ACE 自适配通信环境中文技术文档

下篇：ACE 应用实例

作者：Douglas C. Schmidt http://www.cs.wustl.edu/~schmidt/
译者：马维达 http://www.flyingdonkey.com/

第 1 章 应用模式语言开发应用级网关

Douglas C. Schmidt

摘 要

通信应用的开发者必须致力于应对在开发中反复出现的、与效率、可扩展性和健壮性相关的设计挑战。这些挑战常常独立于应用特有的需求。成功的开发者通过应用适当的模式和模式语言来解决这些挑战。但是，传统上，这些模式被锁在专家级开发者的头脑里、或是深埋在复杂的系统源码中。本论文的主要目的是描述一种作为面向对象通信软件基础的模式语言。除了描述该语言中的各种模式，论文还阐释了为什么了解模式间的关系及权衡可以帮助指导可复用通信框架和应用的构造。

1.1 介绍

通信软件是一组构造现代分布式系统及应用（比如 Web 服务、分布式对象、协作式应用，以及电子商务系统[1]）的服务和协议。构建、维护和增强高质量的通信软件是困难的。开发者必须对许多复杂问题有深入了解，比如服务的初始化和分布、并发控制、流控制、错误处理、事件循环集成，以及容错。在由有经验的软件开发者所创建的成功的通信应用中必定包藏着这些问题的有效解决方案。

将成功的通信软件解决方案的本质与特定实现的细节分离开来并非是不重要的。即使软件使用组织良好的面向对象（OO）框架和组件编写，要确定关键的角色和关系仍有可能是很困难的。而且，操作系统（OS）平台*特性*，比如多线程的有无；或是应用*需求*，比如最优努力 vs. 容错错误处理，常常是不同的。这些不同可能会掩盖在同一领域的不同应用的解决方案之间、其底层体系结构的共性。

捕捉成功的通信软件的核心共性是相当重要的，因为：

1. *它为负责增强和维护现有软件的程序员保存了重要的设计信息*。这些信息常常只驻留在起初的开发者的头脑中。因此，如果没有明确地为这些设计信息编写文档，它们就有可能随时间而消逝，从而增大软件的熵，并降低软件的可维护性和质量。
2. *它有助于对正在构建新通信系统的开发者的设计选择进行指导*。通过为它们的领域中常见的陷阱和缺陷编写文档，模式可以帮助开发者选择适当的体系结构、协议和平台特性，而不用将时间和努力浪费在（重新）实现低效或易错的解决方案上。

本论文的目的是通过例子来演示捕捉和传达成功的通信软件的本质的途径；在文中描述了用于构建应用级*网关*（gateway）的模式语言，该网关在分布在整个通信系统中的*对端*（peer）之间进行消息路由。模式所表示的是构建软件时所发生的问题的成功解决方案[2]。当相关模式被编织在一起，它们就构成了一种语言，有助于：

- 为讨论软件开发问题定义词汇表；以及
- 为这些问题的有序解决提供步骤。

通过确定通信软件开发中的基本挑战，研究和应用模式及模式语言可帮助开发者增强他们的解决方案的质量。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结构风格；消除非功能性的压力，比如可复用性、可移植性和可扩展性；以及避开通常只能通过昂贵的“试－错”方法来学习的开发陷阱和缺陷[3]。

本论文根据模式语言来介绍应用级网关的 OO 体系结构和设计；该语言用于指导可复用和网关专用的框架及组件的构造。应用级网关对于可靠性、性能和可扩展性有着严格的要求。因此，它们是介绍出现在大多数通信软件中的关键模式的结构、参与者和效果的极好样本。

本论文中描述的模式语言是在构建广泛的通信系统时揭示的，其中包括在线交易处理系统、电信交换管理系统[4]、电子医学成像系统[5]、并行通信子系统[6]、航空任务计算[7]，以及实时 CORBA 对象请求代理（ORB）[8]，等等。尽管这些应用中的特定需求不一样，其通信软件设计挑战却是相似的。因此，该模式语言含有专家的设计经验，可被广泛地复用于通信软件领域，其范围远远超出了在本论文描述的网关例子。

本论文的余下部分组织如下：1.2 概述应用级网关的 OO 软件体系结构；1.3 使用应用级网关作为例子，检查该模式语言中的模式，它们构成了可复用通信软件的基础；1.4 对这些模式和文献中的其他模式进行比较；1.5 给出结束语。

1.2 应用级网关的 OO 软件体系结构

网关是使网络上相互协作的对端去耦合、并使它们进行交互、而又不直接相互依赖的中介者[2]。如图 1-1 所示，通过网关路由的消息含有封装在路由消息中的有效负载。图 1-2 演示应用级网关的软件体系结构中对象之间的结构、关联，以及内部和外部动力特性。该体系结构基于为多种研究和产品通信系统开发网关所积累的大量经验。在我们构建了许多网关应用之后，下面的事实变得清楚起来：它们的软件体系结构极大地独立于用于将消息路由给对端的协议。这样的认识使得图 1-2 中描述的组件已为其他的数千通信软件项目所复用[1]。能如此有系统地复用这些组件的能力源于两个因素：

1. *对通信软件领域中关键模式的动作和交互的理解*。模式在比源码和 OO 设计模型（它所关注的是个体的对象和类）更高的水平上捕捉软件体系结构中的参与者的结构和动力特性。在本论文中描述的某些通信软件模式已经分别建立了文档[1]。尽管个体的模式描述捕捉了有价值的专家设计经验，然而复杂的通信软件系统常含有许多模式；理解这些模式之间的关系对于推动解决在构建通信软件时所产生的巨大挑战、以及相应的文档编写来说是必需的。因此，1.3 根据一种用于通信软件的*模式语言*来描述这些模式之间的交互和关系。在该语言中的模式协同工作来解决通信软件领域里的复杂问题。
2. *对实现这些模式的 OO 框架的开发*。识别在许多通信软件系统中普遍出现的模式有助于使可复用框架组件的开发成形。本论文所基于的网关系统使用 ACE 自适配通信环境框架[9]来实现，ACE 提供了集成的可复用 C++包装外观（wrapper façade）和组件来完成常见的通信软件任务。这些任务包括事件多路分离、事件处理器分派、连接建立、路由、应用服务的动态配置，以及并发控制，等等。此外，ACE 框架还含有 1.3 描述的模式实现。但是，这些模式比它们在 ACE 中的实现要丰富得多，并且已被其他许多通信系统所应用。

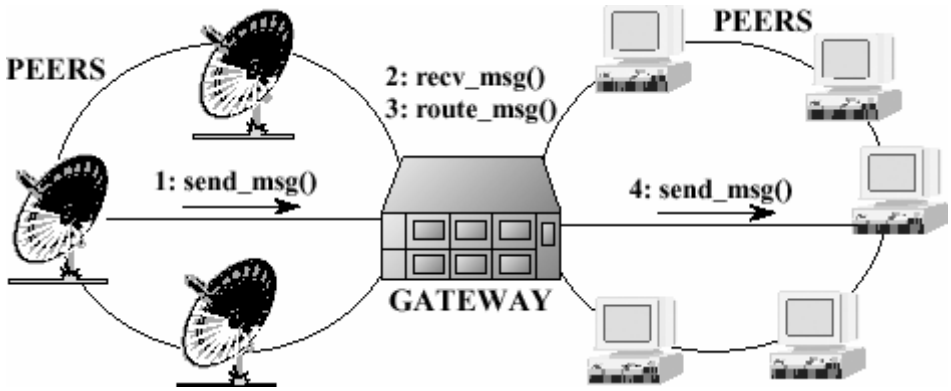

图 1-1 对端和应用级网关的结构和动力特性

这一部分描述怎样复用和扩展多种 ACE 组件，以实现图 1-2 所示的通信网关中的应用无关的和应用特有的组件。在此综述之后，1.3 检查在 ACE 组件之下的模式语言。

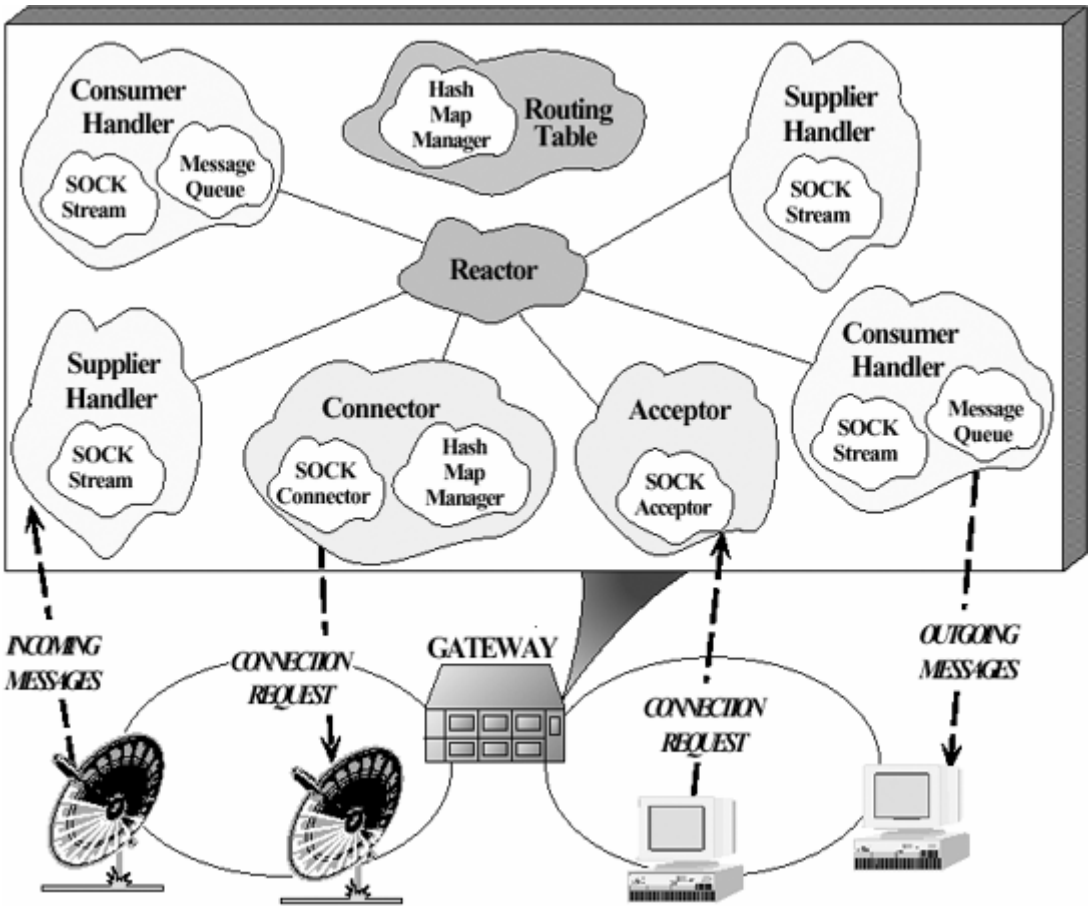

图 1-2 OO 网关软件体系结构

1.2.1 应用无关组件

图 1-2 中的大多数组件都基于 ACE 组件，后者可被复用于其他通信系统中。唯一不能被广泛复用的组件是Supplier和Consumer Handler，它们实现的是与消息格式和网关的路由协议有关的应用特有的细节。网关中应用无关组件的行为概述如下：

进程间通信（IPC）组件：SOCK Stream、SOCK Connector 和 SOCK Acceptor 组件封装 Socket 网络编程接口[9]。这些组件使用*包装外观*模式来实现[1]，该模式通过使开发者与低级、乏味和易错的 Socket 级编程屏蔽开来、简化了可移植通信软件的开发。此外，它们构成了下面描述的更高级的模式和 ACE 组件的基础。

事件多路分离组件：Reactor 是基于 1.3.3 描述的*反应器*模式的 OO 事件多路分离机制。它通过单一的多路分离点来在事件驱动的应用中引导外部的刺激。该设计允许单线程应用高效地在事件句柄之上等待、进行事件多路分离、并分派事件处理器。事件指示某件重要的事情已经发生，例如，新的连接或工作请求的到达。网关中主要的事件源是（1）连接事件，指示建立连接的请求，以及（2）数据事件，指示封装各种有效负载的路由消息，比如命令、状态消息和大块数据传输。

初始化和事件处理组件：在端点间建立连接涉及两种角色：（1）*被动*角色，在特定的地址上初始化通信端点，并被动地等待其他端点连接它，以及（2）*主动*角色，主动地发起连接到一或多个扮演被动角色的端点。Connector（连接器）和 Acceptor（接受器）是分别为初始化网络服务而实现主动和被动角色的工厂[2]。这些组件实现 1.3.5 中描述的*接受器－连接器*模式。网关使用这些组件来与对端建立连接，并生成初始化了的 Supplier 和 Consumer Handler。

为增加系统灵活性，连接可通过以下两种方式建立：

1. *从网关到对端*，常常在网关最初启动时、为建立对端的初始系统配置而进行。
2. *从对端到网关*，常常在系统运行后、无论何时有新对端想要发送或接收路由消息时进行。

在大型系统中，可能会有数打或数百对端连接到单个网关。因此，为加速初始化，网关的 Connector 可以异步地、而不是同步地发起所有连接。异步有助于在长延迟路径上（比如在卫星或长距陆基链路上构建的广域网（WAN））降低连接响应延迟。在 Connector 中包含的底层 SOCK Connector[9]提供低级的异步连接机制。当 SOCK Connector 经由 TCP 连接两个 Socket 端点时，它生成 SOCK Stream 对象，并将其用于在对端和网关之间交换数据。

消息多路分离组件：Map Manger（映射管理器）高效地将外部 id（比如对端路由地址）映射到内部 id（比如 Consumer Handler）。网关使用 Map Manger 来实现 Routing Table（路由表），后者在内部处理发给网关的消息的多路分离和路由。Routing Table 将对端发送的路由消息中包含的寻址信息映射到适当的 Consumer Handler 集。

消息排队组件：Message Queue（消息队列）[9]提供一种通用的排队机制，由网关用于在消息等待被路由给对端时、在 Consumer Handler 中对它们进行缓冲。可配置 Message Queue 来在单线程或多线程环境中高效而健壮地运行。当队列被实例化时，开发者可以选择所需的并发策略。在多线程环境中，Message Queue 使用 *Monitor Object*（管程对象）模式[1]来实现。

1.2.2 应用特有组件

在图 1-2 中仅有两个组件（Supplier 和 Consumer Handler）是特有于网关应用的。这些组件实现 1.3.6 中描述的非阻塞缓冲式 I/O（Non-blocking Buffered I/O）模式。Supplier 和 Consumer Handler 驻留在网关中，在那里它们被用作路由消息的初始源和目的地的代理；这些消息会被发送给在网络上的主机。这两种网关特有组件的行为概述如下：

Supplier Handler：Supplier Handler 负责将到来的消息路由给它们的目的地。当 Reactor 在连接的通信端点上检测到事件时，它就会通知 Supplier Handler。在 Supplier Handler 从该端点上接收完整的路由消息之后，它查询 Routing Table，以确定消息的 Consumer Handler 目的地集。随后它请求选中的 Consumer Handler 转发消息给适当的对端目的地。

Consumer Handler：Consumer Handler 负责将路由消息可靠地递送给它们的目的地。它实现了一种流控制机制，以缓冲由于暂时的网络拥塞或接收者缓冲区空间匮乏、不能立即发送而导致的路由消息的爆发。流控制是一种传输层机制，它确保源对端的数据发送速度不会超过目的对端缓冲和处理数据的能力。例如，如果目的对端的缓冲区空间耗尽，底层的 TCP 协议就会指示相关联网关的 Consumer Handler 停止发送消息，直到目的对端消耗了它现有的数据为止。

网关通过定制、实例化及组合上面描述的 ACE 组件，集成了应用特有和应用无关的组件。如图 1-3 所示，Supplier 和 Consumer Handler 继承自共同的祖先：由 Connector 和 Acceptor 生成的 ACE Svc Handler 类。每个 Svc Handler 都是一个远地相连对端的本地代理[2]。它包含的 SOCK Stream 使得对端可以通过已连接的 Socket 句柄交换消息。

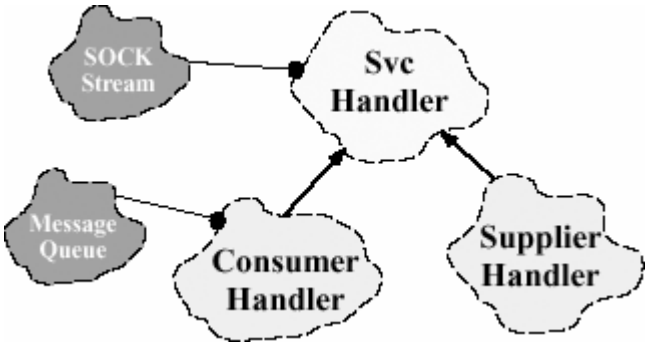

图 1-3 Consumer 和 Supplier Handler 继承层次

Consumer Handler 是根据非阻塞缓冲式 I/O 模式来实现的。因而，它使用 ACE Message Queue 来串连未发送的消息，其顺序为这些消息在流控制机制允许时所必须遵循的递送顺序。在被流控制的连接重又开放后，ACE 框架通知它的 Consumer Handler，后者就开始通过发送消息给对端来排空 Message Queue。

为提高可靠性和性能，本论文中描述的网关利用了传输控制协议（TCP）。TCP 为应用级网关提供了一种可靠、有序、非重复的字节流服务。尽管 TCP 连接天生是双向的，从对端发送给网关的数据使用了

与从网关发送给对端的数据不同的连接。以这样的方式分离输入连接和输出连接有若干优点：

－它简化了网关 Routing Table 的构造；
－它允许更为灵活的连接配置和并发策略；
－它增强了可靠性，因为如果在一个连接上发生错误，Supplier 和 Consumer Handler 可独立地重新连接。

1.3 用于应用级网关的模式语言

1.1 描述了应用级网关的结构和功能。尽管这样的体系结构综述有助于澄清网关中关键组件的行为，它并没有揭示在这些软件组件之下的更深的关系和角色。特别是，它并没有给出网关*为何*以这种特定方式设计、或为何特定组件要以特定方式动作和交互的动机。了解这些关系和角色对于开发、维护和增强通信软件来说是至关紧要的。

一种有效地捕捉和阐明这些关系和角色的方法是对生成它们的*模式语言*进行描述。研究在网关软件之下的模式语言提供了下面两种好处：

1. *确定常见设计挑战的成功解决方案*。在网关体系结构之下的模式语言超出了应用的特定细节，并解决了通信软件开发者所面临的常见挑战。对这种模式语言的彻底了解使得开发者能够将网关软件体系结构广泛地复用于其他系统中，即使是在复用它的算法、实现、接口或详细设计不可行的时候[10]。
2. *减少维护和增强网关软件的工作*。模式语言有助于捕捉多个类和对象之间的协作，并说明其动机。这对于必须维护和增强网关的开发者来说是很重要的。尽管网关设计中的角色和关系已包含在源码中，将它们从周围的实现细节中提取出来可能会是昂贵而易错的。

1.3.1 战略模式

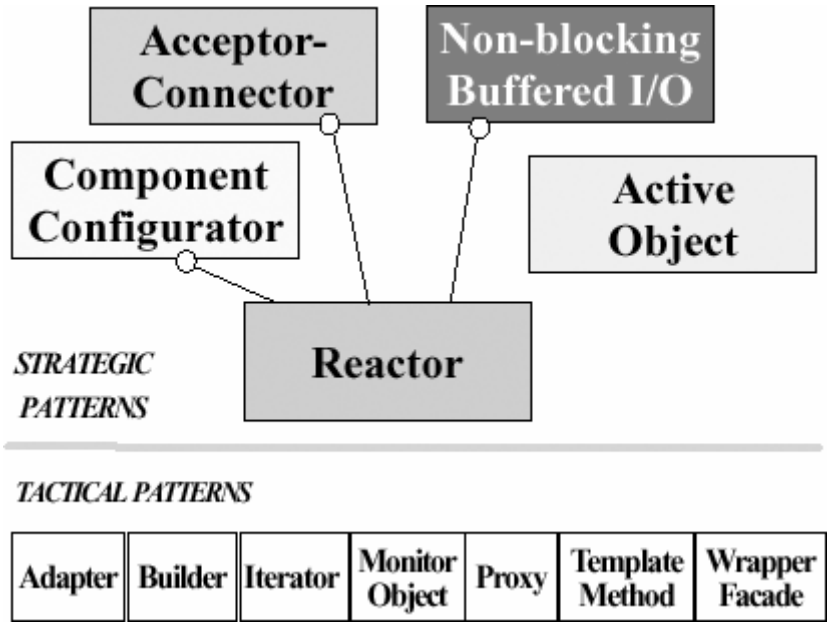

图 1-4 应用级网关模式语言

图 1-4 演示下面五种*战略*模式，它们构成了生成面向连接的应用级网关的模式语言的一部分：

－反应器（Reactor）[1]：该模式构造事件驱动应用，特别是服务器；它们并发地接收来自多个客户的请求，但依次对它们进行处理。
－主动对象（Active Object）[1]：该模式使方法执行与方法调用去耦合，以增强并发、并简化对驻留在它们自己的线程控制中的对象的同步访问。
－组件配置器[1]：该模式允许应用在运行时链接它的组件实现、或解除其链接，而不必修改、重编译或静态地重链接应用。它还可以在不关闭和重启运行中的进程的情况下，将组件重新配置进不同进程。
－接受器－连接器（Acceptor-Connector）[1]：该模式使网络系统中的连接建立及服务初始化与服务处理去耦合。
－非阻塞缓冲式 I/O（Non-blocking Buffered I/O）模式：该模式使输入机制与输出机制去耦合，以正确而可靠地路由数据，而又不会过度地阻塞应用处理。

这五种模式之所以是战略性的，是因为它们显著地影响特定领域中的应用的软件体系结构。例如，当网关遇到拥塞或失败时，1.3.6 描述的非阻塞缓冲式 I/O 模式能确保消息处理不会被中断或无限期地延迟。该模式有助于为使用可靠的面向连接的传输协议（比如 TCP/IP 或 IPX/SPX）的网关维持稳定的服务质量（QoS）。对于健壮、高效和可扩展的通信软件、比如应用级网关的开发来说，对本论文中描述的战略性通信模式的全面了解是必需的。

1.3.2 战术模式

应用级网关实现还使用了许多*战术*模式，比如：

－适配器（Adapter）[2]：该模式将不兼容的接口转换为客户可以使用的接口。网关使用此模式来统一地处理不同类型的路由数据，比如命令、状态信息，以及大块数据。
－构建器（Builder）[2]：该模式为复杂对象的渐进构建提供工厂。网关使用此模式来通过配置文件创建它的 Routing Table。
－迭代器（Iterator）[2]：该模式使对容器的连续访问与容器的表示去耦合。网关使用此模式来将多个 Supplier 和 Consumer Handler 连接到它们的对端，并对它们进行初始化。
－管程对象（Monitor Object）[1]：该模式对并发的方法执行进行同步，以确保在某一时刻只有一个方法在对象中运行。它还允许对象的多个方法相互协作地调度它们的执行序列。网关使用此模式来同步它的 Message Queue 的多线程配置。
－代理（Proxy）[2]：该模式提供了一种本地的代理对象，它代替一个远地对象进行工作。网关使用此模式来使主网关例行代码与（对端位于网络中其他主机上这一事实所导致的）延迟或错误屏蔽开来。
－模板方法（Template Method）[2]：该模式定义一种算法，其中的某些步骤是由派生类来提供的。网关使用此模式来有选择地重定义它的 Connector 和 Acceptor 组件中的特定步骤，以使失败的连接能够被自动重启。
－包装外观（Wrapper Façade）[1]：该模式将现有非 OO API 所提供的功能和数据封装在更为简洁、健壮、可移植、可维护和内聚的 OO 类接口中。ACE 框架使用此模式来提供一组 OS 无关的并发网络编程

组件；网关使用了这些编程组件。

比起前面概述的五种战略模式（它们是领域特有的并有着广泛的设计实现），这些战术模式是领域无关的，并只对软件设计有着相对局部的影响。例如，迭代器是一种战术模式，在网关中被用于顺序地处理 Routing Table 中的条目，而又不违反数据封装。尽管该模式是领域无关的，因而广泛地适用于许多应用，它所专事的问题并非像战略模式（比如非阻塞缓冲式 I/O 或反应器）那样深入而普遍地影响应用级网关软件的设计。但是，全面了解战术模式对于实现高度灵活、能够有弹性地回应应用需求和平台环境变化的软件来说仍是必要的。

这一部分的余下部分详细地描述各种战略模式，并解释它们是怎样被用于网关中的。

1.3.3 反应器模式

意图：反应器模式构造事件驱动的应用，特别是服务器；它们并发地接收来自多个客户的请求，但依次进行处理。

动机与压力：单线程应用必须处理来自多个来源的事件，而又不能无限期地阻塞在任何特定的来源上。下面的一些需要所带来的压力影响了单线程、事件驱动通信软件的设计：

1. *在单线程控制中高效地多路分离来自多个事件源的多种类型事件*。在应用进程中来自多个来源的事件常常必须在事件多路分离一级被处理。通过在这一级处理事件，应用就有可能不再需要更为复杂的线程、同步，或锁定。
2. *扩展应用行为、而又无需改动事件分派框架*。多路分离和分派机制常常是应用无关的，因而也就可以被复用。相反，事件处理器策略更加应用特有。通过分离这些事务，应用策略的改变可以不影响较低级的框架机制。

解决方案：应用反应器模式、以同步地等待指示事件在一或多个事件源（比如已连接 Socket 句柄）上的到达。将多路分离和分派这些事件的机制集成到处理它们的服务中。使这些多路分离和分派机制与在这些服务中对指示事件进行的应用特有的处理去耦合。

结构、参与者和实现：图 1-5 演示反应器模式中的结构和参与者。Reactor 定义的接口用于登记、移除和分派具体的事件处理器对象，比如网关中的 Supplier 或 Consumer Handler。该接口的实现提供了一组应用无关的机制。这些机制执行应用特有的事件处理器的多路分离和分派，以响应不同类型的输入、输出和定时器事件。

Event Handler 定义的抽象接口被 Reactor 用于分派回调方法；这些方法由登记了感兴趣的事件的对象定义。具体的事件处理器是继承自 Event Handler 的类，它有选择地重定义一些回调方法，以便通过应用特有的方式处理事件。

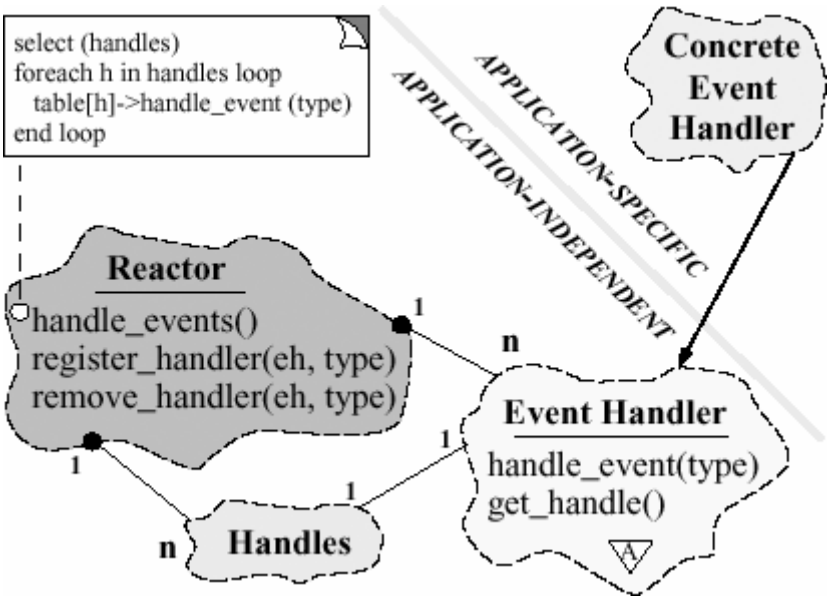

图 1-5 反应器模式中的结构和参与者

动力特性：图 1-6 演示反应器模式中参与者之间的动力特性。这些动力特性可划分为下面两种模式（mode）：

1. *初始化模式*，在其中 Concrete Event Handler 对象被登记到 Reactor；
2. *事件处理模式*，在其中 Reactor 调用已登记的对象上的 upcall，后者随即以一种应用特有的方式对事件进行处理。

使用：在网关中 Reactor 被用于以下类型的事件分派操作：

1. *输入事件*。Reactor 将每个到来的路由消息分派给与它的 Socket 句柄相关联的 Supplier Handler，在这里消息被路由到适当的 Consumer Handler。这种使用情况在图 1-7 中显示。
2. *输出事件*。如 1.3.6 和 1.3.7 所描述的，Reactor 确保外发的路由消息在已进行流控制的 Consumer Handler 之上被可靠地递送。
3. *连接完成事件*。Reactor 分派连接完成事件，该事件指示被异步发起的连接的完成状态。1.3.5 中描述的 Connector 组件使用了这些事件。
4. *连接请求事件*。Reactor 还分派指示被动发起的连接的到达的事件。1.3.5 中描述的 Acceptor 组件使用了这些事件。

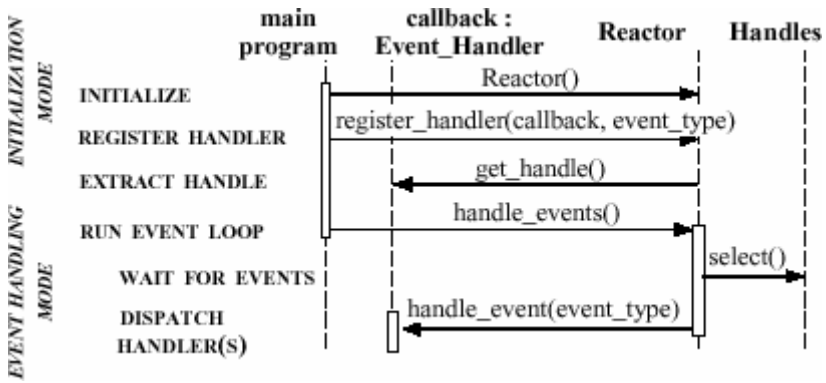

图 1-6 反应器模式的动力特性

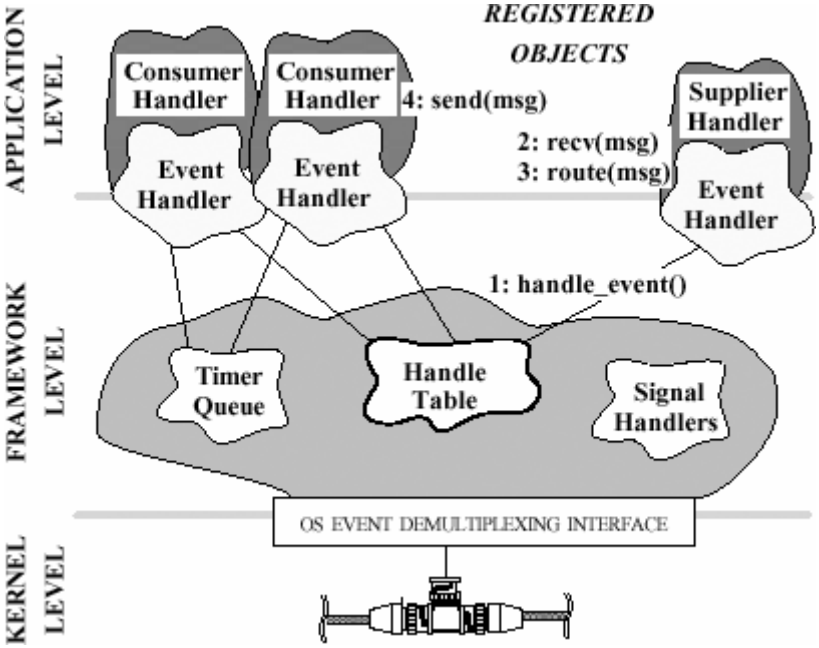

图 1-7 在网关中使用反应器模式

反应器模式已被用于许多单线程事件驱动框架中，比如 Motif、Interviews[11]、系统 V STREAMS[12]、ACE OO 通信框架[9]，以及 CORBA 的一些实现[8]。此外，它还为下面介绍的所有其他的战略模式提供事件多路分离基础构造。

1.3.4 组件配置器模式

意图：组件配置器模式允许应用在运行时链接它的组件实现、或解除其链接，而不必修改、重编译或是静态重链接应用。它还可以在不关闭和重启运行中的进程的情况下，将组件重新配置进不同进程。

动机和压力：以下需求所带来的压力影响了高度灵活和可扩展通信软件的设计：

1. *推迟选择组件的特定实现、直至设计周期中非常迟后的阶段。*推迟这些配置决策、直至安装时或运行时，显著地增加了开发者可用的设计选择。例如，可通过运行时上下文信息来指导实现决策，也可动态地将组件（重）配置进应用中。
2. *通过编写或构造多个独立开发的组件来构建完整应用。*应用的许多重复发生的组件配置和初始化行为应被分解进可复用方法中。这样的事务分离允许在运行时将新版本组件链接进应用中，而不用中断目前正在执行的组件。

解决方案：应用组件配置器模式、以使组件接口与它们的实现去耦合，并使应用不依赖组件实现被配置

进应用进程的时间点。

结构、参与者和实现：图 1-8 演示组件配置器模式的结构和参与者。针对其事件多路分离和分派需要，该模式复用了反应器模式的 Reactor 和 Event Handler。Component 是 Event Handler 的子类，并增加了在 C++对象被动态链接和解除链接时、用于初始化和终止它们的接口。应用特有的组件继承自 Component，并有选择地重定义它的 init 和 fini 方法，以分别实现定制的初始化和终止行为。

Component Repository 记录哪些 Component 正在进行链接和活动。Component Config 是一种外观（façade），它将其他组件的行为改编为“交响乐”[2]，并为运行时组件的链接、启用、挂起、恢复和解除链接提供单一访问点。

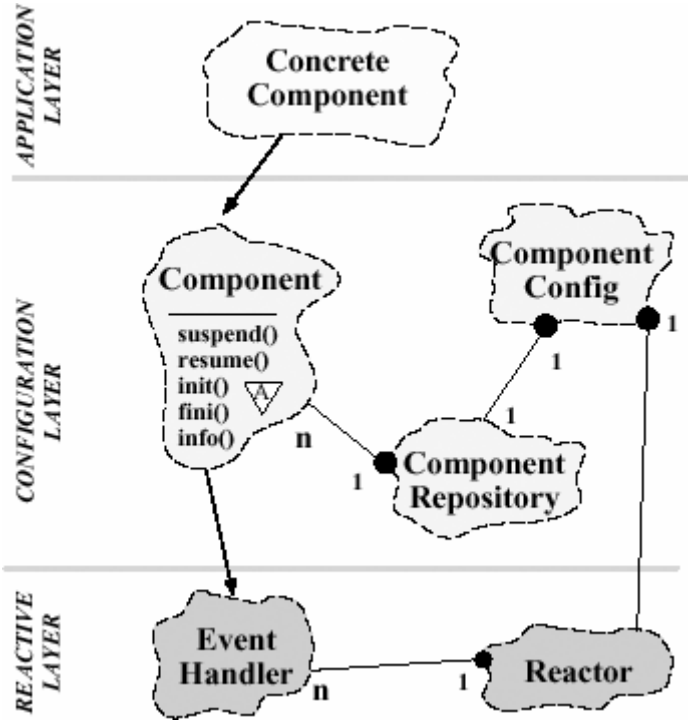

图 1-8 组件配置器模式中的结构和参与者

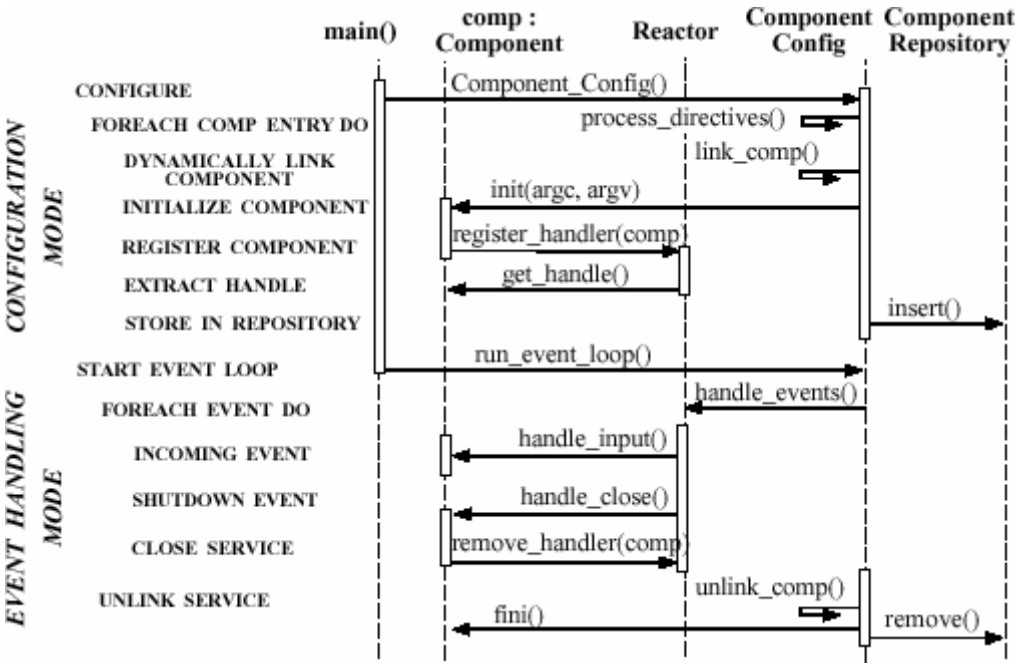

图 1-9 组件配置器模式的动力特性

动力特性：图 1-9 演示组件配置器模式中的参与者之间的动力特性。这些动力特性可被划分为以下两种模式（mode）：

1. *配置模式*，动态地将 Component 链接到应用，或从应用中解除其链接。
2. *事件处理模式*，使用像反应器或主动对象[1]这样的模式来处理到来的事件。

使用：在如图 1-10 所示的网关中使用了组件配置器模式。Reactive Gateway 组件是网关的一种单线程实现，它可通过配置脚本中的命令而被动态链接。要通过多线程实现来动态地替代这一组件，Component Config 只需重新查询它的 comp.conf 文件，解除 Reactive Gateway 的链接，动态地链接 Thread-per-Connection Gateway 或 Thread Pool Gateway，并初始化该新实现。Component Config 外观使用动态链接来高效地实现组件配置器模式。

