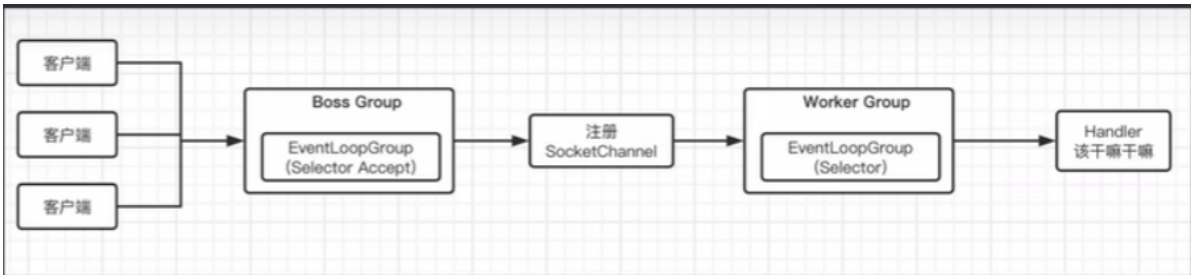
netty基本执行过程



- worker的group只负责socketChannel上的任务，boss负责socketServer上面的注册

基本用法

```
public class Server {
    public static void main(String[] args) {
        // 前面的boss线程group，后面的是worker线程group
        new ServerBootstrap().group(new NioEventLoopGroup(), new
NioEventLoopGroup())
                .channel(NioServerSocketChannel.class)
                .childHandler(new ChannelInitializer<NioSocketChannel>() {
                    @Override
                    protected void initChannel(NioSocketChannel ch) throws
Exception {
                        ch.pipeline()
                            // ctx.firexxxx(msg);数据传给下一个handler，否则就会原路返
回，经过之前的
                            //handler
                                .addLast(new ChannelInboundHandlerAdapter(){
                            @Override
                            public void channelRead(ChannelHandlerContext ctx,
Object msg) throws Exception{
                                ByteBuf byteBuf = (ByteBuf) msg;

 System.out.println(byteBuf.toString(StandardCharsets.UTF_8));
                                System.out.println("channelRead");
//                                super.channelRead(ctx, msg);
                            }

                            @Override
                            public void channelRegistered(ChannelHandlerContext
ctx) throws Exception {
                                System.out.println("channelRegistered");
                                // firexxx会触发下一个handler里面的对应事件
                                ctx.fireChannelRead("AAAAAAAAAA");
                                // super.channelRegistered(ctx);
                            }

                            @Override
                            public void channelInactive(ChannelHandlerContext
ctx) throws Exception {
                                System.out.println("channelInactive");
```

```
                                super.channelInactive(ctx);
                            }

                            @Override
                            public void exceptionCaught(ChannelHandlerContext
ctx, Throwable cause) throws                                    Exception {
                                System.out.println("exceptionCaught");
                                super.exceptionCaught(ctx, cause);
                            }

                            @Override
                            public void
channelUnregistered(ChannelHandlerContext ctx) throws Exception {
                                System.out.println("channelUnregistered");
                                super.channelUnregistered(ctx);
                            }

                            @Override
                            public void
channelReadComplete(ChannelHandlerContext ctx) throws Exception {
                                System.out.println("channelReadComplete");
                                super.channelReadComplete(ctx);
                            }
                        }).addLast(new ChannelInboundHandlerAdapter(){
                                    @Override
                                    public void
channelRegistered(ChannelHandlerContext ctx) throws
Exception {
                                        System.out.println("链路2注册");
                                        super.channelRegistered(ctx);
                                    }

                                    @Override
                                    public void
channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
                                        System.out.println("链路2" +
msg.toString());
                                        super.channelRead(ctx, msg);
                                    }
                        });
                    }
                }).bind(8080);
    }
}
```

EventLoop

- 用于处理各种事件的**线程池**
- 具有异步的特性，在服务器任务分配时应该尽量使用不同线程池去处理大任务

Promise和Future

handler

handler方法和childHandler方法的区别

通过handler添加的handlers是对bossGroup线程组起作用;
通过childHandler添加的handlers是对workerGroup线程组起作用;

netty能发送返回ByteBuf 的,但是却无法返回字符串和byte数组
这是为什么?
因为netty默认是讲byte数组解码编码成ByteBuf
如果想传输不同类型的数据,需要在netty启动类里添加编码器.

handler和pipeline

handler的意义

处理具体的业务逻辑的地方，一般使用方法如下

server端

```
        ChannelFuture channelFuture = new ServerBootstrap()
                .group(new NioEventLoopGroup(), new NioEventLoopGroup())
                .channel(NioServerSocketChannel.class)
                .childHandler(new ChannelInitializer<SocketChannel>() {
                    @Override
                    protected void initChannel(SocketChannel ch) throws
Exception {
                        ch.pipeline().addLast(new
TestInboundHandlerOrderAdapter(1));
                        ch.pipeline().addLast(new
TestInboundHandlerOrderAdapter(2));
                        ch.pipeline().addLast(new
TestInboundHandlerOrderAdapter(3));
                        ch.pipeline().addLast(new ChannelInboundHandlerAdapter()
{
                            @Override
                            public void channelRead(ChannelHandlerContext ctx,
Object msg) throws Exception
                            {
                                ByteBuf byteBuf = ctx.alloc().buffer();
                                byteBuf.writeBytes("HTTP/1.1 200
ok\r\n".getBytes(StandardCharsets.UTF_8));

 byteBuf.writeBytes("\r\n".getBytes(StandardCharsets.UTF_8));
                                byteBuf.writeBytes("hello
netty".getBytes(StandardCharsets.UTF_8));
                                ctx.channel().writeAndFlush(byteBuf);
                            }
                        });
                        ch.pipeline().addLast(new
TestInboundHandlerOrderAdapter(4));
                        ch.pipeline().addLast(new
TestOutboundHandlerOrderAdapter(1));
```

```
                        ch.pipeline().addLast(new
TestOutboundHandlerOrderAdapter(2));
                        ch.pipeline().addLast(new
TestOutboundHandlerOrderAdapter(3));
                    }
                }).bind(8080);
```

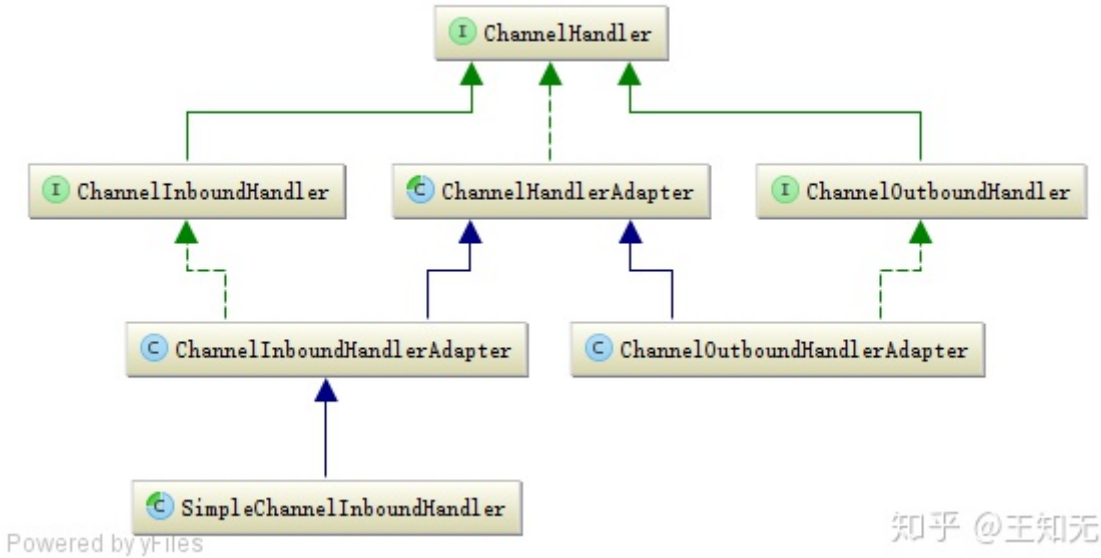
client端

```
ChannelFuture channelFuture = new Bootstrap()
        .group(new NioEventLoopGroup())
        .channel(NioSocketChannel.class)
        .handler(new ChannelInitializer<NioSocketChannel>() {
            @Override
            protected void initChannel(NioSocketChannel ch) throws Exception {
                ch.pipeline().addLast(new StringEncoder());
                ch.pipeline().addLast(new ChannelOutboundHandlerAdapter(){
                    @Override
                    public void write(ChannelHandlerContext ctx, Object msg,
ChannelPromise promise) throws                    Exception {
                        System.out.println("发送消息 " + ((ByteBuf)
msg).toString(StandardCharsets.UTF_8));
                        super.write(ctx, msg, promise);
                    }
                });
                ch.pipeline().addLast(new ChannelInboundHandlerAdapter() {
                    @Override
                    public void channelRead(ChannelHandlerContext ctx, Object
msg) throws Exception {
                        System.out.println("收到消息 " + ((ByteBuf)
msg).toString(StandardCharsets.UTF_8));
                        super.channelRead(ctx, msg);
                    }
                });
            }
        }).connect("localhost", 8080);
```

在ChannelInitializer中的initChannel中使用pipeline添加handler，initChannel中的方法只会执行一次

在最外层调用多次handler或者childHandler会覆盖之前的handler或childHandler，所以正常应该只调用一次

ChannelHandler的继承图



ChannelInboundHandler接口处理入站，ChannelOutboundHandler接口处理出站，SimpleChannelHandler可以泛型接受解码器处理好的数据

handler执行顺序

基本的头尾handler，和普遍执行顺序

基本执行顺序取决于ch.pipline().addLast()的添加顺序

netty会自动在创建服务器时添加头尾handler，后面加入的handler只会在头尾handler之间

入站的请求是顺序执行，出站的请求倒序执行

入站与出站的handler执行

用**i(n)**表示入站handler，用**o(n)**表示出站handler

在下面的pipeline链中

header - i(0) - i(1) - i(2) - i(3) - o(0) - o(1) - i(4) - o(2) - o(3) - o(4) - tail

- 普通请求进入之后会沿着 header -> i(0) -> i(1) -> i(2) -> i(3) -> i(4) 然后结束一次消息发送
- 需要进行write的请求，如果在i(3)或之前，调用**ctx.writeXXXX**中那么将不会经过任何出站处理器，而如果这i(4)进行**ctx.writeXXX**那么，会经过o(1),o(0)
- 入站请求中**ctx.channel().writeAndFlush** 将从 Pipeline 的尾部开始往前找 OutboundHandler 。 **ctx.writeAndFlush** 会从当前 handler 往前找 OutboundHandler。

fireXXX中断或者继续执行handler

handler中如果调用fireXXX，则下一个handler就会被触发对应的XXX方法

```
ch.pipeline().addLast(new SimpleChannelInboundHandler<String>() {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, String msg) throws
Exception {
        if (msg.startsWith("heart")) {
            ctx.channel().writeAndFlush("ok");
        }
        System.out.println(msg);
        ctx.fireChannelRead(msg);
```

```
    }
    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        // ctx.fireChannelActive();
    }
});
ch.pipeline().addLast(new SimpleChannelInboundHandler<String>() {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, String msg) throws
Exception {
        ctx.writeAndFlush("服务器传数据" + msg);
    }
    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        System.out.println("触发第二个active");
        super.channelActive(ctx);
    }
});
```

比如上面的代码，下一个handler里面的channelActive不会被触发，channelRead会触发

注意SimpleInboundHandler的channelRead0方法也需要上一级调用fileChannelRead才会接着触发

SimpleInboundHandler会根据泛型决定接受消息，本质是在channelRead里面调用了用户实现的channelRead0，所以SimpleInboundHandler不要实现channelRead要去实现channelRead0

几种常用父类handler

```
ChannelDuplexHandler 入站出站都处理
SimpleChannelInboundHandler 带泛型的inbond
ChannelInboundHandlerAdapter 入站handler
ChannelOutboundHandler 接口处理出站
```

handler也可以指定工作线程池

```
ChannelPipeline addAfter(EventExecutorGroup group, String baseName, String name,
ChannelHandler handler);
ChannelPipeline addLast(EventExecutorGroup group, ChannelHandler... handlers);
...
```

handler的主要事件

Inbound事件

```
channelRead // 读数据
channelRegistered // channel注册到selector上
channelUnregistered // 上面相反
channelActive // channel开始活跃可用
channelInactive // 与上面相反
userEventTriggered // 前面的handler进行了ctx.fireUserEventTriggered(evtObj)就会被这
个事件收到obj
```

Outbound事件(不清楚)

```
bind // 绑定成功
connect // 有连接过来
disconnect // 连接断开
read //
write // 可以对出站的消息进行处理再发送(主要用)
flush // 当ctx...XXXflush()会触发
```

handler事件

```
handlerAdded // handler添加到了pipeline
handlerRemoved // handler被移除pipeline
exceptionCaught // 有错误出现
```

handler业务执行的核心

入站的核心在channelRead或者channelRead0，出站在write，对业务处理完成之后，通过ctx.fireRead或者ctx.write将处理好的信息向后/前传递，注意内存释放

如果对传入的消息不完全读完或不读会怎样

比如前面的消息是**"message"**，读一个字节，那么剩下的字节就是**"essage"**，若此时不调用release也不传下去，新来了一个消息**"new_message"**，那么在这个handler里面接收的其实是**"essagenew_message"**

@Sharable

标识同一个ChannelHandler的实例可以被多次添加到多个ChannelPipelines中，而且不会出现竞争条件。
如果一个ChannelHandler没有标志@Shareable，在添加到到一个pipeline中时，你需要每次都创建一个新的handler实例，因为它的成员变量是不可分享的。

