

# Netty

## Java I/O

### I/O基础

#### Linux网络I/O模型简介

UNIX编程对I/O模型的分类，共有五种I/O模型：

##### 1.阻塞I/O

最常用的I/O模型就是阻塞I/O，缺省情况下，所有文件操作都是阻塞的。在进程空间中调用recvfrom，其系统调用直到数据包到达且被复制到应用进程的缓冲区中或发生错误时才返回，在此期间一直等待，进程从调用recvfrom开始到它返回的整段时间内都是被阻塞的。

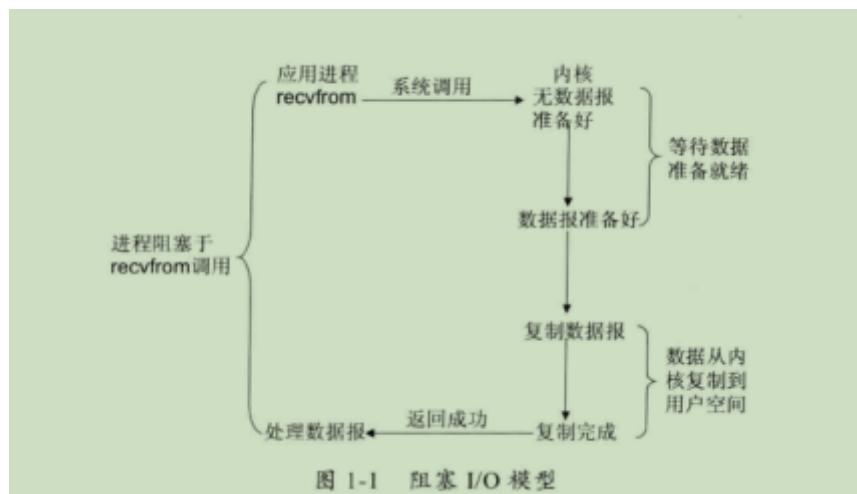


图 1-1 阻塞 I/O 模型

##### 2.非阻塞I/O

recvfrom从应用层到内核的时候，如果该缓冲区没有数据的话，就直接返回一个EWOULDBLOCK错误，一般对非阻塞I/O模型进行轮询检查这个状态。

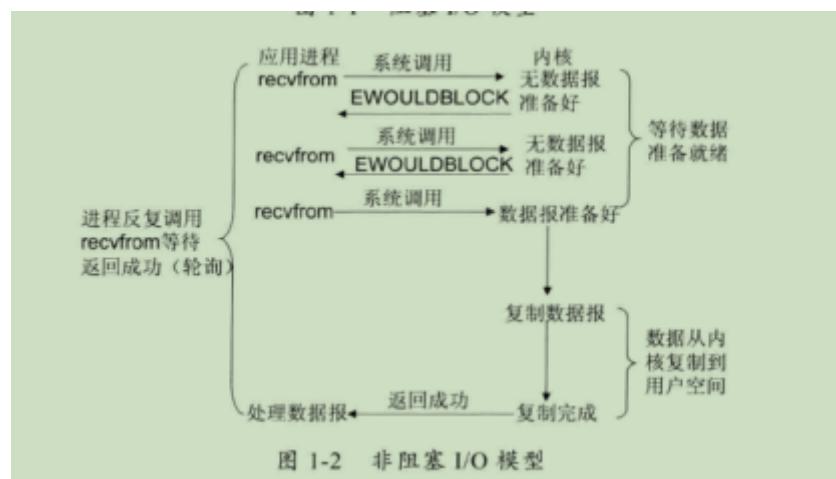
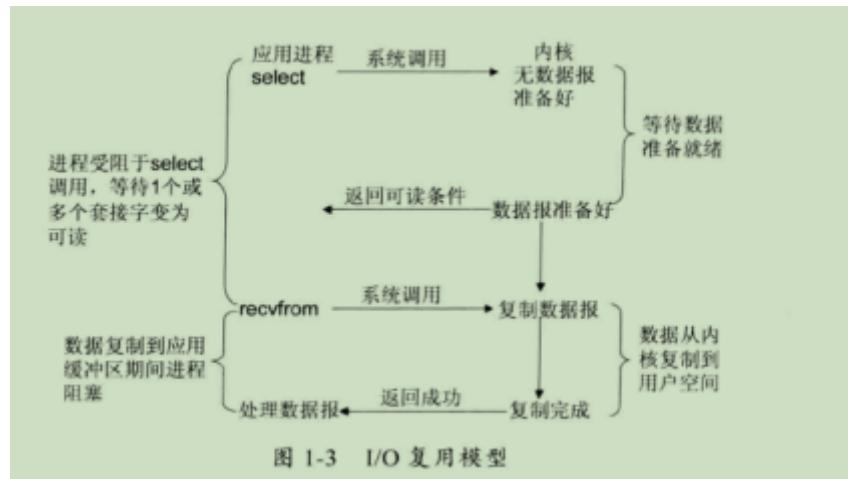


图 1-2 非阻塞 I/O 模型

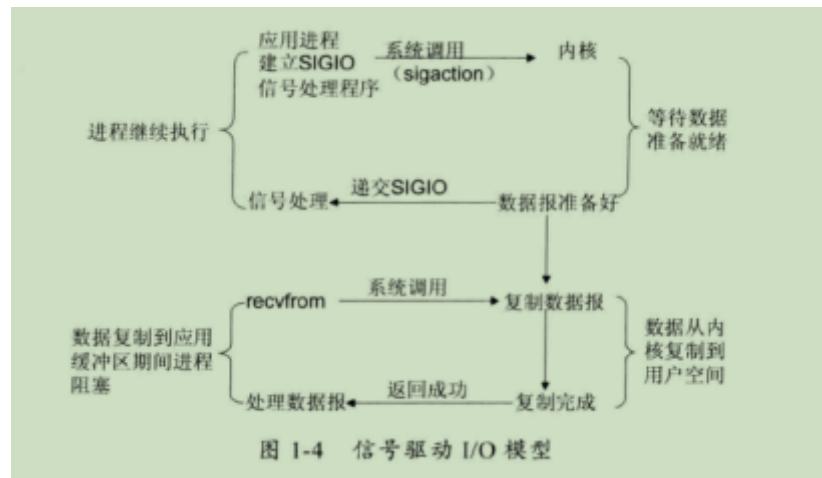
##### 3.I/O复用模型

Linux提供Select/poll，进程通过将一个或多个fd(socket描述符)传递给select或poll系统调用，阻塞在select操作上，这样select/poll可以帮我们侦测多个fd是否处于就绪状态。select/poll是按顺序扫描fd是否就绪。而且支持的fd数量有限，因此它的使用受到了一些约束。Linux还提供一个epoll系统调用，epoll使用基于事件驱动方式代替顺序扫描，因此性能更高。当有fd就绪时，立即回调rollback函数



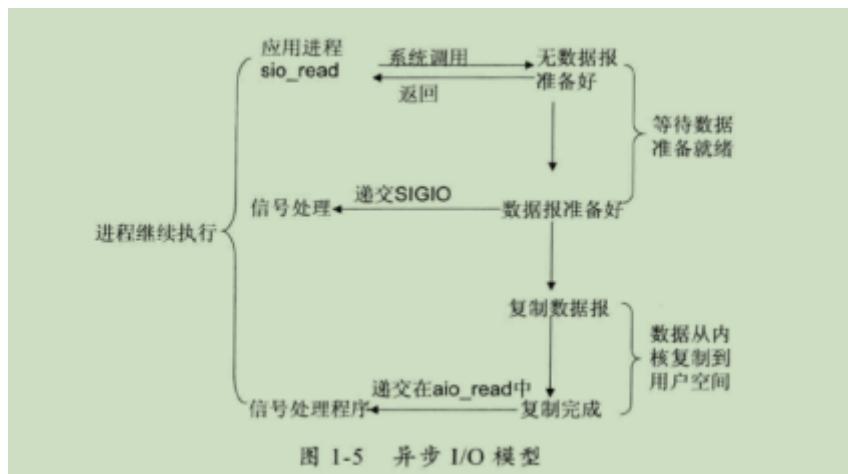
#### 4.信号驱动I/O模型

首先开启套接口信号驱动I/O功能，并通过系统调用sigaction执行一个信号处理函数（此系统调用立即返回，进程继续工作，是非阻塞的）。当数据准备就绪时，内核该进程生成一个SIGIO信号，通过信号回调通知应用程序调用recvfrom来读取数据，并通知主循环函数处理数据。



#### 5.异步I/o

告知内核启动某个操作，并让内核在整个操作完成后（包括将数据从内核复制到用户自己的缓冲区）通知我们。与信号驱动模型的区别是：信号驱动是由内核告知我们何时可以开始一个I/O操作，异步I/O是由内核通知我们何时操作已完成。



## I/O多路复用

在I/O编程中，当需要同时处理多个客户端接入请求时，可以利用多线程或者I/O多路复用技术进行处理。I/O多路复用技术通过把多个I/O的阻塞复用到同一个select阻塞上，从而使得系统在单线程的情况下可以同时处理多个客户端请求。

与传统的多线程模型相比，I/O多路复用最大优势是系统开销小，系统不需要创建新的额外线程或进程，也不需要维护这些线程或进程的运行，降低了系统维护的工作量，节省系统资源，主要应用场景如下：

- 服务器需要同时处理多个处于监听状态或多个连接状态的socket
- 服务器需要同时处理多种网络协议的socket

目前支持I/O多路复用的系统调用有select, pselect, poll, epoll。以前一直使用select，现在改为epoll。epoll对于select的改进

1. 支持一个进程打开的socket描述符（FD）不受限制，仅限于操作系统的最大文件句柄数
2. I/O的效率不会随着FD的数目增加而线性下降  
select/poll每次调用都会线性扫描全部的集合，导致效率呈线性下降。epoll不存在这个问题，它只会对活跃的socket进行操作--因为epoll是根据每个fd上面的callback函数实现的。
3. 使用mmap加速内核与用户空间的消息传递  
无论是select、poll、epoll都需要内核把FD消息通知给用户空间。epoll可以通过内核和用户空间mmap同一块内存来实现。
4. epoll的API更加简单

## JAVA I/O的演进

在JDK1.4出现NIO之前，JAVA的所有Socket通信都是基于同步阻塞的BIO。

NIO主要类和接口如下：

- 进行异步I/O操作的ByteBuffer等
- 进行异步I/O操作的管道Pipe
- 进行各种I/O操作的（异步或同步）的Channel，包括ServerSocketChannel和SocketChannel
- 多种字符集的编码能力和解码能力
- 实现非阻塞I/O操作的duolufuyongselector
- 基于流行的Perl实现的正则表达式类库
- 文件通道FileChannel

不足之处：

- 没有统一的文件属性（如读写权限）
- API能力较弱，如目录的级联创建和递归遍历，需自己实现
- 底层存储系统的一些高级API无法使用
- 所有的文件操作都是同步阻塞调用，不支持异步文件读写操作

JDK1.7正式发布后的NIO2.0，对三个方面进行了改进：

- 提供能够批量获取文件属性的API。还提供了标准文件的SPI，供各个服务提供商扩展实现
- 提供AIO功能，支持基于文件的异步I/O操作和网络套接字的异步操作
- 完成JSR-51定义的通道功能，包括对配置和多播数据的支持

# NIO入门

## NIO类库简介

### 缓冲区Buffer

Buffer是一个对象，包含一些要写入或者要读出的数据。在面向流的I/O中，可以将数据直接写入或者将数据直接读到Stream对象中。

在NIO库中，所有数据都是用缓冲区处理的，在读取数据时，是直接读到缓冲区中的。在写入数据时，写入到缓冲区中。任何时候访问NIO中的数据，都是通过缓冲区进行操作。

缓冲区实质是一个数组。通常是一个字节数组，也可以是其他种类的数组。提供了对数据结构化访问以及维护读写位置等信息。

每一种Java基本类型对应的都有缓冲区：

- ByteBuffer 字节缓冲区
- CharBuffer 字符缓冲区
- ShortBuffer 短整型缓冲区
- IntBuffer 整形缓冲区
- LongBuffer 长整型缓冲区
- FloatBuffer 浮点型缓冲区
- DoubleBuffer 双精度浮点型缓冲区

每一个Buffer类都是Buffer接口的一个子实例。因为大多数标准I/O操作都使用ByteBuffer，所以它在具有一般缓冲区的操作之外还提供了一些特有的操作，以方便网络读写。除了ByteBuffer，每一个Buffer类都有完全一样的操作，只不过处理的数据类型不一样。

### 通道Channel

Channel是一个通道，网络数据通过Channel读取和写入。通道与流的不同之处，在于通道是双向的，流只是在一个方向上移动，通道可以用于读、写或二者同时进行。

### 多路复用器Selector

Selector会不断的轮询注册其上的Channel，如果某个Channel上发生读或者写事件，这个Channel就处于就绪状态，会被Selector轮询出来，然后通过SelectionKey可以获取就绪Channel集合，进行后续操作。

一个多路复用Selector可以同时轮询多个Channel，由于JDK使用了epoll()代替传统的select，所以它并没有最大连接句柄1024/2048的限制。这也意味着只需要一个线程负责Selector的轮询，就可以接入成千上万个客户端。

NIO 服务端通信序列图如图 2-10 所示。

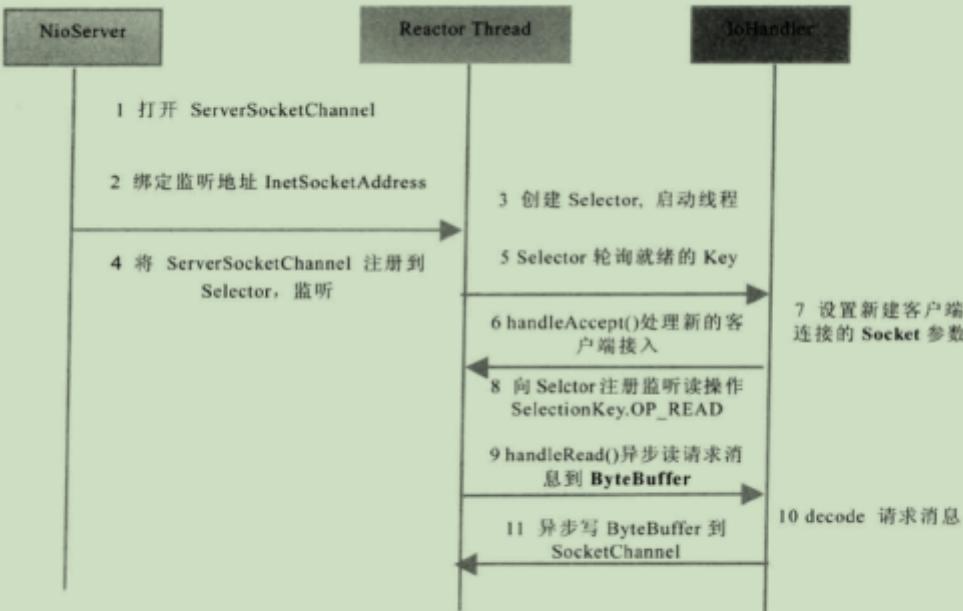


图 2-10 NIO 服务端通信序列图

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

```
ServerSocketChannel acceptorSvr = ServerSocketChannel.open();
```

