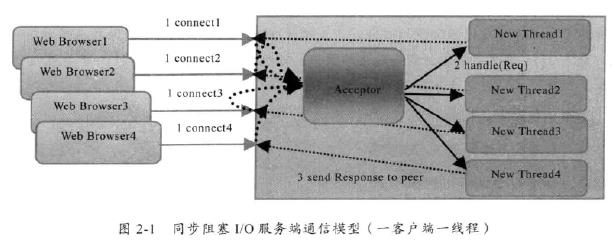
<http://ifeve.com/netty5-user-guide/>

BIO服务端通信模型：采用BIO通信模型的服务端，通常由一个独立的Acceptor线程负责监听客户端的连接，它接收到客户端连接请求之后为每个客户端创建一个新的线程进行链路处理，处理完成之后，通过输出流返回应答给客户端，线程销毁。这就是典型的一请求一应答通信模型。



此模型最大的问题就是缺乏弹性伸缩能力，当客户端并发访问量增加后，服务端的线程个数和客户端并发访问数呈1:1的正比关系 ，由于 线程是java虚拟机非常宝贵的系统 资源 ，当线程数膨胀之后，系统的性能将急剧下降，随着并发访问量的继续增大，系统会发生线程堆溢出，创建新线程失败等问题，最终导致进程宕机或僵死，不对外提供服务。

与Socket类和ServerSocket类相对应，NIO也提供了SocketChannel和ServerSocketChannel两种不同的套接字通道实现 。这两种新增的通道都支持阻塞和非阻塞两种模式。阻塞模式使用非常简单，但是性能和可靠性都不好，非阻塞模式正好相反。一般来说，低负载，低并发的应用程序可以选择同步阻塞I/O以降低编程复杂度;对于高负载，高并发的网络应用使用NIO的非阻塞模式进行开发。

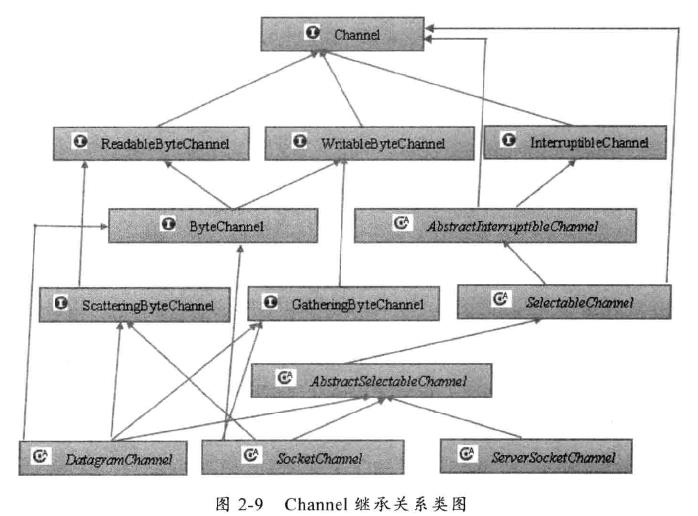
1.缓冲区Buffer

Buffer是一个对象，它包含一些要写入或者要读出的数据。在NIO中所有数据都是用缓冲区处理的。在读取数据时，它是直接读到缓冲区中的；在写入数据时，写入到缓冲区中。任何时候访问NIO中的数据，都是通过缓冲区进行操作。缓冲区实质上是一个数组。最常用的是ByteBuffer, CharBuffer,ShortBuffer,IntBuffer,LongBuffer,FloatBuffer,DoubleBuffer.

2.通道Channel

Channel是一个通道，它就像自来水管一样，网络数据通过Channel读取和写入.通道与流的不同之处在于通道是双向的，流只是在一个方向上移动（一个流必须是InputStream或者OutputStream的子类),而通道可以用于读，写或二者同时进行。因数Channel是全双工的，所以它可以比流更好的映射底层操作系统的API.

