在另一方面，如果 Time Server 所驻留的机器上没有 WWV 接收器，那么 Berkeley 算法的实现可能就更为适合。

图 10-5 显示事务员实现中的改变（相应于时钟同步算法中的改变）。改变发生在分离 Time Server 和

Clerk 的过程中，它们先前是驻留在一起的。 理想地，我们想要改变算法实现，而又不影响其他服务或时间服务的其他组件的执行。使用服务配

置器来实现此要求只需要简单地对 svc.conf 文件作出下面的修改：

# Terminate Clerk

38 WWV 接收器截听 National Institute Standard Time（NIST）广播的短脉冲，以为公众提供通用协调时间（Universal Co ordinated Time，UTC）。

remove Clerk

仅有的额外要求是让服务配置器处理该指令。这可以通过生成外部事件（比如 UNIX SIGHUP 信号、 RPC 通知，或是 Windows NT Registry 事件）来完成。收到该事件，应用将再次查询配置文件，并终止 Clerk 服务的执行。服务配置器将调用 Clerk 的 fini 方法，并从而终止事务员组件的执行。其他服务的执 行不会受影响。

一旦事务员服务被终止，就可以对算法实现进行改变。随后代码就可以被重编译和重链接，以形成 新的 netsvcs DLL。也可以采取类似的方法来将事务员服务增加回服务配置器。可以修改配置文件，用新 指令指定事务员需被动态链接，如下所示：

# Reconfigure a new Clerk. dynamic Clerk Service\*

netsvcs.dll:make\_Clerk()

"-h tango.cs:$TIME\_SERVER\_PORT"

"-h perdita.wuerl:$TIME\_SERVER\_PORT"

"-h atomic-clock.lanl.gov:$TIME\_SERVER\_PORT"

随后生成一个外部事件，致使进程重新读入配置文件，并将事务员组件增加到仓库中。一旦 Clerk 的 init

方法被服务配置器框架调用，事务员组件就将开始执行。

图 10-6 显示像事务员服务这样的 Service 的生存期状态图。

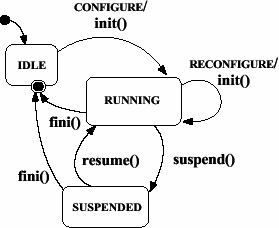


图 10-6 服务的生命周期状态图 注意在终止和移除事务员服务的整个过程中，没有其他活动服务受到实现改变和事务员服务重配置的影

响。替换新的服务实现的容易程度进一步例证了服务配置器模式所提供的灵活性。

**10.9** 已知应用

ts：

服务配置器模式被广泛用于系统和应用编程环境中，包括 UNIX、Windows NT、ACE 和 Java apple

z 现代操作系统设备驱动程序：大多数现代操作系统（比如 Solaris 和 Windows NT）支持动态可配置

的内核级设备驱动程序。这些驱动程序可以经由 init/fini/info 挂钩动态地链接进系统，或从系统中解 除链接。这些操作系统使用服务配置器模式来使管理员能够重配置 OS 内核，而又不必将其关闭、重 编译和静态地重新链接新驱动程序，以及重启系统。

z **UNIX** 网络看守管理：服务配置模式已被用于管理 UNIX 网络看守的“超级服务器”（Superserver）中。 两个广泛可用的网络看守管理框架是 inetd[10]和 listen[11]。两个框架都查询配置文件，后者指定（1） 服务名（比如标准 Internet 服务 ftp、telnet、daytime 和 echo），（2）端口号，用于在其上侦听连接到 这些服务的客户，以及（3）可执行文件，在客户连接时调用并执行服务。两个框架都含有主接受器[5] 进程，它监控一组与这些服务相关联的端口。当客户连接在被监控端口上发生时，接受器进程接受该 连接，并将请求多路分离给适当的预登记的服务处理器。该处理器执行服务（反应式地或在主动对象 中），并将任何结果返回给客户。