2. 绑定监听端口，设置连接为非阻塞模式

```
acceptorSvr.socket().bind(new  
InetSocketAddress(InetAddress.getByName("IP"), port));  
acceptorSvr.configureBlocking(false);
```

3. 创建Reactor线程，创建多路复用器并启动线程

```
Selector selector = Selector.open();  
new Thread(new ReactorTask()).start();
```

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

```
SelectionKey key = acceptorSvr.register(selector, SelectionKey.OP_ACCEPT,  
ioHandler);
```

5. 多路复用器在线程run方法的无限循环体内轮询准备就绪的Key

```
int num = selector.select();  
Set selectedKeys = selector.selectedKeys();  
Iterator it = selectedKeys.iterator();  
while (it.hasNext()) {  
    SelectionKey key = (SelectionKey)it.next();  
    // ... deal with I/O event ...  
}
```

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

```
SocketChannel channel = svrChannel.accept();
```

7. 设置客户端链路为非阻塞模式

```
channel.configureBlocking(false);  
channel.socket().setReuseAddress(true);  
.....
```

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

```
SelectionKey key = socketChannel.register(selector, SelectionKey.OP_READ,  
ioHandler);
```

## 9. 异步读取客户端请求信息到缓冲区

```
int readNumber = channel.read(receivedBuffer);
```

## 10. 对ByteBuffer进行编解码，将解码成功的消息封装成Task，投递到业务线程池

```
Object message = null;
while(buffer.hasRemain())
```

• 29 •

Netty权威指南（第2版）

```
{
    byteBuffer.mark();
    Object message = decode(byteBuffer);
    if (message == null)
    {
        byteBuffer.reset();
        break;
    }
    messageList.add(message );
}
if (!byteBuffer.hasRemain())
byteBuffer.clear();
else
    byteBuffer.compact();
if (messageList != null & !messageList.isEmpty())
{
    for(Object messageE : messageList)
        handlerTask(messageE);
}
```

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

```
socketChannel.write(buffer);
```

如果发送区TCP缓冲区满，会导致写半包，此时，需要注册监听写操作位，循环写，知道整包消息写入TCP缓冲区。

### 2.3.4 NIO 客户端序列图

NIO 客户端创建序列图如图 2-11 所示。

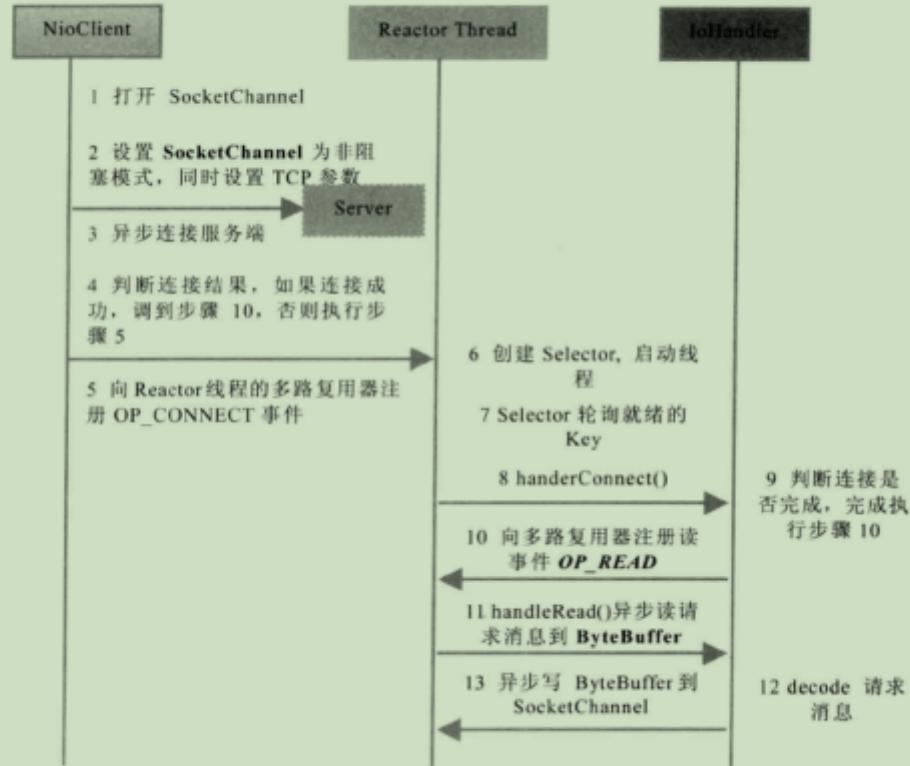


图 2-11 NIO 客户端创建序列图

#### 1. 打开SocketChannel绑定客户端本机地址

```
SocketChannel clientChannel = SocketChannel.open();
```

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

```
clientChannel.configureBlocking(false);
socket.setReuseAddress(true);
socket.setReceiveBufferSize(BUFFER_SIZE);
socket.setSendBufferSize(BUFFER_SIZE);
```

#### 3. 异步连接服务端

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

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

```
if (connected)
{
    clientChannel.register(selector, SelectionKey.OP_READ, ioHandler);
}
else
{
    clientChannel.register(selector, SelectionKey.OP_CONNECT, ioHandler);
}
```

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

```
clientChannel.register(selector, SelectionKey.OP_CONNECT, ioHandler);
```

#### 6. 创建Reactor线程, 创建多路复用器并启动线程

```
Selector selector = Selector.open();
new Thread(new ReactorTask()).start();
```

## 7. 多路复用器在线程run方法的无限循环体内轮询准备就绪的Key

```
int num = selector.select();
Set selectedKeys = selector.selectedKeys();
Iterator it = selectedKeys.iterator();
while (it.hasNext()) {
    SelectionKey key = (SelectionKey)it.next();
    // ... deal with I/O event ...
}
```

## 8. 接受connect事件进行处理

```
if (key.isConnectable())
    //handlerConnect();
```

## 9. 判断连接结果，如果连接成功，注册读事件到多路复用器

```
if (channel.finishConnect())
    registerRead();
```

## 10. 注册读事件到多路复用器

```
clientChannel.register( selector, SelectionKey.OP_READ, ioHandler);
```

## 11. 异步读客户端请求消息到缓冲区

```
int readNumber = channel.read(receivedBuffer);
```

## 12. 对ByteBuffer进行编解码，将编码成功的消息封装成Task，投递到业务线程池

```
Object message = null;
while(buffer.hasRemain())
{
    byteBuffer.mark();
    Object message = decode(byteBuffer);
    if (message == null)
    {
        byteBuffer.reset();
        break;
    }
    messageList.add(message );
}
if (!byteBuffer.hasRemain())
byteBuffer.clear();
else
    byteBuffer.compact();
if (messageList != null & !messageList.isEmpty())
{
    for(Object messageE : messageList)
        handlerTask(messageE);
}
```

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

端

```
socketChannel.write(buffer);
```

## AIO

异步通道提供以下两种方式来获取操作结果：

- 通过java.util.concurrent.Future类来表示异步操作的结果
- 在执行异步操作的时候传入一个java.nio.channels

CompletionHanlder接口的实现类作为操作完成的回调。

## TCP粘包拆包问题解决

## TCP粘包拆包

### 4.1.1 TCP 粘包/拆包问题说明

我们可以通过图解对 TCP 粘包和拆包问题进行说明，粘包问题示例如图 4-1 所示。



图 4-1 TCP 粘包/拆包问题

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

- (1) 服务端分两次读取到了两个独立的数据包，分别是 D1 和 D2，没有粘包和拆包；
- (2) 服务端一次接收到了两个数据包，D1 和 D2 粘合在一起，被称为 TCP 粘包；
- (3) 服务端分两次读取到了两个数据包，第一次读取到了完整的 D1 包和 D2 包的部分内容，第二次读取到了 D2 包的剩余内容，这被称为 TCP 拆包；
- (4) 服务端分两次读取到了两个数据包，第一次读取到了 D1 包的部分内容 D1\_1，第二次读取到了 D1 包的剩余内容 D1\_2 和 D2 包的整包。

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

### TCP粘包拆包的原因

原因有三个：

1. 应用程序write写入的字节大小大于套接口发送缓冲区大小
2. 碱性MSS大小的TCP分段
3. 以太网帧的payload大于MTU进行IP分片

业界的主流协议解决方案：

1. 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
2. 在包尾增加回车换行符进行分割，例如FTP
3. 将消息分为消息头和消息体，消息头中包含表示消息总长度（或者消息体长度）的字段，通常的设计思路为消息头的第一个字段使用int32来表示消息的总长度
4. 更复杂的应用层协议

TCP以流的方式进行数据传输，上层的应用协议为了对消息进行区分，往往采用四种方式：

1. 消息长度固定，累计读取到长度总和为定长LEN的报文后，就认为读取带了一个完整的消息，将计数器置位，重新开始读取下一个数据报文
2. 将回车换行符作为消息结束符，例如FTP
3. 将特殊的分隔符作为消息结束的标志
4. 通过在消息头中定义长度字段来表示消息的总长度

Netty对上述四种应用做了统一的抽象，提供四种解码器来解决对应的问题。

利用LineBasedFrameDecoder解决TCP粘包

LineBasedFrameDecoder工作原理是依次遍历ByteBuf中的可读字节，判断是否有回车或换行符。如果有，以此位置为结束位置，从可读索引位置到结束位置区间的字节就组成了一行。**它是以换行符为结束标志的解码器，支持携带结束符或不携带结束符两种编码方式，并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。

StringDecoder就是将接收到的对象转换成字符串，然后继续调用后面的Handler。

LineBasedFrameDecoder和StringDecoder组合就是按行切换的文本解码器。

DelimiterBasedFrameDecoder可以指定特定的分隔符，可以自动完成以分隔符作为码流结束标识的消息的解码。

FixedLengthFrameDecoder是固定长度的解码器。能够对指定的长度对消息进行自动解码。

## 编解码技术

java序列化缺点

1. 无法跨语言
2. 序列化后的码流太大
3. 序列化性能太低

## 主流编解码框架

### Google的Protobuf（使用二进制进行编码）

全称为Google Protocol Buffers。它将数据结构以.proto文件进行描述，通过代码生成工具，可以生成对应数据结构的POJO对象和ProtoBuf相关的方法和属性。

具有以下特点：

- 结构化数据存储格式（XML、JSON等）
- 高效的编解码性能
- 语言无关、平台无关、扩展性好
- 官方支持Java、c++、Python三种语言

Protobuf的数据描述文件和代码生成机制具有以下优点：

- 文本化的数据结构描述语言，可以实现语言和平台无关，特别适合异构系统间的集成
- 通过标识字段的顺序，可以实现协议的前向兼容
- 自动代码生成，不需要手工编译
- 方便后续的管理和维护，相比于代码，结构化的文档更容易管理和维护

### Facebook的Thrift

支持多种程序语言。在不同语言之间通信，Thrift可以作为高性能的通信中间件使用，支持数据序列化和多种类型的RPC服务。适用于静态的数据交换，需要先确定好他的数据结构，当数据结构发生变化时，必须重新编辑IDL文件，生成代码和编译。

主要有五部分组成

1. 语言系统以及IDL编译器，负责由用户给定的IDL文件生成相应的语言的接口代码。
2. TProtocol：RPC协议层，可以选择多种不同的对象序列化方式，
3. TTransport：RPC的传输层，可以选择不同的传输层实现：如socket、NIO、MemoryBuffer
4. TProcessor：作为协议层和用户提供的服务实现之间的纽带，负责调用服务实现的接口
5. TServer：聚合TProtocol、TTransport和TProcessor等对象

TProtocol就是Thrift的编解码框架。通过IDL描述接口和数据结构定义，支持8种Java基本类型，Map、Set、和List，支持可选和必选定义。因为可以定义数据结构中字段的顺序，所以它也可以支持协议的前向兼容。支持的三种比较经典的编解码方式

1. 通用的二进制编解码
2. 压缩二进制编解码
3. 优化的可选字段压缩编解码

## JBoss Marshalling

JBoss Marshalling是一个Java对象的序列化API包，修正了JDK自带的序列化包的很多问题，但又保持和java.io.Serializable接口的兼容。同时又增加了一些可调参数和附加特性。相对于传统的Java序列化机制，优点如下：

1. 可插拔的类解析器，提供更加便捷的类加载定制策略，通过一个接口可实现定制，
2. 可插拔的对象替换技术，不需要通过继承的方式
3. 可插拔的预定义类缓存表，可以减小序列化的字节数组长度，提升常用类型的对象序列化性能
4. 无需事先java.io.Serializable接口
5. 通过缓存技术提升对象的序列化性能

## MessagePack编解码

MessagePack是一个高效的二进制序列化框架。支持不同语言之间的数据交换。

### MessagePack

特点：

1. 编解码高效，性能高
2. 序列化之后的码流较小
3. 支持跨语言

#### 7.1.2 MessagePack Java API 介绍

如果使用 Maven 工程开发，在你的 pom.xml 中配置 MessagePack 的坐标，如下。

```
<dependency>
    <groupId>org.msgpack</groupId>
    <artifactId>msgpack</artifactId>
    <version>${msgpack.version}</version>
</dependency>
```

它的 API 使用起来非常简单，编码和解码开发示例如下。

```
// Create serialize objects.
List<String> src = new ArrayList<String>();
src.add("msgpack");
src.add("kumofs");
src.add("viver");
MessagePack msgpack = new MessagePack();
// Serialize
byte[] raw = msgpack.write(src);
// 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));
```

## Google ProtoBuf编解码

ProtoBufDecoder仅仅负责解码，不支持读半包。因此，在ProtobufDecoder前面，一定要有能够处理半包的解码器。有以下三种方式供选择：

1. 使用Netty提供的ProtobufVarint32FrameDecoder
2. 集成Netty提供的通用半包解码器LengthFieldBasedFrameDecoder
3. 集成ByteToMessageDecoder类，自己处理半包消息

## HTTP协议开发应用

### HTTP协议介绍

HTTP协议是建立在TCP传输协议之上的应用层协议。是一个属于应用层的面向对象的协议，简捷、快速，适用于分布式超媒体信息系统。

主要特点：

- 支持Client/Server模式
- 简单--客户向服务器请求服务时，只需指定服务URL，携带必要的请求参数或者消息体
- 灵活--HTTP允许传输任意类型的数据对象，传输的内容类型由HTTP消息头中的Content-Type加以标记
- 无状态--HTTP协议是无状态协议，无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要之前的信息，则必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快，负载较轻

### HTTP请求消息

请求由三部分组成：

1. HTTP请求行
2. HTTP消息头
3. HTTP请求正文

请求行以一个方法符开头，以空格分开，后面跟着请求的URI和协议的版本。格式为Method

Request-URI HTTP-Version CRLF。

Method表示请求方法，Request-URI是一个统一的资源标识符，HTTP-Version表示请求的HTTP协议的版本，CRLF表示回车和换行。

请求的方法有很多种，作用如下：

- GET：请求获取Request-URI标识的资源
- POST：在Request-URI所标识的资源后附加新的提交数据
- HEAD：请求获取由Request-URI所表示的资源的响应消息报头
- PUT：请求服务器存储一个资源，并用Request-URI作为其标识
- DELETE：请求服务器删除Request-URI所表示的资源
- TRACE：请求服务器回送收到的请求信息，主要用于测试或诊断
- CONNECT：保留将来使用
- OPTIONS：请求查询服务器的性能，或者查询与资源相关的选项和请求

