无处不在的C/S架构

在这个充斥着云的时代,我们使用的软件可以说99%都是C/S架构的!

- 你发邮件用的Outlook,Foxmail等
- 你看视频用的优酷，土豆等
- 你写文档用的Office365,googleDoc，Evernote等
- 你浏览网页用的IE,Chrome等(B/S是特殊的C/S)
- ……

C/S架构的软件带来的一个明显的好处就是：只要有网络，你可以在任何地方干同一件事。

例如：你在家里使用Office365编写了文档。到了公司，只要打开编辑地址就可以看到在家里编写的文档，进行展示或者继续编辑。甚至在手机上进行阅读与编辑。不再需要U盘拷来拷去了。

C/S架构可以抽象为如下模型：

client-server.png

- C就是Client(客户端),上面的B是Browser(浏览器)
- S就是Server(服务器)：**服务器管理某种资源，并且通过操作这种资源来为它的客户端提供某种服务**

C/S架构之所以能够流行的一个主要原因就是网速的提高以及费用的降低，特别是无线网络速度的提高。试想在2G时代，大家最多就是看看文字网页，小说什么的。看图片，那简直就是奢侈！更别说看视频了！

网速的提高，使得越来越多的人使用网络，例如：优酷，微信都是上亿用户量，更别说天猫双11的瞬间访问量了！这就对服务器有很高的要求！能够快速处理海量的用户请求！那服务器如何能快速的处理用户的请求呢?

高性能服务器

高性能服务器至少要满足如下几个需求：

- 效率高：既然是高性能，那处理客户端请求的效率当然要很高了
- 高可用：不能随便就挂掉了
- 编程简单：基于此服务器进行业务开发需要足够简单
- 可扩展：可方便的扩展功能
- 可伸缩：可简单的通过部署的方式进行容量的伸缩，也就是服务需要无状态

而满足如上需求的一个基础就是高性能的IO!

Socket

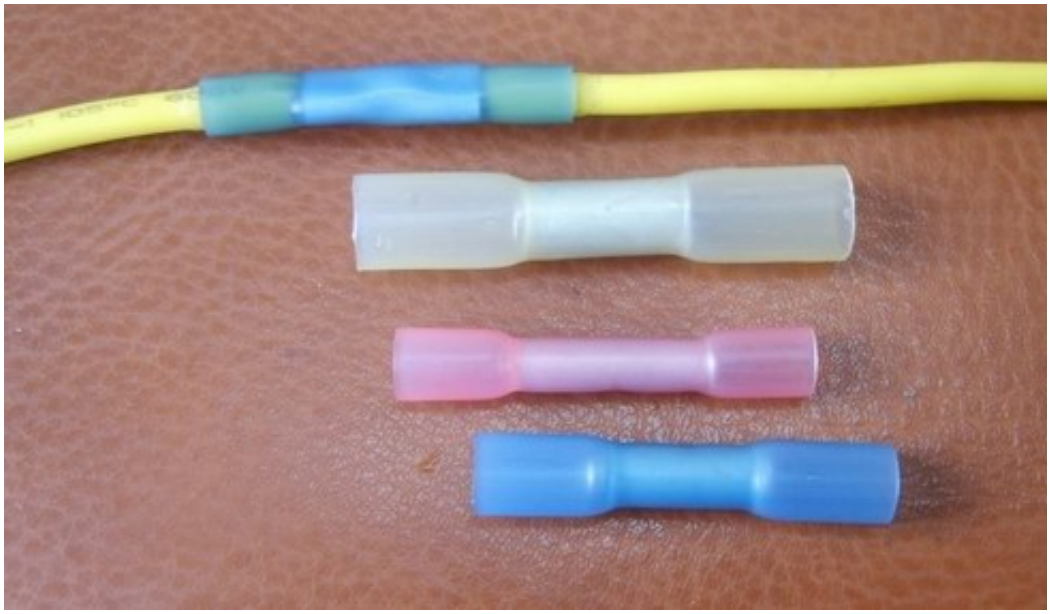
无论你是发邮件，浏览网页，还是看视频～实际底层都是使用的TCP/IP，而TCP/IP的编程抽象就是Socket!