这个注解仅作为文档参考使用，比如说JCIP注解。
没有实际功能，但是编码器/解码器handler不能加

热插拔handler

动态插拔作用范围只在一个channel上

理解ChannelHandlerContext

```
ChannelHandlerContext是handler的上下文，在ChannelHandlerContext上可以获得在pipeline上有关这个handler的所有信息，基于这些信息就可以在不同handler定位到同一个handler，并对在pipeline上的handler进行修改删除添加
```

获得指定handler的上下文对象

```
context = ctx.pipeline().context(xxxHandler.class);
context = ctx.pipeline().context("handlerName");
....
```

在handler中动态删除其他handler

```
ctx.pipeline().remove(this);
ctx.pipeline().remove(xxxx.class);
ctx.pipeline().remove("name");
```

动态插入

```
String webSocketDecoderContextName =
ctx.pipeline().context(WebSocketFrameDecoder.class).name();

 ctx.pipeline().addAfter(webSocketDecoderContextName,"webSocketAggregator", new
WebSocketFrameAggregator(65536));
```

实例

- 添加WebSocket的实例

```
public class WebSocketTest extends SimpleChannelInboundHandler<Object> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, Object msg) throws
Exception {
        if (msg instanceof FullHttpRequest request) {
            handleHttpUpdateToWebSocket(ctx, request);
            removeSelfAndProcessor(ctx);
        }
    }

    private void removeSelfAndProcessor(ChannelHandlerContext ctx) {
        ctx.pipeline().remove(this);
        ctx.pipeline().addLast(new WebSocketFrameTest());
    }


    private void handleHttpUpdateToWebSocket(ChannelHandlerContext ctx,
HttpRequest request) {
        WebSocketServerHandshakerFactory factory = new
WebSocketServerHandshakerFactory(request.uri(), null, false, 65536, false);
        WebSocketServerHandshaker webSocketServerHandshaker =
factory.newHandshaker(request);
        webSocketServerHandshaker.handshake(ctx.channel(), request);
        String webSocketDecoderContextName =
ctx.pipeline().context(WebSocketFrameDecoder.class).name();

 ctx.pipeline().addAfter(webSocketDecoderContextName,"webSocketAggregator", new
WebSocketFrameAggregator(65536));
```

```
    }
}
```

ByteBuf

申请ByteBuf

```
// 默认池化，池化会复用buf，优化性能
ByteBuf buf = ByteBufAllocator.DEFAULT.buffer();
// 会主动扩容
// byteBuf有自动扩容功能，默认长度256
        StringBuilder stringBuilder = new StringBuilder();
        for (int i = 0;i < 300;i++) {
            stringBuilder.append(i);
        }

 buf.writeBytes(stringBuilder.toString().getBytes(StandardCharsets.UTF_8));
```

BytBuf扩容机制

- 如何写入后数据大小未超过 512，则选择下一个 16 的整数倍，例如写入后大小为 12 ，则扩容后 capacity 是 16
- 如果写入后数据大小超过 512，则选择下一个 2^n，例如写入后大小为 513，则扩容后 capacity 是 2^10=1024（2^9=512 已经不够了）
- 扩容不能超过 max capacity 会报错

池化的byteBuf

默认池化，池化会复用buf，优化性能

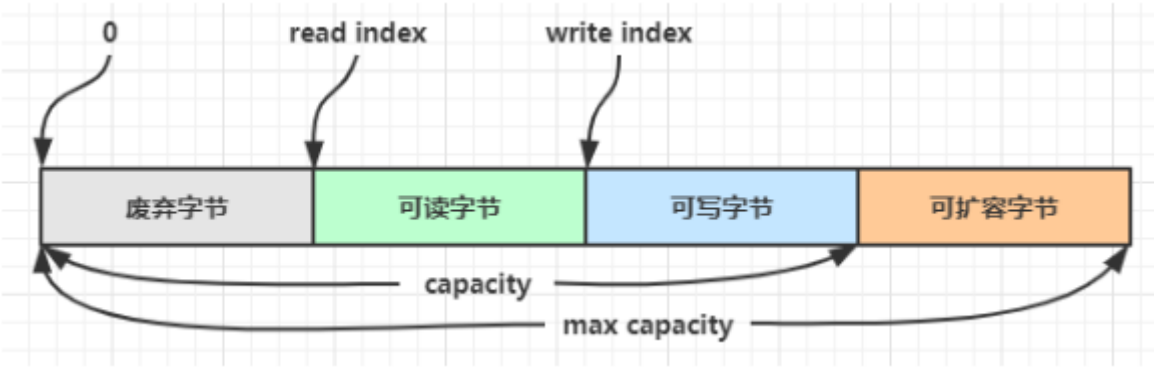
池化的byteBuf内存泄漏问题

因为程序中生成了池化的ByteBuf（PooledByteBuf)。PooledByteBuf的内存空间是从内存池中分配的，并且内部维持了一个计数器，用来记录引用次数。PooledByteBuf在使用完之后要手动调用release()函数，该函数会减小引用次数，减小到0时就会将内存归还在内存池中。如果不调用release()函数，JVM不知道引用计数的存在，释放该对象时，可能还有其他引用在使用该内存空间，该内存空间也无法归还到内存池中，从而导致内存泄漏。

使用ByteBuf应该主动释放不需要的内存

```
msg.release();
// 或者
ReferenceCountUtil.release(msg)
```

ByteBuf基本结构



- 读写指针分离

ByteBuf的基本操作

- 读的api `bf.readXXX()`
- 写的api `bf.writeXXX()`
- 标记当前*读*位置 `bf.markReadIndex()`
- 标记当前*写*位置 `bf.markWriteIndex()`
- 回到现在位置之前的mark位置 `bf.resetReadIndex()` `bf.resetWriteIndex()` 没有mark默认是第一个字节
- 获得读指针位置 `bf.readerIndex()`
- 获得写指针位置 `bf.writerIndex()`
- 重新设置读写指针 `bf.readerIndex(index)` `bf.writerIndex(index)`
- 是否可读 `bf.readable()` (writerIndex > readerIndex)
- 可读字节 `bf.readableBytes()` (writerIndex - readerIndex)
- 跳过几个字节 `bf.skipBytes(length)`

ByteBuf零拷贝(类似于引用)

- 零拷贝都有一个slice或者Retain，retain与内存管理有关

```
ByteBuf slice = origin.slice();
origin.readSlice(...)
```

零拷贝不会产生新的bf，地址仍然在原来的bf上面

slice如果原来的bf被release，那么slice出来的bf也会被release，如果使用readRetainXXX那么就会把原来bf的ReferenceCounted的计数器+1

切出来的对象是XXXSlicedByteBuf类型

ByteBuf复制拷贝

```
buffer.copy()
// 或者
buffer.copy(..)
```

ByteBuf与netty内存管理

- byteBuf继承ReferenceCounted接口，这个接口是实现内存管理的关键

ReferenceCounted接口

实现ReferenceCounted的对象，能够显示的被垃圾回收。当初始化的时候，计数为1。retain（）方法能够增加计数，release() 方法能够减少计数，如果计数被减少到0则对象会被显示回收，再次访问被回收的这些对象将会抛出异常。如果一个对象实现了ReferenceCounted，并且包含有其他对象也实现来ReferenceCounted，当这个对象计数为0被回收的时候，所包含的对象同样会通过release()释放掉。

int refCnt() 返回对象的引用计数.如果返回0,意味着对象已经被回收.

ReferenceCounted retain() 将引用计数增加1 ReferenceCounted retain(int increment) 将引用计数增加指定数量

boolean release() 将引用计数减一, 如果引用计数达到0则回收这个对象. 注意: 返回的boolean值, 当且仅当引用计数变成0并且这个对象被回收才返回true.

boolean release(int decrement) 同上,将引用计数减少指定数量

CompositeByteBuf--组合式byteBuf

ByteBufHolder--用于定义在传输过程中包含ButeBuf的消息包

```
/**
 * A packet which is send or receive.
 */
public interface ByteBufHolder extends ReferenceCounted {

    /**
     * Return the data which is held by this {@link ByteBufHolder}.
     * 一般content是ByteBuf，下面的方法都可以组合ByteBuf的方法使用
     */
    ByteBuf content();

    /**
     * Creates a deep copy of this {@link ByteBufHolder}.
     *
     */
    ByteBufHolder copy();

    /**
     * Duplicates this {@link ByteBufHolder}. Be aware that this will not
automatically call {@link #retain()}.
     */
    ByteBufHolder duplicate();

    /**
     * Duplicates this {@link ByteBufHolder}. This method returns a retained
duplicate unlike {@link #duplicate()}.
     *
     * @see ByteBuf#retainedDuplicate()
     */
    ByteBufHolder retainedDuplicate();

    /**
```

```
     * Returns a new {@link ByteBufHolder} which contains the specified {@code
content}.
     */
    ByteBufHolder replace(ByteBuf content);

    @Override
    ByteBufHolder retain();

    @Override
    ByteBufHolder retain(int increment);

    @Override
    ByteBufHolder touch();

    @Override
    ByteBufHolder touch(Object hint);
}
```

它的默认实现类，一般不使用默认实现

```
/**
 * Default implementation of a {@link ByteBufHolder} that holds it's data in a
{@link ByteBuf}.
 *
 */
public class DefaultByteBufHolder implements ByteBufHolder
```

参考HttpContent的实现

```
private final ByteBuf content;
@Override
public ByteBuf content() {
    return content;
}

@Override
public HttpContent copy() {
    return replace(content.copy());
}

@Override
public HttpContent duplicate() {
    return replace(content.duplicate());
}

@Override
public HttpContent retainedDuplicate() {
    return replace(content.retainedDuplicate());
}

@Override
public HttpContent replace(ByteBuf content) {
    return new DefaultHttpContent(content);
}
```

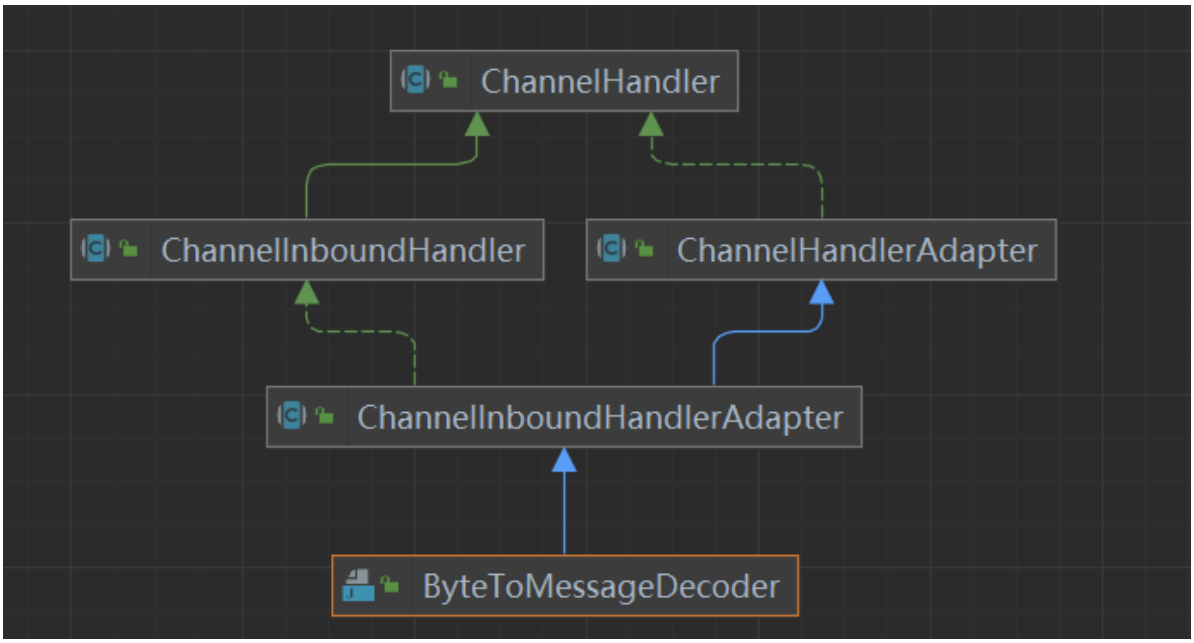
```
@Override
public int refCnt() {
    return content.refCnt();
}
```

自定义协议

核心

解码器 ByteToMessageDecoder

将byteBuf转为T类型的数据



实现方法decoder即可，最后加到out数组里面传递给下面的handler

```
protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out)
```

序列化后的对象是怎么传递给后面的handler的

遍历list把每个obj使用调用一次fireChannelRead

```
Get numElements out of the List and forward these through the pipeline.
static void fireChannelRead(ChannelHandlerContext ctx, List<Object> msgs, int numElements) {
    if (msgs instanceof CodecOutputList) {
        fireChannelRead(ctx, (CodecOutputList) msgs, numElements);
    } else {
        for (int i = 0; i < numElements; i++) {
            ctx.fireChannelRead(msgs.get(i));
        }
    }
}
```

编码器 MessageToByteEncoder<T>

```
protected void encode(ChannelHandlerContext ctx, Person msg, ByteBuf out)
```

实现上面的方法，将序列化数据写入out中，只要执行关键的编码成bytes不需要write

Encoder的自动内存释放机制(重要)

在**Encoder**中调用encode之后netty都会把msg尝试释放掉，如果要继续使用注意retain

```
public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise)
throws Exception {
    ByteBuf buf = null;
    try {
        if (acceptOutboundMessage(msg)) {
            @SuppressWarnings("unchecked")
            I cast = (I) msg;
            buf = allocateBuffer(ctx, cast, preferDirect);
            try {
                encode(ctx, cast, buf);
            } finally {
                // 释放一次
                ReferenceCountUtil.release(cast);
            }

            if (buf.isReadable()) {
                ctx.write(buf, promise);
            } else {
                buf.release();
                ctx.write(Unpooled.EMPTY_BUFFER, promise);
            }
            buf = null;
        } else {
            ctx.write(msg, promise);
        }
    } catch (EncoderException e) {
        throw e;
    } catch (Throwable e) {
        throw new EncoderException(e);
    } finally {
        if (buf != null) {
            buf.release();
        }
    }
}
```