z **Windows NT** 服务控制管理器（**SCM**）：不像 inetd 和 listen，Windows NT 服务控制管理器（SCM） 本身不是端口监控器。就是说，它不提供内建支持来侦听一组 I/O 端口、并在客户请求到达时“按需” 分派服务器进程。相反，它提供基于 RPC 的接口，允许主 SCM 进程自动发起和控制（也就是，暂停、 恢复、终止，等等）管理员安装的服务（比如远程注册表访问）。这些服务将另外作为单服务或多服 务看守进程中的分离线程运行。每个被安装的服务都单独负责配置它自己，监控任意通信端点（可以 比 socket 端口更广泛）。例如，SCM 可以控制命名管道和共享内存。

z **ACE** 自适配通信环境框架：ACE 框架[7]为动态配置和控制通信服务提供了一组 C++机制。ACE Ser vice Configurator 扩展了 inetd、listen 和 SCM 所提供的机制，以自动支持通信服务的动态链接和解除 链接。10.8 中包含的代码显示了怎样将 ACE 框架用于实现分布式时间服务。ACE 所提供的机制受到 了现代操作系统中用于配置和控制设备驱动程序的接口的影响。但是， ACE Service Configurator 框 架聚焦于应用级 Service 对象的动态配置和控制，而非针对内核级设备驱动程序。

z **Java applets**：Java 中的 applet 机制使用了服务配置器模式。Java 支持下载、初始化、启动、挂起、 恢复和终止 applet。它通过提供发起和终止线程的方法（例如，start 和 stop）来提供这样的支持。Ja va applet 中的方法可以使用 Thread.currentThread()来访问它在其中运行的线程，并随后向它发出控制 消息，比如 suspend、resume 和 stop。[12]介绍了一个例子，演示服务配置器模式是怎样被用于 Java applets 的。

**10.10** 相关模式

服务配置器模式的意图与配置（Configuration）模式 [13]相类似。配置模式使分布式应用中相关于服 务配置的、结构上的问题与服务自身的执行去耦合。它被用于在一些框架中配置分布式系统，以支持从 一组组件构造一个分布式系统。以类似的方式，服务配置器模式使服务初始化与服务处理去耦合。主要 的区别是配置模式更多地聚焦于一系列相关服务的主动合成，而服务配置器模式聚焦于在特定端点上的 服务处理器的动态初始化。此外，服务配置器模式还聚焦于使服务行为与服务的并发策略去耦合。

管理器模式（Manager Pattern）[14]通过承担创建和删除一组对象的责任来对其进行管理。此外，它 还提供一个接口，以允许客户访问它管理的对象。服务配置器模式可以使用管理器模式来按照需要创建 和删除服务，以及维护它使用管理器模式创建的服务的仓库。但是，必须为使用管理器模式创建的服务 增加动态配置、初始化、挂起、恢复和终止功能，才能全面地实现服务配置器模式。

服务配置器常常利用反应器[4]模式来为被配置的服务完成事件多路分离和分派。同样地，长执行周 期的、动态配置的服务常常使用主动对象模式[15]。

基于服务配置器的系统的管理接口（比如配置文件或 GUI）提供了一个外观（Façade）[16]。该外观 简化了在服务配置器中执行的应用的管理和控制。

Service 基类提供的虚方法是一些回调“挂钩”[9]。这些挂钩被服务配置器用于发起、挂起、恢复和 终止服务。

可以使用工厂方法（Factory Method）[16]来创建 Service。这允许应用决定创建何种类型的 Service。

**10.11** 结束语

本文描述服务配置器模式，并阐释了它怎样使服务的实现与它们的配置去耦合。这样的去耦合增强 了服务的灵活性和可扩展性。特别地，服务实现可以独立于许多相关于服务配置的问题而持续开发和改 进。此外，服务配置器还提供了重配置服务、而又无需修改、重编译或静态重链接已有代码的功能。