GET一般用于获取/查询资源信息，而POST一般用于更新资源信息。GET和POST主要区别如下：

1. 根据HTTP规范，GET用于信息获取，而且应该是安全和幂等的，POST则表示可能改变服务器上资源的请求
2. GET提交，请求的数据会附在URL之后，就是把数据放置在请求行中，以? 分割URL和传输数据，多个参数用&连接；而POST提交会把数据放置在HTTP消息的包体中，数据不会在地址栏中显示出

来。

3. 传输数据的大小不同，特定浏览器和服务器对URL长度有限制。所以GET携带的参数长度会受到浏览器的限制，而POST由于不是通过URL传值，不会受到限制
4. 安全性。POST比GET安全性高，GET在URL终会暴露隐私数据，而且GET提交的数据还可能造成Cross-site request forgery 攻击。

表 10-1 HTTP 的部分请求消息头列表

名称 (KEY)	作用
Accept	用于指定客户端接受哪些类型的信息。 例如：Accept:image/gif，表明客户端希望接受 GIF 图像格式的资源
Accept-Charset	用于指定客户端接受的字符集。 例如：Accept-Charset:iso-8859-1,gb2312，如果在请求消息中没有设置这个域，默认是任何字符集都可以接受
Accept-Encoding	类似于 Accept，但是它用于指定可接受的内容编码。 例如：Accept-Encoding:gzip,deflate，如果请求消息中没有设置这个域，则服务器假定客户端对各种内容编码都可以接受
Accept-Language	类似于 Accept，但是它用于指定一种自然语言。 例如：Accept-Language:zh-cn，如果请求消息中没有设置这个报头域，则服务器假定客户端对各种语言都可以接受
Authorization	主要用于证明客户端有权查看某个资源。当浏览器访问一个页面时，如果收到服务器的响应代码为 401（未授权），可以发送一个包含 Authorization 请求报头域的请求，要求服务器对其进行认证
Host	发送请求时，该报头域是必需的，用于指定被请求资源的 Internet 主机和端口号，它通常是从 HTTP URL 中提取出来的
User-Agent	允许客户端将它的操作系统、浏览器和其他属性告诉服务器
Content-Length	请求消息体的长度
Content-Type	表示后面的文档属于什么 MIME 类型。Servlet 默认为 text/plain，但通常需要显式地指定为 text/html。 由于经常要设置 Content-Type，因此 HttpServletResponse 提供了一个专用的方法 setContentType
Connection	连接类型

## HTTP响应消息

HTTP响应也是由三个部分组成,分别是状态行、消息报头、响应正文.

状态行的格式为: HTTP-Version States-Code Reason-Phrase CRLF, States-Code表示服务器返回的响应状态代码。

状态代码由三位数字组成，第一个数字定义了响应的类别，由5种可能值。

1. 1xx: 指示信息。表示请求已接收、继续处理。
2. 2xx: 成功，表示已被成功接收、理解、接受
3. 3xx: 重定向。要完成请求必须进行更进一步的操作
4. 4xx: 客户端错误。请求有语法错误或无法实现
5. 5xx: 服务器端错误。服务器未能处理请求

表 10-2 HTTP 响应状态代码和描述信息

状态码	状态描述
200	OK: 客户端请求成功
400	Bad Request: 客户端请求有语法错误, 不能被服务器所理解
401	Unauthorized: 请求未经授权, 这个状态代码必须和 WWW-Authenticate 报头域一起使用
403	Forbidden: 服务器收到请求, 但是拒绝提供服务
404	Not Found: 请求资源不存在
500	Internal Server Error: 服务器发生不可预期的错误
503	Server Unavailable: 服务器当前不能处理客户端的请求, 一段时间后可能恢复正常

响应报头允许服务器传递不能放在状态行中的附加响应信息, 以及关于服务器的信息和对 Request-URI 所标识的资源进行下一步访问的信息。常用的响应报头如表 10-3 所示。

表 10-3 常用的响应报头

名称 (KEY)	作用
Location	用于重定向接收者到一个新的位置, Location 响应报头域常用于更换域名的时候
Server	包含了服务器用来处理请求的软件信息, 与 User-Agent 请求报头域是相对应的
WWW-Authenticate	必须被包含在 401 (未授权的) 响应消息中, 客户端收到 401 响应消息, 并发送 Authorization 报头域请求服务器对其进行验证时, 服务端响应报头就包含该报头域

### JiBx框架 (XML绑定框架)

JiBx是一款非常优秀的XML数据绑定框架。提供灵活的绑定映射文件, 实现数据对象与XML之间的转换, 并且不需要修改既有的Java类。

优点:

1. 转换效率高
2. 配置绑定文件简单
3. 不需要操作xpath文件
4. 不需要写get/set方法
5. 对象的属性名与xml的element名可以不同

两个比较重要的概念: Unmarshal (数据分解) 和Marshal (数据编排) 。

Unmarshal是将XML文件转换为Java对象, 而Marshal是将Java对象编排成xml文件。

## WebSocket协议

HTTP协议的弊端:

1. HTTP协议为半双工协议。半双工协议指数据可以在客户端和服务端两个方向传输, 但是不能同时传输, 意味着在同一时刻, 只有一个方向上的数据传送。
2. HTTP消息冗长繁琐, HTTP消息包括消息头, 消息体, 换行符等。通常情况下采用文本传输, 相比于其他的二进制通信协议, 冗长而繁琐。
3. 针对服务器推送的黑客攻击。例如长时间轮询

WebSocket是一种浏览器与服务器全双工通信的技术。在WebSocket API中, 浏览器和服务器只需要一个握手的动作, 就形成了一条快速通道, 两者就可以互传数据。WebSocket是基于TCP双向全双工进行消息传递。同一时刻, 即可以发送消息, 也可以接受消息。

特点:

1. 单一的TCP连接, 采用全双工模式通信。
2. 对代理, 防火墙和路由器透明
3. 无头部信息、Cookie和身份验证
4. 无安全开销

5. 通过ping/pong帧保持链路激活
6. 服务器可以主动传递消息给客户端，不再需要客户端轮询

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGh1IHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

图 11-3 WebSocket 客户端握手请求消息

为了建立一个 WebSocket 连接，客户端浏览器首先要向服务器发起一个 HTTP 请求，这个请求和通常的 HTTP 请求不同，包含了一些附加头信息，其中附加头信息“Upgrade: WebSocket”表明这是一个申请协议升级的 HTTP 请求。服务器端解析这些附加的头信息，然后生成应答信息返回给客户端，客户端和服务端的 WebSocket 连接就建立起来了，双方可以通过这个连接通道自由地传递信息，并且这个连接会持续存在直到客户端或者服务端的某一方主动关闭连接。

服务端返回给客户端的应答消息如图 11-4 所示。

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzhZRBK+xOo=
Sec-WebSocket-Protocol: chat
```

图 11-4 WebSocket 服务端返回的握手应答消息

请求消息中的“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，把“Sec-WebSocket-Key”加上一个魔幻字符串“258EAFA5-E914-47DA-95CA-C5AB0DC85B11”。使用 SHA-1 加密，然后进行 BASE-64 编码，将结果做为“Sec-WebSocket-Accept”头的值，返回给客户端。

## 生命周期

握手成功之后，服务端和客户端可以通过“messages”的方式进行通信，一个消息由一个或多个帧组成，WebSocket 的消息并补一点给对应一个特定网络层的帧，它可以被分割成多个帧或者被合并。帧都有自己的类型的，属于同一个消息的多个帧具有相同类型的数据。

## Netty私有协议

Netty 协议栈消息定义包含两部分：

- 消息头：
- 消息体。

其具体定义分别如表 12-1 和表 12-2 所示。

表 12-1 Netty 消息定义表 (NettyMessage)

名 称	类 型	长 度	描 述
header	Header	变长	消息头定义
body	Object	变长	对于请求消息，它是方法的参数（作为示例，只支持携带一个参数）；对于响应消息，它是返回值

表 12-2 Netty 协议消息头定义 (Header)

名 称	类 型	长 度	描 述
crcCode	整型 int	32	Netty 消息的校验码，它由三部分组成： 1) 0xA8EF，固定值，表明该消息是 Netty 协议消息，2 个字节； 2) 主版本号：1~255，1 个字节； 3) 次版本号：1~255，1 个字节。 $crcCode = 0xA8EF + \text{主版本号} + \text{次版本号}$

• 225 •

Netty 权威指南（第 2 版）

续表

名 称	类 型	长 度	描 述
length	整型 int	32	消息长度，整个消息，包括消息头和消息体
sessionID	长整型 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: 业务 ONE WAY 消息（既是请求又是响应消息） 3: 握手请求消息 4: 握手应答消息 5: 心跳请求消息 6: 心跳应答消息
priority	Byte	8	消息优先级：0~255
attachment	Map<String, Object>	变长	可选字段，用于扩展消息头

### 12.2.5 Netty 协议支持的字段类型

Netty 协议支持的数据类型如表 12-3 所示。

表 12-3 Netty 协议支持的数据类型

字段类型	备注说明
boolean	包括它的包装类型 Integer
byte	包括它的包装类型 Byte
int	对应于 C/C++ 的 int32
char	包括它的包装类型 Character
short	对应 C/C++ 的 int16
long	对应 C/C++ 的 int64
float	包括它的包装类型 Float
double	包括它的包装类型 Double
string	对应 C/C++ 的 String
list	支持各种 List 的实现
array	支持各种数组的实现
map	支持 Map 的嵌套和泛型
set	支持 Set 的嵌套和泛型

## Netty协议的编码

编码规范如下：

1. cecCode: java.nio.ByteBuffer.putInt(int value)，如果采用其他缓冲区实现，必须与其等价
2. length: java.nio.ByteBuffer.putInt(int value)，如果采用其他缓冲区实现，必须与其等价
3. sessionID: java.nio.ByteBuffer.putLong(long value)，如果采用其他缓冲区实现，必须与其等价
4. type: java.nio.ByteBuffer.put (byte b) 如果采用其他缓冲区实现，必须与其等价

5. priority: java.nio.ByteBuffer.put (byte b) 如果采用其他缓冲区实现，必须与其等价
6. attachment: 它的编码规则为--如果attachment长度为0，表示没有可选附件，则长度编码设为0，java.nio.ByteBuffer.putInt(0); 如果大于0，说明有附件需要编码，具体编码队列如下：
  - 1.首先对附件个数进行编码，java.nio.ByteBuffer.putInt(attachment.size())
  - 2.然后对key进行编码，先编码长度，再将他转换成byte数组之后编码内容
7. body的编码，通过JBoss Marshalling 将其序列化为Byte数组，然后调用java.nio.ByteBuffer.put(byte[] src)将其写入Byte Buffer缓冲区

由于整个消息长度必须等全部字段都编码完成之后才能确认，所以最后需要更新消息头中的length字段，将其重新写入ByteBuffer中。

## Netty协议的解码

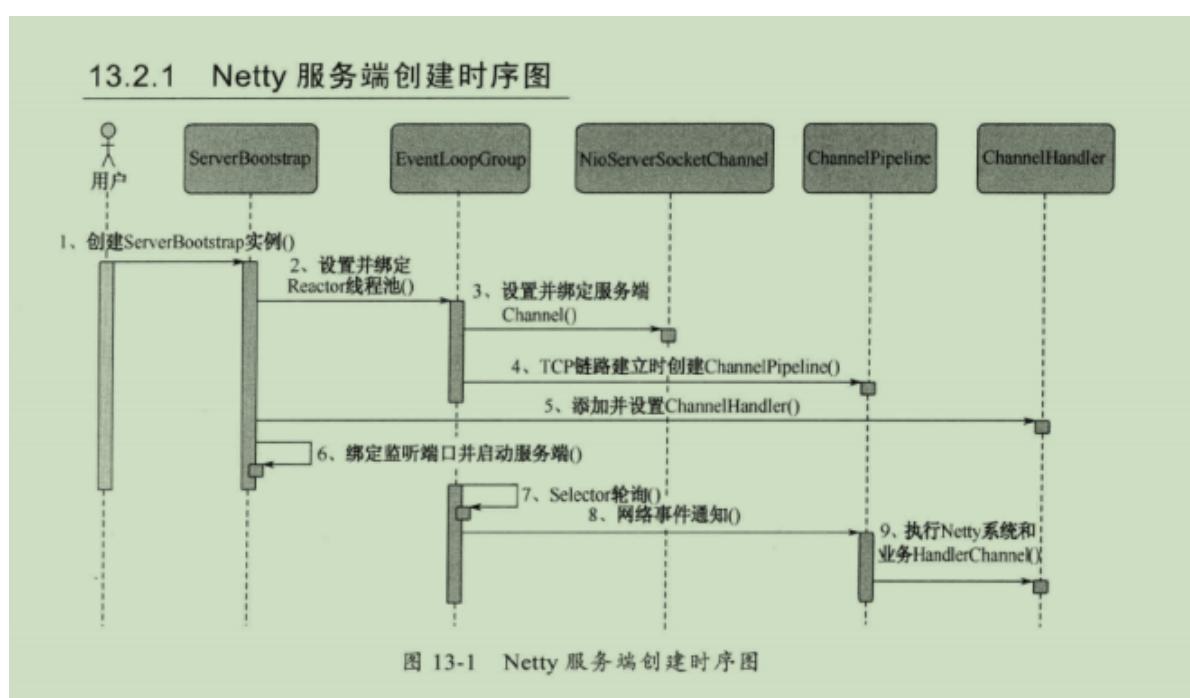
1. cecCode: java.nio.ByteBuffer.getInt()获取验证码字段，如果采用其他缓冲区实现，必须与其等价
2. length: java.nio.ByteBuffer.getInt()获取Netty消息长度，如果采用其他缓冲区实现，必须与其等价
3. sessionID: java.nio.ByteBuffer.getLong()获取会话ID，如果采用其他缓冲区实现，必须与其等价
4. type: java.nio.ByteBuffer.get()取消息类型，如果采用其他缓冲区实现，必须与其等价
5. priority: java.nio.ByteBuffer.get()获取消息优先级，如果采用其他缓冲区实现，必须与其等价
6. attachment: 解码规则为：首先创建一个新的attachment对象，调用java.nio.ByteBuffer.getInt()获取附件长度，如果为0，说明附件为空，解码结束，如果非空，根据长度通过for循环进行解码
7. body: 通过JBoss 的marshaller对其进行解码

## 链路的建立

链路建立需要通过基于IP地址或者号段的和黑名单安全认证机制。实际项目中，也可以通过密钥对用户名和密码进行安全认证。

## Netty源码

### 服务端创建



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的启动相关参数，底层通过门面模式对各种能力进行了抽象和封装。

2. 设置并绑定Reactor线程池。Netty的Reactor线程池是EventLoopGroup，实际上是一个EventLoop数组。EventLoop的职责是处理所有注册到本线程多路复用器Selector上的Channel，Selector的轮询操作由绑定的EventLoop线程run方法驱动，在一个循环体内循环执行。**而且EventLoop的指责不仅仅是处理网络I/O事件，用户自定义的Task和定时任务Task也统一由EventLoop负责。这样就实现了线程模型的统一。从调用层面看，EventLoop线程中不会再启动其他类型的线程，避免了多线程的并发操作和锁竞争**

--源码中的逻辑

将NioServerSocketChannel注册到Reactor线程时，首先判断是否是NioEventLoop自身发起的操作，如果是，不存在并发，直接执行Channel注册；如果是其他线程发起的，则封装成一个Task放入消息队列中异步执行。

3. 设置并绑定服务端的Channel。Netty的ServerBootstrap方法提供了channel方法用于指定服务器的channel类型。**作为NIO服务器，需要创建ServerSocketChannel，Netty对原生NIO类库进行了封装，对应的实现是NioServerSocketChannel。**

--源码中的逻辑

用户可以为启动辅助类和其父类分别指定Handler。两类的Handler的用途不同，子类中的Handler是NioServerSocketChannel对应的ChannelPipeline的Handler；父类的Handler是客户端新接入的连接SocketChannel对应的ChannelPipeline的Handler。

本质的区别是：ServerBootstrap中的Handler是NioServerSocketChannel中使用的，所有的连接该监听端口的客户端都会使用；父类AbstractBootstrap中的Handler是个工厂类，为每个新接入的客户端都创建一个新的Handler。

4. 链路建立的时候创建并初始化ChannelPipeline。ChannelPipeline并不是NIO服务端必须的。它本质是一个负责处理网络事件的职责链，负责管理和执行ChannelHandler。网络事件以事件流的形式在ChannelPipeline中流转，由ChannelPipeline根据ChannelHandler的执行策略调度ChannelHandler的执行。

典型的网络事件如：链路注册、链路激活、链路断开、接收到请求消息、请求消息接收并处理完毕、发送应答消息、链路发生异常、发生用户自定义事件

5. 初始化ChannelPipeline完成后，添加并设置ChannelHandler。ChannelHandler是Netty提供给用户定制和扩展的关键接口。利用ChannelHandler用户可以完成大多数的功能定制。如：消息编解码、心跳、安全认证、TSL/SSL认证、流量控制和流量整形等。

Netty 同时也提供了大量的系统 ChannelHandler 供用户使用，比较实用的系统 ChannelHandler 总结如下。

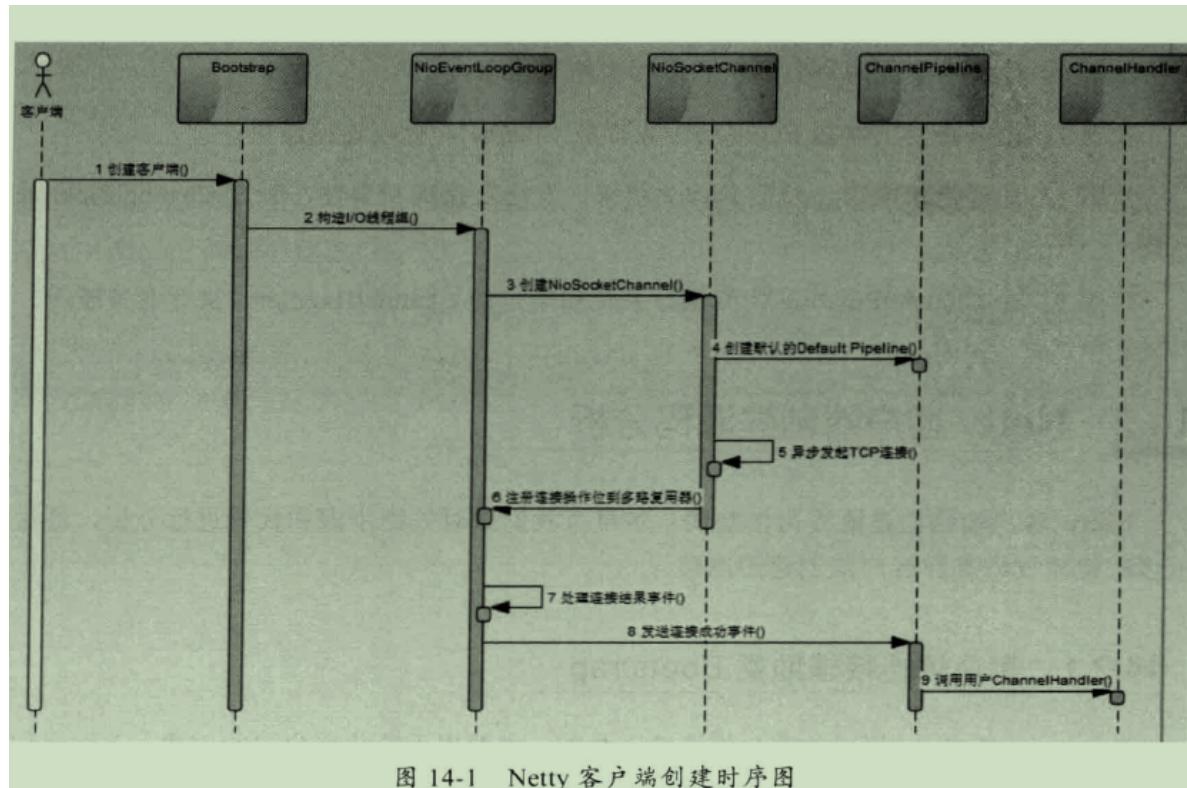
- (1) 系统编解码框架——ByteToMessageCodec;
- (2) 通用基于长度的半包解码器——LengthFieldBasedFrameDecoder;
- (3) 码流日志打印 Handler——LoggingHandler;
- (4) SSL 安全认证 Handler——SslHandler;
- (5) 链路空闲检测 Handler——IdleStateHandler;
- (6) 流量整形 Handler——ChannelTrafficShapingHandler;
- (7) Base64 编解码——Base64Decoder 和 Base64Encoder。

创建和添加 ChannelHandler 的代码示例如下。