MessageToMessageDecoder

ReplayingDecoder简化

- 比如需要读取一个int数据要四个字节，还没有传递来4个字节，这时候如果ByteToMessage中实现，就需要如下尝试

```
 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));
}
```

在replyingDecoder只需要

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

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

ReplyingDecoder的错误重试机制

```
try {
                    decodeRemovalReentryProtection(ctx, replayable, out);
                    if (ctx.isRemoved()) {
                        break;
                    }
                    if (outSize == out.size()) {
                        if (oldInputLength == in.readableBytes() && oldState ==
state) {
                            throw new DecoderException(
                                    StringUtil.simpleClassName(getClass()) +
".decode() must consume the inbound " +
                                    "data or change its state if it did not
decode anything.");
                        } else {
                            continue;
                        }
```

```
                    }
                } catch (Signal replay) {
                    replay.expect(REPLAY);
                    if (ctx.isRemoved()) {
                        break;
                    }

                    // Return to the checkpoint (or oldPosition) and retry.
                    int checkpoint = this.checkpoint;
                    if (checkpoint >= 0) {
                        in.readerIndex(checkpoint);
                    } else {
                    }
                    break;
                }
```

- 可能造成CPU空转，性能其实也不太好

具体例子

自定义序列化Person类传输

Person类

```
@Data
public class Person {
    String name;
    Integer age;
}
```

PersonDecoder类

```
public class PersonDecoder extends ByteToMessageDecoder {

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object>
out) throws Exception {
        ByteBuf slice = in.readSlice(6);
        String prefix = slice.toString(StandardCharsets.UTF_8);
        // 拿到前缀
        System.out.println("prefix : " + prefix);
        // 拿到name大小
        int size = in.readInt();
        System.out.println("name大小 " + size);
        // 拿到name
        ByteBuf name = in.readSlice(size);
        System.out.println("name = " + name.toString(StandardCharsets.UTF_8));
        // 拿到age
        Integer age = in.readInt();
        Person person = new Person();
        person.setAge(age);
        person.setName(name.toString(StandardCharsets.UTF_8));
        // 传输给下面的handler
        out.add(person);
        System.out.println(person);
```

```
    }
}
```

PersonEncoder类

```
public class PersonEncoder extends MessageToByteEncoder<Person> {
    @Override
    protected void encode(ChannelHandlerContext ctx, Person msg, ByteBuf out)
throws Exception {
        byte[] prefix = "person".getBytes(StandardCharsets.UTF_8);
        byte[] name = msg.getName().getBytes(StandardCharsets.UTF_8);
        Integer age = msg.getAge();
        // 写入前缀标识值
        out.writeBytes(prefix);
        // 标识name大小的值的大小，默认4字节
        out.writeInt(name.length);
        // 写入name
        out.writeBytes(name);
        // 写入四个字节的年龄值
        out.writeInt(age);
        System.out.println("编码成功");

    }
}
```

server端使用Decoder

```
ChannelFuture channelFuture = new ServerBootstrap()
        .group(new NioEventLoopGroup(), new NioEventLoopGroup())
        .channel(NioServerSocketChannel.class)
        .childHandler(new ChannelInitializer<NioSocketChannel>() {
            @Override
            protected void initChannel(NioSocketChannel ch) throws Exception {
                ch.pipeline().addLast(new PersonDecoder())
                        .addLast(new SimpleChannelInboundHandler<Person>() {
                            @Override
                            protected void channelRead0(ChannelHandlerContext
ctx, Person msg) throws Exception {
                                System.out.println("解析成功 " + msg);
                                ctx.fireChannelRead(msg);
                            }
                        })
                        .addLast(new ChannelInboundHandlerAdapter() {
                            @Override
                            public void channelRead(ChannelHandlerContext ctx,
Object msg) throws Exception {
                                System.out.println(msg);
                                super.channelRead(ctx, msg);
                            }
                        });
            }
        }).bind(8080);
channelFuture.addListener(future -> {
   System.out.println("服务端启动");
```

```
});
```

client端使用Encoder

```
ChannelFuture channelFuture = new Bootstrap()
        .group(new NioEventLoopGroup())
        .channel(NioSocketChannel.class)
        .handler(new ChannelInitializer<NioSocketChannel>() {
            @Override
            protected void initChannel(NioSocketChannel ch) throws Exception {
               ch.pipeline().addLast(new PersonEncoder());
            }
        }).connect("localhost", 8080);
channelFuture.addListener(future -> {
    System.out.println("客户端启动成功");
});
Person person = new Person();
person.setName("吴涛");
person.setAge(21);
channelFuture.channel().writeAndFlush(person);
```

服务端解析成功

```
服务端启动
prefix : person
name大小 6
name = 吴涛
Person(name=吴涛, age=21)
解析成功 Person(name=吴涛, age=21)
Person(name=吴涛, age=21)
```

编码解码器本质

- 编码器和解码器就是对InboundHandler和OutboundHandler的一种特殊实现，父类有许多主动保护内存不泄露的方法
- 还有MessageToMessage的Encoder和Decoder，另外也有Codec这种Inbound和Outbound都能拦截的编码器(实现了两种接口)
    - MessageToMessage系列不再是**bytBuf**和**T**之间的序列化，也可以是其他类型，要注意的是，如果想要序列化之前的对象继续存在不被父类方法GC，需要手动ReferenceCounted.retain()
    - MessageToMessage可以重写 acceptInboundMessage 方法判断是否丢弃这个消息
- 解码器利用了上面在handler中的**如果对传入的消息不完全读完或不读会怎样**的特性，实现将多个包合成一个包或者一个包拆包的功能

解码器源码中的注意点

```
protected void callDecode(ChannelHandlerContext ctx, ByteBuf in, List<Object>
out) {
   .....
```

```
        while (in.isReadable()) { .....
            int oldInputLength = in.readableBytes();
                decodeRemovalReentryProtection(ctx, in, out);
    .....

            if (out.isEmpty()) {
                // 空out直接退出处理而且消息没有处理过，直接不处理
                if (oldInputLength == in.readableBytes()) {
                    break;
                } else {
                // 空out且消息被处理过了，会循环处理到out不是空
                    continue;
                }
            }
            // 如果out里面有东西，消息却没有被读取过，那么会报这个 没有读消息却返回了数据的错误
            if (oldInputLength == in.readableBytes()) {
                throw new DecoderException(
                        StringUtil.simpleClassName(getClass()) +
                                ".decode() did not read anything but decoded a message.");
            }

            if (isSingleDecode()) {
                break;
            }
     ....
}
```

MessageAggregator消息聚合(重要)

```
public abstract class MessageAggregator<I, S, C extends ByteBufHolder, O extends ByteBufHolder>
        extends MessageToMessageDecoder<I> { ... }
```

使用方法(可以参照下面的Http的聚合器)

几种泛型的解释

名字	定义
`I`	它是 `S`,`C`,`O` 的父接口
`S`	表示开始类型数据，即头数据
`C`	表示内容体类型数据，也有可能表示某一项内容体数据
`O`	表示包含头和内容体的数据，聚合后的完成数据

重写下面的判断方法(用Http的聚合组件做例子)

```
    // 是不是消息开头(核心)
    @Override
    protected boolean isStartMessage(HttpObject msg) throws Exception {
        return false;
    }

    // 是不是消息内容(核心)
    @Override
    protected boolean isContentMessage(HttpObject msg) throws Exception {
        return false;
    }

    // 是不是最后的消息(核心)
    @Override
    protected boolean isLastContentMessage(HttpContent msg) throws Exception {
        return false;
    }

    // 是不是聚合结果的类型(核心)
    @Override
    protected boolean isAggregated(HttpObject msg) throws Exception {
        return false;
    }

    // 验证目前请求的长度是否符合聚合器规定的maxContentLength，new一个聚合器一般会指定
maxContentLength，如
    // new HttpObjectAggregator(65536)
    @Override
    protected boolean isContentLengthInvalid(HttpMessage start, int
maxContentLength) throws Exception {
        return false;
    }

    // 解码过程中是否发送中间信息给客户端比如处理http请求的100-continue就会有用，在Message
是startMessage会调用决定是否传消     // 息给客户端
    @Override
    protected Object newContinueResponse(HttpMessage start, int
maxContentLength, ChannelPipeline pipeline) throws Exception {
        return null;
    }

    // 是否在发送continueMessage后关闭掉连接，和newContinueResponse配套使用
    @Override
    protected boolean closeAfterContinueResponse(Object msg) throws Exception {
        return false;
    }

    // 在发送continueMessage后忽略后面的发送内容，和newContinueResponse配套使用
    @Override
    protected boolean ignoreContentAfterContinueResponse(Object msg) throws
Exception {
        return false;
    }
```

```
    // 最开始解码的整个信息对象，会被赋值给currentMessage，如果start是ByteBufHolder类型，传入的就是自己的content，否则就是     // 一个空的byteBuf(核心)
    @Override
    protected FullHttpMessage beginAggregation(HttpMessage start, ByteBuf content) throws Exception {
        return null;
    }
    // 聚合完毕的后处理钩子
    protected void finishAggregation(FullHttpMessage aggregated) throws Exception { }
```

netty数据聚合器需要重写的几个方法(来自网络)

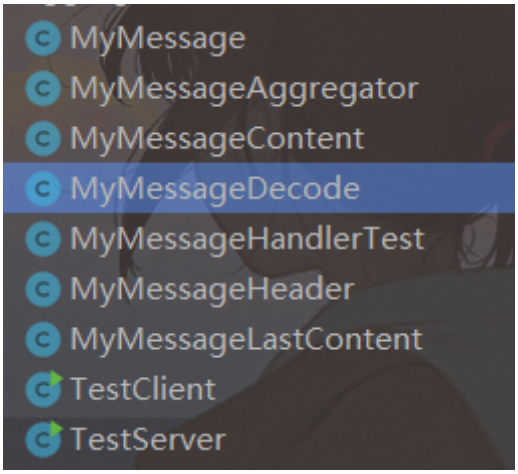
方法	定义
isStartMessage(I msg)	当且仅当给定数据 `msg` 是**开始数据类型**时返回 `true`
isContentMessage(I msg)	当且仅当给定数据 `msg` 是**内容项数据类型**时返回 `true`。
isLastContentMessage(C msg)	当且仅当给定数据 `msg` 是**最后一个内容项数据**时返回true。
isAggregated(I msg)	当且仅当给定数据 `msg` 是**聚合数据类型**时返回 `true`。
isContentLengthInvalid(S start, int maxContentLength)	确定消息开始的内容长度是否已知，以及它是否大于 `maxContentLength`。
newContinueResponse(S start, int maxContentLength, ChannelPipeline pipeline)	如果有需要指定开始消息返回数据，例如 `Http` 请求中 `100-continue` 报头；如果不需要，那么直接返回 `null`。
closeAfterContinueResponse(Object msg)	决定在写入 `newContinueResponse(Object, int, ChannelPipeline)` 的结果之后是否应该关闭通道。
ignoreContentAfterContinueResponse(Object msg)	确定是否应该忽略当前请求/响应的所有对象。 当下一次 `isContentMessage(Object)` 返回true时，这些数据会被忽略掉。
beginAggregation(S start, ByteBuf content)	使用开始类型数据 `S` 和给定缓存区 `content` 创建返回的聚合数据 `O`；给定缓存区 `content` 将会继续接收内容体的数据。如果开始类型数据 `S` 实现了 `ByteBufHolder`，它的数据将被添加到给定缓存区 `content` 中。
finishAggregation(O aggregated)	当聚合消息即将传递给管道中的下一个处理程序前调用。

MessageAggregator在什么时候合并了新消息

在静态的`appendPartialContent`方法中合并了buffer层面的消息，所以不需要手动去尝试合并，可以通过重写`aggregate`函数去定义消息处理后的操作，正如注释`Give the subtypes a chance to merge additional information such as trailing headers.`，给子类机会去添加额外的信息

具体实例

- MyMessage是完整的请求
- MyMessageContent是请求内容
- MyMessageHeader是请求头
- MyMessageLastContent是标记最后一次请求
- MyMessageDecode从byteBuf中解析出来MyMessageHeader或者MyMessageContent还是LastXXX



服务端

```
public class TestServer {

    public static void main(String[] args) {
        ServerBootstrap server = new ServerBootstrap();
        ChannelFuture future = server.group(new NioEventLoopGroup(), new
NioEventLoopGroup())
                .channel(NioServerSocketChannel.class)
                .childHandler(new ChannelInitializer<NioSocketChannel>() {
                    @Override
                    protected void initChannel(NioSocketChannel ch) throws
Exception {
                        ch.pipeline().addLast(new StringDecoder());
                        ch.pipeline().addLast(new StringEncoder());
                        ch.pipeline().addLast(new MyMessageDecode());
                        ch.pipeline().addLast(new MyMessageAggregator(1024));
                        ch.pipeline().addLast(new MyMessageHandlerTest());
                    }
                })
                .bind(8080);
        future.addListener(future1 -> {
            if (future1.isSuccess()) {
                System.out.println("启动成功");
            }
        });
    }
```

```
}
```

客户端

```
public class TestClient {
    public static void main(String[] args) {
        Bootstrap client = new Bootstrap();
        ChannelFuture connected = client.group(new NioEventLoopGroup())
                .channel(NioSocketChannel.class)
                .handler(new ChannelInitializer<NioSocketChannel>() {
                    @Override
                    protected void initChannel(NioSocketChannel ch) throws
Exception {
                        ch.pipeline().addLast(new StringEncoder());
                    }
                }).connect("localhost", 8080);
        connected.addListener(future -> {
           if (future.isSuccess()) {
               System.out.println("连接成功");
           }
        });
        Scanner scanner = new Scanner(System.in);
        String temp;
        while (!(temp = scanner.next()).equals("q")) {
            connected.channel().writeAndFlush(temp.trim()).addListener(f->{
                if (f.isSuccess()) {
                    System.out.println("发送成功");
                }
            });
        }
    }
}
```