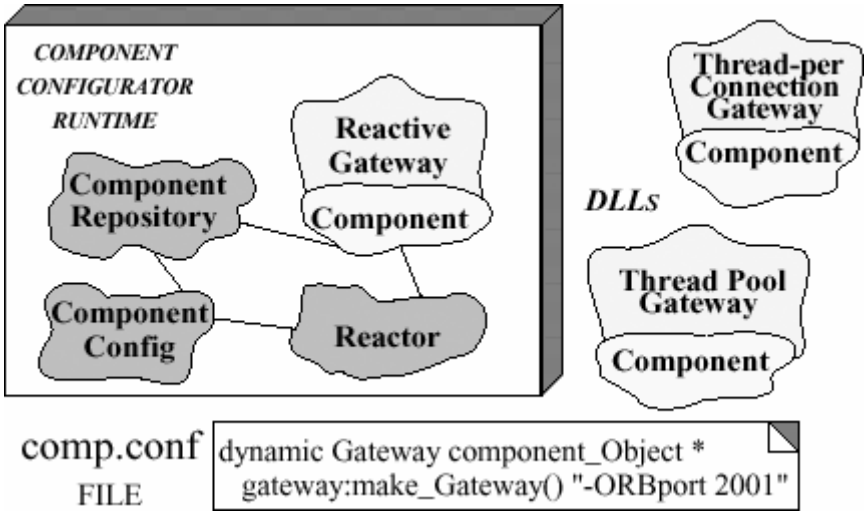

图 1-10 在网关中使用组件配置器模式

组件配置器模式被用于 Windows NT 服务控制管理器（SCM）中，它允许主 SCM 进程自动发起并控制管理员安装的服务。通常，现代操作系统（比如 Solaris、Linux 和 Windows NT）为动态配置的内核级服务驱动程序（它们实现组件配置器模式）提供了支持。组件配置器模式的另一种使用是 Java 中的 applet 机制，它支持 Java applet 的动态下载、初始化、启动、停止，以及终止。

1.3.5 接受器－连接器模式

意图：接受器－连接器模式使网络系统中的连接建立及服务初始化与服务处理去耦合。

动机和压力：面向连接的应用（比如我们的应用级网关）和中间件（比如 CORBA）常常使用像 scoket[13] 这样的较低级的网络编程接口来编写。以下需求所带来的压力影响了使用这些较低级接口编写的服务的初始化：

1. *将连接建立代码复用于每个新服务。*服务的关键特性，比如通信协议或数据格式，应该能够独立于用于建立连接的机制而透明地发展。因为服务特性的变动比连接建立机制更为频繁，分离这些事务有助于降低软件的耦合、并增加代码复用。
2. *使连接建立代码可跨越含有不同网络编程接口的平台而移植。*为接受连接和执行服务而对接受器－连接器的机制进行参数化有助于通过整个地替换这些机制来改善可移植性。这使得连接建立代码可跨越含有不同网络编程接口的平台进行移植，比如有 Socket、但没有 TLI，或者反之亦然。
3. *使用灵活的服务并发策略。*在连接建立后，对端应用使用此连接来交换数据，以执行某种类型的服务，比如远地登录或 HTML 文档传输。服务可以运行在单线程中、在多线程中，或多进程中，而不管连接是如何建立的，或如何被初始化的。
4. *确保被动模式的I/O 句柄不会被偶然用于读写数据。*通过彻底使连接建立逻辑与服务处理逻辑去耦合，被动模式的 Socket 端点不可能被不正确地使用（例如，试图在用于接受连接的被动模式侦听者端口上读写数据）。这消除了一类重要的网络编程错误。
5. *高效地主动与大量对端建立连接。*当应用必须在长延迟 WAN 上与大量对端高效地建立连接时，可能必须使用异步操作，并在非阻塞模式中发起和完成多个连接。

解决方案：应用接受器－连接器模式来使网络应用中的对端服务的连接及初始化与在它们被连接和初始化后所执行的服务处理去耦合。

结构、参与者和实现：图 1-11 演示接受器－连接器模式中的参与者的分层结构。Acceptor 和 Connector 组件是对连接和启用 Svc Handler 所需的资源进行装配的工厂。Svc Handler 是与相连对端交换消息的组件。

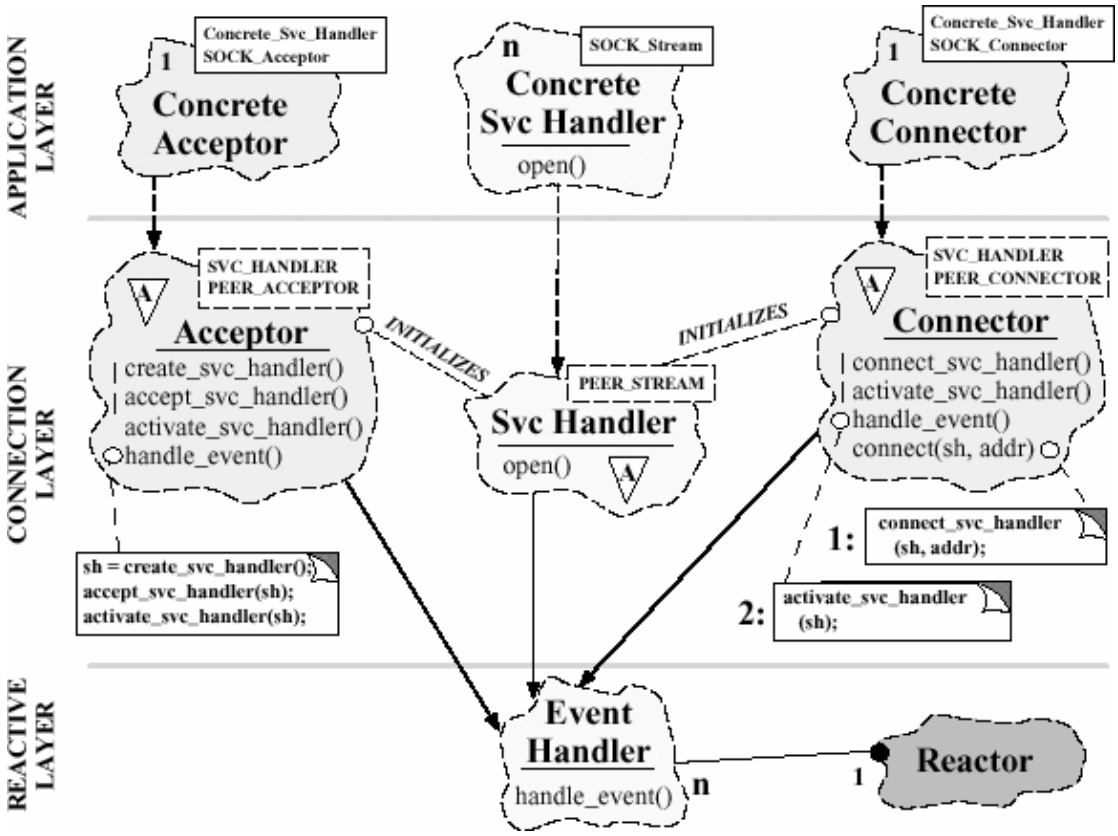

图 1-11 接受器－连接器模式中的结构和参与者

接受器－连接器的连接层中的参与者有效地利用了反应器模式。例如，在 Reactor 通知 Connector、一个先前发起的连接到对端的请求已经完成后，Connector 的异步初始化策略就会建立起一个连接。反应器模式的使用使得多个 Svc Handler 能在单线程控制中被异步地初始化。

为增加灵活性，Acceptor 和 Connector 组件可通过特定类型的 IPC 机制和 SVC HANDLER 来参数化。IPC 机制提供底层传输机制（比如 Socket 或 TLI 的 C++包装外观）来用于建立连接；而 SVC HANDLER 定义的抽象接口用于定义与相连对端进行通信的服务。Svc Handler 可通过 PEER STREAM 端点来参数化。当连接建立时，Acceptor 和 Connector 组件将此端点与它的对端关联起来。

通过从 Event Handler 继承，Svc Handler 可以登记到 Reactor，并使用反应器模式来在与 Acceptor 或 Connector 相同的线程控制中处理它的 I/O 事件。反过来，Svc Handler 也可以使用主动对象模式，在单独的线程中处理它的 I/O 事件。1.3.7 评估了在这两种模式之间进行的折衷。

参数化类型被用于使接受器－连接器模式的连接建立策略与服务类型及连接机制类型去耦合。开发者为这些类型提供模板参数，以生成应用层 Acceptor 或 Connector，比如由网关用于初始化 Supplier 和 Consumer Handler 的 Connector。这样的设计使得开发者可以整个地替换 SVC HANDLER 和 IPC 机制，而不会影响接受器－连接器模式的初始化策略。

注意通过使用[2]中描述的抽象工厂（Abstract Factory）或工厂方法（Factory Method）模式，经由继承和动态绑定可以达到类似程度的去耦合。之所以使用参数化类型来实现此模式，是因为它们改善了运行时效率。一般而言，模板为改善运行时性能而在编译及链接时开销与空间开销之间进行折衷。

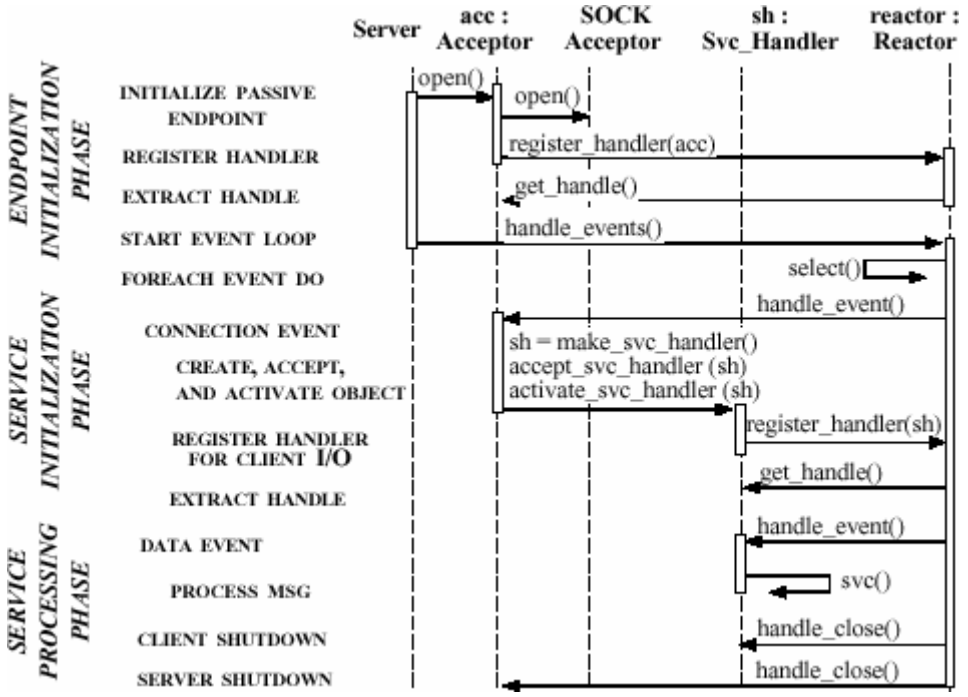

图 1-12 Acceptor 组件的动力特性

动力特性：图 1-12 演示该模式的 Acceptor 组件的参与者之间的动力特性。这些动力特性被划分进以下三个阶段：

1. *端点初始化阶段*，创建由 PEER ACCEPTOR 封装的被动模式端点，这个 PEER ACCEPTOR 被绑定到一个网络地址，比如 IP 地址和端口号。该被动模式端点侦听来自对端的连接请求。它被登记到 Reactor，后者驱动事件循环，在该端点上等待连接请求从对端到达。
2. *服务启用阶段*。因为 Acceptor 继承自 Event Handler，当连接请求事件到达时，Reactor 可以分派 Acceptor 的 handle_event 方法。该方法执行 Acceptor 的 Svc Handler 初始化策略：（1）装配创建新 Concrete Svc Handler 对象所需的资源，（2）将连接接受进该对象，以及（3）通过调用 Svc Handler 的 open 挂钩方法将其启用。
3. *服务处理阶段*。在 Svc Handler 被启用后，它处理在 PEER STREAM 上到来的事件消息。Svc Handler 可以使用像反应器或主动对象[1]这样的模式来处理到来的事件消息。

在该模式的 Connector 组件的参与者之间的动力特性可被划分进下面三个阶段：

1. *连接发起阶段*，主动地连接一或多个 Svc Handler 到它们的对端。连接可被同步或异步地发起。Connector 的 connect 方法实现主动建立连接的策略。
2. *服务初始化阶段*，当 Svc Handler 的连接成功完成时，通过调用它的 open 方法来将其启用。Svc Handler 的 open 方法随即执行服务特有的初始化。
3. *服务处理阶段*，使用在 Svc Handler 和与其相连的对端之间交换的数据来执行应用特有的服务处理。

图 1-13 使用*异步*连接建立来演示动力特性的这三个阶段。注意 Connector 的连接发起阶段是怎样*暂时*与服务初始化阶段分离的。这样的设计使得多个连接发起可在单线程控制中并行地进行。同步连接建立的动力特性也是类似的。在此例中，Connector 将连接发起和服务初始化阶段组合进单个阻塞操作中。

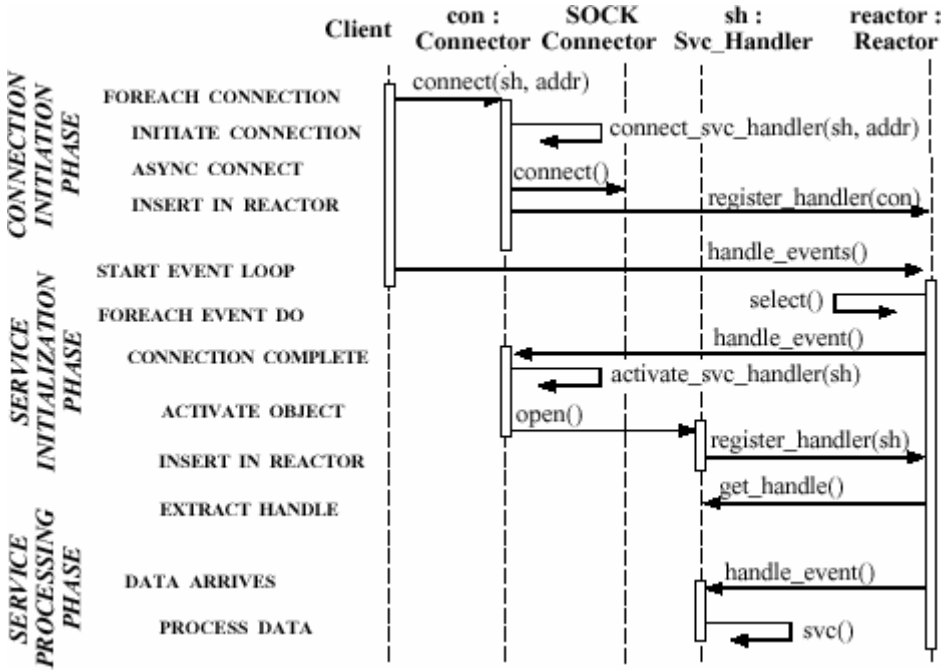

图 1-13 异步 Connector 组件的动力特性

一般而言，同步的连接建立对于以下情况来说是有用的：

－如果建立连接的响应延迟非常低，比如通过回路（loopback）设备与在同一主机上的服务器建立连接。
－如果有多个线程控制可用，并且，使用不同线程来同步地连接每个 Svc Handler 是可行的。
－如果不到连接已建立，客户应用无法执行有用的工作。

相反，异步的连接建立对于以下情况来说是有用的：

－如果连接响应延迟很高，而又有许多对端要连接，例如，在高响应延迟的 WAN 上建立大量连接。
－如果只有单个线程控制可用，例如，如果 OS 平台不提供应用级线程。
－如果客户应用必须在连接正在被建立的同时执行另外的工作，比如刷新 GUI。

情况常常是，网络服务（比如我们的应用级网关）必须在不知道它们将同步还是异步地进行连接的情况下开发。因此，一个通用网络编程框架所提供的组件必须支持多种同步和异步的使用情况。

接受器－连接器模式通过使连接建立逻辑与服务处理逻辑相分离，增加了网络框架组件的灵活性和复用。在（1）Acceptor 及 Connector 组件与（2）Svc Handler 之间的唯一耦合发生在服务初始化阶段，在 Svc Handler 的 open 方法被调用时。在这时，Svc Handler 可以使用任何合适的应用级协议或并发策略来执行它的服务特有的处理。

例如，当消息到达网关时，Reactor 可用于分派 Supplier Handler，以划分消息帧、确定外发路由，并递送消息给它们的 Consumer Handler。但是，Consumer Handler 可以使用不同类型的并发机制（比如 1.3.7 描述的主动对象）来发送数据给远地的目的地。

使用：图 1-14 演示接受器－连接器模式的 Acceptor 组件是怎样在网关扮演被动连接角色时被使用的。在此例中，对端连接到网关，后者使用 Acceptor 来使 Supplier 和 Consumer Handler 的被动初始化与处理器

被初始化后所执行的路由任务去耦合。

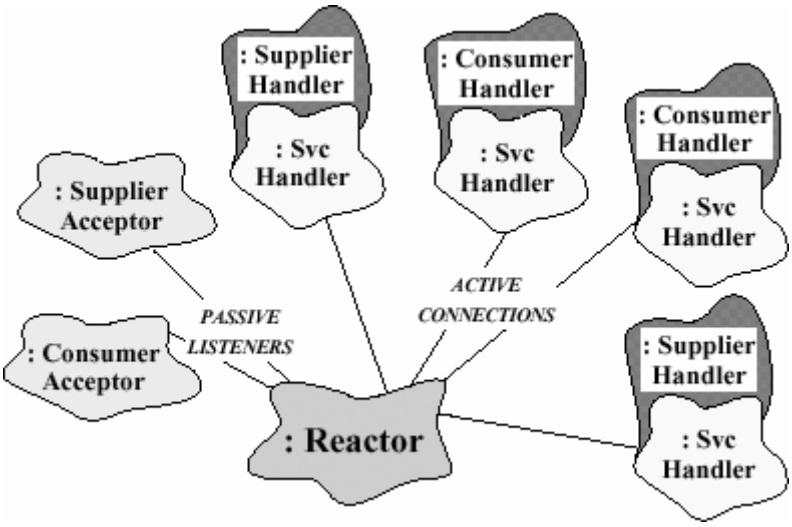

图 1-14 在网关中使用 Acceptor 组件

图 1-15 演示接受器－连接器模式的 Connector 组件怎样被网关用于简化连接大量对端的任务。在此例中，对端地址在网关初始化的过程中从配置文件中读取。网关使用构建器（Builder）模式[2]来将这些地址绑定到动态分配的 Consumer Handler 或 Supplier Handler。因为这些处理器继承自 Svc Handler，可以使用迭代器模式[2]来异步地发起所有连接。随后就使用 Connector 来并行地完成连接。

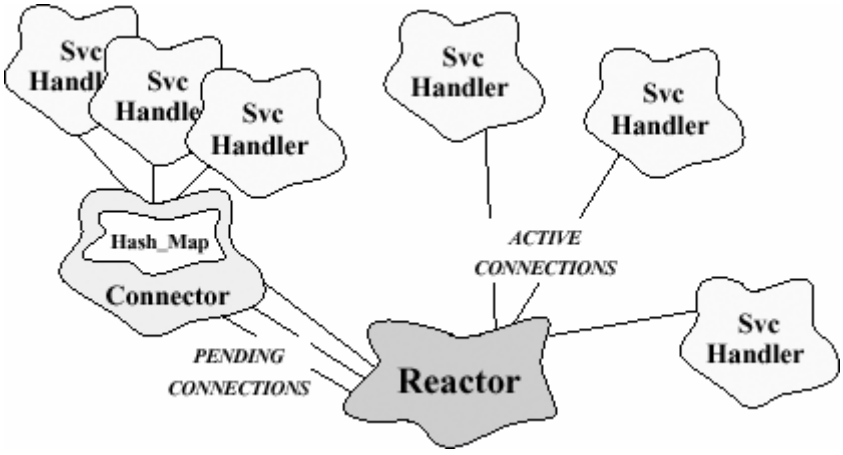

图 1-15 在网关中使用 Connector 组件

图 1-15 显示 Connector 在四个连接被建立后的状态。其他三个还未完成的连接由 Connetor 拥有。如该图中所示，Connector 维护连接还暂未完成的三个 Handler 的表。在连接完成时，Connector 将每个已连接的 Channel 从它的表中移除，并将其启用。在单线程实现中，Supplier Handler 在它们被启用后将自身登记到 Reactor。自此时起，当路由消息到达时，Supplier Handler 接收它们、并转发给 Consumer Handler，后者再将这些消息递送到它们的目的地（这些活动在 1.3.6 中描述）。

除了建立连接，网关还可与 Connector 和 Reactor 联合使用、以确保连接在网络错误发生时重启。通过确保信道在意外断开时（例如，如果对端崩溃，或由于网络拥塞、大量的数据在 Consumer Handler 处

排队等待）自动地重新发起，增强了网关的容错性。如果连接意外失败，通过使用 Reactor 的定时器分派能力，一种指数后退算法可以高效地重启连接。

在像 inentd[13]和 listen[14]这样的网络服务器管理工具中，可以发现接受器－连接器模式的意图和综合体系结构。这些工具利用主接受器进程，在一组通信端口上侦听连接。每个端口被关联到一个与通信有关的服务（比如标准的 Internet 服务 ftp、telnet、daytime 和 echo）。当服务请求到达被监控的端口时，接受器进程接受该请求，并分派适当的预登记处理器来执行服务。

1.3.6 非阻塞缓冲式 I/O 模式

意图：非阻塞缓冲式 I/O 模式使输入机制与输出机制去耦合，从而使数据能被正确而可靠地路由，而不会不适当地阻塞应用处理。

动机与压力：当在到来或外发的网络连接上发生拥塞或失败时，不能中断或无限期地延后网关中的消息路由。因而，在构建健壮的面向连接网关时，必须消除以下需求所带来的压力：

1. *防止行为不端的连接破坏行为良好的连接的 QoS*。输入连接可能会因为对端断开而失败。同样地，作为网络拥塞的结果，输出连接也可能进行流控制。在这些类型的情况下，网关不能在任何单个连接上执行阻塞式 send 或 recv 操作，因为（1）整个网关可能会无限期挂起，或是（2）在其他连接上的消息无法发送或接收，提供给对端的 QoS 将下降。
2. *允许为处理输入和输出使用不同的并发策略*，包括（1）使用反应器模式（1.3.3）的单线程处理，以及（2）使用主动对象模式（1.3.7）的多线程处理。取决于各种因素，比如 CPU 数、上下文切换开销，以及对端数，各种策略适用于各不相同的情况。

解决方案：应用非阻塞缓冲式 I/O 模式、使输入处理与输出处理去耦合，以防止阻塞，并允许将定制的并发策略灵活地配置进应用中。

结构、参与者和实现：图 1-16 演示非阻塞缓冲式 I/O 模式中的参与者的层次结构。I/O 层为 Supplier Handler 提供事件源，为 Consumer Handler 提供事件槽。Supplier Handler 使用 Routing Table 来将路由消息映射到一或多个 Consumer Handler。如果消息不能立即被递送到它们的目的地，就会缓冲在 Message Queue 中，以在后面进行传输。

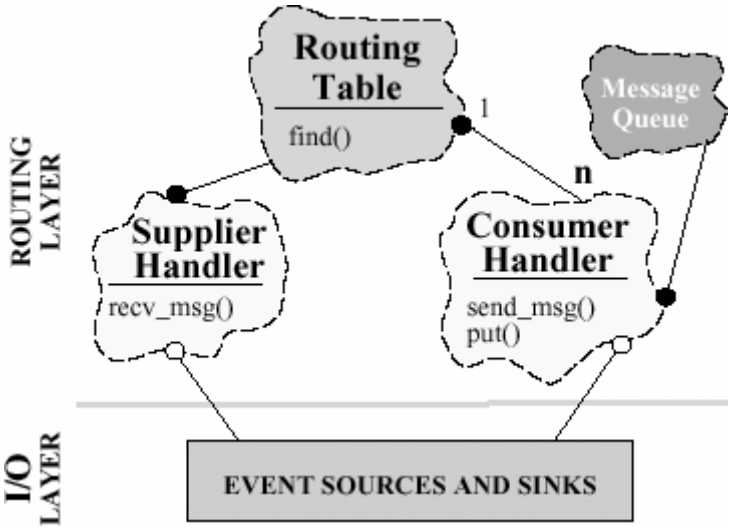

图 1-16 非阻塞缓冲式 I/O 模式中的结构和参与者

因为 Supplier Handler 与 Consumer Handler 是去耦合的，它们的实现可以独立地变化。这样的事务分离十分重要，因为它允许为输入和输出使用不同的并发策略。这样的去耦合的效果在 1.3.7 中进一步讨论。

动力特性：图 1-17 演示非阻塞缓冲式 I/O 模式中的参与者之间的动力特性。这些动力特性可被划分进三个阶段：

1. *输入处理阶段*，Supplier Handler 在其中将到来的 TCP 片段重新装配进完整的路由消息，而又不阻塞应用进程。
2. *路由选择阶段*。在重新装配了完整的消息后，Supplier Handler 查询 Routing Table，以选择负责发送路由消息给它们的对端目的地的 Consumer Handler。
3. *输出处理阶段*，所选择的 Consumer Handler 在其中传输路由消息给它们的目的地，而又不阻塞应用进程。

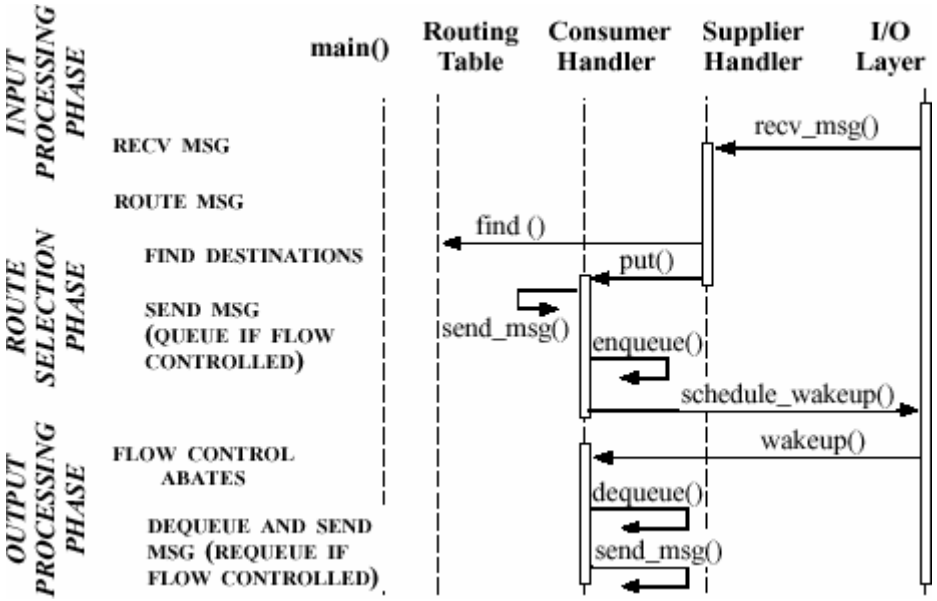

图 1-17 非阻塞缓冲式 I/O 模式的动力特性

使用：在本论文中的其他战略模式——也就是，反应器、连接器、接受器，以及主动对象——可被应用于许多类型的通信软件。相反，非阻塞缓冲式 I/O 模式更紧密地与在对端间路由消息的网关风格的应用耦合在一起。构建可靠的面向连接的网关的主要挑战集中在避免阻塞式 I/O 上。网关需要可靠地对发生在连接上的流控制进行管理；这些连接被 Consumer Handler 用于转发消息给对端。如果网关在拥塞的连接上进行发送时无限期地阻塞，到来的消息就无法被路由，即使其他消息是去往未进行流控制的 Consumer Handler。

1.3.6 的余下部分描述怎样在网关的单线程反应式版本中实现非阻塞缓冲式 I/O 模式（1.3.7 检查非阻塞缓冲式 I/O 模式的多线程主动对象版本）。在此实现中，非阻塞缓冲式 I/O 模式使用 Reactor 来作为网关 I/O 操作的协作式多任务调度器；这些操作在单线程中的不同连接上进行。使用单线程可消除以下开销：

－*同步*：例如，对像 Routing Table 这样的共享对象的访问不需要序列化；以及
－*上下文切换*：例如，所有消息路由都可在单个线程中发生。

在非阻塞缓冲式 I/O 模式的反应式实现中，Supplier Handler 和 Consumer Handler 是 Event Handler 的后代。这样的层次化继承设计使得网关可以在消息到达和流控制情况平息时、通过让 Reactor 分别分派 Supplier 和 Consumer Handler 的 handle_event 方法来路由消息。

使用反应器模式来实现非阻塞缓冲式 I/O 模式涉及以下步骤：

1. *初始化非阻塞式端点*。在 Supplier 和 Consumer Handler 被 Acceptor 或 Connector 启用后，它们的句柄被设置进非阻塞模式。非阻塞式 I/O 的使用对于避免在拥塞的网络链接上发生阻塞来说是必要的。
2. *输入消息的重新装配和路由*。Supplier Handler 以片段的方式接收路由消息。如果整个消息不是立即可用，Supplier Handler 必须缓冲这些片段，并将控制返回给事件循环。这对于防止 Supplier Channel 上的“ head of line ”阻塞来说是必要的。当 Supplier Channel 成功地接收了整个消息、并划分出消息帧时，它使用 Routing Table 来确定适当的递送该消息的 Consumer Handler 集。
3. *消息递送*。所选择的 Consumer Handler 尝试发送消息给目的对端。消息必须以“ 先进先出 ”（FIFO）顺序可靠地递送。为避免阻塞，所有 Consumer Handler 中的 send 操作必须进行检查，以确定网络链接没有进行流控制。如果没有进行流控制，消息才能够被成功发送。图 1-18 右上角的 Consumer Handler 描述了这一情况。但是如果链接已进行流控制，非阻塞缓冲式 I/O 模式实现必须使用不同的策略。图 1-18 右手较低的 Consumer Handler 描述了这一情况。

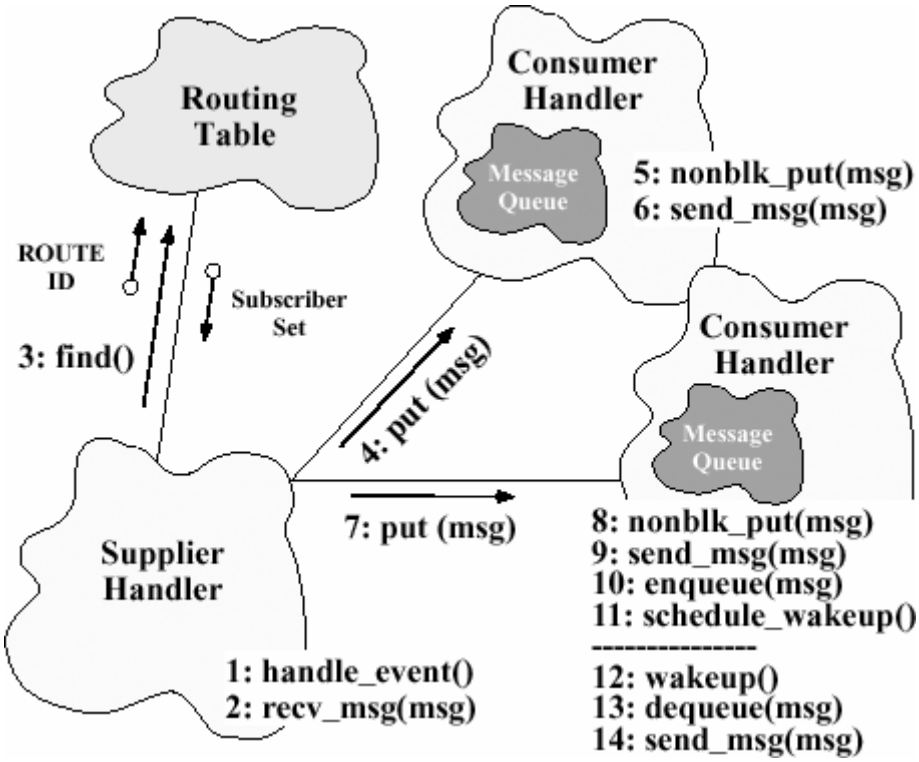

图 1-18 在单线程反应式网关中使用非阻塞缓冲式 I/O 模式

为处理已进行流控制的连接，Consumer Handler 将它正试图发送的消息插入它的 Message Queue 中。它随即指示 Reactor 在流控制情况减轻时回调 Consumer Handler；并返回主事件循环。在可以再次尝试发送时，Reactor 分派 Consumer Handler 的 handle_event 方法，后者就再次尝试该操作。这一系列步骤可能重复多次，直到整个消息被成功地传送。

注意在每次 I/O 操作后，网关总是立即将控制返回给它的主事件循环，而不管它是否发送或接收了整个消息。这是非阻塞缓冲式 I/O 模式的本质——它将消息正确地路由给对端，而不阻塞在任何单个 I/O 信道上。

1.3.7 主动对象模式

意图：主动对象模式使方法执行与方法调用去耦合，以增强并发、并简化对驻留在它们自己的线程控制中的对象的同步访问。

动机和压力：1.3.6 中的单线程网关所用的所有战略模式的层次都在反应器模式之上构建。接受器－连接器和非阻塞缓冲式 I/O 模式都使用反应器来作为在单线程控制中初始化和路由消息的调度器/分派器。一般而言，反应器模式构成了单线程反应式系统中的中央事件循环。例如，在单线程网关实现中，Reactor 提供了并发控制的粗粒度形式，对在进程中的事件多路分离和分派层上的事件处理器的调用进行序列化。这消除了在网关中对额外的同步机制的需要，并使上下文切换开销得以最小化。

反应器模式良好地适用于使用短持续时间回调的应用，比如接受器模式中的被动连接建立。但是，

对于长持续时间操作，比如在网络拥塞期间在已进行流控制的 Consumer Handler 上的阻塞，这种模式并不十分适用。事实上，非阻塞缓冲式 I/O 模式实现中的许多复杂性都源于使用反应器模式来作为协作式多任务机制。一般而言，该模式不足以消除下面的需求所带来的压力：

- *确保在某端点上的阻塞式读写操作不会降低其他端点的 QoS*。如果使用阻塞式 I/O、而不是非阻塞反应式 I/O，网络服务往往更易于编程。之所以有这样的简单性，是因为执行状态可以局限在线程的启用记录（activation record）中，而不是分散在一组由应用开发者显式地维护的控制块中。

解决方案：应用主动对象模式来使在对象上的方法调用与方法执行去耦合。方法调用应该在客户的线程控制中发生，而方法执行应该在另外的线程中发生。此外，所提高的接口应该使客户线程看起来像是在调用普通的方法。

结构、参与者和实现：图 1-19 演示主动对象模式中的结构和参与者。Proxy 将主动对象的公共方法输出给客户。Scheduler 基于同步和调度约束来确定下一个要执行的方法。Activation List 维护待处理 Method Request 的队列。Scheduler 确定这些 Method Request 被执行的顺序（在网关中使用了 FIFO 调度器来维护消息递送的顺序）。Servant 维护实现方法所共享的状态。

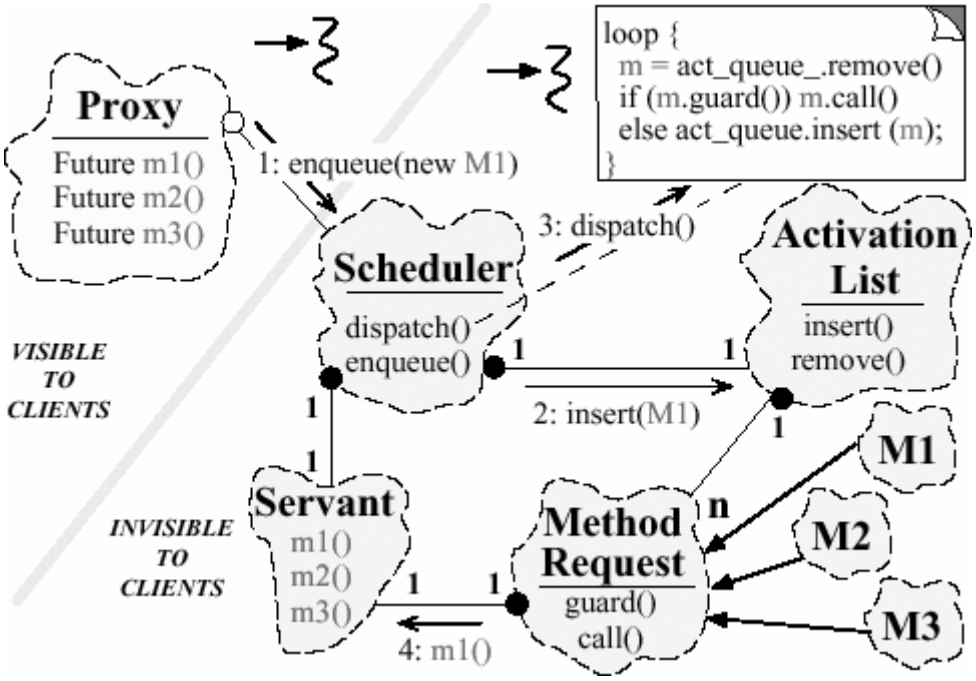

图 1-19 主动对象模式中的结构和参与者

动力特性：图 1-20 演示主动对象模式中的参与者之间的动力特性。这些动力特性被划分进下面的三个阶段：

1. *方法请求构造*。在此阶段，客户应用调用 Proxy 定义的方法，从而触发 Method Request 的创建；Method Request 维护绑定到方法的参数，以及任何其他的方法执行和返回结果所需的绑定。一个到 Future（期货）对象的绑定被返回给方法的调用者。
2. *调度/执行*。在此阶段 Scheduler 获取一个互斥锁，查询 Activation Queue 来确定哪些 Method Request

