# 1 基础

## 1.1 ByteBuf

### 1.1.1 对比 ByteBuffer

**NIO 中的 ByteBuffer**

- 固定长度
- 读写公用一套指针
- 功能有限

**Netty 中的 ByteBuf**

### 1.1.2 特点

#### 1.1.2.1 便捷的读写操作

通过2个位置指针来协助缓冲区的读写，读使用 readerIndex，写使用 writerIndex。

#### 1.1.2.2 动态扩展缓冲区

put 时，效验剩余空间，当容量超过限制后，通过 System.arraycopy 方法来对数组进行扩容操作，重建一个新的 ByteBuf，并将之前的 ByteBuf 复制到新的 ByteBuf 中。

#### 1.1.2.3 引用计数

使用 volcatile 记录引用次数，使用原子对象类型 AtomicIntegerFieldUpdater 来对其进行更新。

> - 不直接使用 AtomicInteger 的原因 这是性能考虑的极致，因为 **AtomicIntegerFieldUpdater** 是静态修饰的，只有1个对象， 记录引用的是int变量，属于栈中的一个变量，而如果使用 AtomicInteger 替换，则会多出 很多堆内存的使用空间。
> - 原理 通过自旋模式，使用 AtomicIntegerFieldUpdater 对引用计数变量进行更新

### 1.1.3 类别

**Heap Buffer（堆缓冲区）**

**Direct Buffer（堆外缓冲区）**

**Composite Buffer（复合缓冲区）**

复合缓冲区表示一部分是堆缓冲区，一部分是堆外缓冲区

```
// 堆缓冲区
ByteBuf heapBuf = Unpooled.buffer(8);
// 堆外缓冲区
ByteBuf directBuf = Unpooled.drectBuffer(16);
// 复合缓冲区
CompositeByteBuf compBuf = Unpooled.compositeBuffer();
// 将堆和堆外缓冲区都添加到复合缓冲区中
compBuf.addComponents(heapBuf, directBuf);
// 删除堆缓冲区
compBuf.removeComponent(0);
// 输出
Iterator<ByteBuf> iter = compBuf.iterator();
while(iter.hasNext()) {
```

```
    System.out.println(iter.next().toString());
}
```

### 1.1.4 创建

```
// 堆缓冲
ByteBuf heapBuf = Unpooled.buffer(8);
// 堆外缓冲
ByteBuf directBuf = Unpooled.directBuffer(16);
// 复合缓冲
CompositeByteBuf compBuf = Unpooled.compositeBuffer();
```

### 1.1.5 PooledDirectByteBuf

基于内存池实现，提前申请一块内存，当需要ByteBuf的时候，就从中申请一片内存。与UnPooledDirectByteBuf基本一致，唯一不同就是内存分配策略。

```
static PooledDirectByteBuf newInstance(int maxCapacity) {
    PooledDirectByteBuf buf = RECYCLER.get();
    buf.reuse(maxCapacity);
    return buf;
}
```

### 1.1.6 注意点

- 通过索引访问不会推进读写的标记

  ```
  ByteBuf buf = Unpooled.buffer(16);
  for (int i = 0; i < 16; i++) {
      buf.writeByte(i + 1);
  }
  // read
  for (int i = 0; i < buf.capacity(); i++) {
      System.out.println(buf.getByte(i));
  }
  ```

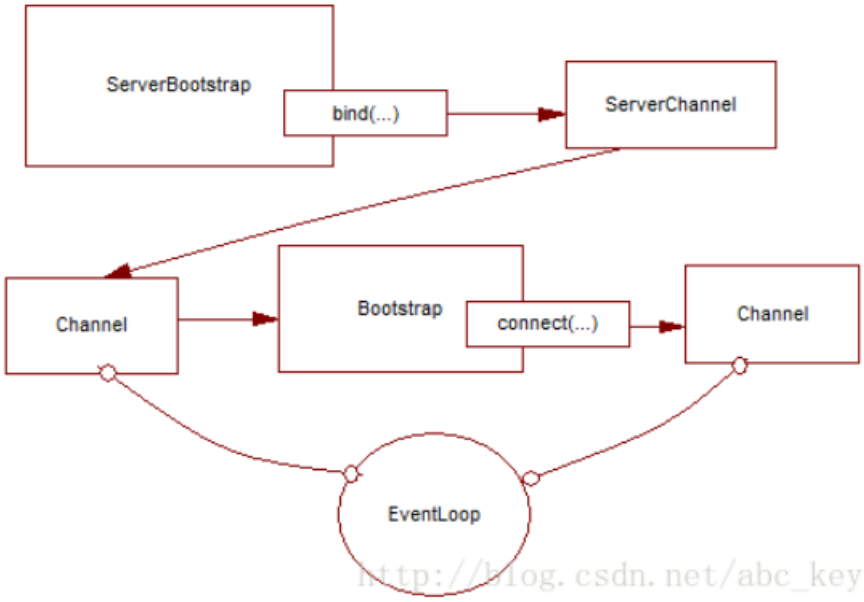
- **readableBytes 方法并不代表发送端发送的可读字节，而是套接字缓冲区中当前存在的字节大小，如果是实时的流传输， 该值是会不断变化的。只有增加了粘包拆包功能后，才能保证readableBytes读取到合适的字节。**

```
private static class InitializerHandler extends
ChannelInitializer<SocketChannel> {
    @Override
    protected void initChannel(SocketChannel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        // 长度适配，会先将缓冲区中的数据控制住
        pipeline.addLast(new LengthFieldBasedFrameDecoder(Integer.MAX_VALUE,
0, 2, 0,2));
        // 对象解码，根据readableBytes放心从缓冲区中读取
        pipeline.addLast(new MessagePackDecoder());
        pipeline.addLast(new LengthFieldPrepender(2));
        pipeline.addLast(new MessagePackEncoder());
        pipeline.addLast(new ServerHandler());
    }
}
```