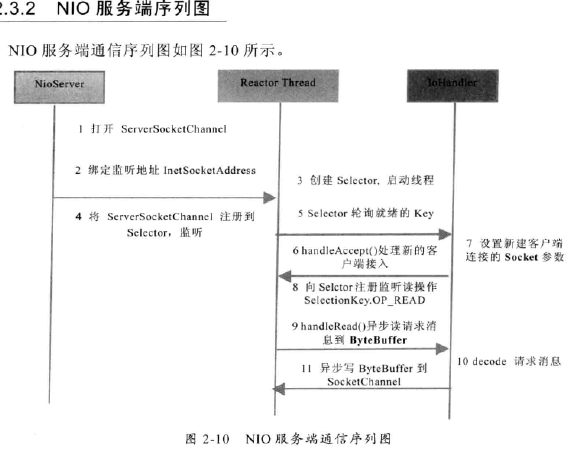
Channel可以分为两大类:用于网络读写的SelectableChannel和用于文件操作的FileChannel.



3.多路复用器Selector

多路复用器提供选择已经就绪的任务的能力。简单来讲，Selector会不断的轮询注册在其上的Channel,如果某个Channle上面发生读或者写事件，这个Channel就处于就绪状态，会被Selector轮询出来，然后通过SelectionKey可以获取就绪Channel的集合，进行后续的I/O操作。

一个多路复用器Selector可以同时轮询多个Channel,由于JDK使用了epoll()代替传统的select实现，所以它并没有最大连接句柄1024/2048的限制。这也就是说只需要一个线程负责Selector的轮询，就可以接入成千上万的客户端。



NIO服务端的主要创建过程：

1.打开ServerSocketChannel,用于监听客户端的连接,它是所有客户端连接的父管道:

ServerSocketChannel acceptorSvr = ServerSocketChannel.open();

2.绑定监听端口，设置连接为非阻塞模式:

acceptorSvr.socket().bind(new InetSocketAddress(inetAddress.getByName("ip"),port);

acceptorSvr.configureBlocking(false);

3.创建Reactor线程，创建多路复用器并启动线程

Selector selector = Selector.open();

New Thread(new ReactorTask()).start();

4.将ServerSocketChannel注册到Reactor线程的多路复用器Selector上，监听ACCEPT事件

SelectionKey key = acceptorSvr.register(selector, SelectionKey.OP\_ACCEPT, ioHandler);

5.多路复用器在线程run方法的无限循环体内轮询准备就绪的key

int num = selector.select();

Set selecedKeys == selector.selectedKeys();

Iterator it = selectedKeys.iterator();

while (it.hasNext()) {

SelectionKey key = (SelectionKey)it.next();

//...deal with I/O event ...

}

6.多路复用器监听到有新的客户端接入，处理新的接入请求，完成TCP三次握手，建立物理链路

SocketChannel channel = svrChannel.accept();

7.设置客户端链路为非阻塞模式

channel.configureBlocking(false);

channel.socket().setReuseAddress(true);

8.将新接入的客户端连接注册到Reactor线程的多路复用器上，监听读操作，读取客户端发送的网络消息

SelectionKey key = socketChannel.register(selector, SelectionKey.OP\_READ, ioHandler);

9.异步读取客户端请求消息到缓冲区

int readNumber = channel.read(receivedBuffer);

10.对ByteBuffer进行编解码，如果有半包消息指针reset，继续读取后续的报文，将解码成功的消息封装成 Task,投递到业务线程池中，进行业务逻辑编排

Object message = null;

while(buffer.hasRemain()) {

byteBuffer.mark();

Object message = decode(byteBuffer);

if (message == null) {

byteBuffer.reset();

break;

}

messageList.add(message);

}

if (!byteBuffer.hasRemain()) {

byteBuffer.clear();

} else {

byteBuffer.compact();