```
.childHandler(new ChannelInitializer<SocketChannel>() {
    @Override
    public void initChannel(SocketChannel ch)
        throws Exception {
        ch.pipeline().addLast(
            new EchoServerHandler());
    }
});
```

6. 绑定并设置监听端口。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，完成之后，会启动监听端口，并将ServerSocketChannel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. 轮循到准备就绪的Channel后，就由Reactor线程NioEventLoop执行ChannelPipeline的相应方法，最终调度并执行ChannelHandler
9. 执行Netty系统ChannelHandler和用户添加定制的ChannelHandler。

## 客户端创建



1. 用户线程创建Bootstrap实例，通过API设置创建客户端的相关参数，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。可以通过构造函数指定I/O线程的个数，默认为CPU内核数的2倍
3. 通过Bootstrap的ChannelFactory和用户指定的Channel类型创建用于客户端连接的NioSocketChannel，它的功能类似于JDK NIO类库提供的SocketChannel
4. 创建默认的Channel Handler Pipeline，用于调度和执行网络事件
5. 异步发起TCP链接，判断是否连接成功。如果成功，则直接将NioSocketChannel注册到多路复用器上，监听读操作位，用于数据报读取和消息发送；如果没有立即连接成功，则注册监听到多路复用器，等待连接结果
6. 注册对应的网络监听状态位到多路复用器
7. 有多路复用器在I/O现场中轮询各Channel，处理连接结果
8. 如果连接成功，设置Future结果，发送连接成功事件，触发ChannelPipeline执行
9. 由ChannelPipeline调度执行系统和用户的ChannelHandler，执行业务逻辑

## ByteBuf和相关辅助类

Bytebuffer的缺点：

1. ByteBuffer长度固定，一旦分配完成，他的容量不能动态的扩展和收缩，当需要编码的POJO对象大于ByteBuffer容量时，会发生索引越界异常。
2. ByteBuffer只有一个标识位置的指针position，读写的时候需要手工调用flip()和rewind()等。使用者必须小心的处理这些API，否则很容易导致程序处理失败

3. ByteBuffer的API功能有限，一些高级的实用的特性不支持，需使用者自己编程实现

## ByteBuf工作原理

ByteBuf依然是Byte数组的缓冲区，基本功能和DK的ByteBuffer基本一致，提供以下功能：

1. 7种Java基本类型、byte数组、ByteBuffer等的读写
2. 缓冲区自身的copy和slice等
3. 设置网络字节序
4. 构造缓冲区实例
5. 操作位置指针等方法

我们看下调用 flip()操作前后的对比。

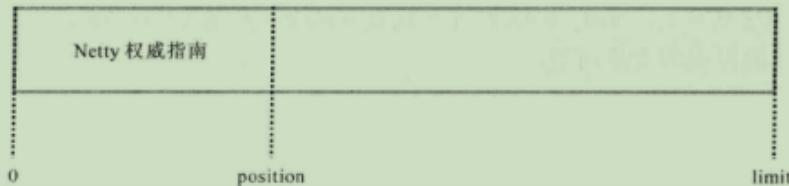


图 15-2 ByteBuffer flip()操作之前

如图 15-2 所示，如果不做 flip 操作，读取到的将是 position 到 capacity 之间的错误内容。

• 290 •

第 15 章 ByteBuf 和相关辅助类

当执行 flip()操作之后，它的 limit 被设置为 position，position 设置为 0，capacity 不变。由于读取的内容是从 position 到 limit 之间，因此，它能够正确地读取到之前写入缓冲区的内容。如图 15-3 所示。

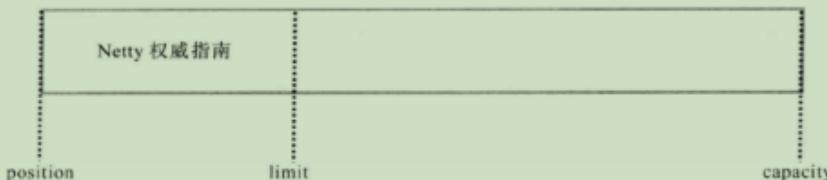


图 15-3 ByteBuffer flip()操作之后

ByteBuf通过两个位置指针来协助缓冲区的读写操作，读操作使用readerIndex，写操作使用writerIndex。

readerIndex和writerIndex的取值一开始都是0，随着数据的写入writerIndex会增加，读取数据会使readerIndex增加，但是它不超过writerIndex。在读取之后，0-readerIndex就被视为discard的，调用discardReadBytes方法，可以释放这部分空间，它的作用类似于Byte Buffer的compact方法。readerIndex和writerIndex之间的数据是可以读取的。writerIndex和capacity之间的空间是可写的。

JDK的ByteBuffer每进行一次put操作，都需要对可用空间进行校验。因为进行put操作时，如果缓冲区剩余空间不足，会发生BufferOverflowException。为了避免这个问题，每进行一次put操作，就会对剩余空间进行校验，如果剩余空间不足，会重新创建一个Bytebuffer，并将之前的Bytebuffer复制到新建的Bytebuffer，并释放原先的Bytebuffer。Netty的ByteBuf对write操作进行了封装，由write操作负责进行剩余可用空间的校验，如果底层缓冲区不足，ByteBuf会自动进行动态扩展

## ByteBuf功能介绍

### 1. 顺序读操作 (read)

ByteBuf 的 read 操作类似于 ByteBuffer 的 get 操作，主要的 API 功能说明如表 15-1 所示。

表 15-1 ByteBuf 的读操作 API 列表

方法名称	返 回 值	功 能 说 明
readBoolean	boolean	从 readerIndex 开始获取 boolean 值, readerIndex 增加 1
readByte	byte	从 readerIndex 开始获取字节值, readerIndex 增加 1
readUnsignedByte	byte	从 readerIndex 开始获取无符号字节值, readerIndex 增加 1
readShort	short	从 readerIndex 开始获取短整型值, readerIndex 增加 2
readUnsignedShort	short	从 readerIndex 开始获取无符号短整型值, readerIndex 增加 2

• 294 •

第 15 章 ByteBuf 和相关辅助类

续表

方法名称	返 回 值	功 能 说 明
readMedium	int	从 readerIndex 开始获取 24 位整型值, readerIndex 增加 3 (注意: 该类型并非 Java 的基本类型, 大多数场景使用不到)
readUnsignedMedium	int	从 readerIndex 开始获取 24 位无符号整型值, readerIndex 增加 3 (注意: 该类型并非 Java 的基本类型, 大多数场景使用不到)
readInt	int	从 readerIndex 开始获取整型值, readerIndex 增加 4
readUnsignedInt	int	从 readerIndex 开始获取无符号整型值, readerIndex 增加 4
readLong	long	从 readerIndex 开始获取长整型值, readerIndex 增加 8
readChar	char	从 readerIndex 开始获取字符值, readerIndex 增加 2
readFloat	float	从 readerIndex 开始获取浮点值, readerIndex 增加 4
readDouble	double	从 readerIndex 开始获取双精度浮点值, readerIndex 增加 8
readBytes(int length)	ByteBuf	将当前 ByteBuf 中的数据读取到新创建的 ByteBuf 中, 读取的长度为 length。操作成功完成之后, 返回的 ByteBuf 的 readerIndex 为 0, writerIndex 为 length 如果读取的长度 length 大于当前操作的 ByteBuf 的可写字节数, 将抛出 IndexOutOfBoundsException, 操作失败
readSlice(int length)	ByteBuf	返回当前 ByteBuf 新创建的子区域, 子区域与原 ByteBuf 共享缓冲区, 但是独立维护自己的 readerIndex 和 writerIndex 新创建的子区域 readerIndex 为 0, writerIndex 为 length 如果读取的长度 length 大于当前操作的 ByteBuf 的可写字节数, 将抛出 IndexOutOfBoundsException, 操作失败

readBytes(ByteBuf dst)	ByteBuf	<p>将当前 ByteBuf 的数据读取到目标 ByteBuf 中, 直到目标 ByteBuf 没有剩余的空间可写</p> <p>操作完成之后, 当前 ByteBuf 的 readerIndex += 读取的字节数</p> <p>如果目标 ByteBuf 可写的字节数大于当前 ByteBuf 可读取的字节数, 则抛出</p> <p>IndexOutOfBoundsException, 操作失败</p>
readBytes(ByteBuf dst, int length)	ByteBuf	<p>将当前 ByteBuf 的数据读取到目标 ByteBuf 中, 读取的字节数长度为 length</p> <p>操作完成之后, 当前 ByteBuf 的 readerIndex += length</p> <p>如果需要读取的字节数长度 length 大于当前 ByteBuf 可读的字节数或者目标 ByteBuf 可写的字节数, 则抛出</p> <p>IndexOutOfBoundsException, 操作失败</p>

• 295 •

Netty 权威指南 ( 第 2 版 )

续表

方法名称	返 回 值	功能说明
readBytes(ByteBuf dst, int dstIndex, int length)	ByteBuf	<p>将当前 ByteBuf 的数据读取到目标 ByteBuf 中, 读取的字节数长度为 length</p> <p>目标 ByteBuf 的起始索引为 dstIndex, 非 writerIndex</p> <p>操作完成之后, 当前 ByteBuf 的 readerIndex += length</p> <p>如果需要读取的字节数长度 length 大于当前 ByteBuf 可读的字节数, 或者 dstIndex 小于 0, 或者 dstIndex+length 大于目标 ByteBuf 的 capacity, 则抛出</p> <p>IndexOutOfBoundsException, 操作失败</p>
readBytes(byte[] dst)	ByteBuf	<p>将当前 ByteBuf 的数据读取到目标 byte 数组中, 读取的字节数长度为 dst.length</p> <p>操作完成之后, 当前 ByteBuf 的 readerIndex += dst.length</p> <p>如果目标字节数组的长度大于当前 ByteBuf 可读的字节数, 则抛出</p> <p>IndexOutOfBoundsException, 操作失败</p>
readBytes(byte[] dst, int dstIndex, int length)	ByteBuf	<p>将当前 ByteBuf 的数据读取到目标 byte 数组中, 读取的字节数长度为 length, 目标字节数组的起始索引为 dstIndex</p> <p>如果 dstIndex 小于 0, 或者 length 大于当前 ByteBuf 的可读字节数, 或者 dstIndex+length 大于 dst.length, 则抛出</p> <p>IndexOutOfBoundsException, 操作失败</p>

readBytes(ByteBuffer dst)	ByteBuf	<p>将当前 ByteBuf 的数据读取到目标 ByteBuffer 中，直到位置指针到达 ByteBuffer 的 limit</p> <p>操作成功完成之后，当前 ByteBuf 的 readerIndex+=dest.remaining()</p> <p>如果目标 ByteBuffer 的可写字节数大于当前 ByteBuf 可读字节数，则抛出 IndexOutOfBoundsException，操作失败</p>
readBytes(OutputStream out, int length)	ByteBuf	<p>将当前 ByteBuf 的数据读取到目标输出流中，读取的字节数长度为 length</p> <p>如果操作成功，当前 ByteBuf 的 readerIndex+=length</p> <p>如果 length 大于当前 ByteBuf 可读取的字节数，则抛出 IndexOutOfBoundsException，操作失败</p> <p>如果读取过程中 OutputStream 自身发生了 I/O 异常，则抛出 IOException</p>

• 296 •

## 第 15 章 ByteBuf 和相关辅助类

续表

方法名称	返 回 值	功能说明
readBytes(GatheringByteChannel out, int length)	int	<p>将当前 ByteBuf 的数据写入到目标 GatheringByteChannel 中，写入的最大字节数长度为 length</p> <p>注意：由于 GatheringByteChannel 是非阻塞 Channel，调用它的 write 操作并不能保证一次能够将所有需要写入的字节数都写入成功，即存在“写半包”问题。因此，它写入的字节数范围为[0,length]</p> <p>如果操作成功，当前 ByteBuf 的 readerIndex+=实际写入的字节数</p> <p>如果需要写入的 length 大于当前 ByteBuf 的可读字节数，则抛出</p> <p>IndexOutOfBoundsException 异常；如果操作过程中 GatheringByteChannel 发生了 I/O 异常，则抛出 IOException，无论抛出何种异常，操作都将失败</p> <p>与其他 read 方法不同的是，本方法的返回值不是当前的 ByteBuf，而是写入 GatheringByteChannel 的实际字节数</p>

## 2. 顺序写操作 ( write )

ByteBuf 的 write 操作类似于 ByteBuffer 的 put 操作，主要的 API 功能说明如表 15-2 所示。

表 15-2 ByteBuf 的写操作 API 列表

方法名称	返 回 值	功能说明
writeBoolean(boolean value)	ByteBuf	将参数 value 写入到当前的 ByteBuf 中 操作成功之后 writerIndex+=1 如果当前 ByteBuf 可写的字节数小于 1，则抛出 IndexOutOfBoundsException，操作失败
writeByte(int value)	ByteBuf	将参数 value 写入到当前的 ByteBuf 中 操作成功之后 writerIndex+=1 如果当前 ByteBuf 可写的字节数小于 1，则抛出 IndexOutOfBoundsException，操作失败

• 297 •

Netty 权威指南 ( 第 2 版 )

续表

方法名称	返 回 值	功能说明
writeShort(int value)	ByteBuf	将参数 value 写入到当前的 ByteBuf 中 操作成功之后 writerIndex+=2 如果当前 ByteBuf 可写的字节数小于 2，则抛出 IndexOutOfBoundsException，操作失败
writeMedium(int value)	ByteBuf	将参数 value 写入到当前的 ByteBuf 中 操作成功之后 writerIndex+=3 如果当前 ByteBuf 可写的字节数小于 3，则抛出 IndexOutOfBoundsException，操作失败
writeInt(int value)	ByteBuf	将参数 value 写入到当前的 ByteBuf 中 操作成功之后 writerIndex+=4 如果当前 ByteBuf 可写的字节数小于 4，则抛出 IndexOutOfBoundsException，操作失败
writeLong(long value)	ByteBuf	将参数 value 写入到当前的 ByteBuf 中 操作成功之后 writerIndex+=8 如果当前 ByteBuf 可写的字节数小于 8，则抛出 IndexOutOfBoundsException，操作失败

writeChar(int value)	ByteBuf	将参数 value 写入到当前的 ByteBuf 中 操作成功之后 writerIndex+=2 如果当前 ByteBuf 可写的字节数小于 2，则抛出 IndexOutOfBoundsException，操作失败
writeBytes(ByteBuf src)	ByteBuf	将源 ByteBuf src 中的所有可读字节写入到当前 ByteBuf 中 操作成功之后 当前 ByteBuf 的 writerIndex+=src.readableBytes 如果源 ByteBuf src 可读的字节数大于当前 ByteBuf 的可写字节数，则抛出 IndexOutOfBoundsException，操作失败
writeBytes(ByteBuf src, int length)	ByteBuf	将源 ByteBuf src 中的可读字节写入到当前 ByteBuf 中，写入的字节数长度为 length 操作成功之后 当前 ByteBuf 的 writerIndex+=length 如果 length 大于源 ByteBuf 的可读字节数或者当前 ByteBuf 的可写字节数，则抛出 IndexOutOfBoundsException，操作失败

• 298 •

## 第 15 章 ByteBuf 和相关辅助类

续表

方法名称	返 回 值	功能说明
writeBytes(ByteBuf src, int srcIndex, int length)	ByteBuf	将源 ByteBuf src 中的可读字节写入到当前 ByteBuf 中，写入的字节数长度为 length，起始索引为 srcIndex 操作成功之后 当前 ByteBuf 的 writerIndex+=length 如果 srcIndex 小于 0，或者 srcIndex + length 大于源 src 的容量；或者写入长度 length 大于当前 ByteBuf 的可写字节数，则抛出 IndexOutOfBoundsException，操作失败
writeBytes(byte[] src)	ByteBuf	将源字节数组 src 中的所有字节写入到当前 ByteBuf 中 操作成功之后 当前 ByteBuf 的 writerIndex+=src.length 如果源字节数组 src 的长度大于当前 ByteBuf 的可写字节数，则抛出 IndexOutOfBoundsException，操作失败

writeBytes(byte[] src, int srcIndex, int length)	ByteBuf	<p>将源字节数组 src 中的字节写入到当前 ByteBuf 中, 写入的字节数长度为 length, 起始索引为 srcIndex</p> <p>操作成功之后 当前 ByteBuf 的 writerIndex+= length</p> <p>如果 srcIndex 小于 0, 或者 srcIndex + length 大于源 src 的容量; 或者写入长度 length 大于当前 ByteBuf 的可写字节数, 则抛出 IndexOutOfBoundsException, 操作失败</p>
writeBytes(ByteBuffer src)	ByteBuf	<p>将源 ByteBuffer src 中所有可读字节写入到当前 ByteBuf 中, 写入的长度为 src.remaining()</p> <p>操作成功之后 当前 ByteBuf 的 writerIndex+= src.remaining()</p> <p>如果源 ByteBuffer src 的可读字节数大于当前 ByteBuf 的可写字节数, 则抛出 IndexOutOfBoundsException, 操作失败</p>
writeBytes(InputStream in, int length)	int	<p>将源 InputStream src 中的内容写入到当前 ByteBuf 中, 写入的最大字节数长度为 length</p> <p>实际写入的字节数可能小于 length</p> <p>操作成功之后当前 ByteBuf 的 writerIndex+= 实际写入的字节数</p> <p>如果 length 大于源 ByteBuf 的可读字节数或者当前 ByteBuf 的可写字节数, 则抛出 IndexOutOfBoundsException, 操作失败</p> <p>如果从 InputStream 读取的时候发生了 I/O 异常, 则抛出 IOException</p>

• 299 •

Netty 权威指南 (第 2 版)

续表

方法名称	返 回 值	功能说明
writeBytes(ScatteringByteChannel in, int length)	int	<p>将源 ScatteringByteChannel src 中的内容写入到当前 ByteBuf 中, 写入的最大字节数长度为 length</p> <p>实际写入的字节数可能小于 length</p> <p>操作成功之后当前 ByteBuf 的 writerIndex+= 实际写入的字节数</p> <p>如果 length 大于源 src 的可读字节数或者当前 ByteBuf 的可写字节数, 则抛出 IndexOutOfBoundsException, 操作失败</p> <p>如果从 ScatteringByteChannel 读取的时候发生了 I/O 异常, 则抛出 IOException</p>
writeZero(int length)	ByteBuf	<p>将当前的缓冲区内容填充为 NUL (0x00), 起始位置为 writerIndex, 填充的长度为 length</p> <p>填充成功之后 writerIndex+= length</p> <p>如果 length 大于当前 ByteBuf 的可写字节数则抛出 IndexOutOfBoundsException 操作失败</p>

## 7. Mark 和 Rest

当对缓冲区进行读操作时，由于某种原因，可能需要对之前的操作进行回滚。读操作并不会改变缓冲区的内容，回滚操作主要就是重新设置索引信息。

对于 JDK 的 ByteBuffer，调用 mark 操作会将当前的位置指针备份到 mark 变量中，当调用 rest 操作之后，重新将指针的当前位置恢复为备份在 mark 中的值，源码代码如图 15-17 所示。

