

课程说明

- Netty编解码器
- TCP粘包/拆包的问题及解决
- 自研RPC实战
- Netty核心源码剖析
- Netty优化建议

1、Netty编解码器

1.1、什么是编解码器

在网络中传输数据时，无论以什么的格式发送（int、String、Long等）都会以字节流的方式进行传递，客户端将原来的格式数据转化为字节，称之为编码（encode），服务端将字节形式转化为原来的格式，称之为解码（decode），编解码统称为codec。

编解码器包括编码器与解码器两部分，编码器负责出站数据操作，解码器负责进站数据操作。

1.2、解码器

解码器是负责进站的数据操作，那么解码器也一定实现了ChannelInboundHandler接口，所以编解码器本质上也是ChannelHandler。

Netty中提供了ByteToMessageDecoder的抽象实现，自定义解码器只需要继承该类，实现decode()即可。Netty也提供了一些常用的解码器实现，基本都是开箱即用的。比如：

- RedisDecoder 基于Redis协议的解码器
- XmlDecoder 基于XML格式的解码器
- JsonObjectDecoder 基于json数据格式的解码器
- HttpObjectDecoder 基于http协议的解码器

Netty也提供了MessageToMessageDecoder，将一种格式转化为另一种格式的解码器，也提供了一些实现：

- StringDecoder 将接收到ByteBuf转化为字符串
- ByteArrayDecoder 将接收到ByteBuf转化字节数组
- Base64Decoder 将由ByteBuf或US-ASCII字符串编码的Base64解码为ByteBuf。

1.2.1、案例

将传入的字节流转化为Integer类型。

```
package cn.itcast.netty.codec;

import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.handler.codec.ByteToMessageDecoder;
```

```
import java.util.List;

public class ByteToIntegerDecoder extends ByteToMessageDecoder {

    /**
     *
     * @param ctx 上下文
     * @param in 输入的ByteBuf消息数据
     * @param out 转化后输出的容器
     * @throws Exception
     */
    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object>
out) throws Exception {
        if(in.readableBytes() >= 4){ //int类型占用4个字节，所以需要判断是否存在有4个字
节，再进行读取
            out.add(in.readInt()); //读取到int类型数据，放入到输出，完成数据类型的转化
        }
    }
}
```

在Handler中使用：

```
package cn.itcast.netty.codec;

import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;

public class ServerHandler extends ChannelInboundHandlerAdapter {

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
        Integer i = (Integer) msg; //这里可以直接拿到Integer类型的数据
        System.out.println("服务端接收到的消息为：" + i);
    }
}
```

在pipeline中添加解码器：

```

@Override
protected void initChannel(SocketChannel ch) throws Exception {
    ch.pipeline()
        .addLast(new ByteToIntegerDecoder())
        .addLast(new ServerHandler());
}

```

1.3、编码器

编码器与解码器是相反的操作，将原有的格式转化为字节的过程，在Netty中提供了MessageToByteEncoder的抽象实现，它实现了ChannelOutboundHandler接口，本质上也是ChannelHandler。

一些实现类：

- ObjectEncoder 将对象（需要实现Serializable接口）编码为字节流
- SocksMessageEncoder 将SocksMessage编码为字节流
- HAProxyMessageEncoder 将HAProxyMessage编码成字节流

Netty也提供了MessageToMessageEncoder，将一种格式转化为另一种格式的编码器，也提供了一些实现：

- RedisEncoder 将Redis协议的对象进行编码
- StringEncoder 将字符串进行编码操作
- Base64Encoder 将Base64字符串进行编码操作

1.3.1、案例

将Integer类型编码为字节进行传递。

自定义编码器：

```

package cn.itcast.netty.codec.client;

import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.handler.codec.MessageToByteEncoder;

public class IntegerToByteEncoder extends MessageToByteEncoder<Integer> {

    @Override
    protected void encode(ChannelHandlerContext ctx, Integer msg, ByteBuf out)
        throws Exception {
        out.writeInt(msg);
    }
}

```

在Handler直接输出数字即可：

```

package cn.itcast.netty.codec.client;

import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;
import io.netty.util.CharsetUtil;

public class ClientHandler extends SimpleChannelInboundHandler<ByteBuf> {

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) throws
Exception {
        System.out.println("接收到服务端的消息: " +
msg.toString(CharsetUtil.UTF_8));
    }

    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        ctx.writeAndFlush(123);
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
throws Exception {
        cause.printStackTrace();
        ctx.close();
    }
}

```

在pipeline中添加编码器:

```

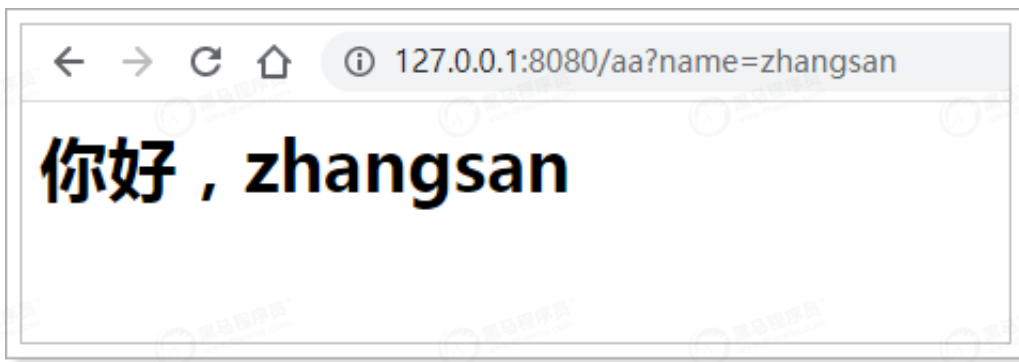
@Override
protected void initChannel(SocketChannel ch) throws Exception {
    ch.pipeline().addLast(new IntegerToByteEncoder());
    ch.pipeline().addLast(new ClientHandler());
}

```

1.4、案例：开发http服务器

在Netty中提供了http的解码器，我们通过该解码器进行http服务器的开发。

实现效果：



Server:

```
package cn.itcast.netty.codec.http;

import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;
import io.netty.handler.codec.http.HttpObjectAggregator;
import io.netty.handler.codec.http.HttpRequestDecoder;
import io.netty.handler.codec.http.HttpResponseEncoder;
import io.netty.handler.stream.ChunkedWriteHandler;

public class NettyHttpServer {

    public static void main(String[] args) throws Exception {

        // 主线程，不处理任何业务逻辑，只是接收客户的连接请求
        EventLoopGroup boss = new NioEventLoopGroup(1);
        // 工作线程，线程数默认是：cpu*2
        EventLoopGroup worker = new NioEventLoopGroup();

        try {
            // 服务器启动类
            ServerBootstrap serverBootstrap = new ServerBootstrap();
            serverBootstrap.group(boss, worker);
            //配置server通道
            serverBootstrap.channel(NioServerSocketChannel.class);
            serverBootstrap.childHandler(new ChannelInitializer<SocketChannel>() {

                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    ch.pipeline()
                        .addLast(new HttpRequestDecoder()) //http请求的解码器
                        //将http请求中的uri以及请求体聚合成一个完整的
                        FullHttpRequest对象
                }
            });
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

器

件传输，防止内存溢出

```
.addLast(new HttpObjectAggregator(1024 * 128))
.addLast(new HttpResponseEncoder()) //http响应的编码

.addLast(new ChunkedWriteHandler()) //支持异步的大文

.addLast(new ServerHandler());

    }
}); //worker线程的处理器

ChannelFuture future = serverBootstrap.bind(8080).sync();
System.out.println("服务器启动完成。。。。");

//等待服务端监听端口关闭
future.channel().closeFuture().sync();
} finally {
    //优雅关闭
    boss.shutdownGracefully();
    worker.shutdownGracefully();
}
}
```

ServerHandler:

```
package cn.itcast.netty.codec.http;

import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelFutureListener;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;
import io.netty.channel.SimpleChannelInboundHandler;
import io.netty.handler.codec.http.*;
import io.netty.util.CharsetUtil;

import java.util.Map;

public class ServerHandler extends SimpleChannelInboundHandler<FullHttpRequest>
{

    @Override
    public void channelRead0(ChannelHandlerContext ctx, FullHttpRequest
request) throws Exception {

        //解析FullHttpRequest，得到请求参数
        Map<String, String> paramMap = new RequestParser(request).parse();
        String name = paramMap.get("name");

        //构造响应对象
```

```

        FullHttpResponse httpResponse = new
DefaultFullHttpResponse(HttpVersion.HTTP_1_1, HttpResponseStatus.OK);
        httpResponse.headers().set(HttpHeaderNames.CONTENT_TYPE,
"text/html;charset=utf-8");

        StringBuilder sb = new StringBuilder();
        sb.append("<h1>");
        sb.append("你好, " + name);
        sb.append("</h1>");

        httpResponse.content().writeBytes(Unpooled.copiedBuffer(sb,
CharsetUtil.UTF_8));

        ctx.writeAndFlush(httpResponse)
                .addListener(ChannelFutureListener.CLOSE); //操作完成后, 将
channel关闭
    }
}

```

RequestParser:

```

package cn.itcast.netty.codec.http;

import io.netty.handler.codec.http.FullHttpRequest;
import io.netty.handler.codec.http.HttpMethod;
import io.netty.handler.codec.http.QueryStringDecoder;
import io.netty.handler.codec.http.multipart.Attribute;
import io.netty.handler.codec.http.multipart.HttpPostRequestDecoder;
import io.netty.handler.codec.http.multipart.InterfaceHttpData;

import java.io.IOException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * HTTP请求参数解析器, 支持GET, POST
 */
public class RequestParser {
    private FullHttpRequest fullReq;

    /**
     * 构造一个解析器
     * @param req
     */
    public RequestParser(FullHttpRequest req) {
        this.fullReq = req;
    }
}

```

```

/**
 * 解析请求参数
 * @return 包含所有请求参数的键值对，如果没有参数，则返回空Map
 *
 * @throws IOException
 */
public Map<String, String> parse() throws Exception {
    HttpMethod method = fullReq.method();

    Map<String, String> parmMap = new HashMap<>();

    if (HttpMethod.GET == method) {
        // 是GET请求
        QueryStringDecoder decoder = new QueryStringDecoder(fullReq.uri());
        decoder.parameters().entrySet().forEach( entry -> {
            // entry.getValue()是一个List，只取第一个元素
            parmMap.put(entry.getKey(), entry.getValue().get(0));
        });
    } else if (HttpMethod.POST == method) {
        // 是POST请求
        HttpPostRequestDecoder decoder = new
        HttpPostRequestDecoder(fullReq);
        decoder.offer(fullReq);

        List<InterfaceHttpData> parmList = decoder.getBodyHttpDatas();

        for (InterfaceHttpData parm : parmList) {

            Attribute data = (Attribute) parm;
            parmMap.put(data.getName(), data.getValue());
        }

    } else {
        // 不支持其它方法
        throw new RuntimeException("不支持其它方法"); // 这是个自定义的异常，可删
掉这一行
    }

    return parmMap;
}
}

```

1.5、对象的编解码

对于JavaBean对象，Netty也支持了Object对象的编解码，其实也就是对象的序列化，要求java对象需要java.io.Serializable接口。

定义javabeen对象：