# 1.2 EventLoop

## 1.2.1 EventLoop

- 不断等待事件发生的一个死循环，是 Channel 执行实际工作的线程，总是绑定一个单一的线程，在其生命周期内不会改变。
- 一个 EventLoop 由一个线程执行，共享 EventLoop 可以确定所有的 Channel 都分配给同一线程的 EventLoop，避免不同线程之间切换的上下文，减少资源开销





## 1.2.2 EventLoopGroup

事件循环集合，并在此期间将 Channel 注册到 Selector 上。

- 服务端一般分为 boosGroup 和 workerGroup
    - bossGroup 接收连接（转发），通常使用1个线程，否则就是内核 * 2；
    - workerGroup 实际处理业务，事件循环组，底层死循环，不停侦测事件；

# 1.3 NioEventLoop

## 1.3.1 NioEventLoop

基于 NIO Selector 实现

### 1.3.2 NioEventLoopGroup

基于 NIO Selector 实现的组

（获取一个 Selector，一般使用 SelectorProvider 的 provider 方法，内部可以看到调用了 SelectorProvider.provider() ）

## 1.4 EventExecutor

### 1.4.1 EventExecutor

类似一个事件线程池

### 1.4.2 EventExecutorGroup

负责提供 EventExecutor，控制生命周期，以及全局的状态

## 1.5 Bootstrap

### 1.5.1 Bootstrap

用于快捷启动客户端通道的类

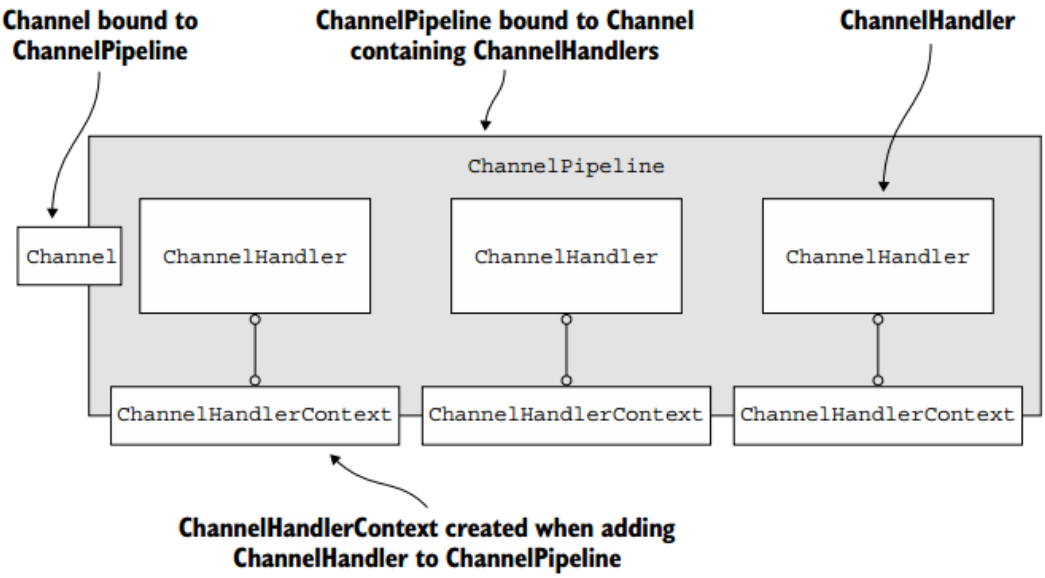
- group(...)：设置 EventLoopGroup 用来处理所有通道的IO事件
- channel(...)：设置通道类型
- localAddress(...)：设置本地地址，也可以通过bind(...)或connect(...)
- option(ChannelOption< T >, T)：设置通道选项
- attr(AttributeKey< T >, T)：设置属性到 Channel
- handler(ChannelHandler ch)：设置 ChannelHandler 用于处理请求事件
- removeAddress(...)：设置连接地址
- connect(...)：创建一个新的 Channel 绑定，连接远程通道
- bind(...)：创建一个新的 Channel 绑定，连接远程通道

### 1.5.2 ServerBootstrap

构造类似 Bootstrap，用于快捷启动服务端通道的类。 通过简单的配置来设置或“引导”程序的一个重要的类。

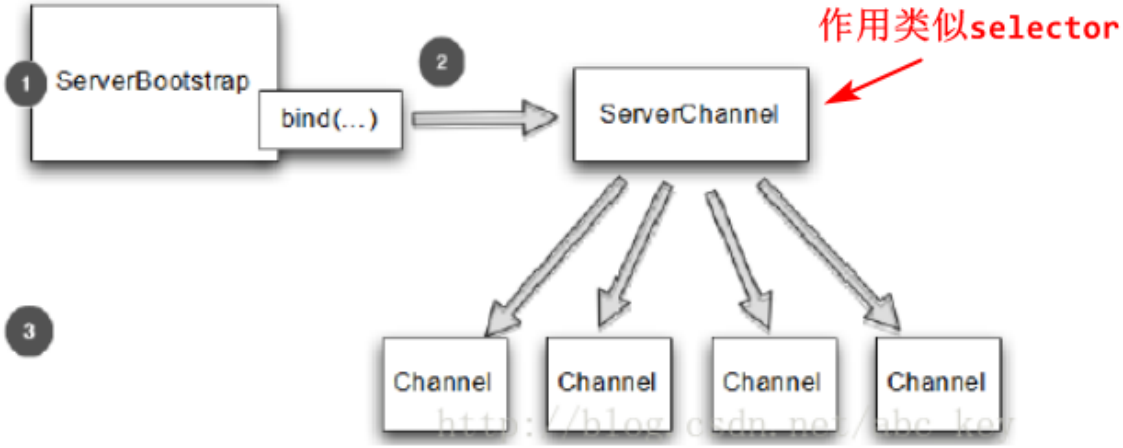
## 1.6 Channel

结构




### 1.6.1 ServerChannel

标记接口，会接收对端发过来的请求，并创建真正的与客户端连接的 child Channel；




### 1.6.2 ServerSocketChannel

TCP/IP 的 ServerChannel

## 1.7 ChannelPipeline

### 1.7.1 描述

是 ChannelHandler 实例的列表，用于处理或截获通道的接收和发送数据

### 1.7.2 过程

- 入栈时，先执行最先添加的 InboundHandler，出栈时，先执行最后添加的 OutboundHandler
- 上一个Handler执行完了，才会执行下一个Handler（很关键的概念）