满足同步约束。Method Request 随即被绑定到当前的 Servant，并可以访问/更新该 Servant 的状态。

3. *返回结果*。最后的阶段将 Method Request 的结果绑定到 Future[15]对象，后者在方法结束执行时将返回值返回给调用者。Future 是一种同步对象，它强制实现“一次写，多次读”同步。随后，任何与此 Future 对象会合（rendezvous）的读者都将对期货进行估算并获取结果值。当 Future 和 Method Request 不再被需要时，它们可被当作垃圾回收。

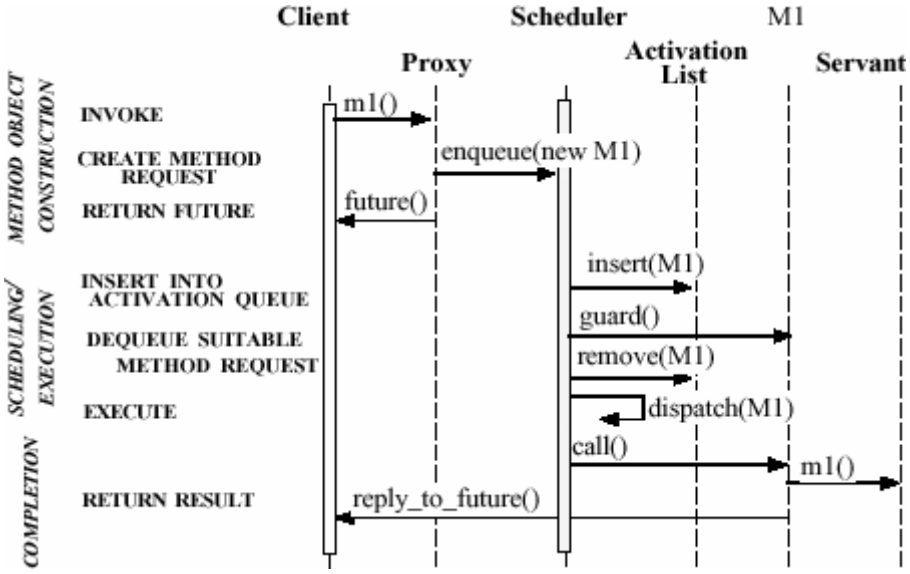

图 1-20 主动对象模式的动力特性

使用：1.3.6 中描述的网关实现是单线程的。它使用非阻塞缓冲式 I/O 模式的反应器模式实现来作为协作式多任务调度器，以分派网关感兴趣的事件。在我们实现了许多单线程网关之后，下面的事实变得清楚起来：使用反应器模式来作为所有网关路由 I/O 操作的基础是易错和难以维护的。例如，开发者很难记住为什么在不能进行 I/O 操作控制时、必须将控制迅速地返回给 Reactor 的事件循环。在单线程网关中，这样的误解变成了错误的常见来源。

为避免这些问题，有许多多线程网关使用了主动对象模式的各种变种来构建。该模式允许 Consumer Handler 在发送消息给对端时独立地阻塞。这一部分的余下部分描述 Consumer Handler 怎样使用主动对象模式来成为多线程的[1]。这样的修改充分简化了非阻塞缓冲式 I/O 模式的实现，因为 Consumer Handler 可以阻塞在它们自己的主动对象线程中，而不会影响其他的 Handler。将 Consumer Handler 实现为主动对象还消除了在使用 Reactor 调度 Consumer Handler 时所需的微妙而易错的协作式多任务编程技术。

图 1-21 演示非阻塞缓冲式 I/O 模式的主动对象版本。注意与图 1-18 中的反应器方案相比，它有多么的简单。之所以会这样，是因为复杂的输出调度逻辑被移进了主动对象，而不是成为应用开发者的责任。

[1] 可以将主动对象模式应用到 Supplier Handler，但这对网关设计只有很小的影响，因为 Reactor 已经支持非阻塞式输入。

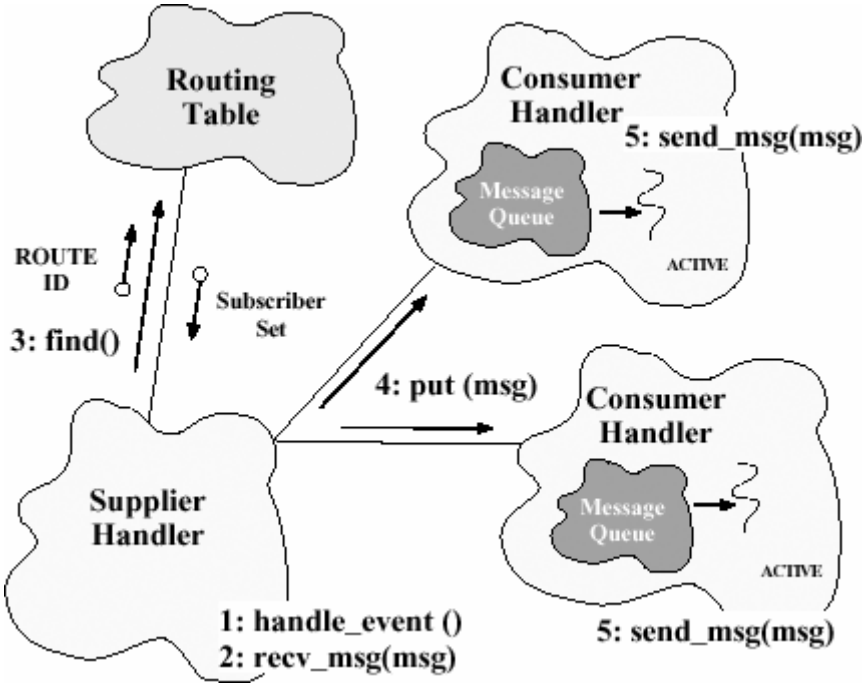

图 1-21 在多线程主动对象网关中使用非阻塞缓冲式 I/O 模式

通过检查在产品网关系统中实现非阻塞缓冲式 I/O 模式的源码，还可以观察到单线程和多线程网关之间的复杂性差异。由于所有围绕着实现的错误处理代码和协议特有的细节，很难通过检查源码来简单地确定这些复杂性的原因。这些细节倾向于掩盖关键的洞见：*在单线程和多线程方案之间的复杂性的主要差异来自于选择了反应器模式，而不是主动对象模式*。

本论文明确地探讨反应器和主动对象模式之间的交互和权衡，从而揭示不同设计选择的效果。一般而言，为紧密相关的模式之间的交互和关系编写文档是有挑战性的、未解决的课题；模式社群正在致力于这一课题的解决。

1.4 相关模式

[2, 16, 1]标识、命名、分类许多基本的的体系结构和设计模式。这一部分检查本论文中描述的模式与文献中的其他模式的关联。注意 1.3.2 中概述的许多战术模式构成了实现本论文中介绍的战略模式的基础。

反应器模式与观察者（Observer）模式[2]有关。在观察者模式中，当一个主题变动时，*多个*相关对象会被自动更新。在反应器模式中，当一个事件发生时，会自动分派一个处理器。因而，尽管事件源可能有多个，反应器为每个事件分派*单个*的处理器。反应器模式还提供了一个外观[2]。外观模式给出一种接口，使应用与子系统中复杂的关系屏蔽开来。反应器模式使应用与执行事件多路分离和事件处理器分派的复杂机制屏蔽开来。

组件配置器模式与构建器（Builder）和中介者（Mediator）模式[2]有关。构建器模式提供的工厂用于渐进地构造复杂对象。中介者协调在它的关联者之间的交互。组件配置器模式提供的工厂用于在运行时将组件配置和初始化进应用中。在运行时，组件配置器模式允许应用所提供的组件被渐进地修改，而不用中断执行中的组件。此外，组件配置器模式还在运行时对被配置进应用的组件与想要更新、挂起、恢复或移除组件的外部管理员之间的交互进行协调。

接受器－连接器模式与模板方法（Template Method）、策略（Strategy）以及工厂方法（Factory Method）模式[2]有关。在模板方法模式中，所编写的算法的某些步骤由派生类来提供。在工厂方法模式中，子类中的方法创建一个伙伴来执行一项特定的任务，但此任务并没有与用于创建该任务的协议耦合在一起。接受器－连接器模式中的 Acceptor 和 Connector 组件是使用模板方法或策略来创建、连接并启用通信信道的处理器的工厂。接受器－连接器模式的意图与[16]中描述的客户/分派器/服务器模式是类似的。它们都关心主动连接建立与后续服务的分离。主要的区别是接受器－连接器模式同时致力于被动/主动及同步/异步的连接建立。

非阻塞缓冲式 I/O 模式与中介者模式有关，后者使软件系统的协作组件去耦合，并允许它们相互交互，而又没有直接的相互依赖。非阻塞缓冲式 I/O 模式专用于消除与网络通信相关的压力。它使用于处理输入消息的机制与用于处理输出消息的机制去耦合，以防止阻塞的发生。此外，非阻塞缓冲式 I/O 模式还允许为输入和输出信道使用不同的并发策略。

1.5 结束语

通过本论文阐释的模式语言，开发者可以在产品通信网关中广泛地复用设计专家经验和软件组件。该语言中的模式阐明了执行核心通信软件任务的对象的结构以及它们之间的协作。这些模式所关注的任务包括事件多路分离和事件处理器分派，应用服务的连接建立和初始化、并发控制，以及路由。

本论文中描述的模式语言和 ACE 框架组件已被作者和他的同事复用于许多产品通信软件系统中，范围从电信、电子医学成像及航空控制项目[10, 5, 7]到学院研究项目[9, 8]。总而言之，通过在高于（1）源码和（2）聚焦于个体对象和类的 OO 设计模式的水平上捕捉软件体系结构中的参与者的结构和动力特性，该模式语言帮助了这些系统中的组件和框架的开发。

对我们使用模式所得到的经验和教训的深入讨论见[4]。基于 ACE、演示本论文中所有模式的单线程和多线程网关的例子可在 http://www.cs.wustl.edu/~schmidt/ACE.html 自由获取。

参考文献

[1] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrency and Distributed Objects, Volume* 2. New York, NY: Wiley & Sons, 2000.

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

[3] J. O. Coplien and D. C. Schmidt, eds., *Pattern Languages of Program Desig*n. Reading, MA: Addison-Wesley, 1995.

[4] D. C. Schmidt, “Experience Using Design Patterns to Develop Reuseable Object-Oriented Communication Software,” *Communications of the ACM (Special Issue on Object-Oriented Experiences*), vol. 38, October 1995.

[5] I. Pyarali, T. H. Harrison, and D. C. Schmidt, “Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging,” in *Proceedings of the 2nd Conference on Object-Oriented Technologies and System*s, (Toronto, Canada), USENIX, June 1996.

[6] D. C. Schmidt and T. Suda, “Measuring the Performance of Parallel Message-based Process Architectures,” in *Proceedings of the Conference on Computer Communications (INFOCOM*), (Boston, MA), pp. 624–633, IEEE, April 1995.

[7] T. H. Harrison, D. L. Levine, and D. C. Schmidt, “The Design and Performance of a Real-time CORBA Event Service,” in *Proceedings of OOPSLA* ’97, (Atlanta, GA), ACM, October 1997.

[8] D. C. Schmidt and C. Cleeland, “Applying a Pattern Language to Develop Extensible ORB Middleware,” in *Design Patterns in Communications* (L. Rising, ed.), Cambridge University Press, 2000.

[9] D. C. Schmidt, “Applying Design Patterns and Frameworks to Develop Object-Oriented Communication Software,” in *Handbook of Programming Languages* (P. Salus, ed.), MacMillan Computer Publishing, 1997.

[10] 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), ACM, August 1995.

[11] M. A. Linton, J. Vlissides, and P. Calder, “Composing User Interfaces with InterViews,” *IEEE Compute*r, vol. 22, pp. 8–22, February 1989.

[12] D. Ritchie, “A Stream Input–Output System,” *AT&T Bell Labs Technical Journa*l, vol. 63, pp. 311–324, Oct. 1984.

[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] R. H. Halstead, Jr., “Multilisp: A Language for Concurrent Symbolic Computation,” *ACMTrans. Programming Languages and System*s, vol. 7, pp. 501–538, Oct. 1985.

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

第 2 章 JAWS：高性能 Web 服务器框架

James C. Hu　　Douglas C. Schmidt[2]

摘 要

通信软件的开发者面临着许多挑战。通信软件包含着固有的复杂性，比如错误检测和恢复；以及随机的复杂性，比如关键概念和组件的持续的重新发现和发明。应对这些挑战需要对面向对象应用框架和模式有全面的了解。本论文阐释我们是怎样将通信软件的框架和模式应用于开发称为 JAWS 的高性能 Web 服务器的。

JAWS 是一种面向对象的框架，支持多种 Web 服务器策略的配置，比如使用异步 I/O 和 LRU 缓存的线程池并发模型 vs. 使用同步 I/O 和 LFU 缓存的 Thread-per-Request 并发模型。因为 JAWS 是一个框架，可以系统地对这些策略进行定制，独立地或协作地进行评估，以决定最佳的策略方案。使用这些方案，JAWS 可以静态地和动态地改变自己的行为，以为给定的软件/硬件平台和工作负载采用最为有效的策略。JAWS 的自适配软件特性使其成为用于构造高性能 Web 服务器的强大应用框架。

2.1 介绍

在过去的几年中，万维网（Web）上的通信流量发生了戏剧性的增长。流量的增长在很大程度上应归于廉价的和无处不在的 Web 浏览器（比如 NCSA Mosaic、Netscape Navigator 和 Internet Explorer）的激增。同样地，Web 协议和浏览器也正日益被应用于专门而昂贵的计算任务，比如西门子[1]和柯达[2]所用的图像处理服务器和像 AltaVista 和 Lexis-Nexis 这样的数据库搜索引擎。

要跟上需求增长的步伐，必须开发高性能 Web 服务器。但是，在开发者配置和优化 Web 服务器时，他们面对的是一组异常丰富的设计策略。例如，开发者必须在广泛的并发模型（比如 Thread-per-Request vs. 线程池）、分派模型（比如同步 vs. 异步分派）、文件缓存模型（比如 LRU vs. LFU），以及协议处理模型（比如 HTTP/1.0 vs. HTTP/1.1）中进行选择。没有哪种配置对于所有硬件/软件平台和工作负载来说都是最佳的[1, 3]。

所有这些可选策略的存在保证了开发者可以定制 Web 服务器、以满足用户的需求。但是，在许多设计和优化策略间进行选择是麻烦而易错的。没有相应的指导，开发者将面临艰巨的任务：从头开始设计 Web 服务器来产生特定的解决方案。这样的系统常常难以维护、定制和调谐，因为许多设计工作都只是花在了使系统可运行上。

[2] 这里描述的工作得以完成，应归功于来自 Siemens AG、Eastman Kodak 和 NSF Research Grant NCR-9628218 的基金支持。

2.1.1 定义

我们将经常引用术语 *OO 类库*、*框架*、*模式*和*组件*。这些术语所指的是用于构建可复用软件系统的工具。OO 类库是一组软件对象实现，它们在用户调用对象方法时提供可复用功能。框架是一种可复用、“半完成”的应用，可被定制以产生自定义应用[4]。模式表示在特定的上下文中、软件开发问题的可反复使用的解决方案[5]。组件指的是一种“可具体化”的对象。OO 类库和框架都是通过实例化和专门化而得以具体化的成组对象。模式组件则通过编码来具体化。

2.1.2 综述

本论文阐释怎样使用 *OO 应用框架*和*设计模式*来产生灵活而高效的 Web 服务器。模式和框架可以协作应用，以改善 Web 服务器的效率和灵活性。模式以一种系统而易于理解的形式捕捉高性能和自适配 Web 服务器的抽象设计和软件体系结构。框架则使用特定的编程语言，比如 C++或 Java，来捕捉 Web 服务器的具体设计、算法和实现。相反，*OO 类库*提供构建应用所必需的原始材料，但没有对怎样将这些片段放在一起进行指导。

本论文聚焦于用于开发 JAWS[1, 3]高性能 Web 服务器的模式和框架。JAWS 既是一个 Web 服务器，又是一个框架，其他类型的服务器可通过它来进行构建。JAWS 框架自身是使用 ACE 框架[6, 7]来开发的。ACE 框架使通信软件领域中的一些关键模式[5]得以具体化。JAWS 和 ACE 中的框架和模式是有代表性的解决方案，已被成功应用于许多通信系统，范围从电信系统管理[8]到企业医学成像[2]和实时航空控制系统[9]等。

本论文被组织如下：2.2 给出对模式和框架的综述，并说明 JAWS 所提供的通信软件框架类型的动机；2.3 阐释怎样应用模式和组件来开发高性能 Web 服务器；2.4 比较 JAWS 与其他高性能 Web 服务器在高速 ATM 网络上的性能；2.5 给出结束语。

2.2 将模式和框架应用于 Web 服务器

为了给数目正在增长的 Internet 和 Intranet 用户提供服务和内容，对于高性能 Web 服务器的需求正在日益增长。Web 服务器的开发者正在努力构建快速、可伸缩和可配置的系统。但是，如果不注意避开一些常见的陷阱和缺陷，其中包括麻烦而易错的低级编程细节、缺乏可移植性，以及广泛的设计选择，这样的任务可能是十分困难的。这一部分给出了这些危险的路标。随后我们描述开发者怎样通过有效利用设计和代码复用，将模式和框架应用于避免这些危险。

2.2.1 Web 服务器软件的常见缺陷

Web 服务器开发者面临着一些反复发生的挑战，这些挑战在很大程度上独立于特定的应用需求。例如，像其他通信软件一样，Web 服务器必须执行多种任务：连接建立、事件处理器分派、进程间通信、内存管理和文件缓存、静态和动态的组件配置、并发、同步，以及持续性。在大多数 Web 服务器中，这些任务是以特定的方式、使用低级的本地 OS 应用编程接口（API）（比如用 C 编写的 Win32 或 POSIX）来实现的。

遗憾的是，本地 OS API 并不是开发 Web 服务器或其他类型的通信中间件和应用[10]的有效途径。下面是与本地 OS API 的使用相关联的常见缺陷：

过多的低级细节：通过本地 OS API 来构建 Web 服务器要求开发者熟悉低级的 OS 细节。开发者必须仔细地追踪每个系统调用返回的错误代码，并在他们的服务器中处理这些特定于 OS 的问题。这样的细节使得开发者的注意力从更广阔的、更为战略性的问题（比如语义和程序结构）上转移开来。例如，使用 wait 系统调用的 UNIX 开发者必须在下面两种错误之间进行区分：由于没有子进程存在而返回的错误和来自信号中断的错误。在后一种情况下，必须重新发出 wait 调用。

持续地重新发现和发明不兼容的更高级编程抽象：常用的对过多的 OS API 细节的补救方法是定义更高级的编程抽象。例如，许多 Web 服务器都创建文件缓存，以避免每次客户请求都要访问文件系统。但是，这些类型的抽象常常被各个开发者或项目独立地重新发现和发明。这样的特定处理妨碍了生产效率，并创建出不兼容的组件，无法迅速地在大型软件组织的项目内和项目间复用。

高错误可能性：由于低级 OS API 缺乏类型安全性，对它们进行编程是麻烦而易错的。例如，大多数 Web 服务器都使用 Socket API[11]来编写。但是，Socket API 中的通信端点被表示为无类型的句柄。这增加了发生微妙的编程错误和运行时错误的可能性。

缺乏可移植性：低级 OS API 出了名地不可移植，即使是在同一 OS 的不同版本间也是如此。例如，Win32 平台上的 Socket API 实现（WinSock）与 UNIX 平台上的实现有着微妙的不同。而且，即使是 Windows NT 的不同版本上的 WinSock 实现也具有不兼容的、与时俱变的错误：在执行非阻塞连接时会导致偶发的失败。

陡峭的学习曲线：由于有过多的细节，掌握 OS 级 API 所需的努力可能是很高的。例如，学习怎样正确地使用 POSIX 异步 I/O[12]来编程十分困难。学习怎样使用异步 I/O 机制来编写可移植的应用甚至会更困难，因为它们在各 OS 平台间有着极大的不同。

不能处理更高的复杂性：OS API 为一些机制定义了基本接口，像进程和线程管理、进程间通信、文件系统，以及内存管理。但是，当应用的大小和复杂性增长时，这些基本接口无法适当地升级。例如，典型的 UNIX 进程只允许缓冲大约 7 个待处理连接[13]。对于被大量访问的、必须处理成百并发客户的 Web 服务器来说，这个数目是不够的。

2.2.2 通过模式和框架克服 Web 服务器的缺陷

软件复用是被广泛称许的减少开发工作量的方法。复用有效利用了有经验的开发者的领域知识和以前的成果。在有效地应用时，复用可以避免重新创建和认证常用的、针对重复发生的应用需求和软件设

计挑战的解决方案。

Java 的 java.lang.net 和 RougeWave Net.h++是两个常见的将可复用 OO 类库应用于通信软件的例子。尽管类库有效地支持小规模的组件复用，它们的范围是严重受限的。特别地，类库不会对相关软件组件族之间的规范控制流和协作进行捕捉。因而，应用基于类库的复用的开发者常常要为每个新应用重新发明和实现整个的软件体系结构。

更为强大的克服上面描述的缺陷的途径是对在成功的 Web 服务器之下的*模式*进行标识，并在*面向对象应用框架*中使这些模式具体化。通过捕捉常见软件开发问题的解决方案，模式和框架有助于减少对关键的 Web 服务器概念和组件的重新发现和发明[5]。

将模式应用于 Web 服务器的好处：模式提供了常见的 Web 服务器微体系中的结构和参与者的文档。例如，反应器（Reactor）[14]和主动对象（Active Object）[15]模式分别被广泛用作 Web 服务器的分派和并发策略。这些模式是已被证明有益于构建灵活而高效的 Web 服务器的对象结构的一般化。

传统上，这些模式类型或者被锁在老练的开发者的头脑里，或者被深埋在源码中。但是，让这样有价值的信息只是放在这些地方是危险而昂贵的。例如，如果不编写文档，有经验的 Web 服务器设计者的洞见可能会随时间而消逝。同样地，可能需要相当的努力才能从现有源码中反向地设计出模式来。因此，为了给负责增强和维护现有软件的开发者保留设计信息，明确地捕捉 Web 服务器模式并编写文档是必要的。而且，特定领域的知识还有助于指导在其他领域中构建新服务器的开发者的设计决策。

将框架应用于 Web 服务器的好处：模式知识有助于减少开发工作和维护代价。但是，只是复用模式并不足以创建灵活而高效的 Web 服务器软件。在模式使抽象设计和体系结构知识复用成为可能的同时，被编写为模式的抽象并不会直接产生可复用的代码[16]。因此，有必要增加对模式的研究，考查它们与框架的创建和使用的关系。通过实现常用设计模式、并分解出常见实现角色，框架可帮助开发者避免对标准的 Web 服务器组件进行昂贵的重新发明。

2.2.3 框架、模式，以及其他复用技术之间的关系

通过集成成组的抽象类，并定义这些类的实例进行协作的标准方式，框架为应用提供了可复用的软件组件[4]。一般而言，组件并不是自包含的，因为它们常常依赖于框架中其他组件所提供的功能。但是，这些组件聚合在一起构成了特定的实现，也就是，应用骨架。可以通过继承和实例化框架中的可复用组件来对骨架进行定制。

Web 服务器中复用的范围可以显著地大于使用传统的函数库或组件的 OO 类库。特别地，2.3 中描述的 JAWS 框架特别为广泛的 Web 服务器任务作了裁剪。这些任务包括服务初始化、错误处理、流控制、事件处理、文件缓存、并发控制和原型流水线操作。重要的是要记住这些任务对于其他许多类型的通信软件来说也是可复用的。

总而言之，框架和组件以下面几种方式增强了基于组件类库的复用技术：

框架定义“半完成”应用，其中包含了特定领域的对象结构和功能：类库提供了一种粒度相对较小的复用。例如，图 2-1 中的类和字符串、复数、数据及位组一样，是典型的低级、相对独立和通用的组件。

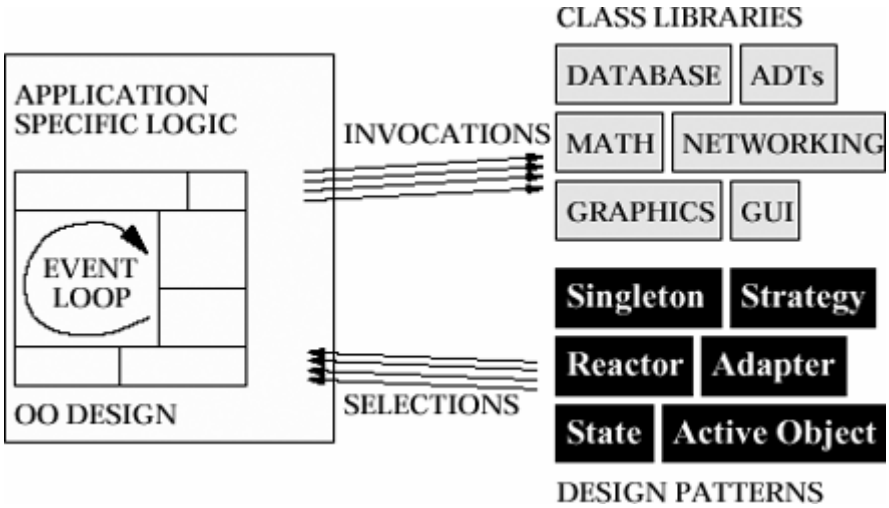

图 2-1 类库组件体系结构

相反，框架中的组件相互协作，来为相关应用族提供可定制的体系结构骨架。完整的应用可以通过从框架组件继承、以及/或者实例化框架组件来合成。如图 2-2 所示，框架减少了应用特有代码的数量，因为特定领域的许多处理被分解进通用的框架组件中。

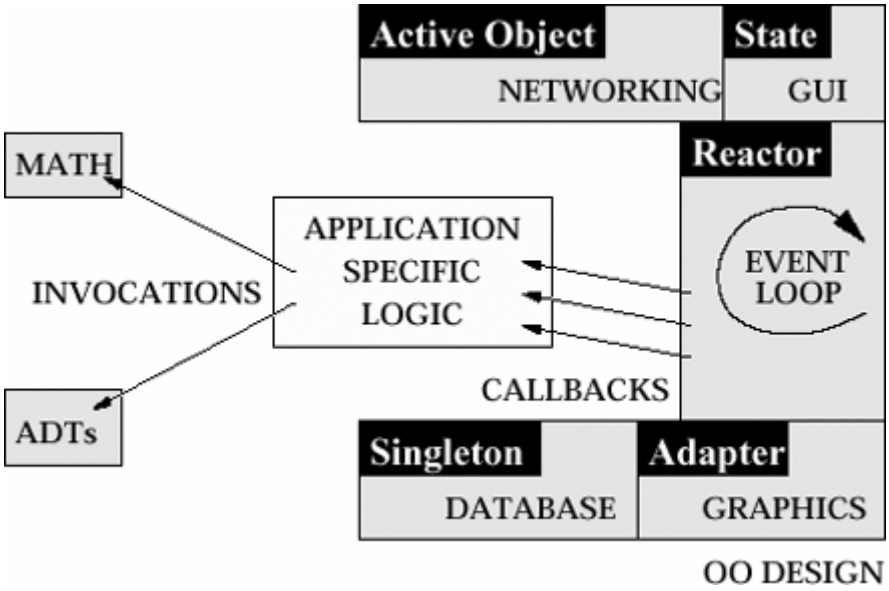

图 2-2 应用框架组件体系结构

框架是主动的，并在运行时显示出“ 控制的反转 ”：类库组件通常*被动地*工作。特别地，类库组件常常从“ 自指引 ”（self-directed）的应用对象那里借用线程控制来完成它们的处理。因为应用对象是自指引的，在很大程度上应用开发者要负责决定怎样组合组件和类，以形成完整的系统。例如，通常要为每个新应用重写管理事件循环、并在可复用和应用特有组件间确定控制流的代码。

通过类库和组件构建的应用的典型结构和动力特性在图 2-1 中演示。该图还演示了设计模式怎样帮助指导类库组件的设计、实现和复用。注意，在提供工具解决特定任务（例如建立网络连接）的同时，类库的存在并没有提供对系统设计的明确指导。特别地，软件开发者要独自负责在他们的应用设计中确定并应用模式。

相对于类库，框架中的组件更为*主动*。特别地，它们通过像反应器（Reactor）[14]和观察者（Observer）[5]这样的事件分派分派模式来管理应用中的规范控制流。框架的回调驱动的运行时体系结构如图 2-2 所示。

图 2-2 演示了框架的一种关键特性：它在运行时的“ 控制的反转 ”。这种设计使得规范的应用处理步骤可由通过框架的反应式分派机制[14]调用的事件处理器对象来定制。在事件发生时，框架的分派器通过调用预登记处理器对象的挂钩方法来进行反应，由该方法完成事件的应用特有的处理。

控制的反转允许框架，而不是每个应用，确定调用哪一组应用特有方法来响应外部事件（比如 HTTP 连接和数据到达 Socket）。作为结果，框架使一组集成的模式具体化、并预先应用进协作的组件中。这样的设计减轻了软件开发者的负担。

在实践中，框架、类库和组件是互相补充的技术。框架常常在内部利用类库和组件来简化框架的开发。例如，JAWS 的一些部分使用由 C++标准模板库（STL）[17]提供的字符串和向量容器来管理连接映射和其他查找结构。此外，由框架事件处理器调用的应用特有的回调常常使用类库组件来完成基本的任务，比如字符串处理、文件管理和数字分析。

为演示怎样成功地应用 OO 模式和框架来开发灵活而高效的通信软件，本论文的余下部分检查 JAWS 框架的结构、使用和性能。

2.3 JAWS 自适配 Web 服务器

将框架和模式应用于通信软件的好处最好通过例子来演示。这一部分描述 JAWS 的结构和功能。JAWS 是一种高性能和自适配的、实现了 HTTP 协议的 Web 服务器。它还是一个平台无关的应用框架，其他类型的通信服务器可以通过它来构建。

2.3.1 JAWS 框架综述

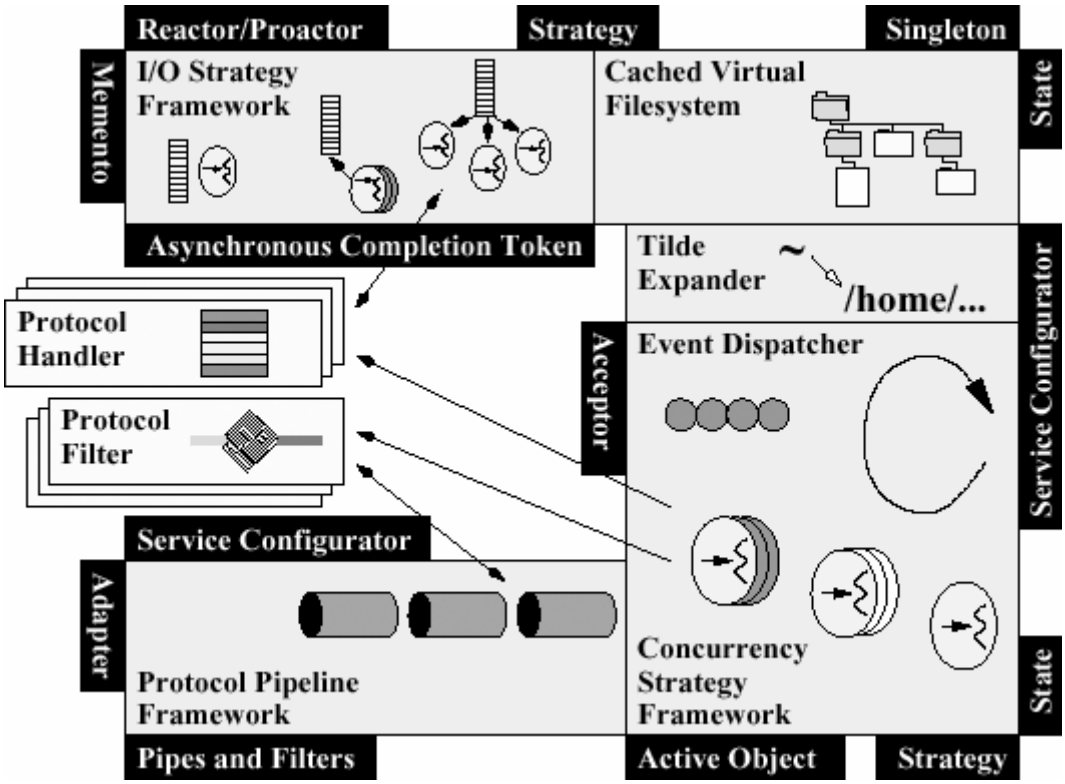

图 2-3 JAWS 框架的体系概览

图 2-3 演示组成 JAWS 自适配 Web 服务器框架的主要结构组件和设计模式。JAWS 的设计允许定制多种 Web 服务器策略，以响应环境因素。这些因素包括*静态*因素，比如对 OS 中的内核级线程及/或异步 I/O 的支持、可用 CPU 的数目；以及*动态*因素，比如 Web 流量模式和工作负载特性。

JAWS 被构造为*框架的框架* (framework of frameworks)。整个 JAWS 框架含有以下组件和框架：*事件分派器*(Event Dispatcher)、*并发策略*(Concurrency Strategy)、*I/O 策略*(I/O Strategy)、*协议流水线*(Protocol Pipeline)、*协议处理器* (Protocol Handler)，以及*缓存式虚拟文件系统*(Cached Virtual Filesystem)。各个框架都被构造为一组使用 ACE[18]中的组件实现的协作对象。JAWS 组件和框架之间的协作由一个模式族进行指导，该模式族在图 2-3 中沿着图的边缘列出。对 JAWS 中的关键框架、组件和模式的概述在下面给出。更为详细的对这些模式怎样应用于 JAWS 的设计的描述将在 2.3.2 中给出。

事件分派器（Event Dispatcher）：该组件负责协调 JAWS 的*并发策略*和它的 *I/O 策略*。Web 客户的被动连接建立遵循*接受器*（Acceptor）模式 [19]。新到来的请求由一种并发策略服务。在事件被处理时，它们被分派到*协议处理器*（Protocol Handler），后者由一种 I/O 策略参数化。从一系列可选方案中选择，以动态绑定到特定并发策略和 I/O 策略的机制遵循*策略*（Strategy）模式 [5]。

并发策略（Concurrency Strategy）：该框架实现的并发机制（比如单线程、Thread-per-Request，或线程池）可被适配性地选择：在运行时使用*状态*（State）模式，或在初始化时预先确定。*服务配置器*（Service Configurator）模式[20]用于在运行时将特定的并发策略配置进 Web 服务器。当并发涉及多线程时，策略会创建遵循*主动对象*（Active Object）模式[15]的协议处理器。

I/O 策略（I/O Strategy）：该框架实现多种 I/O 机制，比如异步、同步和反应式 I/O。多种 I/O 机制可以同时使用。异步 I/O 通过*前摄器*（Proactor）[21]和*异步完成令牌*（Asynchronous Completion Token）[22]模

式来实现。反应式 I/O 通过*反应器*（Reactor）模式[14]来完成。反　　应式 I/O 利用 *Memento* 模式[5]来捕捉请求状态，并使其外在化，以在后面将其恢复。

协议处理器（Protocol Handler）：该框架允许系统开发者将 JAWS 框架应用于 Web 系统应用的变种。*协议处理器*由并发策略和 I/O 策略来参数化。这些策略对协议处理器来说是不透明的（通过使用*适配器*（Adapter）[5]模式）。在 JAWS 中，该组件实现了 HTTP/1.0 请求方法的解析与处理。该抽象使得其他协议（比如 HTTP/1.1 和 DICOM）能够很容易地结合进 JAWS。要增加新协议，开发者只需简单地编写新的协议处理器实现，随后将其配置进 JAWS 框架中。

协议流水线（Protocol Pipeline）：该框架使过滤器操作能够很容易地与正在被*协议处理器*处理的数据进行合成。这种集成是通过采用适配器模式来完成的。流水线遵循用于输入处理的*管道和过滤器*（Pipes and Filters）模式[23]。使用*服务配置器*模式，可在运行时动态链接流水线组件。

缓存式虚拟文件系统（Cached Virtual Filesystem）：该组件通过减少文件系统访问开销来改善 Web 服务器性能。可以遵循策略模式[5]来选择多种缓存策略，比如 LRU、LFU、提示式策略和结构化策略。这使得开发者可以根据有效性来对不同的缓存策略进行裁剪，并静态或动态地配置最佳策略。各个 Web 服务器的缓存通过使用*单体*（Singleton）模式[5]来实例化。

Tilde Expander：该组件是另一种缓存组件，它使用理想哈希表[24]来将简写的用户登录名（例如，~schmidt）映射到用户主目录（例如，/home/cs/faculty/schmidt）。当个人 Web 页面存储在用户主目录中、而用户目录又没有驻留在共同的根上时，该组件能够充分地减少访问系统用户信息文件（比如/etc/passwd）所需的磁盘 I/O 开销。通过服务配置器模式的效力，可以动态地解除 Tilde Expander 的链接，并将其重新链接进服务器（例如，在新用户加入系统时）。

2.3.2 JAWS 中的设计模式综述

图 2-3 中的 JAWS 体系结构图演示了 JAWS 是怎样构造的，但并没有说明它为什么以这种特定的方式构造。要理解 JAWS 为什么包含有像*并发策略*、*I/O 策略*、*协议处理器*和*事件分派器*这样的框架和组件，需要对在通信软件领域（一般而言）和 Web 服务器（特定的）之下的设计模式有更深入的了解。图 2-4 演示与 JAWS 有关的*战略*和*战术*模式。这些模式在下面进行总结。

