# 简介

# 入门程序

public class Server  
{  
 public static void main(String[] args) throws Exception  
 {  
 EventLoopGroup bossGroup = new NioEventLoopGroup();  
 EventLoopGroup workerGroup = new NioEventLoopGroup();  
 try  
 {  
 ServerBootstrap serverBootstrap = new ServerBootstrap();  
 serverBootstrap.group(bossGroup,workerGroup).channel(NioServerSocketChannel.class)  
 .childHandler(new ServerInitializer());  
 ChannelFuture future = serverBootstrap.bind(8899).sync();  
 future.channel().closeFuture().sync();  
 }  
 finally  
 {  
 bossGroup.shutdownGracefully();  
 workerGroup.shutdownGracefully();  
 }  
  
 }  
}

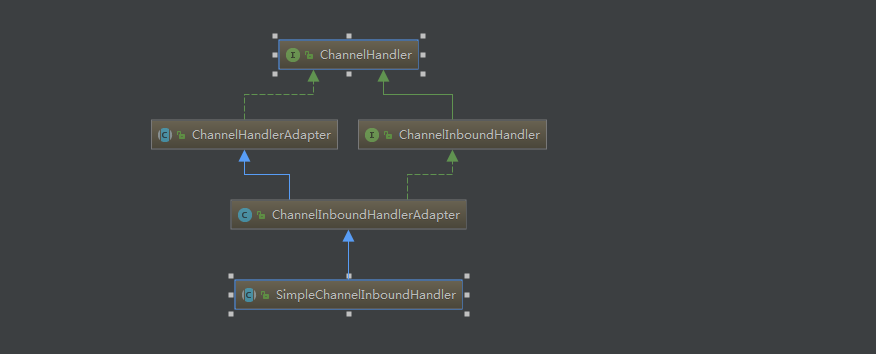
public class ServerInitializer extends ChannelInitializer<SocketChannel>  
{  
 protected void initChannel(SocketChannel ch) throws Exception{  
 ChannelPipeline channelPipeline = ch.pipeline();  
 channelPipeline.addLast("httpServerCodec",new HttpServerCodec());  
 channelPipeline.addLast("serverHandler",new HttpSeverHandler());  
 }  
}

public class HttpSeverHandler extends SimpleChannelInboundHandler<HttpObject>  
{  
 protected void channelRead0(ChannelHandlerContext ctx, HttpObject msg) throws Exception  
 {  
 if(msg instanceof HttpRequest)  
 {  
 ByteBuf content = Unpooled.*copiedBuffer*("hello world", CharsetUtil.*UTF\_8*);  
 FullHttpResponse response = new DefaultFullHttpResponse(HttpVersion.*HTTP\_1\_1*, HttpResponseStatus.*OK*,  
 content);  
 response.headers().set(HttpHeaderNames.*CONTENT\_TYPE*,"text/plain");  
 response.headers().set(HttpHeaderNames.*CONTENT\_LENGTH*,content.readableBytes());  
 ctx.writeAndFlush(response);  
 }  
 }  
}

这是一个简单的Hello World程序

## 回调方法执行流程

由于我们自定义的处理器HttpServerHandler继承了SimpleChannelInboundHandler类，该类有ChannelInboundHandlerAdapter类，ChannelInboundHandlerAdapter类中定义一些回调方法。如channelRegister和channelUnRegiser，channelActive和channelInactive方法等。而ChannelInboundHandlerAdapter有继承了ChannelHandlerAdapter类，该类由handlerAdd和handlerRemoved方法。



