两种IO多路复用模式：Reactor和Proactor 基于事件驱动

一般的，IO多路复用机制都依赖于一个事件多路分离器。分离器可将事件源的IO事件分离出来，并分发到对应的read write事件处理器中。需要预先注册处理的事件和事件处理器（回调函数），事件分离器负责将请求传递给事件处理器。

Reactor采用的是同步IO

Proactor采用的是异步IO

在Reactor中，事件分离器负责等待文件描述符或socket为读写操作准备就绪，然后将就绪事件传递给对应的处理器，最后由处理器负责完成实际的读写工作。

而在Proactor模式中，处理器--或者兼任处理器的事件分离器，只负责发起异步读写操作。IO操作本身由操作系统来完成。传递给操作系统的参数需要包括用户定义的数据缓冲区地址和数据大小，操作系统才能从中得到写出操作所需数据，或写入从socket读到的数据。事件分离器捕获IO操作完成事件，然后将事件传递给对应处理器。比如，在windows上，处理器发起一个异步IO操作，再由事件分离器等待IOCompletion事件。典型的异步模式实现，都建立在操作系统支持异步API的基础之上，我们将这种实现称为“系统级”异步或“真”异步，因为应用程序完全依赖操作系统执行真正的IO工作。

**在Reactor中实现读：**

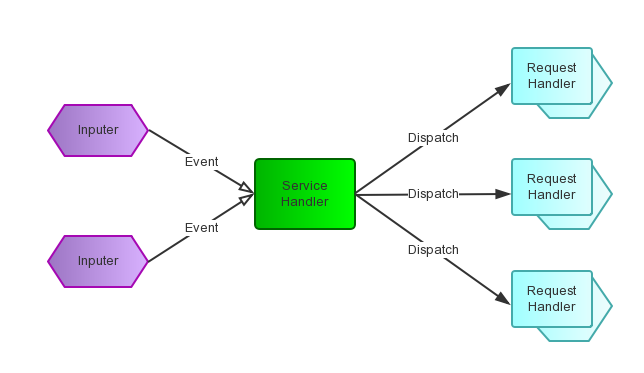
- 注册读就绪事件和相应的事件处理器  
- 事件分离器等待事件  
- 事件到来，激活分离器，分离器调用事件对应的处理器。  
- 事件处理器完成实际的读操作，处理读到的数据，注册新的事件，然后返还控制权。  
**在Proactor中实现读：**

- 处理器发起异步读操作（注意：操作系统必须支持异步IO）。在这种情况下，处理器无视IO就绪事件，它关注的是完成事件。  
- 事件分离器等待操作完成事件  
- 在分离器等待过程中，操作系统利用并行的内核线程执行实际的读操作，并将结果数据存入用户自定义缓冲区，最后通知事件分离器读操作完成。  
- 事件分离器呼唤处理器。  
- 事件处理器处理用户自定义缓冲区中的数据，然后启动一个新的异步操作，并将控制权返回事件分离器。

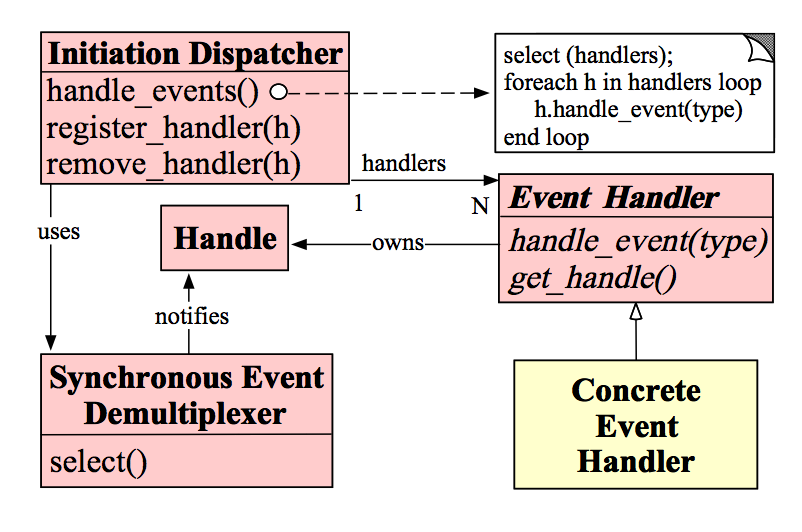
可以看出，两个模式的相同点，都是对某个IO事件的事件通知(即告诉某个模块，这个IO操作可以进行或已经完成)。在结构上，两者也有相同点：demultiplexor负责提交IO操作(异步)、查询设备是否可操作(同步)，然后当条件满足时，就回调handler；不同点在于，异步情况下(Proactor)，当回调handler时，表示IO操作已经完成；同步情况下(Reactor)，回调handler时，表示IO设备可以进行某个操作(can read or can write)。