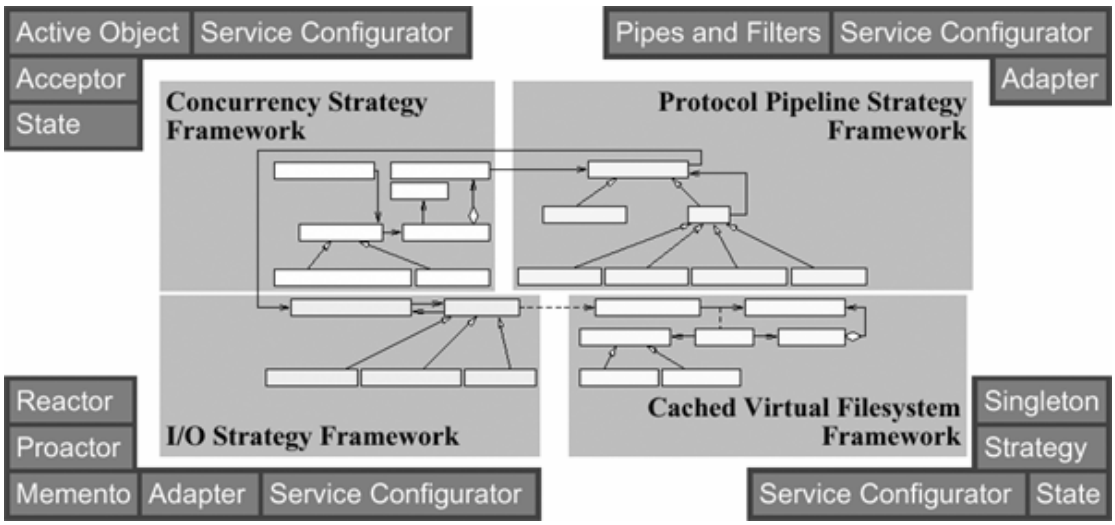

图 2-4 JAWS 框架中使用的设计模式

2.3.2.1 战略模式

下面的模式对于 Web 服务器的整个软件体系结构来说是*战略性*的。它们的使用广泛地影响了系统中大量组件的交互水平。这些模式还被广泛用于指导许多其他类型的通信软件的体系结构。

接受器模式（Acceptor Pattern）：该模式使被动的连接建立与连接一旦建立后所执行的服务去耦合[19]。JAWS 使用接受器模式来独立于它的连接管理策略适配性地改变它的并发和 I/O 策略。图 2-5 演示在 JAWS 的上下文中的接受器模式的结构。*接受器*是一种工厂[5]，无论何时*事件分派器*通知它有连接已从客户到达，它都会创建、接受并启用一个新的协议处理器。

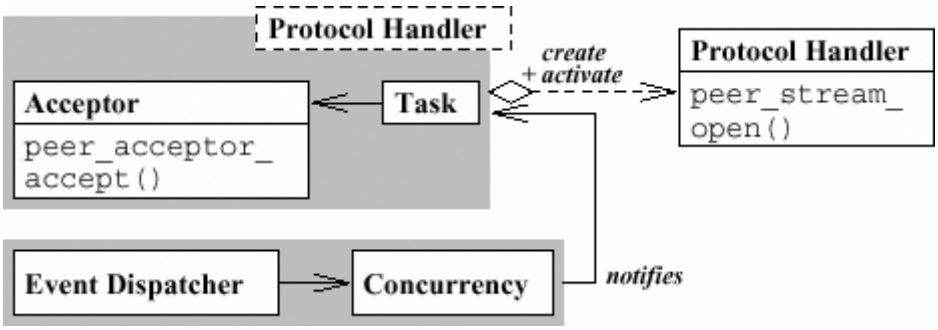

图 2-5 JAWS 中接受器模式的结构

反应器模式（Reactor Pattern）：该模式使服务器应用的同步事件多路分离及事件处理器通知分派逻辑与为响应事件而执行的服务去耦合[14]。JAWS 使用反应器模式来处理来自多个事件源的多个同步事件，而又*无需*轮询所有事件源，或是无限期地阻塞在任何事件源上。图 2-6 演示在 JAWS 上下文中的反应器模式的结构。

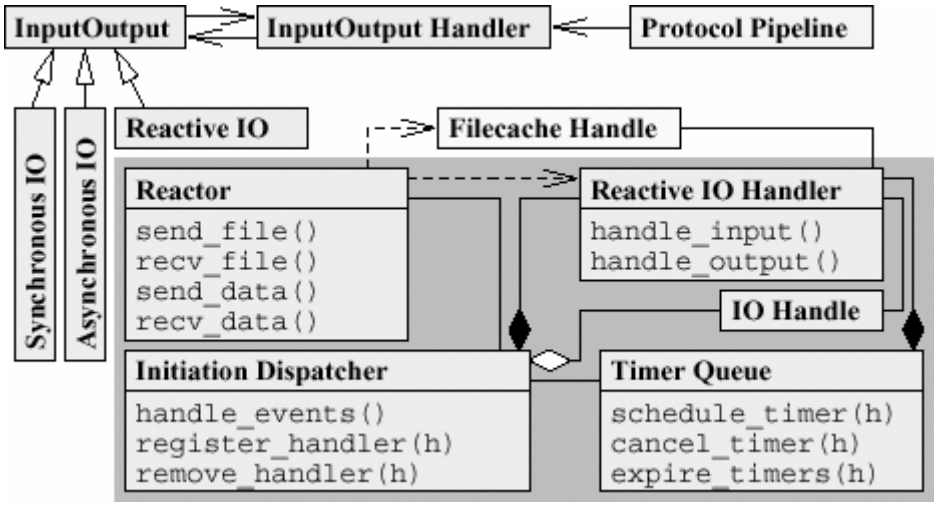

图 2-6 JAWS 中反应器模式的结构

JAWS *Reactive IO Handler*（反应式 I/O 处理器）对象将自身登记到 *Initiation Dispatcher*（发起分派器），以与一些事件（也就是，在由 HTTP 请求建立的连接上的输入和输出）相关联。当与这些 *Reactive IO Handler* 对象相关联的事件发生时，*Initiation Dispatcher* 调用它们的 handle_input 通知挂钩方法。2.3.3.2 介绍的单线程 Web 服务器并发模型使用了反应器模式。

前摄器模式（Proactor Pattern）：该模式使服务器应用的异步事件多路分离及事件处理器完成分派逻辑与为响应事件而执行的服务去耦合[21]。JAWS 使用前摄器模式来在异步地处理其他 I/O 事件的同时执行服务器特有的处理，比如解析请求头。图 2-7 演示在 JAWS 上下文中的前摄器模式的结构。JAWS *Proactive IO Handler*（前摄式 I/O 处理器）对象将自身登记到 *Completion Dispatcher*（完成分派器），以与一些事件（也就是，在由 HTTP 请求建立的连接上的文件接收和递送）相关联。

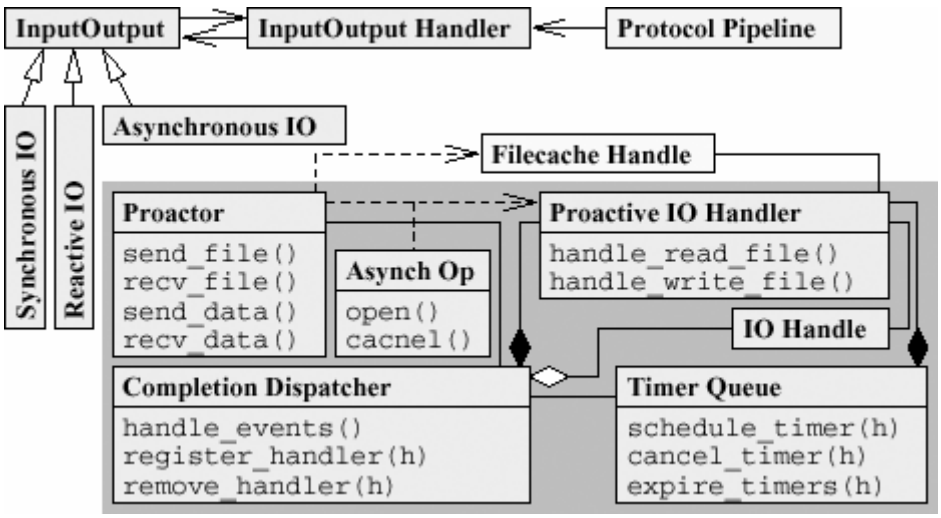

图 2-7 JAWS 中前摄器模式的结构

反应器和前摄器模式之间的主要区别是 *Proactive IO Handler* 定义*完成*挂钩，而 *Reactive IO Handler 处理器*定义*发起*挂钩。因此，当像 recv_file 或 send_file 这样的异步调用的操作完成时，*Completion*

Dispatcher 会调用这些 *Proactive IO Handler* 对象的适当的完成挂钩方法。2.3.3.2 中的线程池的异步变种使用了前摄器模式。

主动对象模式（Active Object Pattern）：该模式使方法调用与方法执行去耦合，允许方法并发地运行[15]。JAWS 使用主动对象模式来在分离的线程控制中并发地执行客户请求。图 2-8 演示在 JAWS 上下文中的主动对象模式的结构。

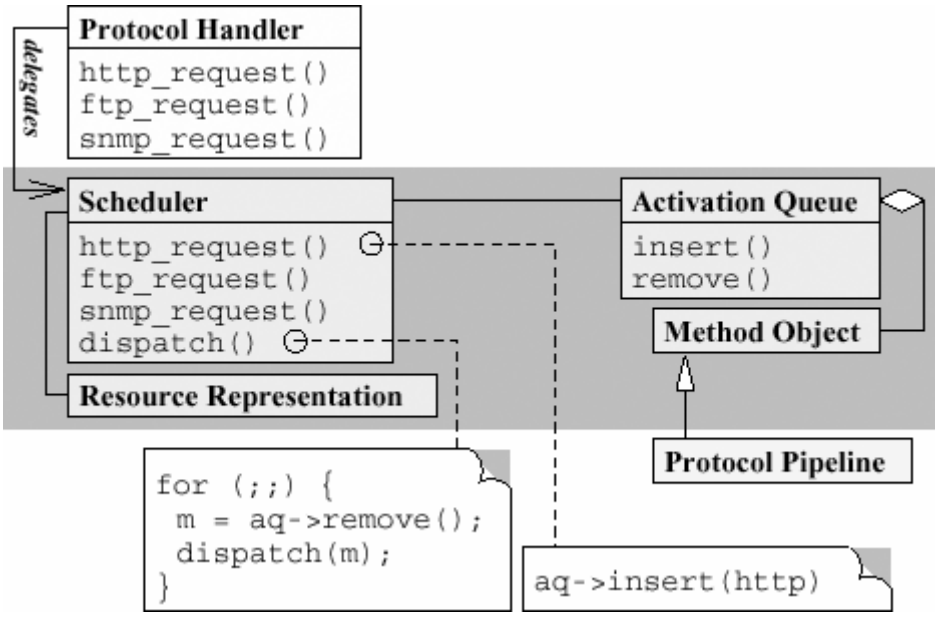

图 2-8 JAWS 中主动对象模式的结构

Protocol Handler（协议处理器）发出请求给 *Scheduler*（调度器），后者将请求方法（比如 HTTP 请求）转换为存储在 *Activation Queue*（启用队列）中的 *Method Object*（方法对象）。运行在与客户分离的线程中的 *Scheduler* 使这些 *Method Object* 出队，并将它们转换回方法调用，以执行指定的协议。在 2.3.3.2 描述的 Thread-per-Request、线程池，以及 Thread-per-Session 并发模型中使用了主动对象模式。

服务配置器模式（Service Configurator Pattern）：该模式使系统中个体组件的实现与它们被配置进系统的*时间*去耦合。JAWS 使用服务配置器模式来在安装时或运行时动态地优化、控制及重配置 Web 服务器策略的行为[25]。图 2-9 演示在*协议流水线过滤器*（Protocol Pipeline Filter）和*缓存策略*（Caching Strategy）的上下文中的服务配置器模式的结构。

该图描述服务配置器模式怎样动态地管理动态链接库（DLL）。这使得框架能够在运行时动态地配置服务器策略的不同实现。*Filter Repository*（过滤器仓库）和 *Cache Strategy Repository*（缓存策略仓库）从 *Service Repository*（服务仓库）继承功能。同样地，策略实现（比如 *Parse Request*（解析请求）和 *LRU Strategy*（LRU 策略））从模式的 *Service*（服务）组件那里借用接口，以使仓库能动态地对它们进行管理。

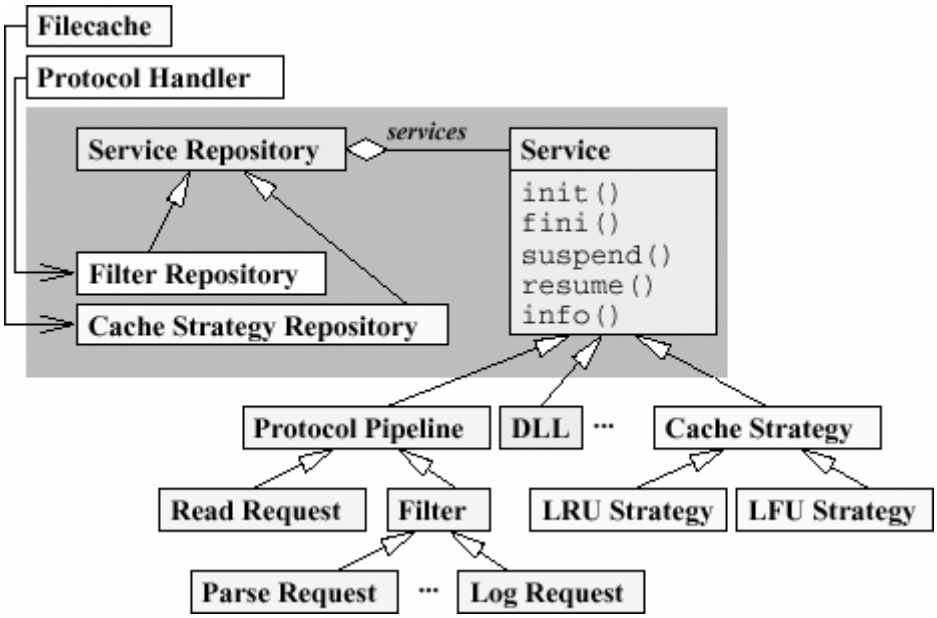

图 2-9 JAWS 中服务配置器模式的结构

2.3.2.2 战术模式

Web 服务器还利用了许多*战术*模式，比起上面描述的战略模式，它们要更为普遍和与领域无关。下列战术模式被用于 JAWS 中：

策略模式（Strategy Pattern）：该模式定义一个算法族，对其中的每个算法进行封装，并使它们成为可互换的[5]。JAWS 大量地使用此模式来有选择地配置不同的缓存替换策略，而又不影响 Web 服务器的核心软件体系结构。

适配器模式（Adapter Pattern）：该模式将不兼容的接口转换为可由客户使用的接口[5]。JAWS 在它的 I/O 策略框架中使用此模式，以统一封装同步、异步和反应式 I/O 操作。

状态模式（State Pattern）：该模式定义一种合成对象，其行为取决于其状态[5]。JAWS 中的 Event Dispatcher 使用状态模式来无缝地支持不同的并发策略，以及同步和异步 I/O。

单体模式（Singleton Pattern）：该模式确保一个类只有一个实例，并提供一个对它进行访问的全局访问点[5]。JAWS 使用单体来确保它的缓存式虚拟文件系统只有一份拷贝存在于 Web 服务器进程中。

相对于早先描述的战略模式，战术模式对软件设计有着相对局部的影响。例如，单体是一种战术模式，常常用于统一 Web 服务器中特定的可全局访问的资源。尽管此模式是领域无关的、因而也是可广泛应用的，它所解决的问题并不像战略模式（比如主动对象和反应器）那样普遍而深入地影响 Web 服务器软件体系结构。但是，要实现高度灵活、能良好地响应应用需求和平台特性的变化的软件，必须全面理解战术模式。

这一部分的余下部分讨论 JAWS 的用于并发、I/O、协议流水线处理和文件缓存的框架的结构。对于每一个框架，我们描述关键的设计挑战、并概述可选方案策略的范围。随后，我们解释各个 JAWS 框架是怎样被结构、以支持可选策略方案的配置的。

2.3.3 并发策略

2.3.3.1 设计挑战

并发策略会显著地影响 Web 系统的设计和性能。对现有 Web 服务器（包括 Roxen、Apache、PHTTPD、Zeus、Netscape 和 Java Web 服务器）的实验研究[3]表明大部分与 I/O 无关的 Web 服务器开销来自 Web 服务器的并发策略。关键的开销包括同步、线程/进程创建，以及上下文切换。因此，选择高效的并发策略对于获取高性能来说是至关紧要的。

2.3.3.2 可选策略方案

选择正确的并发策略并非无关紧要的事情。影响决策的有动态和静态两种因素。*静态*因素可被预先确定。这些因素包括硬件配置（例如，处理器数目、内存数量，以及网络连接速度）、OS 平台（例如，线程和异步 I/O 的可用性），以及 Web 服务器使用情况（例如，数据库连接、图像服务器，或是 HTML 服务器）。*动态*因素是那些在系统执行过程中发生的可检测和可度量的情况。这些因素包括机器负载、并发请求数、动态内存的使用，以及服务器工作负载。

现有的 Web 服务器使用了广泛的并发策略来回应有关的众多因素。这些策略包括单线程并发（例如，Roxen）、基于进程的并发（例如，Apache 和 Zeus），以及多线程并发（例如，Apache 和 JAWS）。每种策略都会产生正面和负面的效果，必须在静态和动态因素的上下文中才能加以分析和评估。这些权衡在下面总结。

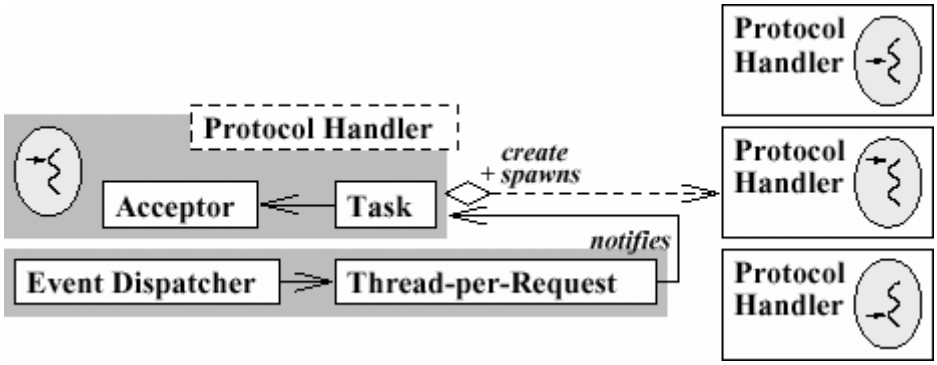

图 2-10 JAWS 中的 Thread-per-Request 策略

Thread-per-Request：该模式在单独的线程控制中处理每个来自客户的请求。因而，当每个请求到达时，就会创建一个新线程来处理该请求。这种设计允许每个线程使用同步 I/O 机制来读写所请求的文件。图 2-10 在 JAWS 框架的上下文中演示此模式。在这里，*接受器*反复地等待连接，创建协议处理器，并派生新线程，以使处理器能够继续处理连接。

Thread-per-Request 的优点是它的简单性和它利用多处理器平台上的并行性的能力。它的主要缺点是缺乏可伸缩性、也就是，正在运行的线程的数目有可能无节制地增长，耗尽可用内存和 CPU 资源。因此，Thread-per-Request 对于轻负载、低延迟的服务器来说是足够的。但是，它可能不适用于那些被频繁访问、执行费时任务的服务器。

Thread-per-Session：*会话*（Session）是客户向服务器做出的一系列请求。在 Thread-per-Session 中，所有这些请求都通过在每个客户与 Web 服务器进程中的单独线程之间的一个连接来提交。因此，该模型在多次请求间分摊了线程创建和连接建立开销。

Thread-per-Session 的资源耗费比 Thread-Per-Request 要少，因为它并不为每个请求派生一个单独的线程。但是，在客户的数量增长时，它还是易于无节制地消耗资源。还有，Thread-per-Session 的使用要求客户和服务器都支持在多个请求间复用已建立连接的概念。例如，如果 Web 客户和 Web 服务器都遵循 HTTP/1.1，就可以在它们之间使用 Thread-per-Session。但是，如果客户或服务器只支持 HTTP/1.0，Thread-per-Session 就会退化为 Thread-per-Request[26, 27]。

线程池（Thread Pool）：在此模型中，在 Web 服务器初始化过程中会预先派生一组线程。每个线程从作业队列中获取一项任务。在线程处理作业的同时，它从线程池中被移除。一旦任务完成，线程就返回池中。如图 2-11 所示，正被获取的作业是*接受器*的完成。当它完成时，线程创建协议处理器，并出借它的线程控制，以使处理器能够处理连接。

线程池比 Thread-per-Request 的开销要少，因为线程创建的代价通过预先派生而被分摊掉了。而且，线程池所能消耗的资源的数量是有限的，因为池的大小是固定的。但是，如果池太小，它可能会被耗尽。这将导致新到来的请求被丢弃或无限期地等待。更进一步，如果池太大，资源耗费可能并不比使用 Thread-per-Request 更好。

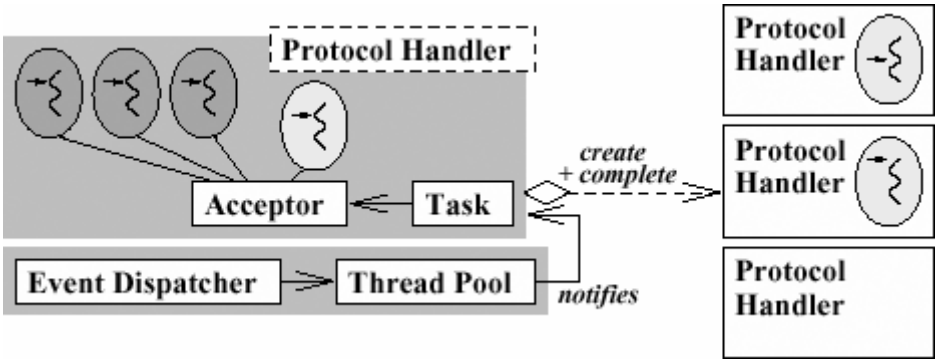

图 2-11 JAWS 中的线程池策略

单线程（Single-Threaded）：在此模型中，所有连接和请求都由同一线程控制来处理。单线程服务器的简单实现依次对请求进行处理。通常这对于高流量的产品服务器来说是不够的，因为后续请求会被阻塞、直到轮到它们进行处理，从而产生不可接受的延迟。

更为成熟的单线程实现使用异步或反应式 I/O（在 2.3.4 描述）来并发地处理多个请求。在支持异步

I/O 的单处理器机器上，单线程并发策略可以比多线程方案执行得更好[1]。因为 JAWS 的 I/O 框架与它的并发框架是不相关的，我们认为单线程并发策略是线程池的池大小为 1 时的一种特例。

[1]和[3]中的实验演示了并发和时间分派策略的选择对负载条件遇到变化的 Web 服务器性能的影响。特别地，没有哪种服务器策略能够为所有情况都提供最佳性能。因而，服务器框架至少应该提供两种程度的自由：

1. *静态适配性*：框架应该允许 Web 服务器开发者选择能最好地满足系统的静态需求的并发策略。例如，多处理器机器可能比单处理器机器更适合多线程并发。
2. *动态适配性*：框架应该允许它的并发策略动态地适配当前的服务器环境，以在服务器负载发生动态变化的情况下取得最佳性能。例如，为了应付意外的负载使用，有可能必须增加线程池中可用线程的数目。

2.3.3.3 JAWS 并发策略框架

如上面所讨论的，没有哪种并发策略在所有情况下都能最佳地执行。但是，也不是所有平台都能够有效地使用所有可用的并发策略。为解决这些问题，JAWS 并发策略框架同时支持相关于它的并发和事件分派策略的静态和动态的适配性。

图 2-12 演示 JAWS 的并发策略框架的 OO 设计。*Event Dispatcher*（事件分派器）和 *Concurrency*（并发）对象依据 *State*（状态）模式来交互。如图中所演示的，server 可以在对 server->dispatch()的连续调用间改变为使用 Thread-per-Connection 或线程池，从而使不同的并发策略产生效果。Thread-per-Connection 策略是对上面讨论的 Thread-per-Request 和 Thread-per-Session 策略的抽象。每种并发机制都使用了 *Task*（任务）。取决于并发的选择，任务可以表示单个*主动对象*，或是一组*主动对象*。*并发*对象的行为遵循*接受器*（Acceptor）模式。这样的体系结构使得服务器开发者能够集成各种可选的并发策略。通过策略配置文件的帮助，服务器可以在运行时动态地选择不同策略，以获得最佳的性能。

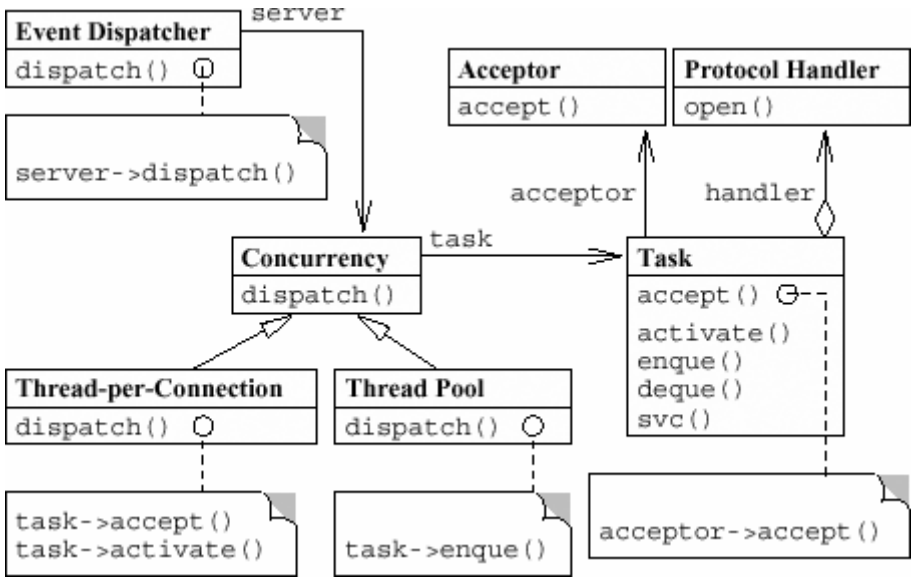

图 2-12 并发策略框架的结构

2.3.4 I/O 策略

2.3.4.1 设计挑战

对 Web 服务器开发者的另一项关键挑战是设计高效的数据获取和递送策略，合起来称为 I/O。围绕高效 I/O 的问题可以是极具挑战性的。系统开发者常常必须安排多个 I/O 操作、以利用硬件/软件平台上可用的并发性。例如，高性能 Web 服务器在并发地解析新获取的来自其他客户的请求时，应该能同时在网络上传输多个文件。

特定类型的 I/O 操作有着与其他类型的 I/O 操作不同的需求。例如，涉及货币基金转账的 Web 事务可能需要*同步地*运行，也就是，用户在事务结束后才能继续其他操作。相反，访问静态信息的 Web，比如基于 CGI 的搜索引擎查询，可以异步地运行，因为它们可以在任何时候被取消。这些不同的需求把我们引向了不同的执行 I/O 的策略。

2.3.4.2 可选策略方案

如上面所指出的，有多种因素影响对 I/O 策略的选择。Web 服务器的设计可使用若干不同的 I/O 策略，比如*同步*、*反应式*和*异步的* I/O。使用这些策略的相关好处在下面讨论。

同步 I/O 策略：同步 I/O 描述在 Web 服务器进程和内核之间的 I/O 交互的模型。在此模型中，内核不到所请求的 I/O 操作完成、部分完成或失败，就不会将线程控制返回给服务器。[1]显示在高速 ATM 网络上的 Windows NT 中，用于小文件传输的同步 I/O 通常执行良好。

同步 I/O 广为 UNIX 服务器程序员所知，并且最容易使用（有争论的）。但是，该模型也有一些缺点。首先，它与单线程并发策略结合在一起，不可能同时执行多个同步 I/O 操作。其次，当使用多个线程（或进程）时，I/O 请求还是有可能无限期地阻塞。因而，有限的资源（比如 Socket 句柄或文件描述符）可能会耗尽，使得服务器不再有响应。

反应式 I/O 策略：早期版本的 UNIX 只提供同步 I/O。系统 V UNIX 引入了非阻塞式 I/O，以避免阻塞问题。但是，非阻塞式 I/O 要求 Web 服务器轮询内核、以发现是否有任何输入可用[11]。反应式 I/O 减轻了同步 I/O 的阻塞问题，而又不诉诸轮询方法。在此模型中，Web 服务器使用 OS 事件多路分离系统调用（例如，UNIX 中的 select，或 Win32 中的 WaitForMultipleObjects）来确定哪一个 Socket 可以执行 I/O。当调用返回时，服务器可在返回的句柄上执行 I/O，也就是，服务器对发生在分开的句柄上的多个事件进行*反应*。

反应式 I/O 被事件驱动应用（比如 X windows）广泛使用，并已被编写为*反应器*（Reactor）设计模式[14]。但是除非小心地封装反应式 I/O，由于管理多个 I/O 句柄的复杂性，这种技术很容易出错。而且，

反应式 I/O 可能无法有效地利用多 CPU。

异步 I/O 策略：异步 I/O 简化了一或多个线程控制中多个事件的多路分离，而又不会阻塞 Web 服务器。当 Web 服务器发起 I/O 操作时，内核在服务器处理其他请求的同时、异步地执行操作直到完成。例如，Windows NT 中的 TransmitFile 操作可以异步地将整个文件从服务器传输到客户去。

异步 I/O 的优点是 Web 服务器不需要在 I/O 请求上阻塞，因为它们是异步完成的。这使得服务器能够高效地为高 I/O 延迟的操作（比如大文件传输）进行伸缩。异步 I/O 的缺点是它在许多 OS 平台（特别是 UNIX）上不可用。此外，编写异步程序比编写同步程序可能要更为复杂[21, 22, 28]。

2.3.4.3 JAWS I/O 策略框架

[1]中的实验研究将不同的服务器策略系统地归属到多种负载条件。结果揭示出各种 I/O 策略在不同的负载条件下的行为也不同。而且，没有哪种 I/O 策略能够在所有负载条件下最优地执行。通过使 I/O 策略动态地适应运行时服务器环境，JAWS I/O 策略框架解决了这一问题。而且，如果新的 OS 提供了一种定制的 I/O 机制（比如，异步分散/集中式 I/O），有可能提供更好的性能，可以很容易地改编 JAWS I/O 策略框架来使用它。

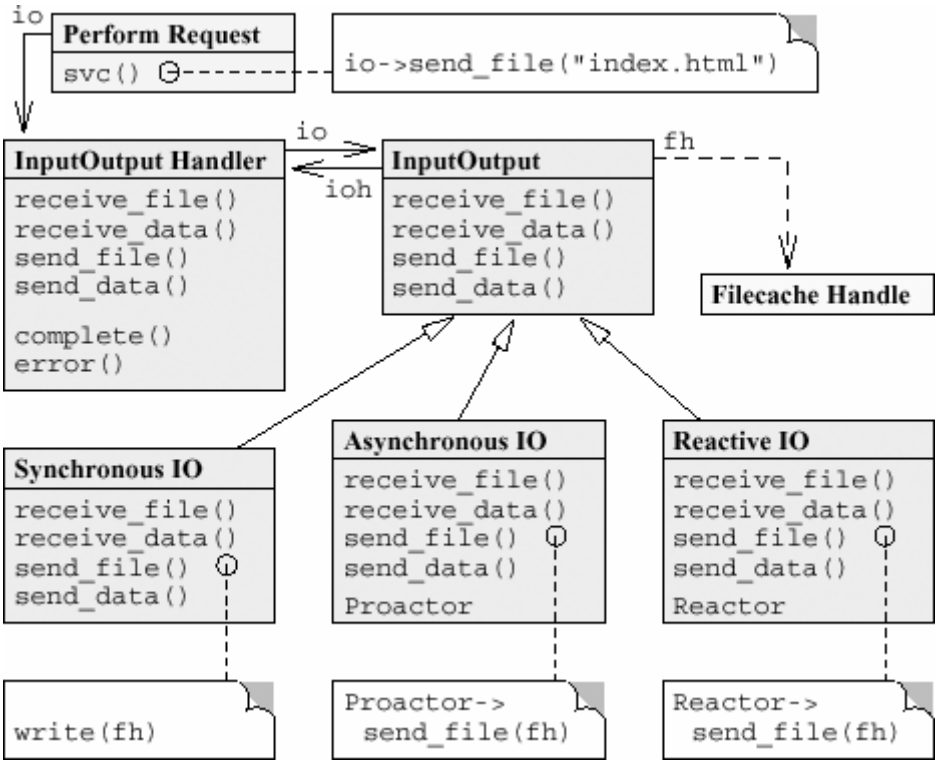

图 2-13 I/O 策略框架的结构

图 2-13 演示 JAWS 所提供的 I/O 策略框架的结构。*Perform Request*（执行请求）是一种 Filter（过滤器），派生自在 2.3.5 中阐释的 Protocol Pipeline（协议流水线）。在此例中，*Perform Request* 发出 I/O 请求

给它的 *InputOutput Handler*（输入输出处理器）。*InputOutput Handler* 将对它发出的 I/O 请求委托给 *InputOutput* 对象。

JAWS 框架提供派生自 InputOutput 的 *Synchronous*、*Asynchronous* 和 *Reactive IO* 组件。各种 I/O 策略使用适当的机制来发出请求。例如，*Synchronous IO* 组件使用传统的阻塞式 read 和 write 系统调用；*Asynchronous IO* 依据*前摄器*模式[21]来执行请求；而 *Reactive IO* 则使用*反应器*模式[14]。

InputOutput 组件由*接受器*通过相关联的流、从 2.3.3 描述的 *Task* 组件创建。文件操作通过 2.3.6 描述的 *Filecache Handle*（文件缓存句柄）组件来执行。例如，send_file 操作将由 Filecache Handle 表示的文件发送给由接受器返回的流。

2.3.5 协议流水线策略

2.3.5.1 设计挑战

早期的 Web 服务器，像 NCSA 最初的 httpd，执行的文件处理非常少。它们只是简单地取得所请求的文件，将其内容传输给请求者。但是，现代的 Web 服务器除了执行文件获取，还进行数据处理。例如，HTTP/1.0 协议可用于确定各种文件属性，比如文件类型（例如，文本、图像、音频或视频）、文件编码和压缩类型、文件大小，以及它的最后修改日期。这些信息通过 HTTP 头返回给请求者。

通过引入 CGI，Web 服务器甚至已经能够执行更为广泛的任务，包括搜索引擎、地图生成、数据库系统连接，以及安全的商业和金融交易，等等。但是，CGI 的限制是服务器必须派生新*进程*来扩展服务器功能。典型地，每个请求要求 CGI 派生它自己的进程来处理它，致使服务器成了 Process-per-Request 服务器，一种性能“抑制剂”[3]。高性能 Web 服务器框架的挑战是：允许开发者扩展服务器功能，而又不诉诸 CGI 进程。

2.3.5.2 可选策略方案

概念上，大多数 Web 服务器都在若干阶段中进行数据流处理或变换。例如，处理 HTTP/1.0 请求的诸阶段可被组织为一系列任务。这些任务涉及（1）读入请求，（2）解析请求，（3）解析请求头信息，（4）执行请求，以及（5）生成请求日志。如图 2-14 所示，这一系列任务构成了处理到来的请求的任务*流水线*（pipeline）。

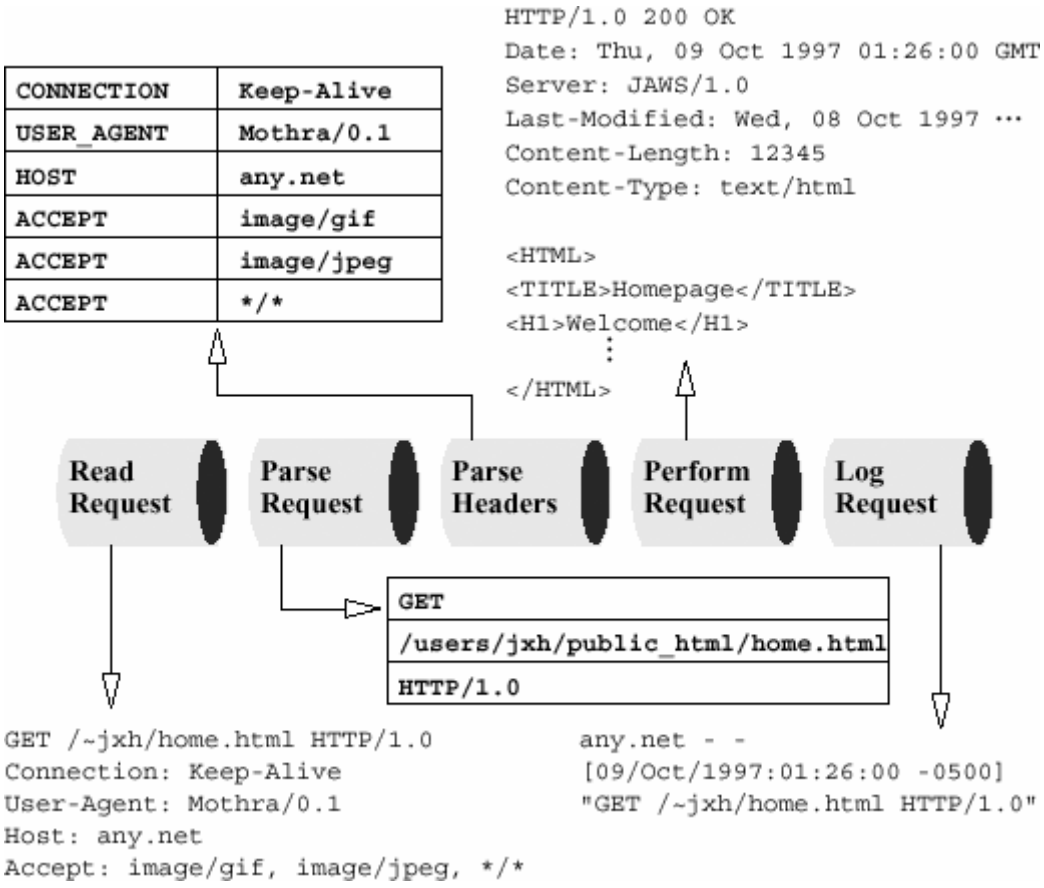

图 2-14 为 HTTP 请求而执行的任务流水线

处理 HTTP/1.0 请求所执行的任务有着固定的结构。因而，图 2-14 演示了一种*静态流水线*配置。对于被请求执行有限数目的处理操作的服务器扩展来说，静态配置是有用的。如果这些操作相对较少，并且已被预先了解的话，它们可以预先制作，并在服务器执行过程中直接使用。静态处理的例子包括：数据整编和数据去整编、通过自定义 Web 协议层进行的数据多路分离，以及编译 applet 代码。将静态流水线结合进服务器中，使得开发者能够扩展服务器功能，而又不求助于派生外部进程。Apache Web 服务器通过使用*模块*（module）提供了这种类型的可扩展性；模块封装静态处理，并且动态地链接到 Web 服务器。