```
package cn.itcast.netty.codec.obj;

public class User implements java.io.Serializable {

    private static final long serialVersionUID = -89217070354741790L;

    private Long id;
    private String name;
    private Integer age;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "User{" +
            "id=" + id +
            ", name='" + name + '\'' +
            ", age=" + age +
            '}';
    }
}
```

1.5.1、服务端

NettyObjectServer:

```

package cn.itcast.netty.codec.obj;

import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;
import io.netty.handler.codec.serialization.ClassResolvers;
import io.netty.handler.codec.serialization.ObjectDecoder;

public class NettyObjectServer {

    public static void main(String[] args) throws Exception {

        // 主线程，不处理任何业务逻辑，只是接收客户的连接请求
        EventLoopGroup boss = new NioEventLoopGroup(1);
        // 工作线程，线程数默认是：cpu*2
        EventLoopGroup worker = new NioEventLoopGroup();

        try {
            // 服务器启动类
            ServerBootstrap serverBootstrap = new ServerBootstrap();
            serverBootstrap.group(boss, worker);
            //配置server通道
            serverBootstrap.channel(NioServerSocketChannel.class);
            serverBootstrap.childHandler(new ChannelInitializer<SocketChannel>() {

                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    ch.pipeline()
                        .addLast(new
ObjectDecoder(ClassResolvers.weakCachingResolver(this.getClass().getClassLoader
()))))
                        .addLast(new ServerHandler());
                }
            }); //worker线程的处理器

            ChannelFuture future = serverBootstrap.bind(6677).sync();
            System.out.println("服务器启动完成。。。。");

            //等待服务端监听端口关闭
            future.channel().closeFuture().sync();
        } finally {
            //优雅关闭
            boss.shutdownGracefully();
            worker.shutdownGracefully();
        }
    }
}

```

```
}  
}
```

ServerHandler:

```
package cn.itcast.netty.codec.obj;  
  
import io.netty.buffer.Unpooled;  
import io.netty.channel.ChannelHandlerContext;  
import io.netty.channel.SimpleChannelInboundHandler;  
import io.netty.util.CharsetUtil;  
  
public class ServerHandler extends SimpleChannelInboundHandler<User> {  
  
    @Override  
    public void channelRead0(ChannelHandlerContext ctx, User user) throws  
Exception {  
        //获取到user对象  
        System.out.println(user);  
  
        ctx.writeAndFlush(Unpooled.copiedBuffer("ok", CharsetUtil.UTF_8));  
    }  
}
```

1.5.2、客户端

NettyObjectClient:

```
package cn.itcast.netty.codec.obj;  
  
import io.netty.bootstrap.Bootstrap;  
import io.netty.channel.ChannelFuture;  
import io.netty.channel.ChannelInitializer;  
import io.netty.channel.EventLoopGroup;  
import io.netty.channel.nio.NioEventLoopGroup;  
import io.netty.channel.socket.SocketChannel;  
import io.netty.channel.socket.nio.NioSocketChannel;  
import io.netty.handler.codec.serialization.ObjectEncoder;  
  
public class NettyObjectClient {  
  
    public static void main(String[] args) throws Exception{  
        EventLoopGroup worker = new NioEventLoopGroup();  
  
        try {  
            // 服务器启动类
```

```

        Bootstrap bootstrap = new Bootstrap();
        bootstrap.group(worker);
        bootstrap.channel(NioSocketChannel.class);
        bootstrap.handler(new ChannelInitializer<SocketChannel>() {
            @Override
            protected void initChannel(SocketChannel ch) throws Exception {
                ch.pipeline().addLast(new ObjectEncoder());
                ch.pipeline().addLast(new ClientHandler());
            }
        });

        ChannelFuture future = bootstrap.connect("127.0.0.1", 6677).sync();

        future.channel().closeFuture().sync();
    } finally {
        worker.shutdownGracefully();
    }
}
}

```

ClientHandler:

```

package cn.itcast.netty.codec.obj;

import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;
import io.netty.util.CharsetUtil;

public class ClientHandler extends SimpleChannelInboundHandler<ByteBuf> {

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) throws
Exception {
        System.out.println("接收到服务端的消息: " +
msg.toString(CharsetUtil.UTF_8));
    }

    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        User user = new User();
        user.setId(1L);
        user.setName("张三");
        user.setAge(20);
        ctx.writeAndFlush(user);
    }

    @Override

```

```

    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
    throws Exception {
        cause.printStackTrace();
        ctx.close();
    }
}

```

1.6、Hessian编解码

JDK序列化使用是比较方便，但是它的性能较差，序列化后的字节大小也比较大，所以一般在项目中不会使用自带的序列化，而是会采用第三方的序列化框架。

我们以Hessian为例，演示下如何与Netty整合进行编解码处理。

导入Hessian依赖：

```

<dependency>
    <groupId>com.caucho</groupId>
    <artifactId>hessian</artifactId>
    <version>4.0.63</version>
</dependency>

```

User对象：

```

package cn.itcast.netty.codec.hessian;

public class User implements java.io.Serializable{

    private static final long serialVersionUID = -8200798627910162221L;

    private Long id;
    private String name;
    private Integer age;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {

```

```

        this.name = name;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "User{" +
            "id=" + id +
            ", name='" + name + '\'' +
            ", age=" + age +
            '}';
    }
}

```

1.6.1、编解码器

Hessian序列化工具类：

```

package cn.itcast.netty.codec.hessian.codec;

import com.caucho.hessian.io.HessianInput;
import com.caucho.hessian.io.HessianOutput;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;

/**
 * Hessian序列化工具类
 *
 */
public class HessianSerializer {

    public <T> byte[] serialize(T obj) {
        ByteArrayOutputStream os = new ByteArrayOutputStream();
        HessianOutput ho = new HessianOutput(os);
        try {
            ho.writeObject(obj);
            ho.flush();
            return os.toByteArray();
        } catch (IOException e) {

```

```

        throw new RuntimeException(e);
    } finally {
        try {
            ho.close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        try {
            os.close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}

public <T> Object deserialize(byte[] bytes, Class<T> clazz) {
    ByteArrayInputStream is = new ByteArrayInputStream(bytes);
    HessianInput hi = new HessianInput(is);
    try {
        return (T) hi.readObject(clazz);
    } catch (IOException e) {
        throw new RuntimeException(e);
    } finally {
        try {
            hi.close();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        try {
            is.close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
}

```

编码器:

```

package cn.itcast.netty.codec.hessian.codec;

import cn.itcast.netty.coder.hessian.User;
import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.handler.codec.MessageToByteEncoder;

public class HessianEncoder extends MessageToByteEncoder<User> {

    private HessianSerializer hessianSerializer = new HessianSerializer();
}

```

```

        protected void encode(ChannelHandlerContext ctx, User msg, ByteBuf out)
        throws Exception {
            byte[] bytes = hessianSerializer.serialize(msg);
            out.writeBytes(bytes);
        }
    }
}

```

解码器：

```

package cn.itcast.netty.codec.hessian.codec;

import cn.itcast.netty.coder.hessian.User;
import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.handler.codec.ByteToMessageDecoder;

import java.util.List;

public class HessianDecoder extends ByteToMessageDecoder {

    private HessianSerializer hessianSerializer = new HessianSerializer();

    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object>
out) throws Exception {
        //复制一份ByteBuf数据，轻复制，非完全拷贝
        //避免出现异常：did not read anything but decoded a message
        //Netty检测没有读取任何字节就会抛出该异常
        ByteBuf in2 = in.retainDuplicate();

        byte[] dst;
        if (in2.hasArray()) { //堆缓冲区模式
            dst = in2.array();
        } else {
            dst = new byte[in2.readableBytes()];
            in2.getBytes(in2.readerIndex(), dst);
        }

        in.skipBytes(in.readableBytes()); //跳过所有的字节，表示已经读取过了

        Object obj = hessianSerializer.deserialize(dst, User.class); //反序列化
        out.add(obj);
    }
}

```

1.6.2、服务端

```

package cn.itcast.netty.codec.hessian;

import cn.itcast.netty.coder.hessian.codec.HessianDecoder;
import cn.itcast.netty.coder.hessian.handler.ServerHandler;
import io.netty.bootstrap.ServerBootstrap;
import io.netty.buffer.UnpooledByteBufAllocator;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelOption;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;

public class NettyHessianServer {

    public static void main(String[] args) throws Exception {

        //      System.setProperty("io.netty.noUnsafe", "true");

        // 主线程，不处理任何业务逻辑，只是接收客户的连接请求
        EventLoopGroup boss = new NioEventLoopGroup(1);
        // 工作线程，线程数默认是：cpu*2
        EventLoopGroup worker = new NioEventLoopGroup();

        try {
            // 服务器启动类
            ServerBootstrap serverBootstrap = new ServerBootstrap();
            serverBootstrap.group(boss, worker);
            //配置server通道
            serverBootstrap.channel(NioServerSocketChannel.class);
            serverBootstrap.childHandler(new ChannelInitializer<SocketChannel>

() {

                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    ch.pipeline()
                        .addLast(new HessianDecoder())
                        .addLast(new ServerHandler());
                }
            }); //worker线程的处理器

            //      serverBootstrap.childOption(ChannelOption.ALLOCATOR,
UnpooledByteBufAllocator.DEFAULT);

            ChannelFuture future = serverBootstrap.bind(6677).sync();
            System.out.println("服务器启动完成。。。。");

            //等待服务端监听端口关闭
            future.channel().closeFuture().sync();

```

```

    } finally {
        //优雅关闭
        boss.shutdownGracefully();
        worker.shutdownGracefully();
    }
}
}

```

```

package cn.itcast.netty.codec.hessian.handler;

import cn.itcast.netty.coder.hessian.User;
import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;
import io.netty.util.CharsetUtil;

public class ServerHandler extends SimpleChannelInboundHandler<User> {

    @Override
    public void channelRead0(ChannelHandlerContext ctx, User user) throws
Exception {
        //获取到user对象
        System.out.println(user);

        ctx.writeAndFlush(Unpooled.copiedBuffer("ok", CharsetUtil.UTF_8));
    }

}

```

1.6.3、客户端

```

package cn.itcast.netty.codec.hessian;

import cn.itcast.netty.coder.hessian.codec.HessianEncoder;
import cn.itcast.netty.coder.hessian.handler.ClientHandler;
import io.netty.bootstrap.Bootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioSocketChannel;
import io.netty.handler.codec.serialization.ObjectEncoder;

public class NettyHessianClient {

    public static void main(String[] args) throws Exception{

```

```

        EventLoopGroup worker = new NioEventLoopGroup();

        try {
            // 服务器启动类
            Bootstrap bootstrap = new Bootstrap();
            bootstrap.group(worker);
            bootstrap.channel(NioSocketChannel.class);
            bootstrap.handler(new ChannelInitializer<SocketChannel>() {
                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    ch.pipeline().addLast(new HessianEncoder());
                    ch.pipeline().addLast(new ClientHandler());
                }
            });

            ChannelFuture future = bootstrap.connect("127.0.0.1", 6677).sync();

            future.channel().closeFuture().sync();
        } finally {
            worker.shutdownGracefully();
        }
    }
}

```