}

if(messageList != null & !messageList.isEmpty()) {

for(Object messageE : messageList){

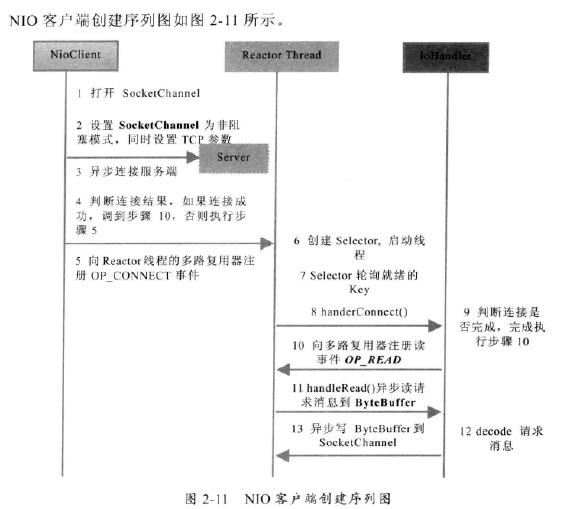
handlerTask(messageE);

}

}

11.将pojo对象encode成ByteBuffer，调用SocketChannel的异步write接口，将消息异步发送给客户端

socketChannel.write(buffer);



NIO客户端操作步骤：

1.打开SocketChannel，绑定客户端本地地址（可选，默认系统会随机分配一个可用的本地地址）

SocketChannel clientChannel = SocketChannel.open();

2.设置SocketChannel为非阻塞模式，同时设置客户端连接的TCP参数

clientChannel.configureBlocking(false);

socket.setReuseAddress(true);

socket.setReceiveBufferSize(BUFFER\_SIZE);

socket.setSendBufferSize(BUFFER\_SIZE);

3.异步连接服务端

boolean connected = clientChannel.connect(new InetSocketAddress("IP","port");

4.判断是否连接成功，如果连接成功，则直接注册读状态位到多路复用器中，如果当前没有连接成功（异步连接，返回false,说明客户端已经发送sync包，服务端没有返回ack包，物理链路还没有建立)

if (connected){

clientChannel.register(selector, SelectionKey.OP\_READ, ioHandler)

} else {

clientChannel.register(selector, SelectionKey.OP\_CONNECT, ioHandler);

}

5.向Reactor线程的多路复用器OP\_CONNECT状态位，监听服务端的TCP ACK应答

clientChannel.register(selector, SelectionKey.OP\_CONNECT,ioHandler);

6.创建Reactor线程，创建多路复用器并启动线程

Selector selector = Selector.open();

New Thread(new ReactorTask()).start();

7.多路复用器在线程run方法的无限循环体内轮询准备就绪的Key

int num = selector.select();

Set selectedkeys = selector.selectedKeys();

Iterator it = selectedKeys.iterator();

while(it.hasNext()){

SelectionKey key = (SelectionKey)it.next();

//....deal with I/O event...

}

8.接收connect事件进行处理

if (key.isConnectable()) {

//handlerConnect

}

9.判断连接结果，如果连接成功，注册读事件到多路复用器

if (channel.finishConnect())

registerRead();

10.注册读事件到多路复用器

clientChannel.register(selector, SelectionKey.OP\_READ, ioHandler);

11.异步读客户端请求消息到缓冲区

int readNumber = channel.read(receivedBuffer);

12.对ByteBuffer进行编解码，如果有半包消息接收缓冲区reset，继续读取后续的报文，将解码成功的消息封装成 Task,投递到业务线程池中，进行业务逻辑编排

Object message = null;

while(buffer.hasRemain()) {

byteBuffer.mark();

Object message = decode(byteBuffer);

if (message == null) {

byteBuffer.reset();

break;

}

messageList.add(message);

}

if (!byteBuffer.hasRemain()) {

byteBuffer.clear();

} else {

byteBuffer.compact();

}

if(messageList != null & !messageList.isEmpty()) {

for(Object messageE : messageList){

handlerTask(messageE);

}

}

13.将pojo对象encode成ByteBuffer，调用SocketChannel的异步write接口，将消息异步发送给客户端

socketChannel.write(buffer);

JDK1.7新异步非阻塞I/O(AIO),如果使用AsynchronousScoketChannel时就不需要注册Selector了，但是实现CompletionHandler去处理，它是异步连接成功之后的方法回调。

NIO类库支持非阻塞读和写操作，相比于这前的同步阻塞读和写，它是异步的。它的关键是多路复用I/O技术，多路复用的核心就是通过Selector来轮询注册在其上的Channel,当发现某个或者多个Channel处于就绪状态后，从阻塞状态返回就绪的Channel的选择键集合，进行I/O操作。