但是，有些情况可能需要*动态*配置操作流水线。它们发生在这样的时候：Web 服务器扩展涉及任意的处理流水线配置，所以数目在本质上是没有限制的。如果有大量中间的流水线组件，可以通过任意的次序排列，就有可能发生上述情况。如果对数据的操作只有在程序执行的过程中才能知道，通过组件动态地构造流水线就提供了一种经济的解决方案。有许多例子可说明动态流水线何时有用，包括：

高级搜索引擎：根据所提供的查询字符串，搜索引擎可以动态地构造数据过滤器。例如，像“ (performance AND (NOT symphonic)) ”这样的查询查找含有单词“ performance ”、但没有单词“ symphonic ”的 Web 页面；它可被实现为由一个肯定的匹配组件和一个否定的匹配组件耦合而成的流水线。

图像服务器：图像服务器可以根据用户所请求的操作来动态地构造过滤器。例如，用户可以请求剪切、缩放、旋转和抖动图像。

自适配 Web 内容：Web 内容可以根据最终用户的特性而动态地递送。例如，个人数字助理（PDA）接收的应该是 Web 内容综述和较小的图像，而工作站可以接收全部的富含多媒体的页面。同样地，家用电脑用户可以选择屏蔽某些类型的内容。

现有的允许开发者动态增强 Web 服务器功能的解决方案不是太针对特定应用，就是太过一般化。例如，Netscape 的 NetDynamics 整个地聚焦于 Web 服务器与现有数据库应用的集成。相反，基于 Java 的 Web 服务器（比如 Sun 的 Java Server 和 W3C 的 Jigsaw）允许进行任意的服务器扩展，因为 Java 应用有能力动态执行任意的 Java 代码。但是，怎样提供可动态配置的操作流水线的问题仍然要由服务器开发者来解决。因此，开发者需要在不利用基于设计模式的应用框架所提供的好处的情况下，自己定制解决方案的设计。

构造服务器来处理逻辑阶段的数据被称为*管道和过滤器*（Pipes and Filters）模式[23]。在该模式帮助开发者考虑怎样组织处理流水线的组件的同时，它并没有提供完成这些工作的框架。没有这样一个框架，改编自定义服务器、以高效地采用新协议特性的任务可能会太过昂贵、困难，或产生维护开销很高的代码。

2.3.5.3 JAWS 协议流水线策略框架

前面的讨论激发了开发者对扩展服务器功能、而又不诉诸外部进程的需要。我们还描述了怎样将*管道和过滤器*模式应用于创建静态和动态的信息处理流水线。JAWS 的*协议流水线框架*被设计用于简化编写这些流水线所需的工作。这是通过提供流水线组件的*任务骨架*（task skeleton）来完成的。当开发者完成流水线组件逻辑时，组件可随即与其他组件组合，以创建一条流水线。已完成组件被存储进仓库中，以使它们可在服务器运行时被访问。这使得服务器框架可以在 Web 服务器执行时按照需要动态地创建流水线。

图 2-15 提供了对框架结构的演示。该图描述*协议处理器*（Protocol Handler），它利用*协议流水线*（Protocol Pipeline）来处理到来的请求。*过滤器*（Filter）组件派生自*协议流水线*，并被用作流水线组件的任务骨架。

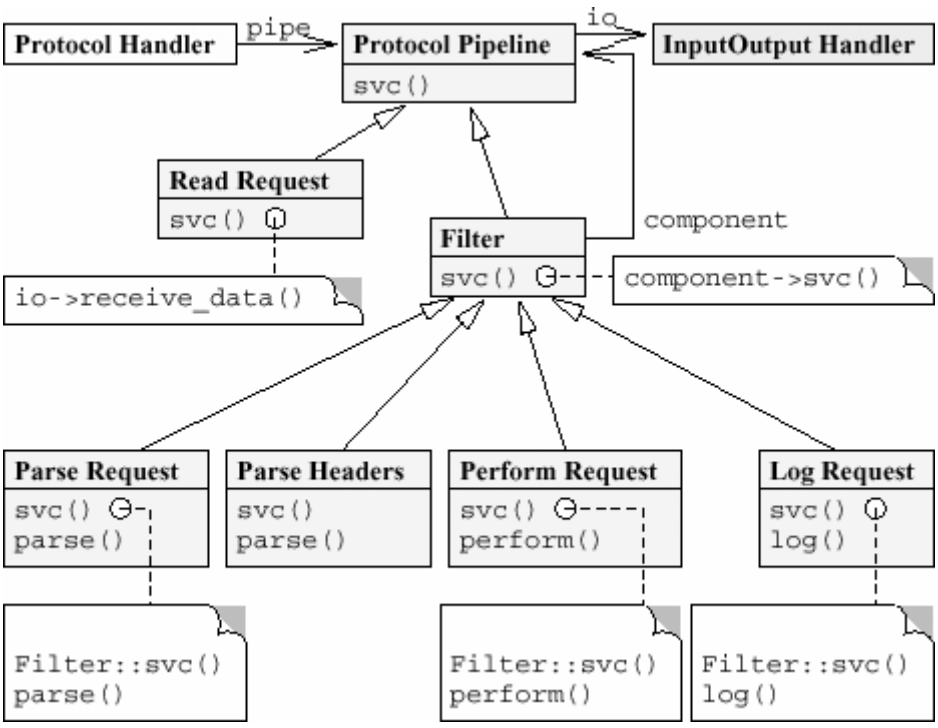

图 2-15 协议流水线框架的结构

流水线的实现者从*过滤器*派生来创建流水线组件。每个流水线组件的 svc 方法首先调用父 svc，由其获取要处理的数据，并随即针对数据执行它的组件特有的逻辑。父 svc 调用前面组件的 svc 方法。因而，组件的组合产生了一条调用链，拖着数据通过流水线。这条调用链在负责获取原始输入的组件那里结束，该组件直接派生自*协议流水线*抽象。

2.3.6 文件缓存策略

2.3.6.1 设计挑战

[3]中的结果显示访问文件系统是 Web 服务器的开销的重要来源。大多数分布式应用都可以从缓存中获益，Web 服务器也不例外。因此，不奇怪，对 Web 服务器性能的研究会聚焦于怎样通过文件缓存来获得更好的性能[29, 30]。

缓存是一种存储媒介，它提供比所需信息通常所在的媒介更为高效的检索。在 Web 服务器的情况下，缓存驻留在服务器的主内存中。因为内存是有限的资源，文件只是暂时地驻留在缓存中。因而，围绕最佳缓存性能的问题是由*数量*（有多少信息应被存储在缓存中）和*持续时间*（信息应在缓存中停留多长时间）来决定的。

2.3.6.2 可选策略方案

分配给缓存的内存大小会极大地影响数量和持续时间。如果内存不足，缓存少量大文件可能是不合需要的，因为缓存许多较少的文件能给出更好的平均性能。如果有更多的内存可用，缓存较大的文件也可能是可行的。

最近最少使用（LRU）缓存：这种缓存替换策略假定大多数对已缓存文件的请求都有着*时间局部性*（temporal locality），也就是，一个被请求的文件将很快被再度请求。因而，当把新文件插入缓存、要求移去另一个文件时，*最近最少使用*的文件将会被移除。该策略与提供具有临时特性的内容（比如每日新闻报告和股票报价）的 Web 系统有关。

最不经常使用（LFU）缓存：这种缓存替换策略假定已被频繁请求的文件最有可能被再度请求，这是另一种形式的时间局部性。因而，*最不经常使用*的缓存文件将是缓存中第一个被替换的文件。该策略与提供相对静态的内容的 Web 系统（比如 Lexis-Nexis 和其他历史事实数据库）有关。

提示式缓存：这种形式的缓存是在[29]中提议的。该策略源于对 Web 页面获取模式的分析，它们似乎表明 Web 页面有着*空间局部性*（spatial locality）。就是说，浏览一个 Web 页面的用户也很有可能浏览该页面中的链接。提示式缓存与“*预取*”（pre-fetching）有关，虽然[29]建议修改 HTTP 协议，以使连接的统计信息（或提示）能够被发回给请求者。此修改允许客户决定预取哪些页面。同样的统计信息还可用于使服务器确定*预先缓存*哪些页面。

结构化缓存：这种缓存了解正在缓存的数据。对于 HTML 页面，结构化缓存指的是存储缓存文件、以支持单个 Web 页面的层次浏览。因而，缓存利用 Web 页面中出现的结构来确定将要传输给客户的最为相关的部分（例如，页面的顶级视图）。对于带宽和主内存有限的客户，比如 PDA，这有可能加快 Web 访问。结构化缓存与数据库中 B 树的使用有关，B 树能够使获取查询数据所需的磁盘访问的次数最小化。

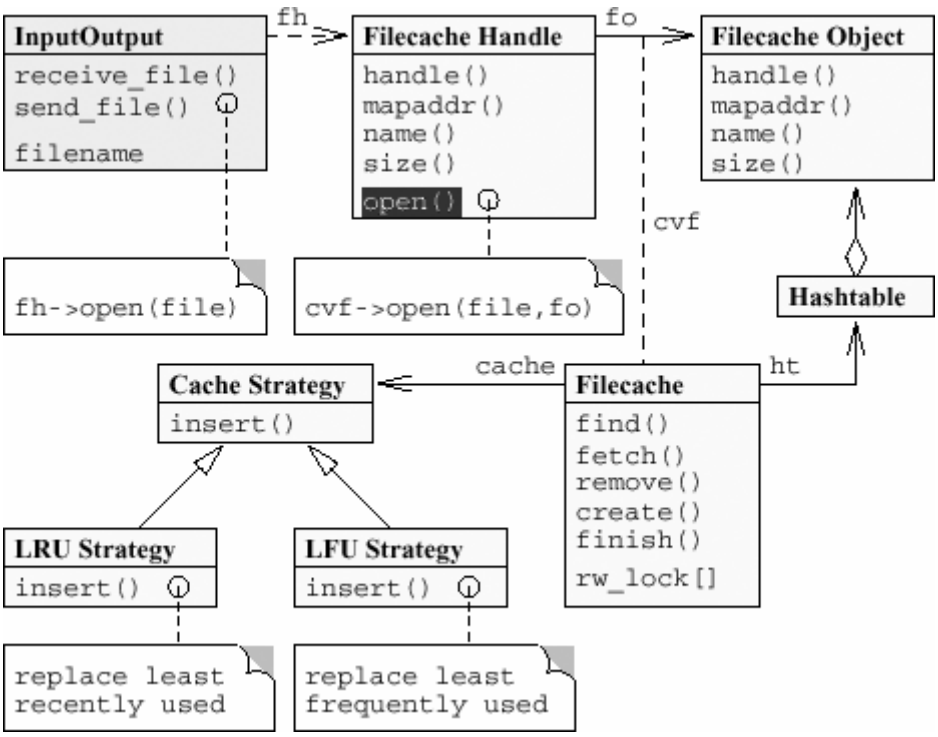

图 2-16 缓存虚拟文件系统的结构

2.3.6.3 JAWS 缓存式虚拟文件系统框架

上面描述的解决方案给出了实现文件缓存的若干策略。但是采用固定的缓存策略并不总能提供最佳的性能[31]。JAWS 缓存式虚拟文件系统框架以两种方法来解决此问题。其一是使开发者很容易地将缓存替换算法和缓存策略集成进框架中。而且，通过利用策略 Profile，可以动态地选择这些算法和策略，以在变化的服务器负载条件下优化性能。

图 2-16 演示 JAWS 中的一些组件，它们互相协作、以构造缓存式虚拟文件系统框架。*InputOutput* 对象对 *Filecache Handle* 进行实例化，文件交互（比如读和写）通过该句柄来进行管理。*Filecache Handle* 引用 *Filecache Object*，后者是由 *Filecache* 组件来管理的。*Filecache Object* 维护在所有引用它的句柄间共享的信息，比如内存映射文件的基地址。*Filecache* 组件是缓存虚拟文件系统框架的心脏。它通过哈希方法来管理 *Filecache Object*，并且在选择 *Cache Strategy* 来处理文件请求时遵循*状态*模式。*Cache Strategy* 组件利用*策略*模式来使不同的缓存算法（比如 LRU 和 LFU）能够互相替换。

2.3.7 JAWS 框架回顾

2.2 激发了对通过框架和模式来构建高性能 Web 服务器的需要。我们使用 JAWS 作为例子来演示框架和模式是怎样使程序员避开开发 Web 服务器软件的常见陷阱的。模式和框架共同支持集成化组件和设计抽象的复用 [4]。

2.3 描述怎样建构 JAWS 框架，以及它所提供的策略。为了清晰地说明 JAWS 的设计组织，我们概述了对 JAWS 框架有最大影响的战略和战术设计模式。最为重要的战略模式有*接受器*、*反应器*、*前摄器*、*主动对象*和*服务配置器*模式，等等。*策略*、*适配器*、*状态*和*单体*模式是 JAWS 中所用的有影响的战术模式。

这些模式被用来提供 JAWS 体系结构的“ 脚手架 ”。它们描述了必需的、形式为较小的组件的协作实体。从这里开始，框架的主要组件（也就是，*并发策略*、*I/O 策略*、*协议流水线*和*缓存式虚拟文件系统*框架）被构建和集成进在一个骨架应用中。通过定制框架的特定子组件（例如，*过滤器*和*缓存策略*），开发者可以使用该骨架来构造专门的 Web 服务器软件系统。

图 2-17 显示 JAWS 框架的所有模式和组件是怎样集成的。JAWS 框架以下面的方式来推动高性能 Web 服务器的构建。首先，它提供若干预配置的并发和 I/O 分派模型、一个提供标准的缓存替换策略的文件缓存，以及一个用于实现协议流水线的框架。其次，JAWS 中所用的模式有助于使 Web 服务器策略与（1）实现细节和（2）配置时间去耦合。这样的去耦合使得新策略可以很容易地集成进 JAWS。最后，JAWS 的设计允许服务器静态和动态地改变它的行为。例如，通过适当的策略 Profile，JAWS 可以适应在 Web 服务器执行期间所发生的不同情况。通过进行动态适配来应用这些可扩展的策略和组件，JAWS 简化了高性能 Web 服务器的开发和配置。

2.4 Web 服务器基准测试床和实验结果

2.3 描述的模式和框架组件使得 JAWS 能够静态和动态地适应它的环境。尽管这样的灵活性是有益的，Web 服务器的成功最终还是要依赖于它怎样满足 Internet 以及企业 Intranet 的性能需求。要满足这些需求，需要对影响 Web 服务器性能的关键因素有全面的了解。

这一部分描述在系统地将不同负载条件应用于 JAWS 的不同配置时，我们所获得的性能结果。这些结果说明了为什么没有哪种配置能够在所有负载条件下最优地执行。这也论证了对像 JAWS 这样的灵活的框架的需要；这样的框架可在变化的负载条件下重配置、以获取最优性能。

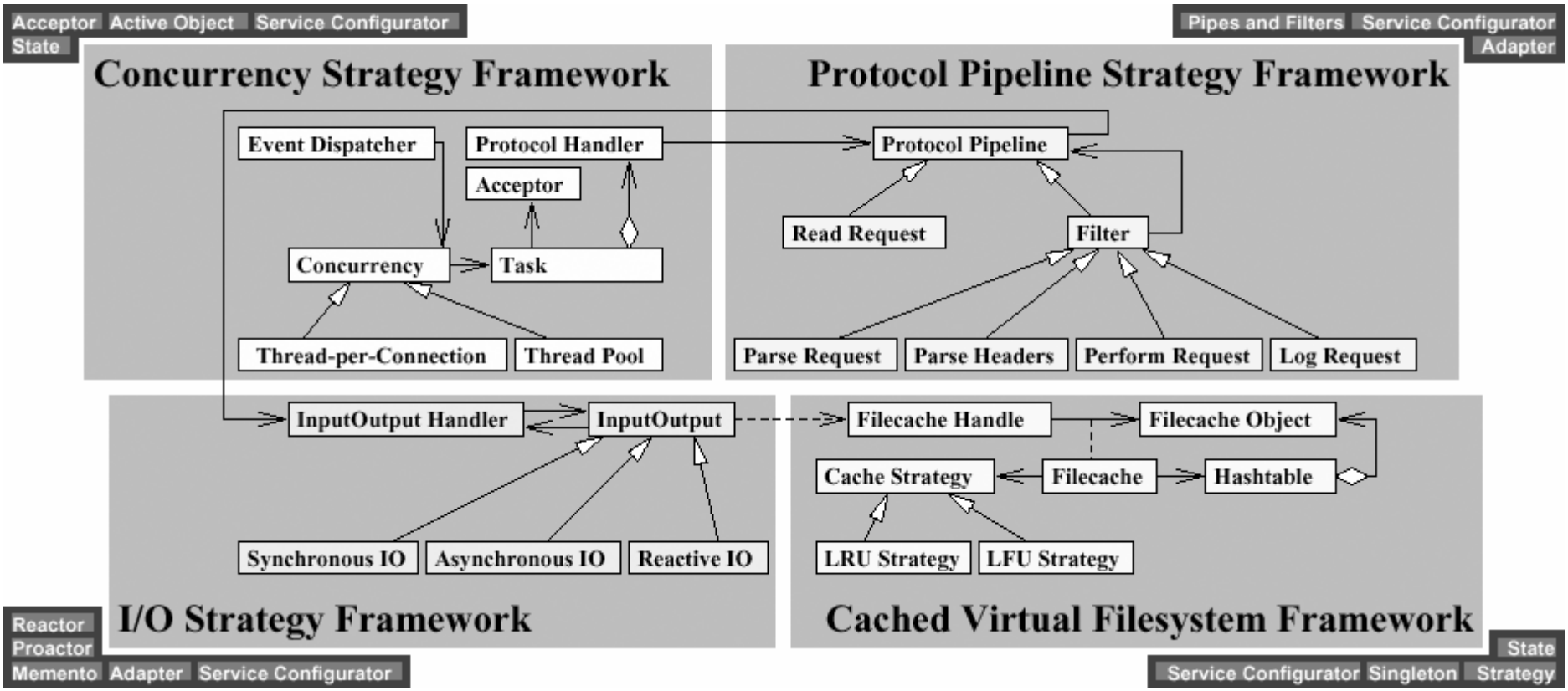
Acceptor
Active Object
Service Configurator
State
Concurrency Strategy Framework
Event Dispatcher
Protocol Handler
Acceptor
Concurrency
Task
Thread-per-Connection
Thread Pool
Pipes and Filters
Service Configurator
Adapter
Protocol Pipeline Strategy Framework
Protocol Pipeline
Read Request
Filter
Parse Request
Parse Headers
Perform Request
Log Request
InputOutput Handler
InputOutput
Synchronous IO
Asynchronous IO
Reactive IO
I/O Strategy Framework
Filecache Handle
Filecache Object
Cache Strategy
Filecache
Hashtable
LRU Strategy
LFU Strategy
Cached Virtual Filesystem Framework
Reactor
Proactor
Memento
Adapter
Service Configurator
State
Service Configurator
Singleton
Strategy

图 2-17 JAWS Web 服务器框架

2.4.1 硬件测试床

我们的硬件测试床如图 2-18 所示。测试床由两台 Micron Millennia PRO2 plus 工作站组成。每台 PRO2 有 128 MB RAM，并配有 2 个 Pentium Pro 处理器。客户机的时钟速度为 200 MHZ，而服务器则为 180 MHZ。此外，每台 PRO2 还有一块由 Efficient Networks, Inc 制造的 ENI-155P-MF-S ATM 卡，并由 Orca 3.01 驱动软件来驱动。两台工作站经由 ATM 网络相连，网络通过 FORE Systems ASX-200BX 运行，最大带宽为 622 Mbps。但是，由于 LAN 竞争模式的限制，我们的测试床的最高带宽大约为 120 Mbps。

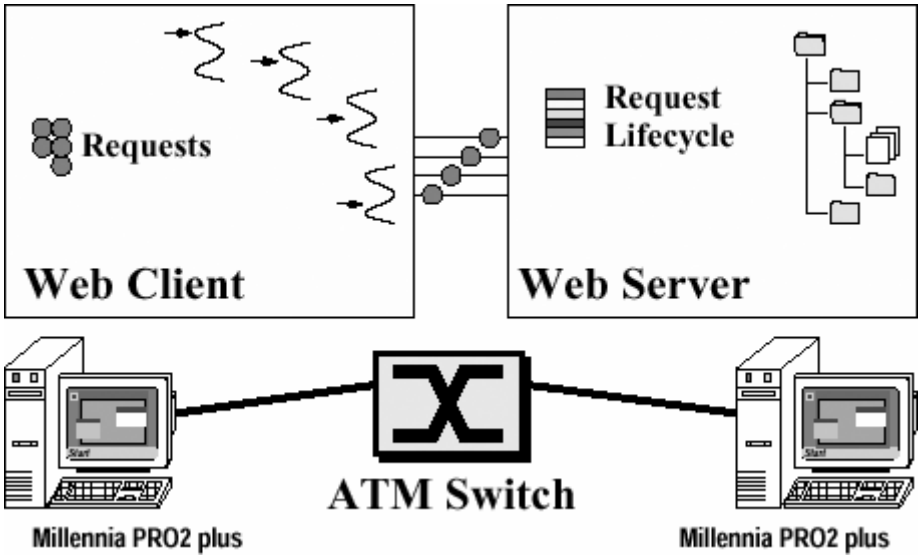

图 2-18 基准测试床概览

2.4.2 软件请求生成器（Software Request Generator）

我们使用 WebSTONE[32] v2.0 基准测试软件来收集客户和服务器端的测试数据。这些数据包括*服务器平均吞吐量*和*客户平均响应时间*。WebSTONE 是一个标准的基准测试工具，可以生成负载请求、模拟典型的 Web 服务器文件访问模式。我们的实验使用 WebSTONE 来为特定大小的文件生成负载及收集统计数据，以确定不同的并发和事件分派策略的影响。

在此测试中所用的文件访问模式如表 2-1 所示。该表显示在流行的服务器上的实际负载条件，其数据来源于 SPEC 所指导的一项对文件访问模式的研究[33]。

文档大小	频度

500 字节	35%
5K 字节	50%
50K 字节	14%
5M 字节	1%

表 2-1 文件访问模式

2.4.3 实验结果

下面给出的结果比较了 JAWS Web 服务器的若干不同的适配版本的性能。我们将讨论不同的事件分派和 I/O 模型对于吞吐量和响应时间的影响。为这次实验，使用了 JAWS 的三种适配版本。

1. **同步 Thread-per-Request**：在此适配版本中，JAWS 被配置成派生一个新线程来处理每个到来的请求，并且使用同步 I/O。
2. **同步线程池**：JAWS 被配置成在使用同步 I/O 的同时、预派生一个线程池来处理到来的请求。
3. **异步线程池**：对于这种配置，JAWS 被配置成在为异步 I/O 使用 TransmitFile 的同时、预先派生一个线程池来处理到来的请求。TransmitFile 是 Win32 函数，它在网络连接上同步或异步地发送文件。

吞吐量被定义为每秒钟客户接收到的平均 bit 数。在客户基准测试软件发送 HTTP 请求之前，启动了一个高精度的用于吞吐量测量的定时器。只要连接在客户端一被关闭，定时器就会停止。接收到的 bit 数包括服务器发送的 HTML 头。

响应时间被定义为从客户发送请求开始，到完整地接受完文件为止、客户所看到的以毫秒为单位的延迟量。它测量在发送 HTTP GET 请求给 Web 服务器之后、并且在内容开始到达客户之前，最终用户必须等待多长时间。客户基准测试软件一发送 HTTP 请求，用于响应时间测量的定时器就会启动；客户一完成对所请求的来自服务器的文件的接收，定时器就会停止。

下面的五幅图分别是实验中所用的不同大小的文件（从 500 字节到 5M 字节，因子为 10）的测试结果。这些文件大小代表在我们的实验中所测量的文件大小的“ 谱段 ”，用以揭示文件大小对性能都有什么影响。

2.4.3.1 吞吐量比较

图 2-19 到图 2-23 演示当所请求文件的大小和服务器点击率系统地增加时，吞吐量的变化情况。如我们所预期的，在每秒的连接数增长时，每个连接的吞吐量通常会退化。这源于正在被维护的同时连接数的增长，导致了每个连接吞吐量的下降。

如图 2-21 所示，当连接负载增长时，对于较小文件，Thread-per-Request 的吞吐量有可能迅速退化。相反，同步线程池实现的吞吐量退化要更为温和。这种差异的原因是 Thread-per-Request 导致更高的线程创建开销，因为要为每个 GET 请求派生一个新线程。相反，线程池策略中的线程创建开销通过在服务器开始执行时预先派生线程而分摊了。

图 2-19 到 2-23 中的结果说明对于小文件（也就是，< 50K 字节），TransmitFile 执行得异常糟糕。我们的实验表明 TransmitFile 的性能直接依赖于同时请求数。我们相信在繁重的服务器负载情况下（也就是，

高点击率），TransmitFile 被迫在内核处理到来的请求时进行等待。这创建了数目很大的同时连接，从而降低了服务器性能。

但是在文件的的大小增长时，TransmitFile 很快就胜过了同步分派模型。例如，在 5M 字节文件（在图 2-23 中显示）的繁重负载下，它超出下一种最为接近的模型大约 40%。TransmitFile 被优化以利用 Windows NT 内核特性，从而减少了数据拷贝和上下切换的次数。

2.4.3.2 响应时间比较

图 2-19 到 2-23 演示当所请求文件的大小和服务器点击率增加时，响应性能的变化情况。如我们所预期的，在每秒的连接数增长时，每个连接的响应时间通常也会变长。这反映出加在服务器上的额外负载，降低了它服务新客户请求的能力。

和前面一样，对于小文件，TransmitFile 执行得异常糟糕。但是，在文件大小增长时，相对于轻负载情况下的同步分派，它的响应时间得到了迅速的改善。

2.4.3.3 基准测试结果总结

如这一部分中的结果所示，取决于并发和事件分派机制，吞吐量和响应时间有着显著的变化。对于小文件，同步线程池策略提供了更好的整体性能。在中等负载下，同步事件分派模型提供的响应时间要略微优于异步模型。但是，在传输大文件的繁重负载情况下，使用 TransmistFile 的异步模型提供了更好的服务质量。因而，在 Windows NT 下，取决于服务器的工作负载和文件请求的分布，理想的 Web 服务器应该将自身适配到合适的事件分派和文件 I/O 模型。

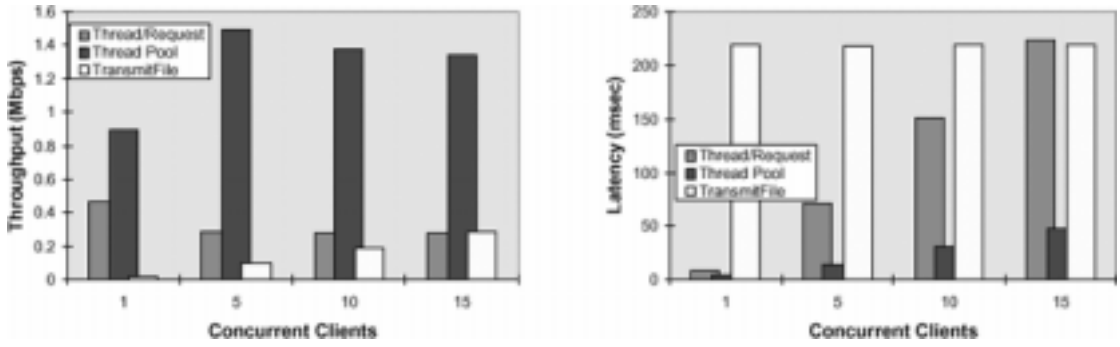

图 2-19 500 字节文件的实验结果

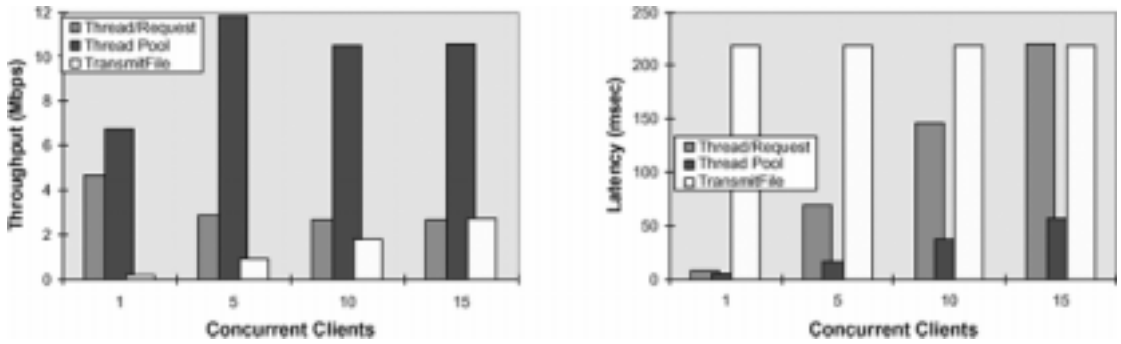

图 2-20 5K 文件的实验结果

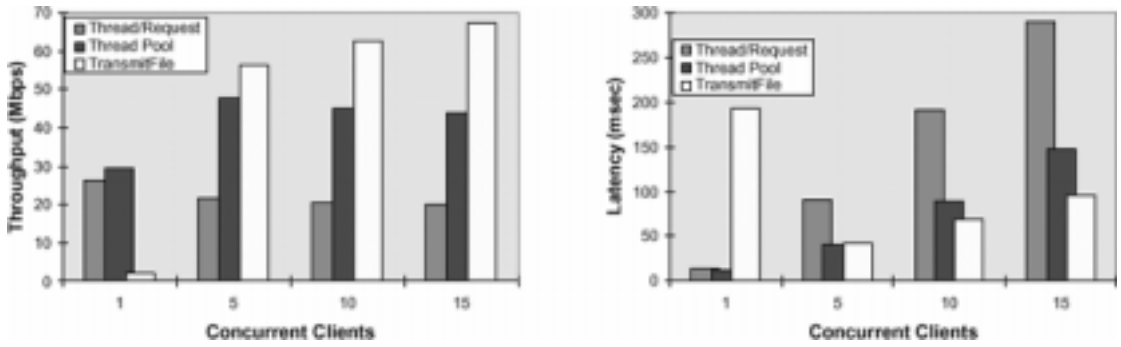

图 2-21 50K 文件的实验结果

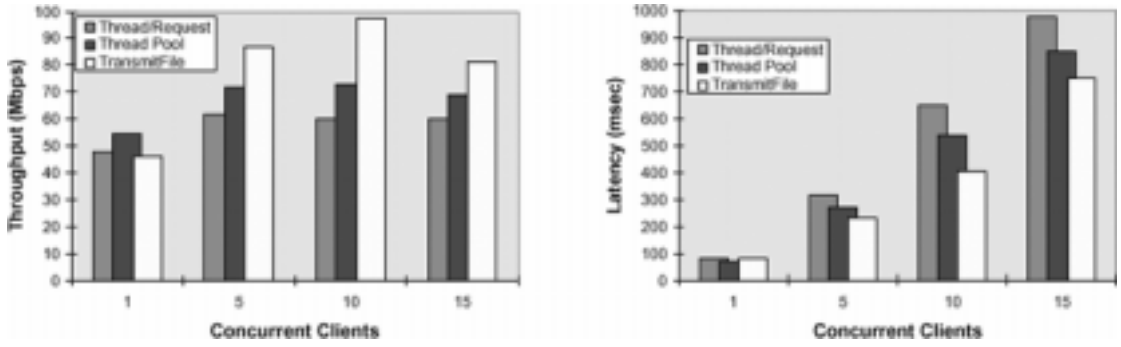

图 2-22 500K 文件的实验结果

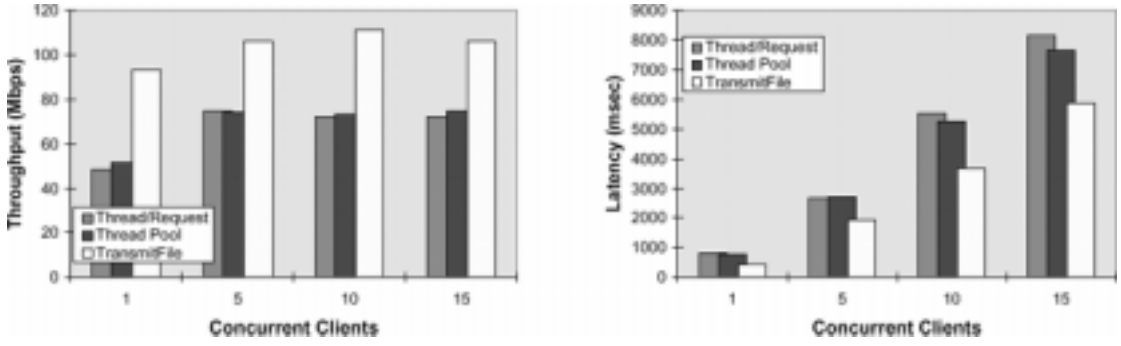

图 2-23 5M 文件的实验结果

2.4.4 用于优化 Web 服务器的技术总结

在我们的研究中，我们已经发现有可能通过更好的服务器设计来改善服务器性能（在[34]中给出了类似的观察报告）。因而，在“硬编码”服务器（也就是，使用固定并发、I/O 和缓存策略的服务器）能够不可否认地提供极好的性能的同时，灵活的服务器框架，比如 JAWS，并不必然与糟糕的性能相联系。

这一部分总结决定 Web 服务器性能的最为重要的因素。这些观察报告基于我们对现有服务器设计和实现策略、以及我们调谐 JAWS 的经验的研究[1, 3]。这些研究揭示了开发高性能 Web 服务器的主要优化目标。

轻量级并发：如在[3]中所看到的，基于进程的并发机制可能会产生糟糕的性能。在多处理器系统中，基于进程的并发机制可以良好地执行，特别是在*进程数目与处理器数目相等*时。在这种情况下，每个处理器可以运行一个 Web 服务器进程，从而使上下文切换开销得以最小化。

通常，进程应被*预先生成*，以避免动态进程创建的开销。但是，更可取的方法是使用轻量级并发机制（例如，使用 POSIX 线程）来使上下文切换开销最小化。和进程一样，动态线程创建开销也可以通过在服务启动时将线程*预先派生*进线程池中来加以避免。

专用 OS 特性：OS 供应商常常会提供能给出更好性能的专用编程接口。例如，Windows NT 4.0 提供 TransmitFile 函数，使用 Windows NT 虚拟内存缓存管理器来获取文件数据。TransmitFile 允许在文件数据之前和之后分别增加数据。这特别适用于 Web 服务器，因为它们通常与所请求文件一起发送 HTTP 头数据。因此，所有给客户的数据可以在单个的系统调用中发送，从而使模式切换开销得以最小化。

通常，这些接口必须相对于标准 API 仔细进行基准测试，以了解特定接口给出更好性能的条件。如 2.4 所示，在使用 TransmitFile 的情况下，我们的实验数据表明，在 Windows NT 上通过 Socket 传输大文件，异步形式的 TransmitFile 是最为高效的机制。

请求生存期系统调用开销：在 Web 服务器中，请求生存期被定义为服务器在它从客户接收 HTTP 请求之后、以及在它发送出所请求文件之前，所必须执行的一系列指令。执行请求生存期所用的时间直接影响客户观察到的响应时间。因此，在此路径上使系统调用开销和其他处理最小化是很重要的。下面描述在 Web 服务器中可以减少这样的开销的各种地方。

- **减少同步：**在处理并发时，常常需要通过同步来序列化对共享资源的访问（比如缓存式虚拟文件系统）。但是，使用同步会使性能恶化。因而，在请求生存期中使所获取（或释放）的锁的数目最小化是很重要的。在[3]中显示了每个请求所进行的平均锁操作数较低的服务器比进行大量锁操作的服务器执行得要好得多。

  在某些情况下，获取和释放锁还可能导致*占先*（preemption）。因而，如果线程读进 HTTP 请求，随后试图获取一个锁，它就有可能会被占先，并可能在它被再次分派之前等待相对较长的时间。这会增加 Web 客户所经受的响应延迟。
- **缓存文件：**如果服务器不进行文件缓存，至少会带来两方面的开销。首先是来自 open 系统调用的开销。其次，反复使用 read 和 write 系统调用来访问文件系统会带来累积的开销，除非文件小到足以在单次调用中被获取或存储。使用在大多数形式的 UNIX 和 Windows NT 上可用的内存映射文件，可以

有效地进行缓存。

- **使用“集中－写”（gather-write）：**在 UNIX 系统中，writev 系统调用允许在单个系统调用中将多个缓冲区写入设备。这对于 Web 服务器来说是有用的，因为除了所请求的文件，典型的服务器响应还含有许多行头信息。通过使用“集中－写”，不需要在发送前将这些头的行连接进单个缓冲区，从而避免了不必要的数据复制。
- **预先计算的 HTTP 响应：**典型的 HTTP 请求使得服务器发回 HTTP 头，其中含有 HTTP 成功代码和所请求文件的 MIME 类型（例如，text/plain）。因为这样的响应是预期情况的一部分，它们可以被*预先计算*。当文件进入缓存时，响应的 HTTP 响应也可以和文件存储在一起。于是当 HTTP 请求到达时，头在缓存中已直接可用了。

日志开销：大多数 Web 服务器允许管理员对他们所服务的不同页面的点击数进行日志记录。日志常常用于评估服务器在一天的不同时间里的负载情况。它还常常因为商业原因而使用，例如，网站可能基于页面的点击率来确定广告的费率。但是，因为以下原因，对 HTTP 请求进行日志记录会产生显著的开销：

- **文件系统开销**：繁忙的 Web 服务器执行相当多的 I/O 调用，从而压迫文件系统和底层硬件。将数据写到日志文件会增加这样的压力，从而导致更低的性能。将日志文件和 HTTP 文件放置在分开的文件系统上（且在分开的物理设备上，如果可能）可以限制这一开销。
- **同步开销：**典型的 Web 服务器有多个活动线程或进程在对请求进行服务。如果这些线程/进程需要将请求记录到公用的共享日志文件，对该日志文件的访问就需要被同步，也就是，在任何时刻至多只有一个线程/进程可以对该共享日志文件进行写操作。这样的同步引入了额外的开销，从而有害于性能。通过使用多个独立的日志文件，可以减少这种开销。如果使用了内存缓冲区，它们就应该被存储在*线程专有存储*（thread-specific storage）中，以消除锁定竞争。
- **反向主机名查找：**客户的 IP 地址对于 Web 服务器来说是在本地的。但是，在日志文件中主机名通常更为有用。因而，客户的 IP 地址需要被转换为相应的主机名。这通常是使用*反向* DNS 查找来完成的。因为常常涉及网络 I/O，这些查找非常昂贵。因此，它们应该被避免，或者异步地完成（使用线程或异步 I/O）。
- **Ident 查找：**Ident 协议[35]允许 Web 服务器获取给定 HTTP 连接的用户名。这通常涉及设置新的到用户机器的 TCP/IP 连接，因而也涉及了“往返旅程”（round-trip）延迟。还有，ident 查找必须在 HTTP 连接仍在活动时执行，因此不能够“懒洋洋”地执行。因而，为获得高性能，只要可能，都应该避免进行这样的查找。

传输层优化：应该配置下面的传输层选项，以提高 Web 服务器在高速网络上的性能：

- **侦听 backlog：**大多数 TCP 实现都在驻留内核的“侦听队列”上缓冲到来的 HTTP 连接，以使服务器能够使用 accept 使它们出队、进行服务。如果 TCP 队列超出了 listen 调用的“backlog”参数，新的连接就将被 TCP 拒绝。因而，如果预计到来的连接的流量将会很大的话，就应该通过设置更大的 backlog 参数（这有可能需要修改 OS 内核）来增大内核队列的容量。
- **Socket 发送缓冲区：**每个 Socket 都有一个与之相关联的发送缓冲区，该缓冲区在数据正通过网络传输时持有这些由服务器发送的数据。为获得高性能，该缓冲区的大小应被设置为所允许的最大值（也就是，大缓冲区）。在 Solaris，这个限制为 64K。
- **Nagle 算法（RFC 896）：**某些 TCP/IP 实现通过实现 Nagle 算法来避免*拥塞*（congestion）。这常常导致数据在实际通过网络发送之前、被网络层所迟滞。若干对响应时间有严格要求的应用（比如 X-Windows）禁用了这一算法（例如，Solaris 支持 TCP_NO_DELAY Socket 选项）。通过强制网络层尽可能快地发送数据，禁用这一算法可以改善响应时间。

2.5 结束语

在计算能力和网络带宽在过去十年间戏剧性增长的同时，开发通信软件所涉及的复杂性同样也发生了戏剧性增长。特别地，高性能 Web 服务器软件的设计和实现是昂贵而易错的。许多代价和工作都来自于对基本的设计模式和框架组件持续的重新发现和发明。而且，越来越多的硬件体系结构异种性和 OS 及网络平台间的差异使得从头开始构建正确、可移植和高效的 Web 服务器变得十分困难。

构建高性能 Web 服务器需要了解服务器中的每个子系统（例如，并发和事件分派、服务器请求处理、文件系统访问和数据传输）对于性能的影响。高效地实现和集成这些子系统要求开发者迎接各种设计挑战，并在可选的解决方案中进行选择。领会在这些可选方案之间进行的权衡对于提供最优性能来说是必需的。但是，在需求（不可避免地）变化时，开发特定设计所需的工作常常并非是经济的。例如，性能可能被证明为不足、可能需要另外的功能，或是软件需要被移植到不同的体系结构。

通过有效利用已被证明的设计模式和实现、以产生可被定制来满足新应用需求的可复用组件，面向对象应用框架和设计模式有助于减少开销、并改善软件的性能。在本论文中描述的 JAWS 框架演示了怎样简化和统一高性能 Web 服务器的开发。JAWS 成功的关键是它能够捕捉常见的通信软件设计模式，并将这些模式统一进灵活的框架组件中；这些组件高效地封装并增强了低级的 OS 机制：并发、事件多路分离、动态配置、文件缓存，等等。这里给出的基准测试结果用于演示使用框架来开发高性能应用的有效性。它照亮了这一事实：可适应不同情况的软件与性能并非是对立的。

JAWS 的源码和文档可在 http://www.cs.wustl.edu/~schmidt/ACE.html 找到。

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] 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.