我一直对Socket的中文翻译很困惑，个人觉得是我所接触的技术名词翻译里最莫名其妙的，没有之一！

Socket中文翻译为”套接字”！什么鬼？在很长的时间里我都无法将其和网络编程关联上！后来专门找了一些资料，最后在知乎上找到了一个还算满意的答案(具体链接，请见文末的参考资料链接)！

Socket的原意是插口，想表达的意思是插口与插槽的关系！”send socket”插到”receive socket”里，建立了链接，然后就可以通信了！

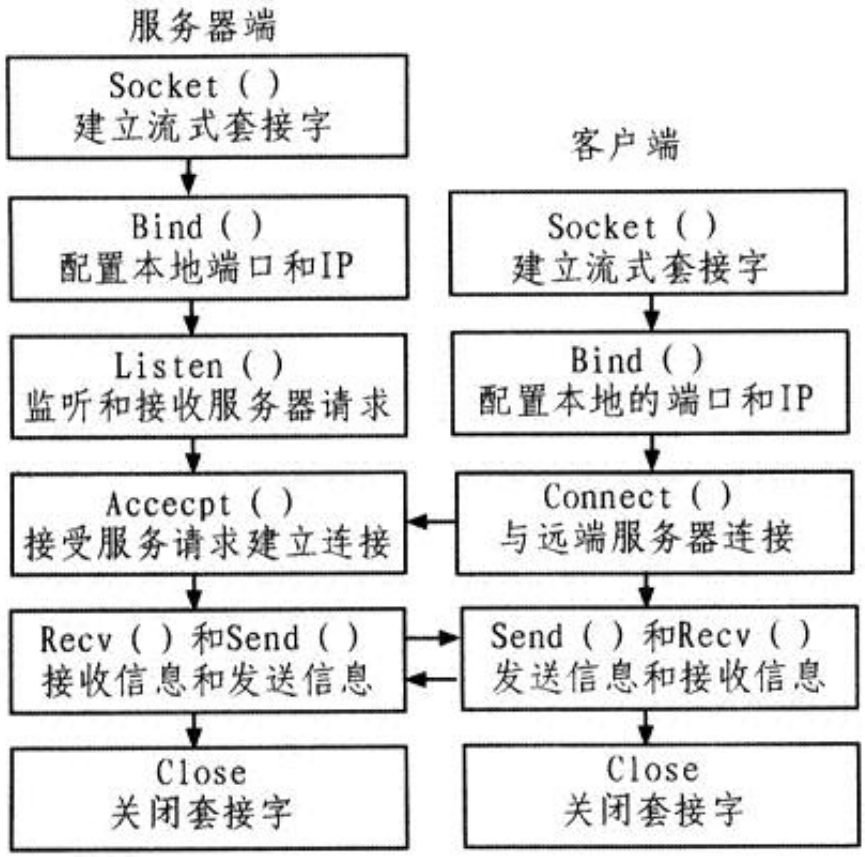
套接字的翻译，应该是参考了套接管(如下图)！从这个层面上来看，是有那么点意思！



%E5%A5%97%E6%8E%A5%E7%AE%A1.jpg

套接字这个翻译已经是标准了，不纠结这个了！

我们看一下Socket之间建立链接及通信的过程！实际上就是对TCP/IP连接与通信过程的抽象:



socket.png

- 服务端Socket会bind到指定的端口上，Listen客户端的”插入”
- 客户端Socket会Connect到服务端
- 当服务端Accept到客户端连接后
- 就可以进行发送与接收消息了
- 通信完成后即可Close

对于IO来说，我们听得比较多的是:

- BIO:阻塞IO
- NIO:非阻塞IO
- 同步IO
- 异步IO

以及其组合:

- 同步阻塞IO
- 同步非阻塞IO
- 异步阻塞IO
- 异步非阻塞IO

那么什么是阻塞IO、非阻塞IO、同步IO、异步IO呢?