MyMessageDecode

```
public class MyMessageDecode extends MessageToMessageDecoder<String> {
    @Override
    protected void decode(ChannelHandlerContext ctx, String msg, List<Object>
out) throws Exception {
        Object res;
        if (msg.startsWith("header")) {
            res = new MyMessageHeader(msg);
        }
        else if (msg.startsWith("last")) {
            res = new
MyMessageLastContent(ctx.alloc().buffer().writeBytes(msg.getBytes(StandardCharse
ts.UTF_8)));
        }
        else {
            res = new
MyMessageContent(ctx.alloc().buffer().writeBytes(msg.getBytes(StandardCharsets.U
TF_8)));
        }
        System.out.println("decode信息 " + msg);
```

```
        out.add(res);
    }
}
```

MyMessageAggregator

```
public class MyMessageAggregator extends MessageAggregator<Object,
MyMessageHeader, MyMessageContent, MyMessage> {


    public MyMessageAggregator(int maxContentLength) {
        super(maxContentLength);
    }

    @Override
    protected boolean isStartMessage(Object msg) throws Exception {
        return msg instanceof MyMessageHeader;
    }

    @Override
    protected boolean isContentMessage(Object msg) throws Exception {
        return msg instanceof MyMessageContent;
    }

    @Override
    protected boolean isLastContentMessage(MyMessageContent msg) throws
Exception {
        return msg instanceof MyMessageLastContent;
    }

    @Override
    protected boolean isAggregated(Object msg) throws Exception {
        return msg instanceof MyMessage;
    }

    @Override
    protected boolean isContentLengthInvalid(MyMessageHeader start, int
maxContentLength) throws Exception {
        return false;
    }

    @Override
    protected Object newContinueResponse(MyMessageHeader start, int
maxContentLength, ChannelPipeline pipeline) throws Exception {
        return null;
    }

    @Override
    protected boolean closeAfterContinueResponse(Object msg) throws Exception {
        return false;
    }


    @Override
```

```
    protected boolean ignoreContentAfterContinueResponse(Object msg) throws
Exception {
        return false;
    }


    @Override
    protected MyMessage beginAggregation(MyMessageHeader start, ByteBuf content)
throws Exception {
        MyMessage res = new MyMessage(content);
        res.setHeaderContent(start.getHeaderMsg());
        return res;
    }

    @Override
    protected void finishAggregation(MyMessage aggregated) throws Exception {
        super.finishAggregation(aggregated);
    }
}
```

几个消息类型

```
public class MyMessageContent extends DefaultByteBufHolder {

    public MyMessageContent(ByteBuf data) {
        super(data);
    }

}
```

```
public class MyMessage extends DefaultByteBufHolder {
    private String headerContent;
    public MyMessage(ByteBuf data) {
        super(data);
    }

    public String getHeaderContent() {
        return headerContent;
    }

    public void setHeaderContent(String headerContent) {
        this.headerContent = headerContent;
    }
}
```

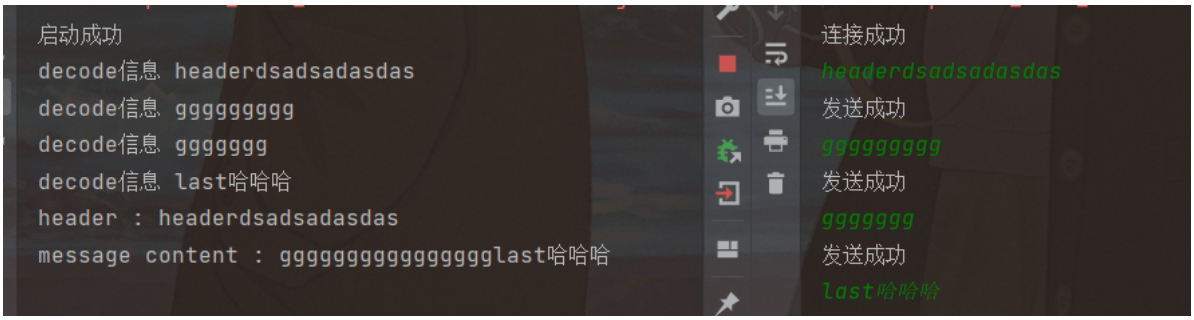
```
public class MyMessageHeader {
    private String headerMsg;

    public String getHeaderMsg() {
        return headerMsg;
    }

    public MyMessageHeader(String headerMsg) {
        this.headerMsg = headerMsg;
    }
}
```

```
public class MyMessageLastContent extends MyMessageContent{
    public MyMessageLastContent(ByteBuf data) {
        super(data);
    }
}
```

MyMessageHandlerTest

```
public class MyMessageHandlerTest extends SimpleChannelInboundHandler<MyMessage>
{
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, MyMessage msg) throws
Exception {
        String headerContent = msg.getHeaderContent();
        System.out.println("header : " + headerContent);
        ByteBuf content = msg.content();
        System.out.println("message content : " +
content.toString(StandardCharsets.UTF_8));
    }
}
```

效果



粘包半包问题

粘包：多个数据报一起发送过去

半包：大的数据报被拆分

解决方案

使用netty提供的编码解码器

LineBasedFrameDecoder使用

```
ch.pipeline().addHandlet(new LineBasedFrameDecoder(length));
```

- 源码核心业务

调用findEndOfLine找到`\n`或者`\r\n`或者其他的符号的Processer寻找临界点，然后发送

- 缺点：findEndOfLine是一个字节一个字节找，效率低

LTC解码器

即**LengthFieldBasedFrameDecoder**

有五个参数

```
public LengthFieldBasedFrameDecoder(
            int maxFrameLength,
            int lengthFieldOffset, int lengthFieldLength,
            int lengthAdjustment, int initialBytesToStrip)
```

1. maxFrameLength - 发送的数据帧最大长度
2. lengthFieldOffset - 定义长度域位于发送的字节数组中的下标。换句话说：发送的字节数组中下标为${lengthFieldOffset}的地方是长度域的开始地方
3. lengthFieldLength - 用于描述定义的长度域的长度。换句话说：发送字节数组bytes时, 字节数组bytes[lengthFieldOffset, lengthFieldOffset+lengthFieldLength]域对应于的定义长度域部分
4. lengthAdjustment - 解析之后，不算initialBytesToStrip跳过的字节，头与content之间的留存字节数，设置为0表示这两段之间没有空隙
5. initialBytesToStrip - 接收到的发送数据包，去除前initialBytesToStrip位

IdleStateHandler心跳/延迟检测

```
public IdleStateHandler(
        int readerIdleTimeSeconds,
        int writerIdleTimeSeconds,
        int allIdleTimeSeconds) {
```

readerIdleTimeSeconds：本端最多多久可以没有收到消息 0不生效

writerIdleTimeSeconds：本端最多多久没有发送消息 0不生效

allIdleTimeSeconds：上面的两个都触发这个 0不生效

```
/**
 * No data was received for a while.
 */
READER_IDLE,
/**
 * No data was sent for a while.
 */
WRITER_IDLE,
/**
 * No data was either received or sent for a while.
 */
ALL_IDLE
```

事件触发后会通过userChannelEvent(evt)方法向下面的handler传递事件

源码

- 是入站出站都执行的handler

```
public class IdleStateHandler extends ChannelDuplexHandler
```

实例

server端

```
public class HeartServer {
    public static void main(String[] args) {
        new ServerBootstrap()
                .group(new NioEventLoopGroup(), new NioEventLoopGroup())
                .channel(NioServerSocketChannel.class)
                .childHandler(new ChannelInitializer<NioSocketChannel>() {
                    @Override
                    protected void initChannel(NioSocketChannel ch) throws
Exception {
                        ch.pipeline().addLast(new IdleStateHandler(10,5,0));
                        ch.pipeline().addLast(new StringEncoder());
                        ch.pipeline().addLast(new ChannelInboundHandlerAdapter()
{
                            @Override
                            public void channelRead(ChannelHandlerContext ctx,
Object msg) throws Exception {
                                ByteBuf byteBuf = (ByteBuf) msg;
                                System.out.println("接收端 " +
byteBuf.toString(StandardCharsets.UTF_8));
                                ctx.channel().writeAndFlush("收到");
                            }

                            @Override
                            public void userEventTriggered(ChannelHandlerContext
ctx, Object evt) throws Exception {
                                IdleStateEvent event = (IdleStateEvent) evt;
```

```
                                if (event.state().equals(IdleState.WRITER_IDLE))
{
                                    ctx.channel().writeAndFlush("test-write");
                                }
                                else if
(event.state().equals(IdleState.READER_IDLE)){
                                    ctx.channel().writeAndFlush("test-read");
                                }
                            }
                        });
                    }
                }).bind(8080).addListener(future -> {
                    System.out.println("服务端启动成功");
                });
    }
}
```

client端

```
public class HeartClient {
    public static void main(String[] args) {
        ChannelFuture channelFuture = new Bootstrap()
                .group(new NioEventLoopGroup())
                .channel(NioSocketChannel.class)
                .handler(new ChannelInitializer<NioSocketChannel>() {
                    @Override
                    protected void initChannel(NioSocketChannel ch) throws
Exception {

                        ch.pipeline().addLast(new StringEncoder());
                        ch.pipeline().addLast(new StringDecoder());
                        ch.pipeline().addLast(new ChannelDuplexHandler() {
                            @Override
                            public void channelInactive(ChannelHandlerContext
ctx) throws Exception {
                                System.out.println("停止服务");
                                ctx.channel().close();
                            }
                        });
                        ch.pipeline().addLast(new ChannelInboundHandlerAdapter()
{
                            @Override
                            public void channelRead(ChannelHandlerContext ctx,
Object msg) throws Exception {
                                System.out.println(msg);
                                super.channelRead(ctx, msg);
                            }
                        });
                        ch.pipeline().addLast(new
SimpleChannelInboundHandler<String>() {
                            @Override
                            protected void channelRead0(ChannelHandlerContext
ctx, String msg) throws Exception {
                                System.out.println("发送过来 " + msg);
                            }
```

```
                        });
                    }
                }).connect("localhost", 8080);
        channelFuture.addListener(future -> {
            System.out.println("连接成功");
        });
        Scanner scanner = new Scanner(System.in);
        String temp;
        while (!(temp = scanner.next()).equals("q")) {
            if (temp.equals("send")) {
                channelFuture.channel().writeAndFlush("\n");
            }
            else {
                channelFuture.channel().writeAndFlush(temp);
            }
        }
        System.out.println("退出发送");
    }
}
```

Http请求的处理

内置Http的Handler

- **HttpRequestDecoder** 即把 ByteBuf 解码到 HttpRequest 和 HttpContent。
- **HttpResponseEncoder** 即把 HttpResponse 或 HttpContent 编码到 ByteBuf。
- **HttpServerCodec** 即 HttpRequestDecoder 和 HttpResponseEncoder 的结合。

上述的httpHandler会传递给下一个handler一个HttpRequest或者HttpResponse对象

请求聚合 HttpObjectAggregator

对于有请求体/返回的http请求，需要这个进行请求聚合，把HttpReuest和HttpResponse聚合为FullHttpRequest和FullHttpResponse

http分块报文结构

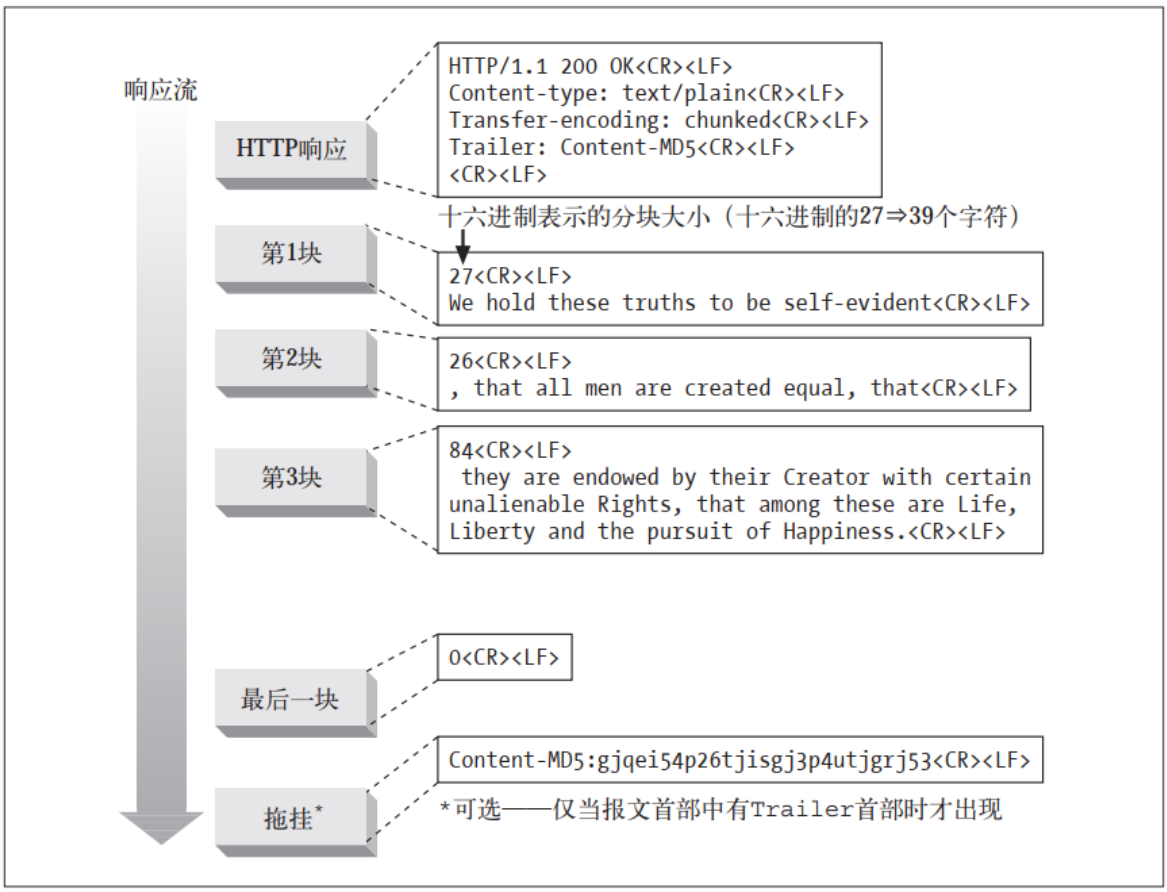


图 15-6　剖析分块编码报文

源码分析

HttpServerCodec

继承CombinedChannelDuplexHandler和HttpServerUpgradeHandler.SourceCodec

```
public final class HttpServerCodec extends
CombinedChannelDuplexHandler<HttpRequestDecoder, HttpResponseEncoder>
        implements HttpServerUpgradeHandler.SourceCodec {
```