伪异步I/O:它的概念完全来源于实践，在NIO之前为了解决Tomcat通信线程同步I/O导致业务线程被挂住的问题，大家想到了一个办法：在通信线程和业务线程之间做个缓冲区，这个缓冲区用于隔离I/O线程和业务间的直接访问，这样业务线程就不会被I/O线程阻塞。而对于后端的业务来说，将消息或者Task放到线程池后就返回了，它不再直接访问I/O线程或者进行I/O读写，这样也就不会被同步阻塞。类似的设计还包括前端启动一组线程，将接收的客户端封装成Task，放在后端的线程池执行，用于解决一连接一线程问题。

用什么方式还需要看业务：如果客户端并发连接数不多，周边对接的网元不多，服务器的负载也不重，那就完全没有必要选择NIO做服务端；如果是相反情况，那就要考虑选择合适的NIO框架进行开发。

为什么不选择java原生的NIO编程原因：

1.NIO的类库和API繁杂，使用麻烦，你需要熟练掌握Selector, ServerSocketChannel,SocketChannel,ByteBuffer等。

2.需要具备其他的额外技能做铺垫，如多线程

3.可靠性能力补齐，工作量和难度大。要处理客户端断连重连，网络闪断，半包读写，失败缓存 ，网络拥塞和异常码流的处理等问题。

为什么选择Netty

它是业界最流行的NIO框架之一，它的健壮性，功能，性能，可定制性和扩展性在同类框架中都是首屈一指的。API简单，开发门槛低。功能强大，预置了多种编解码功能，支持多种主流协议。定制能力强，可以通过ChannelHandler对通信框架进行灵活地扩展。性能高；成熟，稳定。

Netty入门

1.下载Netty http://netty.io/ 解压后选择netty-all-5.0.0.Alpha1.jar

回顾一下NIO服务端开发步骤：

1.创建ServerSocketChannel, 配置它为非阻塞模式；

2.绑定监听，配置TCP参数，例如backlog大小

3.创建一个独立的I/O线程，用于轮询多路复用器Selector上，监听SelectionKey.ACCEPT;

4.创建Selector,将之前创建的ServerSocketChannel注册到Selector上，监听SelectionKey.ACCEPT.

5.启动I/O线程，在循环体中执行Selector.select()方法，轮询就绪的Channel.

6.当轮询到了处于就绪状态的Channel时，需要对其进行判断，如果是OP\_ACCEPT状态，说明 是新的客户端接入 ，则调用ServerSocketChannel.accept()方法接受新的客户端。

7.设置新接入的客户端链路SocketChannel为非阻塞模式，配置其他的一些TCP参数。

8.将SocketChannel注册到Selector，监听OP\_READ操作位；

9.如果轮询的Channel为OP\_READ，则说明SocketChannel中有新的就绪的数据包需要读取，则构造ByteBuffer对象，读取数据包。

10.如果轮询的Channel为OP\_WRITE，说明还有数据没有发送完成，需要继续发送。

Netty代码片断：

//配置服务端的NIO线程组

EventLoopGroup boosGroup = new NioEventLoopGroup();

EventLoopGroup workerGroup= new NioEventLoopGroup();

ServerBootstrap b= new ServerBootstrap();

NioEventLoopGroup是个线程组，它包含了一组NIO线程，专门用于网络事件的处理，实际上它们就是Reactor线程组。这里创建两个的原因是一个boosGroup 用于服务端接受客户端的连接，另一个用于进行SocketChannel的网络读写。ServerBootstrap是Netty用于启动NIO服务端的辅助启动类，目的是降低服务器端的开发复杂度。客户端只需要一组线程组，因为它不需要接收。

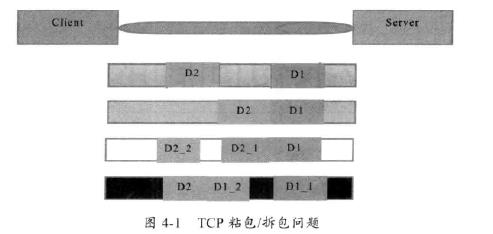
服务端启动辅助类配置完成之后，调用它的bind方法绑定监听端口，随后，调用它的同步阻塞方法sync等待绑定操作完成。完成之后Netty会返回一个ChannelFuture,它的功能类似于JDK的java.util.concurrent.Future,主要用于异步操作的通知回调。

