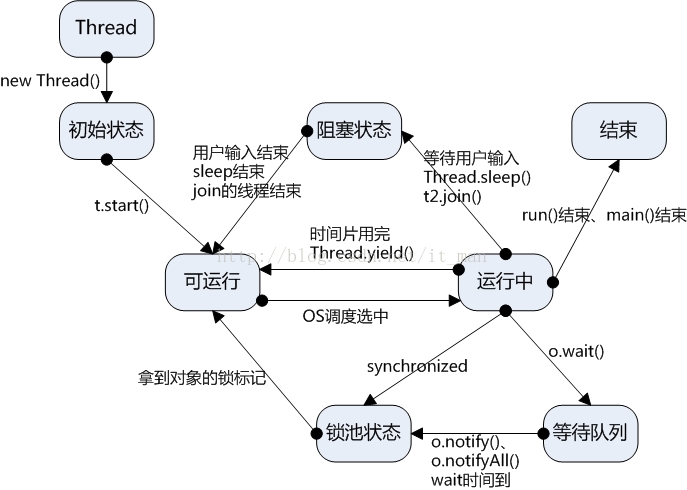
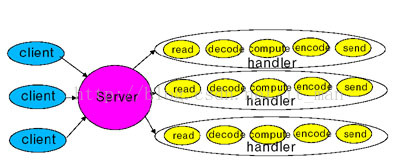
## 线程状态转换图



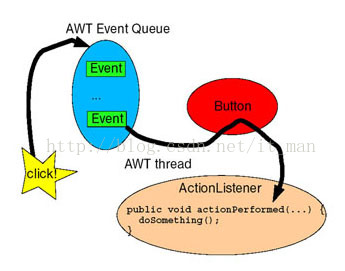
就是非阻塞IO 采用多路分发方式  
举个例子吧，你服务器做一个聊天室，按照以前的阻塞式IO，你必须为每个连接创建一个线程 因为当你调用如 in.read(buf)时，线程会阻塞在这里。而采用nio，只要注册了事件，它内部采用反应模式，当有IO事件发生时，再调度它，而不用等待在那里.

当前分布式计算　Web Services盛行天下，这些网络服务的底层都离不开对socket的操作。他们都有一个共同的结构：  
1. Read request  
2. Decode request  
3. Process service  
4. Encode reply  
5. Send reply

## 经典的网络服务的设计

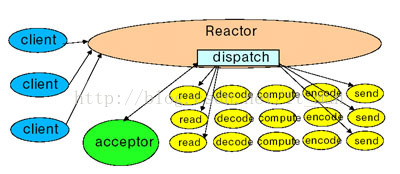
如下图，在每个线程中完成对数据的处理：  


但这种模式在用户负载增加时，性能将下降非常的快。我们需要重新寻找一个新的方案，保持数据处理的流畅，很显然，事件触发机制是最好的解决办法，当有事件发生时，会触动handler,然后开始数据的处理。

Reactor模式类似于AWT中的Event处理：  


## Reactor模式参与者

1.Reactor 负责响应IO事件，一旦发生，广播发送给相应的Handler去处理,这类似于AWT的thread  
2.Handler 是负责非堵塞行为，类似于AWT ActionListeners；同时负责将handlers与event事件绑定，类似于AWT addActionListener

如图：  


[**Java**](http://lib.csdn.net/base/javaee)的NIO为reactor模式提供了实现的基础机制，它的Selector当发现某个channel有数据时，会通过SlectorKey来告知我们，在此我们实现事件和handler的绑定。

