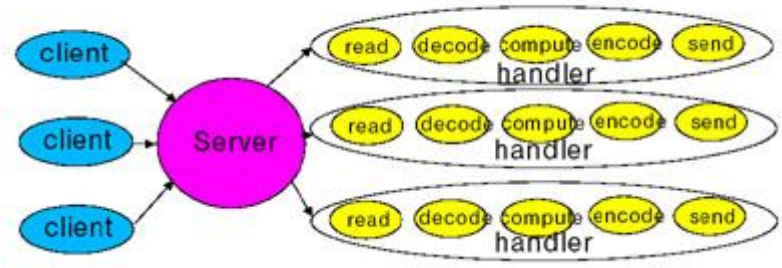
Reactor 模式和 NIO

本文可看成是对 Doug Lea Scalable IO in Java 一文的翻译。

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：

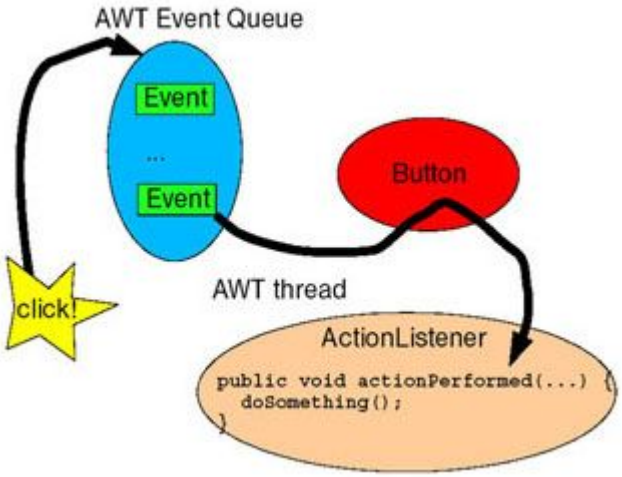
1. Read request
2. Decode request
3. Process service
4. Encode reply
5. Send reply

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：

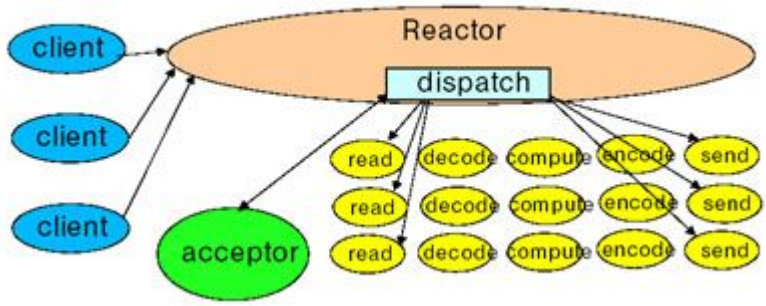


Reactor 模式参与者

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 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。