```
public final Buffer mark() {  
    mark = position;  
    return this;  
}
```

图 15-17 mark 操作之后的缓冲区位置指针

调用 reset 操作之后如图 15-18 所示。

Netty 的 ByteBuf 也有类似的 rest 和 mark 接口，因为 ByteBuf 有读索引和写索引，因此，它总共有 4 个相关的方法，分别如下。

- markReaderIndex：将当前的 readerIndex 备份到 markedReaderIndex 中；
- resetReaderIndex：将当前的 readerIndex 设置为 markedReaderIndex；
- markWriterIndex：将当前的 writerIndex 备份到 markedWriterIndex；
- resetWriterIndex：将当前的 writerIndex 设置为 markedWriterIndex。

详细的功能自行百度

### 15.2.1 ByteBuf 的主要类继承关系

首先，我们通过主要功能类库的继承关系图（见图 15-24），来看下 ByteBuf 接口的不同实现。

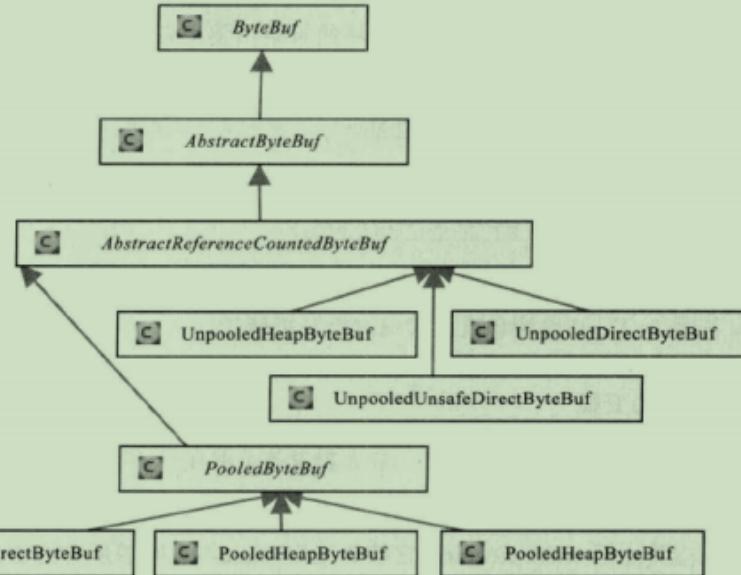


图 15-24 ByteBuf 主要功能类继承关系图

从内存分配的角度看，ByteBuf可以分为两类：

1. 堆内存字节缓冲区（HeapByteBuf）：特点是内存的分配和回收速度快，可以被JVM自动回收，缺点是如果进行Socket的I/O读写，需要额外做一次内存复制，将堆内存对应的缓冲区复制到内核 Channel 中，性能会有一定的下降
2. 直接内存字节缓冲区（DirectByteBuf）：非堆内存，它在堆外进行内存分配，相比于堆内存，他的分配和回收速度慢一些，但是将它写入或者从Socket Channel 中读取时，少了一次内存复制，

速度比堆内存快。

从内存回收角度看，也分为两类：

1. 基于对象池的ByteBuf：可以重用ByteBuf对象，自己维护了一个内存池。可以循环利用创建的ByteBuf，提升内存使用效率，降低高负载导致的频繁的GC。
2. 普通的ByteBuf

## UnpooledHeapByteBuf

UnpooledHeapByteBuf是基于堆内存进行内存分配的字节缓冲区，它没有基于对象池技术实现，意味着每次I/O读写都会创建一个新的UnpooledHeapByteBuf，频繁进行大块内存的分配和回收对象能会造成一定影响。

```
//聚合了一个ByteBufAllocator，用于UnpooledHeapByteBuf的内存分配
private final ByteBufAllocator alloc;
//缓冲区（直接用JDK的ByteBuffer替换byte数组也是可行的【因为ByteBuffer底层也是byte数组】，直接使用byte数组的原因就是提升性能和更加便捷的进行位操作）
byte[] array;
//用于实现Netty Byte Buf到JDK NIO ByteBuffer 的转换
private ByteBuffer tmpNioBuf;
```

## 动态扩展缓冲区

```
protected final void checkNewCapacity(int newCapacity) {
    ensureAccessible();
    if (checkBounds) {
        //1.首先对新容量进行合法性校验，如果大于容量上限或者小于0，则抛出异常
        if (newCapacity < 0 || newCapacity > maxCapacity()) {
            throw new IllegalArgumentException("newCapacity: " + newCapacity +
(expected: 0-" + maxCapacity() + ')');
        }
    }
}

@Override
public ByteBuf capacity(int newCapacity) {
    checkNewCapacity(newCapacity);

    int oldCapacity = array.length;
    byte[] oldArray = array;
    //2.判断新容量值是否大于当前的缓冲区容量，如果大于进行动态扩展
    if (newCapacity > oldCapacity) {
        //创建新的缓冲区字节数组
        byte[] newArray = allocateArray(newCapacity);
        //进行内存复制，将旧的字节数组复制到新创建的字节数组
        System.arraycopy(oldArray, 0, newArray, 0, oldArray.length);
        //替换旧的字节数组
        setArray(newArray);
        freeArray(oldArray);
    } else if (newCapacity < oldCapacity) {
        //小于当前的缓冲区容量，不需要动态扩展，但需要截取当前缓冲区容量创建一个新的子缓冲
区
        byte[] newArray = allocateArray(newCapacity);
        int readerIndex = readerIndex();
        //1.判断读索引是否小于新的容量值
        if (readerIndex < newCapacity) {
```