## 我们来看看Reactor模式代码:

|  |
| --- |
| public class Reactor implements Runnable{  　　final Selector selector; 　　final ServerSocketChannel serverSocket;  　　Reactor(int port) throws IOException { 　　　　selector = Selector.open(); 　　　　serverSocket = ServerSocketChannel.open(); 　　　　InetSocketAddress address = new InetSocketAddress(InetAddress.getLocalHost(),port); 　　　　serverSocket.socket().bind(address);  　　　　serverSocket.configureBlocking(false); 　　　　//向selector注册该channel 　　　　 SelectionKey sk =serverSocket.register(selector,SelectionKey.OP\_ACCEPT);  　　　　logger.debug("-->Start serverSocket.register!");  　　　　//利用sk的attache功能绑定Acceptor 如果有事情，触发Acceptor 　　　　sk.attach(new Acceptor()); 　　　　logger.debug("-->attach(new Acceptor()!"); 　　}  　　public void run() { // normally in a new Thread 　　　　try { 　　　　while (!Thread.interrupted()) 　　　　{ 　　　　　　selector.select(); 　　　　　　Set selected = selector.selectedKeys(); 　　　　　　Iterator it = selected.iterator(); 　　　　　　//Selector如果发现channel有OP\_ACCEPT或READ事件发生，下列遍历就会进行。 　　　　　　while (it.hasNext()) 　　　　　　　　//来一个事件 第一次触发一个accepter线程 　　　　　　　　//以后触发SocketReadHandler 　　　　　　　　dispatch((SelectionKey)(it.next())); 　　　　　　　　selected.clear(); 　　　　　　} 　　　　}catch (IOException ex) { 　　　　　　　　logger.debug("reactor stop!"+ex); 　　　　} 　　}  　　//运行Acceptor或SocketReadHandler 　　void dispatch(SelectionKey k) { 　　　　Runnable r = (Runnable)(k.attachment()); 　　　　if (r != null){ 　　　　　　// r.run();  　　　　} 　　}  　　class Acceptor implements Runnable { // inner 　　　　public void run() { 　　　　try { 　　　　　　logger.debug("-->ready for accept!"); 　　　　　　SocketChannel c = serverSocket.accept(); 　　　　　　if (c != null) 　　　　　　　　//调用Handler来处理channel 　　　　　　　　new SocketReadHandler(selector, c); 　　　　　　} 　　　　catch(IOException ex) { 　　　　　　logger.debug("accept stop!"+ex); 　　　　} 　　　　} 　　} } |

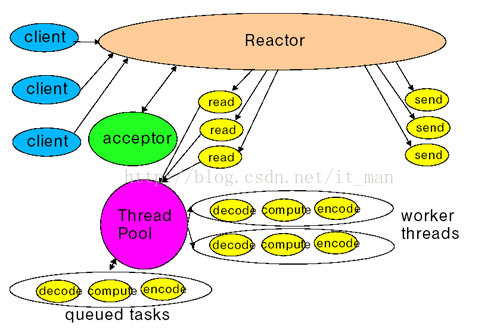
以上代码中巧妙使用了SocketChannel的attach功能，将Hanlder和可能会发生事件的channel链接在一起，当发生事件时，可以立即触发相应链接的Handler。

## 再看看Handler代码:

|  |
| --- |
| public class SocketReadHandler implements Runnable {  　　public static Logger logger = Logger.getLogger(SocketReadHandler.class);  　　private Test test=new Test();  　　final SocketChannel socket; 　　final SelectionKey sk;  　　 static final int READING = 0, SENDING = 1; 　　int state = READING;  　　public SocketReadHandler(Selector sel, SocketChannel c) 　　　　throws IOException {  　　　　socket = c;  　　　　socket.configureBlocking(false); 　　　　 sk = socket.register(sel, 0);  　　　　//将SelectionKey绑定为本Handler 下一步有事件触发时，将调用本类的run方法。 　　　　//参看dispatch(SelectionKey k) 　　　　sk.attach(this);  　　　　//同时将SelectionKey标记为可读，以便读取。 　　　　sk.interestOps(SelectionKey.OP\_READ); 　　　　sel.wakeup(); 　　}  　　public void run() { 　　　　try{ 　　　　// test.read(socket,input); 　　　　　　readRequest() ; 　　　　}catch(Exception ex){ 　　　　logger.debug("readRequest error"+ex); 　　　　} 　　}  /\*\* \* 处理读取data \* @param key \* @throws Exception \*/ private void readRequest() throws Exception {  　　ByteBuffer input = ByteBuffer.allocate(1024); 　　input.clear(); 　　try{  　　　　int bytesRead = socket.read(input);  　　　　......  　　　　//激活线程池 处理这些request 　　　　requestHandle(new Request(socket,btt));  　　　　.....  　　}catch(Exception e) { 　　}  } |