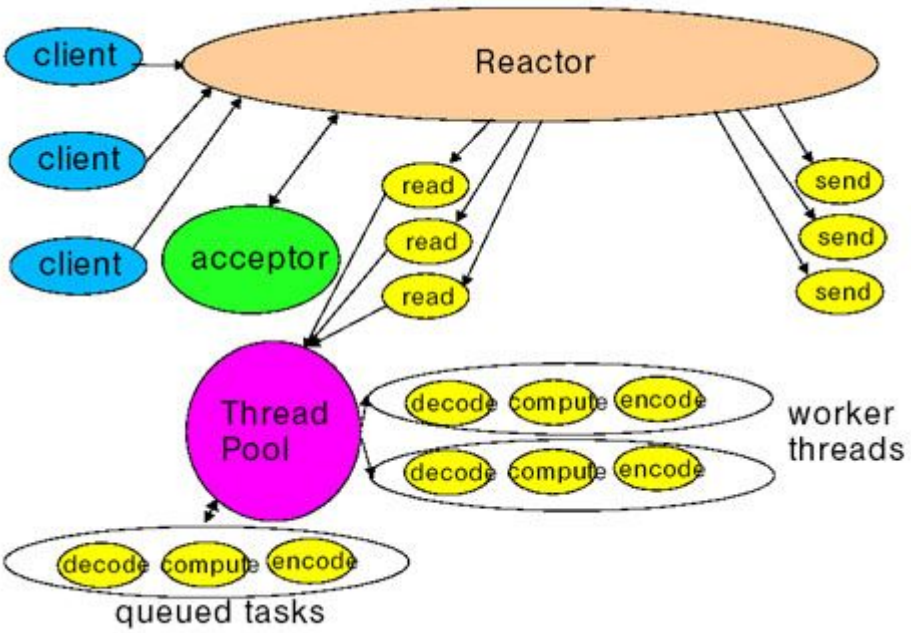
再看看 Handler 代码:

```
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。

Netty 原理和使用

Netty 是一个高性能 事件驱动的异步的非堵塞的 IO(NIO)框架，用于建立 TCP 等底层的连接，基于 Netty 可以建立高性能的 Http 服务器。支持 HTTP、WebSocket 、Protobuf、 Binary TCP |和 UDP，Netty 已经被很多高性能项目作为其 Socket 底层基础，如 HornetQ Infinispan Vert.x
Play Framework Finangle 和 Cassandra。其竞争对手是：Apache MINA 和 Grizzly。

传统堵塞的 IO 读取如下：

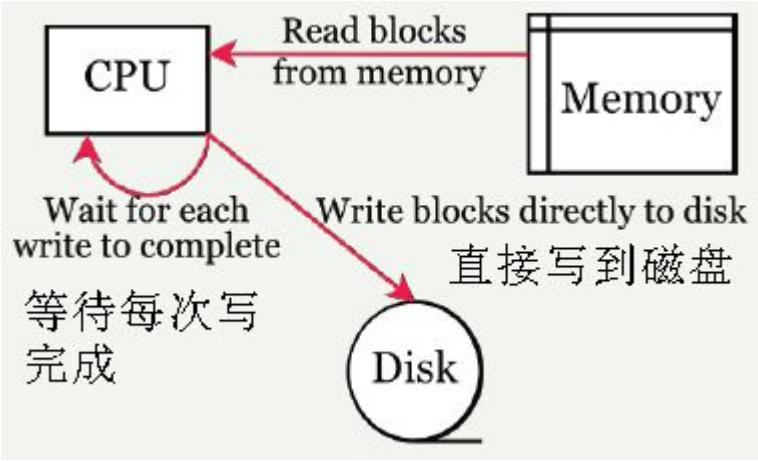
```
InputStream is = new FileInputStream("input.bin");
 int byte = is.read(); // 当前线程等待结果到达直至错误
```

而使用 NIO 如下：

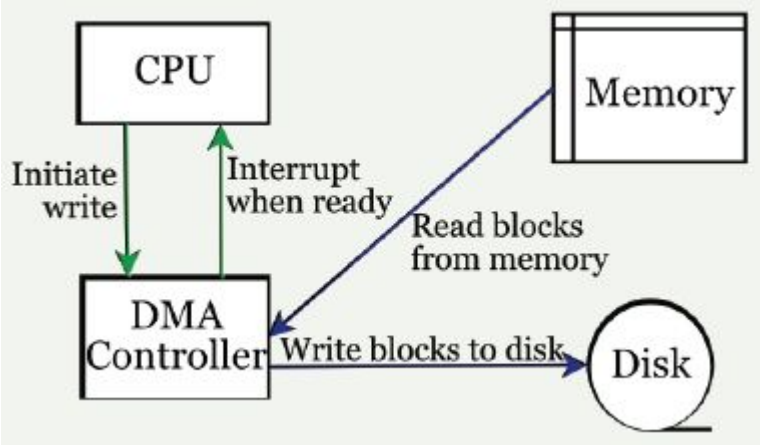
```
while (true) {
   selector.select(); // 从多个通道请求事件
   Iterator it = selector.selectedKeys().iterator();