> 比如：粘包拆包的处理，编解码之前，一般会放上，半包处理器 LengthFieldBasedFrameDecoder ， 这样就可以很方便的控制当前 Socket 缓冲区中的数据，半包处理器总是会根据用户设置，控制一个 完整的数据缓冲区，供开发者操作，**就像一个阀门一样。**

### 1.7.3 常用

addLast(ChannelHandler handler)：表示在 Pipeline 末尾添加 ChannelHandler

### 1.7.4 通道的 Option

ChannelOption（选项）来帮助引导配置，可用各种选项配置底层连接详细信息，比如：keep-alive（保持活跃），timeout（超时时间）等

### 1.7.5 通道的 Attributes

Attributes（属性），传递一些属性，只能本机上传递，并不能相互传递，比如：将用户信息与通道关联起来

示例：

```
// 创建属键对象
final AttributeKey<Integer> id = AttributeKey.valueOf("ID");
// 客户端引导对象
Bootstrap b = new Bootstrap();
// 设置EventLoop，设置通道类型
b.group(new NioEventLoopGroup()).channel(NioSocketChannel.class)
// 设置ChannelHandler
    .handler(new SimpleChannelInboundHandler<ByteBuf>() {
        @Override
        protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg)
            throws Exception {
            System.out.println("Reveived data");
            msg.clear();
        }
        @Override
        public void channelRegistered(ChannelHandlerContext ctx)
            throws Exception {
            // 通道注册后执行，获取属性值
            // 注意：如果在服务端获取，是获取不到该ID的值的
            Integer idValue = ctx.channel().attr(id).get();
            System.out.println(idValue);
        }
    });
// 设置通道Option
b.option(ChannelOption.SO_KEEPALIVE,
true).option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 5000);
// 设置通道属性
b.attr(id, 123456);
ChannelFuture f = b.connect("localhost", 8080);
f.synUninterruptibly();
```

## 1.8 ChannelHandlerContext

### 1.8.1 描述

- 每个 ChannelHandler 被添加到 ChannelPipeline 后，都会创建一个 ChannelHandlerContext，并与之绑定

- 一般创建一个 Client，就应该有一套 Pipeline，也就是说每个客户端 Channel 对应一组 ChannelHandlerContext
- ChannelHandlerContext 允许 ChannelHandler 与其他的 ChannelHandler 实现进行交互

### 1.8.2 执行全部

- 调用 Channel 的方法
- 调用 ChannelPipeline 的方法

```
// 在任意的ChannelHandler的事件方法中调用
// 使用Channel
Channel channel = ctx.channel();
channel.write(Unpooled.copiedBuffer("Hello", CharsetUtil.UTF_8));
// 使用ChannelPipeline
ChannelPipeline pipeline = ctx.pipeline();
pipeline.write(Unpooled.copiedBuffer("World", CharsetUtil.UTF_8));
```