处理理只需要继承ChannelHandlerAdapter对网络事件进行读写操作。

Tcp粘包/拆包问题的解决之道

无论是服务端还是客户端，当我们读取或者发送消息的时候，都需要考虑TCP底层的粘包/拆包机制。

TCP是个“流”协议，所谓流，就是没有界限的一串数据。大家可以想想河里的流水，它们是连成一片的，其间并没有分界线，TCP底层并不了解上层业务数据的具体含义，它会根据TcP缓冲区的实际情况进行包的划分，所以在业务上认为，一个完整的包可能会被TCP拆分成多个包进行发送，也就是可能把多个小的包封装成一个大的数据包发送，这就是所谓的tcp粘包和拆包的问题。



假设客户端分别发送了两个数据包D1和D2给服务端，由于服务端一次读取到的字节数是不确定的，可能存在以下4种情况：

1.服务端分两次读取到了两个独立的数据包,分别是D1和D2,没有粘包和拆包。

2.服务端一次接收到两个数据包，D1和D2粘合在一起，被称为Tcp粘包。

3.服务端分别两次读取到了两个数据包，第一次读取到了完整的D1包和D2包的部分内容，第二次读取到了D2包的剩余内容，这被称为TCP拆包；

4.服务端分两次读取到两个数据包，第一次读取到了D1包的部分内容D1\_1，第二次读取到了D1包的剩余内容D1\_2和D2包的整包。

如果此时服务端tcp接收滑窗非常小，而数据包D1和D2比较大，很有可能会发生第5种可能，即服务端分多次才能将D1和D2包接收完全，期间发生多次拆包。

TCP粘包/拆包发生的原因：

1.应用程序write写入的字节大小大于套接口发送缓冲区大小；

2.进行MSS大小的TCP分段；

3.以太网帧的payload大于 MTU进行IP分片。



粘包问题的解决策略：

由于底层的TCP无法理解上层的业务数据，所以在底层是无法保证数据包不被拆分和重组的，这个问题只能通过上层的应用协议栈设计来解决，根据业力量的主流协议解决方案，可以归纳如下：

1.消息定长，例如每个报文的大小为固定长度200字节，如果不够，空位补空格；

2.在包尾增加回车换行符进行分割，例如ftp协议；

3.将消息分为消息头和消息体，消息头中包含表示消息总长度（或者消息体长度）的字段，通常设计思想为消息头的第一个字段使用int32来表示消息的总长度。

4.更复杂的应用层协议。

如果并发量大或发送大报文之后，就会存在粘包和拆包问题。如果代码没有考虑到就会出现解码错位或错误。 Netty有半包解码器解决这问题 LineBasedFrameDecoder 和 StringDecoder来解决TCP粘包问题。

利用LineBasedFrameDecoder解决TCP粘包问题：

SocketChannel.pipeline().addLast(new LineBasedFrameDecoder(1024));

SocketChannel.pipeline().addLast(new StringDecoder());

LineBasedFrameDecoder的工作原理是它依次遍历ByteBuf中的可读字节，判断看是否有"\n"或"\r\n"，如果有，就以此位置为结束位置，从可读索引到结束位置区间的字节就级成了一行。它是以换行符为结束标志 的解码器，支持携带结束符或者不携带结束符两种解码方式，同时支持配置单行的最大长度。如果连续读取到最大长度后仍然没有发现换行符，就会抛出异常，同时忽略掉之前读到的异常码流。

StringDecoder的功能非常简单，就是将接收到的对象转换成字符串，然后继续调用后面的Handler。LineBasedFrameDecoder + StringDecoder组合就是按行切换的文本解码器，它被设计用来支持TCP的粘包和拆包。