   while (it.hasNext()) {
     SelectorKey key = (SelectionKey) it.next();
     handleKey(key);
     it.remove();
   }
 }
```

堵塞与非堵塞原理

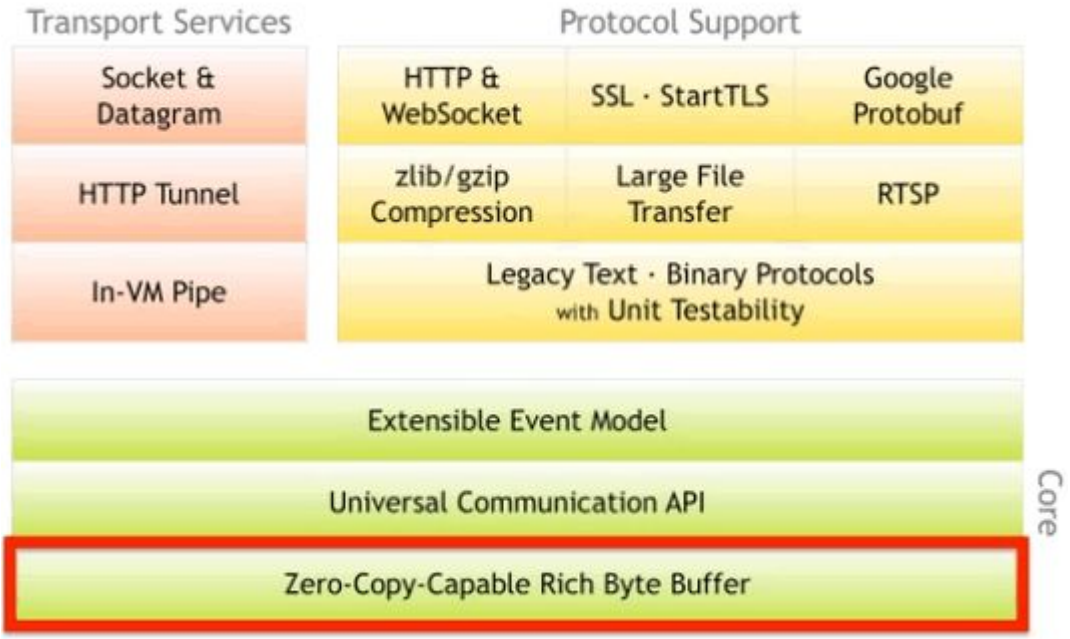
传统硬件的堵塞如下，从内存中读取数据，然后写到磁盘，而 CPU 一直等到磁盘写完成，磁盘的写操作是慢的，这段时间 CPU 被堵塞不能发挥效率。



使用非堵塞的 DMA 如下图：CPU 只是发出写操作这样的指令，做一些初始化工作，DMA 具体执行，从内存中读取数据，然后写到磁盘，当完成写后发出一个中断事件给 CPU。这段时间 CPU 是空闲的，可以做别的事情。这个原理称为 Zero.copy 零拷贝。

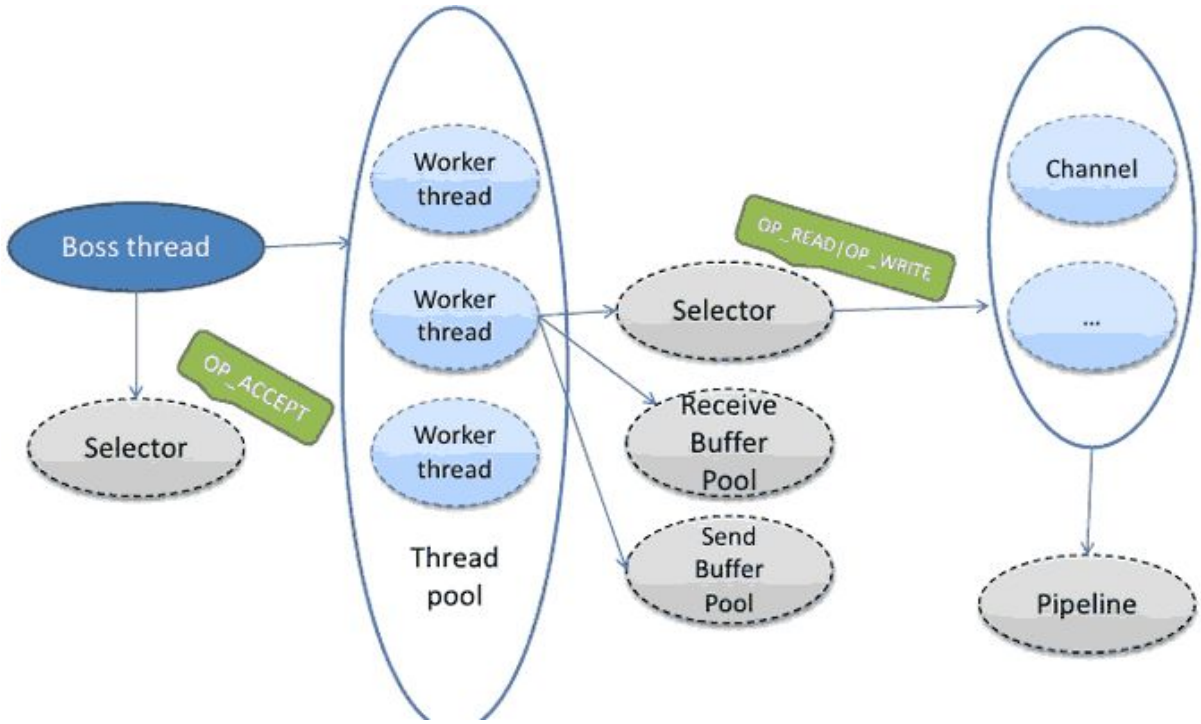


Netty 底层基于上述 Java NIO 的零拷贝原理实现：



比较

- Tomcat 是一个 Web 服务器，它是采取一个请求一个线程，当有1000客户端时，会耗费很多内存。通常一个线程将花费 256kb 到1mb 的 stack 空间。
- Node.js 是一个线程服务于所有请求，在错误处理上有限制
- Netty 是一个线程服务于很多请求，如下图，当从 Java NIO 获得一个 Selector 事件，将激活通道 Channel。



演示

Netty 的使用代码如下：