- 一个IO操作其实分成了两个步骤：发起IO请求和实际的IO操作
- 阻塞IO和非阻塞IO的区别在于第一步：发起IO请求是否会被阻塞，如果阻塞直到完成那么就是传统的阻塞IO;如果不阻塞，那么就是非阻塞IO
- 同步IO和异步IO的区别就在于第二个步骤是否阻塞，如果实际的IO读写阻塞请求进程，那么就是同步IO，因此阻塞IO、非阻塞IO、IO复用、信号驱动IO都是同步IO;如果不阻塞，而是操作系统帮你做完IO操作再将结果返回给你，那么就是异步IO

举个不太恰当的例子 ： 比如你家网络断了，你打电话去中国电信报修!

- 你拨号—客户端连接服务器
- 电话通了—连接建立
- 你说：“我家网断了,帮我修下”—发送消息
- 说完你就在那里等，那么就是阻塞IO
- 如果正好你有事，你放下带电话，然后处理其他事情了，过一会你来问下，修好了没—那就是非阻塞IO
- 如果客服说：“马上帮你处理，你稍等”—同步IO
- 如果客服说：“马上帮你处理，好了通知你”，然后挂了电话—异步IO

本文只讨论BIO和NIO,AIO使用度没有前两者普及，暂不讨论!

下面从代码层面看看BIO与NIO的流程!

BIO

- 客户端代码

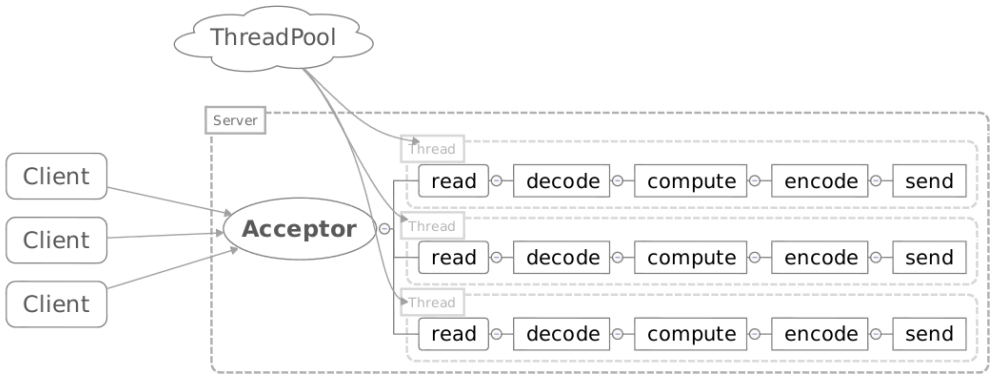
```
//Bind,Connect
Socket client = new Socket("127.0.0.1",7777);
//读写
PrintWriter pw = new PrintWriter(client.getOutputStream());
BufferedReader br=
      new BufferedReader(new InputStreamReader(System.in));
```

客户端代码

```
Socket socket;
//Bind,Listen
ServerSocket ss = new ServerSocket(7777);
while (true) {
    //Accept
    socket = ss.accept();
    //一般新建一个线程执行读写
    BufferedReader br = new BufferedReader(
          new InputStreamReader(socket  .getInputStream()));
    System.out.println("you input is : " + br.readLine());
}
```

- 上面的代码可以说是学习Java的Socket的入门级代码了
- 代码流程和前面的图可以一一对上

模型图如下所示：



bio.png

BIO优缺点

- 优点
  - 模型简单
  - 编码简单
- 缺点
  - 性能瓶颈低

优缺点很明显。这里主要说下缺点：主要瓶颈在线程上。每个连接都会建立一个线程。虽然线程消耗比进程小，但是一台机器实际上能建立的有效

线程有限，以Java来说，1.5以后，一个线程大致消耗1M内存！且随着线程数量的增加，CPU切换线程上下文的消耗也随之增加，在高过某个阀值后，继续增加线程，性能不增反降！而同样因为一个连接就新建一个线程，所以编码模型很简单！

就性能瓶颈这一点，就确定了BIO并不适合进行高性能服务器的开发！像Tomcat这样的Web服务器，从7开始就从BIO改成了NIO，来提高服务器性能！

NIO

- NIO客户端代码(连接)

```
//获取socket通道
SocketChannel channel = SocketChannel.open();
channel.configureBlocking(false);
//获得通道管理器
selector=Selector.open();
channel.connect(new InetSocketAddress(serverIp, port));
//为该通道注册SelectionKey.OP_CONNECT事件
channel.register(selector, SelectionKey.OP_CONNECT);
```