## Reactor

Reactor模式是基于事件驱动，有一个或多个并发输入源，有一个servicehandler，有多个requesthandler，这个servicehandler会同步的将输入的请求多路复用的分发给相应的requesthandler。

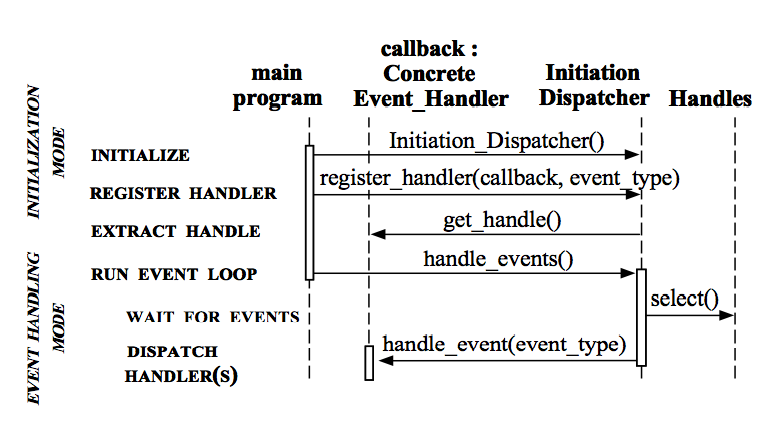


### Reactor模式结构



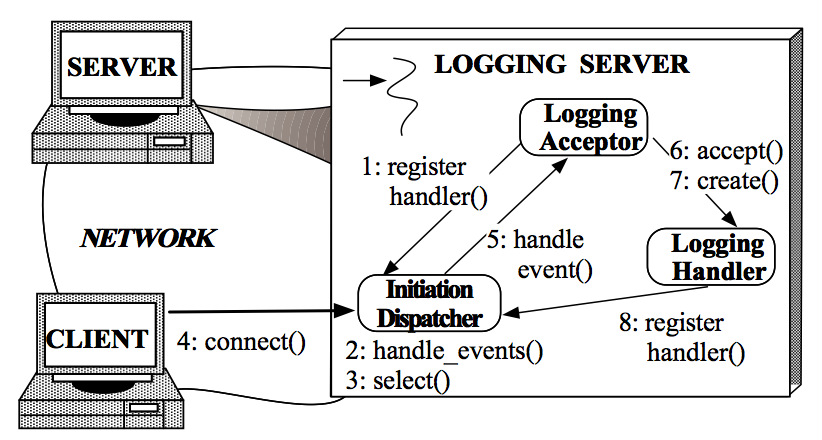
**Handle：**即操作系统中的句柄，是对资源在操作系统层面上的一种抽象，它可以是打开的文件、一个连接(Socket)、Timer等。由于Reactor模式一般使用在网络编程中，因而这里一般指Socket Handle，即一个网络连接（Connection，在Java NIO中的Channel）。这个Channel注册到Synchronous Event Demultiplexer中，以监听Handle中发生的事件，对ServerSocketChannnel可以是CONNECT事件，对SocketChannel可以是READ、WRITE、CLOSE事件等。  
**Synchronous Event Demultiplexer：**同步事件多路分发器，阻塞等待一系列的Handle中的事件到来，如果阻塞等待返回，即表示在返回的Handle中可以不阻塞的执行返回的事件类型。这个模块一般使用操作系统的select来实现。在Java NIO中用Selector来封装，当Selector.select()返回时，可以调用Selector的selectedKeys()方法获取Set<SelectionKey>，一个SelectionKey表达一个有事件发生的Channel以及该Channel上的事件类型。上图的“Synchronous Event Demultiplexer ---notifies--> Handle”的流程如果是对的，那内部实现应该是select()方法在事件到来后会先设置Handle的状态，然后返回。不了解内部实现机制，因而保留原图。  
**Initiation Dispatcher：**用于管理Event Handler，即EventHandler的容器，用以注册、移除EventHandler等；另外，它还作为Reactor模式的入口调用Synchronous Event Demultiplexer的select方法以阻塞等待事件返回，当阻塞等待返回时，根据事件发生的Handle将其分发给对应的Event Handler处理，即回调EventHandler中的handle\_event()方法。  
**Event Handler：**事件处理接口，定义事件处理方法：handle\_event()，以供InitiationDispatcher回调使用。  
**Concrete Event Handler：**事件EventHandler接口，实现特定事件处理逻辑。