```
Channel channel = ...
 ChannelFuture cf = channel.write(data);
 cf.addListener(
     new ChannelFutureListener() {
       @Override
         public void operationComplete(ChannelFuture future)  throws
Exception {
           if(!future.isSuccess() {
                 future.cause().printStacktrace();
                 ...
           }
           ...
       }
 });
 ...
 cf.sync();
```

通过引入观察者监听，当有数据时，将自动激活监听者中的代码运行。

我们使用 Netty 建立一个服务器代码：

```
public class EchoServer {
```

```
    private final int port;

    public EchoServer(int port) {
         this.port = port;
     }

    public void run() throws Exception {
         // Configure the server.
         EventLoopGroup bossGroup = new NioEventLoopGroup();
         EventLoopGroup workerGroup = new NioEventLoopGroup();
         try {
             ServerBootstrap b = new ServerBootstrap();
             b.group(bossGroup,
workerGroup).channel(NioServerSocketChannel.class).option(ChannelOpti
on.SO_BACKLOG, 100)
                     .handler(new
LoggingHandler(LogLevel.INFO)).childHandler(new
ChannelInitializer<SocketChannel>() {
                         @Override
                         public void initChannel(SocketChannel ch)
throws Exception {
                             ch.pipeline().addLast(
                             // new LoggingHandler(LogLevel.INFO),
                                     new EchoServerHandler());
                         }
                     });

            // Start the server.
             ChannelFuture f = b.bind(port).sync();

            // Wait until the server socket is closed.
             f.channel().closeFuture().sync();
         } finally {
             // Shut down all event loops to terminate all threads.
             bossGroup.shutdownGracefully();
             workerGroup.shutdownGracefully();
         }
     }