```

        int writerIndex = writerIndex();
        //2.判断写索引是否大于新的容量值
        if (writerIndex > newCapacity) {
            writerIndex(writerIndex = newCapacity);
        }
        //3.通过内存复制System.arraycopy将当前可读的字节数组复制到新创建的子缓冲区
        System.arraycopy(oldArray, readerIndex, newArray, readerIndex,
writerIndex - readerIndex);
    } else {
        //如果新容量值小于读索引，说明没有可读的字节数组需要复制到新创建的缓冲区，将
        //读写索引设置为新的容量值即可
        setIndex(newCapacity, newCapacity);
    }
    setArray(newArray);
    freeArray(oldArray);
}
return this;
}

private void setArray(byte[] initialArray) {
    array = initialArray;
    //动态扩容完成后，将原来的试图tmpNioBuf设置为空
    tmpNioBuf = null;
}

```

## 字节数组复制

```

//检验合法性
protected final void checkSrcIndex(int index, int length, int srcIndex, int
srcCapacity) {
    //检验index和length的值，如果小于0，抛出异常，再对两者之和进行判断，如果大于缓冲区的容
    //量，抛出异常
    checkIndex(index, length);
    if (checkBounds) {
        //与上述检查类似
        checkRangeBounds(srcIndex, length, srcCapacity);
    }
}

@Override
public ByteBuf setBytes(int index, byte[] src, int srcIndex, int length) {
    checkSrcIndex(index, length, srcIndex, src.length);
    //进行字节数组的复制
    System.arraycopy(src, srcIndex, array, index, length);
    return this;
}

```

## 转换为JDK ByteBuffer

```

@Override
public ByteBuffer nioBuffer(int index, int length) {
    ensureAccessible();
    return ByteBuffer.wrap(array, index, length).slice();
}

```

## PooledByteBuf内存池原理

### PoolArena

PoolArena本身是指一块区域，在内存管理中，Memory Arena指的是内存中一大块连续的区域，PoolArena是Netty的内存池实现类。

不同的框架，Memory Arena的实现是不同的，Netty的PoolArena是由多个Chunk组成的大块内存区域，而每个chunk则由一个或者多个Page组成。因此，对内存的组织和管理也就主要集中在如何管理和组织chunk和page。

```
final PooledByteBufAllocator parent;

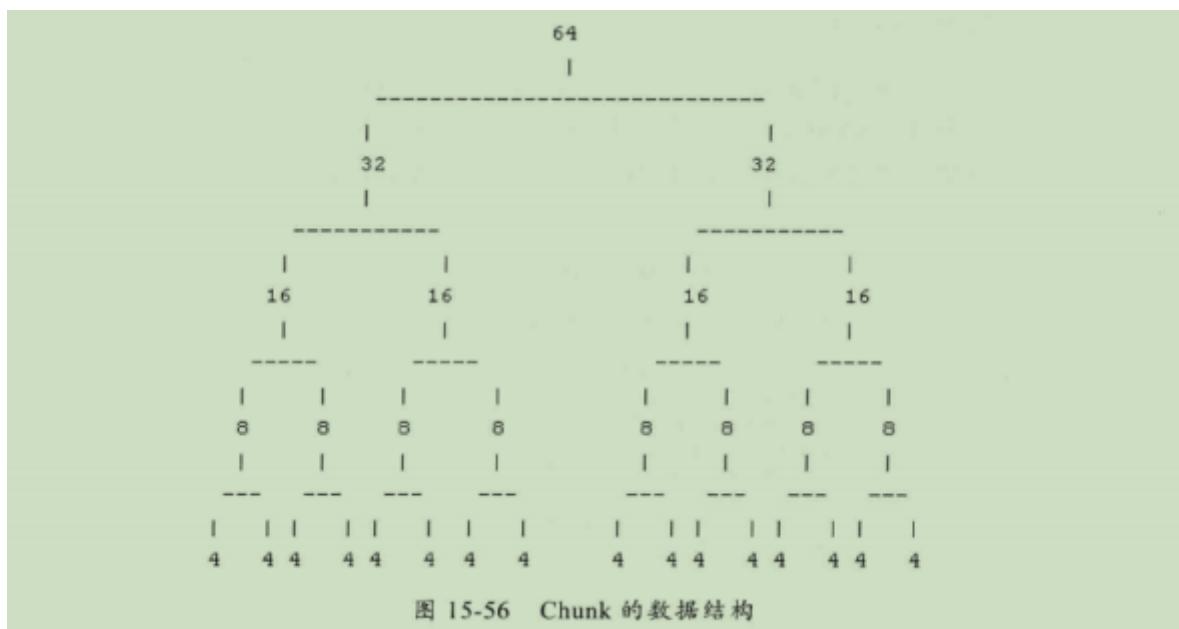
private final int maxOrder;
final int pageSize;
final int pageShifts;
final int chunkSize;
final int subpageOverflowMask;
final int numSmallSubpagePools;
final int directMemoryCacheAlignment;
final int directMemoryCacheAlignmentMask;
private final PoolSubpage<T>[] tinySubpagePools;
private final PoolSubpage<T>[] smallSubpagePools;

private final PoolChunkList<T> q050;
private final PoolChunkList<T> q025;
private final PoolChunkList<T> q000;
private final PoolChunkList<T> qInit;
private final PoolChunkList<T> q075;
private final PoolChunkList<T> q100;

private final List<PoolChunkListMetric> chunkListMetrics;
```

### PoolChunk

Chunk主要是用来组织和管理多个Page的内存分配和释放，再Netty中，Chunk中的Page被构建成一个二叉树，假设一个chunk由16个Page组成，则page结构如下：



Page的大小是4个字节，chunk就是64个字节，整棵树有五层，第一层（叶子节点的那一层）用来分配所有page的内存，以此类推。

每个节点记录自己在整個Memory Arena中的偏移地址，当一个节点代表的内存区域被分配出去之后，这个节点会标记为已分配，自这个节点以下的所有节点在后面的内存分配请求中都会被忽略。

对树的遍历采用深度优先的算法，但是选择哪个节点继续遍历则是随机的，并不是通常的深度优先算法那样总是访问左边的子节点。

## PoolSubpage

对于小于一个page的内存，Netty再page中完成分配。每个Page会被切分成大小相等的多个存储块，存储块的大小由第一次申请的内存大小决定。假如一个page的大小是8个字节，如果第一次申请的块大小是4个字节，那么这个page就包含两个存储块。

一个Page只能用于分配与第一次申请时大小相同的内存。比如：一个4字节的Page，如果第一次分配了1字节的内存，那么后面这个Page就只能继续分配1字节的内存；如果需要申请2字节的内存，就需要再另一个Page中分配。

Page中存储区域的使用状态是通过一个Long数组维护的，数组中每个long的每一位标识一个块存储区域的占用情况。0表示未占用，1表示已占用。对于一个4字节的Page来说，如果这个Page用来分配1个字节的存储区域，那么Long数组中就只有一个Long类型的元素，这个数值的低4位标识各个存储区域的占用情况。对于一个128字节的page，如果这个page也是用来分配一个字节的存储区域，那么long数组中就包含两个元素，总共128位。

## 内存回收策略

无论是chunk还是page，都是通过状态位来表示内存是否可用，不同之处是chunk通过在二叉树上节点进行标识，page是通过维护块的使用状态来标识。

## PooledDirectByteBuf

是基于内存池实现的。与UnpooledHeapByteBuf不同的就是缓冲区的分配和销毁策略不同。

### 创建字节缓冲区

```
static PooledDirectByteBuf newInstance(int maxCapacity) {
    //由于是基于内存池实现的，新创建的时候不能直接new一个对象，而是要从内存池中获取
    PooledDirectByteBuf buf = RECYCLER.get();
    //设置引用计数器的值
    buf.reuse(maxCapacity);
    return buf;
}

final void reuse(int maxCapacity) {
    //设置缓冲区的最大容量
    maxCapacity(maxCapacity);
    //设置引用计数器为1
    setRefCnt(1);
    setIndex0(0, 0);
    discardMarks();
}
```

## 复制新的字节缓冲区实例

```
@Override  
public ByteBuf copy(int index, int length) {  
    //首先对索引和长度进行合法性校验,  
    checkIndex(index, length);  
    //如果是基于内存池的分配器, 会从内存池中获取可用的ByteBuf, 如果是非池, 则直接创建新的  
    ByteBuf  
    ByteBuf copy = alloc().directBuffer(length, maxCapacity());  
    copy.writeBytes(this, index, length);  
    return copy;  
}
```

## ByteBuf相关辅助类

### ByteBufHolder

ByteBufHolder是ByteBuf的容器。例如HTTP协议的请求消息和应答消息都可以携带消息体。这个消息体在Netty中就是ByteBuf对象。由于不同的协议消息体可以包含不同的协议字段和功能。需要对ByteBuf进行封装。所以Netty抽象出了ByteBufHolder对象。

使用者继承该接口可以封装自己的实现。

### ByteBufAllocator

ByteBufAllocator是字节缓冲区分配器，按照Netty的缓冲区实现不同，共有两种不同的分配器。基于内存池的字节缓冲区分配器和普通的字节缓冲区分配器。

下面我们给出 ByteBufAllocator 的主要 API 功能列表（表 15-3）。

表 15-3 ByteBufAllocator 主要 API 功能列表

方法名称	返回值说明	功能说明
buffer()	ByteBuf	分配一个字节缓冲区，缓冲区的类型由 ByteBufAllocator 的实现类决定
buffer(int initialCapacity)	ByteBuf	分配一个初始容量为 initialCapacity 的字节缓冲区，缓冲区的类型由 ByteBufAllocator 的实现类决定
buffer(int initialCapacity, int maxCapacity)	ByteBuf	分配一个初始容量为 initialCapacity，最大容量为 maxCapacity 的字节缓冲区，缓冲区的类型由 ByteBufAllocator 的实现类决定
ioBuffer(int initialCapacity, int maxCapacity)	ByteBuf	分配一个初始容量为 initialCapacity，最大容量为 maxCapacity 的 direct buffer，因为 direct buffer 的 I/O 操作性能更高
heapBuffer(int initialCapacity, int maxCapacity)	ByteBuf	分配一个初始容量为 initialCapacity，最大容量为 maxCapacity 的 heap buffer
directBuffer(int initialCapacity, int maxCapacity)	ByteBuf	分配一个初始容量为 initialCapacity，最大容量为 maxCapacity 的 direct buffer
compositeBuffer(int maxNumComponents)	CompositeByteBuf	分配一个最大容量为 maxCapacity 的 CompositeByteBuf，内存类型由 ByteBufAllocator 的实现类决定
isDirectBufferPooled()	boolean	是否使用了直接内存内存池

## CompositeByteBuf

CompositeByteBuf允许将多个ByteBuf的实例组装在一起，形成一个统一的视图。

CompositeByteBuf的作用。例如某个协议的POJO对象包含两部分，消息头和消息体，它们都是ByteBuf对象。需要对消息进行编码的时候需要进行整合。如果使用JDK，有以下两种方式：

1. 将某个ByteBuffer复制到另一个ByteBuffer中，或者创建一个新的ByteBuffer，将两者复制到新建的ByteBuffer
2. 通过List或数组等容器，将消息头和消息体放到容器中进行统一维护和处理。

CompositeByteBuf实现

```
//维护了ByteBuf的位置偏移量信息
private static final class Component {
    final ByteBuf buf;
    final int length;
    int offset;
    int endOffset;

    Component(ByteBuf buf) {
        this.buf = buf;
        length = buf.readableBytes();
    }
}

private static final ByteBuffer EMPTY_NIO_BUFFER =
Unpooled.EMPTY_BUFFER.nioBuffer();
private static final Iterator<ByteBuf> EMPTY_ITERATOR = Collections.
<ByteBuf>emptyList().iterator();

private final ByteBufAllocator alloc;
private final boolean direct;
//维护了一个Component集合，Component就是ByteBuf的包装实现类
private final ComponentList components;
private final int maxNumComponents;

private boolean freed;

//添加Component
public CompositeByteBuf addComponents(int cIndex, ByteBuf... buffers) {
    addComponents0(false, cIndex, buffers, 0, buffers.length);
    consolidateIfNeeded();
    return this;
}

//移除Component
public CompositeByteBuf removeComponent(int cIndex) {
    checkComponentIndex(cIndex);
    Component comp = components.remove(cIndex);
    comp.freeIfNecessary();
    if (comp.length > 0) {
        // only need to call updateComponentOffsets if the length was > 0
        updateComponentOffsets(cIndex);
    }
    return this;
}
```

```
}
```

## ByteBufUtil

### 常用方法

```
//对需要编码的字符串src按照指定的字符集进行编码，利用指定的ByteBufAllocator生成一个新的  
ByteBuf  
ByteBuf encodeString(ByteBufAllocator alloc, CharBuffer src, Charset charset)  
//使用指定的ByteBuf和Charset进行解码，获取解码后的字符串  
String decodeString(ByteBuf src, int readerIndex, int len, Charset charset)
```

## Channel

io.netty.channel.Channel是Netty网络操作抽象类，它聚合了一组功能，包括但不限于网络的读写、客户端发起连接、主动关闭连接、链路关闭、获取双方网络地址等。还包括获取该Channel的EventLoop，获取缓冲分配器ByteBufAllocator和pipeline。

### Channel工作原理

为什么不适用原生的Channel，而要使用 Netty的Channel?

1. JDK的SocketChannel和ServerSocketChannel没有统一的Channel接口供业务开发者使用
2. JDK的SocketChannel和ServerSocketChannel只要职责是网络I/O操作，由于它们是SPI接口，由具体的虚拟机厂家来实现，所以通过SPI功能类来扩展难度很大。
3. Netty的Channel需要能够跟Netty的整体框架融合在一起，例如I/O模型，基于ChannelPipeline的定制模型，以及基于元数据描述配置化的TCP参数等，这些JDK的SocketChannel和ServerSocketChannel都没有提功
4. 自定义的Channel更加灵活

Netty的Channel的设计理念：

1. 在Channel接口层，采用Facade模式进行统一封装，将网络I/O操作、网络I/O相关联的其他操作封装起来，对外统一提供
2. Channel接口的定义尽量大而全，为SocketChannel和ServerSocketChannel提供统一的视图，由不同子类实现不同功能。
3. 具体实现采用聚合而非包含，将相关的功能类聚合在Channel

## Channel功能