我们在HttpServerHandler类重写这些方法，观察方法执行的顺序：

public class HttpSeverHandler extends SimpleChannelInboundHandler<HttpObject>  
{  
 protected void channelRead0(ChannelHandlerContext ctx, HttpObject msg) throws Exception  
 {  
 if(msg instanceof HttpRequest)  
 {  
 HttpRequest request = (HttpRequest)msg;  
 System.*out*.println("request methon name:"+request.method().name());  
 URI url = new URI(request.uri());  
 if("/favicon.ico".equals(url.getPath()))  
 {  
 System.*out*.println("request url is favicon.ico");  
 return ;  
 }  
 ByteBuf content = Unpooled.*copiedBuffer*("hello world", CharsetUtil.*UTF\_8*);  
 FullHttpResponse response = new DefaultFullHttpResponse(HttpVersion.*HTTP\_1\_1*, HttpResponseStatus.*OK*,  
 content);  
 response.headers().set(HttpHeaderNames.*CONTENT\_TYPE*,"text/plain");  
 response.headers().set(HttpHeaderNames.*CONTENT\_LENGTH*,content.readableBytes());  
 ctx.writeAndFlush(response);  
 ctx.channel().close();  
 }  
 }  
 @Override  
 public void channelRegistered(ChannelHandlerContext ctx) throws Exception{  
 System.*out*.println("channel registered");  
 super.channelRegistered(ctx);  
 }  
  
 @Override  
 public void channelUnregistered(ChannelHandlerContext ctx) throws Exception{  
 System.*out*.println("channel unregistered");  
 super.channelUnregistered(ctx);  
 }  
 @Override  
 public void channelActive(ChannelHandlerContext ctx) throws Exception{  
 System.*out*.println("channel active");  
 super.channelActive(ctx);  
 }  
 @Override  
 public void channelInactive(ChannelHandlerContext ctx) throws Exception{  
 System.*out*.println("channel inactive");  
 super.channelInactive(ctx);  
 }  
  
 @Override  
 public void handlerAdded(ChannelHandlerContext ctx) throws Exception{  
 System.*out*.println("handler Added");  
 super.handlerAdded(ctx);  
 }  
  
 @Override  
 public void handlerRemoved(ChannelHandlerContext ctx) throws Exception{  
 System.*out*.println("handler removed");  
 super.handlerRemoved(ctx);  
 }  
}

测试结果：

handler Added

channel registered

channel active

request methon name:GET

channel inactive

channel unregistered

handler removed

从结果我们可以发现，首先执行是handlerAdded方法，这是客户端已经于服务器建立了链接，在执行channelRegister方法，channelActive方法。最后在客户端于服务端失去连接时会调用handlerRemoved方法。

## 多客户端通信程序

public class MyChatServer  
{  
 public static void main(String[] args) throws Exception  
 {  
 EventLoopGroup bossGroup = new NioEventLoopGroup();  
 EventLoopGroup workerGroup = new NioEventLoopGroup();  
 try  
 {  
 ServerBootstrap serverBootstrap = new ServerBootstrap();  
 serverBootstrap.group(bossGroup,workerGroup).channel(NioServerSocketChannel.class)  
 .childHandler(new MyChatServerInitalizer());  
 ChannelFuture future = serverBootstrap.bind(3434).sync();  
 future.channel().closeFuture().sync();  
 }  
 finally{  
 bossGroup.shutdownGracefully();  
 workerGroup.shutdownGracefully();  
 }  
 }  
}

public class MyChatServerInitalizer extends ChannelInitializer<SocketChannel>  
{  
 @Override  
 protected void initChannel(SocketChannel ch) throws Exception{  
 ChannelPipeline pipeline = ch.pipeline();  
 pipeline.addLast(new DelimiterBasedFrameDecoder(4096, Delimiters.*lineDelimiter*()));  
 pipeline.addLast(new StringDecoder(CharsetUtil.*UTF\_8*));  
 pipeline.addLast(new StringEncoder(CharsetUtil.*UTF\_8*));  
 pipeline.addLast(new MyChatServerHandler());  
 }  
}

public class MyChatServerHandler extends SimpleChannelInboundHandler<String>  
{  
 //为啥需要static修饰  
 private static ChannelGroup *group* = new DefaultChannelGroup(GlobalEventExecutor.*INSTANCE*);  
 @Override  
 protected void channelRead0(ChannelHandlerContext ctx, String msg) throws Exception{  
 Channel channel = ctx.channel();  
 System.*out*.println(*group*.size());  
 *group*.forEach(ch->{  
 if(channel!=ch)  
 {  
 ch.writeAndFlush(channel.remoteAddress()+" 发送消息："+msg+"\n");  
 }  
 else  
 {  
 ctx.writeAndFlush("[自己] "+msg+"\n");  
 }  
 });  
  