 }
```

这段代码调用：在9999端口启动

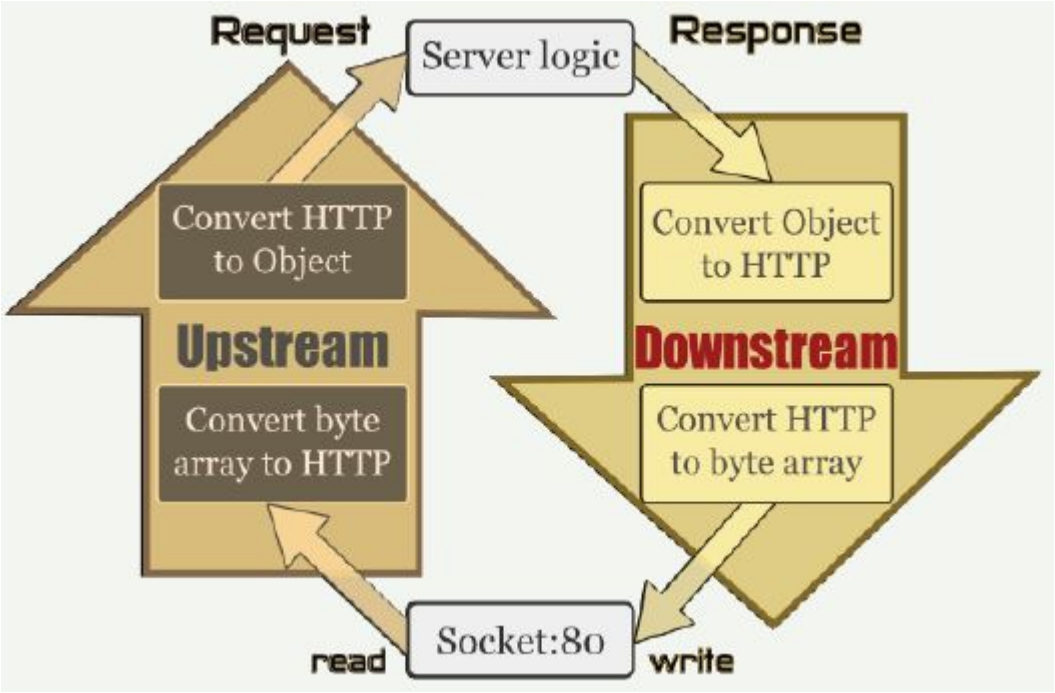
```
new EchoServer(9999).run();
```

我们需要完成的代码是 EchoServerHandler：

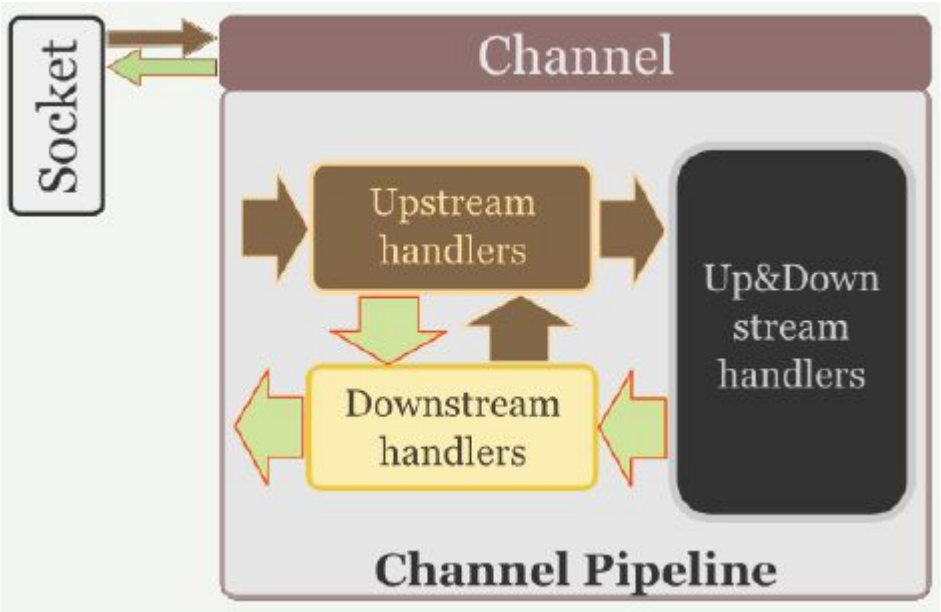
```
public class EchoServerHandler extends ChannelInboundHandlerAdapter {

    private static final Logger logger =
Logger.getLogger(EchoServerHandler.class.getName());

