第8章 前摄器（Proactor）：用于为异步事件多路分离和分派处理器的对象行为模式

Irfan Pyarali　　Tim Harrison　　Douglas C. Schmidt　　Thomas D. Jordan

摘 要

现代操作系统为开发并发应用提供了多种机制。同步多线程是一种流行的机制，用于开发同时执行多个操作的应用。但是，线程常常有很高的性能开销，并且需要对同步模式和原理有深入的了解。因此，有越来越多的操作系统支持异步机制，在减少多线程的大量开销和复杂性的同时，提供了并发的好处。

本论文中介绍的前摄器（Proactor）模式描述怎样构造应用和系统，以有效地利用操作系统支持的异步机制。当应用调用异步操作时，OS代表应用执行此操作。这使得应用可以让多个操作同时运行，而又不需要应用拥有相应数目的线程。因此，通过使用更少的线程和有效利用OS对异步操作的支持，前摄器模式简化了并发编程，并改善了性能。

8.1 意图

前摄器模式支持多个事件处理器的多路分离和分派，这些处理器由异步事件的*完成*来触发。通过集成完成事件（completion event）的多路分离和相应的事件处理器的分派，该模式简化了异步应用的开发。

8.2 动机

这一部分提供使用前摄器模式的上下文和动机。

8.2.1 上下文和压力

前摄器模式应该被用于应用需要并发执行操作的性能好处、又不想受到同步多线程或反应式编程的约束时。为说明这些好处，设想一个需要并发执行多个操作的网络应用。例如，一个高性能Web服务器必须并发处理发送自多个客户的HTTP请求[1, 2]。图8-1 显示了Web浏览器和Web服务器之间的典型交

互。当用户指示浏览器打开一个URL时，浏览器发送一个HTTP GET请求给Web服务器。收到请求，服务器就解析并校验请求，并将指定的文件发回给浏览器。

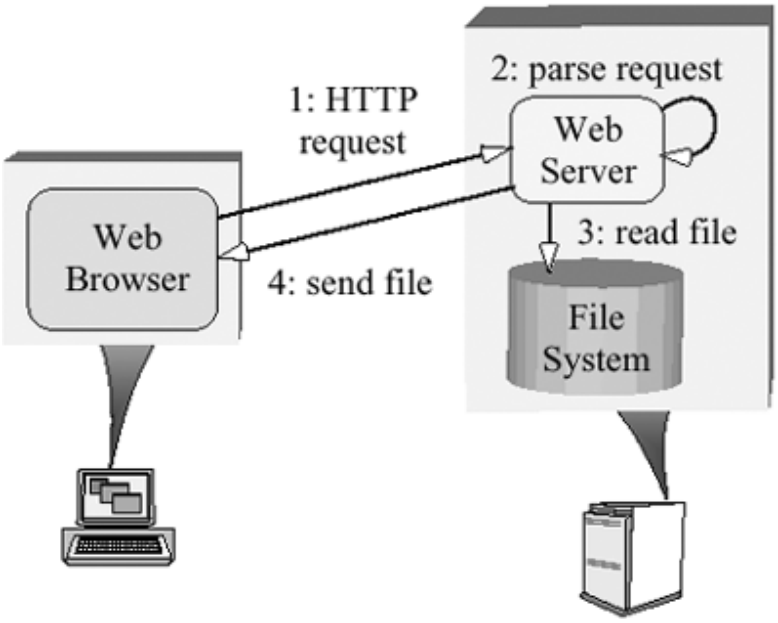

图8-1 典型的Web服务器通信软件体系结构

开发高性能Web服务器要求消除以下压力：

- *并发性*：服务器必须同时执行多个客户请求；
- *效率*：服务器必须最小化响应延迟、最大化吞吐量，并避免不必要地使用CPU；
- *编程简单性*：服务器的设计应该简化高效的并发策略的使用；
- *可适配性*：应该使继承新的或改进的传输协议（比如HTTP 1.1[3]）所带来的维护代价最小化。

Web服务器可以使用若干并发策略来实现，包括多个同步线程、反应式同步事件分派和前摄式异步事件分派。下面，我们检查传统方法的缺点，并解释前摄器模式是怎样提供一种强大的技术，为高性能并发应用而支持高效、灵活的异步事件分派策略的。

8.2.2 传统并发模型的常见陷阱和缺陷

同步的多线程和反应式编程是实现并发的常用方法。这一部分描述这些编程模型的缺点。

8.2.2.1 通过多个同步线程实现的并发

或许最为直观的实现并发Web服务器的途径是使用*同步的多线程*。在此模型中，多个服务器线程同时处理来自多个客户的HTTP GET请求。每个线程同步地执行连接建立、HTTP请求读取、请求解析和文件传输操作。作为结果，每个操作都阻塞直到完成。

同步线程的主要优点是应用代码的简化。特别是，Web服务器为服务客户A的请求所执行的操作在很大程度上独立于为服务客户B的请求所需的操作。因而，很容易在分离的线程中对不同的请求进行服务，因为在线程之间共享的状态数量很少；这也最小化了对同步的需要。而且，在分离的线程中执行应用逻辑也使得开发者可以使用直观的顺序命令和阻塞操作。

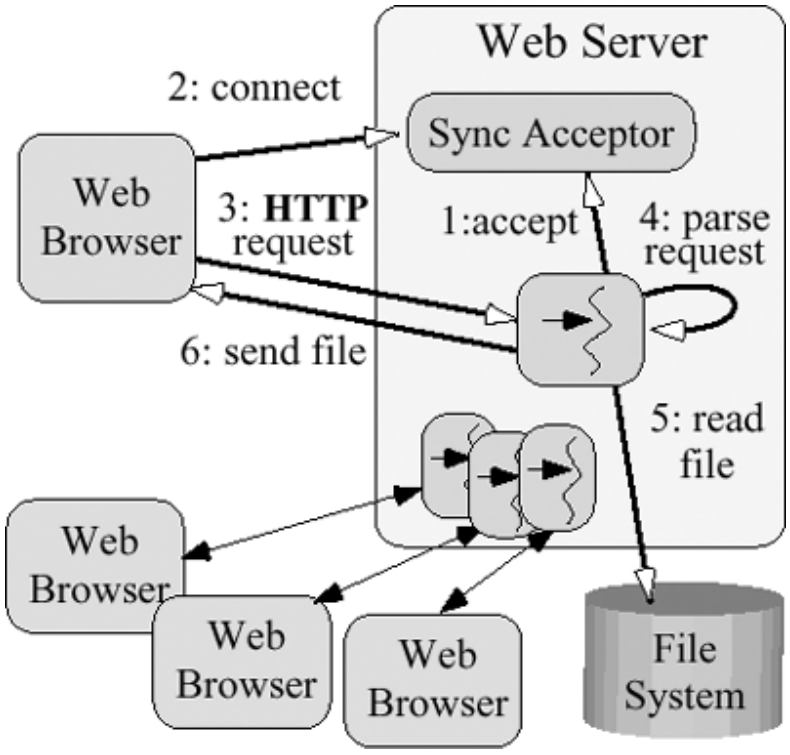

图8-2 多线程Web服务器体系结构

图8-2显示使用同步线程来设计的Web服务器怎样并发地处理多个客户请求。该图显示的Sync Acceptor对象封装服务器端用于同步接受网络连接的机制。使用“Thread Per Connection”并发模型，各个线程为服务HTTP GET请求所执行的一系列步骤可被总结如下：