```
public interface Channel extends AttributeMap, ChannelOutboundInvoker,  
Comparable<Channel> {  
    //从当前的channel中读取数据到第一个inbound缓冲区中，如果数据被成功读取，触发  
    ChannelHandler.channelRead事件。读取操作API调用完成后，紧接着会触发  
    ChannelHandler.channelReadComplete事件，这样业务的ChannelHandler可以决定是否需要继续读  
    取数据。如果有读操作请求被挂起，则后续的读操作会被忽略  
    Channel read();  
    //将之前写入到发送环形数组中的消息全部写入到目标channel，发送通信对方  
    Channel flush();  
    //请求将当前的msg通过channelPipeline写入到目标channel。注意write操作只是将消息存入到消  
    息发送环形数组，并没有真正发送，只有调用flush操作才会被写入到channel  
    ChannelFuture write(Object msg);
```

```

//与ChannelFuture write(Object msg)类似，携带了ChannelPromise参数负责设置写入操作的结果
ChannelFuture write(Object msg, ChannelPromise promise);
//主动关闭当前连接，通过ChannelPromise设置操作结果并进行结果通知，无论是否成功，都可以通过ChannelPromise获取结果。该操作会级联出发ChannelPipeline中所有的ChannelHandler的ChannelHandler.close事件
ChannelFuture close(ChannelPromise promise);
//请求断开与远程通信对端的连接并使用ChannelPromise来获取操作结果的通知消息。该操作会级联出发ChannelPipeline中所有的ChannelHandler的ChannelHandler.disconnect事件
ChannelFuture disconnect(ChannelPromise promise);
//客户端使用指定的服务端地址remoteAddress发起连接请求，如果连接因为应答超时而失败，ChannelFuture中的操作结果就是ConnectTimeoutException。如果连接被拒绝，则是ConnectException。该方法会级联触发ChannelHandler.connect事件
ChannelFuture connect(SocketAddress remoteAddress);
//与ChannelFuture connect(SocketAddress remoteAddress);类似，不同的是需要先绑定本机的地址
ChannelFuture connect(SocketAddress remoteAddress, SocketAddress localAddress);
//绑定指定的本机socket地址，该方法级联触发ChannelHandler.bind事件
ChannelFuture bind(SocketAddress localAddress);
//获取当前Channel配置信息
ChannelConfig config();
//判断当前Channel是否打开
boolean isOpen();
//判断当前的Channel是否注册到EventLoop上
boolean isRegistered();
//判断当前Channel时候已经处于激活状态
boolean isActive();
//获取当前Channel的源数据描述信息，包括TCP参数信息
ChannelMetadata metadata();
//获取当前Channel的本机绑定地址
SocketAddress localAddress();
//获取当前Channel通信的远程Socket地址
SocketAddress remoteAddress();

//Channel需要注册到EventLoop的多路复用器，用于处理I/O事件，通过此方法可以获取到Channel注册的EventLoop
EventLoop eventLoop();
//对于服务端Channel而言，父Channel为空，对于客户端的Channel而言，父Channel是创建它的ServerSocketChannel
Channel parent();
/**
 * 获取Channel标识的id()，返回的ChannelId是channel的唯一标识，生成策略如下
 * 1. 假期的MAC地址等可以代表全局唯一的消息
 * 2. 当前进程ID
 * 3. 当前系统时间毫秒数
 * 4. 当前系统时间纳秒数
 * 5. 32位随机整型数
 * 6. 32位自增的序列数
 */
ChannelId id();
}

```

## Channel源码

服务端 NioServerSocketChannel 的继承关系类图如图 16-2 所示。

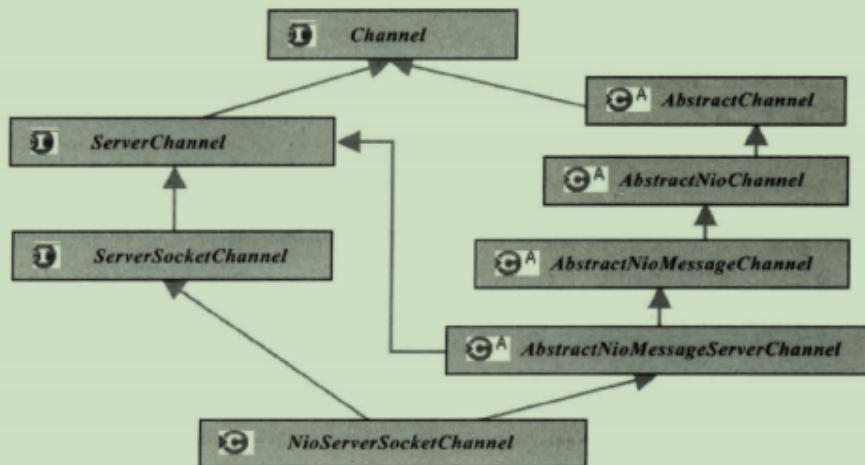


图 16-2 NioServerSocketChannel 继承关系类图

• 343 •

Netty 权威指南（第 2 版）

客户端 NioSocketChannel 的继承关系类图如图 16-3 所示。

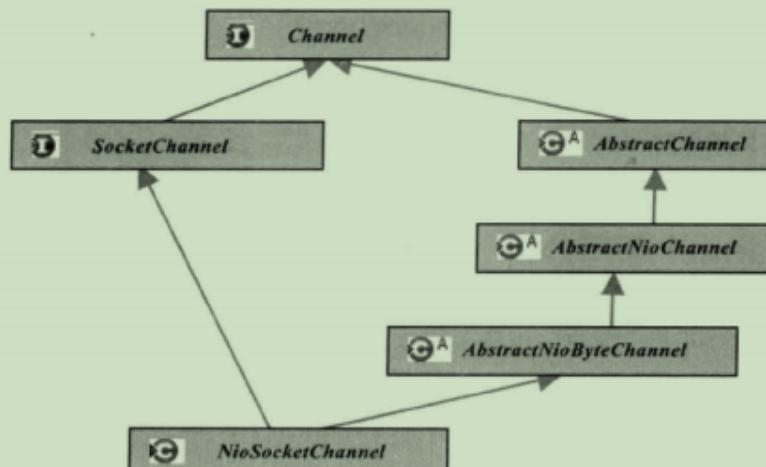


图 16-3 NioSocketChannel 继承关系类图

## Unsafe功能

Unsafe接口实际上是Channel接口的辅助接口，他不应该被用户代码直接调用，实际的I/O的写操作都是由unsafe接口负责完成的。

表 16-1 Unsafe API 功能列表

方法名	返回值	功能说明
invoker()	ChannelHandlerInvoker	返回默认使用的 ChannelHandlerInvoker
localAddress()	SocketAddress	返回本地绑定的 Socket 地址
remoteAddress()	SocketAddress	返回通信对端的 Socket 地址
register(ChannelPromise promise)	void	注册 Channel 到多路复用器上，一旦注册操作完成，通知 ChannelFuture
bind(SocketAddress localAddress, ChannelPromise promise)	void	绑定指定的本地地址 localAddress 到当前的 Channel 上，一旦完成，通知 ChannelFuture
connect(SocketAddress remoteAddress, SocketAddress localAddress, ChannelPromise promise)	void	绑定本地的 localAddress 之后，连接服务端，一旦操作完成，通知 ChannelFuture

• 364 •

## 第 16 章 Channel 和 Unsafe

续表

方法名	返回值	功能说明
disconnect(ChannelPromise promise)	void	断开 Channel 的连接，一旦完成，通知 ChannelFuture
close(ChannelPromise promise)	void	关闭 Channel 的连接，一旦完成，通知 ChannelFuture
closeForcibly()	void	强制立即关闭连接
beginRead()	void	设置网络操作位为读用于读取消息
write(Object msg, ChannelPromise promise)	void	发送消息，一旦完成，通知 ChannelFuture
flush()	void	将发送缓冲数组中的消息写入到 Channel 中
voidPromise()	ChannelPromise	返回一个特殊的可重用和传递的 ChannelPromise，它不用于操作成功或者失败的通知器，仅仅作为一个容器被使用
outboundBuffer()	ChannelOutboundBuffer	返回消息发送缓冲区

## ChannelPipeline和ChannelHandler

ChannelPipeline和ChannelHandler机制类似于Servlet和Filter过滤器，这类拦截器实际上就是一种变形的责任链模式，主要是为了时间的拦截和用户业务逻辑的定制。

Netty的Channel过滤器实现原理与Servlet Filter机制一致，它将Channel的数据管道抽象为ChannelPipeline，消息在ChannelPipeline中流动和传递。ChannelPipeline持有I/O事件拦截器ChannelHandler的链表，由ChannelHandler对I/O事件进行拦截和处理，可以方便的通过新增和删除ChannelHandler来实现不同的业务逻辑定制，不需要对已有的ChannelHandler进行修改，能够实现对修改封闭和对扩展的支持。

## ChannelPipeline功能

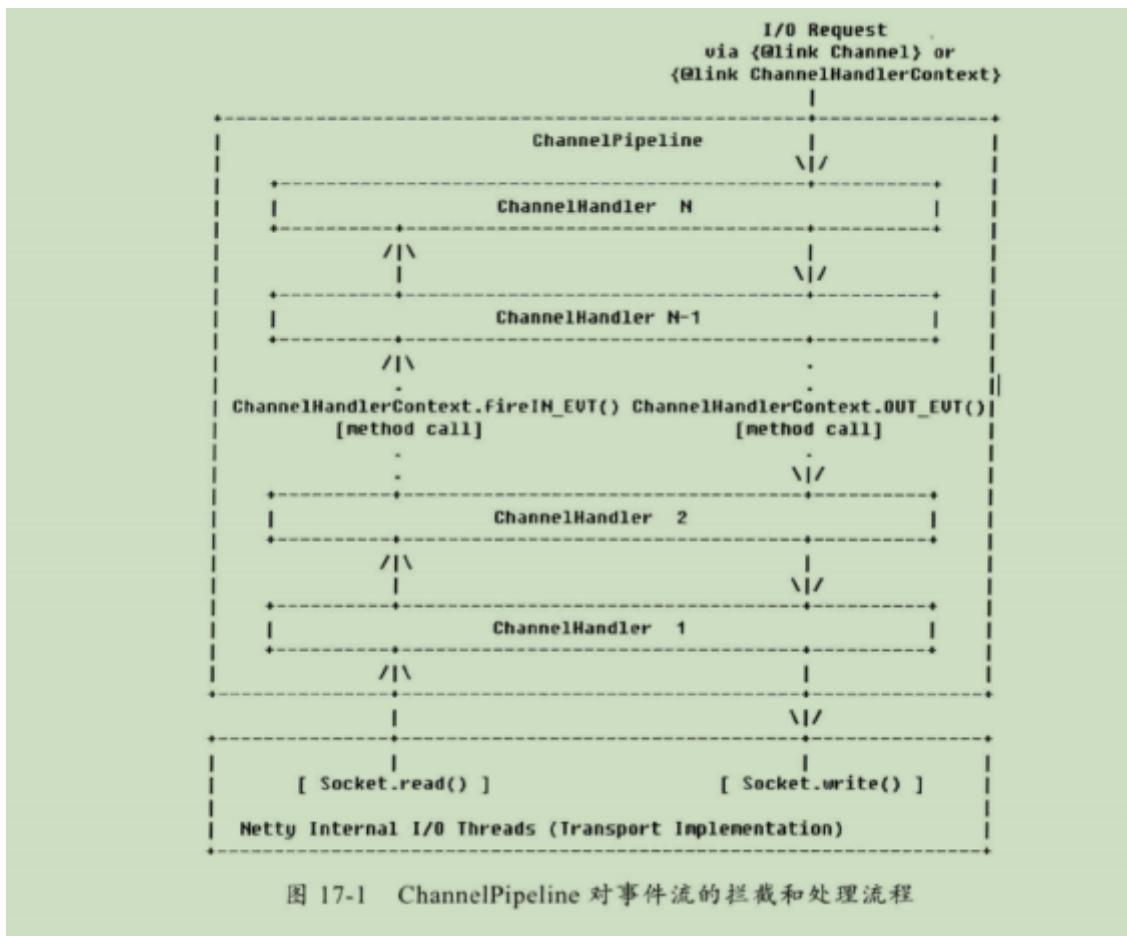


图 17-1 ChannelPipeline 对事件流的拦截和处理流程

```
* +-----+-----+
* |          ChannelPipeline          |   |
* |          \|\|   |
* | +-----+-----+-----+-----+-----+
* | | Inbound Handler N |       | Outbound Handler 1 |   |
* | +-----+-----+-----+-----+-----+
* | /|\|           |           \|\|   |
* | |           |           |           |   |
* | +-----+-----+-----+-----+-----+
* | | Inbound Handler N-1 |       | Outbound Handler 2 |   |
* | +-----+-----+-----+-----+-----+
* | /|\|           |           \|\|   |
* | |           |           |           |   |
* | +-----+-----+-----+-----+-----+
* | | Inbound Handler 2 |       | Outbound Handler M-1 |   |
* | +-----+-----+-----+-----+-----+
* | /|\|           |           \|\|   |
* | |           |           |           |   |
* | +-----+-----+-----+-----+-----+
* | | Inbound Handler 1 |       | Outbound Handler M |   |
* | +-----+-----+-----+-----+-----+
* | /|\|           |           \|\|   |
* | |           |           |           |   |
* +-----+-----+-----+-----+-----+
```



该图展示了一个消息被ChannelPipeline的ChannelHandler链拦截和处理的全过程，消息的读取和发送处理全流程为：

1. 底层的SocketChannel read()方法读取ByteBuf，触发ChannelRead事件，由I/O线程NioEventLoop调用ChannelPipeline的fireChannelRead () 方法，将消息传输到ChannelPipeline中。
2. 消息依次被ChannelHandler链中的ChannelHandler拦截处理，整个过程中，任何ChannelHandler都可以中断当前的流程，结束消息的传递
3. 调用ChannelHandlerContext的write方法发送消息，消息从最后一个ChannelHandler开始，经过整个过滤器链，最终被添加到消息发送缓冲区中等到刷新和发送，在此过程中也可以中断消息的传递，例如编码失败时，需要中断流程，构造异常的Future返回。

Netty中的事件分为Inbound事件和outbound事件。

Inbound时间通常由I/O线程触发，例如TCP链路建立、链路关闭事件、读事件、异常通知事件等

触发Inbound事件的方法如下：

1. ChannelHandlerContext.fireChannelRegistered(); Channel注册事件
2. ChannelHandlerContext.fireChannelActive () ; TCP链路建立成功，Channel激活事件
3. ChannelHandlerContext.fireChannelRead (Obj) ; 读事件
4. ChannelHandlerContext.fireChannelReadComplete () ; 读操作完成通知事件
5. ChannelHandlerContext.fireExceptionCaught (Throwable) 异常通知时间
6. ChannelHandlerContext.fireUserEventTriggered (Obj) ; 用户自定义事件
7. ChannelHandlerContext.fireChannelWritabilityChanged () ; Channel的可写状态变化通知事件
8. ChannelHandlerContext.fireChannelInactive () TCP连接关闭，链路不可用通知事件

outbound事件通常由用户主动发起的网络I/O操作触发。例如：用户发起的连接操作、绑定操作、消息发送等操作

1. ChannelHandlerContext.bind () 绑定本地地址事件
2. ChannelHandlerContext.connect () 连接服务端事件
3. ChannelHandlerContext.write () 发送事件
4. ChannelHandlerContext.flush () 刷新事件
5. ChannelHandlerContext.read () 读事件
6. ChannelHandlerContext.disconnect () 断开连接事件
7. ChannelHandlerContext.close () 关闭当前Channel事件

## 自定义拦截器

通常ChannelHandler只需继承ChannelHandlerAdapter类覆盖自己关心的方法即可。

## 构建pipeLine

用户不需要自己创建pipeline，因为使用ServerBootstrap或者Bootstrap启动服务端或客户端时，Netty会为每个Channel连接创建一个独立的pipeline。我们只需将自定义的拦截器加入到Pipeline中。