### Reactor模式模块之间的交互



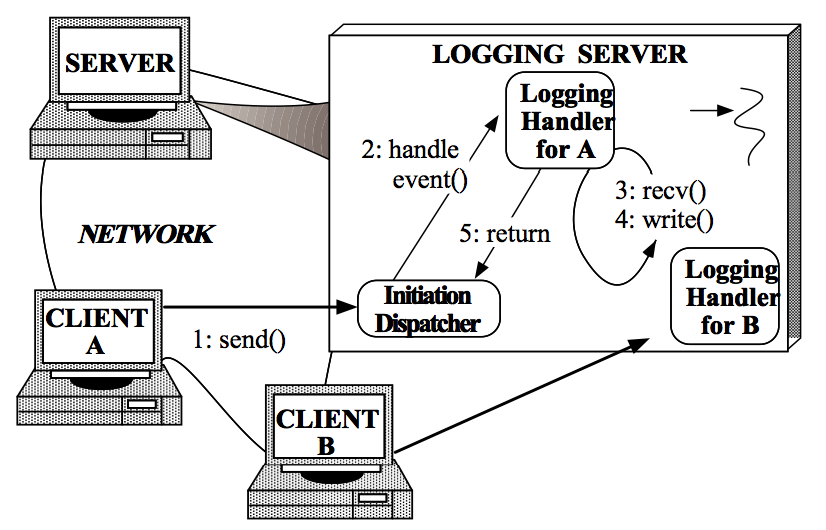
1. 初始化InitiationDispatcher，并初始化一个Handle到EventHandler的Map。  
   2. 注册EventHandler到InitiationDispatcher中，每个EventHandler包含对相应Handle的引用，从而建立Handle到EventHandler的映射（Map）。  
   3. 调用InitiationDispatcher的handle\_events()方法以启动Event Loop。在Event Loop中，调用select()方法（Synchronous Event Demultiplexer）阻塞等待Event发生。  
   4. 当某个或某些Handle的Event发生后，select()方法返回，InitiationDispatcher根据返回的Handle找到注册的EventHandler，并回调该EventHandler的handle\_events()方法。  
   5. 在EventHandler的handle\_events()方法中还可以向InitiationDispatcher中注册新的Eventhandler，比如对AcceptorEventHandler来，当有新的client连接时，它会产生新的EventHandler以处理新的连接，并注册到InitiationDispatcher中。

### Reactor模式实现



Logging Server中的Reactor模式实现分两个部分：Client连接到Logging Server和Client向Logging Server写Log

1. Logging Server注册LoggingAcceptor到InitiationDispatcher。  
   2. Logging Server调用InitiationDispatcher的handle\_events()方法启动。  
   3. InitiationDispatcher内部调用select()方法（Synchronous Event Demultiplexer），阻塞等待Client连接。  
   4. Client连接到Logging Server。  
   5. InitiationDisptcher中的select()方法返回，并通知LoggingAcceptor有新的连接到来。   
   6. LoggingAcceptor调用accept方法accept这个新连接。  
   7. LoggingAcceptor创建新的LoggingHandler。  
   8. 新的LoggingHandler注册到InitiationDispatcher中(同时也注册到Synchonous Event Demultiplexer中)，等待Client发起写log请求。



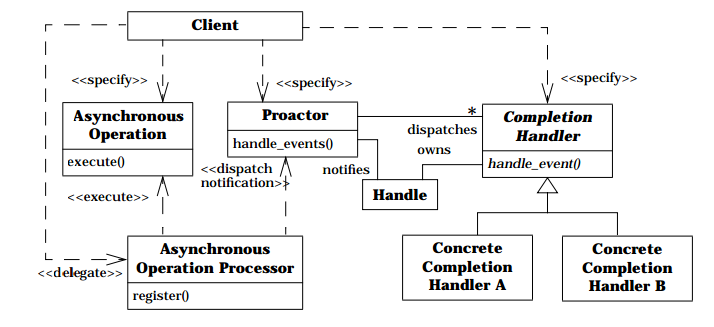
1. Client发送log到Logging server。  
   2. InitiationDispatcher监测到相应的Handle中有事件发生，返回阻塞等待，根据返回的Handle找到LoggingHandler，并回调LoggingHandler中的handle\_event()方法。  
   3. LoggingHandler中的handle\_event()方法中读取Handle中的log信息。  
   4. 将接收到的log写入到日志文件、数据库等设备中。  
   3.4步骤循环直到当前日志处理完成。  
   5. 返回到InitiationDispatcher等待下一次日志写请求。