[3] J. Hu, S. Mungee, and D. C. Schmidt, “Principles for Developing and Measuring High-performance Web Servers over ATM,” in *Proceeedings of INFOCOM ’*98, March/April 1998.

[4] R. Johnson, “Frameworks = Patterns + Components,” *Communications of the AC*M, vol. 40, Oct. 1997.

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

[7] D. C. Schmidt, “Applying Design Patterns and Frameworks to Develop Object-Oriented Communication Software,” in *Handbook of Programming Languages* (P. Salus, ed.), MacMillan Computer Publishing, 1997.

[8] 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.

[9] T. H. Harrison, D. L. Levine, and D. C. Schmidt, “The Design and Performance of a Real-time CORBA Event Service,” in

Proceedings of OOPSLA '97, (Atlanta, GA), ACM, October 1997.

[10] D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible and Maintainable ORB Middleware," *Communications of the ACM, to appea*r, 1998.

[11] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating Syste*m. Addison Wesley, 1996.

[12] "Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application: Program Interface (API) [C Language]," 1995.

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

[14] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J.O.Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, MA: Addison-Wesley, 1995.

[15] 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.

[16] D. C. Schmidt, "Experience Using Design Patterns to Develop Reuseable Object-Oriented Communication Software," *Communications of the ACM (Special Issue on Object-Oriented Experiences*), vol. 38, October 1995.

[17] A. Stepanov and M. Lee, "The Standard Template Library," Tech. Rep. HPL-94-34, Hewlett-Packard Laboratories, April 1994.

[18] 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.

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

[20] P. Jain and D. C. Schmidt, "Service Configurator: A Pattern for Dynamic Configuration of Services," in *Proceedings of the 3rd Conference on Object-Oriented Technologies and System*s, USENIX, June 1997.

[21] T. Harrison, I. Pyarali, D. C. Schmidt, and T. Jordan, "Proactor – An Object Behavioral Pattern for Dispatching Asynchronous Event Handlers," in *The 4th Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-34*), September 1997.

[22] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Asynchronous Completion Token: an Object Behavioral Pattern for Efficient Asynchronous Event Handling," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

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

[24] D. C. Schmidt, "GPERF: A Perfect Hash Function Generator," in *Proceedings of the 2nd C++ Conferenc*e, (San Francisco, California), pp. 87–102, USENIX, April 1990.

[25] P. Jain and D. C. Schmidt, "Service Configurator: A Pattern for Dynamic Configuration and Reconfiguration of Communication Services," in *The 3rd Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-07*), February 1997.

[26] T. Berners-Lee, R. T. Fielding, and H. Frystyk, "Hypertext Transfer Protocol – HTTP/1.0," Informational RFC 1945, Network Working Group, May 1996. Available from http://www.w3.org/.

[27] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," Standards Track RFC 2068, Network Working Group, January 1997. Available from http://www.w3.org/.

[28] 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, MA: Addison-Wesley, 1996.

[29] J. C. Mogul, "Hinted caching in the Web," in *Proceedings of the Seventh SIGOPS European Workshop: Systems Support for*

*Worldwide Application*s, 1996.

[30] S. Williams, M. Abrams, C. R. Standridge, G. Abdulla, and E. A. Fox, “Removal Policies in Network Caches for World Wide Web Documents,” in *Proceedings of SIGCOMM ’9*6, (Stanford, CA), pp. 293–305, ACM, August 1996.

[31] E. P. Markatos, “Main memory caching of web documents,” in *Proceedings of the Fifth International World Wide Web Conferenc*e, May 1996.

[32] Gene Trent and Mark Sake, “WebSTONE: The First Generation in HTTP Server Benchmarking.” Silicon Graphics, Inc. whitepaper, February 1995. Available from http://www.sgi.com/.

[33] A. Carlton, “An Explanation of the SPECweb96 Benchmark.” Standard Performance Evaluation Corporation whitepaper, 1996. Available from http://www.specbench.org/.

[34] H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud’hommeaux, H. W. Lie, and C. Lilley, “Network Performance Effects of HTTP/1.1, CSS1, and PNG,” in *To appear in Proceedings of ACM SIGCOMM ’9*7, 1997.

[35] M. S. Johns, “Identification Protocol,” *Network Information Center RFC 141*3, Feb. 1993.

第 3 章 应用模式语言开发可扩展 ORB 中间件

Douglas C. Schmidt Chris Cleeland[3]

摘 要

分布式对象计算是下一代通信软件的基础。对象请求代理（Object Request Broker，ORB）是分布式对象计算的心脏，它使得许多麻烦而易错的分布式编程任务得以自动化。像许多通信软件一样，传统的 ORB 使用静态配置设计，难以移植、优化和发展。同样地，要扩展传统的 ORB，必须修改源代码，从而要求重新编译、链接和重启运行中的 ORB 以及与它们相关联的应用对象。

本论文对可扩展 ORB 中间件的研究作出的贡献有两个。首先，它给出的个案研究阐释怎样将模式语言用于开发可动态配置、并可为特定应用需求和系统特性进行定制的 ORB。其次，我们对应用这种模式语言来降低复杂性和改善常见的 ORB 任务的可维护性（比如连接管理、数据传输、多路分离，以及并发控制）的效果进行了量化。

3.1 介绍

有四种趋势正在塑造商业软件开发的未来。首先，软件工业正在从“从零开始”的应用*编程*迁移到使用可复用组件来*集成*应用[1]。其次，*分布式技术*提供远地方法调用和/或面向消息的中间件来简化应用协作，软件开发者对这样的技术有着巨大的需求。第三，业界正在努力定义标准的软件基础组织框架、以允许应用在*异种环境*间无缝地协同工作 [2]。最后，下一代分布式应用，比如视频点播、电话会议以及航空控制系统，要求保证响应时间、带宽和可靠性的*服务质量*(QoS) [3]。

顺应这些趋势的一种关键软件技术是*分布式对象计算（DOC）中间件*。DOC 中间件便利了在异种分布式环境中本地和远地应用组件的协作。它的目标是消除开发和发展分布式应用和服务的许多麻烦、易错和不可移植的方面。特别地，DOC 中间件使得一些常见的网络编程任务得以自动化，比如对象定位、实现启动（也就是，服务器和对象启用）、封装不同体系结构的字节序和参数类型大小的差异（也就是，参数整编）、容错，以及安全。对象请求代理（ORB）是 DOC 中间件的心脏，比如 CORBA[4]、DCOM[5]，以及 JAVA RMI[6]。

本论文描述我们怎样应用*模式语言*来开发和发展可动态配置、比静态配置的中间件要更容易扩展的 ORB 中间件。一般而言，通过将相关解决方案族应用于标准的软件开发问题，模式语言有助于减少软件概念和组件的持续的重新发送和发明[7]。例如，要为常见通信软件体系结构中的参与者之间的角色和关系编写文档，模式语言是十分有用的[8]。本论文中介绍的模式语言是在[9]中介绍的模式语言的一般化，并已被成功地用于构建灵活、高效、事件驱动和并发的通信软件，包括 ORB 中间件。

[3] 此工作部分地得到了 ATD、BBN、Boeing、Cisco、DARPA contract 9701516、Motorola Commercial Government and Industrial Solutions、Motorola Laboratories、Siemens 和 Sprint 的支持。

为使我们的讨论更集中，本论文介绍了一项个案研究，演示我们怎样将此模式语言应用于开发 *The ACE ORB*（TAO）[10]。TAO 是可自由使用、高度可扩展的 ORB，它瞄准的是有实时 QoS 需求的应用，包括航空任务计算[11]、多媒体应用[12]，以及分布式交互模拟[13]。可扩展设计是 TAO 的新颖之处，它的 ORB 在模式语言的指导下，能够进行动态定制、以满足和适应特定应用的 QoS 需求和网络/终端系统特性。

本论文的余下部分组织如下：3.2 给出 CORBA 和 TAO 的综述；3.3 说明使用动态配置的动机，并描述一种模式语言，用以应对在开发可扩展 ORB 时所面临的关键设计挑战； 3.3.5 评价并量化模式语言对 ORB 中间件的贡献；3.4 给出结束语。

3.2 CORBA 和 TAO 综述

这一部分概述 CORBA 参考模型，并描述 TAO 提供给高性能和实时应用的增强特性。

3.2.1 CORBA 参考模型综述

CORBA 对象请求代理（ORB）[14]允许客户调用分布式对象上的操作，而不用关心以下问题：

对象位置：CORBA 对象可以与客户共驻一处，或是分布在远地服务器上，而不会影响它的实现或使用。

编程语言：CORBA 支持的语言包括 C、C++、Java、Ada95、COBOL，以及 Smalltalk，等等。

OS 平台：CORBA 运行在许多平台上，包括 Win32、UNIX、MVS，以及实时嵌入式系统，比如 VxWorks、Chorus 和 LynxOS。

通信协议和互连：CORBA 可在以下通信协议和互连上运行：TCP/IP、IPX/SPX、FDDI、ATM、Ethernet、Fast Ethernet、嵌入式系统底板，以及共享内存。

硬件：CORBA 使应用和源于硬件多样性的副作用（比如不同的存储方案和数据类型大小/范围）屏蔽开来。

图 3-1 演示 CORBA 参考模型中的组件，它们相互进行协作来提供上面概述的可移植性、互操作性和透明性。

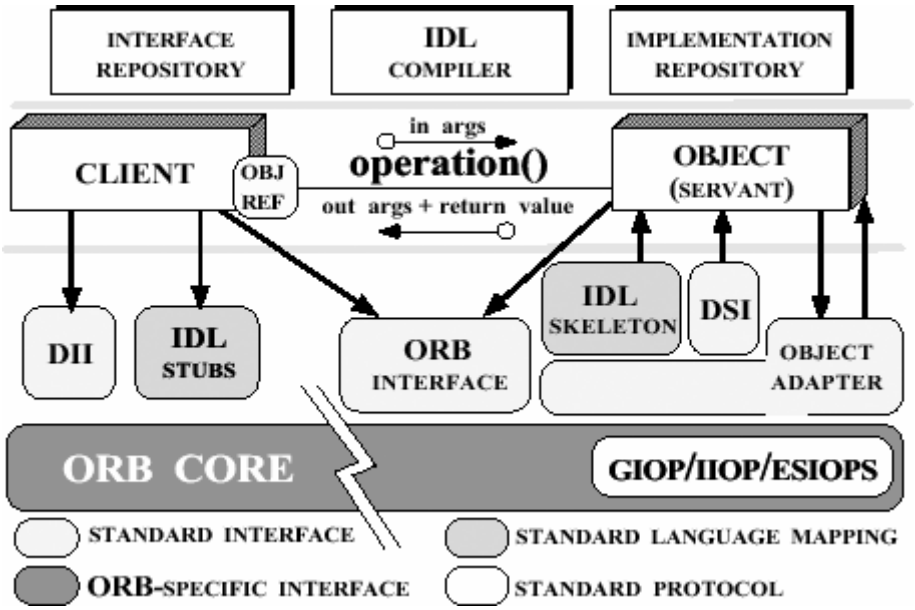

图 3-1 CORBA 参考模型中的组件

CORBA 参考模型中的每个组件概述如下：

客户（Client）：客户是一种获取对象的引用、在其上调用操作以执行应用任务的*角色*（role）。对象可以在远地、或与客户共驻一处。客户可以像访问本地对象那样访问远地对象，也就是，使用 object->operation(args)。图 3-1 显示下面描述的底层 ORB 组件怎样将远地操作请求从客户透明地传送给对象。

对象（Object）：在 CORBA 中，对象是 OMG 接口定义语言（IDL）接口的的实例。各个对象被*对象引用*（object reference）所标识，对象引用与一或多个路径相关联，通过这些路径客户可以访问服务器上的对象。*对象 ID* 将对象与它的实现（称为仆人，servant）相关联，它在对象适配器（Object Adapter）的范围内是唯一的。在对象的生存期内，有一或多个仆人与其关联，以实现它的接口。

仆人（Servant）：此组件实现由 OMG IDL 接口定义的操作。在面向对象（OO）语言（比如 C++和 Java）中，仆人使用一或多个类实例来实现。在非 OO 语言（比如 C）中，仆人通常使用函数和结构来实现。客户从不与仆人直接交互，而总是通过对象引用所标识的对象来进行。以*对象*作为 RefinedAbstraction、*仆人*作为 ConcreteImplementor，对象和它的仆人共同构成了桥接（Bridge）模式[15]的一种实现。

ORB 核心（ORB Core）：当客户调用对象上的操作时，ORB 核心负责将请求递送给对象；如果有任何响应，就将其返回给客户。ORB 核心是作为链接进客户和服务器应用的运行时库来实现的。对于远地执行的对象、遵循 CORBA 的 ORB 核心通过 General Inter-ORB Protocol（GIOP）的一种版本（比如运行在 TCP 传输协议之上的 Internet Inter-ORB Protocol，IIOP）来进行通信。此外，还可以定义定制的 Environment-Specific Inter-ORB Protocol（ESIOP）。

ORB 接口（ORB Interface）：ORB 是可通过多种方式（例如，通过一或多个进程或一组库）实现的抽象。为使应用与实现细节去耦合，CORBA 规范定义了 ORB 的接口。该 ORB 接口提供标准的操作来初始化和关闭 ORB、将对象引用转换为字符串或反向转换，以及为通过*动态调用接口*（dynamic invocation

interface，DII）发送的请求创建参数表。

OML IDL Stub 和 Skeleton：IDL Stub 和 Skeleton 分别被用作客户及仆人与 ORB 之间的“ 胶水 ”。Stub 实现*代理*（Proxy）模式[15]，并提供强类型的*静态调用接口*（SII）；该接口将应用参数整编（marshal）为通用的消息级表示。相反，Skeleton 实现*适配器*（Adapter）模式[15]，并将消息级表示“ 去整编 ”为对应用来说有意义的有类型参数。

IDL 编译器（IDL Compiler）：IDL 编译器将 OMG IDL 定义转换为自动生成的 Stub 和 Skeleton；其源码采用某种应用编程语言，比如 C++或 Java。除了提供编程语言透明性，IDL 编译器还消除了常见的网络编程错误来源，并可以进行自动的编译器优化[17]。

动态调用接口（DII）：DII 允许用户在运行时生成请求，对于不具有所访问接口的编译时知识的应用来说这是有用的。DII 还允许客户发出*缓召的同步调用*（deferred synchronous call），使请求与两路（two-way）操作的响应部分去耦合，从而避免客户会阻塞到仆人响应为止。CORBA SII Stub 同时支持同步和异步*两路*操作，也就是，请求/响应操作；以及*单路*操作，也就是，“ 只请求 ”操作。

动态 Skeleton 接口（DSI）：DSI 是客户的 DII 在服务器端的对应物。DSI 允许 ORB 将请求递送给仆人，而这些仆人不具有所实现的 IDL 接口的编译时知识。发出请求的客户不需要知道服务器 ORB 是使用静态 Skeleton 还是动态 Skeleton。同样地，服务器也不需要知道客户是使用 DII 还是 SII 来调用请求的。

对象适配器（Object Adapter）：对象适配器是使仆人与对象相关联的合成组件，它创建对象引用、将到来的请求多路分离给仆人，并与 IDL Skeleton 协作，以分派仆人上适当的操作 upcall。对象适配器使得 ORB 能够支持各种类型的具有类似需求的仆人。这样的设计产生了更小更简单的 ORB，可以支持广泛的对象粒度、生存期、策略、实现风格和其他特性。

接口仓库（Interface Repository）：接口仓库提供关于 IDL 接口的运行时信息。使用这些信息，程序有可能遇到在程序编译时其接口还属未知的对象，并且仍然能够确定在该对象上的合法操作，并使用 DII 来发出调用。此外，接口仓库还提供公共区域来存储与 CORBA 对象的接口相关联的其他信息，比如 Stub 和 Skeleton 的类型库。

实现仓库（Implementation Repository）：实现仓库[18]包含的信息允许 ORB 启用服务器来对仆人进行处理。实现仓库中的大多数信息都是特定于某种 ORB 或 OS 环境的。此外，实现仓库还提供了公共区域来存储与服务器相关联的信息，比如管理控制、资源分配、安全，以及启用模式。

3.2.2 TAO 综述

TAO 是高性能实时 ORB 终端系统，其应用目标是具有确定性和可统计 QoS 需求、以及最优努力（best-effort）需求的应用。TAO 的 ORB 终端系统含有图 3-2 中所示的网络接口、OS、通信协议，以及遵循 CORBA 的中间件组件和服务。TAO 支持标准的 CORBA 参考模型[14]和实时 CORBA 规范[19]，并且，它还具有一些增强特性，可用于确保高性能及实时应用的高效、可预测和可伸缩的 QoS 行为。此外，TAO 也可以很好地适用于一般的分布式应用。下面，我们概述图 3-2 中所示的 TAO 组件的特性。

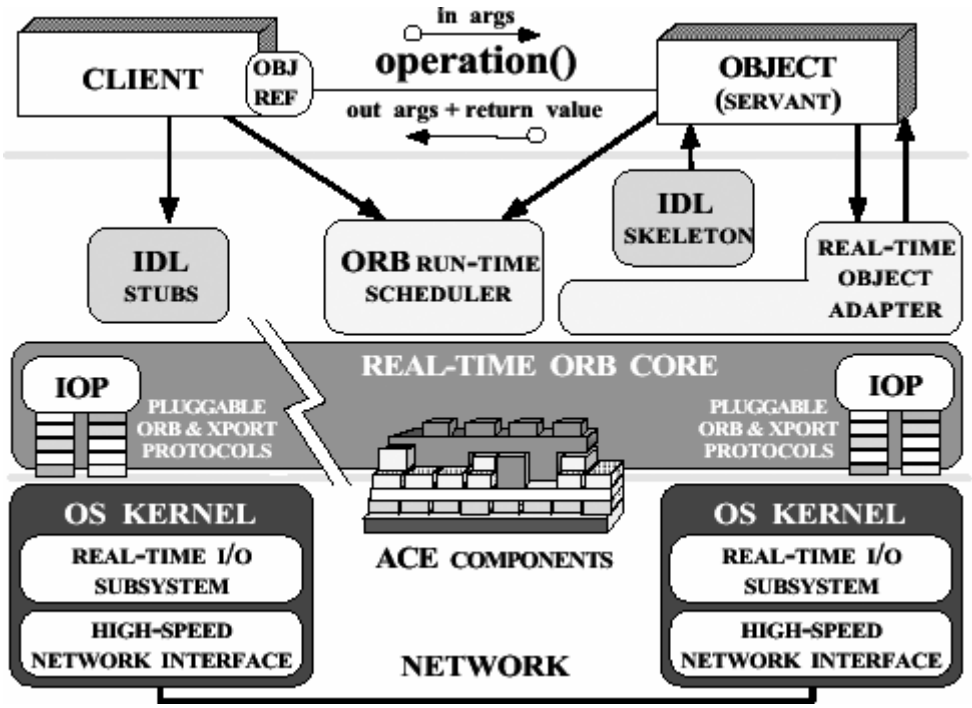

图 3-2 TAO 实时 ORB 终端系统中的组件

优化的 IDL Stub 和 Skeleton：IDL Stub 和 Skeleton 分别执行应用操作参数的整编和去整编。TAO 的 IDL 编译器生成的 Stub/Skeleton 可以有选择地使用高度优化编译的和/或解释式的（去）整编[20]。这样的灵活性允许应用开发者有选择地对时间和空间需求进行权衡，这对于高性能、实时、和/或嵌入式分布系统来说是至关紧要的。

实时对象适配器（Real-time Object Adapter）：对象适配器使仆人与 ORB 相关联，并将到来的请求多路分离给仆人。TAO 的实时对象适配器使用理想哈希[21]和主动式多路分离[22]优化来在恒定的 O（1）时间内分派仆人操作，而不管 IDL 接口中定义的活动连接、仆人及操作的数目。

运行时调度器（Run-time Scheduler）：TAO 的运行时调度器[19]将应用 QoS 需求（比如限定端到端响应时间以及满足定时调度最后期限）映射到 ORB 终端/网络资源（比如 CPU、内存、网络连接，以及存储设备）。TAO 的运行时调度器同时支持静态[10]和动态的[23]实时调度策略。

实时 ORB 核心（Real-time ORB Core）：ORB 核心将客户请求递送给对象适配器，并将响应（如果有的话）返回给客户。TAO 的实时 ORB 核心[24]使用多线程、占先式、基于优先级的连接和并发体系结构[20]来提供高效和可预测的 CORBA 协议引擎。TAO 的 ORB 核心允许将定制的协议插入 ORB，而不会影响标准的 CORBA 应用编程模型[25]。

实时 I/O 子系统（Real-time I/O subsystem）：TAO 的实时 I/O（RIO）子系统[26]把对 CORBA 的支持扩展进 OS 中。RIO 分配优先级给实时 I/O 线程，以实施应用组件和 ORB 终端系统资源的可调度性。在与高级硬件集成时（比如下面描述的高速网络接口），RIO 可以（1）及早在优先级化的内核线程上执行 I/O 事件的多路分离，以避免基于线程的优先级调换，以及（2）维护清晰的优先级流，以避免基于包的优先级调换。TAO 还可以高效地，并尽可能地在缺乏高级 QoS 特性的传统 I/O 子系统上可预测地运行。

高速网络接口（High-speed network interface）：“菊花链”（daisy-chained）网络接口是 TAO 的 I/O 子系统的核心，它含有一或多个 ATM 端口互连控制器（APIC）芯片[27]。APIC 被设计用于维持 2.4 Gbps 的双向合计数据率，使用零复制缓冲优化来避免在终端层之间复制数据。TAO 还运行在传统的实时互连上，比如 VME 底板和多处理器共享内存环境；以及 TCP/IP 上。

TAO 内部情况（TAO internals）：TAO 的开发使用了称为 ACE[28]的较低级中间件。ACE 实现通信软件的核心并发和分布模式[8]，并且提供可复用的 C++包装外观和框架组件来支持高性能实时应用和像 TAO 这样的较高级中间件的 QoS 需求。ACE 和 TAO 运行在广泛的 OS 平台上，包括 Win32、大多数版本的 UNIX，以及实时操作系统，比如 Sun/Chorus ClassiX、LynxOS 和 VxWorks。

为加速实现我们的项目目标，并避免重新发明已有的组件，我们在 SunSoft IIOP 之上对 TAO 进行开发。SunSoft IIOP 是可自由使用的 Internet Inter-ORB Protocol（IIOP）版本 1.0 的 C++参考实现。尽管 SunSoft IIOP 提供了 CORBA ORB 的核心特性，它还是有以下局限：

缺乏标准的 ORB 特性：尽管 SunSoft IIOP 提供了 ORB 核心、IIOP 1.0 协议引擎，以及 DII 和 DSI 实现，它还缺乏 IDL 编译器、接口仓库和实现仓库，以及可移植对象适配器（Portable Object Adapter，POA）。TAO 实现了所有这些遗漏的特性，并提供了更新的 CORBA 特性：异步方法调用[29]、实时 CORBA[19]特性[30]，以及容错 CORBA 特性[31, 32]。

缺乏 IIOP 优化：由于大量的整编/去整编开销、数据复制，以及很高的函数调用开销，SunSoft IIOP 在高速网络上执行得很糟糕。因此，我们应用了一系列优化法则模式[22]，显著地改善了它的性能[33]。指导我们的优化法则包括：（1）为常见情况进行优化，（2）消除不必要的浪费，（3）用高效的专用方法来替换通用方法，（4）如果可能，预先计算值，（5）存储冗余状态，以加速昂贵的操作，（6）在层与层间传递信息，以及（7）为处理器缓存亲缘性进行优化。

本论文并不讨论 TAO 怎样解决上面概述的 SunSoft IIOP 的局限，[10, 20]对它们进行了详细讨论。相反，我们关注 TAO 怎样在使用模式保留它的 QoS 能力的同时、实现克服了以下 SunSoft IIOP 局限的 ORB：

缺乏可移植性：像大多数通信软件应用一样，SunSoft IIOP 是直接使用低级网络和 OS API（比如 socket、select 和 POSIX Pthreads）编写的。这些 API 不仅麻烦而易错，它们还不能跨平台移植，例如，许多操作系统都缺乏 Pthreads 支持。3.3.3.1 演示我们怎样使用*包装外观*（Wrapper Façade）模式[15]来改善 TAO 的可移植性。

缺乏可配置性：像许多 ORB 和其他中间件一样，SunSoft IIOP 是被静态配置的，这使得不直接修改源码、就难以对其进行扩展。这违背了 TAO 的一个关键设计目标：对不同应用需求和系统环境的*动态*适配。3.3.3.7 和 3.3.3.8 解释我们是怎样使用*抽象工厂*（Abstract Factory）[15]、*策略*（Strategy）[15]和*组件配置器*（Component Configurator）[8]模式来为不同的使用情况简化 TAO 的配置的。

缺乏软件内聚性：像许多应用一样，SunSoft IIOP 专门解决特定的问题，也就是，实现 ORB 核心和 IIOP 协议引擎。它使用紧耦合的、特定的、且将关键 ORB 设计决策硬编码的实现来完成这一工作。3.3.3.7 和 3.3.3.6 阐释我们怎样使用*抽象工厂*和*策略*来减少不必要的耦合，并在将 SunSoft IIOP 发展为 TAO 时增加内聚性。

3.3 应用模式语言构建可扩展 ORB 中间件

3.3.1 我们为何需要可动态配置的中间件

ORB 中间件的一个关键动机是将复杂、较低级的分布式系统基础构造任务从应用开发者处转移到 ORB 开发者那里。ORB 开发者负责实现可复用的中间件组件，由它们来处理连接管理、进程间通信、并发、传输端点多路分离、调度、分派、（去）整编，以及错误处理。这些组件通常被编译进运行时 ORB 库中、与使用 ORB 组件的应用对象链接，并在一或多个 OS 进程中执行。

尽管这样的事务分离可以简化应用开发，它还有可能会产生不灵活和低效的应用及中间件体系结构。其主要原因是许多传统 ORB 都是由 *ORB 开发者*在编译时和链接时*静态*配置的，而不是由*应用开发者*在安装时或运行时*动态*配置的。静态配置的 ORB 有以下缺点[28]：

不灵活性：静态配置的 ORB 将各个组件的*实现*与内部 ORB 组件的*配置*紧密地耦合在一起，也就是，哪些组件一起工作以及它们怎样一起工作。结果，对静态配置的 ORB 进行扩展需要修改已有源码。在商业的非开放源码的 ORB 中，应用开发者可能无法获得这些代码。

即使源码可用，扩展静态配置的 ORB 还需要重新编译和链接。而且，任何目前正在执行的 ORB 及其相关对象都必须关闭并重启。这样的静态重配置过程对于某些应用领域并不很适用，比如需要 7 * 24 可用性的电信呼叫处理[34]。

低效率：静态配置的 ORB 在空间和时间两方面都可能是低效的。如果不必要的组件总是被静态地配置进 ORB 中，就可能会产生空间的低效率。这可能会增加 ORB 的内存占用，迫使应用为它们不需要的特性付出空间代价。对于嵌入式系统，比如蜂窝电话或电信中继线卡[35]，过大的内存占用是特别成问题的。

时间低效则可能来自限制 ORB 使用静态配置的算法或数据结构来处理关键任务，从而使得应用开发者难以定制 ORB 来处理新的使用情况。例如，实时航空控制系统[11]常常可以离线地实例化它们的所有仆人。这些系统可以从这样的 ORB 中获益：它使用理想哈希或主动式多路分离[36]来将到来的请求多路分离给仆人。因而，对于关键任务系统来说，被静态配置以使用通用的、“一劳永逸”的多路分离*策略*（比如动态哈希）的 ORB 的执行可能会很糟糕。

理论上，上面描述的静态配置的缺点是*内在于* ORB 的，不应该直接影响应用开发者。但是实际上，应用开发者会不可避免地受到影响，因为 ORB 中间件的质量、可移植性、可用性，以及性能都被降低了。因此，改善 ORB 可扩展性的有效方式是开发可进行静态*和*动态配置的 ORB 中间件。

通过动态配置，开发者可以有选择性地集成关键 ORB 策略（比如连接管理、通信、并发、多路分离、调度，以及分派）的定制实现。这样的设计使得 ORB 开发者能够集中关注 ORB 组件的*功能*，而不用过早地使自己陷入这些组件的的特定*配置*。而且，动态配置还使得应用开发者和 ORB 开发者能够在系统生存期的后期（也就是，在安装时或运行时）改变设计决策。

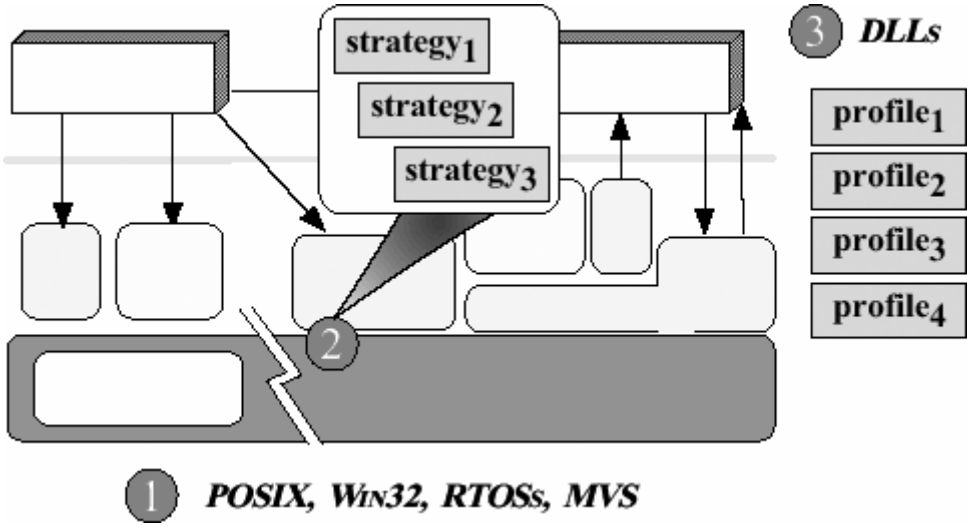

图 3-3 ORB 可扩展性维度

图 3-3 演示以下 ORB 可扩展性的关键维度：

1. **使 ORB 转向新平台：**这需要 ORB 使用模块化组件来实现；这些组件将 ORB 与不可移植的系统机制（比如用于线程、通信和事件多路分离的那些机制）相屏蔽。诸如 POSIX、Win32、VxWorks 和 MVS 这样的 OS 平台提供了多种多样的系统机制。
2. **自定义实现策略：**可以为特定的应用需求进行裁剪。例如，可以定制 ORB 组件来满足实时系统中的最后期限[11]。同样地，可以定制 ORB 组件来适应特定的系统特性，比如异步 I/O 或高速 ATM 网络。
3. **自定义策略的动态配置：**它通过只动态链接特定的 ORB“特质”所必需的那些策略来将定制带到下一层级。例如，不同的应用领域，比如医学系统或电信呼叫处理，可能需要对并发、调度或分派策略的组合进行定制。在运行时从动态链接库（DLL）中链接并配置这些策略可以（1）降低 ORB 的内存占用，以及（2）使应用开发者能够扩展 ORB，而无需访问或改变原来的源码。

下面，我们沿着上面概述的每一维度描述用于增强 TAO 的可扩展性的模式语言。

3.3.2 改善 ORB 可扩展性的模式语言综述

这一部分使用 TAO 作为个案研究，演示通过降低组件间的耦合来帮助应用和 ORB 开发者构建、维护和扩展通信软件的模式语言。图 3-4 显示该模式语言中我们用于为 TAO 开发可扩展 ORB 体系结构的模式。详细地描述每一模式、或讨论 TAO 中使用的所有模式已超出了本论文的范围。相反，我们聚焦于关键模式是怎样改善实时 ORB 中间件的可扩展性和性能的。[9, 15]中的参考文献有对这些模式的全面描述，[8]解释这些模式是怎样被编织在一起来构成模式语言的。

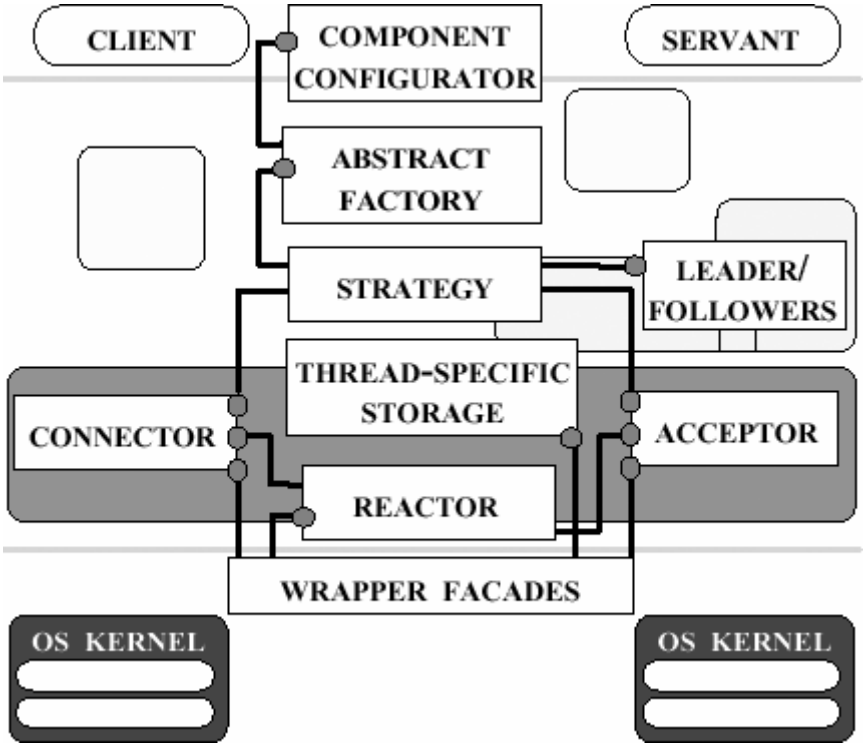

图 3-4 将模式语言应用于 TAO

下面概述这一语言中的模式的意图和使用：

包装外观（Wrapper Façade）[8]：该模式将现有非 OO API 所提供的功能和数据封装在更为简洁、健壮、可移植、可维护和内聚的 OO 类接口中。TAO 使用此模式来避开 OS 专有的系统调用（比如 Socket API 或 POSIX 线程）的麻烦、不可移植，以及非类型安全的编程。