1. 每个线程同步地阻塞在accept socket调用中，等待客户连接请求；
2. 客户连接到服务器，连接被接受；
3. 新客户的HTTP请求被同步地从网络连接中读取；
4. 请求被解析；
5. 所请求的文件被同步地读取；
6. 文件被同步地发送给客户。

附录A.1中有一个将同步线程模型应用于Web服务器的C++代码例子。

如上所述，每个并发地连接的客户由一个专用的服务器线程服务。在继续为其他HTTP请求服务之前，该线程同步地完成一个被请求的操作。因此，要在服务多个客户时执行同步I/O，Web服务器必须派生多个线程。尽管这种同步线程模式是直观的，且能够相对高效地映射到多CPU平台上，它还是有以下缺点：

线程策略与并发策略被紧耦合：这种体系结构要求每个相连客户都有一个专用的线程。通过针对可用资源（比如使用线程池来对应CPU的数目）、而不是正被并发服务的客户的数目来调整其线程策略，可能会更好地优化一个并发应用；

更大的同步复杂性：线程可能会增加序列化对服务器的共享资源（比如缓存文件和Web页面点击日志）的访问所必需的同步机制的复杂性；

更多的性能开销：由于上下文切换、同步和CPU间的数据移动[4]，线程的执行可能很低效；

不可移植性：线程有可能在有些平台上不可用。而且，根据对占先式和非占先式线程的支持，OS平台之间的差异非常大。因而，很难构建能够跨平台统一运作的多线程服务器。

作为这些缺点的结果，多线程常常不是开发并发Web服务器的最为高效的、也不是最不复杂的解决方案。

8.2.2.2 通过反应式同步事件分派实现的并发

另一种实现同步Web服务器的常用方法是使用*反应式事件分派*模型。反应堆（Reactor）模式描述应用怎样将Event Handler登记到Initiation Dispatcher。Initiation Dispatcher通知Event Handler何时能发起一项操作而不阻塞。

单线程并发Web服务器可以使用反应式事件分派模型，它在一个事件循环中等待Reactor通知它发起适当的操作。Web服务器中反应式操作的一个例子是Acceptor（接受器）[6]到Initiation Dispatcher的登记。当数据在网络连接上到达时，分派器回调Acceptor，后者接受网络连接，并创建HTTP Handler。于是这个HTTP Handler就登记到Reactor，以在Web服务器的单线程控制中处理在那个连接上到来的URL请求。

图8-3和图8-4显示使用反应式事件分派设计的Web服务器怎样处理多个客户。图8-3显示当客户连接到Web服务器时所采取的步骤。图8-4显示Web服务器怎样处理客户请求。图8-3的一系列步骤可被总结如下：

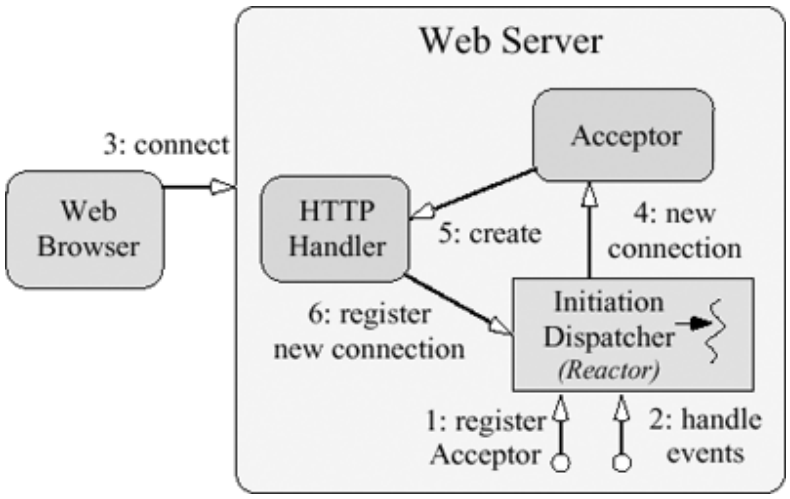

图8-3 客户连接到反应式Web服务器

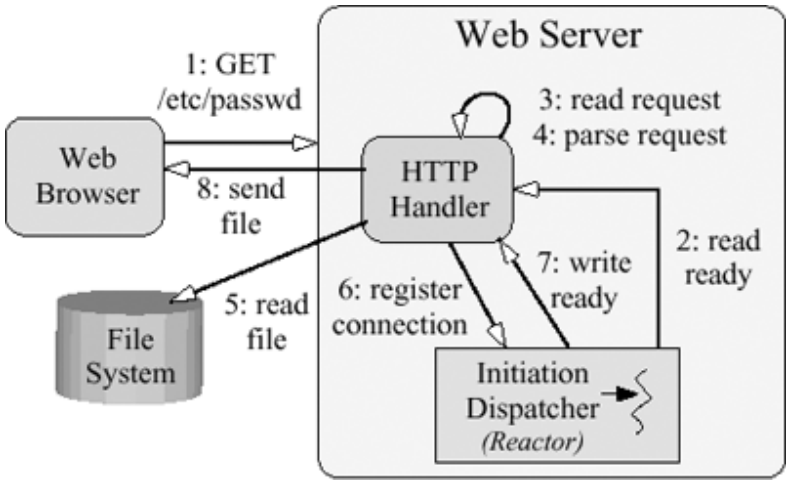

图8-4 客户发送HTTP请求到反应式Web服务器

1. Web服务器将Acceptor登记到Initiation Dispatcher，以接受新连接；
2. Web服务器调用Initiation Dispatcher的事件循环；
3. 客户连接到Web服务器；
4. Initiation Dispatcher将新连接请求通知Acceptor，后者接受新连接；
5. Acceptor创建HTTP Handler，以服务新客户；
6. HTTP Handler将连接登记到Initiation Dispatcher，以读取客户请求数据（就是说，在连接变得“读就绪”时）；
7. HTTP Handler服务来自新客户的请求。

图8-4显示反应式Web服务器为服务HTTP GET请求所采取的一系列步骤。该过程描述如下：

1. 客户发送HTTP GET请求；
2. 当客户请求数据到达服务器时，Initiation Dispatcher通知HTTP Handler；

3. 请求以非阻塞方式被读取，于是如果操作会导致调用线程阻塞，读操作就返回EWOULDBLOCK（步骤2和3将重复直到请求被完全读取）；
4. HTTP Handler解析HTTP请求；
5. 所请求的文件从文件系统中被同步读取；
6. 为发送文件数据（就是说，当连接变得“写就绪”时），HTTP Handler将连接登记到Initiation Dispatcher；
7. 当TCP连接变得写就绪时，Initiation Dispatcher通知HTTP Handler；
8. HTTP Handler以非阻塞方式将所请求文件发送给客户，于是如果操作会导致调用线程阻塞，写操作就返回EWOULDBLOCK（步骤7和8将重复直到数据被完全递送）。