服务配置器还使其配置的服务的管理集中化。通过使通用的服务初始化任务自动化（比如打开和关 闭文件、获取和释放锁，等等），这样的集中化可以简化编程工作。此外，集中式管理还对服务的生存期 提供了更大的控制。

服务配置器模式已被广泛地用于许多环境中。本文使用了一个用 C++编写的分布式时间服务作为例 子来演示服务配置器模式。使分布式时间服务的组件的开发与它们被配置进系统的时间点去耦合的能力 例证了服务配置器模式所提供的灵活性。这样的去耦合允许开发者采用不同的分布式时间算法开发不同 的事务员。配置特定事务员的决策成为运行时决策，从而带来了更大的灵活性。本文还显示了怎样将服 务配置器模式用于动态重配置分布式时间服务，而又无须修改、重编译或静态重链接运行中的服务器。 服务配置器模式被广泛用于许多环境中，比如 Solaris 和 Windows NT 的设备驱动程序、像 inetd 这

样的 Internet 超级服务器，Windows NT 服务控制管理器，以及 ACE 框架。在每种情况下，服务配置器 都使服务的实现与服务的配置得以去耦合。这样的去耦合对应用的可扩展性和灵活性都提供了支持。

参考文献

[1] I. Pyarali, T. H. Harrison, and D. C. Schmidt, “Design and Performance of an Object-Oriented Framework for High- Performance Electronic Medical Imaging,” *USENIX Computing System*s, vol. 9, November/December 1996.

[2] R. Gusella and S. Zatti, “The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3B SD,” *IEEE Transactionson Software Engineerin*g, vol. 15, pp. 847–853, July 1989.

[3] F. Cristian, “Probabilistic Clock Synchronization,” *Distributed Computin*g, vol. 3, pp. 146–158, 1989.

[4] D. C. Schmidt, “Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Disp atching,” in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wes ley, 1995.

[5] D. C. Schmidt, “Design Patterns for Initializing Network Services: Introducing the Acceptor and Connector Patterns,”

*C++ Repor*t, vol. 7, November/December 1995.

[6] D. C. Schmidt, “Connector: a Design Pattern for Actively Initializing Network Services,” *C++ Repor*t, vol. 8, Januar y 1996.

[7] D. C. Schmidt and T. Suda, “An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Co mmunication Systems,” *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Syst*

*ems*), vol. 2, pp. 280–293, December 1994.

[8] R. G. Lavender and D. C. Schmidt, “Active Object: an Object Behavioral Pattern for Concurrent Programming,” in *P*

*attern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1

996.

[9] W. Pree, *Design Patterns for Object-Oriented Software Developmen*t. Reading, MA: Addison-Wesley, 1994. [10] W. R. Stevens, *UNIX Network Programming, First Editio*n. Englewood Cliffs, NJ: Prentice Hall, 1990. [11] S. Rago, *UNIX System V Network Programmin*g. Reading, MA: Addison-Wesley, 1993.

[12] P. Jain and D. C. Schmidt, “Service Configurator: A Pattern for Dynamic Configuration of Services,” in *Proceeding s of the* 3rd *Conference on Object-Oriented Technologies and System*s, USENIX, June 1997.

[13] S. Crane, J. Magee, and N. Pryce, “Design Patterns for Binding in Distributed Systems,” in *The OOPSLA ’95 Work shop on Design Patterns for Concurrent, Parallel, and Distributed Object-Oriented System*s, (Austin, TX), ACM, Oct. 19

95.

[14] P. Sommerland and F. Buschmann, “The Manager Design Pattern,” in *Proceedings of the* 3rd *Pattern Languages of*

*Programming Conferenc*e, September 1996.

[15] R. G. Lavender and D. C. Schmidt, “Active Object: an Object Behavioral Pattern for Concurrent Programming,” in *Proceedings of the* 2nd *Annual Conference on the Pattern Languages of Program*s, (Monticello, Illinois), pp. 1–7, Septem ber 1995.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Softwar*e. Reading, MA: Addison-Wesley, 1995.