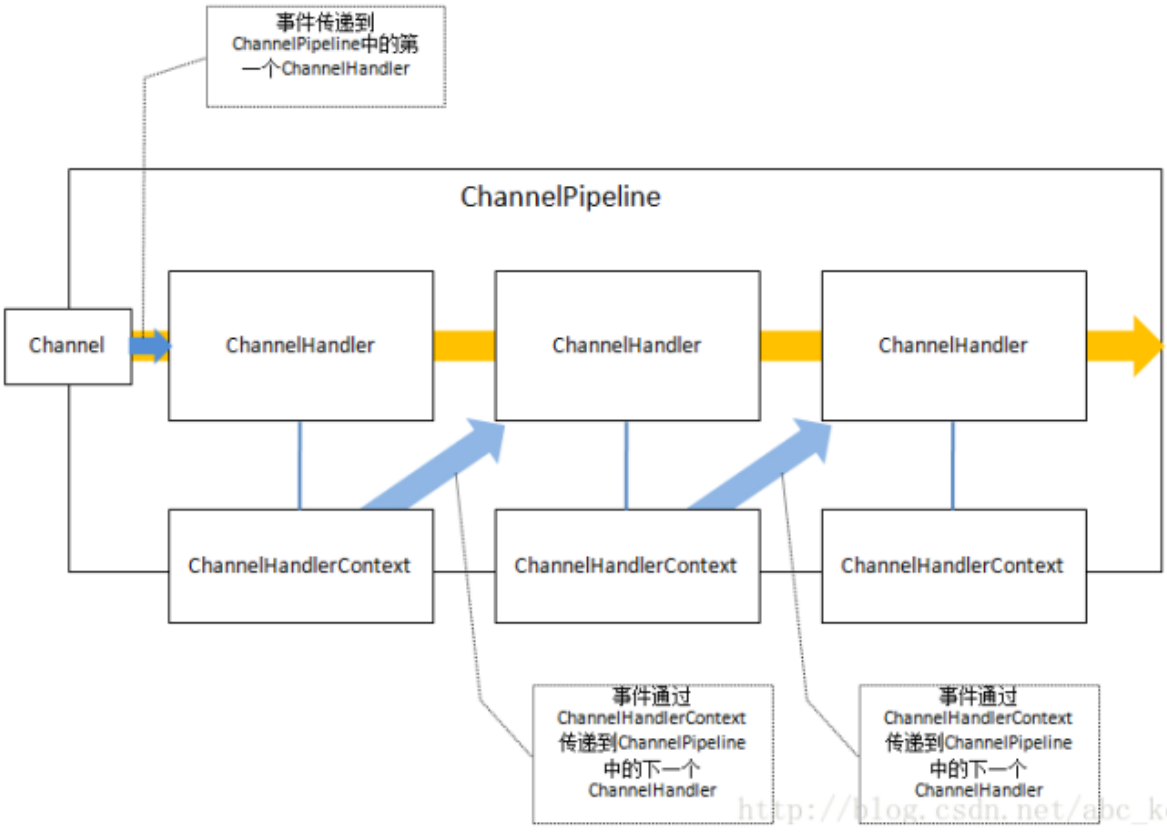
### 1.8.3 执行部分

- 调用 ChannelHandlerContext

```
// 在指定的ChannelHandler下重写事件方法，并调用ChannelHandlerContext执行操作
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) {
    ctx.write(Unpooled.copiedBuffer("Hello", CharsetUtil.UTF_8));
    ctx.flush();
}
```

### 1.8.4 流程图

线性流程



### 1.8.5 @Sharable

ChannelHandler 如果带有 @Sharable 注解，则可以被添加到多个 ChannelPipeline 中， 意味着单个 ChannelHandler 实例可以有多个 ChannelHandlerContext，此时需要注意 线程安全的问题。

### 1.8.6 实现

#### ChannelInboundHandler

入站执行的Handler

| 回调方法 | 触发时机 | client | server |
| --- | --- | --- | --- |
| bind | bind操作执行前触发 | false | true |
| connect | connect 操作执行前触发 | true | false |
| disconnect | disconnect 操作执行前触发 | true | false |
| close | close操作执行前触发 | false | true |
| deregister | deregister操作执行前触发 | | |
| read | read操作执行前触发 | true | true |
| write | write操作执行前触发 | true | true |
| flush | flush操作执行前触发 | true | true |

#### InboundHandler 实现

- ChannelInboundHandlerAdapter

  需要手动调用 ReferenceCountUtil.release(msg) 释放消息

- SimpleChannelInboundHandler

  会自动释放消息

#### ChannelOutboundHandler

出站执行的Handler

| 回调方法 | 触发时机 | client | server |
| --- | --- | --- | --- |
| channelRegistered | 当前channel注册到EventLoop | true | true |
| channelUnregistered | 当前channel从EventLoop取消注册 | true | true |
| channelActive | 当前channel激活的时候 | true | true |
| channelInactive | 当前channel不活跃的时候，也就是当前channel到了它生命周期末 | true | true |
| channelRead | 当前channel从远端读取到数据 | true | true |
| channelReadComplete | channel read消费完读取的数据的时候被触发 | true | true |
| userEventTriggered | 用户事件触发的时候 | | |
| channelWritabilityChanged | channel的写状态变化的时候触发 | | |

### 1.8.7 ChannelInitializer

- ChannelInitializer 抽象类用来初始化 ChannelPiepline 中的 ChannelHandler。

- 通道被注册到 EventLoop 后就会调用 ChannelInitializer，完成初始化之后，会自动删除

# 2 编解码器

## 2.1 解码器

### 2.1.1 分类

- 字节 >>> 消息
- 消息 >>> 消息
- 消息 >>> 字节

### 2.1.2 实现

#### 2.1.2.1 ByteToMessageDecoder

用于字节解码成消息，或字节解码成其他序列化字节，常用于将字节消息解码成POJO对象。

- decode(...)

  参数：`ChannelHandlerContext ctx, ByteBuf in, List<Object> out`

  将 ByteBuf 数据解码成其他形式的数据

  > 例如：客户端接收到一个整型的字节码，服务器将数据读入 ByteBUf 并经过 ChannelPipeline 中的每个 Handler 进行处理

```
public class ToIntegerDecoder extends ByteToMessageDecoder {
 @Override
 protected void decode(ChannelHandlerContext ctx, ByteBuf in,
                       List<Object> out) throws Exception {
     if (in.readableBytes() >= 4) {
         out.add(in.readInt());
     }
 }
}
```

#### 2.1.2.2 ReplayingDecoder< S >

ReplayingDecoder 是 byte-to-message 解码的一种特殊的抽象基类，使用 ReplayingDecoder **无需检查缓冲区**是否有 足够的字节。**若字节足够，则正常读取；若没有足够的字节则会停止解码（这也是局限性）。**

```
public class ToIntegerDecoder extends ByteToMessageDecoder {
    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in,
                          List<Object> out) throws Exception {
        // 不需要判断是否有足够字节
        out.add(in.readInt());
    }
}
```

#### 2.1.2.3 MessageToMessageDecoder< I >

用于将消息对象转换成消息对象

- decode(...)

  参数：`ChannelHandlerContext ctx, I msg, List<Object> out`

```
// 将接受的Integer消息转成String类型
public class IntegerToStringDecoder extends MessageToMessageDecoder<Integer>
{
    @Override
    protected void decode(ChannelHandlerContext ctx, Integer msg,
                          List<Object> out) throws Exception {
        out.add(String.valueOf(msg));
    }
}
```

## 2.2 编码器

### 2.2.1 分类

- 消息对象 >>> 消息对象
- 消息对象 >>> 字节码

### 2.2.2 实现

#### 2.2.2.1 MessageToByteEncoder< I >

将处理好的数据从转成字节码，以便在网络中传输。

- encode(...)

  参数：`ChannelHandlerContext ctx, I msg, ByteBuf out`

```
// 将Integer值编码成byte[]
public class IntegerToByteEncoder extends MessageToByteEncoder<Integer> {
    @Override
    protected void encode(ChannelHandlerContext ctx, Integer msg,
                          ByteBuf out) throws Exception {
        out.writeInt(msg);
    }
}
```

#### 2.2.2.2 MessageToMessageEncoder< I >

消息编码成其他消息

- encode(...)