反应器（Reactor）[8]：该模式构造事件驱动的应用、特别是服务器，它们从多个客户那里并发地接收请求，但依次对它们进行处理。当 I/O 事件在 OS 中发生时，TAO 使用此模式来同步地通知 ORB 特有的处理器。反应器模式驱动 TAO 的 ORB 核心中的主事件循环，接受连接并接收/发送客户请求/响应。

接受器－连接器（Acceptor-Connector）[8]：该模式使连接建立及服务初始化与服务处理去耦合。TAO 在服务器和客户上的 ORB 核心中使用此模式，以被动和主动地建立独立于底层传输机制的 GIOP 连接。

领导者/跟随者（Leader/Followers）[8]：该模式提供一种高效的并发模型，在其中多个线程轮流享有一组事件源，以检测、多路分离、分派和处理发生在事件源上的服务请求。TAO 使用此模式来便利多种并发策略的使用；这些策略可在运行时被灵活地配置进 ORB 核心。

线程专有存储（Thread-Specific Storage）[8]：该模式允许多个线程使用“逻辑上全局”的访问点来获取相对于线程来说的局部对象，而又不会为每次访问对象带来锁定开销。TAO 使用此模式来使实时应用的锁竞争和优先级调换最小化。

策略（Strategy）[15]：该模式提供的抽象用于从若干候选算法中进行选择，并将其包装进对象中。在整个软件体系结构中，TAO 大量地使用此模式来为并发、通信、调度和多路分离配置自定义 ORB 策略。

抽象工厂（Abstract Factory）[15]：该模式提供单一组件来构造多个相关对象。TAO 使用此模式来将它的成打的*策略*对象合并进一些可管理的抽象工厂中，这些工厂可一起被方便而一致地重配置进客户和服务器中。TAO 组件使用这些工厂来访问相关策略，而不用明确地指定它们的子类名。

组件配置器（Component Configurator）[8]：该模式允许应用在运行时链接它的组件实现，或解除其链接，而不必修改、重编译或静态重链接此应用。它还支持将组件重配置进不同的进程中，而不必关闭和重启运行中的进程。TAO 使用此模式来动态地替换*抽象工厂*实现，以在运行时定制 ORB 的特性。

组成此模式语言的模式的使用并不局限于 ORB 或通信中间件。它们已被应用于其他许多通信软件领域中，包括电信呼叫处理和交换、航空航班控制系统、多媒体电话会议，以及分布式交互模拟，等等。

3.3.3 怎样使用模式语言来应对 ORB 设计挑战

在下面的讨论中，我们概述在开发可扩展实时 ORB 时，关键的设计挑战所带来的压力。我们还将描述我们的模式语言中的哪些模式消除了这些压力，并解释 TAO 是怎样使用它们的。此外，我们还显示了在 ORB 中，怎样因为这些模式的缺乏而未能消除这些压力。为具体地演示后面这一点，我们将 TAO 和 SunSoft IIOP 作了比较。因为 TAO 发展自 SunSoft IIOP，所以它提供了理想的基准来评估模式对于 ORB 中间件的软件质量的影响。

3.3.3.1 通过包装外观模式封装低级系统机制

上下文：ORB 的一种角色是将应用特有的客户和仆人与低级系统编程的细节相屏蔽。因而，ORB 开发者，而不是应用开发者，负责处理麻烦的低级网络编程任务，比如多路分离事件、通过网络发送和接收 GIOP 消息，以及派生线程来并发地执行客户请求。图 3-5 演示 SunSoft IIOP 使用的一种常用方法，它在内部使用系统机制（比如 socket、select 和 POSIX threads）来进行编程。

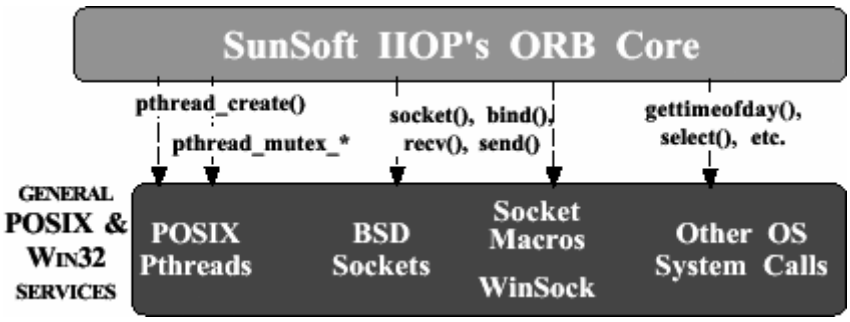

图 3-5 SunSoft IIOP 操作系统接口

问题：开发 ORB 是十分困难的。如果开发者使用低级的系统机制，开发会变得困难；这些机制使用像 C

这样的语言编写，常常会产生以下问题：

- ***ORB 开发者必须熟悉许多 OS 平台：***使用系统级 C API 实现 ORB 迫使开发者去处理不可移植、麻烦和易错的 OS 特性，比如使用无类型的 Socket 句柄来标识传输端点。而且，这些 API 不能跨平台移植。例如，Win32 缺少 POSIX Pthreads，并且它的 socket 和 select 的语义有着微妙的不同。
- ***更多的维护工作：***构建 ORB 的一种方法是通过 ORB 源码中的显式的条件编译指令来处理可移植性变化。但是，如 3.3.5 所显示的，使用条件编译来处理平台特有的变化在*各个方面*都增加了源码的复杂性。开发者很难对这样的 ORB 进行扩展，因为平台特有的细节分散在实现的源码文件的各个地方。
- ***不一致的编程范式：***系统机制是通过 C 风格的函数调用来访问的，这会导致与 C++（我们用以实现 TAO 的语言）支持的 OO 编程风格的“阻抗失配”。

我们怎样才能在实现 ORB 时避免访问低级的系统机制呢？

解决方案—>包装外观模式：使用*包装外观*模式是避免直接访问系统机制的有效方式 [8]，它是外观模式[15]的一种变种。外观模式的意图是简化子系统的接口。包装外观的意图更为具体：它提供类型安全、模块化和可移植的 OO 接口，封装低级的、独立的系统机制，比如 socket，select 和 POSIX 线程。通常，包装外观应在现有系统级 API 不可移植和非类型安全时应用。

在 TAO 中使用包装外观模式：TAO 通过 ACE[28]提供的包装外观来访问所有系统机制。图 3-6 演示 ACE C++包装外观怎样通过使用类型安全的 OO 接口来封装和增强本地 OS 的并发、通信、内存管理、事件多路分离和动态链接机制，改善 TAO 的健壮性和可移植性。ACE 提供的 OO 封装减少了 TAO 对直接访问弱类型系统 API 的需要。因而，C++编译器可以在编译时、而不是等到问题在运行时发生的时候，检测类型系统违例。

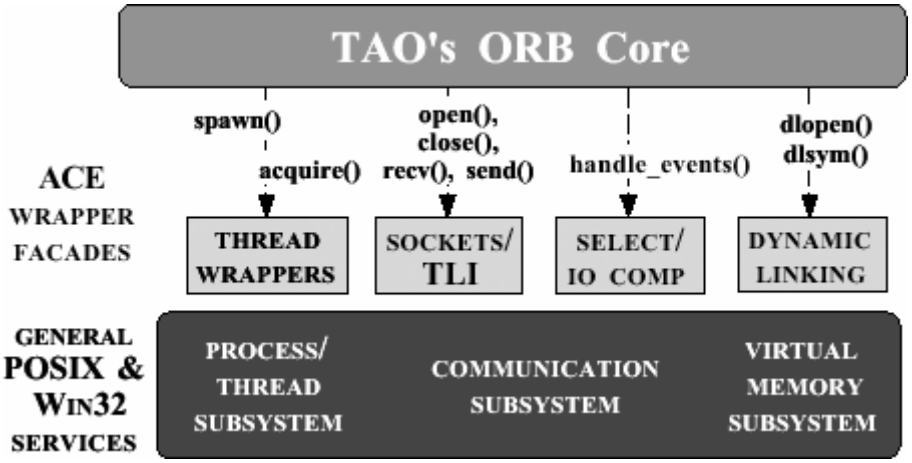

图 3-6 将包装外观模式用于封装本地 OS 机制

ACE 包装外观使用 C++特性来消除额外的类型安全性和抽象层次所带来的性能开销。例如，内联被用于消除调用小方法的额外开销。同样地，静态方法被用于消除每次调用都传递 C++ this 指针的开销。

尽管 ACE 包装外观解决了若干常见的低级开发问题，它们还仅仅是迈向开发可扩展 ORB 的第一步。这一部分描述的其他模式在 ACE 包装外观提供的封装之上构建，以致力于更有挑战性的 ORB 设计问题。

3.3.3.2 使用反应器模式多路分离 ORB 核心事件

上下文：ORB 核心负责对来自多个客户的 I/O 事件进行多路分离，并分派与它们相关联的事件处理器。例如，服务器端的 ORB 核心侦听新客户连接，并从相连的客户那里读取 GIOP 请求、或将 GIOP 响应写往相连的客户。为确保有多个客户时的响应性，ORB 核心使用 OS 事件多路分离机制来等待连接、读取或写入事件在多个 Socket 句柄上发生。常见的事件多路分离机制包括 select、WaitForMultipleObjects、I/O 完成端口，以及线程。

图 3-7 演示 SunSoft IIOP 的典型的事件多路分离序列。在（1）中，服务器通过（2）调用对象适配器的 get_request、进入它的事件循环。get_request 随即（3）调用 server_endpoint 的静态方法 block_for_connection。该方法管理服务器端连接管理的所有方面，范围从连接建立到 GIOP 协议处理。ORB 阻塞（4）在 select 上，直到 I/O 事件（比如连接事件或请求事件）发生。当请求事件发生时，block_for_connection 多路分离该请求给特定的 server_endpoint，并（5）分派该事件给此端点。随后 ORB 核心中的 GIOP 引擎（6）从 Socket 获取数据，并将其传递给对象适配器，后者将其多路分离、整编，并（7）分派适当的方法 upcall 给用户提供的仆人。

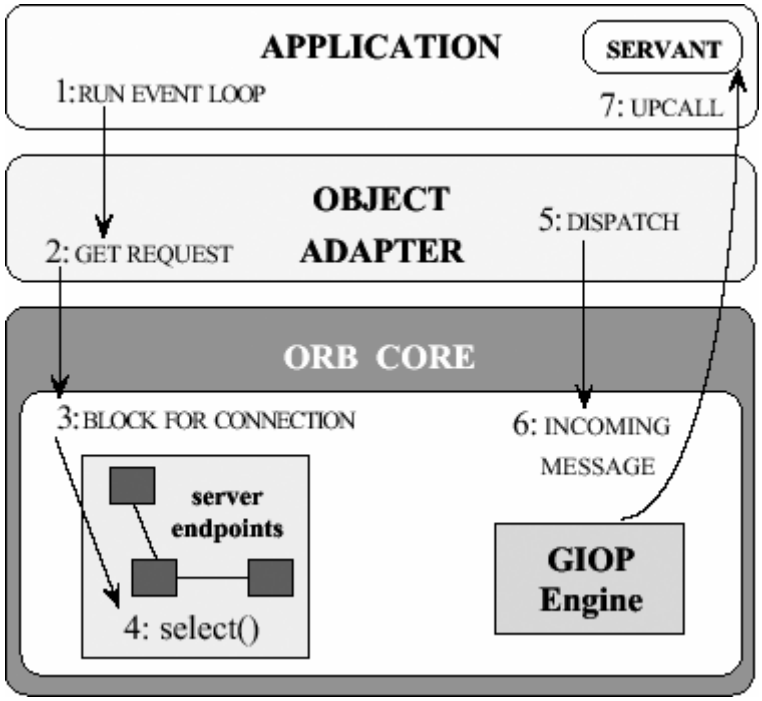

图 3-7 SunSoft IIOP 事件循环

问题：开发 ORB 核心的一种方式是将其硬编码为使用某种事件多路分离机制，比如 select。但是，仅依赖一种机制是不合需要的，因为没有哪种方案在所有平台上都是高效的，或能满足所有的应用需求。例如，异步 I/O 完成端口在 Windows NT 上是异常高效的[37]，而同步线程在 Solaris 上是高效的多路分离机制。

开发 ORB 核心的另一种方式是将它的事件多路分离代码与执行 GIOP 协议处理的代码紧密地耦合在一起。例如，SunSoft IIOP 的事件多路分离逻辑不是一种自包含的组件。相反，它与后续的由对象适配器和 IDL Skeleton 进行的客户请求事件的处理紧密地缠绕在一起。但是，在这种情况下，多路分离代码不

能被其他通信中间件应用（比如 HTTP 服务器[37]或视频点播服务器）当作“ 黑盒子 ”组件来复用。而且，如果引入了新的 ORB 线程策略或对象适配器请求调度算法，就必须重写 ORB 核心的相当一部分。

那么 ORB 实现怎样使自身与特定的事件多路分离机制去耦合，并使它的多路分离代码与它的处理代码去耦合呢？

解决方案—>反应器模式：应用*反应器*模式是降低耦合并增加 ORB 核心的可扩展性的有效方式 [8]。该模式支持同步的多路分离和多个*事件处理器*的分派，这些处理器由可并发到达的来自多个来源的事件触发。反应器模式通过集成事件的多路分离和相应的事件处理器的分派，简化了事件驱动的应用。一般而言，反应器模式应该在下面的情况下使用：应用或组件（比如 ORB 核心）必须处理来自多个客户的并发事件、而又不能与单一的低级机制（比如 select）紧密地耦合在一起。

注意包装外观的应用并不足以解决上面概述的事件多路分离问题。Select 的包装外观可以略微改善 ORB 核心的可移植性。但是，仅仅是这个模式并不能消除这样的需要：完全使低级的事件多路分离逻辑与 ORB 核心中较高级的客户请求处理去耦合。认清包装外观的局限，并随之应用反应器模式来克服此局限，是对模式语言、而不仅是独立的模式进行应用的好处之一。

在 TAO 中使用反应器模式：如图 3-8 所示，TAO 使用反应器模式来在它的 ORB 核心中驱动主事件循环。TAO 服务器（1）在 ORB 核心的 *Reactor* 中发起事件循环，在这里它（2）阻塞在 select 上，直到有 I/O 事件发生。当 GIOP 请求事件发生时，Reactor 将请求多路分离给适当的事件处理器；此处理器是与每个相连 Socket 关联的 GIOP Connection_Handler。随后 Reactor（3）调用 Connection_Handler::handle_input，它（4）将请求分派给 TAO 的对象适配器。对象适配器将请求多路分离给适当的仆人的 upcall 方法，并且（5）分派此 upcall。

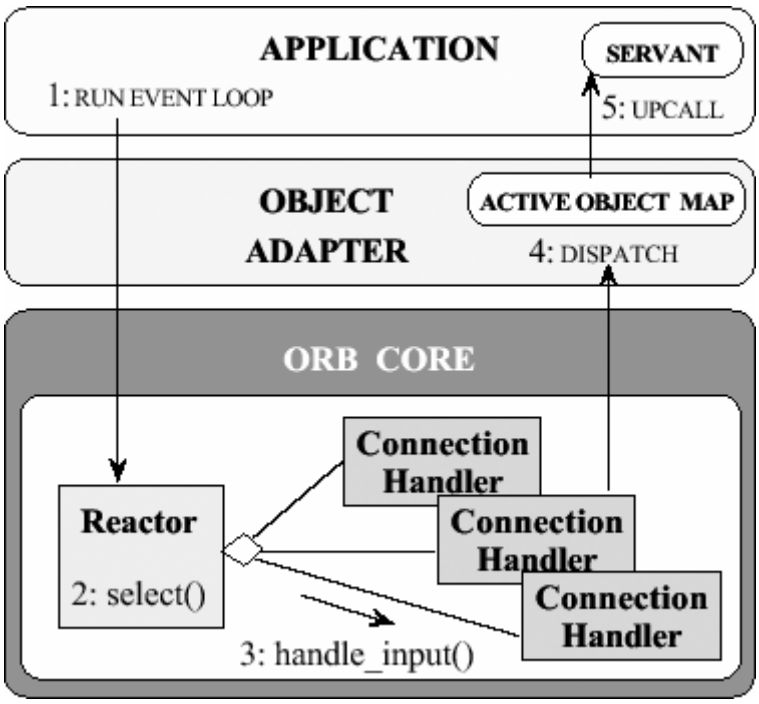

图 3-8 在 TAO 的事件循环中使用反应器模式

通过使 TAO 的 ORB 核心的事件处理部分与底层的 OS 事件多路分离机制去耦合，反应器模式增强了 TAO 的可扩展性。例如，WaitForMultipleObjects 事件多路分离系统调用可被用在 Windows NT 上，而

select 可被用在 UNIX 平台上。而且，反应器模式简化了新事件处理器的配置。例如，增加新的 Secure_Connection_Handler、来执行所有网络通信的加密/解密将不会影响 Reactor 的实现。最后，与 SunSoft IIOP 中的事件多路分离代码（它与一种使用情况紧耦合在一起）不同，TAO 所用的反应器模式[8]的 ACE 实现已被应用于其他许多 OO 事件驱动的应用中，范围从 HTTP 服务器[37]到实时航空系统的基础结构[11]。

3.3.3.3 使用接受器－连接器模式在 ORB 中管理连接

上下文：管理连接是 ORB 核心的另一项关键责任。例如，实现 IIOP 协议的 ORB 核心必须建立 TCP 连接，并为每个 IIOP server_endpoint 初始化协议处理器。通过使连接管理逻辑局限在 ORB 核心中，应用特有的仆人可以只聚焦于处理客户请求，而不用处理低级的网络编程任务。

但是 ORB 核心并不只*局限于*在 IIOP 和 TCP 传输之上运行。例如，在 TCP 可以可靠地传输 GIOP 请求的同时，它的流控制和拥塞控制算法可能会阻碍它被作为实时协议来使用[10]。同样地，当客户和仆人共驻于同一终端系统时，使用共享内存传输机制可能更为高效。因而，ORB 核心应该灵活到能支持多种传输机制[16]。

问题：CORBA 体系结构明确地使（1）ORB 核心执行的连接管理任务与（2）应用特有的仆人所执行的请求处理去耦合。但是，实现 ORB 的内部连接管理活动的常见方法是使用低级的网络 API，比如 socket。同样地，ORB 连接建立协议常常与它的通信协议紧密地耦合在一起。

例如，图 3-9 演示 SunSoft IIOP 的连接管理结构。SunSoft IIOP 的客户端实现了一种硬编码的连接缓存*策略*，它使用了 client_endpoint 对象的链表。如图 3-9 所示，无论何时（1）client_endpoint::lookup 被调用，都会遍历该链表，以找到一个未用的端点。如果在缓存中没有到服务器的未用端点，一个新的连接（2）被发起；否则现有连接就会被复用。同样地，服务器端使用 server_endpoint 对象的链表来生成（3）select 事件多路分离机制所需的读/写位掩码。该链表维护的被动传输端点（4）接受连接并（5）接收来自与服务器相连的客户的请求。

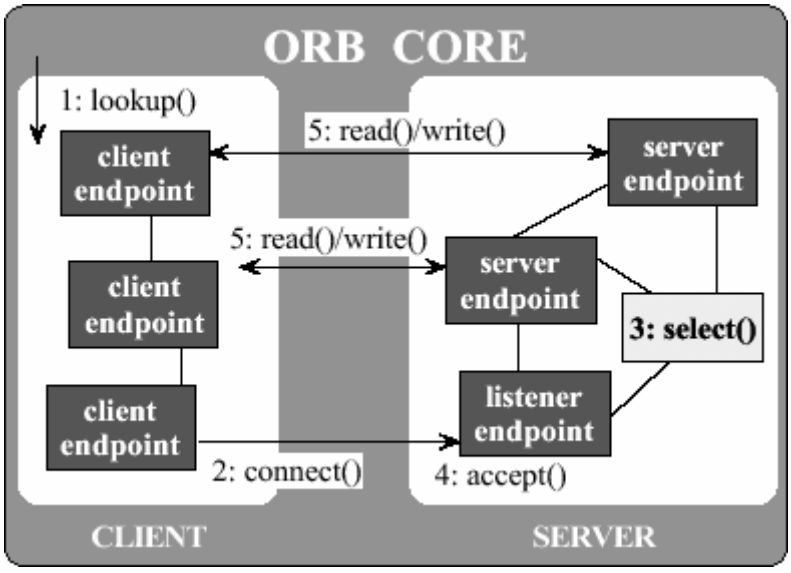

图 3-9 SunSoft IIOP 中的连接管理

SunSoft IIOP 的设计的问题是它紧密地耦合（1）使用 Socket 网络编程 API 的 ORB 的连接管理实现与（2）使用 GIOP 通信协议的 TCP/IP 连接建立协议，从而产生了以下缺点：

- **不灵活性：**如果 ORB 的连接管理数据结构和算法过于紧密地缠绕在一起，修改 ORB 核心就需要相当的努力。例如，ORB 与 Socket API 的紧密耦合使得开发者难以改变底层的传输机制（例如要使用共享内存而不是 Socket）。因而，可能很难将这样的紧耦合 ORB 核心移植到新的通信机制，比如 ATM、光纤信道，或共享内存；或不同的网络编程 API，比如 TLI 或 Win32 命名管道。
- **低效率：**通过允许 ORB 开发者和应用开发者在软件开发周期的后期选择适当的实现（例如，在系统地剖析性能之后），许多内部的 ORB 策略都可被优化。例如，为减少锁竞争和开销，多线程实时 ORB 客户可能需要在线程专有存储[8]中存储传输端点。类似地，CORBA 服务器的并发策略可能需要每个连接运行在它自己的线程中，以消除每次请求的锁定开销。但是，如果连接管理机制是硬编码的，并与其他的内部 ORB 策略紧密地绑定，系统就难以接纳新的策略。

那么 ORB 核心的连接管理组件怎样才能支持多种传输，并允许在开发周期的后期灵活地（重）配置与连接有关的行为呢？

解决方案—>接受器－连接器模式：应用*接受器－连接器*模式是增加 ORB 核心连接管理和初始化的灵活性的有效方式 [8]。该模式使连接初始化与连接端点被初始化后所执行的处理去耦合。其中的 Acceptor 组件，也就是，ORB 核心的服务器端，负责*被动的*初始化。相反，其中的 Connector 组件，也就是，ORB 核心的客户端，负责*主动的*初始化。一般而言，接受器－连接器模式应该在这样的情况下应用：客户/服务器中间件必须允许灵活地配置网络编程 API、且必须维护初始化角色的适当分离。

在 TAO 中使用接受器－连接器模式：TAO 联合使用接受器－连接器模式与反应器模式来处理 GIOP/IIOP 通信的连接建立。在 TAO 的客户端 ORB 核心中，Connector 发起到服务器的连接，以响应操作调用或到远地对象的显式绑定。在 TAO 的服务器端 ORB 核心中，Acceptor 创建 GIOP Connection Handler 来服务每个新的客户连接。Acceptor 和 Conntion_Handler 都派生自 Event_Handler，从而使得它们可以被 Reactor 自动分派。

TAO 的 Acceptor 和 Connector 可通过任意的传输机制来配置，比如 ACE 包装外观所提供的 Socket 或 TLI。此外，如 3.3.3.4 所描述的，TAO 的 Acceptor 和 Connector 可以通过自定义的策略来参数化，以选择适当的并发机制。

图 3-10 演示*接受器－连接器*模式在 TAO 的 ORB 核心中的使用。当客户（1）调用远地操作时，它通过 Strategy_Connector 发出 connect 调用。这个 Strategy_Connector（2）查询它的 *connection strategy*，以获取一个连接。在此例中，客户使用 “ 缓存式连接策略 ”、重复利用到服务器的连接，并只在现有连接全忙时创建新连接。这样的缓存策略使连接设置时间最小化，从而减少了端到端的请求响应时间。

在服务器端 ORB 核心中，Reactor 通知 TAO 的 Strategy_Acceptor（3）接受新连接的客户，并创建 Connection_Handler。Strategy_Acceptor 将并发机制的选择委托给 TAO 的*并发策略*中的一种，例如，在 3.3.3.4 描述的反应式、thread-per-connection，或 thread-per-priority 策略。在 Connetion_Handler 在 ORB 核心中被启用后，（4）它在连接上执行必要的 GIOP 协议处理，（5）并最终通过 TAO 的对象适配器分派（6）请求给适当的的仆人。

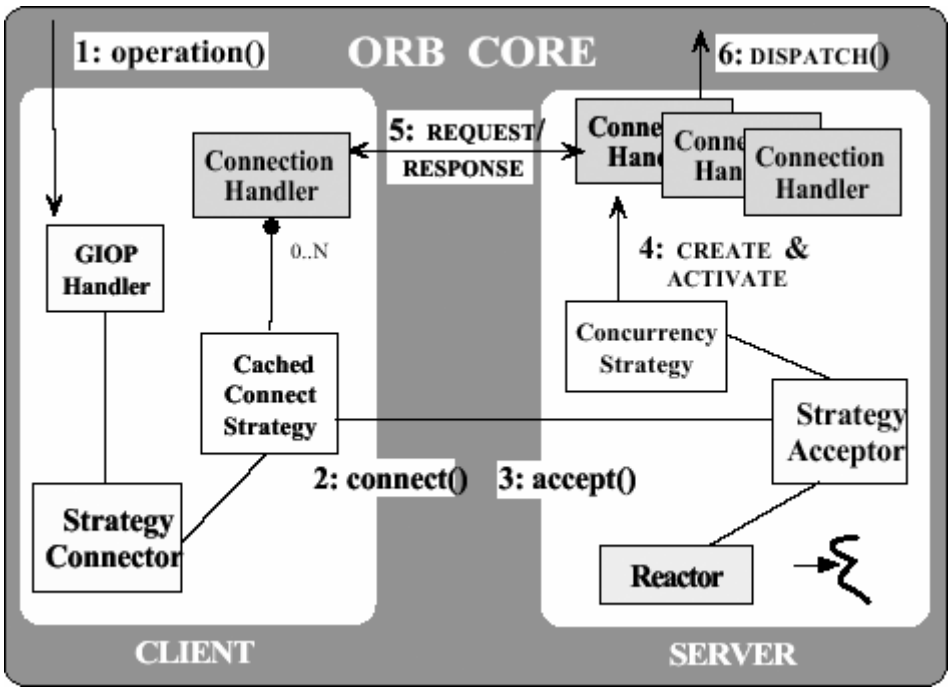

图 3-10 在 TAO 的连接管理中使用接受器－连接器模式

3.3.3.4 使用领导者/跟随者模式简化 ORB 并发

上下文：在对象适配器将客户请求分派给适当的仆人后，仆人就执行该请求。执行可能发生在与接收它的 Connection_Handler 相同的线程控制中，相反，也可能发生在不同的线程中，与其他请求并发执行。

实时 CORBA 规范[38]定义了一种线程池 API。此外，CORBA 规范还为应用定义了 POA 接口，以指定由单个线程、或使用 ORB 内部的多线程策略来处理所有请求。要满足应用的 QoS 需求，开发高效地实现这些不同的并发 API 的 ORB 是很重要的[24]。并发允许长时间运行的操作同时执行，而不会妨碍其他操作的进行。同样地，要使实时系统的分派响应时间最小化，占先式多线程的使用至关紧要[11]。

并发常常通过 OS 平台上的多线程能力来实现。例如，SunSoft IIOP 支持图 3-11 中所示的两种并发体系结构：单线程反应式体系结构和 thread-per-connection 体系结构。SunSoft IIOP 的反应式并发体系结构在单线程中使用 select 来将每个到达的请求分派给单独的 server_endpoint 对象，后者随即从适当的 OS 内核队列中读取请求。在（1）中，请求到达，并由 OS 进行排队。然后，select（2）通知相关联的 server_endpoint，有等待中的请求。Server_endpoint 最后（3）从队列中读取请求并进行处理。

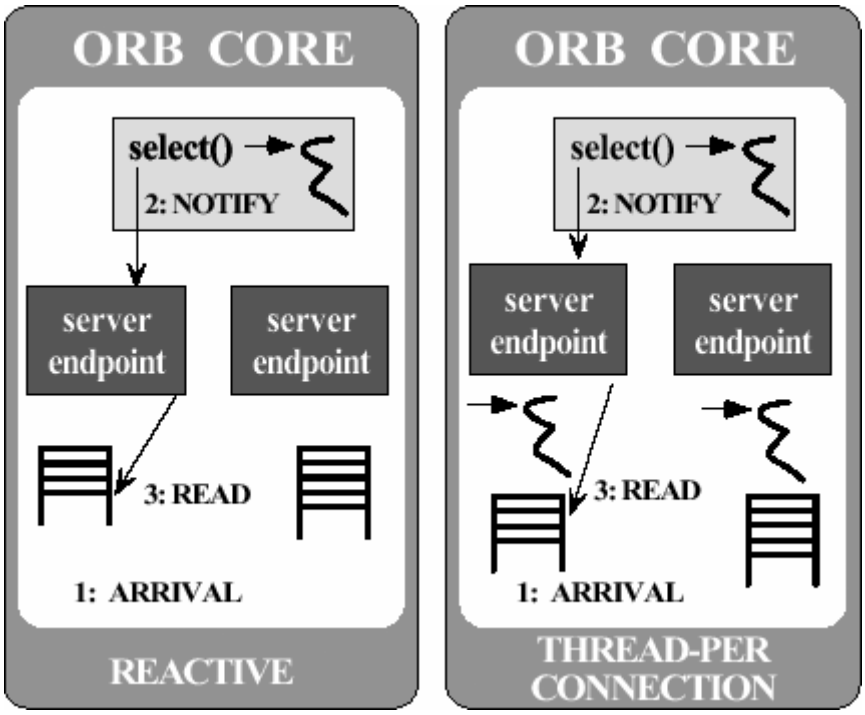

图 3-11 SunSoft IIOP 并发体系结构

相反，SunSoft IIOP 的 thread-per-connection 体系结构在它自己的线程控制中执行每个 server_endpoint，为所有在它的线程内的连接上到达的请求进行服务。在连接建立后，select 在连接的描述符上等待事件。当（1）OS 接收到请求时，线程执行 select（2）从队列中读取一个请求，并（3）将它传给 server_endpoint 进行处理。

问题：在许多 ORB 中，并发体系结构是直接使用 OS 平台的多线程 API 来编程的，比如 POSIX 线程 API[39]。但是，这种方法有若干缺点：

- ***不可移植***：线程 API 是与平台高度相关的。即使是 IEEE 标准，比如 POSIX 线程[39]，在许多被广泛使用的 OS 平台上也不可用，包括 Win32、VsWorks 和 pSoS。在 API 间不仅没有直接的语法映射，甚至连清晰的语义映射也没有。例如，POSIX 线程支持延缓的线程取消，而 Win32 线程不支持。而且，尽管 Win32 有线程终止 API，Win32 文档强烈建议*不要*使用它，因为它在线程退出后并没有释放所有线程资源。再者，甚至于 POSIX 线程实现也是不可移植的，因为许多 UNIX 供应商支持不同的 Pthreads 规范草案。
- ***难以正确编程***：除了可移植性，编写多线程 ORB 也是困难的，因为应用和 ORB 开发者必须确保对共享数据的访问在 ORB 和它的仆人中被适当地序列化。此外，健壮地终止并发地运行在多线程中的仆人所需的技术是复杂、不可移植和不直观的。
- ***不可扩展***：ORB 并发策略的选择极大地依赖于外部因素，像应用需求和网络/终端系统特性。例如，对于单处理器上的短持续时间、局限于计算的请求来说，反应式单线程[8]是一种适当的策略。但是，如果这些外部因素发生变化，ORB 的设计应该足够地可扩展，以处理替换的并发策略，比如线程池或 thread-per-priority[24]。

在使用低级线程 API 来开发 ORB 时，它们难以通过新的并发策略来进行扩展，而又*不*影响其他 ORB 组件。例如，给 SunSoft IIOP 增加 thread-per-request 体系结构需要进行大量改动，以（1）在协议处理过

程中将请求存储在*线程专有存储*（TSS）变量里，（2）将专有钥通过对象适配器中的调度和去整编步骤传递给 TSS 变量，以及（3）在分派仆人上的操作之前、访问存储在 TSS 中的请求。因而，并没有一种容易的方法来修改 SunSoft IIOP 的并发体系结构，而不彻底改变它的内部结构。

那么 ORB 怎样才能支持简单、可扩展和可移植的并发机制呢？

解决方案—>领导者/跟随者模式：应用*领导者/跟随者*模式是增加 ORB 并发策略的可移植性、正确性和可扩展性的有效方式 [8]。该模式提供一种高效的并发模型，在其中多个线程依次享有一组事件源，以检测、多路分离、分派和处理发生在事件源上的服务请求。通常，领导者/跟随者模式应该在应用需要使上下文切换、同步和数据复制开销最小化、同时还允许多个线程并发运行时使用。

在*包装外观*提供了可移植性的基础的同时，它们还只是在低级本地 OS API 之上的一层语法“ 薄板 ”。而且，外观的语义行为在多个平台间仍有可能会变化。因此，领导者/跟随者模式定义了一种更高级的并发抽象，把 TAO 与低级线程外观的复杂性屏蔽开来。通过为 ORB 开发者提高抽象层次，领导者/跟随者模式使得定义更为可移植、灵活，以及方便地编程的 ORB 并发策略变得更为容易。例如，如果在线程池中的线程数为 1，领导者/跟随者模式就会像反应器模式一样工作。

在 TAO 中使用领导者/跟随者模式：TAO 使用领导者/跟随者模式来将 GIOP 事件多路分离给在线程池中的 Connection_Handler。在使用此模式时，应用预先派生*固定*数目的线程。当这些线程调用 TAO 的标准的 ORB::run 方法时，一个线程将成为领导者，并等待 GIOP 事件。在领导者线程检测到事件后，它将一个任意的线程提升为下一个领导者，随后将事件多路分离给与其相关联的 Connection_Handler，由后者相对于 ORB 中的其他线程并发地处理此事件。这一系列步骤在图 3-12 中显示。

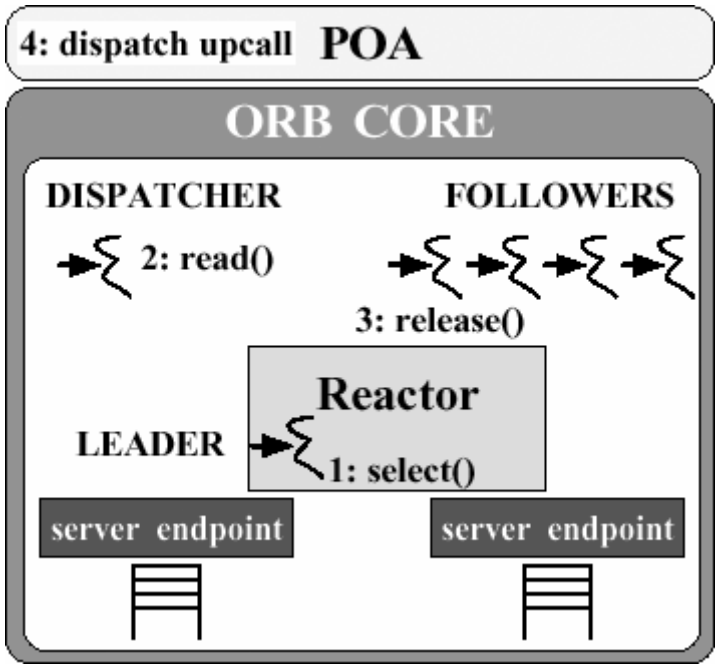

图 3-12 使用领导者/跟随者模式构造 TAO 的并发策略

如图 3-12 所示，程序分配线程池，选择领导者线程，以（1）为所有服务器进程中的仆人而在连接上进行 select。当请求到达时，领导者线程（2）将其读入内部缓冲区。如果它是对仆人的合法请求，在池中的一个跟随者线程就被释放，成为新的领导者（3）且该领导者线程分派 upcall（4）。在 upcall 被分派后，原来的领导者变成跟随者，并返回线程池。新的请求被排队在 Socket 端点中，直到池中有线程可用

来执行请求。

3.3.3.5 通过线程专有存储模式减少锁竞争和优先级调换

上下文：领导者/跟随者模式允许 ORB 中的应用和组件并发地运行。但是，并发的主要缺点是需要*序列化*对共享资源的访问。在 ORB 中，常见的共享资源包括动态内存堆、CORBA::ORB_init 初始化工厂所创建的 ORB 伪对象引用、POA 中的*主动对象*映射[22]，以及先前描述的 Acceptor、Connector 和 Reactor 组件。完成序列化的常用方法是在被多个线程共享的各个资源上使用互斥锁。

问题：在理论上，通过同时执行多个指令流，使 ORB 多线程化可以改善性能。此外，通过允许每个线程同步地、而不是反应式地或异步地执行，多线程还可以简化 ORB 的内部设计。但是，在实践中，多线程 ORB 常常并不比单线程 ORB 执行得更好，甚或会更糟，原因是（1）获取/释放锁的代价，以及（2）当高优先级和低优先级线程竞争同样的锁时所产生的优先级调换[40]。此外，由于用于避免竞争状态和死锁的复杂的并发控制协议，开发者还难以对多线程 ORB 进行编程。

解决方案—>线程专有存储模式：使用*线程专有存储*模式是使“序列化对 ORB 中共享资源的访问”所需的锁定数量最小化的有效方式 [8]。该模式允许 ORB 中的多个线程使用一个逻辑上全局的访问点来获取线程专有的数据，而*不会*给每次访问带来锁定开销。

通常，线程专有存储应在下述情况下被应用：必须通过全局可见的访问点来访问由在各个线程内的对象共享的数据；此访问点在“逻辑上”与其他线程共享，但对于每个线程来说在“物理上”却是唯一的。

在 TAO 中使用线程专有存储模式：TAO 使用线程专有存储模式来最小化实时应用的锁竞争和优先级调换。TAO 中的各个线程在内部使用线程专有存储来存储它的 ORB 核心组件，例如，Reactor、Acceptor 和 Connector。如图 3-13 所示，当线程访问这些组件中的任何一个时，线程通过使用专有钥来作为线程内部的线程专有状态的索引而获取该组件。因而，访问线程专有 ORB 状态不需要额外的锁定。

