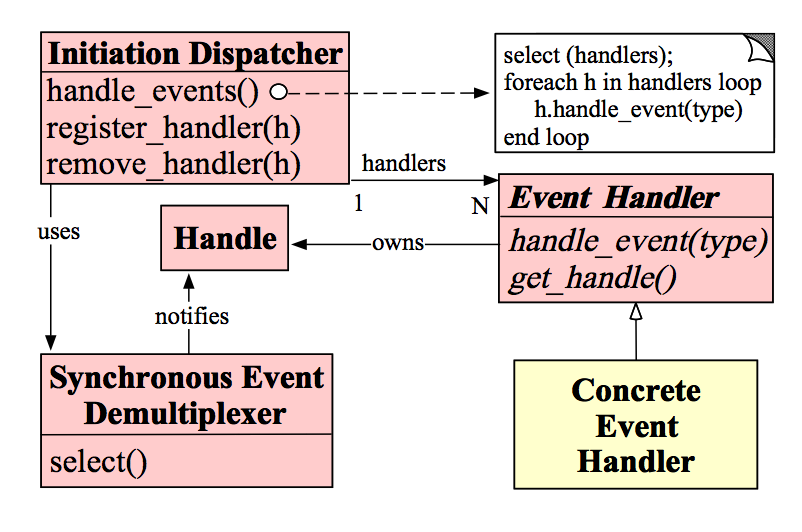
Reactor模式

事件驱动模型

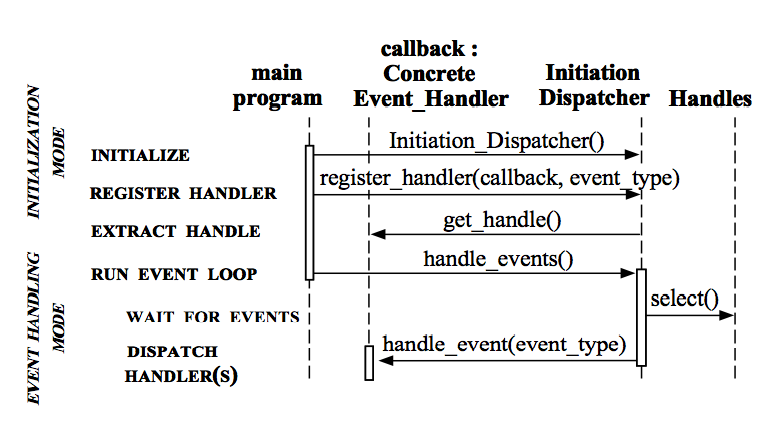
和AWT的Listener模式的对比，将事件写入到queue中，使用轮询选择关注的事件，并将其分发给关心这个事件的处理器。

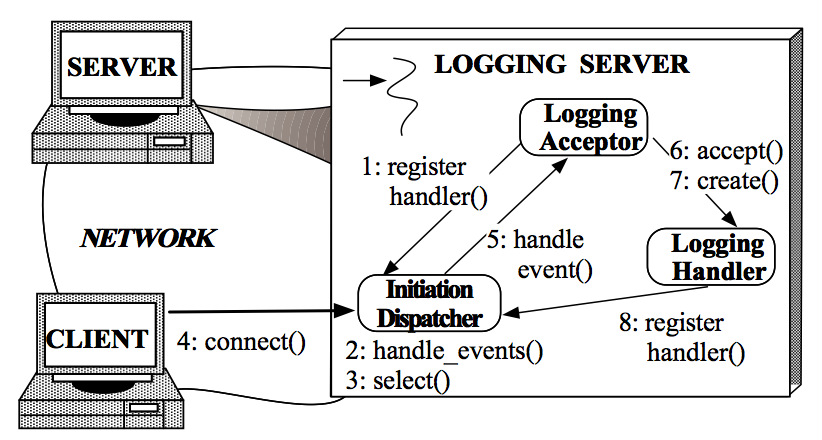
本质上这样是对事件进行了异步处理，通过异步来解决堵塞过程中等待导致的资源空闲问题，提高资源利用率的同时，也提高了整体系统的吞吐率。

对应Reactor模式以及其他类似变种模式也是围绕着这个思路，将处理流程进行切割，使用事件轮询(Queue作为缓冲)，或者直接使用多路分发器(任务直接分发给处理器)，对流程进行解耦和和异步化处理。

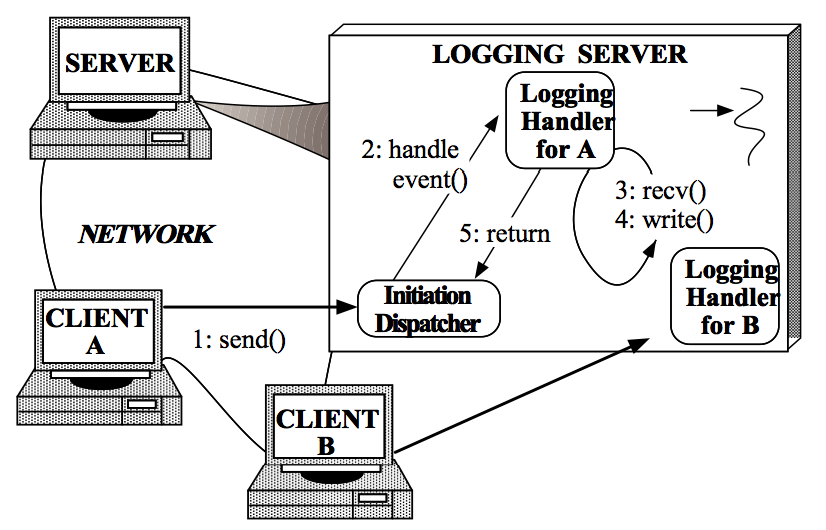


Synchronoys event demultiplexer会堵塞等待返回，实际上是NIO中selector作用





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。

Thread per request

server导致阻塞的原因：

1、serversocket的accept方法，阻塞等待client连接，直到client连接成功。

2、线程从socket inputstream读入数据，会进入阻塞状态，直到全部数据读完。

3、线程向socket outputstream写入数据，会阻塞直到全部数据写完。

client导致阻塞的原因：

1、client建立连接时会阻塞，直到连接成功。

2、线程从socket输入流读入数据，如果没有足够数据读完会进入阻塞状态，直到有数据或者读到输入流末尾。

3、线程从socket输出流写入数据，直到输出所有数据。

4、socket.setsolinger()设置socket的延迟时间，当socket关闭时，会进入阻塞状态，直到全部数据都发送完或者超时。

改进：采用基于事件驱动的设计，当有事件触发时，才会调用处理器进行数据处理。

Doug Lea 的Scalable IO In Java从最简单的单线程Reactor、Acceptor、Handler演化为将Handler中的处理逻辑多线程化，实现类似Proactor的模式。此时所有的IO操作还是单线程的，再演化为一个Main Reactor来处理CONNECT事件(Acceptor)，多个Sub Reactor来处理READ、WRITE等事件（Handler），这些Sub Reactor可以分别在自己的线程中执行，从而IO操作也多线程化，最后这样正是Netty中使用的模型。

/\*\*半包问题\*\*/

TCP/IP协议簇基于流(Stream-based)传输，消息或者数据会被分成一个个数据包（packets）进行传输，因此在发送端发送的消息会被打散并发送，而在接受端则无法保证接收到的顺序和发送的顺序保持一致。因此一般需要在接收端进行数据包的重组。