  参数：`ChannelHandlerContext ctx, I msg, List<Object> out`

```
// 将Integer值编码成String
public class IntegerToStringEncoder extends MessageToMessageEncoder<Integer>
{
    @Override
    protected void encode(ChannelHandlerContext ctx, Integer msg,
                          List<Object> out) throws Exception {
        out.add(String.valueOf(msg));
    }
}
```

## 2.3 编解码

### 2.3.1 ByteToMessageCodec< I >

- encode(...)

  参数：`ChannelHandlerContext ctx, I msg, ByteBuf out`

- decode(...)

  参数：`ChannelHandlerContext ctx, ByteBuf in, List<Object> out`

### 2.3.2 MessageToMessageCodec

- encode(...)

  参数：`ChannelHandlerContext ctx, OUTBOUND_IN msg, List<Object> out`

- decode(...)

  参数：`ChannelHandlerContext ctx, INBOUND_IN msg, List<Object> out`

## 2.4 特殊编解码

### 2.4.1 CombinedChannelDuplexHandler

结合编码器和解码器

- 解码器

```
// 解码器，将byte转成char
public class ByteToCharDecoder extends ByteToMessageDecoder {
    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in,
                          List<Object> out) throws Exception {
        while (in.readableBytes() >=2 ) {
            out.add(Character.valueOf(in.readChar()));
        }
    }
}
```

- 编码器

```
// 编码器，将char转成byte
public class CharToByteEncoder extends MessageToByteEncoder<Character> {
    @Override
    protected void encode(ChannelHandlerContext ctx, Character msg,
                          ByteBuf out) throws Exception {
        out.writeChar(msg);
    }
}
```

- 结合器

```
// 继承CombinedChannelDuplexHandler,绑定解码器和编码器
public class CharCodec extends 继承
CombinedChannelDuplexHandler<ByteToCharDecoder, CharToByteEncoder> {
    public CharCodec() {
        super(new ByteToCharDecoder(), new CharToByteEncoder());
    }
}
```

# 3 粘包 / 拆包

## 3.1 描述

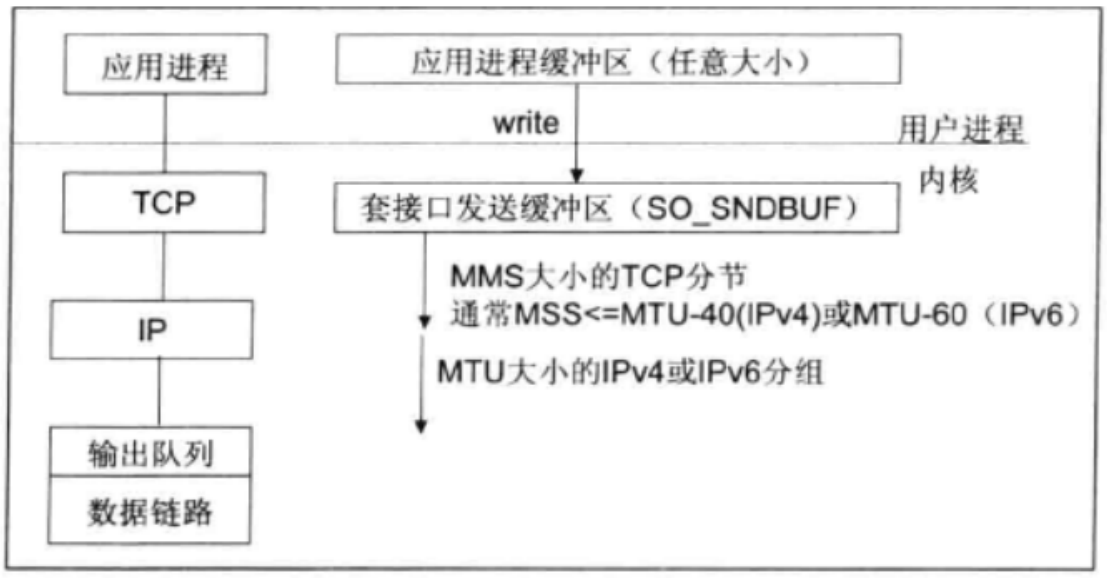
TCP 是个 “流” 协议，就是没有界限的一串数据。TCP 底层并不了解上层业务数据的具体含义， 他会根据 TCP 缓冲区的实际情况进行包的划分，所以在业务上人为，一个完整的包可能会被 TCP 拆分成多个包进行发送，也有可能把多个小包封装成一个大的数据包进行发送。

## 3.2 产生原因

- 接收端的缓冲区大小与发送端的数据不一致

```
// 接收端每次read的缓冲字节是可控的，本质还是流
byte[] bytes = new byte[5];
in.readBytes(bytes);
```

- MSS TCP协议的概念，表示每次TCP传输最大的数据分段；
- MTU 硬件规定的以太网最大传输单元（以太帧的 payload 大于 MTU 进行 IP 分片）；




## 3.3 解决策略

> 由于底层的 TCP 无法理解上层业务数据，所以只能通过协议设计来解决（类似前后端公约）；

- **消息定长**

  例如：每个报文的大小固定长度 200 字节，如果不够，空位补空格；

- **自定义分隔符**

  在包尾增加回车换行符进行分割。例如：FTP 协议；

- **划分区域记录长度**

  将消息分为消息头和消息体，消息头中包含表示消息总长度的字段， 通常涉及思路为消息头的第一个字段用 int32 来表示消息的总长度

## 3.4 实现

### 3.4.1 LineBasedFrameDecoder

- 工作原理是遍历 ByteBuf 中可读字节，**判断是否有 “\n” 或 "\r\n"**， 如果有，就此为止为结束为止。（即以换行符为结束标志的几码器）
- 支持配置单行的最大长度，如果连续读取到最大长度后，仍没有 发现换行符，则抛出异常；
- 如果没有发现换行符，则将数据存入缓存，直到下次数据过来；

```
ChannelPipeline p = ch.pipeline();
p.addLast(new LineBasedFrameDecoder(1024));
```