附录A.2中有一个将反应式事件分派模型应用于Web服务器的C++代码例子。

因为Initiation Dispatcher运行在单线程中，网络I/O操作以非阻塞方式运行在Reactor的控制之下。如果当前操作的进度停止了，操作就被转手给Initiation Dispatcher，由它监控系统操作的状态。当操作可以再度前进时，适当的Event Handler会被通知。

反应式模式的主要优点是可移植性，粗粒度并发控制带来的低开销（就是说，单线程不需要同步或上下文切换），以及通过使应用逻辑与分派机制去耦合所获得的模块性。但是，该方法有以下缺点：

复杂的编程：如从前面的列表所看到的，程序员必须编写复杂的逻辑，以保证服务器不会在服务一个特定客户时阻塞。

缺乏多线程的OS支持：大多数操作系统通过select系统调用[7]来实现反应式分派模型。但是，select不允许多于一个的线程在同一个描述符集上等待。这使得反应式模型不适用于高性能应用，因为它没有有效地利用硬件的并行性。

可运行任务的调度：在支持占先式线程的同步多线程体系结构中，将可运行线程调度并时分（time-slice）到可用CPU上是操作系统的责任。这样的调度支持在反应式体系结构中不可用，因为在应用中只有一个线程。因此，系统的开发者必须小心地在所有连接到Web服务器的客户之间将线程分时。这只能通过执行短持续时间、非阻塞的操作来完成。

作为这些缺点的结果，当硬件并行可用时，反应式事件分派不是最为高效的模型。由于需要避免使用阻塞I/O，该模式还有着相对较高的编程复杂度。

8.2.3 解决方案：通过前摄式操作实现的并发

当OS平台支持异步操作时，一种高效而方便的实现高性能Web服务器的方法是使用*前摄式事件分派*。使用前摄式事件分派模型设计的Web服务器通过一或多个线程控制来处理异步操作的完成。这样，*通过集成完成事件多路分离（completion event demultiplexing）和事件处理器分派，前摄器模式简化了异步的Web服务器*。

异步的Web服务器将这样来利用前摄器模式：首先让Web服务器向OS发出异步操作，并将回调方法登记到Completion Dispatcher（完成分派器），后者将在操作完成时通知Web服务器。于是OS代表Web服务器执行操作，并随即在一个周知的地方将结果排队。Completion Dispatcher负责使完成通知出队，并执行适当的、含有应用特有的Web服务器代码的回调。

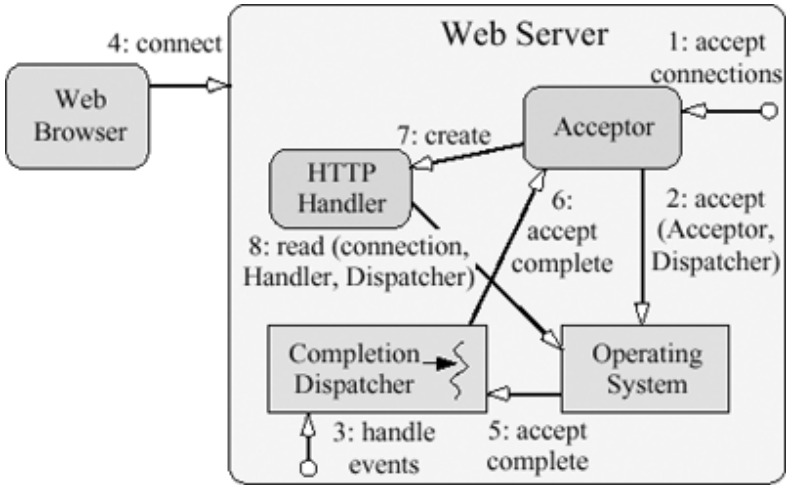

图8-5 客户连接到基于前摄器的Web服务器

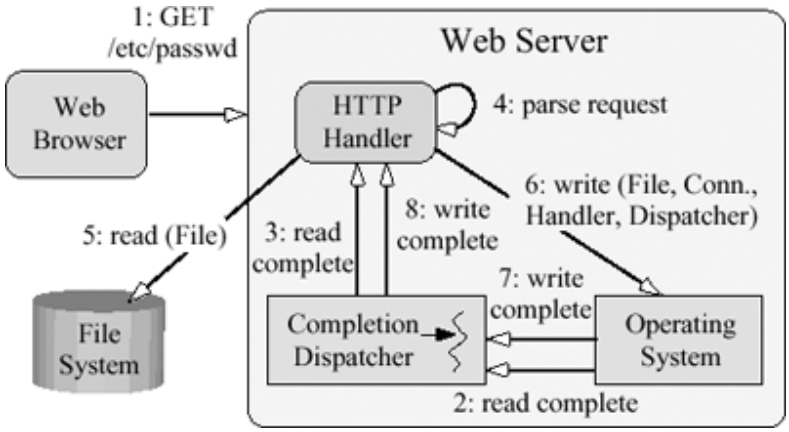

图8-6 客户发送请求给基于前摄器的Web服务器

图8-5和图8-6显示使用前摄式事件分派设计的Web服务器怎样在一或多个线程中并发地处理多个客户。图8-5显示当客户连接到Web服务器时所采取的一系列步骤。

1. Web服务器指示Acceptor发起异步接受；
2. 接受器通过OS发起异步接受，将其自身作为Completion Handler和Completion Dispatcher的引用传递；并将用于在异步接受完成时通知Acceptor；
3. Web服务器调用Completion Dispatcher的事件循环；
4. 客户连接到Web服务器；
5. 当异步接受操作完成时，操作系统通知Completion Dispatcher；

6. Completion Dispatcher通知接受器；
7. Acceptor创建HTTP Handler；
8. HTTP Handler发起异步操作，以读取来自客户的请求数据，并将其自身作为Completion Handler和Completion Dispatcher的引用传递；并将用于在异步读取完成时通知HTTP Handler。

图8-6 显示前摄式Web服务器为服务HTTP GET请求所采取的步骤。这些步骤解释如下：

1. 客户发送HTTP GET请求；
2. 读取操作完成，操作系统通知Completion Dispatcher；
3. Completion Dispatcher通知HTTP Handler（步骤2和3将重复直到整个请求被接收）；
4. HTTP Handler解析请求；
5. HTTP Handler同步地读取所请求的文件；
6. HTTP Handler发起异步操作，以把文件数据写到客户连接，并将其自身作为Completion Handler和Completion Dispatcher的引用传递；并将用于在异步写入完成时通知HTTP Handler。
7. 当写操作完成时，操作系统通知Completion Dispatcher；
8. 随后Completion Dispatcher通知Completion Handler（步骤6-8将重复直到文件被完全递送）。

8.8中有一个将前摄式事件分派模型应用于Web服务器的C++代码例子。

使用前摄器模式的主要优点是可以启动多个并发操作，并可并行运行，而不要求应用必须拥有多个线程。操作被应用异步地启动，它们在OS的I/O子系统中运行直到完成。发起操作的线程现在可以服务另外的请求了。