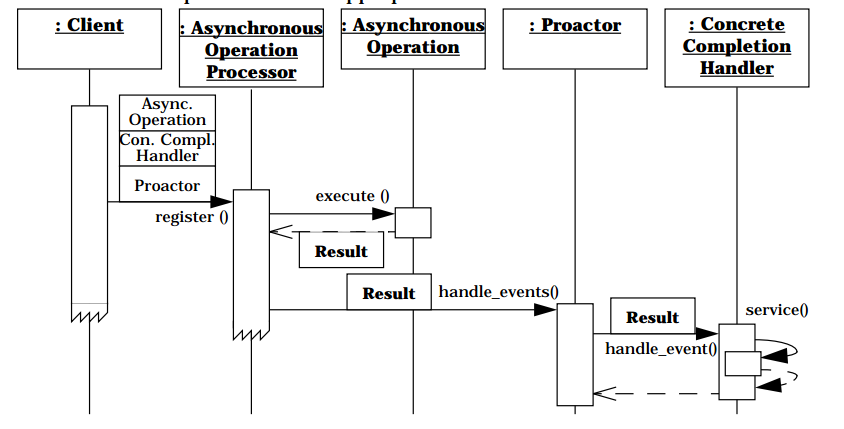
### Reactor模式缺点

Reactor模式的缺点貌似也是显而易见的：  
1. 相比传统的简单模型，Reactor增加了一定的复杂性，因而有一定的门槛，并且不易于调试。  
2. Reactor模式需要底层的Synchronous Event Demultiplexer支持，比如Java中的Selector支持，操作系统的select系统调用支持，如果要自己实现Synchronous Event Demultiplexer可能不会有那么高效。  
3. Reactor模式在IO读写数据时还是在同一个线程中实现的，即使使用多个Reactor机制的情况下，那些共享一个Reactor的Channel如果出现一个长时间的数据读写，会影响这个Reactor中其他Channel的相应时间，比如在大文件传输时，IO操作就会影响其他Client的相应时间，因而对这种操作，使用传统的Thread-Per-Connection或许是一个更好的选择，或则此时使用Proactor模式。

## Proactor模式

从上面 Reactor 模式中，发现服务端数据的接收和发送都占用了用户状态（还有一种内核态），这样服务器的处理操作就在数据的读写上阻塞花费了时间，节省这些时间的办法是借助操作系统的异步读写；异步读写在调用的时候可以传递回调函数或者回送信号，当异步操作完毕，内核会自动调用回调函数或者发送信号。Proactor 就是这么做的，所以很依赖操作系统。

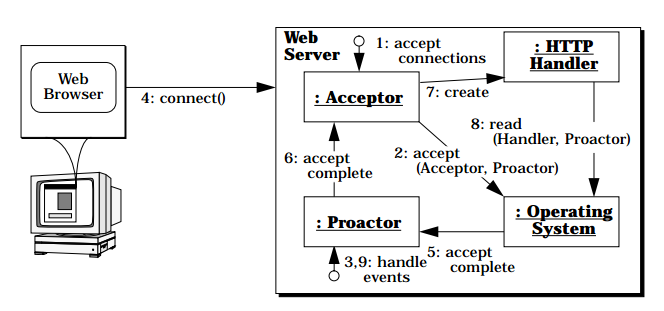


Proactor 的实现主要有三个部分：异步操作处理器，Proactor 和 事件处理函数。其中：

- 异步操作处理器，很依赖操作系统的异步处理机制，如若操作系统没有实现，我们可以自行模拟，即开专门的数据读写线程，数据读写完毕触发相应的时间（如果有注册的话）；

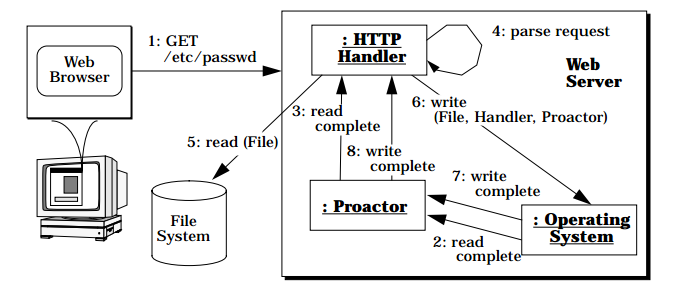
- Proactor，会接收异步操作的提醒，调用相应的事件处理函数，它有自己的 event loop；