- NIO客户端代码(监听)

```
while(true){
   //选择注册过的io操作的事件(第一次为SelectionKey.OP_CONNECT)
  selector.select();
  while(SelectionKey key : selector.selectedKeys()){
     if(key.isConnectable()){
        SocketChannel channel=(SocketChannel)key.channel();
        if(channel.isConnectionPending()){
           channel.finishConnect();//如果正在连接，则完成连接
        }
        channel.register(selector, SelectionKey.OP_READ);
     }else if(key.isReadable()){ //有可读数据事件。
        SocketChannel channel = (SocketChannel)key.channel();
        ByteBuffer buffer = ByteBuffer.allocate(10);
        channel.read(buffer);
        byte[] data = buffer.array();
        String message = new String(data);
        System.out.println("recevie message from server:, size:"
           + buffer.position() + " msg: " + message);
     }
  }
}
```

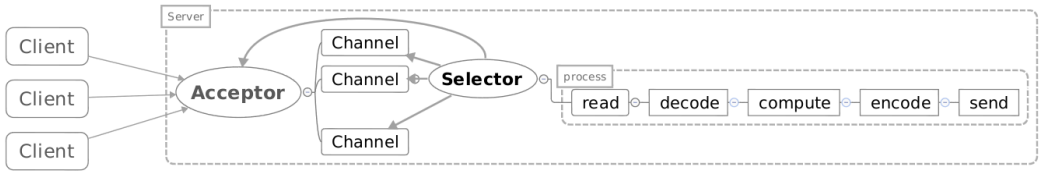
- NIO服务端代码(连接)

```
//获取一个ServerSocket通道
ServerSocketChannel serverChannel = ServerSocketChannel.open();
serverChannel.configureBlocking(false);
serverChannel.socket().bind(new InetSocketAddress(port));
//获取通道管理器
selector = Selector.open();
//将通道管理器与通道绑定，并为该通道注册SelectionKey.OP_ACCEPT事
serverChannel.register(selector, SelectionKey.OP_ACCEPT);
```

- NIO服务端代码(监听)

```
while(true){
   //当有注册的事件到达时，方法返回，否则阻塞。
  selector.select();
  for(SelectionKey key : selector.selectedKeys()){
     if(key.isAcceptable()){
        ServerSocketChannel server =
            (ServerSocketChannel)key.channel();
        SocketChannel channel = server.accept();
        channel.write(ByteBuffer.wrap(
         new String("send message to client").getBytes()));
        //在与客户端连接成功后，为客户端通道注册SelectionKey.OP_REA
        channel.register(selector, SelectionKey.OP_READ);
     }else if(key.isReadable()){//有可读数据事件
        SocketChannel channel = (SocketChannel)key.channel();
        ByteBuffer buffer = ByteBuffer.allocate(10);
        int read = channel.read(buffer);
        byte[] data = buffer.array();
        String message = new String(data);
        System.out.println("receive message from client, size:"
           + buffer.position() + " msg: " + message);
     }
  }
}
```

NIO模型示例如下：



nio.png

- Acceptor注册Selector，监听accept事件
- 当客户端连接后，触发accept事件
- 服务器构建对应的Channel，并在其上注册Selector，监听读写事件
- 当发生读写事件后，进行相应的读写处理

NIO优缺点

- 优点
  - 性能瓶颈高
- 缺点
  - 模型复杂
  - 编码复杂
  - 需处理半包问题

NIO的优缺点和BIO就完全相反了!性能高，不用一个连接就建一个线程，可以一个线程处理所有的连接！相应的，编码就复杂很多，从上面的代码就可以明显体会到了。还有一个问题，由于是非阻塞的，应用无法知道什么时候消息读完了，就存在了半包问题!