注意在Handler里面又执行了一次attach，这样，覆盖前面的Acceptor，下次该Handler又有READ事件发生时，将直接触发Handler.从而开始了数据的读　处理　写　发出等流程处理。

将数据读出后，可以将这些数据处理线程做成一个线程池，这样，数据读出后，立即扔到线程池中，这样加速处理速度：



更进一步，我们可以使用多个Selector分别处理连接和读事件。

一个高性能的Java网络服务机制就要形成，激动人心的集群并行计算即将实现。

<http://www.jdon.com/concurrent/reactor.htm>

--------------

Java　NIO非堵塞应用通常适用用在I/O读写等方面，我们知道，系统运行的性能瓶颈通常在I/O读写，包括对端口和文件的操作上，过去，在打开一个I/O通道后，read()将一直等待在端口一边读取字节内容，如果没有内容进来，read()也是傻傻的等，这会影响我们程序继续做其他事情，那么改进做法就是开设线程，让线程去等待，但是这样做也是相当耗费资源的。

[**Java**](http://lib.csdn.net/base/java)NIO非堵塞技术实际是采取Reactor模式，或者说是Observer模式为我们监察I/O端口，如果有内容进来，会自动通知我们，这样，我们就不必开启多个线程死等，从外界看，实现了流畅的I/O读写，不堵塞了。

Java NIO出现不只是一个技术性能的提高，你会发现网络上到处在介绍它，因为它具有里程碑意义，从JDK1.4开始，Java开始提高性能相关的功能，从而使得Java在底层或者并行分布式计算等操作上已经可以和C或Perl等语言并驾齐驱。

如果你至今还是在怀疑Java的性能，说明你的思想和观念已经完全落伍了，Java一两年就应该用新的名词来定义。从JDK1.5开始又要提供关于线程、并发等新性能的支持，Java应用在游戏等适时领域方面的机会已经成熟，Java在稳定自己中间件地位后，开始蚕食传统C的领域。

本文主要简单介绍NIO的基本原理，在下一篇文章中，将结合Reactor模式和著名线程大师[**Doug Lea**](http://gee.cs.oswego.edu/dl/)的一篇文章深入讨论。

NIO主要原理和适用。

NIO 有一个主要的类Selector,这个类似一个观察者，只要我们把需要探知的socketchannel告诉Selector,我们接着做别的事情，当有事件发生时，他会通知我们，传回一组SelectionKey,我们读取这些Key,就会获得我们刚刚注册过的socketchannel,然后，我们从这个Channel中读取数据，放心，包准能够读到，接着我们可以处理这些数据。

Selector内部原理实际是在做一个对所注册的channel的轮询访问，不断的轮询(目前就这一个[**算法**](http://lib.csdn.net/base/datastructure))，一旦轮询到一个channel有所注册的事情发生，比如数据来了，他就会站起来报告，交出一把钥匙，让我们通过这把钥匙来读取这个channel的内容。