```

package cn.itcast.netty.codec.hessian.handler;

import cn.itcast.netty.coder.hessian.User;
import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;
import io.netty.util.CharsetUtil;

public class ClientHandler extends SimpleChannelInboundHandler<ByteBuf> {

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) throws
    Exception {
        System.out.println("接收到服务端的消息: " +
        msg.toString(CharsetUtil.UTF_8));
    }

    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        User user = new User();
        user.setId(1L);
        user.setName("张三");
        user.setAge(20);
        ctx.writeAndFlush(user);
    }
}

```

```
@Override
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
throws Exception {
    cause.printStackTrace();
    ctx.close();
}
}
```

2、TCP粘包/拆包的问题及解决

2.1、ReplayingDecoder

在前面案例中，当需要获取int数据时，需要进行判断是否够4个字节，如果解码业务过于复杂的话，这样的判断会显得非常的繁琐，在Netty中提供了ReplayingDecoder就可以解决这样的问题。

ReplayingDecoder也是继承了ByteToMessageDecoder进行的扩展。

javadoc文档中的一个示例：

```
public class IntegerHeaderFrameDecoder extends ByteToMessageDecoder {

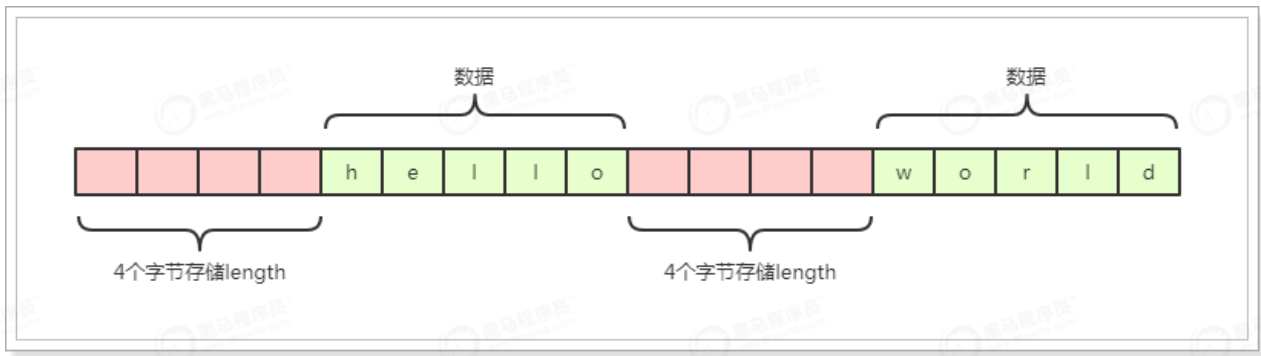
    @Override
    protected void decode(ChannelHandlerContext ctx,
                          ByteBuf buf, List<Object> out) throws Exception {

        if (buf.readableBytes() < 4) {
            return;
        }

        buf.markReaderIndex();
        int length = buf.readInt();

        if (buf.readableBytes() < length) {
            buf.resetReaderIndex();
            return;
        }

        out.add(buf.readBytes(length));
    }
}
```



使用ReplayingDecoder后:

```
public class IntegerHeaderFrameDecoder
    extends ReplayingDecoder<Void> {

    protected void decode(ChannelHandlerContext ctx,
        ByteBuf buf, List<Object> out) throws Exception {

        out.add(buf.readBytes(buf.readInt()));
    }
}
```

基本原理:

- 使用了特殊的ByteBuf, 叫做ReplayingDecoderByteBuf, 扩展了ByteBuf
- 重写了ByteBuf的readXxx()等方法, 会先检查可读字节长度, 一旦检测到不满足要求就直接抛出REPLAY (REPLAY继承ERROR)
- ReplayingDecoder重写了ByteToMessageDecoder的callDecode()方法, 捕获Signal并在catch块中重置ByteBuf的readerIndex。
- 继续等待数据, 直到有了数据后继续读取, 这样就可以保证读取到需要读取的数据。
- 类定义中的泛型 S 是一个用于记录解码状态的状态机枚举类, 在state(S s)、checkpoint(S s)等方法中会用到。在简单解码时也可以用java.lang.Void来占位。

需要注意:

- buffer的部分操作 (readBytes(ByteBuffer dst)、retain()、release()等方法会直接抛出异常)
- 在某些情况下会影响性能 (如多次对同一段消息解码)

TCP是基于流的, 只保证接收到数据包分片顺序, 而不保证接收到的数据包每个分片大小。因此在使用ReplayingDecoder时, 即使不存在多线程, 同一个线程也可能多次调用decode()方法。在decode中修改ReplayingDecoder的类变量时必须小心谨慎。

//这是一个错误的例子:

//消息中包含了2个integer, 代码中decode方法会被调用两次, 此时队列size不等于2, 这段代码达不到期望结果。

```
public class MyDecoder extends ReplayingDecoder<Void> {
    private final Queue<Integer> values = new LinkedList<Integer>();
    @Override
    public void decode(ByteBuf buf, List<Object> out) throws Exception {
        // A message contains 2 integers.
    }
}
```

```

        values.offer(buf.readInt());
        values.offer(buf.readInt());
        assert values.size() == 2;
        out.add(values.poll() + values.poll());
    }
}

//正确的做法:
public class MyDecoder extends ReplayingDecoder<Void> {
    private final Queue<Integer> values = new LinkedList<Integer>();
    @Override
    public void decode(ByteBuf buf, List<Object> out) throws Exception {
        // Revert the state of the variable that might have been changed
        // since the last partial decode.
        values.clear();

        // A message contains 2 integers.
        values.offer(buf.readInt());
        values.offer(buf.readInt());
        // Now we know this assertion will never fail.
        assert values.size() == 2;
        out.add(values.poll() + values.poll());
    }
}

```

ByteToIntegerDecoder2的实现:

```

package cn.itcast.netty.coder;

import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.handler.codec.ByteToMessageDecoder;
import io.netty.handler.codec.ReplayingDecoder;

import java.util.List;

public class ByteToIntegerDecoder2 extends ReplayingDecoder<Void> {

    /**
     * @param ctx 上下文
     * @param in 输入的ByteBuf消息数据
     * @param out 转化后输出的容器
     * @throws Exception
     */
    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object>
out) throws Exception {
        out.add(in.readInt()); //读取到int类型数据, 放入到输出, 完成数据类型的转化
    }
}

```

```
}
```

2.2、什么是TCP粘包/拆包问题？

TCP是流传递的，所谓流，就是一串没有界限的数据，服务端接收到客户端发来的数据，并不确定这是一条数据，还是多条数据，应该如何拆包，服务端是不知道的。

所以，客户端与服务端就需要约定好拆包的规则，客户端按照此规则进行粘包，而服务端按照此规则进行拆包，这就是TCP的粘包与拆包，如果不约定好，就会出现服务端不能按照期望拿到数据。

实际上，彼此约定的规则就是协议，自定义协议就是自定义规则。

2.2.1、案例：演示TCP粘包/拆包问题

客户端：向服务端发送10条消息，并且记录服务端返回的消息的数量。

```
package cn.itcast.netty.tcpackage.client;

import io.netty.bootstrap.Bootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioSocketChannel;

public class NettyClient {

    public static void main(String[] args) throws Exception{
        EventLoopGroup worker = new NioEventLoopGroup();

        try {
            // 服务器启动类
            Bootstrap bootstrap = new Bootstrap();
            bootstrap.group(worker);
            bootstrap.channel(NioSocketChannel.class);
            bootstrap.handler(new ChannelInitializer<SocketChannel>() {
                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    ch.pipeline().addLast(new ClientHandler());
                }
            });

            ChannelFuture future = bootstrap.connect("127.0.0.1", 5566).sync();

            future.channel().closeFuture().sync();
        } finally {
            worker.shutdownGracefully();
        }
    }
}
```

```

    }
}

package cn.itcast.netty.tcppackage.client;

import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;
import io.netty.util.CharsetUtil;

public class ClientHandler extends SimpleChannelInboundHandler<ByteBuf> {

    private int count;

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) throws
Exception {
        System.out.println("接收到服务端的消息: " +
msg.toString(CharsetUtil.UTF_8));
        System.out.println("接收到服务端的消息数量: " + (++count));
    }

    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        for (int i = 0; i < 10; i++) {
            ctx.writeAndFlush(Unpooled.copiedBuffer("from client a message!",
CharsetUtil.UTF_8));
        }
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
throws Exception {
        cause.printStackTrace();
        ctx.close();
    }
}

```

服务端：接收消息并且记录消息的数量，向客户端发送响应。

```

package cn.itcast.netty.tcppackage.server;

import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.EventLoopGroup;

```

```

import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;

public class NettyServer {

    public static void main(String[] args) throws Exception {

        // 主线程，不处理任何业务逻辑，只是接收客户的连接请求
        EventLoopGroup boss = new NioEventLoopGroup(1);
        // 工作线程，线程数默认是：cpu*2
        EventLoopGroup worker = new NioEventLoopGroup();

        try {
            // 服务器启动类
            ServerBootstrap serverBootstrap = new ServerBootstrap();
            serverBootstrap.group(boss, worker);
            //配置server通道
            serverBootstrap.channel(NioServerSocketChannel.class);
            serverBootstrap.childHandler(new ChannelInitializer<SocketChannel>
() {

                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    ch.pipeline()
                        .addLast(new ServerHandler());
                }
            }); //worker线程的处理器

            ChannelFuture future = serverBootstrap.bind(5566).sync();
            System.out.println("服务器启动完成。。。。");

            //等待服务端监听端口关闭
            future.channel().closeFuture().sync();
        } finally {
            //优雅关闭
            boss.shutdownGracefully();
            worker.shutdownGracefully();
        }
    }
}

```

```

package cn.itcast.netty.tcpackage.server;

import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;
import io.netty.channel.SimpleChannelInboundHandler;

```

```
import io.netty.util.CharsetUtil;

public class ServerHandler extends SimpleChannelInboundHandler<ByteBuf> {

    private int count;

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) throws
Exception {
        System.out.println("服务端接收到消息: " +
msg.toString(CharsetUtil.UTF_8));
        System.out.println("服务端接收到消息数量: " + (++count));

        ctx.writeAndFlush(Unpooled.copiedBuffer("ok", CharsetUtil.UTF_8));
    }
}
```

测试结果:

#服务端

服务器启动完成。。。。。

服务端接收到消息: from client a message!from client a message!from client a message!from client a message!from client a message!from client a message!from client a message!from client a message!

服务端接收到消息数量: 1

#客户端

接收到服务端的消息: ok

接收到服务端的消息数量: 1

#测试结果与期望不符

2.3、解决方法

一般来讲有3中方法解决TCP的粘包与拆包问题:

- 在发送的数据包中添加头, 在头里存储数据的大小, 服务端就可以按照此大小来读取数据, 这样就知道界限在哪里了。
- 以固定的长度发送数据, 超出的分多次发送, 不足的以0填充, 接收端就以固定长度接收即可。
- 在数据包之间设置边界, 如添加特殊符号, 这样, 接收端通过这个边界就可以将不同的数据包拆分开。

2.4、实战: 解决TCP的粘包/拆包问题

2.4.1、自定义协议