半包问题

简单看一下下面的图就能理解半包问题了!

package01.png

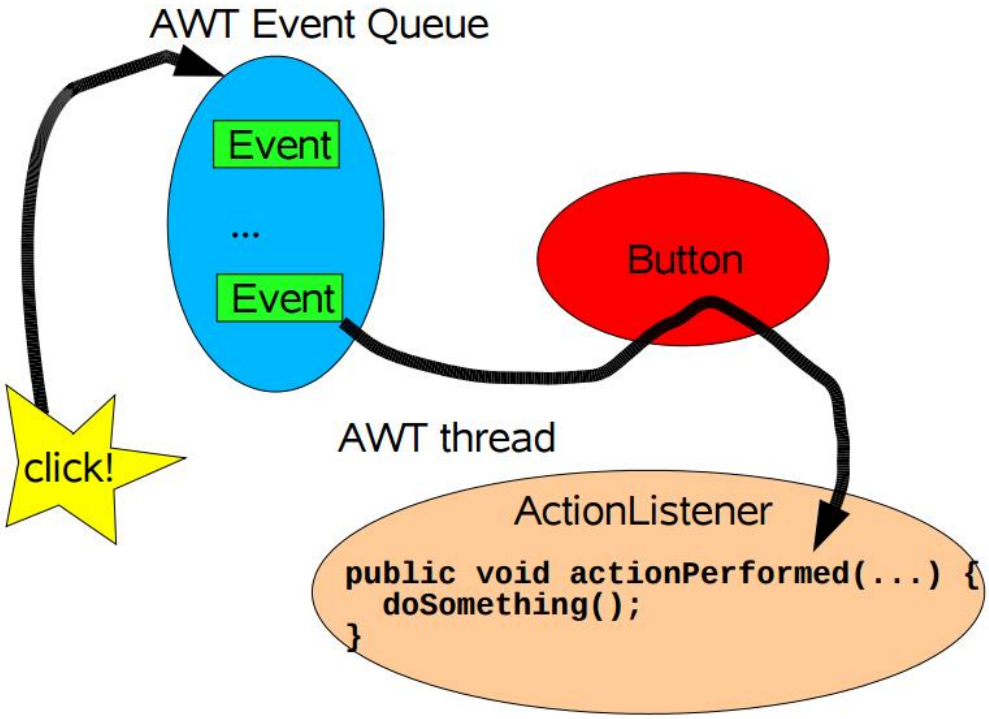
package02.png

我们知道TCP/IP在发送消息的时候，可能会拆包(如上图1)！这就导致接收端无法知道什么时候收到的数据是一个完整的数据。例如:发送端分别发送了ABC,DEF,GHI三条信息，发送时被拆成了AB,CDRFG,H,I这四个包进行发送，接受端如何将其进行还原呢？在BIO模型中，当读不到数据后会阻塞，而NIO中不会!所以需要自行进行处理!例如，以换行符作为判断依据，或者定长消息发生，或者自定义协议!

NIO虽然性能高，但是编码复杂，且需要处理半包问题！为了方便的进行NIO开发，就有了Reactor模型!

Reactor模型

- AWT Events



awt.jpg

Reactor模型和AWT事件模型很像，就是将消息放到了一个队列中，通过异步线程池对其进行消费！

Reactor中的组件

- Reactor:Reactor是IO事件的派发者。
- Acceptor:Acceptor接受client连接，建立对应client的Handler，并向Reactor注册此Handler。
- Handler:和一个client通讯的实体，按这样的过程实现业务的处理。一般在基本的Handler基础上还会有更进一步的层次划分， 用来抽象诸如decode，process和encoder这些过程。比如对Web Server而言，decode通常是HTTP请求的解析， process的过程会进一步涉及到Listener和Servlet的调用。业务逻辑的处理在Reactor模式里被分散的IO事件所打破， 所以Handler需要有适当的机制在所需的信息还不全（读到一半）的时候保存上下文，并在下一次IO事件到来的时候（另一半可读了）能继续中断的处理。为了简化设计，Handler通常被设计成