看一下这个类的构造函数，发现都调用了init这个方法

```
/**
 * Creates a new instance with the specified decoder options.
 */
public HttpServerCodec(int maxInitialLineLength, int maxHeaderSize, int
maxChunkSize, boolean validateHeaders) {
    init(new HttpServerRequestDecoder(maxInitialLineLength, maxHeaderSize,
maxChunkSize, validateHeaders),
            new HttpServerResponseEncoder());
}

/**
 * Creates a new instance with the specified decoder options.
 */
public HttpServerCodec(int maxInitialLineLength, int maxHeaderSize, int
maxChunkSize, boolean validateHeaders,
```

```
                      int initialBufferSize) {
    init(
      new HttpServerRequestDecoder(maxInitialLineLength, maxHeaderSize,
maxChunkSize,
              validateHeaders, initialBufferSize),
      new HttpServerResponseEncoder());
}
```

进入init方法，发现这个方法是CombinedChannelDuplexHandler<HttpRequestDecoder, HttpResponseEncoder>里面的

观察这个类

```
/**
 *  Combines a {@link ChannelInboundHandler} and a {@link
ChannelOutboundHandler} into one {@link ChannelHandler}.
 */
public class CombinedChannelDuplexHandler<I extends ChannelInboundHandler, O
extends ChannelOutboundHandler>
        extends ChannelDuplexHandler {
```

发现是一个既可以处理入站又可以处理出站的handler，不同的是可以指定两个处理泛型，并使用两个handler处理了对应事件

```
   @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
        assert ctx == inboundCtx.ctx;
        if (!inboundCtx.removed) {
            inboundHandler.channelRead(inboundCtx, msg);
        } else {
            inboundCtx.fireChannelRead(msg);
        }
    }
....
@Override
public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise)
throws Exception {
    assert ctx == outboundCtx.ctx;
    if (!outboundCtx.removed) {
        outboundHandler.write(outboundCtx, msg, promise);
    } else {
        outboundCtx.write(msg, promise);
    }
}
```

HttpServerCodec传入init的两个类只是对HttpRequestDecoder和HttpResponseEncoder进行了简单包装，本质就是原来的东西，所以接下来终点就是分析这两个类的处理

HttpRequestDecoder

这个类里面除了一个构造空请求的方法之外，其他的都是对HttpObjectDecoder的进一步包装，所以接下来分析HttpObjectDecoder

```
public class HttpRequestDecoder extends HttpObjectDecoder
```

HttpObjectDecoder

简单看一下注释

```
/**
Decodes ByteBufs into HttpMessages and HttpContents.
...*/
public abstract class HttpObjectDecoder extends ByteToMessageDecoder
```

首先定义了一堆常量，如果构造函数不执行参数，就会用默认的这些

```
// 请求行最大长度
public static final int DEFAULT_MAX_INITIAL_LINE_LENGTH = 4096;
// 最大请求头长度
public static final int DEFAULT_MAX_HEADER_SIZE = 8192;
// 是否默认支持分块传输
public static final boolean DEFAULT_CHUNKED_SUPPORTED = true;
// 支持请求默认分块
public static final boolean DEFAULT_ALLOW_PARTIAL_CHUNKS = true;
// 最大分块大小
public static final int DEFAULT_MAX_CHUNK_SIZE = 8192;
// 默认验证报头
public static final boolean DEFAULT_VALIDATE_HEADERS = true;
public static final int DEFAULT_INITIAL_BUFFER_SIZE = 128;
public static final boolean DEFAULT_ALLOW_DUPLICATE_CONTENT_LENGTHS = false;
```

直接来看decode

```
@Override
protected void decode(ChannelHandlerContext ctx, ByteBuf buffer, List<Object>
out) throws Exception {
    // 重置，让handler可以处理下一个请求
    if (resetRequested) {
        resetNow();
    }

    switch (currentState) {
    // 这里没有做什么，而且因为没有break，所以进入下面的READ_INITIAL
    case SKIP_CONTROL_CHARS:
        // Fall-through
    // READ_INITIAL代表刚开始读
    case READ_INITIAL: try {
        // 读取请求行，详情见下面的LineParser解释
        AppendableCharSequence line = lineParser.parse(buffer);
        // 读取没成功，代表请求行数据还没发完，等待发完
        if (line == null) {
```

```
            return;
        }
        // 分离请求头的信息，详情见下面的分析
        String[] initialLine = splitInitialLine(line);
        if (initialLine.length < 3) {
            // 信息没有三个就要等待
            // Invalid initial line - ignore.
            currentState = State.SKIP_CONTROL_CHARS;
            return;
        }
        // 解析完毕，生成带有请求行的空信息
        message = createMessage(initialLine);
        // 设置状态为等待读取头信息的状态
        currentState = State.READ_HEADER;
        // fall-through
    } catch (Exception e) {
        out.add(invalidMessage(buffer, e));
        return;
    }
    case READ_HEADER: try {
        // 重点读取头信息的这个函数，见下面的readerHeaders
        State nextState = readHeaders(buffer);
        // 代表着没读完，继续等着
        if (nextState == null) {
            return;
        }
        // 设置状态
        currentState = nextState;
        switch (nextState) {
        // 没有请求体，直接处理完结束，并重置组件各项数据
        case SKIP_CONTROL_CHARS:
            // fast-path
            // No content is expected.
            out.add(message);
            out.add(LastHttpContent.EMPTY_LAST_CONTENT);
            resetNow();
            return;
        // 意味着读取分块数据，先把消息体传过去，不改变组件状态，下一次就到外面Switch语句里面
正式处理分块
        case READ_CHUNK_SIZE:
            // 先检查是否支持分块，不支持就报错
            if (!chunkedSupported) {
                throw new IllegalArgumentException("Chunked messages not
supported");
            }
            // Chunked encoding - generate HttpMessage first.  HttpChunks will
follow.
            out.add(message);
            return;
        default:
            /**
             * <a href="https://tools.ietf.org/html/rfc7230#section-3.3.3">RFC
7230, 3.3.3</a> states that if a
             * request does not have either a transfer-encoding or a content-
length header then the message body
```

```
             * length is 0. However for a response the body length is the number
of octets received prior to the
             * server closing the connection. So we treat this as variable
length chunked encoding.
             */
            long contentLength = contentLength();
            // 处理content-length = 0 或者没有的情况
            // 直接传下去，不解析请求体，留给后面的处理
            if (contentLength == 0 || contentLength == -1 &&
isDecodingRequest()) {
                out.add(message);
                out.add(LastHttpContent.EMPTY_LAST_CONTENT);
                resetNow();
                return;
            }
            // 保证不是分块数据
            assert nextState == State.READ_FIXED_LENGTH_CONTENT ||
                    nextState == State.READ_VARIABLE_LENGTH_CONTENT;
            // 放入信息发送
            out.add(message);

            if (nextState == State.READ_FIXED_LENGTH_CONTENT) {
                // chunkSize will be decreased as the READ_FIXED_LENGTH_CONTENT
state reads data chunk by chunk.
                // 因为定长没有分块的情况下，数据可能会很大，所以处理的时候还是类似分块的逻
辑，一批一批处理，只不过不用从                    // 每个分块里面获取分块长度，读请求头时就获
取完毕了，这里用chunkSize充当整个块的大小，放着对contentLength变                    // 量的
修改
                chunkSize = contentLength;
            }

            // We return here, this forces decode to be called again where we
will decode the content
            // 这里由ByteToMessageDecoder调用循环处理，符合 out有数据而 buffer没读完的情
况
            return;
        }
    } catch (Exception e) {
        out.add(invalidMessage(buffer, e));
        return;
    }
    case READ_VARIABLE_LENGTH_CONTENT: {
        // Keep reading data as a chunk until the end of connection is reached.
        // 读取可变长度状态会一直读取请求体内容，直到超时或者客户断开
        int toRead = Math.min(buffer.readableBytes(), maxChunkSize);
        if (toRead > 0) {
            ByteBuf content = buffer.readRetainedSlice(toRead);
            out.add(new DefaultHttpContent(content));
        }
        return;
    }
    case READ_FIXED_LENGTH_CONTENT: {
        // 和上面READ_HEADERS default块里面讲的处理逻辑一样，每读一点，chunkSize(即剩下的
contentLength)就会减去一点
        // 直到0结束发送一个LastContent
```

```
        int readLimit = buffer.readableBytes();

        // Check if the buffer is readable first as we use the readable byte
count
        // to create the HttpChunk. This is needed as otherwise we may end up
with
        // create an HttpChunk instance that contains an empty buffer and so is
        // handled like it is the last HttpChunk.
        //
        // See https://github.com/netty/netty/issues/433
        if (readLimit == 0) {
            return;
        }

        int toRead = Math.min(readLimit, maxChunkSize);
        if (toRead > chunkSize) {
            toRead = (int) chunkSize;
        }
        ByteBuf content = buffer.readRetainedSlice(toRead);
        chunkSize -= toRead;

        if (chunkSize == 0) {
            // Read all content.
            out.add(new DefaultLastHttpContent(content, validateHeaders));
            resetNow();
        } else {
            out.add(new DefaultHttpContent(content));
        }
        return;
    }
    /**
     * everything else after this point takes care of reading chunked content.
basically, read chunk size,
     * read chunk, read and ignore the CRLF and repeat until 0
     */
    // 使用LineParser这个组件来读分块长度
    case READ_CHUNK_SIZE: try {
        // 解析分块头信息
        AppendableCharSequence line = lineParser.parse(buffer);
        if (line == null) {
            return;
        }
        // 从分块头信息获得分块大小
        int chunkSize = getChunkSize(line.toString());
        this.chunkSize = chunkSize;
        // 分块大小等于0代表着分块读完
        if (chunkSize == 0) {
            currentState = State.READ_CHUNK_FOOTER;
            return;
        }
        // 分块不为0就需要去读content
        currentState = State.READ_CHUNKED_CONTENT;
        // fall-through
    } catch (Exception e) {
        out.add(invalidChunk(buffer, e));
```

```
            return;
        }
        // 读取分块，为了防止分块过大，产生读的性能问题，对分块内容也进行了一批一批处理
        case READ_CHUNKED_CONTENT: {
            assert chunkSize <= Integer.MAX_VALUE;
            int toRead = Math.min((int) chunkSize, maxChunkSize);
            if (!allowPartialChunks && buffer.readableBytes() < toRead) {
                return;
            }
            toRead = Math.min(toRead, buffer.readableBytes());
            if (toRead == 0) {
                return;
            }
            HttpContent chunk = new
DefaultHttpContent(buffer.readRetainedSlice(toRead));
            chunkSize -= toRead;
            // 发给下一个handler
            out.add(chunk);
            // 这个分块还有数据，继续读取
            if (chunkSize != 0) {
                return;
            }
            // 这个分块没有数据了，意味着读\r\n 即READ_CHUNK_DELIMITER
            currentState = State.READ_CHUNK_DELIMITER;
            // fall-through
        }
        case READ_CHUNK_DELIMITER: {
            // 分块读完，重新设置到READ_CHUNK_SIZE，读取下一个分块
            final int wIdx = buffer.writerIndex();
            int rIdx = buffer.readerIndex();
            while (wIdx > rIdx) {
                byte next = buffer.getByte(rIdx++);
                if (next == HttpConstants.LF) {
                    currentState = State.READ_CHUNK_SIZE;
                    break;
                }
            }
            buffer.readerIndex(rIdx);
            return;
        }
        // 分块全部读完，发送lastContent，重置本组件
        case READ_CHUNK_FOOTER: try {
            LastHttpContent trailer = readTrailingHeaders(buffer);
            if (trailer == null) {
                return;
            }
            out.add(trailer);
            resetNow();
            return;
        } catch (Exception e) {
            out.add(invalidChunk(buffer, e));
            return;
        }
        case BAD_MESSAGE: {
            // Keep discarding until disconnection.
```

```
            buffer.skipBytes(buffer.readableBytes());
            break;
        }
        case UPGRADED: {
            int readableBytes = buffer.readableBytes();
            if (readableBytes > 0) {
                // Keep on consuming as otherwise we may trigger an
DecoderException,
                // other handler will replace this codec with the upgraded protocol
codec to
                // take the traffic over at some point then.
                // See https://github.com/netty/netty/issues/2173
                out.add(buffer.readBytes(readableBytes));
            }
            break;
        }
        default:
            break;
        }
}
```