    @Override
     public void channelRead(ChannelHandlerContext ctx, Object msg)
throws Exception {
         ctx.write(msg);
     }

    @Override
     public void channelReadComplete(ChannelHandlerContext ctx) throws
Exception {
         ctx.flush();
     }

    @Override
     public void exceptionCaught(ChannelHandlerContext ctx, Throwable
cause) {
         // Close the connection when an exception is raised.
         logger.log(Level.WARNING, "Unexpected exception from
downstream.", cause);
         ctx.close();
     }
 }
```

原理

一个 Netty 服务器的原理如下：



图中每次请求的读取是通过 UpStream 来实现，然后激活我们的服务逻辑如 EchoServerHandler，而服务器向外写数据，也就是响应是通过 DownStream 实现的。每个通道 Channel 包含一对 UpStream 和 DownStream，以及我们的 handlers（EchoServerHandler），如下图，这些都是通过 channel pipeline 封装起来的，数据流在管道里流动，每个 Socket 对应一个 ChannelPipeline。



CHANNELPIPELINE 是关键，它类似 Unix 的管道，有以下作用：

- 为每个 Channel 保留 ChannelHandlers ，如 EchoServerHandler

- 所有的事件都要通过它
- 不断地修改：类似 unix 的 SH 管道： echo "Netty is shit...." | sed -e 's/is /is the /'
- 一个 Channel 对应一个 ChannelPipeline
- 包含协议编码解码 安全验证 SSL/TLS 和应用逻辑

客户端代码

前面我们演示了服务器端代码，下面是客户端代码：

```
public class EchoClient {
     private final String host;
     private final int port;
     private final int firstMessageSize;

    public EchoClient(String host, int port, int firstMessageSize) {
         this.host = host;
         this.port = port;
         this.firstMessageSize = firstMessageSize;
     }

    public void run() throws Exception {
         // Configure the client.
         EventLoopGroup group = new NioEventLoopGroup();
         try {
             Bootstrap b = new Bootstrap();

b.group(group).channel(NioSocketChannel.class).option(ChannelOption.TCP_NODELAY, true).handler(new ChannelInitializer<SocketChannel>() {
                 @Override
                 public void initChannel(SocketChannel ch) throws Exception {
                     ch.pipeline().addLast(
                     // new LoggingHandler(LogLevel.INFO),
                             new EchoClientHandler(firstMessageSize));
                 }
             });

            // Start the client.
             ChannelFuture f = b.connect(host, port).sync();

            // Wait until the connection is closed.
             f.channel().closeFuture().sync();
```

```
        } finally {
            // Shut down the event loop to terminate all threads.
            group.shutdownGracefully();
        }
    }
}
```

客户端的应用逻辑 EchoClientHandler：

```
public class EchoClientHandler extends ChannelInboundHandlerAdapter {

    private static final Logger logger =
Logger.getLogger(EchoClientHandler.class.getName());

    private final ByteBuf firstMessage;

    /**
     * Creates a client-side handler.
     */
    public EchoClientHandler(int firstMessageSize) {
        if (firstMessageSize <= 0) {
            throw new IllegalArgumentException("firstMessageSize: " +
firstMessageSize);
        }
        firstMessage = Unpooled.buffer(firstMessageSize);
        for (int i = 0; i < firstMessage.capacity(); i++) {
            firstMessage.writeByte((byte) i);
        }
    }

    @Override
    public void channelActive(ChannelHandlerContext ctx) {
        ctx.writeAndFlush(firstMessage);
        System.out.print("active");
    }

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg)
throws Exception {
        ctx.write(msg);
        System.out.print("read");
    }

    @Override
    public void channelReadComplete(ChannelHandlerContext ctx) throws
```

```
Exception {
          ctx.flush();
          System.out.print("readok");
      }

    @Override
     public void exceptionCaught(ChannelHandlerContext ctx, Throwable
cause) {
          // Close the connection when an exception is raised.
          logger.log(Level.WARNING, "Unexpected exception from
downstream.", cause);
          ctx.close();
      }

}
```

Java NIO 原理和使用

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 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 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)，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。