```
package cn.itcast.netty.tcpackage2;
```

```

public class MyProtocol {

    private Integer length; //数据头：长度

    private byte[] body; //数据体

    public Integer getLength() {
        return length;
    }

    public void setLength(Integer length) {
        this.length = length;
    }

    public byte[] getBody() {
        return body;
    }

    public void setBody(byte[] body) {
        this.body = body;
    }
}

```

2.4.2、编解码器

编码器：

```

package cn.itcast.netty.tcpackage2;

import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.handler.codec.MessageToByteEncoder;

public class MyEncoder extends MessageToByteEncoder<MyProtocol> {

    @Override
    protected void encode(ChannelHandlerContext ctx, MyProtocol msg, ByteBuf
out) throws Exception {
        out.writeInt(msg.getLength());
        out.writeBytes(msg.getBody());
    }
}

```

解码器：

```

package cn.itcast.netty.tcpackage2;

```

```

import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.handler.codec.ReplayingDecoder;

import java.util.List;

public class MyDecoder extends ReplayingDecoder<Void> {

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object>
out) throws Exception {

        int length = in.readInt(); //获取长度
        byte[] data = new byte[length]; //根据长度定义byte数组
        in.readBytes(data); //读取数据

        MyProtocol myProtocol = new MyProtocol();
        myProtocol.setLength(length);
        myProtocol.setBody(data);

        out.add(myProtocol);
    }

}

```

2.4.3、客户端

```

package cn.itcast.netty.tcpackage2.client;

import cn.itcast.netty.tcpackage2.MyProtocol;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;
import io.netty.util.CharsetUtil;

public class ClientHandler extends SimpleChannelInboundHandler<MyProtocol> {

    private int count;

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, MyProtocol msg)
throws Exception {
        System.out.println("接收到服务端的消息: " + new String(msg.getBody(),
CharsetUtil.UTF_8));
        System.out.println("接收到服务端的消息数量: " + (++count));
    }

    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        for (int i = 0; i < 10; i++) {

```

```

        byte[] data = "from client a message!".getBytes(CharsetUtil.UTF_8);
        MyProtocol myProtocol = new MyProtocol();
        myProtocol.setLength(data.length);
        myProtocol.setBody(data);
        ctx.writeAndFlush(myProtocol);
    }
}

@Override
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
throws Exception {
    cause.printStackTrace();
    ctx.close();
}
}

```

```

package cn.itcast.netty.tcpackage2.client;

import cn.itcast.netty.tcpackage2.MyDecoder;
import cn.itcast.netty.tcpackage2.MyEncoder;
import io.netty.bootstrap.Bootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioSocketChannel;

public class NettyClient {

    public static void main(String[] args) throws Exception{
        EventLoopGroup worker = new NioEventLoopGroup();

        try {
            // 服务器启动类
            Bootstrap bootstrap = new Bootstrap();
            bootstrap.group(worker);
            bootstrap.channel(NioSocketChannel.class);
            bootstrap.handler(new ChannelInitializer<SocketChannel>() {
                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    ch.pipeline().addLast(new MyEncoder());
                    ch.pipeline().addLast(new MyDecoder());
                    ch.pipeline().addLast(new ClientHandler());
                }
            });

            ChannelFuture future = bootstrap.connect("127.0.0.1", 5566).sync();

```

```

        future.channel().closeFuture().sync();
    } finally {
        worker.shutdownGracefully();
    }
}
}

```

2.4.4、服务端

```

package cn.itcast.netty.tcpackage2.server;

import cn.itcast.netty.tcpackage2.MyProtocol;
import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;
import io.netty.util.CharsetUtil;

public class ServerHandler extends SimpleChannelInboundHandler<MyProtocol> {

    private int count;

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, MyProtocol msg)
    throws Exception {
        System.out.println("服务端接收到消息: " + new String(msg.getBody(),
CharsetUtil.UTF_8));
        System.out.println("服务端接收到消息数量: " + (++count));

        byte[] data = "ok".getBytes(CharsetUtil.UTF_8);
        MyProtocol myProtocol = new MyProtocol();
        myProtocol.setLength(data.length);
        myProtocol.setBody(data);

        ctx.writeAndFlush(myProtocol);
    }
}

```

```

package cn.itcast.netty.tcpackage2.server;

import cn.itcast.netty.tcpackage2.MyDecoder;
import cn.itcast.netty.tcpackage2.MyEncoder;
import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.EventLoopGroup;

```

```

import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;

public class NettyServer {

    public static void main(String[] args) throws Exception {

        // 主线程，不处理任何业务逻辑，只是接收客户的连接请求
        EventLoopGroup boss = new NioEventLoopGroup(1);
        // 工作线程，线程数默认是：cpu*2
        EventLoopGroup worker = new NioEventLoopGroup();

        try {
            // 服务器启动类
            ServerBootstrap serverBootstrap = new ServerBootstrap();
            serverBootstrap.group(boss, worker);
            //配置server通道
            serverBootstrap.channel(NioServerSocketChannel.class);
            serverBootstrap.childHandler(new ChannelInitializer<SocketChannel>
() {

                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    ch.pipeline()
                        .addLast(new MyDecoder())
                        .addLast(new MyEncoder())
                        .addLast(new ServerHandler());
                }
            }); //worker线程的处理器

            ChannelFuture future = serverBootstrap.bind(5566).sync();
            System.out.println("服务器启动完成。。。。");

            //等待服务端监听端口关闭
            future.channel().closeFuture().sync();
        } finally {
            //优雅关闭
            boss.shutdownGracefully();
            worker.shutdownGracefully();
        }
    }
}

```

2.4.5、测试

#客户端

接收到服务端的消息：ok

接收到服务端的消息数量：1

接收到服务端的消息：ok

接收到服务端的消息数量: 2
接收到服务端的消息: ok
接收到服务端的消息数量: 3
接收到服务端的消息: ok
接收到服务端的消息数量: 4
接收到服务端的消息: ok
接收到服务端的消息数量: 5
接收到服务端的消息: ok
接收到服务端的消息数量: 6
接收到服务端的消息: ok
接收到服务端的消息数量: 7
接收到服务端的消息: ok
接收到服务端的消息数量: 8
接收到服务端的消息: ok
接收到服务端的消息数量: 9
接收到服务端的消息: ok
接收到服务端的消息数量: 10

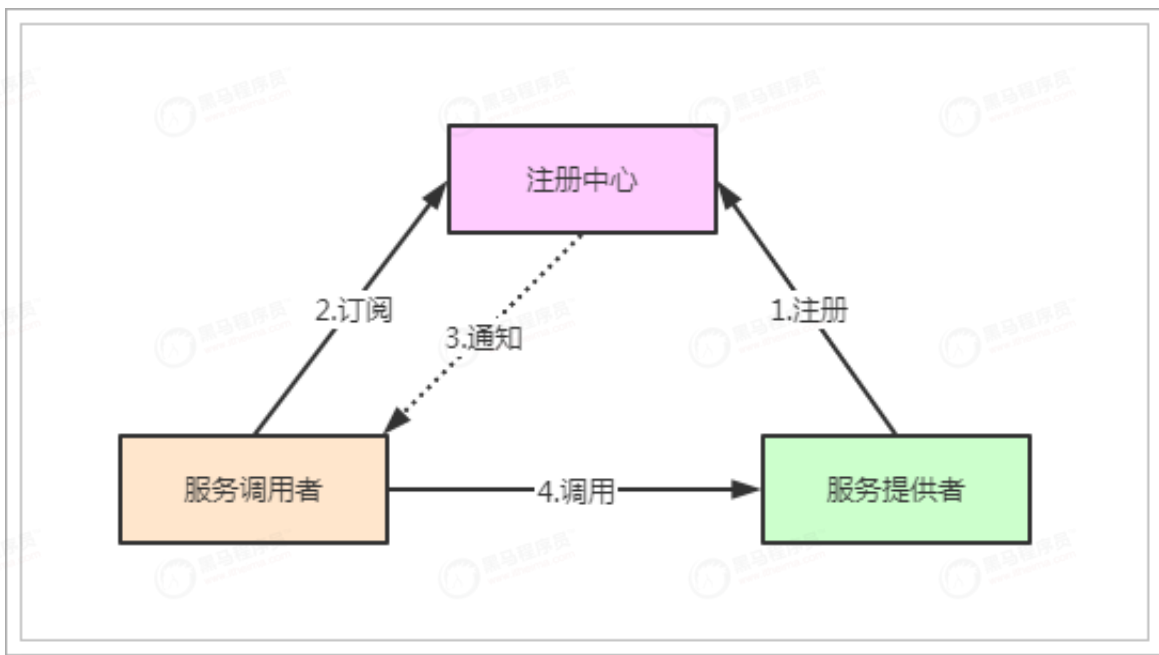
#服务端

服务器启动完成。。。。。
服务端接收到消息: from client a message!
服务端接收到消息数量: 1
服务端接收到消息: from client a message!
服务端接收到消息数量: 2
服务端接收到消息: from client a message!
服务端接收到消息数量: 3
服务端接收到消息: from client a message!
服务端接收到消息数量: 4
服务端接收到消息: from client a message!
服务端接收到消息数量: 5
服务端接收到消息: from client a message!
服务端接收到消息数量: 6
服务端接收到消息: from client a message!
服务端接收到消息数量: 7
服务端接收到消息: from client a message!
服务端接收到消息数量: 8
服务端接收到消息: from client a message!
服务端接收到消息数量: 9
服务端接收到消息: from client a message!
服务端接收到消息数量: 10

#测试结果符合预期

3、自研RPC实战

3.1、自研RPC设计说明



- 服务端

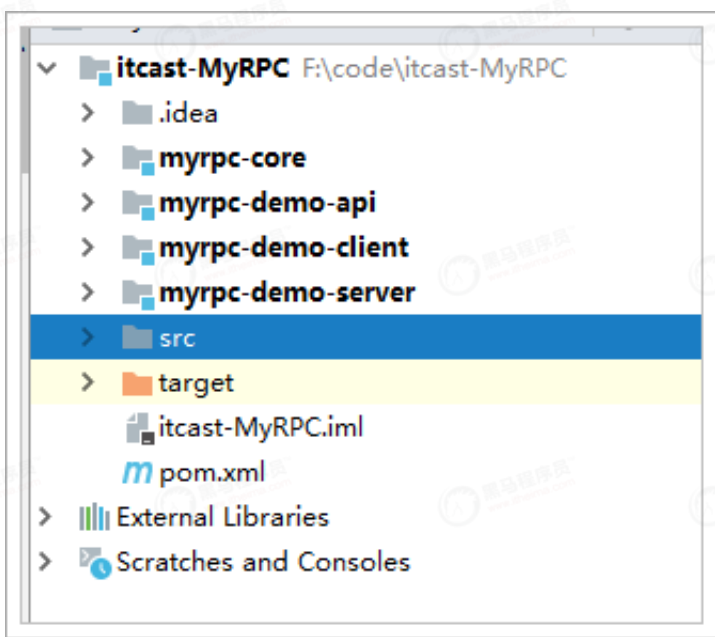
- 接收到客户端发来的消息后，进行解码操作
- 根据消息中的接口信息，通过反射找到其实现类，执行目标方法
- 将返回的数据再进行编码操作，发送给客户端

- 客户端

- 客户端通过代理的方式，获取到接口的代理类
- 通过代理类进行发送消息，实际上是向服务端发送消息
- 通过编码器将消息进行编码操作
- 接收到服务端的响应后，进行解码操作，返回数据给方法的调用方

- 注册中心

- 注册中心并非课程重点，故不做实现



工程说明：

- myrpc-core rpc核心实现

- myrpc-demo-api 示例api定义
- myrpc-demo-client 客户端示例
- myrpc-demo-server 服务端示例

3.2、自定义协议

自定义协议，需要分别定义请求对象和响应对象，用于数据的传输。

需要注意的是，RpcRequest、RpcResponse要序列化传输，需要实现java.io.Serializable接口。