 }  
 @Override  
 public void handlerAdded(ChannelHandlerContext ctx) throws Exception  
 {  
 Channel channel = ctx.channel();  
 *group*.writeAndFlush("[服务器]-"+channel.remoteAddress()+" 加入\n");  
 *group*.add(channel);  
 }  
 @Override  
 public void handlerRemoved(ChannelHandlerContext ctx) throws Exception  
 {  
 Channel channel = ctx.channel();  
 ctx.writeAndFlush("[服务器]-"+channel.remoteAddress()+" 离开\n");  
 }  
 @Override  
 public void channelActive(ChannelHandlerContext ctx) throws Exception{  
 Channel channel = ctx.channel();  
 System.*out*.println(channel.remoteAddress()+" 上线");  
 }  
 @Override  
 public void channelInactive(ChannelHandlerContext ctx) throws Exception{  
 Channel channel = ctx.channel();  
 System.*out*.println(channel.remoteAddress()+" 下线");  
 }  
}

public class MyChatClient  
{  
 public static void main(String[] args) throws Exception  
 {  
 EventLoopGroup eventLoopGroup = new NioEventLoopGroup();  
 try  
 {  
 Bootstrap bootstrap = new Bootstrap();  
 bootstrap.group(eventLoopGroup).channel(NioSocketChannel.class)  
 .handler(new MyChatClientInitalizer());  
 Channel channel = bootstrap.connect("localhost",3434).sync().channel();  
 try(BufferedReader reader = new BufferedReader(new InputStreamReader(System.*in*)))  
 {  
 while(true){  
 channel.writeAndFlush(reader.readLine()+"\r\n");  
 }  
 }  
 }finally{  
 eventLoopGroup.shutdownGracefully();  
 }  
 }  
}

public class MyChatClientInitalizer extends ChannelInitializer<SocketChannel>  
{  
 @Override  
 protected void initChannel(SocketChannel ch) throws Exception{  
 ChannelPipeline pipeline = ch.pipeline();  
 pipeline.addLast(new DelimiterBasedFrameDecoder(4096, Delimiters.*lineDelimiter*()));  
 pipeline.addLast(new StringDecoder(CharsetUtil.*UTF\_8*));  
 pipeline.addLast(new StringEncoder(CharsetUtil.*UTF\_8*));  
 pipeline.addLast(new MyChatClientHandler());  
 }  
}

public class MyChatClientHandler extends SimpleChannelInboundHandler<String>  
{  
 @Override  
 protected void channelRead0(ChannelHandlerContext ctx, String msg) throws Exception{  
 System.*out*.println(msg);  
 }  
}

## 实现心跳检测

## 实现WebSocket通信

public class MyServer  
{  
 public static void main(String[] args)throws Exception  
 {  
 EventLoopGroup bossGroup = new NioEventLoopGroup();  
 EventLoopGroup workerGroup = new NioEventLoopGroup();  
 try{  
 ServerBootstrap serverBootstrap = new ServerBootstrap();  
 serverBootstrap.group(bossGroup,workerGroup).channel(NioServerSocketChannel.class)  
 .handler(new LoggingHandler()).childHandler(new WebSocketChannelInitalizer());  
 ChannelFuture future = serverBootstrap.bind(new InetSocketAddress(6767)).sync();  
 future.channel().closeFuture().sync();  
 }  
 finally{  
 bossGroup.shutdownGracefully();  
 workerGroup.shutdownGracefully();  
 }  
 }  
}