例如，在上面的例子中，Completion Dispatcher可以是单线程的。当HTTP请求到达时，单个Completion Dispatcher线程解析请求，读取文件，并发送响应给客户。因为响应是被异步发送的，多个响应就有可能同时被发送。而且，同步的文件读取可以被异步的文件读取取代，以进一步增加并发的潜力。如果文件读取是被异步完成的，HTTP Handler所执行的唯一的同步操作就只剩下了HTTP协议请求解析。

前摄式模型的主要缺点是编程逻辑至少和反应式模型一样复杂。而且，前摄器模式可能会难以调试，因为异步操作常常有着不可预测和不可重复的执行序列，这就使分析和调试复杂化了。8.7描述怎样应用其他模式（比如异步完成令牌[8]）来简化异步应用编程模型。

8.3 适用性

当具有以下一项或多项条件时使用前摄器模式：

- 应用需要执行一个或多个不阻塞调用线程的异步操作；
- 当异步操作完成时应用必须被通知；
- 应用需要独立于它的I/O模型改变它的并发策略；
- 通过使依赖于应用的逻辑与应用无关的底层构造去耦合，应用将从中获益；
- 当使用多线程方法或反应式分派方法时，应用的执行将很低效，或是不能满足性能需求。

8.4 结构和参与者

在图8-7中使用OMT表示法演示了前摄器模式的结构。

前摄器模式中的关键参与者包括：

前摄发起器（Proactive Initiator。Web服务器应用的主线程）：

- Proactive Initiator是应用中任何发起Asynchronous Operation（异步操作）的实体。它将Completion Handler和Completion Dispatcher登记到Asynchronous Operation Processor（异步操作处理器），此处理器在操作完成时通知前摄发起器。

完成处理器（Completion Handler。Acceptor和HTTP Handler）：

- 前摄器模式将应用所实现的Completion Handler接口用于Asynchronous Operation完成通知。

异步操作（Asynchronous Operation。Async_Read、Async_Write和Async_Accept方法）：

- Asynchronous Operation被用于代表应用执行请求（比如I/O和定时器操作）。当应用调用Asynchronous Operation时，操作的执行*没有*借用应用的线程控制。因此，从应用的角度来看，操作是被*异步地*执行的。当Asynchronous Operation完成时，Asynchronous Operation Processor将应用通知委托给Completion Dispatcher。

异步操作处理器（Asynchronous Operation Processor。操作系统）：

- Asynchronous Operation是由Asynchronous Operation Processor来运行直至完成的。该组件通常由OS实现。

完成分派器（Completion Dispatcher。Notification Queue）：

- Completion Dispatcher负责在Asynchronous Operation完成时回调应用的Completion Handler。当Asynchronous Operation Processor完成异步发起的操作时，Completion Dispatcher代表应用执行应用回调。

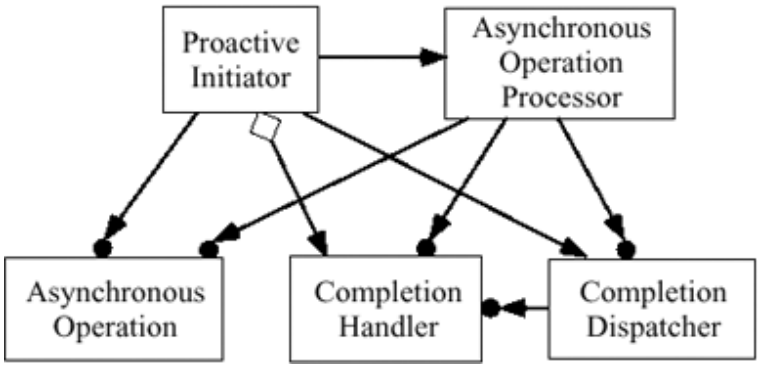

图8-7 前摄器模式中的参与者

8.5 协作

有若干良好定义的步骤被用于所有Asynchronous Operation。在高水平的抽象上，应用异步地发起操作，并在操作完成时被通知。图8-8显示在模式参与者之间必定发生的下列交互：

1. **前摄发起器发起操作：**为执行异步操作，应用在Asynchronous Operation Processor上发起操作。例如，Web服务器可能要求OS在网络上使用特定的socket连接传输文件。要请求这样的操作，Web服务器必须指定要使用哪一个文件和网络连接。而且，Web服务器必须指定（1）当操作完成时通知哪一个Completion Handler，以及（2）一旦文件被传输，哪一个Completion Dispatcher应该执行回调。
2. **异步操作处理器执行操作：**当应用在Asynchronous Operation Processor上调用操作时，它相对于其他应用操作异步地运行这些操作。现代操作系统（比如Solaris和Windows NT）在内核中提供异步的I/O子系统。
3. **异步操作处理器通知完成分派器：**当操作完成时，Asynchronous Operation Processor取得在操作被发起时指定的Completion Handler和Completion Dispatcher。随后Asynchronous Operation Processor将Asynchronous Operation的结果和Completion Handler传递给Completion Dispatcher，以用于回调。例如，如果文件已被异步传输，Asynchronous Operation Processor可以报告完成状态（比如成功或失败），以及写入网络连接的字节数。
4. **完成分派器通知应用：**Completion Dispatcher在Completion Handler上调用完成挂钩，将由应用指定的任何完成数据传递给它。例如，如果异步读取完成，通常一个指向新到达数据的指针将会被传递给Completion Handler。

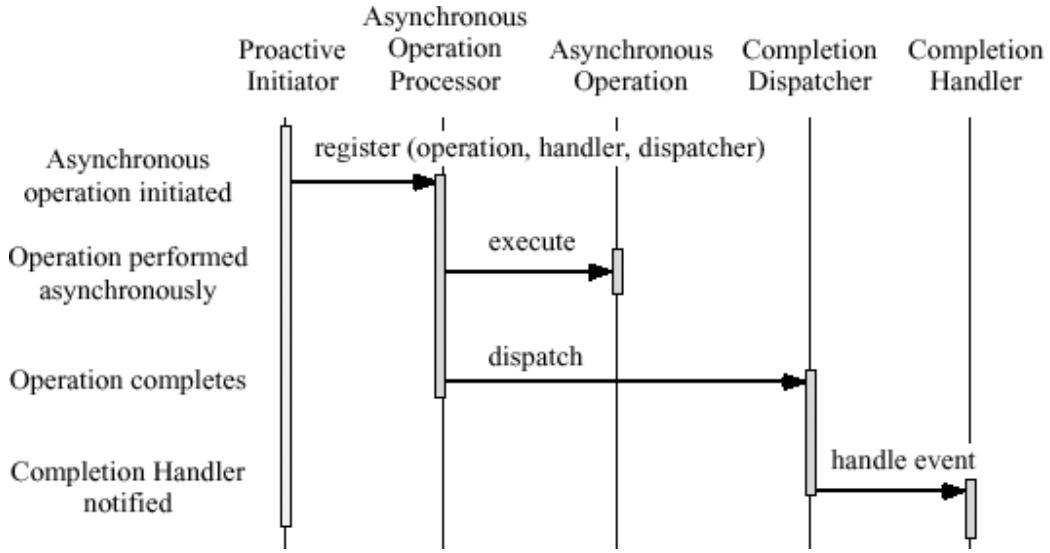

图8-8 前摄器模式的交互图

8.6 效果

这一部分详述使用前摄器模式的效果。

8.6.1 好处

前摄器模式提供以下好处：

增强事务分离：前摄器模式使应用无关的异步机制与应用特有的功能去耦合。应用无关的机制成为可复用组件，知道怎样多路分离与Asynchronous Operation相关联的完成事件，并分派适当的由Completion Handler定义的回调方法。同样地，应用特有的功能知道怎样执行特定类型的服务（比如HTTP处理）。