TCP以流的方式进行数据传输，上层的应用协议为了对消息进行区分，往往采用如下4种方式：

1.消息长度固定，累计读取到长度总和为定长LEN的报文后，就认为读取到了一个完整的消息：将计数器置位，重新开始读取下一个数据报。

2.将回车换行符作为消息结束符。

3.将特殊的分隔符作为消息的结束标志，回车换行符就是一种特殊的结束分隔符。

4.通过在消息头中定义长度字段来标识 消息的总长度。

争对以上有这样的解码器：DelimiterBasedFrameDecoder和FixedLengthFrameDecoder;它们都能解决TCP粘包/拆包导致的读半包问题。

public void initChannel(SocketChannel ch){

ByteBuf delimeter = Unpooled.copiedBuffer("$\_".getBytes());

ch.pipeline().addLast(new DelimiterBasedFrameDecoder(1024, delimiter)); //1024代表单条信息长度

ch.pipeline().addLast(new StringDecoder());

ch.pipleline().addLast(new ....EchoServerHandler());

}

首先创建分隔符缓冲对象ByteBuf，如上使用"$\_"作为分隔符。创建DelimiterBasedFrameDecoder对象，将其加入到ChannelPipeline中。

类似以上addLast(new FixedLengthFrameDecoder(20)); //按照20个字节长度对请求消息进行截取

编解码技术：

基于java提供的对象输入/输出流ObjectInputStream和ObjectOutputStream可以直接把java对象作为可存储的字节数组写入文件，也可以传输到网络上。实现Serializable

Java序列化的主要目的是：网络传输；对象持久化；

当时行远程跨进程服务调用时，需要把被传输的java对象编码为字节数组或者ByteBuffer对象。而当远程服务读到Bytebuffer对象或者字节数组时，需要将其解码为发送时的Java对象。这被称为Java对象编解码技术。在(RPC)远程服务调用时还是有缺点，它无法跨语言，序列化后码流太大,序列化性能太低。

业界主流的编码解码框架：

Google的 Protobuf

它将数据结构以.proto文件进行描述，通过代码生成工具可以生成对应数据结构的pojo对象和Protobuf相关的方法和属性.它的特点：结构化数据存储格式(xml,json等); 高效的编解码性能;语言无关，平台无关，扩展性好；官方支持java,c++和Python三种语言；

xml解析的时间开销和xml为了可读性而牺牲的空间开销都非常大，因此不适合做高性能的通信协议。Protobuf使用二进制编码，在空间和性能上具有更大的优势 。

FaceBook和Thrift;

JBoss Marshalling:是一个java对象序列化api包，修正了jdk自带的序列化包的很多问题，兼容java.io.Serializable.优点：可插拨的类解析器，提供更加便捷的类加载定制策略，通过一个接口即可实现定制；可插拨的对象替换技术，不需要通过继承的方式；可插拨的预定义类缓存表，可以减小序列化的字节数组长度，提升常用类型的对象序列化性能；无须实现 java.io.Serializable接口，即可实现java序列化；通过缓存技术提升对象的序列化性能。

MessagePack编解码

它是一个高效的二进制序列化框架，像json一样支持不同语言间的数据交换，但是它的性能更快，序列化之后的码流也更小。

public void initChannel(SocketChannel ch){

ch.pipeline().addLast("msgpack decode", new MsgpackDecoder());

ch.pipeline().addLast("msgpack decode", new MsgpackEncoder());

}

粘包支持：

客户端，服务端一样。

public void initChannel(SocketChannel ch){

ch.pipeline().addLast("frameDecoder",new LengthFieldBasedFrameDecoder(65535, 0, 2, 0, 2));

ch.pipeline().addLast("msgpack decoder", new MsgpackDecoder());

ch.pipeline().addLast("encoder", new LengthFieldPrepender(2));

ch.pipline().addLast("encoder",new MsgpackEncoder());