### 3.4.2 DelimiterBasedFrameDecoder

- 自定义结束标记
- 如果连续读取到最大长度后，仍没有发现标志，则抛出异常；

```
ChannelPipeline p = ch.pipeline();
p.addLast(new DelimiterBasedFrameDecoder(1024,
        Unpooled.copiedBuffer("$".getBytes())));
```

### 3.4.3 FixedLengthFrameDecoder

- 定长消息解码（皆可作用于收发操作）
- **如果是半包消息，则会缓存半包消息，并等待下一个包达到后进行拼包，直到读取一个完整的包；**

```
ChannelPipeline p = ch.pipeline();
p.addLast(new FixedLengthFrameDecoder(5));
...
```

### 3.4.4 LengthFieldBasedFrameDecoder

与 LengthFieldPrepender 配合使用，常用设置如下（最好需要自己根据文档配置，非常灵活）

```
// MAX,0,2,0,2
pipeline.addLast(new LengthFieldBasedFrameDecoder(Integer.MAX_VALUE, 0, 2,
0,2));
pipeline.addLast(new MessagePackDecoder());
// 2
pipeline.addLast(new LengthFieldPrepender(2));
pipeline.addLast(new MessagePackEncoder());
```

# 4 序列化

## 4.1 常用

- JDK 的序列化 私有协议，并不能跨语言，并且序列化后码流太大

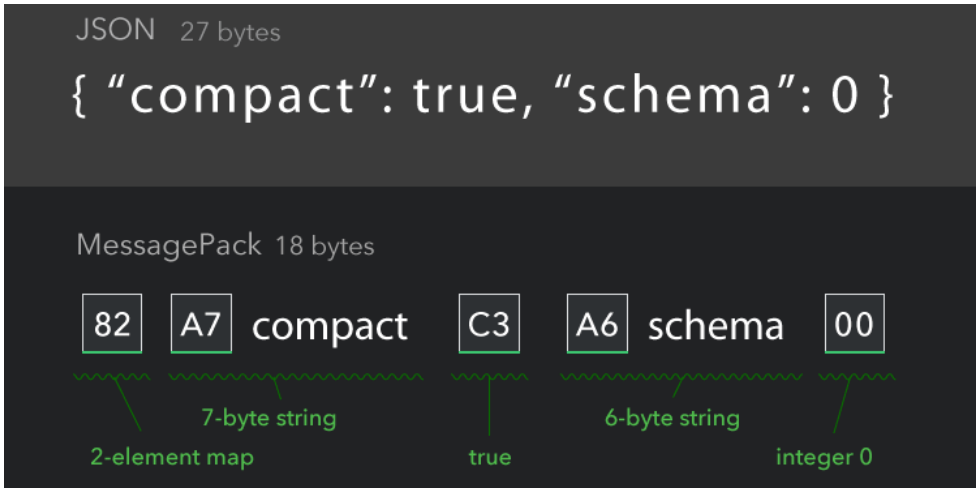
- Protobuf 结构化数据存储格式（XML，JSON 等）；编解码性能高；跨语言
- Thrift 功能强大，自带 Server 与 Client 的 TCP 代码

## 4.2 推荐

### 4.2.1 MessagePack

#### 4.2.1.1 描述

存储结构类似于 JSON，但是性能比 JSON 更好




#### 4.2.1.2 示例

- 简单示例

```
// Create serialize objects.
List<String> src = new ArrayList<>();
src.add("msgpack");
src.add("kumofs");
src.add("viver");
MessagePack msgpack = new MessagePack();
// Serialize
byte[] raw = msgpack.write(src);
System.err.println(Arrays.toString(raw));
// Deserialize directly using a template
List<String> dst1 = msgpack.read(raw, Templates.tList(Templates.TString));
System.out.println(dst1.get(0));
System.out.println(dst1.get(1));
System.out.println(dst1.get(2));
// Or, Deserialze to Value then convert type.
Value dynamic = msgpack.read(raw);
List<String> dst2 = new Converter(dynamic)
    .read(Templates.tList(Templates.TString));
System.out.println(dst2.get(0));
System.out.println(dst2.get(1));
System.out.println(dst2.get(2));
```

- 复杂示例
  - POJO

```
// 必须要加@Message注解
@Message
public class UserInfo implements Serializable {
```

```
    private String name;
    private Integer age;

    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() {
        final StringBuilder sb = new StringBuilder("{");
        sb.append("\"name\":\"")
                .append(name).append('\"');
        sb.append(",\"age\":")
                .append(age);
        sb.append('}');
        return sb.toString();
    }
}
```

- Encoder

```
public class MessagePackEncoder extends MessageToByteEncoder<UserInfo>
{
    @Override
    protected void encode(ChannelHandlerContext ctx, UserInfo userInfo,
ByteBuf out) throws Exception {
        MessagePack messagePack = new MessagePack();
        byte[] raw = messagePack.write(userInfo);
        out.writeBytes(raw);
    }
}
```

- Decoder

```
public class MessagePackDecoder extends ByteToMessageDecoder {
    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in,
List<Object> out)
        throws Exception {
        byte[] bytes = new byte[in.readableBytes()];
        // 方法1
        in.readBytes(bytes);
        // 方法2
```

```
            // in.getBytes(in.readerIndex(), bytes, 0, length);
            MessagePack messagePack = new MessagePack();
            UserInfo value = messagePack.read(bytes, UserInfo.class);
            System.err.println(value);
        }
    }
```

#### 4.2.1.3 总结

- 需要序列化的 POJO 对象上必须加上 org.msgpack.annotation.Message 注解：@Message
- MessagePack序列化只会取一次对象，多的字节会丢弃