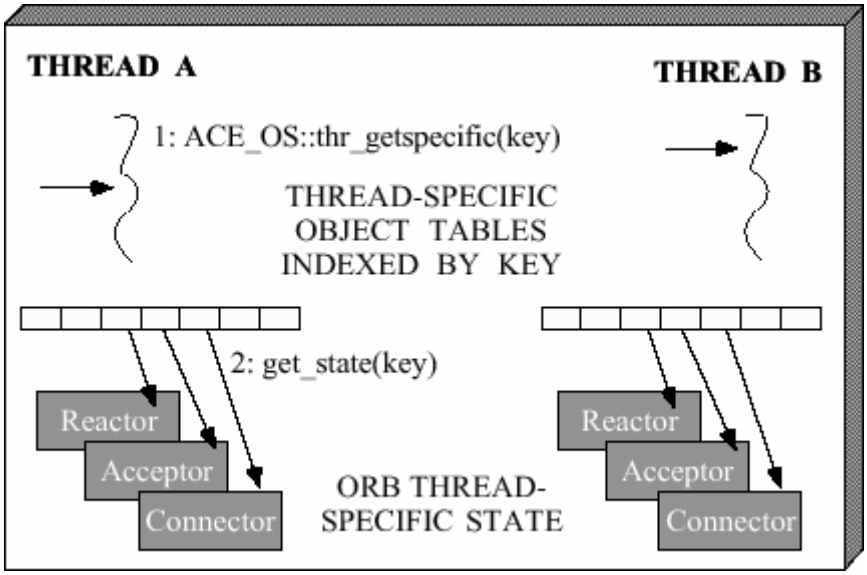

图 3-13 在 TAO 中使用线程专有存储模式

3.3.3.6 通过策略模式支持可互换的 ORB 行为

上下文：可扩展 ORB 必须在它们的对象适配器中支持多种请求多路分离和调度策略。同样地，它们必须在它们的 ORB 核心里支持多种连接建立、请求传递，以及并发请求处理策略。

问题：开发 ORB 的一种方式是只提供静态的、不可扩展的策略，它们典型地通过以下方式来配置：

- ***预处理器宏*：**某些策略是由预处理器宏的值来决定的。例如，因为线程并不是在所有 OS 平台上都可用，常常会使用条件编译来选择一种可行的并发模型。
- ***命令行选项*：**其他的一些策略是由命令行上标志的有无来控制的。例如，命令行选项可用于有选择地为支持多线程的平台而启用某些 ORB 并发策略。[24]。

虽然这两种配置方法被广泛地使用，它们仍然是不灵活的。例如，预处理器宏仅支持编译时策略选择，而命令行选项只传达了有限数量的信息给 ORB。而且，这些硬编码的配置策略完全脱离了任何它们可能会影响的代码。因而，想要使用这些选项的 ORB 组件必须（1）知道它们的存在，（2）了解值的范围，以及（3）为每个值提供适当的实现。这样的限制使其难以开发高度可扩展的、通过可透明地配置的策略编写的 ORB。

那么 ORB 怎样（1）允许通过与其他 ORB 组件无关和透明的方式来替换组件策略的子集，以及（2）封装每种策略的状态和行为来使对一个组件的变动不会偶然地影响整个 ORB？

解决方案—>策略模式：应用*策略*模式是支持多种可透明地“插用”（pluggable）的 ORB 策略的有效方法 [15]。该模式分解出算法可选方案间的相似性，并显式地将策略名与它的算法和状态关联起来。而且，策略模式还去掉了对策略实现的词法依赖，因为应用仅仅通过公共的基类接口来访问专门化的行为。通常，策略模式应该在应用的行为可通过多种可互换策略进行配置时使用。

在 TAO 中使用策略模式：TAO 使用多种策略来分解出常常被硬编码进传统 ORB 的行为。图 3-14 中演示了若干这样的策略。例如，TAO 在它的对象适配器中支持多种请求多路分离策略（例如，理想哈希 vs. 主动式多路分离[36]），并在它的 ORB 核心中支持多种连接管理策略（例如，进程级缓存式连接 vs. 线程专有缓存式连接）和处理器并发策略（例如，反应式 vs. 领导者/跟随者的变种）。

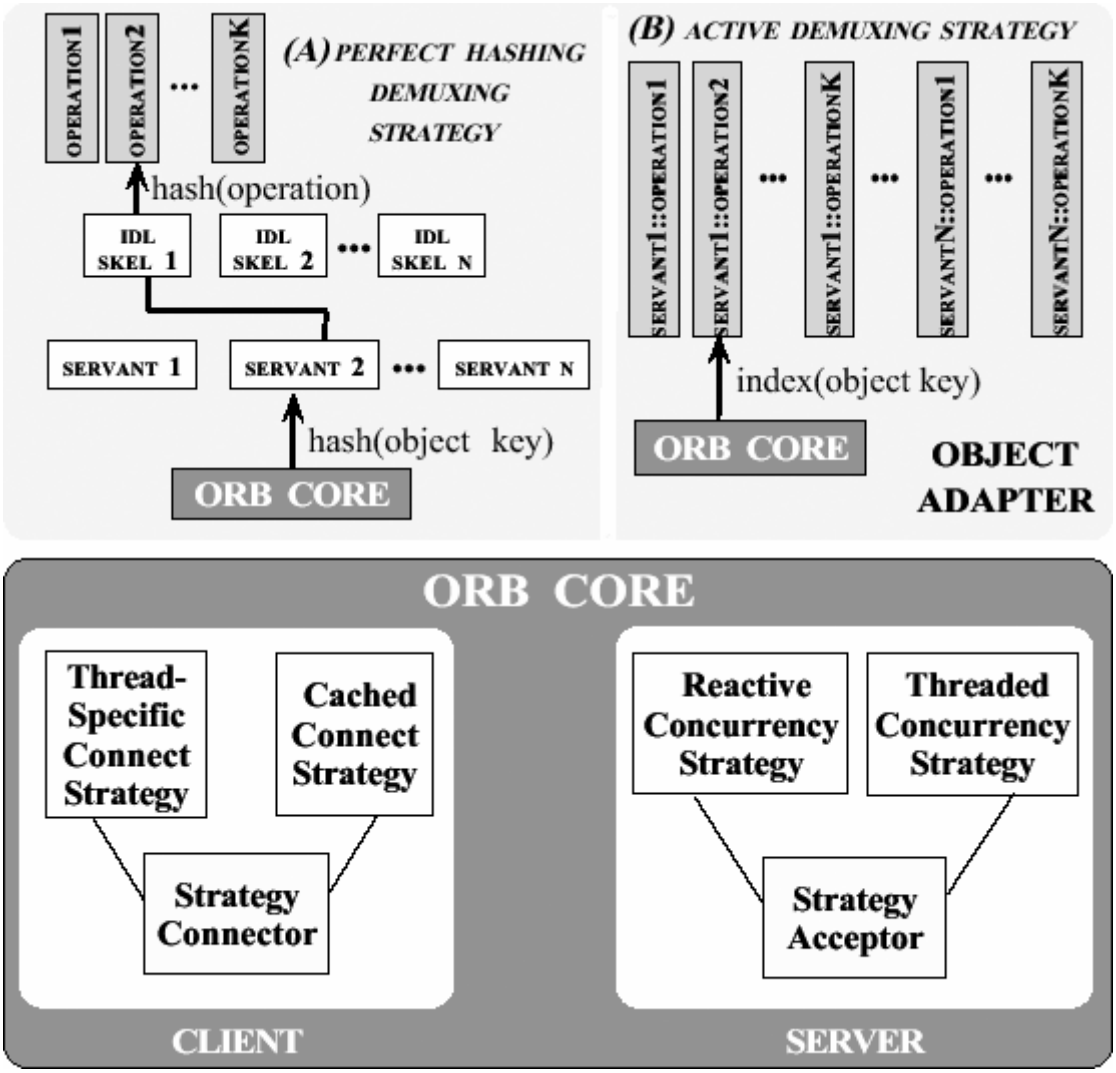

图 3-14 TAO 中的 ORB 核心和 POA 策略

3.3.3.7 使用抽象工厂模式合并 ORB 策略

上下文：TAO 支持许多潜在的策略变种。表 3-1 显示了一个简单的策略例子，用于创建 TAO 的两种配置。配置 1 是航空控制应用，它具有确定性的实时需求[11]。配置 2 是电子医学成像应用[41]，它具有高吞吐量需求。一般而言，正确地合成所有 ORB 策略必须消除下面两种需求带来的压力：（1）确保语义兼容的策略的配置，以及（2）简化大量独立策略的管理。

	策略配置			
应用	*并发*	*分派*	*多路分离*	*协议*
航空控制	Thread-per-priority	基于优先级	理想哈希	VME 底板
医学成像	Thread-per-connection	FIFO	主动式多路分离	TCP/IP

表 3-1 实例应用和它们的 ORB 策略配置

问题：在复杂的 ORB 软件（以及其他类型的软件）中大量使用策略模式会产生不合需要的副作用：可扩展性变得难以管理，原因如下：

- ***配置和改进的复杂化：***ORB 源码可能会与硬编码的对策略类型的引用搅在一起，从而使配置和改进复杂化。例如，在特定的应用领域中（比如实时航空控制或医学成像），许多独立的策略必须和谐地工作。但是，挨个地通过名字来确定这些策略需要很麻烦地用一组策略（在另一领域中可能会不同）来替换所选择的策略。
- ***语义不兼容：***特定的 ORB 策略配置并不总是能兼容地交互。例如，表 3-1 中所示的用于调度请求的 FIFO 策略可能不能与 thread-per-priority 并发体系结构一起工作。问题源于“ 以到达顺序调度请求 ”（也就是，FIFO 排队） vs. “ 基于相对优先级分派请求 ”（也就是，占先式的基于优先级的线程分派）之间的语义不兼容。而且，某些策略只在特定的前提被满足时才有用。例如，理想哈希多路分离策略通常只对那些对所有仆人进行离线配置的系统才是可行的[22]。

高度可配置的 ORB 怎样才能减少管理它的众多策略所带来的复杂性，并在组合离散的策略时实现语义的一致性呢？

解决方案—>抽象工厂模式：应用*抽象工厂*模式是将多种 ORB 策略合并进语义兼容的配置的有效方式[15]。该模式提供单一访问点，集成所有用于配置 ORB 的策略。具体的子类随即聚合相互兼容的应用特有或领域特有的策略，这些策略可以通过一种在语义上有意义的方式被一起替换。一般而言，抽象工厂模式应在下述情况下使用：应用必须合并许多策略的配置，每种策略都具有必须一起改变的多种可选方案。

在 TAO 中使用抽象工厂模式：TAO 的所有 ORB 策略都被合并进两个抽象工厂中，这两个工厂被实现为单体（Singleton）[15]。如图 3-15 所示，一个工厂封装客户特有的策略，另一个工厂封装服务器特有的策略。这些抽象工厂封装服务器中的请求多路分离、调度和分派策略，以及在客户和服务器中都有的并发策略。通过使用抽象工厂模式，TAO 可以方便而一致地配置不同的 ORB 特性。

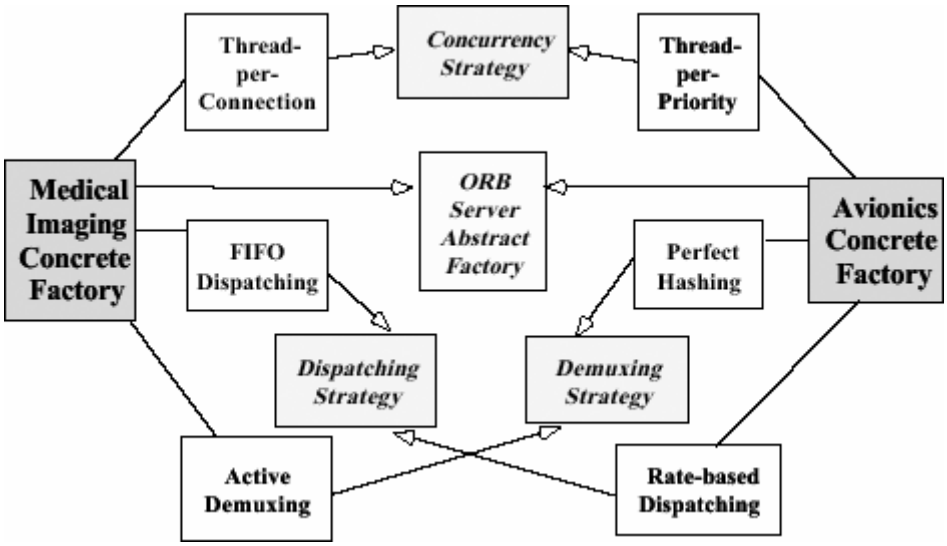

图 3-15 TAO 中所用的工厂

3.3.3.8 通过组件配置器模式动态地配置 ORB

上下文：有许多计算资源（比如内存和 CPU）的成本正在持续下降。但是，ORB 还是必须避免过度地消耗有限的系统资源。对于需要小内存占用和可预测的 CUP 使用的嵌入式及实时系统来说，这样的节约是特别必要的。许多应用还可以从*动态*扩展 ORB 的能力（也就是，允许它们的策略在运行时被配置）中获益。

问题：尽管策略和抽象工厂模式简化了 ORB 针对特定应用需求和系统特性的定制，这些模式还是有可能为可扩展 ORB 带来以下问题：

- ***高资源占用*：**策略模式的广泛使用可能会极大地扩大被配置进 ORB 中的策略的数目，从而增加运行 ORB 所需的系统资源。
- ***不可避免的系统停机时间*：**如果使用抽象工厂来在编译时或链接时静态地配置策略，就很难增强现有的策略或增加新策略，而不（1）改变*策略*的消费者或*抽象工厂*的现有源码，（2）重新编译和链接 ORB，以及（3）重启运行中的 ORB 和它们的应用仆人。

尽管没有明确地使用策略模式，SunSoft IIOP 的确允许应用在运行时改变特定的 ORB 策略。但是，这些不同的策略必须在编译时被静态地配置进 SunSoft IIOP。而且，随着可选方案数目的增长，实现它们所需的代码数量也会增长。例如，图 3-16 演示 SunSoft IIOP 的改变并发策略的方法。

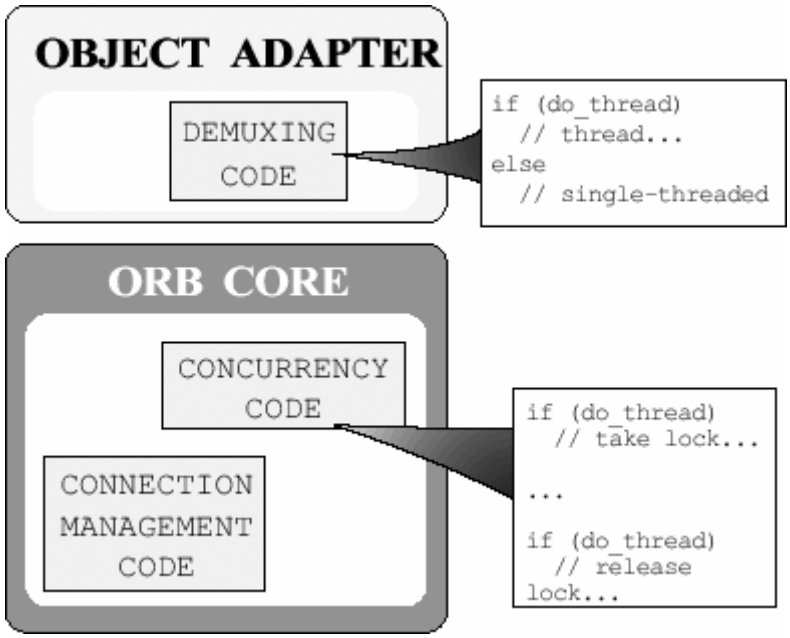

图 3-16 SunSoft IIOP 硬编码的策略使用

可能会被并发策略的选择影响的每个区域的代码都被相信是独立于其他区域而工作的。这样的决策点的增殖增加了代码、以及未来的增强和维护的复杂性。而且，指定策略的数据类型的选择使得集成新的并发体系结构变复杂了，因为可能必须改变类型（bool）、还有将策略分类符解码为动作的程序结构，

if (do_thread) then … else …。

一般而言，静态配置只对少量的、固定数目的策略来说是可行的。但是，静态地配置复杂的 ORB 中间件（1）使改进复杂化，（2）增加了系统资源占用，以及（3）导致不可避免的用于修改现有组件的系统停机时间。

那么 ORB 实现怎样才能减少策略和抽象工厂模式的普遍使用所带来的“过大、过于静态”的副作用呢？

解决方案—>组件配置器模式：应用*组件配置器模式*是增强 ORB 的动态性的有效方式 [8]。该模式使用显式的动态链接[8]机制来在安装时和/或运行时在 ORB 中获取、利用，以及/或者移去自定义策略和抽象工厂对象的运行时地址绑定。广泛可用的显式动态链接机制包括 SVR4 UNIX 中的 dlopen/dlsym/dlclose 函数[42]和 Windows NT 的 WIN32 子系统[43]中的 LoadLibrary/GetProcAddress 函数。TAO 所用的 ACE 包装外观可移植地封装了这些 OS API。

通过使用组件配置器模式，使得 ORB 策略的*行为*与策略实现*何时*被配置进 ORB 得以去耦合。例如，ORB 策略可在编译时、安装时、甚或运行时从动态链接库（DLL）链接进 ORB。而且，通过允许应用开发者和/或管理员只动态链接特定 ORB 特性所需的那些组件，组件配置器模式可以减少 ORB 的内存占用。

一般而言，组件对象模式应被用于（1）应用想要动态配置它的构成组件时，以及（2）由于可能性的数目过多、或不能预计值的范围，传统技术，比如命令行参数，不足以胜任时。

在 TAO 中使用组件配置器：TAO 联合使用组件配置器模式和策略及抽象工厂模式来动态地安装它所需的策略，而不用（1）重新编译或静态重链接现有的代码，或是（2）终止并重启现有的 ORB 和它的应用仆人。这样的设计允许为特定的平台和应用需求而裁剪 TAO 的行为，而无需访问或修改 ORB 的源码。

此外，组件配置器模式允许应用在运行时定制 TAO 的特性。例如，在 TAO 的 ORB 初始化阶段，它使用 OS 提供的链接机制（由 ACE 包装外观封装）来为特定的使用情况链接适当的具体工厂。图 3-17 显示为 TAO 所支持的不同应用领域（航空控制和医学成像）而调谐的两个工厂。

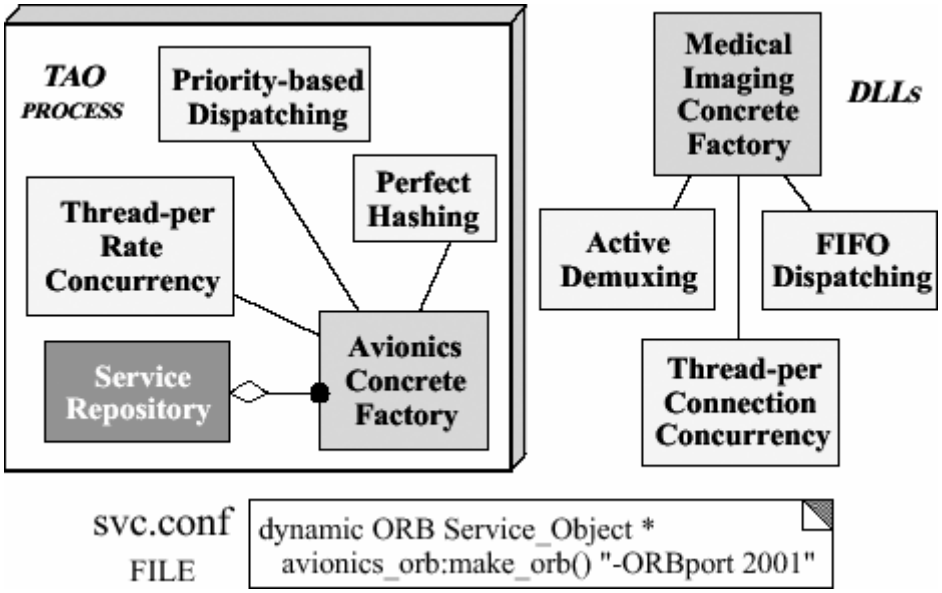

图 3-17 在 TAO 中使用组件配置器模式

在图 3-17 所示的配置中，组件配置器已查询 comp.conf 脚本，并在进程中安装了航空控制具体工厂。

使用这一 ORB 特性的应用将通过一组特定的并发、多路分离和分派策略来配置。医学成像具体工厂驻留在外在于现有 ORB 进程的的 DLL 中。为配置不同的 ORB 特性，此工厂可以在 TAO 的 ORB 服务器初始化阶段动态安装。

3.3.4 设计挑战和应对它们的模式总结

表 3-2 总结了在“ ORB 设计挑战 ”与“ 模式语言中我们用于在 TAO 中应对这些挑战的模式 ”之间的映射。该表的焦点是由个体模式所消除的压力。但是，TAO 还可以从模式语言中*多个*模式之间的协作中获益。例如，接受器和连接器模式利用反应器模式来通知它们何时在 OS 级上有连接事件发生。

压力	**应对模式**
抽象低级系统调用	*包装外观*
ORB 事件多路分离	*反应器*
ORB 连接管理	*接受器－连接器*
高效并发模型	*领导者/跟随者*
可插式策略	*策略*
组相类似的初始化	*抽象工厂*
动态运行时配置	*组件配置器*

表 3-2 压力和消除它们的模式总结

而且，模式常常必须进行协作，以减少孤立地应用它们所带来的缺点。例如，在 TAO 中使用抽象工厂模式的原因是避免策略模式的大量使用所导致的复杂性。尽管策略模式简化了为特定应用需求和网络/终端系统特性定制 ORB 的工作，人工地管理大量的策略交互仍然是麻烦而易错的。

3.3.5 评估模式对 ORB 中间件的贡献

3.3.3 描述了 TAO 中所用的模式语言，并从质量方面评估了这些模式是怎样帮助克服 SunSoft IIOP 的设计局限的。下面的讨论进一步从质量方面来评估将模式应用于 ORB 中间件的好处。

3.3.5.1 证据何在？

使用模式语言实现 TAO 在软件可复用性和可维护性方面产生了显著的、可以度量的改善。其结果在表 3-3 中总结。该表比较了 TAO 和 SunSoft IIOP 的以下规格：

1. 实现关键 ORB 任务（比如连接管理、请求传送、Socket 和请求多路分离、整编，以及分派）所需的方法数目。

2. 这些方法的非注释代码行（LOC）的总数。
3. 方法的 McCabe Cyclometric 复杂度规格 v(G)。v(G)规格使用图形原理来使代码复杂度与代码模块中可使用的基本路径的数目相关联。在 C++中，模块被定义为方法。

在 TAO 中，模式使用显著地减少了专用代码的数量和特定操作的复杂度。例如，在客户端*连接管理*操作中的总代码行缩减了 5 倍。而且，这一组件的复杂度实质上缩减了 16 倍。LOC 和复杂度的缩减源于以下因素：

- 这些 ORB 任务是我们在开发 TAO 时的最初工作的焦点。
- ACE 中的模式和组件（特别地，接受器、连接器和反应器）包含了连接管理和 Socket 多路分离的许多细节。

其他方面没有产生同样多的改善。特别地，GIOP *调用*任务的大小实际上反而增大了，而且还维持了不变的 v(G)。这样的增大有两个原因：

1. 应用于这些情况中的主要模式是包装外观，它通过 ACE 包装来替换低级系统调用，但并没有分解出公共的策略；以及
2. SunSoft IIOP 没有捕捉所有的错误条件，而 TAO 做得要更为完整。因此，TAO 中的额外代码对于提供更为健壮的 ORB 来说是必要的。

有系统地应用模式确实可以改善复杂软件的可维护性，最有说服力的证据在图 3-18 中显示。该图演示了 TAO 中受影响方法的 v(G)百分率的分布。如图中所示，大多数 TAO 的代码都以简单的方式被构造，大约 70%的方法的 v(G)落入了范围 1-5 中。

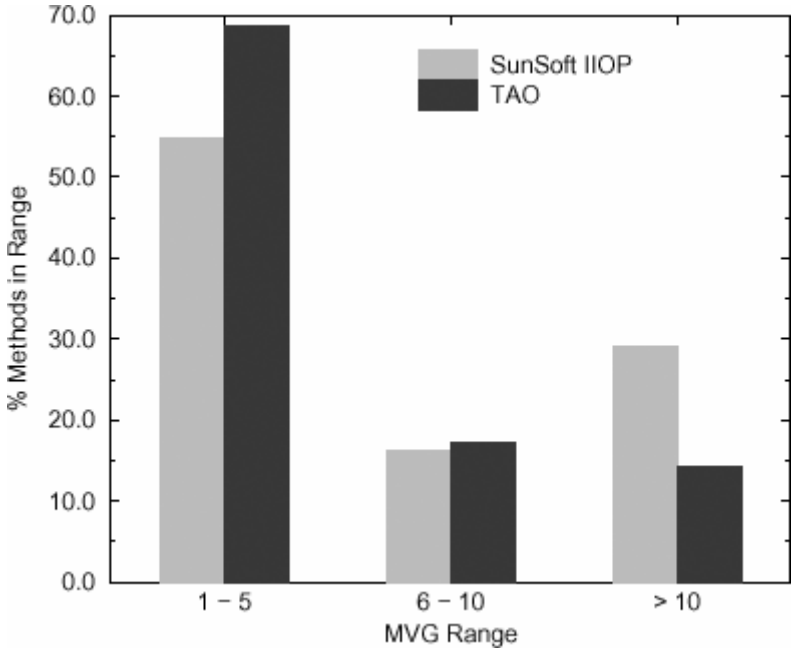

图 3-18 ORB 方法的 v(G)分布

相反，在 SunSoft IIOP 的方法有相当的百分率（55%）在该范围中的同时，剩下的许多方法（29%）

有着大于 10 的 v(G)。这种差异的原因是 SunSoft IIOP 使用了整体式的编码风格，有着很长的方法。例如，v(G)大于 10 的方法的平均长度超过 80 LOC。这产生了过于复杂、难以调试和理解的代码。

在 TAO 中，大多数整体式的 SunSoft IIOP 方法都在模式集成时被分解为较小的方法。TAO 的多数（86%）方法有着 10 以下的 v(G)。在此数目中，大约 70%有着 1 到 5 之间的 v(G)。TAO 中相对很少（14%）的 v(G)大于 10 的方法在很大程度上都未作变动地来自于原来的 SunSoft IIOP TypeCode 解释器。TAO 后来的版本已经完全移去了 TypeCode 解释器，并用由 TAO 的 IDL 编译器自动生成的 Stub 和 Skeleton 来作了替换。因而，TAO ORB 开发者再也不需要维护这些代码了。

一般而言，SunSoft IIOP 中整体式方法的使用不仅增加了它的维护工作，而且还降低了它的性能，因为处理器缓存的命中率减少了[20]，。因此，我们计划试验其他模式的应用，比如*命令与模板方法*（Command and Template Method）[15]，以将这些整体式方法简化和优化成较小的、更内聚的方法。

3.3.5.2 有何好处？

总之，将模式语言应用于 TAO 产生了下列好处：

更高的可扩展性：通过允许可扩展性被“设计进”ORB，像抽象工厂、策略和组件配置器这样的模式简化了特定的应用领域中 TAO 的配置。相反，缺乏这些模式的 DOC 中间件相当地难以扩展。

增强的设计清晰性：通过将模式语言应用于 TAO，我们不仅开发了更为可扩展的 ORB，还为表达 ORB 中间件设计发明了更丰富的词汇表。特别地，模式语言捕捉并清晰地表达了 ORB 中的复杂对象结构的设计原理。而且，通过依照在许多类型的软件系统中重复出现的设计需求来描述 ORB 的体系结构，它还有助于阐明 ORB 的结构，并揭示其动机。模式的富有表现力的能力使得我们能够简洁地传达复杂软件系统（比如 TAO）的设计。在我们持续地学习 ORB 及组成它们的模式的同时，我们预期我们的词汇表还将会增长，并演变成一种更为全面的模式语言。

增长的可移植性和复用：TAO 在 ACE 框架之上构建，后者提供了许多关键通信软件模式的实现[28]。ACE 的使用简化了 TAO 到多种 OS 平台的移植，因为 ACE 维护者承担了大多数移植工作。此外，因为 ACE 框架富含可配置的高性能、实时、面向网络的组件，我们可以通过有效利用该框架来获得相当可观的代码复用。表 3 所示的代码行（LOC）的稳固下降表明了这一点。

3.3.5.3 有何缺点？

模式语言的使用也可能会带来一些缺点。我们在下面总结这些缺点，并讨论我们怎样来在 TAO 中使它们最小化。

抽象代价：许多模式使用间接（indirection）来增加组件耦合。例如，反应器模式使用虚方法来分离应用特有的 Event Handler 逻辑与通用的事件多路分离和分派逻辑。使用这些模式实现所带来的额外的间接可能会潜在地降低性能。为减少这些缺点，我们小心地应用了 C++编程语言特性（比如内联函数和模板）以及其他优化（比如消除去整编开销[20]和多路分离开销[36]）来最小化性能开销。作为结果，TAO 比原

来的硬编码 SunSoft IIOP[20]要快很多。

额外的外部依赖：在 SunSoft IIOP 仅依赖系统级接口和库的同时，TAO 依赖于 ACE 中的包装外观。因为 ACE 封装了广泛的低级 OS 机制，将它移植到新平台所需的工作可能会高于移植仅使用了 OS API 的子集的 SunSoft IIOP。但是，因为 ACE 已经被移植到许多平台，移植到新平台的工作相对就少了。大多数平台变化源已被隔离到 ACE 中的少数模块中。

3.4 结束语

本论文介绍了一项个案研究，演示我们怎样将模式语言应用于增强 TAO 的可扩展性；TAO 是一种可动态配置的 ORB，它的应用目标是有高性能和实时需求的分布式应用。我们发现了质量和数量的证据，证明模式语言的使用有助于阐明执行关键 ORB 任务的组件的结构、以及它们之间的协作。这些任务包括事件多路分离和事件处理器分派、连接建立和应用服务初始化、并发控制，以及动态配置。此外，通过使用于处理客户请求的轻量级的和优化的策略的透明配置成为可能，模式还改善了 TAO 的性能和可预测性。

应用模式语言来指导 TAO 的设计的主要好处是，该语言中的模式的系统应用显著地促进了 ORB 的去耦合和面向对象结构。我们所用的模式被应用的次序大致与它们在 3.3.3 出现的次序相同。TAO 的每次改进都有效地利用了前面的改进的结果。这样的反复过程揭示了新的洞见：可应用模式语言中的哪些模式、以及怎样将它们应用于后续阶段。

ACE 和 TAO 完整的 C++源码、例子和文档可在 http://www.cs.wustl.edu/~schmidt/TAO.html 自由获取。

参考文献

[1] R. Johnson, “Frameworks = Patterns + Components,” *Communications of the AC*M, vol. 40, Oct. 1997.

[2] S. Vinoski, “CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments,” *IEEE Communications Magazin*e, vol. 14, February 1997.

[3] J. A. Zinky, D. E. Bakken, and R. Schantz, “Architectural Support for Quality of Service for CORBA Objects,” *Theory and Practice of Object System*s, vol. 3, no. 1, 1997.

[4] Object Management Group, *The Common Object Request Broker: Architecture and Specificatio*n, 2.2 ed., Feb. 1998.

[5] D. Box, *Essential CO*M. Addison-Wesley, Reading, MA, 1997.

[6] A. Wollrath, R. Riggs, and J. Waldo, “A Distributed Object Model for the Java System,” *USENIX Computing System*s,vol.9,November/ December 1996.

[7] D. C. Schmidt, “Experience Using Design Patterns to Develop Reuseable Object-Oriented Communication Software,” *Communications of the ACM (Special Issue on Object-Oriented Experiences*), vol. 38, October 1995.

[8] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrency and Distributed Objects, Volume* 2. New York, NY: Wiley & Sons, 2000.

[9] D. C. Schmidt, “Applying a Pattern Language to Develop Application-level Gateways,” in *Design Patterns in Communications* (L. Rising, ed.), Cambridge University Press, 2000.

[10] D. C. Schmidt, D. L. Levine, and S. Mungee, “The Design and Performance of Real-Time Object Request Brokers,” *Computer*

*Communication*s, vol. 21, pp. 294–324, Apr. 1998.

[11] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), ACM, October 1997.

[12] S. Mungee, N. Surendran, and D. C. Schmidt, "The Design and Performance of a CORBA Audio/Video Streaming Service," in *Proceedings of the Hawaiian International Conference on System Science*s, Jan. 1999.

[13] C. O'Ryan, D. C. Schmidt, and D. Levine, "Applying a Scalable CORBA Events Service to Large-scale Distributed Interactive Simulations," in *Proceedings of the 5th Workshop on Object-oriented Real-time Dependable System*s, (Montery, CA), IEEE, Nov. 1999.

[14] Object Management Group, *The Common Object Request Broker: Architecture and Specificatio*n, 2.3 ed., June 1999.

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

[16] C. O'Ryan, F. Kuhns, D. C. Schmidt, and J. Parsons, "Applying Patterns to Develop a Pluggable Protocols Framework for ORB Middleware," in *Design Patterns in Communications* (L. Rising, ed.), Cambridge University Press, 2000.

[17] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom, "Flick: A Flexible, Optimizing IDL Compiler," in *Proceedings of ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, (Las Vegas, NV), ACM, June 1997.

[18] M. Henning, "Binding, Migration, and Scalability in CORBA," *Communications of the ACM special issue on CORB*A, vol. 41, Oct. 1998.

[19] Object Management Group, *Realtime CORBA Joint Revised Submissio*n, OMG Document orbos/99-02-12 ed., March 1999.

[20] A. Gokhale and D. C. Schmidt, "Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems," *Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia System*s, vol. 17, Sept. 1999.

[21] D. C. Schmidt, "GPERF: A Perfect Hash Function Generator," in *Proceedings of the 2nd C++ Conferenc*e, (San Francisco, California), pp. 87–102, USENIX, April 1990.

[22] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, "Using Principle Patterns to Optimize Real-time ORBs," *Concurrency Magazin*e, vol. 8, no. 1, 2000.

[23] C. D. Gill, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middlewar*e, to appear 2000.

[24] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers," *Journal of Real-time Systems, special issue on Real-time Computing in the Age of the Web and the Internet*, To appear 2000.

[25] C. O'Ryan, F. Kuhns, D. C. Schmidt, O. Othman, and J. Parsons, "The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware," in *Proceedings of the Middleware 2000 Conferenc*e, ACM/IFIP, Apr. 2000.

[26] F. Kuhns, D. C. Schmidt, C. O'Ryan, and D. Levine, "Supporting High-performance I/O in QoS-enabled ORB Middleware," *Cluster Computing: the Journal on Networks, Software, and Application*s, 2000.

[27] Z. D. Dittia, G. M. Parulkar, and J. R. Cox, Jr., "The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques," in *Proceedings of INFOCOM '97*, (Kobe, Japan), pp. 179–187, IEEE, April 1997.

[28] D. C. Schmidt, "Applying Design Patterns and Frameworks to Develop Object-Oriented Communication Software," in *Handbook of Programming Languages* (P. Salus, ed.), MacMillan Computer Publishing, 1997.

[29] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, M. Kircher, and J. Parsons, "The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging," in *Proceedings of the Middleware 2000 Conferenc*e, ACM/IFIP, Apr. 2000.

[30] C. O'Ryan, D. C. Schmidt, F. Kuhns, M. Spivak, J. Parsons, I. Pyarali, and D. Levine, "Evaluating Policies and Mechanisms for Supporting Embedded, Real-Time Applications with CORBA 3.0," in *Proceedings of the 6th IEEE Real-Time Technology and Applications Symposiu*m, (Washington DC), IEEE, May 2000.

[31] B. Natarajan, A. Gokhale, D. C. Schmidt, and S. Yajnik, "DOORS: Towards High-performance Fault-Tolerant CORBA," in *Proceedings of the 2nd International Symposium on Distributed Objects and Applications (DOA 2000)*, (Antwerp, Belgium), OMG, Sept. 2000.

[32] B. Natarajan, A. Gokhale, D. C. Schmidt, and S. Yajnik, "Applying Patterns to Improve the Performance of Fault-Tolerant CORBA," in *Proceedings of the 7th International Conference on High Performance Computing (HiPC 2000)*, (Bangalore, India), ACM/IEEE, Dec. 2000.

[33] J. Hu, S. Mungee, and D. C. Schmidt, “Principles for Developing and Measuring High-performance Web Servers over ATM,” in *Proceeedings of INFOCOM ’9*8, March/April 1998.

[34] F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. Magalhaes, and R. Campbell, “Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB,” in *Proceedings of the Middleware 2000 Conferenc*e, ACM/IFIP, Apr. 2000.

[35] M. Roman, M. D. Mickunas, F. Kon, and R. Campbell, “LegORB and Ubiquitous CORBA,” in *Reflective Middleware Worksho*p,ACM/IFIP, Apr. 2000.

[36] A. Gokhale and D. C. Schmidt, “Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks,” *Transactions on Computin*g, vol. 47, no. 4, 1998.

[37] J. Hu and D. C. Schmidt, “JAWS: A Framework for High Performance Web Servers,” i n *Domain-Specific Application Frameworks: Frameworks Experience by Industry* (M. Fayad and R. Johnson, eds.), Wiley & Sons, 1999.

[38] Object Management Group, *OMG Real-time Request for Proposa*l, OMG Document ptc/97-06-20 ed., June 1997.

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

[40] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, “Alleviating Priority Inversion and Non-determinism in Real-time CORBA ORB Core Architectures,” in *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposiu*m, (Denver, CO), IEEE, June 1998.

[41] 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.

[42] R. Gingell, M. Lee, X. Dang, and M. Weeks, “Shared Libraries in SunOS,” in *Proceedings of the Summer 1987 USENIX Technical Conferenc*e, (Phoenix, Arizona), 1987.

[43] D. A. Solomon, *Inside Windows NT, 2nd Ed.* Redmond, Washington: Microsoft Press, 2nd ed., 1998.

[44] T. J. McCabe, “A Complexity Measure,” *IEEE Transactions on Software Engineerin*g, vol. SE-2, Dec. 1976.