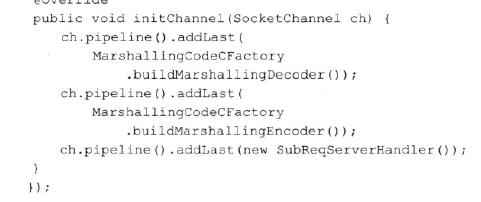
}

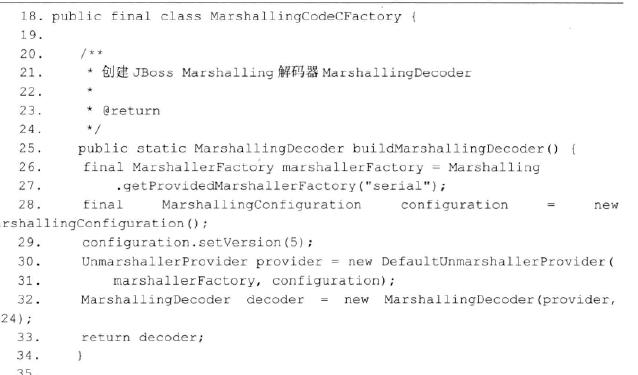
在MessagePack解码器之前增加LengthFieldBasedFrameDecoder,用于处理半包消息，这样后面的MsgpackDecoder接收到的永远是整包的消息。

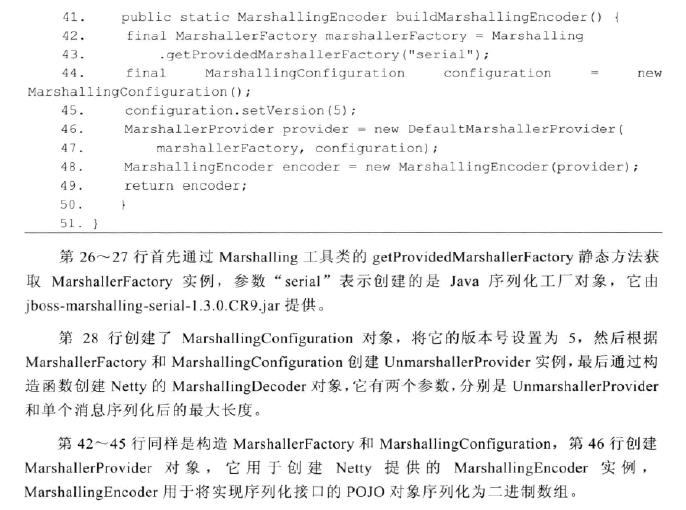
JBoss Marshalling

下载jboss-marshalling-1.3.0和jboss-marshalling-serial-1.3.0

<https://www.jboss.org/jbossmarshalling/downloads>



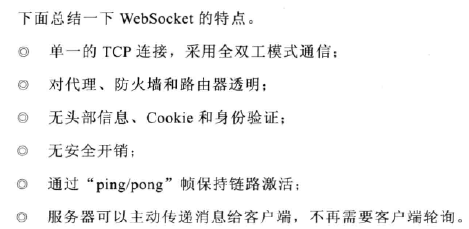




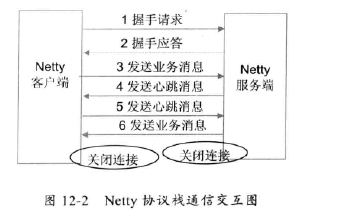
HTML5定义了WebSocket协议 ，能更好地节省服务器资源和带宽并达到实时通信。

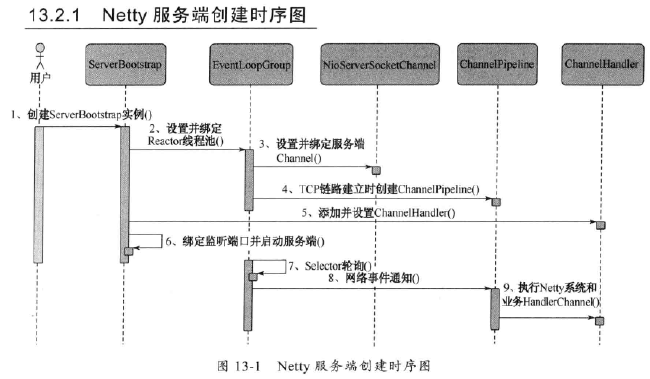
在WebSocket API中，浏览器和服务器只需要做一个握手的动作，然后浏览器和服务器之间就形成了一条快速通道，两都都可以直接互相传送数据了。