了解了这个基本原理，我们结合代码看看使用，在使用上，也在分两个方向，一个是线程处理，一个是用非线程，后者比较简单，看下面代码：

|  |
| --- |
| import java.io.\*; import java.nio.\*; import java.nio.channels.\*; import java.nio.channels.spi.\*; import java.net.\*; import java.util.\*;  /\*\* \* \* @author Administrator \* @version \*/  public class NBTest {  　　/\*\* Creates new NBTest \*/ 　　public NBTest() 　　{ 　　}  　　public void startServer() throws Exception 　　{ 　　int channels = 0; 　　int nKeys = 0; 　　int currentSelector = 0;  　　//使用Selector 　　Selector selector = Selector.open();  　　//建立Channel 并绑定到9000端口 　　ServerSocketChannel ssc = ServerSocketChannel.open(); 　　InetSocketAddress address = new InetSocketAddress(InetAddress.getLocalHost(),9000);  　　ssc.socket().bind(address);  　　//使设定non-blocking的方式。 　　ssc.configureBlocking(false);  　　//向Selector注册Channel及我们有兴趣的事件 　　SelectionKey s = ssc.register(selector, SelectionKey.OP\_ACCEPT); 　　printKeyInfo(s);  　　while(true) //不断的轮询 　　{ 　　　　debug("NBTest: Starting select");  　　　　//Selector通过select方法通知我们我们感兴趣的事件发生了。 　　　　nKeys = selector.select(); 　　　　//如果有我们注册的事情发生了，它的传回值就会大于0 　　　　if(nKeys > 0) 　　　　{ 　　　　　　debug("NBTest: Number of keys after select operation: " +nKeys);  　　　　　　//Selector传回一组SelectionKeys 　　　　　　//我们从这些key中的channel()方法中取得我们刚刚注册的channel。 　　　　　　Set selectedKeys = selector.selectedKeys(); 　　　　　　Iterator i = selectedKeys.iterator(); 　　　　　　while(i.hasNext()) 　　　　　　{ 　　　　　　 　 s = (SelectionKey) i.next(); 　　　　　　 　 printKeyInfo(s); 　　　　　　 　 debug("NBTest: Nr Keys in selector: " +selector.keys().size());  　　　　　　 　 //一个key被处理完成后，就都被从就绪关键字（ready keys）列表中除去 　　　　　　 　 i.remove(); 　　　　　　 　 if(s.isAcceptable()) 　　　　　　 　 { 　　　　　　 　 　 // 从channel()中取得我们刚刚注册的channel。 　　　　　　 　 　 Socket socket = ((ServerSocketChannel)s.channel()).accept().socket(); 　　　　　　 　 　 SocketChannel sc = socket.getChannel();  　　　　　　 　 　 sc.configureBlocking(false); 　　　　　　 　 　 sc.register(selector, SelectionKey.OP\_READ |SelectionKey.OP\_WRITE); 　　　　　　 　 　 　　　　　　 　 　 System.out.println(++channels); 　　　　　　 　 } 　　　　　　 　 else 　　　　　　 　 { 　　　　　　 　 　 debug("NBTest: Channel not acceptable"); 　　　　　　 　 } 　　　　　 } 　　　} 　　　else 　　　{ 　　　　　　debug("NBTest: Select finished without any keys."); 　　　}  　 }  }  private static void debug(String s) { 　 System.out.println(s); }  private static void printKeyInfo(SelectionKey sk) { 　 String s = new String();  　 s = "Att: " + (sk.attachment() == null ? "no" : "yes"); 　 s += ", Read: " + sk.isReadable(); 　 s += ", Acpt: " + sk.isAcceptable(); 　 s += ", Cnct: " + sk.isConnectable(); 　 s += ", Wrt: " + sk.isWritable(); 　 s += ", Valid: " + sk.isValid(); 　 s += ", Ops: " + sk.interestOps(); 　 debug(s); }  /\*\* \* @param args the command line arguments \*/ public static void main (String args[]) { 　 NBTest nbTest = new NBTest(); 　 try 　 { 　 　 nbTest.startServer(); 　 } 　 　 catch(Exception e) 　 { 　 　 e.printStackTrace(); 　 } }  } |

这是一个守候在端口9000的noblock server例子，如果我们编制一个客户端程序，就可以对它进行互动操作，或者使用telnet 主机名　90000 可以链接上。

通过仔细阅读这个例程，相信你已经大致了解NIO的原理和使用方法，下一篇，我们将使用多线程来处理这些数据，再搭建一个自己的Reactor模式。