public class WebSocketChannelInitalizer extends ChannelInitializer<SocketChannel>  
{  
 @Override  
 protected void initChannel(SocketChannel ch) throws Exception{  
 ChannelPipeline pipeline = ch.pipeline();  
 pipeline.addLast(new HttpServerCodec());  
 pipeline.addLast(new ChunkedWriteHandler());  
 pipeline.addLast(new HttpObjectAggregator(8192));  
 pipeline.addLast(new WebSocketServerProtocolHandler("/ws"));  
 pipeline.addLast(new TextWebSocketFrameHandler());  
 }  
}

public class TextWebSocketFrameHandler extends SimpleChannelInboundHandler<TextWebSocketFrame>  
{  
 @Override  
 protected void channelRead0(ChannelHandlerContext ctx, TextWebSocketFrame msg) throws Exception  
 {  
 System.*out*.println("收到消息："+msg.text());  
 ctx.writeAndFlush(new TextWebSocketFrame("服务器时间："+ LocalDateTime.*now*()));  
 }  
 @Override  
 public void handlerAdded(ChannelHandlerContext ctx) throws Exception{  
 System.*out*.println("handlerAdded:"+ctx.channel().id().asLongText());  
 }  
 @Override  
 public void handlerRemoved(ChannelHandlerContext ctx) throws Exception{  
 System.*out*.println("handlerAdded:"+ctx.channel().id().asLongText());  
 }  
  
 @Override  
 public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception{  
 System.*out*.println("异常发生");  
 ctx.close();  
 }  
}

客户端为：index.html。

## 整合Protocol Buffer

依赖添加：

<dependency>  
 <groupId>com.google.protobuf</groupId>  
 <artifactId>protobuf-java</artifactId>  
 <version>3.8.0</version>  
</dependency>  
<dependency>  
 <groupId>com.google.protobuf</groupId>  
 <artifactId>protobuf-java-util</artifactId>  
 <version>3.3.1</version>  
</dependency>

编写.proto文件

syntax = "proto2";  
package code.lsh.protobuf ;  
option optimize\_for = SPEED ;  
option java\_package = "code.lsh.protobuf";  
option java\_outer\_classname = "DataInf";  
message Student{  
 required string name = 1;  
 optional int32 age = 2;  
 optional string address = 3;  
}

public class ProtoBufServer  
{  
 public static void main(String[] args) throws Exception  
 {  
 EventLoopGroup bossGroup = new NioEventLoopGroup();  
 EventLoopGroup workerGroup = new NioEventLoopGroup();  
 try  
 {  
 ServerBootstrap serverBootstrap = new ServerBootstrap();  
 serverBootstrap.group(bossGroup,workerGroup).channel(NioServerSocketChannel.class)  
 .handler(new LoggingHandler()).childHandler(new ProtoBufServerInitalizer());  
 ChannelFuture future = serverBootstrap.bind(1212).sync();  
 future.channel().closeFuture().sync();  
 }finally{  
  
 }  
 }  
}

public class ProtoBufServerInitalizer extends ChannelInitializer<SocketChannel>  
{  
 @Override  
 protected void initChannel(SocketChannel ch) throws Exception{  
 ChannelPipeline pipeline = ch.pipeline();  
 pipeline.addLast(new ProtobufVarint32FrameDecoder());  
 pipeline.addLast(new ProtobufVarint32LengthFieldPrepender());  
 pipeline.addLast(new ProtobufDecoder(DataInf.Student.*getDefaultInstance*()));  
 pipeline.addLast(new ProtoBufServerHandler());  
 }  
}

public class ProtoBufServerHandler extends SimpleChannelInboundHandler<DataInf.Student>  
{  
 @Override  
 protected void channelRead0(ChannelHandlerContext ctx, DataInf.Student msg) throws Exception{  
 System.*out*.println(msg.getName());  
 System.*out*.println(msg.getAge());  
 System.*out*.println(msg.getAddress());  
 }  
}