改善应用逻辑可移植性：通过允许接口独立于执行事件多路分离的底层OS调用而复用，它改善了应用的可移植性。这些系统调用检测并报告可能同时发生在多个事件源之上的事件。事件源可以是I/O端口、定时器、同步对象、信号，等等。在实时POSIX平台上，异步I/O函数由aio API族[9]提供。在Windows NT中，I/O完成端口和重叠式（overlapped）I/O被用于实现异步I/O[10]。

完成分派器封装了并发机制：使Completion Dispatcher与Asynchronous Operation Processor去耦合的一个好处是应用可以通过多种并发策略来配置Completion Dispatcher，而不会影响其他参与者。如8.7所讨论的，Completion Dispatcher可被配置使用包括单线程和线程池方案在内的若干并发策略。

线程策略被与并发策略去耦合：因为Asynchronous Operation Processor代表Proactive Initiator完成可能长时间运行的操作，应用不会被迫派生线程来增加并发。这使得应用可以独立于它的线程策略改变它的并发策略。例如，Web服务器可能只想每个CPU有一个线程，但又想同时服务更多数目的客户。

提高性能：多线程操作系统执行上下文切换，以在多个线程控制中轮换。虽然执行一次上下文切换的时间保持相当的恒定，如果OS上下文要切换到空闲线程的话，在大量线程间轮换的总时间可以显著地降低应用性能。例如，线程可以轮询OS以查看完成状态，而这是低效率的。通过只激活那些有事件要处理的合理的线程控制，前摄器模式能够避免上下文切换的代价。例如，如果没有待处理的GET请求，Web服务器不需要启用HTTP Handler。

应用同步的简化：只要Completion Handler不派生另外的线程控制，可以不考虑、或只考虑少许同步问题而编写应用逻辑。Completion Handler可被编写为就好像它们存在于一个传统的单线程环境中一样。例如，Web服务器的HTTP GET处理器可以通过Async Read操作（比如Windows NT TransmitFile函数[1]）来访问磁盘。

8.6.2 缺点

前摄器模式有以下缺点：

难以调试：以前摄器模式编写的应用可能难以调试，因为反向的控制流在构架基础结构和应用特有的处理器上的回调方法之间来回振荡。这增加了在调试器中对构架的运行时行为的“单步跟踪”的困难度，因为应用开发者可能不了解或不能获得构架的代码。这与试图调试使用LEX和YACC编写的编译器的词法分析器和解析器时所遇到的问题是类似的。在这些应用中，当线程控制是在用户定义的动作例程中时，调试是相当直接的。但是一旦线程控制返回到所生成的有限确定自动机（Deterministic Finite Automate，DFA）骨架时，就很难跟住程序逻辑了。

调度和控制未完成操作：Proactive Initiator可能没有对Asynchronous Operation的执行顺序的控制。因此，Asynchronous Operation Processor必须被小心设计，以支持Asynchronous Operation的优先级和取消处理。

8.7 实现

前摄器模式可以通过许多方式实现。这一部分讨论实现前摄器模式所涉及的步骤。

8.7.1 实现异步操作处理器

实现前摄器模式的第一步是构建Asynchronous Operation Processor。该组件负责代表应用异步地执行操作。因此，它的两项主要责任是输出Asynchronous Operation API和实现Asynchronous Operation Engine以完成工作。

8.7.1.1 定义异步操作API

Asynchronous Operation Processor必须提供API、允许应用请求Asynchronous Operation。在设计这些API时有若干压力需要考虑：

可移植性：此API不应约束应用或它的Proactive Initiator使用特定的平台。

灵活性：常常，异步API可以为许多类型的操作共享。例如，异步I/O操作常常被用于在多种介质（比如网络和文件）上执行I/O。设计支持这样的复用的API可能是有益的。

回调：当操作被调用时，Proactive Initiator必须登记回调。实现回调的一种常用方法是让调用对象（客户）输出接口、让调用者知道（服务器）。因此，Proactive Initiator必须通知Asynchronous Operation Processor，当操作完成时，哪一个Completion Handler应被回调。

完成分派器：因为应用可以使用多个Completion Dispatcher，Proactive Initiator还必须指示由哪一个Completion Dispatcher来执行回调。

给定所有这些问题，考虑下面的用于异步读写的API。Asynch_Stream类是用于发起异步读写的工厂。一旦构造，可以使用此类来启动多个异步读写。当异步读取完成时，Asynch_Stream::Read_Result将通过Completion_Handler上的handler_read回调方法被回传给handler。类似地，当异步写入完成时，Asynch_Stream::Write_Result将通过Completion_Handler上的handler_write回调方法被回传给handler。

```
class Asynch_Stream

// = TITLE

// A Factory for initiating reads

// and writes asynchronously.

{

// Initializes the factory with information
```

```
// which will be used with each asynchronous
// call. <handler> is notified when the
// operation completes. The asynchronous
// operations are performed on the <handle>
// and the results of the operations are
// sent to the <Completion_Dispatcher>.
Asynch_Stream (Completion_Handler &handler,
                                    HANDLE handle,
                                    Completion_Dispatcher *);

// This starts off an asynchronous read.
// Upto <bytes_to_read> will be read and
// stored in the <message_block>.
int read (Message_Block &message_block,
                            u_long bytes_to_read,
                      const void *act = 0);

// This starts off an asynchronous write.
// Upto <bytes_to_write> will be written
// from the <message_block>.
int write (Message_Block &message_block,
                            u_long bytes_to_write,
                      const void *act = 0);

...
};
```

8.7.1.2 实现异步操作引擎

Asynchronous Operation Processor必须含有异步执行操作的机制。换句话说，当应用线程调用Asynchronous Operation时，必须不借用应用的线程控制而执行此操作。幸好，现代操作系统提供了用于Asynchronous Operation的机制（例如，POSIX 异步I/O和WinNT重叠式I/O）。在这样的情况下，实现模式的这一部分只需要简单地将平台API映射到上面描述的Asynchronous Operation API。

如果OS平台不提供对Asynchronous Operation的支持，有若干实现技术可用于构建Asynchronous Operation Engine。或许最为直观的解决方案是使用专用线程来为应用执行Asynchronous Operation。要实现线程化的Asynchronous Operation Engine，有三个主要步骤：

1. **操作调用：**因为操作将在与进行调用的应用线程不同的线程控制中执行，必定会发生某种类型的线程同步。一种方法是为每个操作派生一个线程。更为常用的方法是为Asynchronous Operation Processor而管理一个专用线程池。该方法可能需要应用线程在继续进行其他应用计算之前将操作请求排队。
2. **操作执行：**既然操作将在专用线程中执行，所以它可以执行“阻塞”操作，而不会直接阻碍应用的进展。例如，在提供异步I/O读取机制时，专用线程可以在从socket或文件句柄中读时阻塞。
3. **操作完成：**当操作完成时，应用必须被通知到。特别是，专用线程必须将应用特有的通知委托给Completion Dispatcher。这要求在线程间进行另外的同步。