```
package cn.itcast.myrpc.core.base;

public abstract class BaseRpcBean implements java.io.Serializable{

    private String requestId; //请求id

    public String getRequestId() {
        return requestId;
    }

    public void setRequestId(String requestId) {
        this.requestId = requestId;
    }
}
```

```
package cn.itcast.myrpc.core.base;

import java.util.Arrays;

public class RpcRequest extends BaseRpcBean {

    private static final long serialVersionUID = -4403913516658154989L;

    private long createTime; //创建请求时间
    private String className; //类名称，全包路径
    private String methodName; //执行方法名
    private Class<?>[] parameterTypes; //方法中的参数类型
    private Object[] parameters; //执行方法传入的参数

    // getter、setter、toString方法省略，实际使用时自己生成
}
```

```

package cn.itcast.myrpc.core.base;

public class RpcResponse extends BaseRpcBean{

    private static final long serialVersionUID = -7276479460065645174L;

    private String errorMsg; //错误消息
    private Object result; //结果数据

    // getter、setter、toString方法省略，实际使用时自己生成
}

```

3.3、编解码器

编解码器是在服务端以及客户端通用的，所以设计时需要考虑其通用性，不能将泛型对象硬编码到代码中。

编码器：

```

package cn.itcast.myrpc.core.codec;

import cn.itcast.myrpc.core.base.BaseRpcBean;
import cn.itcast.myrpc.core.util.HessianSerializer;
import cn.itcast.myrpc.core.util.MySerializer;
import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.handler.codec.MessageToByteEncoder;

/**
 * 编码器
 */
public class MyEncoder extends MessageToByteEncoder<BaseRpcBean> {

    private static MySerializer hessianSerializer = new HessianSerializer();

    @Override
    protected void encode(ChannelHandlerContext ctx, BaseRpcBean msg, ByteBuf out) throws Exception {
        byte[] bytes = hessianSerializer.serialize(msg);

        out.writeInt(bytes.length);
        out.writeBytes(bytes);
    }
}

```

解码器：

```

package cn.itcast.myrpc.core.codec;

import cn.itcast.myrpc.core.base.BaseRpcBean;
import cn.itcast.myrpc.core.util.HessianSerializer;
import cn.itcast.myrpc.core.util.MySerializer;
import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.handler.codec.ReplayingDecoder;

import java.util.List;

/**
 * 解码器
 */
public class MyDecoder<T> extends BaseRpcBean<> extends ReplayingDecoder<Void> {

    private static MySerializer hessianSerializer = new HessianSerializer();

    private Class<T> clazz;

    public MyDecoder(Class<T> clazz) {
        this.clazz = clazz;
    }

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object>
out) throws Exception {
        byte[] bytes = new byte[in.readInt()];
        in.readBytes(bytes);

        BaseRpcBean rpcBean = hessianSerializer.deserialize(bytes, clazz);
        out.add(rpcBean);
    }
}

```

Hessian序列化:

```

package cn.itcast.myrpc.core.util;

public interface MySerializer {

    /**
     * 将对象序列化成字节数组
     *
     * @param obj
     * @return
     */
    <T> byte[] serialize(T obj);
}

```

```

/**
 * 将字节数组反序列化成对象
 *
 * @param bytes
 * @param clazz
 * @return
 */
<T> T deserialize(byte[] bytes, Class<T> clazz);
}

```

```

package cn.itcast.myrpc.core.util;

import com.caucho.hessian.io.HessianInput;
import com.caucho.hessian.io.HessianOutput;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;

/**
 * Hessian序列化工具类
 */
public class HessianSerializer implements MySerializer {

    public <T> byte[] serialize(T obj) {
        ByteArrayOutputStream os = new ByteArrayOutputStream();
        HessianOutput ho = new HessianOutput(os);
        try {
            ho.writeObject(obj);
            ho.flush();
            return os.toByteArray();
        } catch (IOException e) {
            throw new RuntimeException(e);
        } finally {
            try {
                ho.close();
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
            try {
                os.close();
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }
    }
}

```

```

public <T> T deserialize(byte[] bytes, Class<T> clazz) {
    ByteArrayInputStream is = new ByteArrayInputStream(bytes);
    HessianInput hi = new HessianInput(is);
    try {
        return (T) hi.readObject(clazz);
    } catch (IOException e) {
        throw new RuntimeException(e);
    } finally {
        try {
            hi.close();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        try {
            is.close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
}

```

3.4、服务端

3.4.1、NettyServer

```

package cn.itcast.myrpc.core.server;

import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioServerSocketChannel;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class NettyServer {

    private static final Logger LOGGER =
        LoggerFactory.getLogger(NettyServer.class);

    public static void start(String host, int port) {
        doStart(host, port);
    }

    private static void doStart(String host, int port) {
        // 主线程，不处理任何业务逻辑，只是接收客户的连接请求
        EventLoopGroup boss = new NioEventLoopGroup(1);
    }
}

```

```

// 工作线程, 线程数默认是: cpu*2
EventLoopGroup worker = new NioEventLoopGroup();

try {
    // 服务器启动类
    ServerBootstrap serverBootstrap = new ServerBootstrap();

    serverBootstrap.group(boss, worker) //设置线程组
        .channel(NioServerSocketChannel.class) //配置server通道
        .childHandler(new ServerInitializer()); //worker线程的处理器

    ChannelFuture future = serverBootstrap.bind(host, port).sync();
    LOGGER.info("服务器启动完成, 地址为: " + host + ":" + port);

    //等待服务端监听端口关闭
    future.channel().closeFuture().sync();
} catch (Exception e) {
    LOGGER.error("服务器启动失败!", e);
} finally {
    //优雅关闭
    boss.shutdownGracefully();
    worker.shutdownGracefully();
}

}

}

```

3.4.2、ServerInitializer

```

package cn.itcast.myrpc.core.server;

import cn.itcast.myrpc.core.base.RpcRequest;
import cn.itcast.myrpc.core.codec.MyDecoder;
import cn.itcast.myrpc.core.codec.MyEncoder;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.socket.SocketChannel;

public class ServerInitializer extends ChannelInitializer<SocketChannel> {

    @Override
    protected void initChannel(SocketChannel ch) throws Exception {
        ch.pipeline()
            .addLast(new MyDecoder(RpcRequest.class)) //解码器, 需要解码的对象
            .addLast(new MyEncoder()) //编码器, 用于数据的响应
            .addLast(new ServerHandler()); //自定义逻辑
    }
}

```

```
}
```

3.4.3、ServerHandler

```
package cn.itcast.myrpc.core.server;

import cn.itcast.myrpc.core.base.RpcRequest;
import cn.itcast.myrpc.core.base.RpcResponse;
import cn.itcast.myrpc.core.util.ClassUtil;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

public class ServerHandler extends SimpleChannelInboundHandler<RpcRequest> {

    private static final Logger LOGGER =
        LoggerFactory.getLogger(ServerHandler.class);

    private static final Map<Class<?>, Object> OBJECT_MAP = new HashMap<>();

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, RpcRequest request)
        throws Exception {
        RpcResponse rpcResponse = new RpcResponse();
        rpcResponse.setRequestId(request.getRequestId());

        LOGGER.info("开始处理消息: requestId = " + request.getRequestId());

        try {
            Class<?> clazz = Class.forName(request.getClassName());
            if (!OBJECT_MAP.containsKey(clazz)) {
                //获取接口的实现类, 这里只获取第一个实现类, 忽略其他实现类
                ArrayList<Class> allClassByInterface =
                    ClassUtil.getAllClassByInterface(clazz);
                for (Class c : allClassByInterface) {
                    //将对象缓存起来, 提升效率
                    OBJECT_MAP.put(clazz, c.newInstance());
                    break;
                }
            }

            //通过反射找到方法执行
        }
    }
}
```

```

        Method method = clazz.getMethod(request.getMethodName(),
request.getParameterTypes());
        method.setAccessible(true);
        Object result = method.invoke(OBJECT_MAP.get(clazz),
request.getParameters());

        rpcResponse.setResult(result);
    } catch (Exception e) {
        LOGGER.error("处理失败... requestId = " + request.getRequestId(),
e);

        //出错
        rpcResponse.setErrorMsg("error");
    }

    ctx.writeAndFlush(rpcResponse);
}
}

```

3.4.4、ClassUtil

```

package cn.itcast.myrpc.core.util;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.File;
import java.net.JarURLConnection;
import java.net.URL;
import java.util.ArrayList;
import java.util.Enumeration;
import java.util.List;
import java.util.jar.JarEntry;
import java.util.jar.JarFile;

/**
 * 获取接口的所有实现类 理论上也可以用来获取类的所有子类
 * 查询路径有限制，只局限于接口所在模块下，比如pandora-gateway,而非整个pandora（会递归搜索该文件夹下所有的实现类）
 * 路径中不可含中文，否则会异常。若要支持中文路径，需对该模块代码中url.getPath() 返回值进行urldecode.
 */
public class ClassUtil {
    private static final Logger LOG = LoggerFactory.getLogger(ClassUtil.class);

    public static ArrayList<Class> getAllClassByInterface(Class clazz) {
        ArrayList<Class> list = new ArrayList<>();
        // 判断是否是一个接口
    }

```

```

        if (clazz.isInterface()) {
            try {
                ArrayList<Class> allClass =
getAllClass(clazz.getPackage().getName());
                /**
                 * 循环判断路径下的所有类是否实现了指定的接口 并且排除接口类自己
                 */
                for (int i = 0; i < allClass.size(); i++) {
                    /**
                     * 判断是不是同一个接口
                     */
                    // isAssignableFrom:判定此 Class 对象所表示的类或接口与指定的
Class
                    // 参数所表示的类或接口是否相同, 或是否是其超类或超接口
                    if (clazz.isAssignableFrom(allClass.get(i))) {
                        if (!clazz.equals(allClass.get(i))) {
                            // 自身并不加进去
                            list.add(allClass.get(i));
                        }
                    }
                }
            } catch (Exception e) {
                LOG.error("出现异常{}", e.getMessage());
                throw new RuntimeException("出现异常" + e.getMessage());
            }
        }
        LOG.info("class list size :" + list.size());
        return list;
    }

    /**
     * 从一个指定路径下查找所有的类
     *
     * @param packagename
     */
    private static ArrayList<Class> getAllClass(String packagename) {

        LOG.info("packageName to search: " + packagename);
        List<String> classNameList = getClassNames(packagename);
        ArrayList<Class> list = new ArrayList<>();

        for (String className : classNameList) {
            try {
                list.add(Class.forName(className));
            } catch (ClassNotFoundException e) {
                LOG.error("load class from name failed:" + className +
e.getMessage());
            }
        }
    }
}

```

```

        throw new RuntimeException("load class from name failed:" +
        className + e.getMessage());
    }
}
LOG.info("find list size :" + list.size());
return list;
}