LineParser是什么

是HttpObjectDecoder一个用来解析请求行的类

```
// 继承自HeaderParser
private final class LineParser extends HeaderParser {

    LineParser(AppendableCharSequence seq, int maxLength) {
        super(seq, maxLength);
    }
    // 可见是调用了父类的parse方法，详情见下面的HeaderParser
    @Override
    public AppendableCharSequence parse(ByteBuf buffer) {
        // Suppress a warning because HeaderParser.reset() is supposed to be
called
        reset();    // lgtm[java/subtle-inherited-call]
        return super.parse(buffer);
    }

    @Override
    // 重写了父类的process
    public boolean process(byte value) throws Exception {
        // 必须是在刚开始读的阶段，这里专门用来处理请求行
        if (currentState == State.SKIP_CONTROL_CHARS) {
            char c = (char) (value & 0xFF);
            // 遇到空格就停住
            if (Character.isISOControl(c) || Character.isWhitespace(c)) {
                increaseCount();
                return true;
            }
            currentState = State.READ_INITIAL;
        }
```

```
        // 调用HeaderParser主要是获取\r\n位置，换句话说，HeaderParser这个方法既可以当做解析请求头，分块传输头，那么再加上        //LineParser在这个方法上面的加强，使得LineParser的这个方法可以解析请求行，请求头，分块传输块头的一切有\r\n的数据体
        return super.process(value);
    }

    @Override
    protected TooLongFrameException newException(int maxLength) {
        return new TooLongHttpLineException("An HTTP line is larger than " +
maxLength + " bytes.");
    }
}
```

HeaderParser是什么

```
// 本质是一个ByteProcesser，用来和ByteBuf的forEachByte一起使用，范围process方法返回true的index
private static class HeaderParser implements ByteProcessor {
    private final AppendableCharSequence seq;
    private final int maxLength;
    int size;

    HeaderParser(AppendableCharSequence seq, int maxLength) {
        this.seq = seq;
        this.maxLength = maxLength;
    }
    // parse方法实现
    public AppendableCharSequence parse(ByteBuf buffer) {
        final int oldSize = size;
        // 重置这一段，让下次也能调用
        seq.reset();
        // 注意这里会执行process里面的方法，这个line的长度精确测量
        int i = buffer.forEachByte(this);
        if (i == -1) {
            size = oldSize;
            return null;
        }
        // 读过的就舍弃
        buffer.readerIndex(i + 1);
        return seq;
    }

    public void reset() {
        size = 0;
    }

    @Override
    public boolean process(byte value) throws Exception {
        // 拿到字符
        char nextByte = (char) (value & 0xFF);
        // 是不是换行符，换行符不加入数据
        if (nextByte == HttpConstants.LF) {
            int len = seq.length();
            // Drop CR if we had a CRLF pair
            // \r\n的处理，剔除之前的\r
```

```
            if (len >= 1 && seq.charAtUnsafe(len - 1) == HttpConstants.CR) {
                -- size;
                seq.setLength(len - 1);
            }
            return false;
        }
        // 保证能精确测量到这个line的长度
        increaseCount();
        // 其他的字符添加到返回值里面
        seq.append(nextByte);
        return true;
    }

    protected final void increaseCount() {
        if (++ size > maxLength) {
            // TODO: Respond with Bad Request and discard the traffic
            //    or close the connection.
            //       No need to notify the upstream handlers - just log.
            //       If decoding a response, just throw an exception.
            throw newException(maxLength);
        }
    }

    protected TooLongFrameException newException(int maxLength) {
        return new TooLongHttpHeaderException("HTTP header is larger than " +
maxLength + " bytes.");
    }
}
```

readHeaders方法详解

```
// 读取每个头信息加到message中并返回状态
// 这个buffer还是decode里面那个buffer，所以可以一直拿到数据
private State readHeaders(ByteBuf buffer) {
    final HttpMessage message = this.message;
    // 装备往里面放请求头
    final HttpHeaders headers = message.headers();
    // 读取头的一行
    AppendableCharSequence line = headerParser.parse(buffer);
    // 没读到就继续等待发送
    if (line == null) {
        return null;
    }
    // 读到了一行请求头
    if (line.length() > 0) {
        do {
            // 处理line上的空字符
            char firstChar = line.charAtUnsafe(0);
            if (name != null && (firstChar == ' ' || firstChar == '\t')) {
                //please do not make one line from below code
                //as it breaks +XX:OptimizeStringConcat optimization
                String trimmedLine = line.toString().trim();
                String valueStr = String.valueOf(value);
                value = valueStr + ' ' + trimmedLine;
            } else {
```

```
                // name不为null就表示刚才解析出来了一个
                if (name != null) {
                    headers.add(name, value);
                }
                // 分离请求头等待下一次循环添加到httpHeaders中
                splitHeader(line);
            }
            // 读下一个头信息并处理
            line = headerParser.parse(buffer);
            // 同样未读到就等待
            if (line == null) {
                return null;
            }
        } while (line.length() > 0);
    }
    // 因为添加总比解析慢一步，这里就要加一下最后一个header信息
    // Add the last header.
    if (name != null) {
        headers.add(name, value);
    }
    // 重置方便下次调用
    // reset name and value fields
    name = null;
    value = null;

    // 下面是根据请求头信息中的content-leangth和transfer-encoding为chunk时做一番处理，分为以下几种情况
    // 有content-length，
    // 没有content-length但是有chunk(分段请求)
    // 没content-length和chunk

    // Done parsing initial line and headers. Set decoder result.
    // 把请求行大小和请求头大小放到一个临时变量中
    HttpMessageDecoderResult decoderResult = new HttpMessageDecoderResult(lineParser.size, headerParser.size);
    message.setDecoderResult(decoderResult);
    // 拿到所有content-length
    List<String> contentLengthFields = headers.getAll(HttpHeaderNames.CONTENT_LENGTH);
    // content-length存在
    if (!contentLengthFields.isEmpty()) {
        // 下面两步是协议相关的设置
        HttpVersion version = message.protocolVersion();
        boolean isHttp10OrEarlier = version.majorVersion() < 1 || (version.majorVersion() == 1
                && version.minorVersion() == 0);
        // Guard against multiple Content-Length headers as stated in
        // https://tools.ietf.org/html/rfc7230#section-3.3.2:
        // 获得content-length，这里有个allowDuplicateContentLengths表示是否允许多个content-length
        // contentLength是HttpObjectParser的属性，获取不了就是-1
        contentLength = HttpUtil.normalizeAndGetContentLength(contentLengthFields,
                isHttp10OrEarlier, allowDuplicateContentLengths);
        // 有content-length就付给headers
```

```
            if (contentLength != -1) {
                headers.set(HttpHeaderNames.CONTENT_LENGTH, contentLength);
            }
        }
        // 这个在这里总是false，详情见源码
        if (isContentAlwaysEmpty(message)) {
            HttpUtil.setTransferEncodingChunked(message, false);
            return State.SKIP_CONTROL_CHARS;

        }
        // 下面的处理就是分不分块的问题
        // 代表着拥有transfer-encoding:chunk
        /**
            public static boolean isTransferEncodingChunked(HttpMessage message) {
            return
message.headers().containsValue(HttpHeaderNames.TRANSFER_ENCODING,
HttpHeaderValues.CHUNKED, true);
        }
        */
        else if (HttpUtil.isTransferEncodingChunked(message)) {
            // transfer的优先级比content-length要高的，所以返回READ_CHUNK_SIZE
            if (!contentLengthFields.isEmpty() && message.protocolVersion() ==
HttpVersion.HTTP_1_1) {
                handleTransferEncodingChunkedWithContentLength(message);
            }
            return State.READ_CHUNK_SIZE;
        } else if (contentLength() >= 0) {
            // 有content-length就使用固定长度模式
            return State.READ_FIXED_LENGTH_CONTENT;
        } else {
            // 可变长度处理，没有content-length和chunk，代表着没分块的整体大数据
            return State.READ_VARIABLE_LENGTH_CONTENT;
        }
    }
```

split系列方法分离请求行/请求头数据

splitInitialLine

```
// 可以看到主要就把三个信息分离出来作为字符串数组返回
private static String[] splitInitialLine(AppendableCharSequence sb) {
    int aStart;
    int aEnd;
    int bStart;
    int bEnd;
    int cStart;
    int cEnd;
    // 第一个数据开始位置
    aStart = findNonSPLenient(sb, 0);
    // 第一个空格位置
    aEnd = findSPLenient(sb, aStart);
    // 第二个数据开始位置
    bStart = findNonSPLenient(sb, aEnd);
    // 第二个空格
    bEnd = findSPLenient(sb, bStart);
```

```
    // 第三个数据开始位置
    cStart = findNonSPLenient(sb, bEnd);
    // 找到结尾
    cEnd = findEndOfString(sb);

    return new String[] {
            sb.subStringUnsafe(aStart, aEnd),
            sb.subStringUnsafe(bStart, bEnd),
            cStart < cEnd? sb.subStringUnsafe(cStart, cEnd) : "" };
}
```

splitHeader

```
// 分离请求头信息主要就是 name 冒号 请求头值三样，要注意这里的name和value其实是在
HttpObjectDecoder中，所以可以被其他方法拿
//到这个void函数处理的结果
private void splitHeader(AppendableCharSequence sb) {
    final int length = sb.length();
    // 用于分割的index
    int nameStart;
    int nameEnd;
    int colonEnd;
    int valueStart;
    int valueEnd;

    nameStart = findNonWhitespace(sb, 0);
    for (nameEnd = nameStart; nameEnd < length; nameEnd ++) {
        char ch = sb.charAtUnsafe(nameEnd);
        // https://tools.ietf.org/html/rfc7230#section-3.2.4
        //
        // No whitespace is allowed between the header field-name and colon. In
        // the past, differences in the handling of such whitespace have led to
        // security vulnerabilities in request routing and response handling. A
        // server MUST reject any received request message that contains
        // whitespace between a header field-name and colon with a response code
        // of 400 (Bad Request). A proxy MUST remove any such whitespace from a
        // response message before forwarding the message downstream.
        if (ch == ':' ||
                // In case of decoding a request we will just continue processing
and header validation
                // is done in the DefaultHttpHeaders implementation.
                //
                // In the case of decoding a response we will "skip" the
whitespace.
                (!isDecodingRequest() && isOWS(ch))) {
            break;
        }
    }

    if (nameEnd == length) {
        // There was no colon present at all.
        throw new IllegalArgumentException("No colon found");
    }
```

```
    for (colonEnd = nameEnd; colonEnd < length; colonEnd ++) {
        if (sb.charAtUnsafe(colonEnd) == ':') {
            colonEnd ++;
            break;
        }
    }

    name = sb.subStringUnsafe(nameStart, nameEnd);
    valueStart = findNonWhitespace(sb, colonEnd);
    if (valueStart == length) {
        value = EMPTY_VALUE;
    } else {
        valueEnd = findEndOfString(sb);
        value = sb.subStringUnsafe(valueStart, valueEnd);
    }
}
```

HttpRequestDecoder源码总结

本质调用了HttpObjectDecoder

netty的http解析过程，是一个循环往复的过程，根据http协议特点，会解析request，也会解析response，会首先构造一个 HttpMessage 来存储request或者response的 请求行和header信息
在HttpMessage后面，会有一个或者多个HttpContent信息，HttpContent中主要会包ByteBuf信息。
在结尾，会带上一个LastHttpContent 类型的作为结尾。

HttpResponseEncoder

本质调用了HttpObjectEncoder，接下来详细看HttpObjectEncoder

HttpObjectEncoder

看出来是继承MessageToMessageEncoder

```
public abstract class HttpObjectEncoder<H extends HttpMessage> extends
MessageToMessageEncoder<Object> {
```