8.7.2 实现完成分派器

当Completion Dispatcher从Asynchronous Operation Processor接收到操作完成通知时，它会回调与应用对象相关联的Completion Handler。实现Completion Dispatcher涉及两个问题：（1）实现回调以及（2）定义用于执行回调的并发策略。

8.7.2.1 实现回调

Completion Dispatcher必须实现一种机制，Completion Handler通过它被调用。这要求Proactive Initiator在发起操作时指定一个回调。下面是常用的回调可选方案：

回调类：Completion Handler输出接口、让Completion Dispatcher知道。当操作完成时，Completion Dispatcher回调此接口中的方法，并将已完成操作的有关信息传递给它（比如从网络连接中读取的字节数）。

函数指针：Completion Dispatcher通过回调函数指针来调用Completion Handler。该方法有效地打破了Completion Dispatcher和Completion Handler之间的知识依赖。这有两个好处：

1. Completion Handler不会被迫输出特定的接口；以及
2. 在Completion Dispatcher和Completion Handler之间不需要有编译时依赖。

会合点：Proactive Initiator可以设立事件对象或条件变量，用作Completion Dispatcher和Completion Handler之间的会合点。这在Completion Handler是Proactive Initiator时最为常见。在Asynchronous Operation运行至完成的同时，Completion Handler处理其他的活动。Completion Handler将在会合点周期性地检查完成状态。

8.7.2.2 定义完成分派器并发策略

当操作完成时，Asynchronous Operation Processor将会通知Completion Dispatcher。在这时，Completion Dispatcher可以利用下面的并发策略中的一种来执行应用回调：

动态线程分派：Completion Dispatcher可为每个Completion Handler动态分配一个线程。动态线程分派可通过大多数多线程操作系统来实现。在有些平台上，由于创建和销毁线程资源的开销，这可能是所列出的Completion Dispatcher实现技术中最为低效的一种，

后反应式分派（Post-reactive dispatching）：Completion Dispatcher可以发信号给Proactive Initiation所设立的事件对象或条件变量。尽管轮询和派生阻塞在事件对象上的子线程都是可选的方案，最为高效的后反应式分派方法是将事件登记到Reactor。后反应式分派可以通过POSIX实时环境中的aio_suspend和Win32环境中的WaitForMultipleObjects来实现。

Call-through**分派：**来自Asynchronous Operation Processor的线程控制可被Completion Dispatcher借用，以执行Completion Handler。这种“周期偷取”策略可以通过减少空闲线程的影响范围来提高性能。在一些老操作系统会将上下文切换到空闲线程、又只是从它们切换出去的情况下，这种方法有着收回“失去的”时间的巨大潜力。

Call-through分派在Windows NT中可以使用ReadFileEx和WriteFileEx Win32函数来实现。例如，线程控制可以使用这些调用来等待信号量被置位。当它等待时，线程通知OS它进入了一种称为“可报警等待状态”（alterable wait state）的特殊状态。在这时，OS可以占有对等待中的线程控制的栈和相关资源的控制，以执行Completion Handler。

线程池分派：由Completion Dispatcher拥有的线程池可被用于Completion Handler的执行。在池中的每个线程控制已被动态地分配到可用的CPU。线程池分派可通过Windows NT的I/O完成端口来实现。

在考虑上面描述的Completion Dispatcher技术的适用性时，考虑表8-1中所示的OS环境和物理硬件的可能组合：

线程模型	**系统类型**	
	单处理器	多处理器
单线程	A	B
多线程	C	D

表8-1 Completion Dispatcher并发策略

如果你的OS只支持同步I/O，那就参见反应堆模式[5]。但是，大多数现代操作系统都支持某种类型的异步I/O。

在表8-1的A和B组合中，假定你不等待任何信号量或互斥体，后反应方式的异步I/O很可能是最好的。否则，Call-through实现或许更能回应你的问题。在C组合中，使用Call-through方法。在D组合中，使用线程池方法。在实践中，系统化的经验测量对于选择最为合适的可选方案来说是必需的。

8.7.3 实现完成处理器

Completion Handler的实现带来以下考虑。

8.7.3.1 状态完整性

Completion Handler可能需要维护关于特定请求的状态信息。例如，OS可以通知Web服务器，只有一部分文件已被写到网络通信端口。作为结果，Completion Handler可能需要重新发出请求，直到文件被完全写出，或连接变得无效。因此，它必须知道原先指定的文件，还剩多少字节要写，以及在前一个请求开始时文件指针的位置。

没有隐含的限制来阻止Proactive Initiator将多个Asynchronous Operation请求分配给单个Completion Handler。因此，Completion Handler必须在完成通知链中——“系上”请求特有的状态信息。为完成此工作，Completion Handler可以利用异步完成令牌（Asynchronous Completion Token）模式[8]。

8.7.3.2 资源管理

与在任何多线程环境中一样，使用前摄器模式的Completion Handler还是要由它自己来确保对共享资源的访问是线程安全的。但是，Completion Handler不能跨越多个完成通知持有共享资源。否则，就有发生“用餐哲学家问题”的危险[11]。

该问题在于一个合理的线程控制永久等待一个信号量被置位时所产生的死锁。通过设想一个由一群哲学家出席的宴会可以演示这一问题。用餐者围绕一个圆桌就座，在每个哲学家之间只有一支筷子。当哲学家觉得饥饿时，他必须获取在他左边和在他右边的筷子才能用餐。一旦哲学家获得一支筷子，不到吃饱他们就不会放下它。如果所有哲学家都拿起在他们右边的筷子，就会发生死锁，因为他们将永远也不可能拿到左边的筷子。

8.7.3.3 占先式策略（Preemptive Policy）

Completion Dispatcher类型决定在执行时一个Completion Handler是否可占先。当与动态线程和线程池分派器相连时，Completion Handler自然可占先。但是，当与后反应式Completion Dispatcher

相连时，Completion Handler并没有对其他Completion Handler的占先权。当由Call-through分派器驱动时，Completion Handler相对于在可报警等待状态的线程控制也没有占先权。

一般而言，处理器不应该执行持续时间长的同步操作，除非使用了多个完成线程，因为应用的总体响应性将会被显著地降低。这样的危险可以通过增强的编程训练来降低。例如，所有Completion Handler被要求用作Proactive Initiator，而不是去执行同步操作。

8.8 示例代码

这一部分显示怎样使用前摄器模式来开发Web服务器。该例子基于ACE构架[4]中的前摄器实现。

当客户连接到Web服务器时，HTTP_Handler的open方法被调用。于是服务器就通过在Asynchronous Operation完成时回调的对象（在此例中是this指针）、用于传输数据的网络连接，以及一旦操作完成时使用的Completion Dispatcher（proactor_）来初始化异步I/O对象。随后读操作异步地启动，而服务器返回事件循环。

当Async read操作完成时，分派器回调HTTP_Handler::handle_read_stream。如果有足够的数据，客户请求就被解析。如果整个客户请求还未完全到达，另一个读操作就会被异步地发起。