public class ProtoBufClient  
{  
 public static void main(String[] args)throws Exception{  
 EventLoopGroup elg = new NioEventLoopGroup();  
 try  
 {  
 Bootstrap bootstrap = new Bootstrap();  
 bootstrap.group(elg).channel(NioSocketChannel.class).handler(new ProtoBufClientInitalizer());  
 ChannelFuture future = bootstrap.connect("localhost",1212).sync();  
 future.channel().closeFuture().sync();  
 }finally{  
 elg.shutdownGracefully();  
 }  
 }  
}

public class ProtoBufClientInitalizer extends ChannelInitializer<SocketChannel>  
{  
 @Override  
 protected void initChannel(SocketChannel ch) throws Exception{  
 ChannelPipeline pipeline = ch.pipeline();  
 pipeline.addLast(new ProtobufVarint32FrameDecoder());  
 pipeline.addLast(new ProtobufVarint32LengthFieldPrepender());  
 pipeline.addLast(new ProtobufDecoder(DataInf.Student.*getDefaultInstance*()));  
 pipeline.addLast(new ProtobufEncoder());  
 pipeline.addLast(new ProtoBufClientHandler());  
 }  
}

public class ProtoBufClientHandler extends SimpleChannelInboundHandler<DataInf.Student>  
{  
 @Override  
 protected void channelRead0(ChannelHandlerContext ctx, DataInf.Student msg) throws Exception{  
 }  
  
 @Override  
 public void channelActive(ChannelHandlerContext ctx) throws Exception{  
 DataInf.Student s1 = DataInf.Student.*newBuilder*().setName("张三")  
 .setAddress("黄河科技学院").setAge(20).build();  
 ctx.writeAndFlush(s1);  
 }  
}

使用这种编码，我们在客户端只能传递DataInf.Student类型，这样就失去了ProtoBuf的优势了。如在.proto文件中，如果定义了多个message，这样我们在客户端可以传递任意类型的message，这该如果解决？

定义.proto文件：

syntax = "proto2" ;  
package code.lsh.netty.demo6;  
option optimize\_for = SPEED;  
option java\_package = "code.lsh.netty.demo6";  
option java\_outer\_classname = "MyMessageData";  
message MyMessage{  
 enum DataType{  
 PersonType = 1;  
 DogType = 2;  
 CatType = 3;  
 }  
 required DataType data\_type = 1;  
 oneof dataBody{  
 Person person = 2;  
 Dog dog = 3;  
 Cat cat = 4;  
 }  
}  
message Person{  
 optional string name = 1;  
 optional int32 age = 2;  
 optional string address = 3 ;  
}  
message Dog{  
 optional string name = 1;  
 optional int32 age = 2 ;  
}  
message Cat{  
 optional string name = 1;  
 optional string city = 2;  
}

修改ProtoBufServerInitalizer和ProtoBufClientInitalizer类的initChannel方法

pipeline.addLast(new ProtobufDecoder(MyMessageData.MyMessage.*getDefaultInstance*()));