MessageToMessageEncoder的泛型代表什么

代表着消息Encode之后的类型

encode方法执行逻辑

有三种可能的情况

- msg是HttpMessage
- msg是HttpContent
- msg是ByteBuf
- msg是FileRegion

```
protected void encode(ChannelHandlerContext ctx, Object msg, List<Object> out)
throws Exception {
    ByteBuf buf = null;
    if (msg instanceof HttpMessage) {
        if (state != ST_INIT) {
```

```
            throw new IllegalStateException("unexpected message type: " +
StringUtil.simpleClassName(msg)
                    + ", state: " + state);
        }

        @SuppressWarnings({ "unchecked", "CastConflictsWithInstanceof" })
        H m = (H) msg;
        // 开一个buf空间
        buf = ctx.alloc().buffer((int) headersEncodedSizeAccumulator);
        // Encode the message.
        // 这个方法是编码请求行
        /**
            protected void encodeInitialLine(ByteBuf buf, HttpResponse response)
throws Exception {
                response.protocolVersion().encode(buf);
                buf.writeByte(SP);
                response.status().encode(buf);
                ByteBufUtil.writeShortBE(buf, CRLF_SHORT);
            }
        */
        encodeInitialLine(buf, m);
        // isContentAlwaysEmpty(m)一直返回false，就看
HttpUtil.isTransferEncodingChunked，通过这个来确定是不是
        // chunked编码格式
        state = isContentAlwaysEmpty(m) ? ST_CONTENT_ALWAYS_EMPTY :
                HttpUtil.isTransferEncodingChunked(m) ? ST_CONTENT_CHUNK :
ST_CONTENT_NON_CHUNK;
        // 检查msg的头信息
        // ST_CONTENT_ALWAYS_EMPTY表示这个msg返回体是不是总是空的，在HttpMessage情况下
state == ST_CONTENT_ALWAYS_EMPTY
        // 是false，这个函数没有做任何处理
        sanitizeHeadersBeforeEncode(m, state == ST_CONTENT_ALWAYS_EMPTY);
        // 编码头部信息，详情见下面的encodeHeaders
        encodeHeaders(m.headers(), buf);
        // 写入一个\r\n代表请求头结束
        ByteBufUtil.writeShortBE(buf, CRLF_SHORT);
        //
        headersEncodedSizeAccumulator = HEADERS_WEIGHT_NEW *
padSizeForAccumulation(buf.readableBytes()) +
                                        HEADERS_WEIGHT_HISTORICAL *
headersEncodedSizeAccumulator;
    }

    // Bypass the encoder in case of an empty buffer, so that the following idiom
works:
    //
    //
ch.write(Unpooled.EMPTY_BUFFER).addListener(ChannelFutureListener.CLOSE);
    //
    // See https://github.com/netty/netty/issues/2983 for more information.
    // 如果msg是Buf，简单检查一下是否还可以读，不可以就return等待
    if (msg instanceof ByteBuf) {
        final ByteBuf potentialEmptyBuf = (ByteBuf) msg;
        if (!potentialEmptyBuf.isReadable()) {
            out.add(potentialEmptyBuf.retain());
```

```
            return;
        }
    }
    // httpContent，byteBuf和fileRegion类型
    if (msg instanceof HttpContent || msg instanceof ByteBuf || msg instanceof
FileRegion) {
        switch (state) {
            // 意味着http传输的时候，第一个消息必须是httpMessage否则就会报错!
            case ST_INIT:
                throw new IllegalStateException("unexpected message type: " +
StringUtil.simpleClassName(msg)
                        + ", state: " + state);
            // 不是分块传
            case ST_CONTENT_NON_CHUNK:
                // 拿到数据长度，详情见下面的contentLength方法
                final long contentLength = contentLength(msg);
                // 只对有响应体的处理
                if (contentLength > 0) {
                    if (buf != null && buf.writableBytes() >= contentLength &&
msg instanceof HttpContent           /* 必须是httpContent才处理 */) {
                        // merge into other buffer for performance reasons
                        // 往buf里面写响应体数据
                        buf.writeBytes(((HttpContent) msg).content());
                        // 放到out里面发送
                        out.add(buf);
                    } else {
                        /* 不是HttpContent */
                        if (buf != null) {
                            out.add(buf);
                        }
                        /* 这里只进行了retain */
                        out.add(encodeAndRetain(msg));
                    }
                   // 最后一个LastContent，重置组件
                    if (msg instanceof LastHttpContent) {
                        state = ST_INIT;
                    }

                    break;
                }

                // fall-through!
            case ST_CONTENT_ALWAYS_EMPTY:

                if (buf != null) {
                    // We allocated a buffer so add it now.
                    out.add(buf);
                } else {
                    // Need to produce some output otherwise an
                    // IllegalStateException will be thrown as we did not write
anything
                    // Its ok to just write an EMPTY_BUFFER as if there are
reference count issues these will be
                    // propagated as the caller of the encode(...) method will
release the original
```

```
                    // buffer.
                    // Writing an empty buffer will not actually write anything
on the wire, so if there is a user
                    // error with msg it will not be visible externally
                    out.add(Unpooled.EMPTY_BUFFER);
                }

                break;
            // 处理分块的消息
            case ST_CONTENT_CHUNK:
                // HttpMessage有请求头和content，在第一次接收的时候会把头信息解析到buf里
面，所以需要发送一次头部信息
                if (buf != null) {
                    // We allocated a buffer so add it now.
                    out.add(buf);
                }
                // 详见下面的encodeChunkedContent
                encodeChunkedContent(ctx, msg, contentLength(msg), out);

                break;
            default:
                throw new Error();
        }

        if (msg instanceof LastHttpContent) {
            state = ST_INIT;
        }
    } else if (buf != null) {
        out.add(buf);
    }
}
```

encodeHeaders方法

拿到headers的迭代器，调用HttpHeadersEncoder.encoderHeader往buf里面写数据

```
protected void encodeHeaders(HttpHeaders headers, ByteBuf buf) {
    Iterator<Entry<CharSequence, CharSequence>> iter =
headers.iteratorCharSequence();
    while (iter.hasNext()) {
        Entry<CharSequence, CharSequence> header = iter.next();
        HttpHeadersEncoder.encoderHeader(header.getKey(), header.getValue(),
buf);
    }
}
```

HttpHeadersEncoder.encoderHeader方法就是普通的写数据

```
static void encoderHeader(CharSequence name, CharSequence value, ByteBuf buf) {
    final int nameLen = name.length();
    final int valueLen = value.length();
    final int entryLen = nameLen + valueLen + 4;
    buf.ensureWritable(entryLen);
    int offset = buf.writerIndex();
```

```
    writeAscii(buf, offset, name);
    offset += nameLen;
    ByteBufUtil.setShortBE(buf, offset, COLON_AND_SPACE_SHORT);
    offset += 2;
    writeAscii(buf, offset, value);
    offset += valueLen;
    ByteBufUtil.setShortBE(buf, offset, CRLF_SHORT);
    offset += 2;
    buf.writerIndex(offset);
}
```

contentLength方法

分别对三种不同的类型进行了处理

```
private static long contentLength(Object msg) {
    if (msg instanceof HttpContent) {
        return ((HttpContent) msg).content().readableBytes();
    }
    if (msg instanceof ByteBuf) {
        return ((ByteBuf) msg).readableBytes();
    }
    if (msg instanceof FileRegion) {
        return ((FileRegion) msg).count();
    }
    throw new IllegalStateException("unexpected message type: " +
StringUtil.simpleClassName(msg));
}
```

encodeChunkedContent

解析分块的HttpContent

```
private void encodeChunkedContent(ChannelHandlerContext ctx, Object msg, long
contentLength, List<Object> out) {
    // 有分块内容
    if (contentLength > 0) {
        // 书写分块的协议
        String lengthHex = Long.toHexString(contentLength);
        ByteBuf buf = ctx.alloc().buffer(lengthHex.length() + 2);
        buf.writeCharSequence(lengthHex, CharsetUtil.US_ASCII);
        ByteBufUtil.writeShortBE(buf, CRLF_SHORT);
        out.add(buf);
        out.add(encodeAndRetain(msg));
        out.add(CRLF_BUF.duplicate());
    }
    // 是最后的内容
    if (msg instanceof LastHttpContent) {
        // 处理tailer问题
        HttpHeaders headers = ((LastHttpContent) msg).trailingHeaders();
        if (headers.isEmpty()) {
            out.add(ZERO_CRLF_CRLF_BUF.duplicate());
        } else {
```

```
            ByteBuf buf = ctx.alloc().buffer((int)
trailersEncodedSizeAccumulator);
            ByteBufUtil.writeMediumBE(buf, ZERO_CRLF_MEDIUM);
            encodeHeaders(headers, buf);
            ByteBufUtil.writeShortBE(buf, CRLF_SHORT);
            trailersEncodedSizeAccumulator = TRAILERS_WEIGHT_NEW *
padSizeForAccumulation(buf.readableBytes()) +
                                             TRAILERS_WEIGHT_HISTORICAL *
trailersEncodedSizeAccumulator;
            out.add(buf);
        }
    } else if (contentLength == 0) {
        // Need to produce some output otherwise an
        // IllegalStateException will be thrown
        out.add(encodeAndRetain(msg));
    }
}
```

HttpMessage，HttpContent等接口与实现类的作用

这两个接口构成了netty描述数据的基本逻辑

HttpMessage下面的接口和实现类是**Http协议普遍实现，下面的接口主要聚焦在http头部分，没有content**，httpContent是**传输http体与分块http协议的接口，主要聚焦在http体部分**

httpMessage

httpMessage实现了一般化的http元素的方法

```
public interface HttpMessage extends HttpObject {

    /**
     * @deprecated Use {@link #protocolVersion()} instead.
     */
    @Deprecated
    HttpVersion getProtocolVersion();

    /**
     * Returns the protocol version of this {@link HttpMessage}
     */
    HttpVersion protocolVersion();

    /**
     * Set the protocol version of this {@link HttpMessage}
     */
    HttpMessage setProtocolVersion(HttpVersion version);

    /**
     * Returns the headers of this message.
     */
    HttpHeaders headers();
}
```

HttpMessage旗下有三个关键接口 HttpRequest，HttpResponse和没有继承HttpMessage的枚举类HttpHeaders

httpRequest和HttpResponse

在httpMessage接口上添加了更多具体实现的方法

```
public interface HttpRequest extends HttpMessage {

    /**
     * @deprecated Use {@link #method()} instead.
     */
    @Deprecated
    HttpMethod getMethod();

    /**
     * Returns the {@link HttpMethod} of this {@link HttpRequest}.
     *
     * @return The {@link HttpMethod} of this {@link HttpRequest}
     */
    HttpMethod method();

    /**
     * Set the {@link HttpMethod} of this {@link HttpRequest}.
     */
    HttpRequest setMethod(HttpMethod method);

    /**
     * @deprecated Use {@link #uri()} instead.
     */
    @Deprecated
    String getUri();

    /**
     * Returns the requested URI (or alternatively, path)
     *
     * @return The URI being requested
     */
    String uri();

    /**
     *  Set the requested URI (or alternatively, path)
     */
    HttpRequest setUri(String uri);

    @Override
    HttpRequest setProtocolVersion(HttpVersion version);
}
```

```
public interface HttpResponse extends HttpMessage {

    /**
     * @deprecated Use {@link #status()} instead.
     */
    @Deprecated
    HttpResponseStatus getStatus();

    /**
```

```
     * Returns the status of this {@link HttpResponse}.
     *
     * @return The {@link HttpResponseStatus} of this {@link HttpResponse}
     */
    HttpResponseStatus status();

    /**
     * Set the status of this {@link HttpResponse}.
     */
    HttpResponse setStatus(HttpResponseStatus status);

    @Override
    HttpResponse setProtocolVersion(HttpVersion version);
}
```

httpRequest和HttpResponse提供的默认实现类

DefaultHttpRequest和DefaultHttpResponse两个默认实现类

一般都使用这两个类或者在这两类上面扩展

httpContent

httpContent下面主要是有lastContent和非lastContent的区别，在传输过程中标记是否是最后的msg

```
/**
 * The last {@link HttpContent} which has trailing headers.
 */
public interface LastHttpContent extends HttpContent
```

httpContent一般使用它的或者拓展默认实现类

```
/**
 * The default {@link HttpContent} implementation.
 */
public class DefaultHttpContent extends DefaultHttpObject implements HttpContent
```

```
/**
 * The default {@link LastHttpContent} implementation.
 */
public class DefaultLastHttpContent extends DefaultHttpContent implements
LastHttpContent
```

注意：httpContent内部最重要的content属性是byteBuf

fullHttpMessage--结合httpMessage和httpContent

```
/**
 * Combines {@link HttpMessage} and {@link LastHttpContent} into one
 * message. So it represent a <i>complete</i> http message.
 */
public interface FullHttpMessage extends HttpMessage, LastHttpContent
```

下面有两个接口，为FullHttpRequest和FullHttpResponse

一般使用和拓展的是这两个接口的默认实现类

```
public class DefaultFullHttpRequest extends DefaultHttpRequest implements
FullHttpRequest
```

```
public class DefaultFullHttpResponse extends DefaultHttpResponse implements
FullHttpResponse
```

HttpObjectAggregator

先看这个组件的注释

```
/*
A ChannelHandler that aggregates an HttpMessage and its following HttpContents
into a single FullHttpRequest or FullHttpResponse (depending on if it used to
handle requests or responses) with no following HttpContents. It is useful when
you don't want to take care of HTTP messages whose transfer encoding is
'chunked'. Insert this handler after HttpResponseDecoder in the ChannelPipeline
if being used to handle responses, or after HttpRequestDecoder and
HttpResponseEncoder in the ChannelPipeline if being used to handle requests.
*/
```

可见这个组件的功能是将HttpMessage和这个消息的Content聚合到一个FullHttpMessage中，放在Http的decoder后面

本质调用的是MessageAggregator方法，将前面传过来的HttpMessage，HttpContent聚合为FullHttpMessage

WebSocket

WebSocket协议是怎么升级的?

- 客户端发送一个http/https的GET请求，请求头中包含如下内容，这个http/https请求不同点在于是以ws://或者wss://开头的，提示浏览器和客户端这是个websocket请求

```
GET ws://localhost:8080/ HTTP/1.1
Connection: Upgrade
Sec-WebSocket-key: 随机Base64字符窜
upgrade: websocket
Sec-WebSocket-Version: 13
```

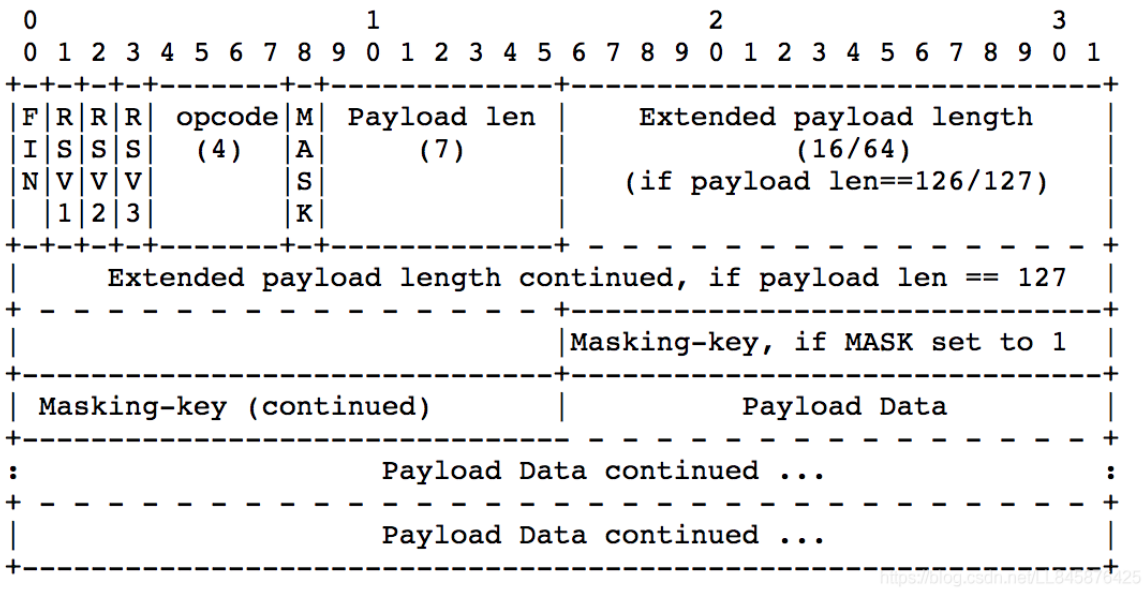
- 服务端响应101状态码，代表这是一个请求升级的响应，返回一个Sec-WebSocket-Accept，这个值是如下步骤算出来的
    - 客户端的Sec-WebSocket-key连接上魔法值258EAFA5-E914-47DA-95CA-C5AB0DC85B11，魔法值是固定的任何websocket都是一样的
    - 连接后的字符串，根据ASCII码得到字节数组，用sha1算法进行加密再得到一个字节数组
    - 最后将得到的字节数组取base64加密，返还给客户端，客户端接收后就说明websocket握手建立连接成功