在对GET请求的响应中，服务器对所请求文件进行内存映射，并将文件数据异步地写往客户。当写操作完成时，分派器回调HTTP_Handler::handle_write_stream，从而释放动态分配的资源。

附录中含有两个其他的代码实例，使用同步的线程模型和同步的（非阻塞）反应式模型实现Web服务器。

```
class HTTP_Handler

: public Proactor::Event_Handler

// = TITLE

// Implements the HTTP protocol

// (asynchronous version).

//

// = PATTERN PARTICIPANTS

// Proactive Initiator = HTTP_Handler

// Asynch Op = Network I/O

// Asynch Op Processor = OS

// Completion Dispatcher = Proactor

// Completion Handler = HTPP_Handler

{

public:

void open (Socket_Stream *client)

{
```

```
        // Initialize state for request
        request_.state_ = INCOMPLETE;

        // Store reference to client.
        client_ = client;

        // Initialize asynch read stream
        stream_.open (*this, client_->handle (), proactor_);

        // Start read asynchronously.
        stream_.read (request_.buffer (),
        request_.buffer_size ());
}

// This is called by the Proactor
// when the asynch read completes
void handle_read_stream(u_long bytes_transferred)
{
        if (request_.enough_data(bytes_transferred))
                parse_request ();
        else
                // Start reading asynchronously.
                stream_.read (request_.buffer (),

        request_.buffer_size ());
}

void parse_request (void)
{
        // Switch on the HTTP command type.
        switch (request_.command ())
        {
        // Client is requesting a file.
                case HTTP_Request::GET:
                // Memory map the requested file.
                file_.map (request_.filename ());
```

```
                // Start writing asynchronously.
                stream_.write (file_.buffer (), file_.buffer_size ());
                break;

                // Client is storing a file
        // at the server.
        case HTTP_Request::PUT:
                // ...
        }
}

void handle_write_stream(u_long bytes_transferred)
{
        if (file_.enough_data(bytes_transferred))
                // Success....
        else
                // Start another asynchronous write
                stream_.write (file_.buffer (), file_.buffer_size ());
        }

private:
// Set at initialization.
Proactor *proactor_;

// Memory-mapped file_;
Mem_Map file_;

// Socket endpoint.
Socket_Stream *client_;

// HTTP Request holder
HTTP_Request request_;

// Used for Asynch I/O
Asynch_Stream stream_;
```

```
};
```

8.9 已知应用

下面是一些被广泛记载的前摄器的使用：

Windows NT中的I/O完成端口：Windows NT操作系统实现了前摄器模式。Windows NT支持多种Asynchronous Operation，比如接受新网络连接、读写文件和socket，以及通过网络连接传输文件。操作系统就是Asynchronous Operation Processor。操作结果在I/O完成端口（它扮演Completion Dispatcher的角色）上排队。

异步I/O操作的UNIX AIO族：在有些实时POSIX平台上，前摄器模式是由aio API族[9]来实现的。这些OS特性非常类似于上面描述的Windows NT的特性。一个区别是UNIX信号可用于实现真正异步的Completion Dispatcher（Windows NT API不是真正异步的）。

ACE Proactor：ACE自适配通信环境 [4]实现了前摄器组件，它封装Windows NT上的I/O完成端口，以及POSIX平台上的aio API。ACE前摄器抽象提供Windows NT所支持的标准C API的OO接口。这一实现的源码可从ACE网站http://www.cs.wustl.edu/~schmidt/ACE.html获取。

Windows NT中的异步过程调用（Asynchronous Procedure Call）：有些系统（比如Windows NT）支持异步过程调用（APC）。APC是在特定线程的上下文中异步执行的函数。当APC被排队到线程时，系统发出软件中断。下一次线程被调度时，它将运行该APC。操作系统所发出的APC被称为*内核模式*APC。应用所发出的APC被称为*用户模式*APC。

8.10 相关模式

图8-9演示与前摄器相关的模式。

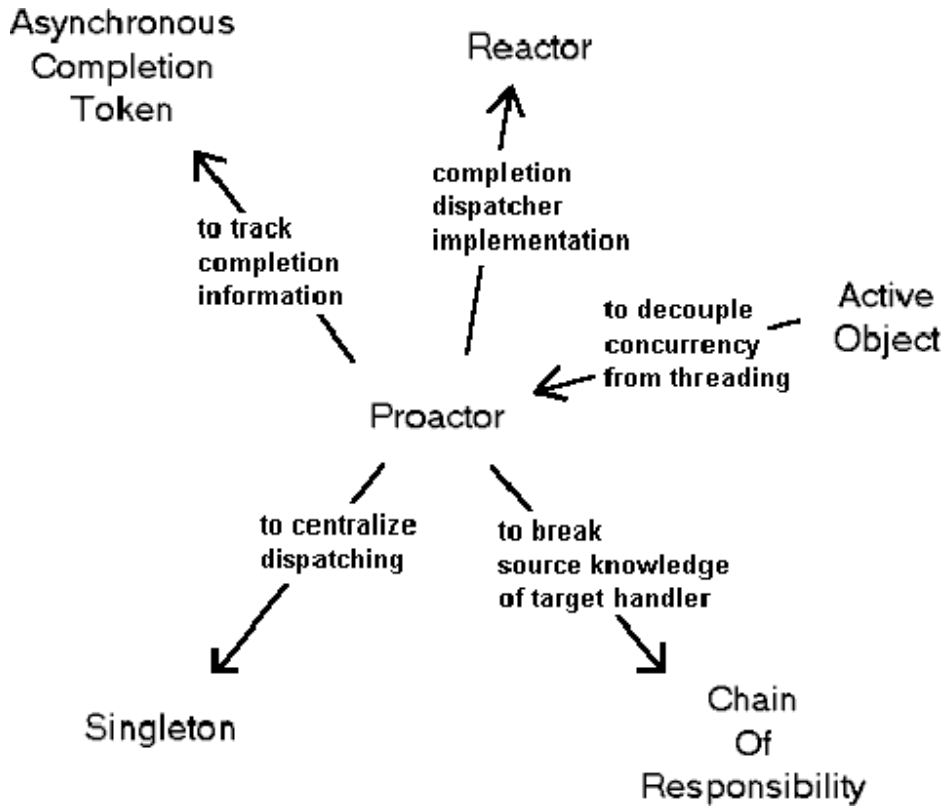

图8-9 前摄器模式的相关模式

异步完成令牌（ACT）模式[8]通常与前摄器模式结合使用。当Asynchronous Operation完成时，应用可能需要比简单的通知更多的信息来适当地处理事件。异步完成令牌模式允许应用将状态高效地与Asynchronous Operation的完成相关联。

前摄器模式还与观察者（Observer）模式[12]（在其中，当单个主题变动时，相关对象也会自动更新）有关。在前摄器模式中，当来自多个来源的事件发生时，处理器被自动地通知。一般而言，前摄器模式被用于异步地将多个输入源多路分离给与它们相关联的事件处理器，而观察者通常仅与单个事件源相关联。

前摄器模式可被认为是同步反应堆模式[5]的一种*异步的*变体。反应堆模式负责多个事件处理器的多路分离和分派；它们在可以*同步地发起*操作而不会阻塞时被触发。相反，前摄器模式也支持多个事件处理器的多路分离和分派，但它们是被*异步事件的完成*触发的。