/**
 * 获取某包下所有类
 *
 * @param packageName 包名
 * @return 类的完整名称
 */
public static List<String> getClassName(String packageName) {

    List<String> fileNames = null;
    ClassLoader loader = Thread.currentThread().getContextClassLoader();
    String packagePath = packageName.replace(".", "/");
    URL url = loader.getResource(packagePath);
    if (url != null) {
        String type = url.getProtocol();
        LOG.debug("file type : " + type);
        if (type.equals("file")) {
            String fileSearchPath = url.getPath();
            LOG.debug("fileSearchPath: " + fileSearchPath);
            fileSearchPath = fileSearchPath.substring(0,
fileSearchPath.indexOf("/classes"));
            LOG.debug("fileSearchPath: " + fileSearchPath);
            fileNames = getClassNameByFile(fileSearchPath);
        } else if (type.equals("jar")) {
            try {
                JarURLConnection jarURLConnection = (JarURLConnection)
url.openConnection();
                JarFile jarFile = jarURLConnection.getJarFile();
                fileNames = getClassNameByJar(jarFile, packagePath);
            } catch (java.io.IOException e) {
                throw new RuntimeException("open Package URL failed: " +
e.getMessage());
            }

        } else {
            throw new RuntimeException("file system not support! cannot
load MsgProcessor! ");
        }
    }
    return fileNames;
}

```

```

/**
 * 从项目文件获取某包下所有类
 *
 * @param filePath 文件路径
 * @return 类的完整名称
 */
private static List<String> getClassNameByFile(String filePath) {
    List<String> myClassName = new ArrayList<String>();
    File file = new File(filePath);
    File[] childFiles = file.listFiles();
    for (File childFile : childFiles) {
        if (childFile.isDirectory()) {
            myClassName.addAll(getClassNameByFile(childFile.getPath()));
        } else {
            String childFilePath = childFile.getPath();
            if (childFilePath.endsWith(".class")) {
                childFilePath =
childFilePath.substring(childFilePath.indexOf("\\classes") + 9,
childFilePath.lastIndexOf("."));
                childFilePath = childFilePath.replace("\\", ".");
                myClassName.add(childFilePath);
            }
        }
    }

    return myClassName;
}

/**
 * 从jar获取某包下所有类
 *
 * @return 类的完整名称
 */
private static List<String> getClassNameByJar(JarFile jarFile, String
packagePath) {
    List<String> myClassName = new ArrayList<String>();
    try {
        Enumeration<JarEntry> entrys = jarFile.entries();
        while (entrys.hasMoreElements()) {
            JarEntry jarEntry = entrys.nextElement();
            String entryName = jarEntry.getName();
            //LOG.info("entrys jarfile:"+entryName);
            if (entryName.endsWith(".class")) {
                entryName = entryName.replace("/", ".").substring(0,
entryName.lastIndexOf("."));
                myClassName.add(entryName);
                //LOG.debug("Find Class :"+entryName);
            }
        }
    }
}

```

```

    } catch (Exception e) {
        LOG.error("发生异常:" + e.getMessage());
        throw new RuntimeException("发生异常:" + e.getMessage());
    }
    return myClassName;
}
}

```

3.5、客户端

3.5.1、NettyClient

```

package cn.itcast.myrpc.core.client;

import cn.itcast.myrpc.core.base.RpcRequest;
import io.netty.bootstrap.Bootstrap;
import io.netty.channel.Channel;
import io.netty.channel.ChannelFuture;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioSocketChannel;

public class NettyClient {

    private Channel channel;

    private EventLoopGroup worker;

    public void start(String host, int port) {
        try {
            doStart(host, port);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void doStart(String host, int port) throws Exception {
        //定义工作线程组
        this.worker = new NioEventLoopGroup();
        Bootstrap bootstrap = new Bootstrap();
        bootstrap.group(this.worker)
            .channel(NioSocketChannel.class)
            .handler(new ClientInitializer());

        //连接到远程服务
        ChannelFuture future = bootstrap.connect(host, port).sync();

        //保留channel对象，方便后面通过该通道发送消息
    }
}

```

```

        this.channel = future.channel();
    }

    public void close() {
        if (null != this.channel && this.channel.isActive()) {
            this.channel.close();
        }
        if (null != this.worker && !this.worker.isShutdown()) {
            this.worker.shutdownGracefully();
        }
    }

    /**
     * 发送消息
     *
     * @param request
     */
    public void sendMsg(RpcRequest request) {
        this.channel.writeAndFlush(request);
    }
}

```

3.5.2、ClientInitializer

```

package cn.itcast.myrpc.core.client;

import cn.itcast.myrpc.core.base.RpcResponse;
import cn.itcast.myrpc.core.codec.MyDecoder;
import cn.itcast.myrpc.core.codec.MyEncoder;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.socket.SocketChannel;

public class ClientInitializer extends ChannelInitializer<SocketChannel> {

    @Override
    protected void initChannel(SocketChannel ch) throws Exception {
        ch.pipeline()
            .addLast(new MyDecoder(RpcResponse.class))
            .addLast(new MyEncoder())
            .addLast(new ClientHandler());
    }
}

```

3.5.3、ClientHandler

```

package cn.itcast.myrpc.core.client;

import cn.itcast.myrpc.core.base.RpcResponse;

```

```

import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class ClientHandler extends SimpleChannelInboundHandler<RpcResponse> {

    private static final Logger LOGGER =
        LoggerFactory.getLogger(ClientHandler.class);

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, RpcResponse msg)
        throws Exception {

        LOGGER.info("接收到服务端的消息: requestId = " + msg.getRequestId());

        //接收到服务端的消息后, 通知响应已经完成
        //通过RpcFutureResponse中的Map获取到RpcFutureResponse对象, 进行方法回调
        RpcFutureResponse rpcFutureResponse =
            RpcFutureResponse.getRpcFutureResponse(msg.getRequestId());
        if(null != rpcFutureResponse){
            rpcFutureResponse.setResponse(msg);
        }else{
            LOGGER.warn("没有找到对应的RpcFutureResponse对象~ requestId = " +
                msg.getRequestId());
        }
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
        throws Exception {
        LOGGER.error("客户端出错~ ", cause);
        ctx.close();
    }
}

```

3.5.4、RpcFutureResponse

```

package cn.itcast.myrpc.core.client;

import cn.itcast.myrpc.core.base.RpcRequest;
import cn.itcast.myrpc.core.base.RpcResponse;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

```

```
/**
 * 用于异步的响应处理
 */
public class RpcFutureResponse implements Future<RpcResponse> {

    //存储requestId与RpcFutureResponse对象的映射关系
    private static final Map<String, RpcFutureResponse> MAP = new
    ConcurrentHashMap<>();

    //是否处理完成, 默认为false
    private boolean done = false;

    //定义一个对象用于锁操作
    private final Object lock = new Object();

    //响应对象
    private RpcResponse response;

    /**
     * 在构造函数中将当前对象存储到MAP中
     *
     * @param rpcRequest
     */
    public RpcFutureResponse(RpcRequest rpcRequest) {
        MAP.put(rpcRequest.getRequestId(), this);
    }

    /**
     * 根据请求id查询RpcFutureResponse对象
     *
     * @param requestId
     * @return
     */
    public static RpcFutureResponse getRpcFutureResponse(String requestId) {
        return MAP.get(requestId);
    }

    @Override
    public boolean cancel(boolean mayInterruptIfRunning) {
        //TODO 暂不实现
        return false;
    }

    @Override
    public boolean isCancelled() {
        //TODO 暂不实现
        return false;
    }
}
```

```

/**
 * 是否完成
 *
 * @return
 */
@Override
public boolean isDone() {
    return done;
}

/**
 * 获取数据, 没有超时时间
 *
 * @return
 * @throws InterruptedException
 * @throws ExecutionException
 */
@Override
public RpcResponse get() throws InterruptedException, ExecutionException {
    return this.get(-1, TimeUnit.MILLISECONDS);
}

/**
 * 获取数据, 指定了超时时间
 *
 * @param timeout 单位为毫秒
 * @param unit
 * @return
 * @throws InterruptedException
 * @throws ExecutionException
 */
@Override
public RpcResponse get(long timeout, TimeUnit unit) throws
InterruptedException, ExecutionException {
    if (isDone()) {
        return this.response;
    }

    synchronized (lock) { //确保只有一个线程在一个时刻执行操作
        try {
            if (timeout < 0) { //如果没有设置超时时间的话, 就一直等待
                lock.wait();
            } else {
                // 如果设置了超时时间的话, 就在给定时间内进行等待
                long timeoutMillis = (TimeUnit.MILLISECONDS == unit) ?
timeout : TimeUnit.MILLISECONDS.convert(timeout, unit);
                lock.wait(timeoutMillis);
            }
        }
    }
}

```

```

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    if (isDone()) {
        return this.response;
    }

    throw new RuntimeException("获取数据，等待超时。。。");
}

public void setResponse(RpcResponse response){
    this.response = response;
    synchronized (lock) { //确保只有一个线程在一个时刻执行操作
        this.done = true;
        this.lock.notifyAll(); //唤醒等待的线程
    }

    //删除已缓存的对象
    MAP.remove(response.getRequestId());
}
}

```

3.5.5、ClientInvocationHandler

```

package cn.itcast.myrpc.core.client;

import cn.itcast.myrpc.core.base.RpcRequest;
import cn.itcast.myrpc.core.base.RpcResponse;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.util.UUID;
import java.util.concurrent.TimeUnit;

/**
 * 动态代理的业务逻辑实现
 */
public class ClientInvocationHandler implements InvocationHandler {

    private NettyClient nettyClient;

    /**
     * 需要将NettyClient传入，因为要基于此发送消息数据
     *
     * @param nettyClient
     */
    public ClientInvocationHandler(NettyClient nettyClient) {

```

```

        this.nettyClient = nettyClient;
    }

    /**
     * 执行的业务逻辑
     *
     * @param proxy
     * @param method
     * @param args
     * @return
     * @throws Throwable
     */
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
    Throwable {
        //获取类、方法、参数等信息
        String className = method.getDeclaringClass().getName();
        String methodName = method.getName();
        Class<?>[] parameterTypes = method.getParameterTypes();
        Object[] parameters = args;

        //封装请求对象
        RpcRequest request = new RpcRequest();
        request.setRequestId(UUID.randomUUID().toString());
        request.setCreateTime(System.currentTimeMillis());
        request.setClassName(className);
        request.setMethodName(methodName);
        request.setParameterTypes(parameterTypes);
        request.setParameters(parameters);

        //定义异步的响应对象
        RpcFutureResponse futureResponse = new RpcFutureResponse(request);

        //向服务端发送消息
        nettyClient.sendMsg(request);

        //获取异步响应的消息
        RpcResponse rpcResponse = futureResponse.get(5, TimeUnit.SECONDS);
        if (rpcResponse.getErrMsg() != null) {
            throw new RuntimeException(rpcResponse.getErrMsg());
        }
        return rpcResponse.getResult();
    }
}

```

3.5.6、BeanFactory

```

package cn.itcast.myrpc.core.client;

import java.lang.reflect.Proxy;