### 4.2.2 Protobuf

Netty 封装了 Protobuf 的编解码，非常方便

```
public class TestServerInitializer extends ChannelInitializer<SocketChannel> {
    @Override
    protected void initChannel(SocketChannel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        // 以下4个
        pipeline.addLast(new ProtobufVarint32FrameDecoder());
        pipeline.addLast(new
ProtobufDecoder(MyDataInfo.MyMessage.getDefaultInstance()));
        pipeline.addLast(new ProtobufVarint32LengthFieldPrepender());
        pipeline.addLast(new ProtobufEncoder());
        pipeline.addLast(new TestServerHandler());
    }
}
```

# 5 WebSocket 协议开发

## 5.1 特点

- 全双工模式通信
- 对代理、防火墙和路由器透明
- 无Cookie 和 身份验证
- 无安全开销
- 服务器可以主动传递消息给客户端，不需要轮询

> WebSocket 是一个协议，而 Socket 是一个套接字的技术栈

## 5.2 建立连接

- WebSocket 客户端握手请求消息

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGh1IHNDFLUGLSJDF==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

- WebSocket 服务端返回的握手应答消息

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

> - 浏览器附加头信息 “Upgrade:WebSocket"，表明该次请求需要升级协议。
> - Sec-WebSocket-Key 是随机的，服务器端会用这些数据来构造出一个 SHA-1 的信息摘要，再加上一个魔幻字符串。 使用 SHA-1 加密，然后进行 BASE-64 编码，将结果作为 "Sec-WebSocket-Accept" 头的值，返回给客户端。

## 5.3 Netty 整合

Netty 基于 HTTP 协议栈开发了 WebSocket 协议栈，可以很方便的开发 WebSocket 客户端和服务端。

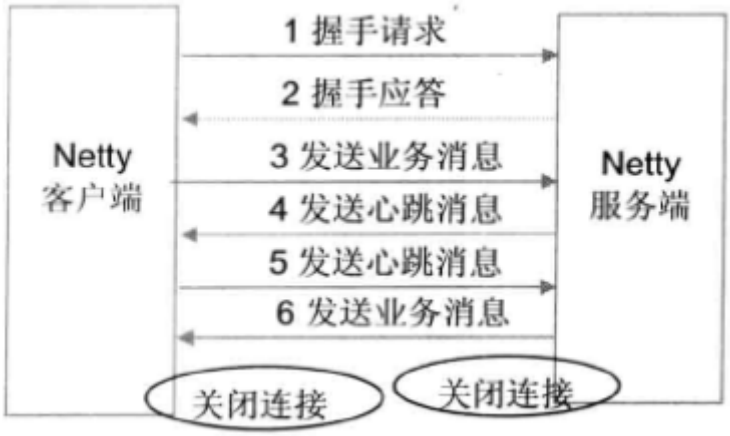
```
public class MyServer {

    public static void main(String[] args) throws Exception {
        EventLoopGroup bossGroup = new NioEventLoopGroup();
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            ServerBootstrap serverBootstrap = new ServerBootstrap();
            serverBootstrap.group(bossGroup,
workerGroup).channel(NioServerSocketChannel.class).
                    handler(new LoggingHandler(LogLevel.INFO)).
                    childHandler(new WebSocketChannelInitializer());
            ChannelFuture channelFuture = serverBootstrap.bind(new
InetSocketAddress(8899)).sync();
            channelFuture.channel().closeFuture().sync();
        } finally {
            bossGroup.shutdownGracefully();
            workerGroup.shutdownGracefully();
        }
    }

    public static class WebSocketChannelInitializer extends
ChannelInitializer<SocketChannel> {

        @Override
        protected void initChannel(SocketChannel ch) {
            ChannelPipeline pipeline = ch.pipeline();
            // 将请求或应答编解码成 HTTP 消息
            pipeline.addLast(new HttpServerCodec());
            // 向客户端发送HTML5文件，主要用于支持浏览器和服务端进行WebSocket通信
```

```
            pipeline.addLast(new ChunkedWriteHandler());
            // 将HTTP消息的多个部分组合成一条完整的HTTP消息
            pipeline.addLast(new HttpObjectAggregator(8192));
            // 增加标识协议头
            pipeline.addLast(new WebSocketServerProtocolHandler("/ws"));
            // 增加 WebSocket服务端的Handler
            pipeline.addLast(new TextWebSocketFrameHandler());
        }
    }


    public static class TextWebSocketFrameHandler extends
SimpleChannelInboundHandler<TextWebSocketFrame> {

        @Override
        protected void channelRead0(ChannelHandlerContext ctx,
TextWebSocketFrame msg) throws Exception {
            System.out.println("收到消息:  " + msg.text());

            ctx.channel().writeAndFlush(new TextWebSocketFrame("服务器时间:  " +
LocalDateTime.now()));
        }

        @Override
        public void handlerAdded(ChannelHandlerContext ctx) throws Exception {
            System.out.println("handlerAdded: " +
ctx.channel().id().asLongText());
        }

        @Override
        public void handlerRemoved(ChannelHandlerContext ctx) throws Exception {
            System.out.println("handlerRemoved: " +
ctx.channel().id().asLongText());
        }

        @Override
        public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
throws Exception {
            System.out.println("异常发生");
            ctx.close();
        }
    }
}
```

# 6 私有协议开发

## 6.1 通信模型

私有协议架构图




1. Netty 协议栈客户端发送握手请求消息，携带节点 ID 等有效身份认证信息；
2. Netty 协议栈服务端对握手请求消息进行合法性效验，返回登录成功的握手应答；
3. 链路建立成功后，客户端发送业务消息
4. 链路成功后，服务端发送心跳消息
5. 链路建立成功后，客户端发送心跳消息
6. 链路建立成功后，服务端发送业务消息
7. 服务端退出后，服务端关闭连接，客户端感知对方关闭连接后，被动关闭客户端连接