- 事件处理函数，事件触发，执行操作；



* web 服务器指定（1）接收器，此接收器相当于服务器的客户端，它可以启动异步的 accept 操作；
* 接收器调用操作系统上的异步接收操作（2），并传递自己和 Proactor 的引用；异步接收操作结束后，前者用作事件处理函数，后者会回过头来分发事件；*注：传递 Proactor 是为了让操作系统通知正确的 Proactor，可能会存在多个 Proactor；传递接收器自己是为了在异步接收操作结束后 Proactor 能调用正确的事件处理函数，以下同理。*
* web 服务器调用 Proactor 的事件循环；（3）
* web 浏览器连接 web 服务器；（4）
* 异步接收操作结束后，操作系统产生事件（通过回调或者信号）并通知 Proactor（5），Proactor 收到后会调用相应的事件处理函数，即交由接收器处理；（6）
* 接收器生成 HTTP 处理器，执行操作；（7）
* HTTP 处理器解析事件，启动异步读操作（8），获取来自浏览器的 GET 请求。同样，HTTP 处理器传递自己和 Proactor 的引用；
* web 服务器的控制权交还回 Proactor 的事件循环。（9）

接收get请求后，会处理数据



* 浏览器发送（1）一个 HTTP GET 请求；
* 异步读操作结束后，操作系统会通知 Proactor，Proactor 分发给事件处理函数；（2，3）
* 事件处理器解析请求。（4）2-4 步骤会重复，指导所有的数据都接收为止；
* 事件处理器产生答复数据；（5）
* HTTP 处理器启动异步写操作（6），传输应答数据，同样的这里还会传递处理器自己和 Proactor；
* 异步写操作结束，操作系统通知 Proactor（7），Proactor 分发给事件处理函数（8）。6-8 步骤会重复直到所有的数据写完为止。至此，一个请求回复完成。

## Proactor和Reactor对比

**主动和被动**

以主动写为例：   
Reactor将handle放到select()，等待可写就绪，然后调用write()写入数据；写完处理后续逻辑；   
Proactor调用aoi\_write后立刻返回，由内核负责写操作，写完后调用相应的回调函数处理后续逻辑；

可以看出，Reactor被动的等待指示事件的到来并做出反应；它有一个等待的过程，做什么都要先放入到监听事件集合中等待handler可用时再进行操作；   
Proactor直接调用异步读写操作，调用完后立刻返回；

**实现**

Reactor实现了一个被动的事件分离和分发模型，服务等待请求事件的到来，再通过不受间断的同步处理事件，从而做出反应；

Proactor实现了一个主动的事件分离和分发模型；这种设计允许多个任务并发的执行，从而提高吞吐量；并可执行耗时长的任务（各个任务间互不影响）

**优点**

Reactor实现相对简单，对于耗时短的处理场景处理高效；   
操作系统可以在多个事件源上等待，并且避免了多线程编程相关的性能开销和编程复杂性；   
事件的串行化对应用是透明的，可以顺序的同步执行而不需要加锁；   
事务分离：将与应用无关的多路分解和分配机制和与应用相关的回调函数分离开来，

Proactor性能更高，能够处理耗时长的并发场景；

**缺点**

Reactor处理耗时长的操作会造成事件分发的阻塞，影响到后续事件的处理；

Proactor实现逻辑复杂；依赖操作系统对异步的支持，目前实现了纯异步操作的操作系统少，实现优秀的如windows IOCP，但由于其windows系统用于服务器的局限性，目前应用范围较小；而Unix/Linux系统对纯异步的支持有限，应用事件驱动的主流还是通过select/epoll来实现；

**适用场景**

Reactor：同时接收多个服务请求，并且依次同步的处理它们的事件驱动程序；   
Proactor：异步接收和同时处理多个服务请求的事件驱