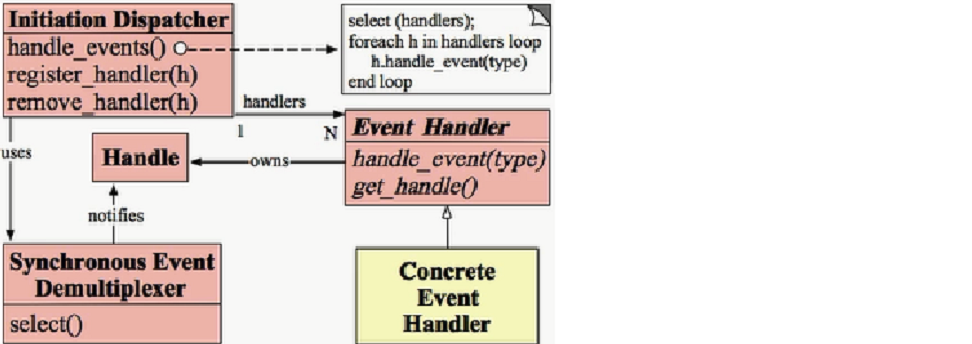
public class ProtoBufServerHandler extends SimpleChannelInboundHandler<MyMessageData.MyMessage>  
{  
 @Override  
 protected void channelRead0(ChannelHandlerContext ctx, MyMessageData.MyMessage msg) throws Exception  
 {  
 MyMessageData.MyMessage.DataType dataType = msg.getDataType();  
 if(dataType == MyMessageData.MyMessage.DataType.*PersonType*){  
 MyMessageData.Person person = msg.getPerson();  
 System.*out*.println(person.toString());  
 }else if(dataType == MyMessageData.MyMessage.DataType.*DogType*){  
 MyMessageData.Dog dog = msg.getDog();  
 System.*out*.println(dog.toString());  
 }else if(dataType == MyMessageData.MyMessage.DataType.*CatType*){  
 MyMessageData.Cat cat = msg.getCat();  
 System.*out*.println(cat);  
 }  
 }  
}

public class ProtoBufClientHandler extends SimpleChannelInboundHandler<MyMessageData.MyMessage>  
{  
 @Override  
 public void channelActive(ChannelHandlerContext ctx) throws Exception{  
 int num = new Random().nextInt(3);  
 MyMessageData.MyMessage myMessage = null;  
 if(num ==0){  
 myMessage = MyMessageData.MyMessage.*newBuilder*()  
 .setDataType(MyMessageData.MyMessage.DataType.*PersonType*)  
 .setPerson(MyMessageData.Person.*newBuilder*().setName("zhangsan")  
 .setAge(20).setAddress("luyi").build()).build();  
 }if(num == 1){  
 myMessage = MyMessageData.MyMessage.*newBuilder*()  
 .setDataType(MyMessageData.MyMessage.DataType.*DogType*)  
 .setDog(MyMessageData.Dog.*newBuilder*().setName("Dog").setAge(20).build()).build();  
 }else{  
 myMessage = MyMessageData.MyMessage.*newBuilder*()  
 .setDataType(MyMessageData.MyMessage.DataType.*CatType*)  
 .setCat(MyMessageData.Cat.*newBuilder*().setName("Cat").setCity("Cat\_Home").build()).build();  
 }  
 }  
 @Override  
 protected void channelRead0(ChannelHandlerContext ctx, MyMessageData.MyMessage msg) throws Exception{  
 }  
}

# Reactor模式

参考：nio.pdf

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



1. Handler(句柄或是描述符)：本质上表示一种资源，是由操作系统提供的：该资源用于表示一个个事件，比如说文件描述符，或是针对网络编程中的Socket描述符。时间既可以来自外部，也可以来自内部；外部事件比如说客户端的连接请求，客户端发送过来的数据等；内部事件比如说操作系统产生的定时任务等，它本质就是一个文件描述符，Handler是事件产生的发源地。
2. Synchronout Event Demultiplexer(同步时间分离器)：它本身是一个系统调用，用于等待事件的发生（事件可能是一个，也可能是多个）。调用方在调用它的时候会被阻塞，一直阻塞到同步事件分离器有事件产生为止，对于Linux来说，同步事件分离器指的就是常用的I/)多路复用机制，比如说select ,poll,epoll等，在Java NIO领域中，同步事件分离器对应的组件就是Selector；
3. Event Handler(事件处理器)：本身由多个回调方法构成，这些回调方法构成了与应用相关的对于某个事件的反馈机制。Netty相对于Java NIO来说，在事件处理器这个角色上进行了一个升级，它为我们开发者提供了大量的回调方法，供我们在特定的事件产生时实现相应的回调方法进行业务逻辑的处理。

# 源码分析

## ChannelPipeline