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: cDtkcbGhK3SAHZv53UXMUl92hJU=
```

- 演示计算Sec-Websocket-Accept的值

```
public class TestWebsocketReceive extends
SimpleChannelInboundHandler<FullHttpRequest> {
    private boolean buildWebSocket = false;
    public final static String SEC_WEBSOCKET_ACCEPT_MAGIC_STRING = "258EAFA5-
E914-47DA-95CA-C5AB0DC85B11";
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, FullHttpRequest msg)
throws Exception {
        System.out.println(msg);
        if (!buildWebSocket) {
            DefaultFullHttpResponse response = new
DefaultFullHttpResponse(HttpVersion.HTTP_1_1,
HttpResponseStatus.SWITCHING_PROTOCOLS);
            response.headers().set("Upgrade", "websocket");
            response.headers().set("Connection", "Upgrade");
            String accept = getSecWebsocketAccept(msg.headers().get("Sec-
WebSocket-Key"));
            System.out.println(accept);
            response.headers().set("Sec-WebSocket-Accept", accept);
            ctx.channel().writeAndFlush(response);
            buildWebSocket = true;
        }
    }


    private String getSecWebsocketAccept(String template) {
        try {
            MessageDigest digest = MessageDigest.getInstance("sha1");
            // 连接魔法值，取字节数组
            byte[] bytes = (template +
SEC_WEBSOCKET_ACCEPT_MAGIC_STRING).getBytes(StandardCharsets.US_ASCII);
            // sha1加密
            byte[] temp = digest.digest(bytes);
            // base64加密
            return Base64.getEncoder().encodeToString(temp);
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        }
        return "";
    }
}
```

websocket消息结构



FIN，指明Frame是否是一个Message里最后Frame（之前说过一个Message可能又多个Frame组成）；1bit，是否为信息的最后一帧
RSV1-3，默认是0 (必须是0)，除非有扩展定义了非零值的意义。
Opcode，这个比较重要，有如下取值是被协议定义的

0x00 denotes a continuation frame
0x01 表示一个text frame
0x02 表示一个binary frame
0x03 ~~ 0x07 are reserved for further non-control frames,为将来的非控制消息片段保留测操作码
0x08 表示连接关闭
0x09 表示 ping (心跳检测相关)
0x0a 表示 pong (心跳检测相关)
0x0b ~~ 0x0f are reserved for further control frames,为将来的控制消息片段保留的操作码

Mask，这个是指明“payload data”是否被计算掩码。这个和后面的Masking-key有关，如果设置为1,掩码键必须放在masking-key区域，客户端发送给服务端的所有消息，此位的值都是1；
Payload len，数据的长度，
Masking-key，0或者4bit，只有当MASK设置为1时才有效。，给一个Websocket中掩码的意义
Payload data，帧真正要发送的数据，可以是任意长度，但尽管理论上帧的大小没有限制，但发送的数据不能太大，否则会导致无法高效利用网络带宽，正如上面所说Websocket提供分片。
Extension data：扩展数据，如果客户端和服务端没有特殊的约定，那么扩展数据长度始终为0
Application data：应用数据，

netty中的WebSocket

WebSocketServerHandshakerFactory和WebSocketServerHandshaker

创建一个WebSocketServerHandshakerFactory，调用factory.newHandshaker(request)对传来的httpRequest处理根据请求头中的WebSocket版本号返回一个对应的WebSocketServerHandshaker，调用WebSocketServerHandshaker上的webSocketServerHandshaker.handshake(ctx.channel(), request)方法，对pipeline做处理，这个方法会先向客户端发送一个升级响应头，随后删除pipeline上的HttpObjectAggregator，HttpContentCompressor，HttpRequestDecoder，HttpServerCodec，HttpResponseEncoder，再加上对应版本的WebSocketFrameEncoder和WebSocketFrameDecoder

handshake方法如下面所示

```
public final ChannelFuture handshake(Channel channel, FullHttpRequest req,
                                     HttpHeaders responseHeaders, final
ChannelPromise promise) {

    if (logger.isDebugEnabled()) {
        logger.debug("{} WebSocket version {} server handshake", channel,
version());
    }
    FullHttpResponse response = newHandshakeResponse(req, responseHeaders);
    ChannelPipeline p = channel.pipeline();
    if (p.get(HttpObjectAggregator.class) != null) {
        p.remove(HttpObjectAggregator.class);
    }
    if (p.get(HttpContentCompressor.class) != null) {
        p.remove(HttpContentCompressor.class);
    }
    ChannelHandlerContext ctx = p.context(HttpRequestDecoder.class);
    final String encoderName;
    if (ctx == null) {
        // this means the user use an HttpServerCodec
        ctx = p.context(HttpServerCodec.class);
        if (ctx == null) {
            promise.setFailure(
                    new IllegalStateException("No HttpDecoder and no
HttpServerCodec in the pipeline"));
            return promise;
        }
        p.addBefore(ctx.name(), "wsencoder", newWebSocketEncoder());
        p.addBefore(ctx.name(), "wsdecoder", newWebsocketDecoder());
        encoderName = ctx.name();
    } else {
        p.replace(ctx.name(), "wsdecoder", newWebsocketDecoder());

        encoderName = p.context(HttpResponseEncoder.class).name();
        p.addBefore(encoderName, "wsencoder", newWebSocketEncoder());
    }
    channel.writeAndFlush(response).addListener(new ChannelFutureListener() {
        @Override
        public void operationComplete(ChannelFuture future) throws Exception {
            if (future.isSuccess()) {
                ChannelPipeline p = future.channel().pipeline();
                p.remove(encoderName);
                promise.setSuccess();
            } else {
                promise.setFailure(future.cause());
            }
        }
    });
    return promise;
}
```

使用例

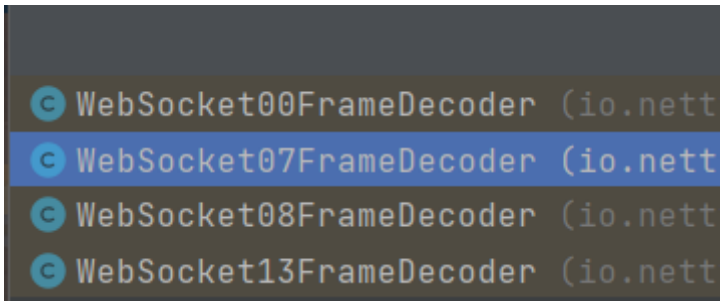
```
WebSocketServerHandshakerFactory factory = new
WebSocketServerHandshakerFactory(request.uri(), null, false, 65536, false);
WebSocketServerHandshaker webSocketServerHandshaker =
factory.newHandshaker(request);
// 如果是不支持的协议会返回null，这里最好接一个判断null的if
webSocketServerHandshaker.handshake(ctx.channel(), request);
```

WebSocketServerHandshaker

利用了netty流水线上handler热插拔的原理，统一了每个webSocket的管理

WebSocketFrame的Decoder和Encoder

- 总共有4个版本，直接来看13版本的实现



- 13版本继承了8版本，并且所有构造函数都是调用的8版本的构造函数

```
public class WebSocket13FrameDecoder extends WebSocket08FrameDecoder
```

```
public WebSocket13FrameDecoder(WebSocketDecoderConfig decoderConfig) {
    super(decoderConfig);
}
```

- 具体源码就不看了主要看decode和encode处理的类型是什么

```
public class WebSocket08FrameEncoder extends
MessageToMessageEncoder<WebSocketFrame> implements WebSocketFrameEncoder
```

从下面的代码中可以看到，webSocket的decode大概率同样也需要聚合器

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

    ......
                if (frameOpcode == OPCODE_PING) {
                    out.add(new PingWebSocketFrame(frameFinalFlag, frameRsv,
payloadBuffer));
                    payloadBuffer = null;
                    return;
                }
                if (frameOpcode == OPCODE_PONG) {
                    out.add(new PongWebSocketFrame(frameFinalFlag, frameRsv,
payloadBuffer));
```

```
                    payloadBuffer = null;
                    return;
                }
                if (frameOpcode == OPCODE_CLOSE) {
                    receivedClosingHandshake = true;
                    checkCloseFrameBody(ctx, payloadBuffer);
                    out.add(new CloseWebSocketFrame(frameFinalFlag, frameRsv,
payloadBuffer));
                    payloadBuffer = null;
                    return;
                }
    ......
                if (frameOpcode == OPCODE_TEXT) {
                    out.add(new TextWebSocketFrame(frameFinalFlag, frameRsv,
payloadBuffer));
                    payloadBuffer = null;
                    return;
                } else if (frameOpcode == OPCODE_BINARY) {
                    out.add(new BinaryWebSocketFrame(frameFinalFlag, frameRsv,
payloadBuffer));
                    payloadBuffer = null;
                    return;
                } else if (frameOpcode == OPCODE_CONT) {
                    out.add(new ContinuationWebSocketFrame(frameFinalFlag,
frameRsv,
                                                           payloadBuffer));
                    payloadBuffer = null;
                    return;
                } else {
     ......

}
```

WebSocketFrameAggregator聚合器

查看这个类的继承结构

```
public class WebSocketFrameAggregator
        extends MessageAggregator<WebSocketFrame, WebSocketFrame,
ContinuationWebSocketFrame, WebSocketFrame> {
```

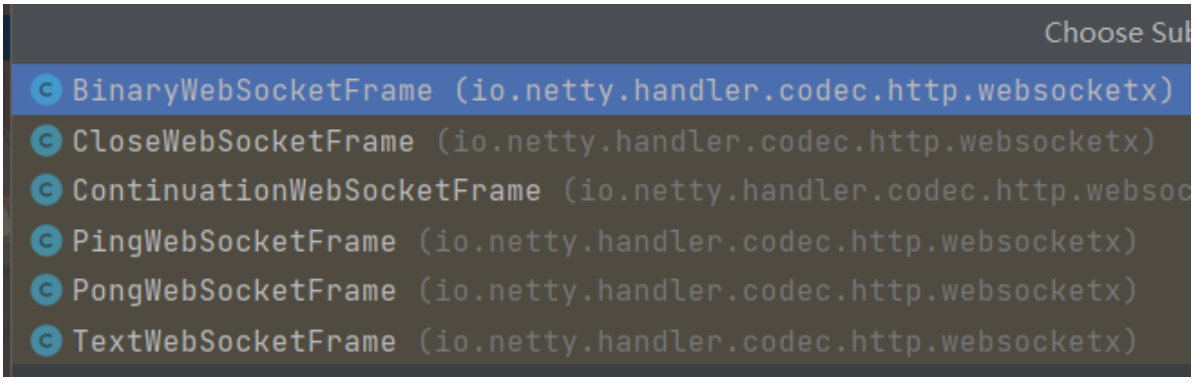
可见这个类是把WebSocketFrame和多个ContinuationWebSocketFrame聚合成了一个大的WebSocketFrame，注意这几个方法判断，说明传输应该是 Text/BinaryWebSocketFrame -> ContinuationWebSocketFrame -> ... -> ContinuationWebSocketFrame(isFinalFragment = true)

```
protected boolean isStartMessage(WebSocketFrame msg) throws Exception {
    return msg instanceof TextWebSocketFrame || msg instanceof
BinaryWebSocketFrame;
}

@Override
protected boolean isContentMessage(WebSocketFrame msg) throws Exception {
    return msg instanceof ContinuationWebSocketFrame;
}

@Override
protected boolean isLastContentMessage(ContinuationWebSocketFrame msg) throws
Exception {
    return isContentMessage(msg) && msg.isFinalFragment();
}
```

这个聚合器要手动添加，一般用在不只一个frame的大请求中

```
private void handleHttpUpdateToWebSocket(ChannelHandlerContext ctx, HttpRequest
request) {
    WebSocketServerHandshakerFactory factory = new
WebSocketServerHandshakerFactory(request.uri(), null, false, 65536, false);
    WebSocketServerHandshaker webSocketServerHandshaker =
factory.newHandshaker(request);
    webSocketServerHandshaker.handshake(ctx.channel(), request);
    String webSocketDecoderContextName =
ctx.pipeline().context(WebSocketFrameDecoder.class).name();
    ctx.pipeline().addAfter(webSocketDecoderContextName,"webSocketAggregator",
new WebSocketFrameAggregator(65536));
}
```

WebSocketFrame的下面的类



BinaryWebSocketFrame

用来传输二进制数据

CloseWebSocketFrame

告诉对方即将关闭通信，客户端和服务端都可以发送

ContinuationWebSocketFrame

用来传递分片的文本或者二进制数据

TextWebSocketFrame

文本数据帧

Ping/PongWebSocketFrame

用于Ping/Pong维护长链接

> webSocket的ping pong响应是一种用于维持长连接和检测网络状态的机制。它是WebSocket协议中的两种控制帧，分别为ping帧和pong帧，操作码为0x09和0x0A。服务器可以发送ping帧给浏览器，浏览器收到后必须返回pong帧。如果在一定时间内没有收到对方的响应，就认为连接断开了。