WebSocket基于Tcp双向全工进行消息传递，在同一时刻，即可以发送消息，也可能接收消息，相比http的半双工协议，性能得到很大提升。



WebSocket设计出来的目的就是要取代轮询和Comet技术，使客户端浏览器具备像c/s架构下桌面系统一样的实时通信能力。浏览器通过javascript向服务器发出建立Websocket连接的请求，连接建立以后，客户端和服务器端可以通过tcp连接直接交换数据。因为websocket连接本质上就是一个tcp连接，所以在数据传输和稳定性和数据传输量的大小方面，有很大的性能优势。





1.创建ServerBootstrap实例。ServerBootstrap是Netty服务端的启动辅助类，它提供了一系列的方法用于设置服务端启动相关的参数。

2.设置并绑定Reactor线程池。Netty的Reactor线程池是EventLoopGroup,它实际就是EventLoop的数组。EventLoop的职责是处理所有注册到本线程多路复用器Selector上的Channel,Selector的轮询操作是由绑定的EventLoop线程run方法驱动，在一个循环体内循环执行。EventLoop的职责不仅仅是处理网络I/O事件，用户自定义的Task和定时任务Task也统一由EventLoop负责处理，这样线程模型就实现了统一。从调试层面看，也不存在从EventLoop线程中再启动其他类型的线程用于异步执行另外的任务，这样就避免了多线程并发操作和锁竞争，提升了I/O线程的处理和调度性能。

3.设置并绑定服务端Channel.作为NIO服务端，需要创建ServerSocketChannel,Netty对原生的NIO类库进行了封装，对应实是NioServerSocketChannel.

4.链路建立的时候创建并初始化ChannelPipeline.ChannelPipeline并不是NIO服务端必需的，它本质誻一个负责处理网络事件的职责链，负责管理和执行ChannelHandler.

5.初始化ChannelPipeline完成之后，添加并设置ChannelHandler.ChannelHandler是Netty提供给用户定制和扩展的关键接口。它可以完成大多数的功能定制，如消息编解码，心跳，安全认证等。

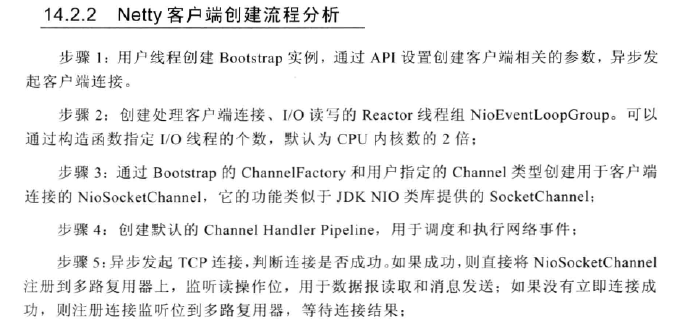
6.绑定并启动监听端口。

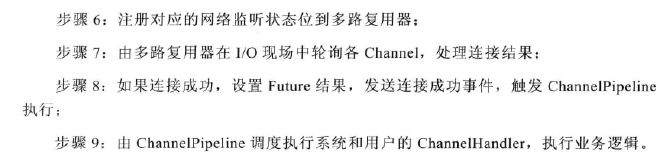
7.Selector轮询。由Reactor线程NioEventLoop负责调度和执行Selector轮询操作，选择准备就绪的Channel集合。

8.当轮询到准备就绪的Channel之后，就由Reactor线程NioEventLoop执行ChannelPipeline的相应方法，最终调度并执行ChannelHandler.

9.执行Netty系统ChannelHandler和用户添加定制的ChannelHandler.ChannelPipeline根据网络事件的类型，调度并执行ChannelHandler.

NioEventLoopGroup实际就是Reactor线程池，负责调度和执行客户端的接入，网络读写事件的处理，用户自定义任务和定时任务的执行。





Netty的ChannelPipeline和ChannelHandler机制类似于Servlet和Filter过滤器，这类拦截器实际上是职责链模式的一种变形，主要是为了方便事件的拦截和用户业务逻辑的定制。

ChannelPipeline是ChannelHandler的容器，它负责ChannelHandler的管理和事件拦截与调度。