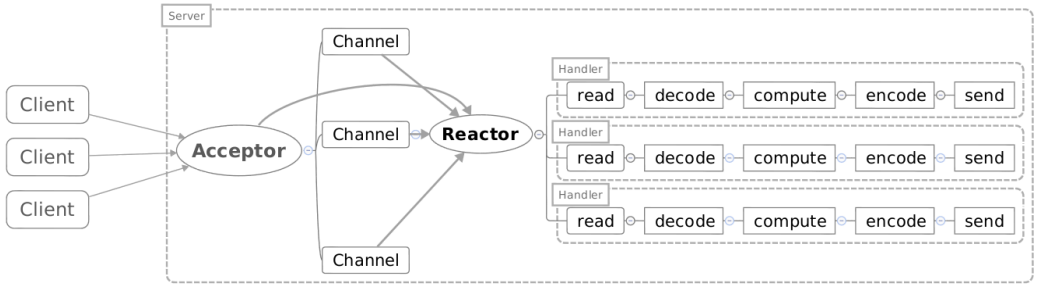
状态机，按GoF的state pattern来实现。

对应上面的NIO代码来看:

- Reactor：相当于有分发功能的Selector
- Acceptor：NIO中建立连接的那个判断分支
- Handler：消息读写处理等操作类

Reactor从线程池和Reactor的选择上可以细分为如下几种：

Reactor单线程模型

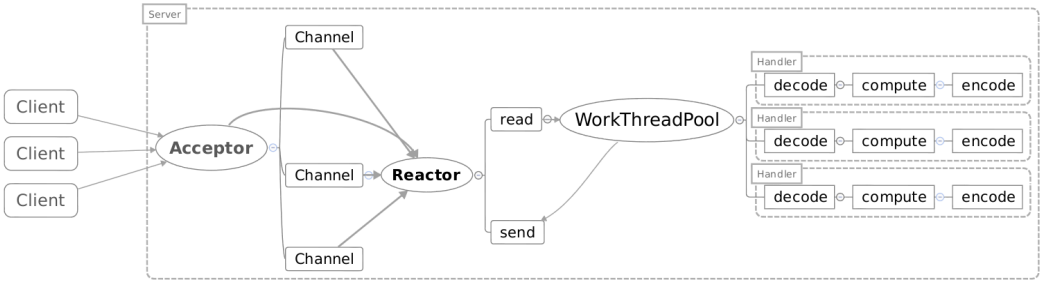


reactor1.png

这个模型和上面的NIO流程很类似，只是将消息相关处理独立到了Handler中去了！

虽然上面说到NIO一个线程就可以支持所有的IO处理。但是瓶颈也是显而易见的！我们看一个客户端的情况，如果这个客户端多次进行请求，如果在Handler中的处理速度较慢，那么后续的客户端请求都会被积压，导致响应变慢！所以引入了Reactor多线程模型!

Reactor多线程模型

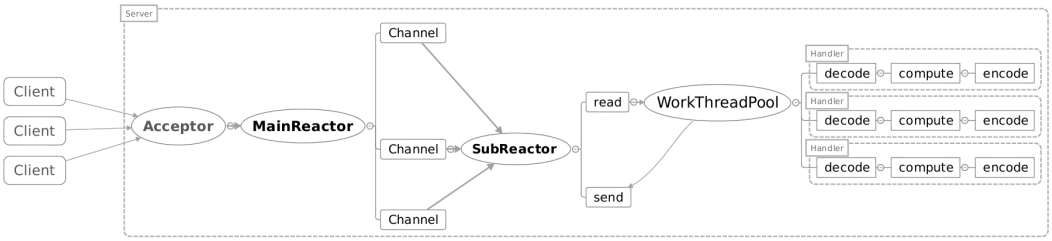


reactor2.png

Reactor多线程模型就是将Handler中的IO操作和非IO操作分开，操作IO的线程称为IO线程，非IO操作的线程称为工作线程!这样的话，客户端的请求会直接被丢到线程池中，客户端发送请求就不会堵塞！

但是当用户进一步增加的时候，Reactor会出现瓶颈！因为Reactor既要处理IO操作请求，又要响应连接请求！为了分担Reactor的负担，所以引入了主从Reactor模型!