## 6.2 消息定义

### 6.2.1 消息头

| 名称 | 类型 | 长度 | 描述 |
| --- | --- | --- | --- |
| crcCode | int | 32 | 1）0xABEF：固定值，表明是 Netty 协议消息，2个字节<br>2）主版本号：1~255，1个字节<br>3）次版本号：1~255，1个字节 |
| length | int | 64 | 消息长度，包括消息头和消息体 |
| sessionID | long | 64 | 集群节点内全局唯一，由会话ID生成器生成 |
| type | Byte | 8 | 0：业务请求消息；1：业务响应消息；2：既是请求又是响应<br>3：握手请求消息；4：握手应答消息；5：心跳请求消息<br>6：心跳应答消息 |
| priority | Byte | 8 | 消息优先级：0~255 |
| attachment | Map | 变长 | 可选，用于扩展消息头 |

### 6.2.1 消息体

选择合适的序列化方式，以及编解码方式

## 6.3 可靠性保障

- 心跳机制
- 重连机制
- 重复登录保护
- 消息缓存重发

## 6.4 安全性设计

- 长连接采用基于 IP 地址的安全认证机制，服务端对握手请求消息的 IP 地址进行合法性效验（黑白名单）；
- 基于密钥和 AES 加密的用户名和密码认证机制，也可采用 SSL/TSL 安全传输；

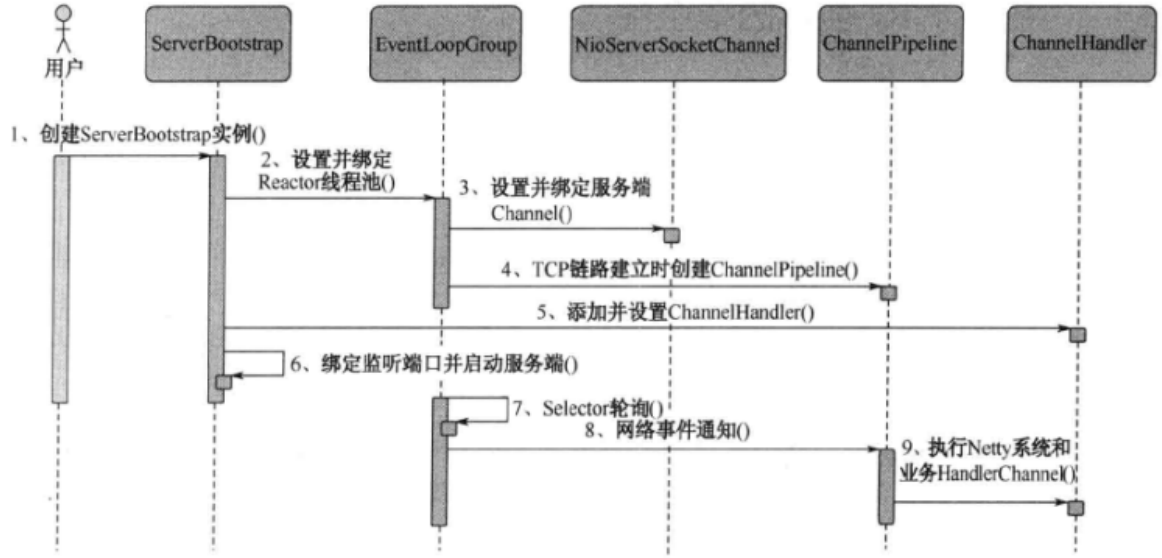
### 6.5 可扩展性

- Netty 预留了 attachment 可变长的消息头字段，可以用于扩展业务字段，例如：消息流水号、业务自定义消息头等。

# 7 服务端原理

## 7.1 过程

时序图




1. 创建 ServerBootstrap 实例。ServerBootstrap 是 Netty 服务端的启动辅助类，提供了一系列方法用于设置 服务端启动相关参数。
2. 设置并绑定 Reactor 线程池。EventLoopGroup 是 Netty 的 Reactor 线程池，实际是 EventLoop 的数组。 EventLoop 主要**处理所有注册到本线程中 Selector 上的 Channel**，Selector 的轮询操作由绑定的 EventLoop 线程的 run 方法驱动，在一个循环体内循环执行。
3. 设置并绑定服务端 Channel。Java NIO 服务端需要创建 ServerSocketChannel，而 Netty 对其进行封装， 利用反射方式，只需要指定实现，不需要关注具体细节。
4. 链路建立的时候创建并初始化 ChannelPiepeline。
5. 添加并设置 ChannelHandler。
6. 绑定并启动监听端口。在绑定监听端口之前系统会做一系列的初始化和检测工作，完成后将 ServserSocketChannel 注册到 Selector 上监听客户端连接。
7. Selector 轮询。由 Reactor 线程 NioEventLoop 负责调度和执行 Selector 轮询操作，选择准备就绪的 Channel 集合。
8. 当轮询到准备就绪的 Channel 后，就由 Reactor 线程 NioEventLoop 执行 ChannelPipeline 相应方法。

## 7.2 细节

- bind 方法主要是将 NioServerSocketChannel 注册到 NioEventLoop 的 Selector 上

# 8 客户端原理

## 8.1 过程

1. 用户线程创建 Bootstrap 实例，通过 API 设置创建客户端相关的参数，异步发起客户端连接。
2. 创建处理客户端连接、I/O 读写的 Reactor 线程组 NioEventLoopGroup。
3. 通过 Bootstrap 的 ChannelFactory 和用户指定的 Channel 类型创建用于客户端连接 NioSocketChannel（功能类似于 NIO 的 SocketChannel）
4. 创建默认的 ChannelHandlerPipeline
5. 异步发起 TCP 连接，并注册到 Selector 上
6. 注册对应的网络监听状态到 Selector
7. Selector 轮询各 Channel，处理连接结果
8. 处理 Future 结果，触发 ChannelPiepeline