主动对象（Active Object）模式[13]使方法执行与方法调用去耦合。前摄器模式也是类似的，因为Asynchronous Operation Processor代表应用的Proactive Initiator来执行操作。就是说，两种模式都可用于实现Asynchronous Operation。前摄器模式常常用于替代主动对象模式，以使系统并发策略与线程模型去耦合。

前摄器可被实现为单体（Singleton）[12]。这对于在异步应用中，将事件多路分离和完成分派集中到单一的地方来说是有用的。

责任链（Chain of Responsibility，COR）模式[12]使事件处理器与事件源去耦合。在Proactive Initiator与Completion Handler的隔离上，前摄器模式也是类似的。但是，在COR中，事件源预先不知道哪一个处理器将被执行（如果有的话）。在前摄器中，Proactive Initiator完全知道目标处理器。但是，通过建立一个Completion Handler（它是由外部工厂动态配置的责任链的入口），这两种模式可被结合在一起：。

8.11 结束语

前摄器模式包含了一种强大的设计范式，支持高性能并发应用的高效而灵活的事件分派策略。前摄器模式提供并发执行操作的性能助益，而又不强迫开发者使用同步多线程或反应式编程。

参考文献

[1] J. Hu, I. Pyarali, and D. C. Schmidt, "Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-speed Networks," in *Proceedings of the 2nd Global Internet Conference*, IEEE, November 1997.

[2] J. Hu, I. Pyarali, and D. C. Schmidt, "Applying the Proactor Pattern to High-Performance Web Servers," in *Proceedings of the 10th International Conference on Parallel and Distributed Computing and Systems*, IASTED, Oct. 1998.

[3] J. C. Mogul, "The Case for Persistent-connection HTTP," in *Proceedings of ACMSIGCOMM '95 Conference in Computer Communication Review*, (Boston, MA, USA), pp. 299－314, ACM Press, August 1995.

[4] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6th USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[5] 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.

[6] 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.

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

[8] 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.

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

[10] *Microsoft Developers Studio, Version 4.2 - Software Development Kit*, 1996.

[11] E. W. Dijkstra, "Hierarchical Ordering of Sequential Processes," *Acta Informatica*, vol. 1, no. 2, pp. 115－138, 1971.

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

[13] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Proceedings of the 2nd Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), pp. 1－7, September 1995.

附录A 可选实现

本附录概述用于开发前摄器模式的可选实现的代码。下面，我们检查使用多线程的同步I/O和使用单线程的反应式I/O。

A.1 多个同步线程

下面的代码显示怎样使用线程池同步I/O来开发Web服务器。当客户连接到服务器时，池中的一个线程接受连接，并调用HTTP_Handler中的open方法。随后服务器同步地从网络连接读取请求。当读操作完成时，客户请求随之被解析。在对GET请求的响应中，服务器对所请求文件进行内存映射，并将文件数据同步地写往客户。注意阻塞I/O是怎样使Web服务器能够遵循2.2.1中所概述的步骤的。

```
class HTTP_Handler
// = TITLE
// Implements the HTTP protocol
// (synchronous threaded version).
//
// = DESCRIPTION
// This class is called by a
// thread in the Thread Pool.
{
public:
void open (Socket_Stream *client)
{
       HTTP_Request request;

       // Store reference to client.
       client_ = client;

       // Synchronously read the HTTP request
       // from the network connection and
       // parse it.
       client_->recv (request);
       parse_request (request);
}

void parse_request (HTTP_Request &request)
{
       // Switch on the HTTP command type.
       switch (request.command ())
```

```
        {
        // Client is requesting a file.
        case HTTP_Request::GET:
                // Memory map the requested file.
                Mem_Map input_file;
                input_file.map (request.filename());

                // Synchronously send the file
                // to the client. Block until the
                // file is transferred.
                client_->send (input_file.data (),
                input_file.size ());
                break;

        // Client is storing a file at
        // the server.
        case HTTP_Request::PUT:
                // ...
        }
}

private:
// Socket endpoint.
Socket_Stream *client_;

// ...
};
```

A.2 单线程反应式事件分派

下面的代码显示怎样将反应堆模式用于开发Web服务器。当客户连接到服务器时，HTTP_Handler::open方法被调用。服务器登记I/O句柄和在网络句柄“读就绪“时回调的对象（在此例中是this指针）。然后服务器返回事件循环。

当请求数据到达服务器时，reactor_回调HTTP_Handler::handle_input方法。客户数据以非阻塞方式被读取。如果有足够的数据，客户请求就被解析。如果整个客户请求还没有到达，应用就返回反应堆事件循环。

在对GET请求的响应中，服务器对所请求的文件进行内存映射；并在反应堆上登记，以在网络连接变为“写就绪”时被通知。当向连接写入数据不会阻塞调用线程时，reactor_就回调HTTP_Handler::handler_output方法。当所有数据都已发送给客户时，网络连接被关闭。

```
class HTTP_Handler :

public Reactor::Event_Handler

// = TITLE

// Implements the HTTP protocol

// (synchronous reactive version).

//

// = DESCRIPTION

// The Event_Handler base class

// defines the hooks for

// handle_input()/handle_output().

//

// = PATTERN PARTICIPANTS

// Reactor = Reactor

// Event Handler = HTTP_Handler

{

public:

void open (Socket_Stream *client)

{

       // Initialize state for request

       request_.state_ = INCOMPLETE;


       // Store reference to client.

       client_ = client;


       // Register with the reactor for reading.

       reactor_->register_handler

                           (client_->handle (),

                           this,

                           Reactor::READ_MASK);
```

```
}

// This is called by the Reactor when
// we can read from the client handle.
void handle_input (void)
{
	int result = 0;

	// Non-blocking read from the network
	// connection.
	do
		result = request_.recv (client_->handle ());
	while (result != SOCKET_ERROR && request_.state_ == INCOMPLETE);

	// No more progress possible,
	// blocking will occur
	if (request_.state_ == INCOMPLETE && errno == EWOULDBLOCK)
		reactor_->register_handler
				(client_->handle (),
				this,
				Reactor::READ_MASK);
	else
		// We now have the entire request
		parse_request ();
}

void parse_request (void)
{
	// Switch on the HTTP command type.
	switch (request_.command ())
	{
	// Client is requesting a file.
	case HTTP_Request::GET:

	// Memory map the requested file.
	file_.map (request_.filename ());
```

```
        // Transfer the file using Reactive I/O.
        handle_output ();
        break;

        // Client is storing a file at
        // the server.
        case HTTP_Request::PUT:
                // ...
        }
}

void handle_output (void)
{
        // Asynchronously send the file
        // to the client.
        if (client_->send (file_.data (),
                           file_.size ())
                           == SOCKET_ERROR
                           && errno == EWOULDBLOCK)
                // Register with reactor...
        else
                // Close down and release resources.
                handle_close ();
}

private:
// Set at initialization.
Reactor *reactor_;

// Memory-mapped file_;
Mem_Map file_;

// Socket endpoint.
Socket_Stream *client_;
```

```
// HTTP Request holder.

HTTP_Request request_;

};
```