主从Reactor模型



reactor3.png

主Reactor用于响应连接请求，从Reactor用于处理IO操作请求！

Netty

Netty是一个高性能NIO框架，其是对Reactor模型的一个实现！

- Netty客户端代码

```
EventLoopGroup workerGroup = new NioEventLoopGroup();
try {
    Bootstrap b = new Bootstrap();
    b.group(workerGroup);
    b.channel(NioSocketChannel.class);
    b.option(ChannelOption.SO_KEEPALIVE, true);
    b.handler(new ChannelInitializer<SocketChannel>() {
        @Override
        public void initChannel(SocketChannel ch) throws Exception {
            ch.pipeline().addLast(new TimeClientHandler());
        }
    });
    ChannelFuture f = b.connect(host, port).sync();
    f.channel().closeFuture().sync();
} finally {
    workerGroup.shutdownGracefully();
}
```

- Netty Client Handler

```
public class TimeClientHandler extends ChannelInboundHandlerAdapte
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        ByteBuf m = (ByteBuf) msg;
        try {
            long currentTimeMillis =
                (m.readUnsignedInt() - 2208988800L) * 1000L;
            System.out.println(new Date(currentTimeMillis));
            ctx.close();
        } finally {
            m.release();
        }
    }
    @Override
    public void exceptionCaught(ChannelHandlerContext ctx,
            Throwable cause) {
        cause.printStackTrace();
        ctx.close();
    }
}
```

```
EventLoopGroup bossGroup = new NioEventLoopGroup();
EventLoopGroup workerGroup = new NioEventLoopGroup();
try {
    ServerBootstrap b = new ServerBootstrap();
    b.group(bossGroup, workerGroup)
     .channel(NioServerSocketChannel.class)
     .childHandler(new ChannelInitializer<SocketChannel>() {
        @Override
        public void initChannel(SocketChannel ch) throws Exception {
            ch.pipeline().addLast(new TimeServerHandler());
        }
     })
     .option(ChannelOption.SO_BACKLOG, 128)
     .childOption(ChannelOption.SO_KEEPALIVE, true);
    // Bind and start to accept incoming connections.
    ChannelFuture f = b.bind(port).sync();
    f.channel().closeFuture().sync();
} finally {
    workerGroup.shutdownGracefully();
    bossGroup.shutdownGracefully();
}
```

- Netty Server Handler

```
public class TimeServerHandler extends ChannelInboundHandlerAdapt
    @Override
    public void channelActive(final ChannelHandlerContext ctx) {
        final ByteBuf time = ctx.alloc().buffer(4);
        time.writeInt((int)
            (System.currentTimeMillis() / 1000L + 2208988800L));
        final ChannelFuture f = ctx.writeAndFlush(time);
        f.addListener(new ChannelFutureListener() {
            @Override
            public void operationComplete(ChannelFuture future) {
                assert f == future;
                ctx.close();
            }
        });
    }
    @Override
    public void exceptionCaught(ChannelHandlerContext ctx,
                                Throwable cause) {
        cause.printStackTrace();
        ctx.close();
    }
}
```

y服务器代码来看，与Reactor模型进行对应！

tLoopGroup就相当于是Reactor，bossGroup对应主

r,workerGroup对应从Reactor

eServerHandler就是Handler

开头的方法配置的是客户端channel，非child开头的方法配置的是服务端channel

具体Netty内容，请访问Netty官网！

Netty的问题

Netty开发中一个很明显的问题就是回调，一是打破了线性编码习惯，二就是Callback Hell！

看下面这个例子：

```
a.doing1();  //1
a.doing2();  //2
a.doing3();  //3
```

1,2,3处代码如果是同步的，那么将按顺序执行！但是如果不是同步的呢？我还是希望2在1之后执行，3在2之后执行！怎么办呢？想想AJAX!我们需要写类似如下这样的代码！

```
a.doing1(new Callback(){
    public void callback(){
        a.doing2(new Callback(){
            public void callback(){
                a.doing3();
            }
        })
    }
});
```

法解决这个问题呢？其实不难，实现一个类似Future的功能！取结果时，进行阻塞，当得到结果后再继续往下走！实现方案，使用锁了，还有一个就是使用RingBuffer。经测试，使用比使用锁TPS有2000左右的提高！

pigeon.png

参考资料

- Socket为什么要翻译成套接字?
- Reactor论文
- Doug Lea《Scalable IO in Java》
- Netty源码
- 剖析Disruptor:为什么会这么快?
- 剖析Disruptor:为什么会这么快?（中文）
- Java SE1.6中的Synchronized
- 线程安全的无锁RingBuffer实现
- Java NIO类库Selector机制解析（上）
- Java NIO类库Selector机制解析（下）