/**
 * 通过动态代理生成代理对象
 */
public class BeanFactory {

    private NettyClient nettyClient;

    public BeanFactory(NettyClient nettyClient){
        this.nettyClient = nettyClient;
    }

    public <T> T getBean(Class<T> clazz) {
        return (T) Proxy.newProxyInstance(
            Thread.currentThread().getContextClassLoader(),
            new Class<?>[] {clazz},
            new ClientInvocationHandler(nettyClient)
        );
    }

}

```

3.6、示例

3.6.1、myrpc-demo-api

pom.xml:

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>itcast-MyRPC</artifactId>
        <groupId>cn.itcast.myrpc</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>myrpc-demo-api</artifactId>

</project>

```

定义pojo:

```
package cn.itcast.myrpc.order.pojo;

import java.util.Arrays;

public class Item implements java.io.Serializable{

    private static final long serialVersionUID = -1768340664273809021L;

    private Long itemId;
    private String title;
    private String[] pics;
    private Long price;

    public Long getItemId() {
        return itemId;
    }

    public void setItemId(Long itemId) {
        this.itemId = itemId;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String[] getPics() {
        return pics;
    }

    public void setPics(String[] pics) {
        this.pics = pics;
    }

    public Long getPrice() {
        return price;
    }

    public void setPrice(Long price) {
        this.price = price;
    }

    @Override
    public String toString() {
```

```
        return "Item{" +
            "itemId=" + itemId +
            ", title='" + title + '\'' +
            ", pics=" + Arrays.toString(pics) +
            ", price=" + price +
            '}';
    }
}
```

```
package cn.itcast.myrpc.order.pojo;

public class Order implements java.io.Serializable{

    private static final long serialVersionUID = 8069637444287105148L;

    private String orderId;
    private Long userId;
    private Long date;
    private Integer itemCount;
    private Long totalPrice;

    public String getOrderId() {
        return orderId;
    }

    public void setOrderId(String orderId) {
        this.orderId = orderId;
    }

    public Long getUserId() {
        return userId;
    }

    public void setUserId(Long userId) {
        this.userId = userId;
    }

    public Long getDate() {
        return date;
    }

    public void setDate(Long date) {
        this.date = date;
    }

    public Integer getItemCount() {
        return itemCount;
    }
}
```

```

    public void setItemCount(Integer itemCount) {
        this.itemCount = itemCount;
    }

    public Long getTotalPrice() {
        return totalPrice;
    }

    public void setTotalPrice(Long totalPrice) {
        this.totalPrice = totalPrice;
    }

    @Override
    public String toString() {
        return "Order{" +
            "orderId='" + orderId + '\'' +
            ", userId=" + userId +
            ", date=" + date +
            ", itemCount=" + itemCount +
            ", totalPrice=" + totalPrice +
            '}';
    }
}

```

服务接口定义：

```

package cn.itcast.myrpc.order.service;

import cn.itcast.myrpc.order.pojo.Item;
import cn.itcast.myrpc.order.pojo.Order;

import java.util.List;

public interface OrderService {

    Order submitOrder(Long userId, List<Item> itemList);

}

```

3.6.2、myrpc-demo-server

pom.xml:

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"

```

```

        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>itcast-MyRPC</artifactId>
        <groupId>cn.itcast.myrpc</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>myrpc-demo-server</artifactId>

    <dependencies>
        <dependency>
            <groupId>cn.itcast.myrpc</groupId>
            <artifactId>myrpc-core</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
        <dependency>
            <groupId>cn.itcast.myrpc</groupId>
            <artifactId>myrpc-demo-api</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
    </dependencies>

</project>

```

服务实现:

```

package cn.itcast.myrpc.order.service;

import cn.itcast.myrpc.order.pojo.Item;
import cn.itcast.myrpc.order.pojo.Order;

import java.util.List;
import java.util.UUID;

public class OrderServiceImpl implements OrderService {

    @Override
    public Order submitOrder(Long userId, List<Item> itemList) {

        Order order = new Order();
        order.setOrderId(UUID.randomUUID().toString());
        order.setDate(System.currentTimeMillis());
        order.setItemCount(itemList.size());
        order.setUserId(userId);
    }
}

```

```

        Long count = 0L;
        for (Item item : itemList) {
            count += item.getPrice();
        }

        order.setTotalPrice(count);

        return order;
    }
}

```

启动服务:

```

package cn.itcast.myrpc.order;

import cn.itcast.myrpc.core.server.NettyServer;

public class MyServer {

    public static void main(String[] args) {
        NettyServer.start("127.0.0.1", 5566);
    }

}

```

3.6.3、myrpc-demo-client

pom.xml:

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>itcast-MyRPC</artifactId>
        <groupId>cn.itcast.myrpc</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>myrpc-demo-client</artifactId>

    <dependencies>
        <dependency>
            <groupId>cn.itcast.myrpc</groupId>

```

```

        <artifactId>myrpc-core</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
    <dependency>
        <groupId>cn.itcast.myrpc</groupId>
        <artifactId>myrpc-demo-api</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
</dependencies>

</project>

```

ClientServer:

```

package cn.itcast.myrpc.order;

import cn.itcast.myrpc.core.client.BeanFactory;
import cn.itcast.myrpc.core.client.NettyClient;
import cn.itcast.myrpc.order.pojo.Item;
import cn.itcast.myrpc.order.pojo.Order;
import cn.itcast.myrpc.order.service.OrderService;

import java.util.ArrayList;
import java.util.List;

public class ClientServer {

    public static void main(String[] args) {
        NettyClient nettyClient = new NettyClient();

        nettyClient.start("127.0.0.1", 5566);

        BeanFactory beanFactory = new BeanFactory(nettyClient);

        OrderService orderService = beanFactory.getBean(OrderService.class);

        List<Item> itemList = new ArrayList<>();
        Item item = new Item();
        item.setItemId(2001L);
        item.setPrice(100L);
        item.setTitle("铅笔");
        itemList.add(item);

        item = new Item();
        item.setItemId(2002L);
        item.setPrice(50L);
        item.setTitle("橡皮");
        itemList.add(item);
    }
}

```

```

        for (int i = 0; i < 10; i++) {
            Order order = orderService.submitOrder(1001L, itemList);
            System.out.println("返回数据: " + order);
        }

        nettyClient.close();
    }
}

```

3.7、测试

服务端：

```

[nioEventLoopGroup-3-1] [cn.itcast.myrpc.core.server.ServerHandler]-[INFO] 开始
处理消息: requestId = 51209bb0-a86e-4363-b966-aa989a3c3908
[nioEventLoopGroup-3-1] [cn.itcast.myrpc.core.server.ServerHandler]-[INFO] 开始
处理消息: requestId = 0400ab1a-b0d7-4f37-8ad5-4ea6401a03f6
[nioEventLoopGroup-3-1] [cn.itcast.myrpc.core.server.ServerHandler]-[INFO] 开始
处理消息: requestId = 776f501f-7963-4917-983e-4f8e38e70903
[nioEventLoopGroup-3-1] [cn.itcast.myrpc.core.server.ServerHandler]-[INFO] 开始
处理消息: requestId = caac11ae-f695-4135-8d9f-231425d36699
[nioEventLoopGroup-3-1] [cn.itcast.myrpc.core.server.ServerHandler]-[INFO] 开始
处理消息: requestId = 29841e2f-310f-47e3-9d74-e3b9e153e378
[nioEventLoopGroup-3-1] [cn.itcast.myrpc.core.server.ServerHandler]-[INFO] 开始
处理消息: requestId = f9f13486-2841-45f5-8d6c-a017d14a4101
[nioEventLoopGroup-3-1] [cn.itcast.myrpc.core.server.ServerHandler]-[INFO] 开始
处理消息: requestId = c994ecac-c427-41a0-87b3-d0e66d67551e
[nioEventLoopGroup-3-1] [cn.itcast.myrpc.core.server.ServerHandler]-[INFO] 开始
处理消息: requestId = 81708019-d7d7-4679-946f-3da89b944651
[nioEventLoopGroup-3-1] [cn.itcast.myrpc.core.server.ServerHandler]-[INFO] 开始
处理消息: requestId = b513d5d7-50b8-434a-937b-97d8bba8c65e

```

客户端：

```

[nioEventLoopGroup-2-1] [io.netty.util.ResourceLeakDetectorFactory]-[DEBUG]
Loaded default ResourceLeakDetector: io.netty.util.ResourceLeakDetector@2f4b36
[nioEventLoopGroup-2-1] [cn.itcast.myrpc.core.client.ClientHandler]-[INFO] 接收
到服务端的消息: requestId = e72bf59a-7149-4d35-84e4-e463fe14858e
返回数据: Order{orderId='34245455-2034-4da3-94d3-926a46d018ed', userId=1001,
date=1593161672169, itemCount=2, totalPrice=150}
[nioEventLoopGroup-2-1] [cn.itcast.myrpc.core.client.ClientHandler]-[INFO] 接收
到服务端的消息: requestId = 51209bb0-a86e-4363-b966-aa989a3c3908
返回数据: Order{orderId='b895fd40-7dc1-4176-bfc1-e71867b4162e', userId=1001,
date=1593161672188, itemCount=2, totalPrice=150}

```

```
[nioEventLoopGroup-2-1] [cn.itcast.myrpc.core.client.ClientHandler]-[INFO] 接收到服务端的消息: requestId = 0400ab1a-b0d7-4f37-8ad5-4ea6401a03f6
返回数据: Order{orderId='dbe3f5be-7a6d-4e1d-af73-6600f76c14a5', userId=1001, date=1593161672191, itemCount=2, totalPrice=150}
[nioEventLoopGroup-2-1] [cn.itcast.myrpc.core.client.ClientHandler]-[INFO] 接收到服务端的消息: requestId = 776f501f-7963-4917-983e-4f8e38e70903
返回数据: Order{orderId='d00d5490-7f03-470b-9d61-e35b66883d8b', userId=1001, date=1593161672194, itemCount=2, totalPrice=150}
[nioEventLoopGroup-2-1] [cn.itcast.myrpc.core.client.ClientHandler]-[INFO] 接收到服务端的消息: requestId = caac11ae-f695-4135-8d9f-231425d36699
返回数据: Order{orderId='6fb2146b-97af-47d2-a782-4b6cc8a3b108', userId=1001, date=1593161672196, itemCount=2, totalPrice=150}
[nioEventLoopGroup-2-1] [cn.itcast.myrpc.core.client.ClientHandler]-[INFO] 接收到服务端的消息: requestId = 29841e2f-310f-47e3-9d74-e3b9e153e378
返回数据: Order{orderId='ae2f0b0c-445f-4500-9c61-086faa3b866f', userId=1001, date=1593161672196, itemCount=2, totalPrice=150}
[nioEventLoopGroup-2-1] [cn.itcast.myrpc.core.client.ClientHandler]-[INFO] 接收到服务端的消息: requestId = f9f13486-2841-45f5-8d6c-a017d14a4101
返回数据: Order{orderId='ec0da70a-b557-4dbf-ba59-4745b98a37d4', userId=1001, date=1593161672196, itemCount=2, totalPrice=150}
[nioEventLoopGroup-2-1] [cn.itcast.myrpc.core.client.ClientHandler]-[INFO] 接收到服务端的消息: requestId = c994ecac-c427-41a0-87b3-d0e66d67551e
返回数据: Order{orderId='bce9265f-e155-41b7-b789-d17da44ca4cd', userId=1001, date=1593161672196, itemCount=2, totalPrice=150}
[nioEventLoopGroup-2-1] [cn.itcast.myrpc.core.client.ClientHandler]-[INFO] 接收到服务端的消息: requestId = 81708019-d7d7-4679-946f-3da89b944651
返回数据: Order{orderId='58f34bb9-279f-46a6-9a8d-7c1b569b4230', userId=1001, date=1593161672196, itemCount=2, totalPrice=150}
[nioEventLoopGroup-2-1] [cn.itcast.myrpc.core.client.ClientHandler]-[INFO] 接收到服务端的消息: requestId = b513d5d7-50b8-434a-937b-97d8bba8c65e
返回数据: Order{orderId='6e5cbe4f-9060-43ee-b0c8-25d2ea345771', userId=1001, date=1593161672196, itemCount=2, totalPrice=150}
[nioEventLoopGroup-2-1] [io.netty.buffer.PoolThreadCache]-[DEBUG] Freed 6 thread-local buffer(s) from thread: nioEventLoopGroup-2-1
```

4、Netty核心源码剖析

4.1、服务端启动过程剖析

4.1.1、创建服务端Channel

主要流程:

- ServerBootstrap对象的bind()方法, 也是入口方法
- AbstractBootstrap中的initAndRegister()进行创建Channel
 - 创建Channel的工作由ReflectiveChannelFactory反射类中的newChannel()方法完成。
- NioServerSocketChannel中的构造方法中, 通过jdk nio底层的SelectorProvider打开ServerSocketChannel。

- 在AbstractNioChannel的构造方法中，设置channel为非阻塞：ch.configureBlocking(false);
- 通过的AbstractChannel的构造方法，创建了id、unsafe、pipeline内容。
- 通过NioServerSocketChannelConfig获取tcp底层的一些参数

4.1.2、初始化服务端Channel

主流程：

- AbstractBootstrap中的initAndRegister()进行初始化channel，代码：init(channel);
- 在ServerBootstrap中的init()方法设置channelOptions以及Attributes。
- 紧接着，将用户自定义参数、属性保存到局部变量currentChildOptions、currentChildAttrs，以供后面使用
- 如果设置了serverBootstrap.handler()的话，会加入到pipeline中。
- 添加连接器ServerBootstrapAcceptor，有新连接加入后，将自定义的childHandler加入到连接的pipeline中：

```
ch.eventLoop().execute(new Runnable() {
    @Override
    public void run() {
        pipeline.addLast(new ServerBootstrapAcceptor(
            ch, currentChildGroup,
            currentChildHandler, currentChildOptions, currentChildAttrs));
    }
});
```

```
@Override
@SuppressWarnings("unchecked")
public void channelRead(ChannelHandlerContext ctx, Object msg) { //当客户端
    有连接时才会执行
    final Channel child = (Channel) msg;

    child.pipeline().addLast(childHandler); //将自定义的childHandler加入到连接
    的pipeline中

    setChannelOptions(child, childOptions, logger);
    setAttributes(child, childAttrs);

    try {
        childGroup.register(child).addListener(new ChannelFutureListener()
        {
            @Override
            public void operationComplete(ChannelFuture future) throws
            Exception {
                if (!future.isSuccess()) {
                    forceClose(child, future.cause());
                }
            }
        });
    }
```

```

    });
} catch (Throwable t) {
    forceClose(child, t);
}
}

```

4.1.3、注册selector

主要流程：

- initAndRegister()方法中的ChannelFuture regFuture = config().group().register(channel); 进行注册
- 在io.netty.channel.AbstractChannel.AbstractUnsafe#register()中完成实际的注册
 - AbstractChannel.this.eventLoop = eventLoop; 进行eventLoop的赋值操作，后续的IO事件工作将在由该eventLoop执行。
 - 调用register0(promise)中的doRegister()进行实际的注册
 - io.netty.channel.nio.AbstractNioChannel#doRegister进行了方法实现

```

//通过jdk底层进行注册多路复用器
//javaChannel() --前面创建的channel
//eventLoop().unwrappedSelector() -- 获取selector
//注册感兴趣的事件为0，表明没有感兴趣的事件，后面会进行重新注册事件
//将this对象以attachment的形式注册到selector，方便后面拿到当前对象的内容
selectionKey =
javaChannel().register(eventLoop().unwrappedSelector(), 0, this);

```

4.1.4、绑定端口

主要流程：

- 入口在io.netty.bootstrap.AbstractBootstrap#doBind0()，启动一个线程进行执行绑定端口操作
- 调用io.netty.channel.AbstractChannelHandlerContext#bind(java.net.SocketAddress, io.netty.channel.ChannelPromise)方法，再次启动线程执行
- 最终调用io.netty.channel.socket.nio.NioServerSocketChannel#doBind()方法进行绑定操作

```

//通过jdk底层的channel进行绑定
@SuppressJava6Requirement(reason = "Usage guarded by java version check")
@Override
protected void doBind(SocketAddress localAddress) throws Exception {
    if (PlatformDependent.javaVersion() >= 7) {
        javaChannel().bind(localAddress, config.getBacklog());
    } else {
        javaChannel().socket().bind(localAddress, config.getBacklog());
    }
}

```

- 什么时候进行更新selector的主从事件?

- 最终在io.netty.channel.nio.AbstractNioChannel#doBeginRead()方法中完成的

```
protected void doBeginRead() throws Exception {
    // Channel.read() or ChannelHandlerContext.read() was called
    final SelectionKey selectionKey = this.selectionKey;
    if (!selectionKey.isValid()) {
        return;
    }

    readPending = true;

    final int interestOps = selectionKey.interestOps();
    if ((interestOps & readInterestOp) == 0) {
        selectionKey.interestOps(interestOps | readInterestOp); //设置
        感兴趣的事件为OP_ACCEPT
    }
}

//在NioServerSocketChannel的构造方法中进行了赋值
public NioServerSocketChannel(ServerSocketChannel channel) {
    super(null, channel, SelectionKey.OP_ACCEPT);
    config = new NioServerSocketChannelConfig(this,
        javaChannel().socket());
}
```

4.2、连接请求过程源码剖析

4.2.1、新连接的接入

主要流程:

- 入口在
io.netty.channel.nio.NioEventLoop#processSelectedKey(java.nio.channels.SelectionKey,
io.netty.channel.nio.AbstractNioChannel)中
 - 进入NioMessageUnsafe的read()方法
 - 调用io.netty.channel.socket.nio.NioServerSocketChannel#doReadMessages() 方法, 创建jdk底层的channel, 封装成NioSocketChannel添加到List容器中

```
@Override
protected int doReadMessages(List<Object> buf) throws Exception {
    SocketChannel ch = SocketUtils.accept(javaChannel());

    try {
        if (ch != null) {
            buf.add(new NioSocketChannel(this, ch));
            return 1;
        }
    }
```

```

    } catch (Throwable t) {
        logger.warn("Failed to create a new channel from an
accepted socket.", t);

        try {
            ch.close();
        } catch (Throwable t2) {
            logger.warn("Failed to close a socket.", t2);
        }
    }

    return 0;
}

```

- 创建NioSocketChannel对象
 - new NioSocketChannel(this, ch), 通过new的方式进行创建
 - 调用super的构造方法
 - 传入SelectionKey.OP_READ事件标识
 - 创建id、unsafe、pipeline对象
 - 设置非阻塞 ch.configureBlocking(false);
 - 创建NioSocketChannelConfig对象

4.2.2、注册读事件

- 在io.netty.channel.nio.AbstractNioMessageChannel.NioMessageUnsafe中的:

```

for (int i = 0; i < size; i++) {
    readPending = false;
    pipeline.fireChannelRead(readBuf.get(i)); //传播读事件
}

```

- 在io.netty.channel.AbstractChannelHandlerContext#invokeChannelRead(java.lang.Object)方法中:

```
private void invokeChannelRead(Object msg) {
    if (invokeHandler()) {
        try {
            //执行channelRead, 需要注意的是, 第一次执行是HeadHandler, 第二次是
            ServerBootstrapAcceptor
            //通过ServerBootstrapAcceptor进入和 新连接接入的 注册selector相同的
            逻辑进行注册以及事件绑定
            ((ChannelInboundHandler) handler()).channelRead(this, msg);
        } catch (Throwable t) {
            invokeExceptionCaught(t);
        }
    } else {
        fireChannelRead(msg);
    }
}
```

5、Netty优化建议

5.1、零拷贝

Netty的零拷贝主要体现在三个方面:

- Bytebuf 使用的是用池化的Direct Buffer类型使用的堆外内存, 不需要进行字节缓冲区的二次拷贝, 如果使用堆内存, JVM会先拷贝到堆内, 再写入Socket, 就多了一次拷贝。
- CompositeByteBuf将多个ByteBuf封装成一个ByteBuf, 在添加ByteBuf时不需要进程拷贝。
- Netty的文件传输类DefaultFileRegion的transferTo方法将文件发送到目标channel中, 不需要进行循环拷贝, 提升了性能。

5.2、使用EventLoop的任务调度

在EventLoop的支持线程外使用channel:

```
channel.eventLoop().execute(new Runnable() {
    @Override
    public void run() {
        channel.writeAndFlush(data)
    }
});
```

而不是直接使用channel.writeAndFlush(data);

前者会直接放入channel所对应的EventLoop的执行队列, 而后者会导致线程的切换。

在writeAndFlush的底层, 如果没有通过eventLoop执行的话, 就会重新启动新的线程执行。

5.3、减少ChannelPipeline的调用长度

```

public class YourHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelActive(ChannelHandlerContext ctx) {
        // BAD (most of the times)
        ctx.channel().writeAndFlush(msg);
        // GOOD
        ctx.writeAndFlush(msg);
    }
}

```

前者是将msg从整个ChannelPipeline中走一遍，所有的handler都要经过，而后者是从当前handler一直到pipeline的尾部，调用更短。

5.4、减少ChannelHandler的创建

如果channelhandler是无状态的（即不需要保存任何状态参数），那么使用Sharable注解，并在bootstrap时只创建一个实例，减少GC。否则每次连接都会new出handler对象。

```

@ChannelHandler.Sharable
public class StatelessHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelActive(ChannelHandlerContext ctx) {}
}

public class MyInitializer extends ChannelInitializer<Channel> {
    private static final ChannelHandler INSTANCE = new StatelessHandler();
    @Override
    public void initChannel(Channel ch) {
        ch.pipeline().addLast(INSTANCE);
    }
}

```

同时需要注意ByteToMessageDecoder之类的编解码器是有状态的，不能使用Sharable注解。

5.5、一些配置参数的设置

ServerBootstrap启动时，通常bossGroup只需要设置为1即可，因为ServerSocketChannel在初始化阶段，只会注册到某一个eventLoop上，而这个eventLoop只会有一个线程在运行，所以没有必要设置为多线程。而IO线程，为了充分利用CPU，同时考虑减少线上下文切换的开销，通常设置为CPU核数的两倍，这也是Netty提供的默认值。

在对于响应时间有高要求的场景，使用.childOption(ChannelOption.TCP_NODELAY, true)和.option(ChannelOption.TCP_NODELAY, true)来禁用nagle算法，不等待，立即发送。