```

pipeline = ch.pipeline();
pipeline.addLast("decoder", new MyProtocolDecoder());
pipeline.addLast("encoder", new MyProtocolEncoder());

```

## ChannelPipeline的主要特性

ChannelPipeline支持运行动态的添加或者删除ChannelHandler，在某些场景下这个特性非常实用。

ChannelPipeline是线程安全的，意味着N个业务线程可以并发的操作ChannelPipeline，而不会存在多线程并发问题。但是ChannelHandler不是线程安全的，所以用户需要自己保证ChannelHandler的线程安全。

## ChannelPipeline源码分析

ChannelPipeline实际上是一个ChannelHandler的容器，内部维护了一个ChannelHandler的链表和迭代器，可以方便的实现ChannelHandler的查找、添加、替换和删除。

### 17.2.1 ChannelPipeline 的类继承关系图

ChannelPipeline 的类继承关系比较简单，如图 17-3 所示。

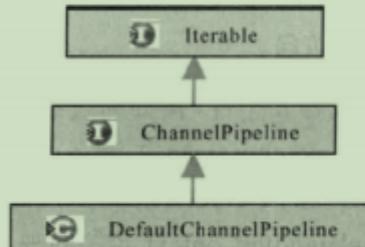


图 17-3 ChannelPipeline 类继承关系图

### ChannelPipeline对ChannelHandler的管理

```
//由于ChannelPipeline支持运行期动态修改，因此存在两种潜在的多线程并发访问场景
//1.I/O线程和用户业务线程的并发访问
//2.用户多个线程之间的并发访问
//为了保证ChannelPipeline的线程安全性，需要通过线程安全容器或者锁来保证并发访问的安全，Netty
//使用了synchronized关键字，保证同步块内的所有操作的原子性。
public synchronized void addBefore(String baseName, String name, ChannelHandler
handler) {
    DefaultChannelHandlerContext ctx = getContextOrDie(baseName);
    if (ctx == head) {
        addFirst(name, handler);
    } else {
        checkDuplicateName(name);
        DefaultChannelHandlerContext newCtx = new
DefaultChannelHandlerContext(ctx.prev, ctx, name, handler);

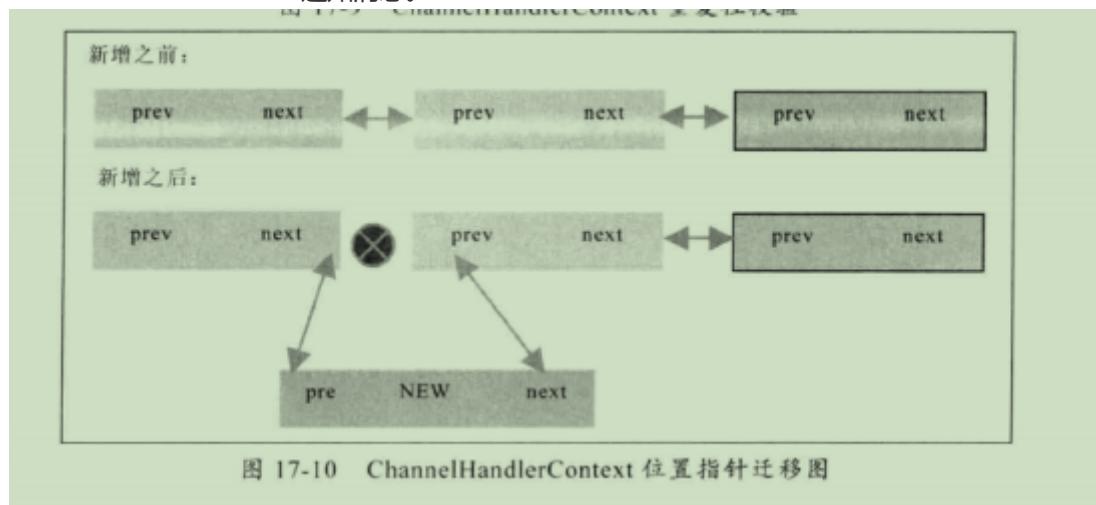
        callBeforeAdd(newCtx);

        ctx.prev.next = newCtx;
        ctx.prev = newCtx;
        name2ctx.put(name, newCtx);

        callAfterAdd(newCtx);
    }
}
```

对新增的ChannelHandler名进行重复性校验，如果已有同名的ChannelHandler，抛出异常。校验通过后，使用新增的ChannelHandler等参数构造一个新的DefaultChannelHandlerContext。将新创建的DefaultChannelHandlerContext添加到当前的pipeline中。首先对添加的ChannelHandlerContext做重复性校验，如果ChannelHandlerContext不是可以在多个ChannelPipeline中共享的，且已经被添加到

ChannelPipeline中，则抛出异常。加入成功之后，缓存ChannelHandlerContext，发送新增ChannelHandlerContext通知消息。



### ChannelPipeline中的inbound事件

当发生某个I/O事件时，都会产生一个事件，事件在pipeline中传播和处理，它是事件的总入口。由于网络的I/O相关的事件有限，因此Netty对这些事件进行了统一抽象，Netty自身和用户的ChannelHandler对感兴趣的事件进行拦截和处理。

pipeline中以fireXXX命名的方法都是从I/O线程流向用户业务Handler的inbound事件，他们虽功能不同，但处理步骤类似：

1. 调用HeadHandler对应的fireXXX命名
2. 执行事件相关的逻辑操作

### ChannelPipeline的outBound事件

用户线程或代码发起的I/O操作被称为outBound事件，事实上inBound和outBound是Netty自身根据事件在pipeline中的流向抽象出来的术语。

Pipeline本身并不直接进行I/O操作，I/O操作最终是由Unsafe和Channel来实现的。Pipeline负责将I/O事件通过TailHandler进行调度和传播，最终调用Unsafe的I/O方法进行操作。

## ChannelHandler功能

ChannelHandler主要类似于Servlet的Filter过滤器，负责对I/O事件或者I/O操作进行拦截和处理，可以选择性的拦截和处理感兴趣的事件，也可以透传或终止事件的传递。

ChannelHandler支持注解，目前支持两种注解

1. Sharable：多个ChannelPipeline公用同一个ChannelHandler
2. Skip：被Skip注解的方法不会被调用，直接被忽略

### ChannelHandlerAdapter

Netty提供了ChannelHandlerAdapter基类，它的所有接口实现都是事件透传，如果用户ChannelHandler关心某个事件，只需覆盖ChannelHandlerAdapter对应的方法，对于不关心的，可以直接继承使用父类的方法。这样就不需要每个用户的ChannelHandler都要实现ChannelHandler的所有接口。

## **ByteToMessageDecoder**

用户的编码器继承ByteToMessageDecoder，只需要实现void decode(ChannelHandlerContext ctx, ByteBuf in, List out)抽象方法即可完成ByteBuf到POJO对象的编码。

但是ByteToMessageDecoder并没有考虑TCP粘包和拆包的场景，需用户自己实现。

## **MessageToMessageDecoder**

MessageToMessageDecoder是Netty的二次编码器，职责是将一个对象二次解码为其他对象。

从SocketChannel读取到的TCP数据包时ByteBuf，就是字节数组，我们首先需要将ByteBuf缓冲区中的数据读取出来，并解码为Java对象；然后对Java对象根据某些规则进行二次解码，将其解码成另一个POJO对象，因为MessageToMessageDecoder在ByteToMessageDecoder之后，所以称之为二次解码器。

用户的解码器只需实现void decode (ChannelHandlerContext ctx, I msg, List out) 抽象方法即可。因为它是将一个POJO解码为另一个POJO，所以一般不会涉及到半包的处理。

## **LengthFieldBasedFrameDecoder**

通常由四种做法来区分一个整包消息：

1. 固定长度，例如每120个字节代表一个整包消息，不足的前面补0。
2. 通过回车换行符区分消息
3. 通过分隔符区分整包消息
4. 通过指定长度来标识整包消息

如果消息时按照长度区分的，LengthFieldBasedFrameDecoder可以自定处理粘包和半包问题，只需要传入正确的参数。

## **其他功能**

请自行百度

# **EventLoop和EventLoopGroup**

## **Netty的线程模型**

### **Reactor单线程模型**

Reactor单线程模型，是指所有的I/O操作都在同一个NIO线程上面完成，NIO线程职责如下：

1. 作为NIO服务端，接受客户端的TCP连接
2. 作为NIO客户端，向服务端发起TCP连接
3. 读取通信对端的请求或者应答消息
4. 向通信对端发送消息请求或应答消息

由于Reactor模式使用的是异步非阻塞I/O，所有的I/O操作都不会导致阻塞，理论上一个线程可以独立处理所有的I/O操作。

但是这种模型不适用高负载、大并发的应用场景：

- 一个NIO线程同时处理成百上千的链路，性能上无法支撑
- 当NIO线程负载过重之后，处理速度变慢，会导致大量客户端连接超时，超时后会进行重发，更加重了NIO线程的负载，最终导致消息的大量积压和处理超时，成为系统的瓶颈
- 一旦NIO线程意外断开，或者进入死循环，会导致整个系统通信模块不可用，不能接受和处理消息，造成节点故障

Reactor 单线程模型如图 18-1 所示。

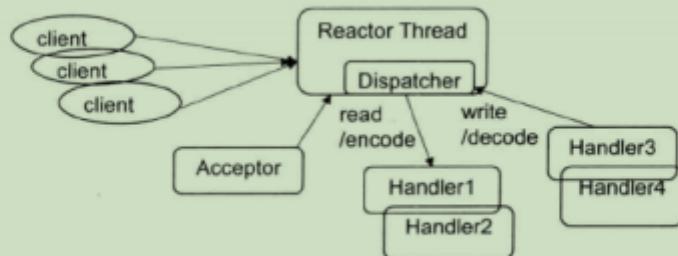


图 18-1 Reactor 单线程模型

### Reactor多线程模型

Reactor多线程特点：

- 有一个专门的NIO线程-Acceptor线程用于监听服务端，接受客户端的TCP连接请求
- 网络I/O操作--读写等由一个NIO线程池负责，线程池可以采用标准的DK线程池实现，包含一个任务队列和N个可用的线程，由这些NIO线程负责消息的读取、编码、解码和发送
- 一个NIO线程可以同时处理N条链路，但是一个链路只对应一个NIO线程，防止发生并发操作问题

Reactor 多线程模型与单线程模型最大的区别就是有一组 NIO 线程来处理 I/O 操作，它原理如图 18-2 所示。

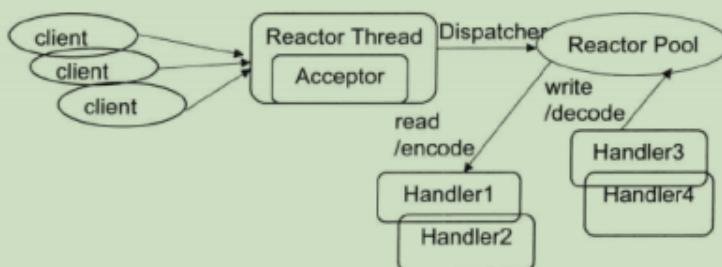


图 18-2 Reactor 多线程模型

### 主从Reactor多线程模型

主从Reactor多线程模型特点：

服务端用于接收客户端连接的不再是一个单独的NIO线程，而是一个独立的NIO线程池。Acceptor接收到客户端的TCP连接请求并处理完成后，将新建的SocketChannel注册到I/O线程池（sub reactor线程池）的某个线程上，由它负责SocketChannel的编解码工作。Acceptor线程池仅仅用户客户端的登录、握手和安全认证，一但链路建立成功，就将链路注册到后端subReactor线程池的I/O线程上，由I/O线程负责后续的I/O操作。

它的线程模型如图 18-3 所示。

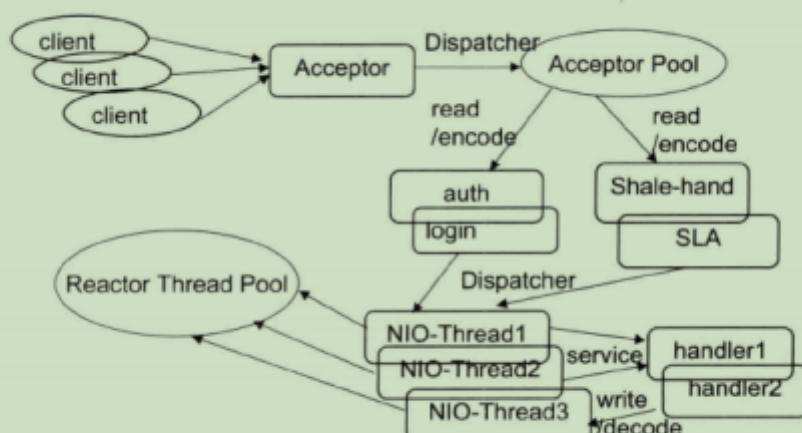


图 18-3 主从 Reactor 多线程模型

在Netty官方文档中，推荐使用该模型。

## Netty的线程模型

Netty的线程模型取决于用户的启动参数配置，通过设置不同的参数，Netty可以同时支持Reactor单线程模型，Reactor多线程模型和主从多线程模型。

服务端启动的时候，创建了两个NioEventLoopGroup，它们是两个独立的Reactor线程池，一个用于接收客户端的TCP连接，一个用于处理I/O相关的读写操作，或者执行系统Task、定时任务Task等。

Netty用于接收客户端请求的线程池职责如下：

1. 接收客户端TCP连接，初始化Channel参数
2. 将链路状态变更事件通知给ChannelPipeline

Netty处理I/O操作的Reactor线程池职责如下：

1. 异步读取通信对端的数据包，发送读事件到ChannelPipeline
2. 异步发送消息到通信对端，调用ChannelPipeline的消息发送接口
3. 执行系统调用Task
4. 执行定时任务Task

Netty的NioEventLoop读取到消息后，直接调用ChannelPipeline的fireChannelRead()。只要用户不主动切换线程，一直都是由NioEventLoop调用用户的Handler，期间不进行线程切换。这种串行化的处理方式避免了多线程导致的锁竞争。

## NioEventLoop源码分析

### NioEventLoop

```
/**  
 * The NIO {@link Selector}.  
 */  
private Selector selector; //多路复用器对象  
private Selector unwrappedSelector;  
private SelectedSelectionKeySet selectedKeys;  
  
private final SelectorProvider provider;
```

Selector的初始化非常简单，直接调用Selector.open()方法就能创建并打开一个新的Selector。Netty对Selector的selectedKeys进行了优化，用户可以通过io.netty.noKeySetOptimization开关决定是否启用该优化项。默认不打开selectedKeys的优化项。

Selector初始化：

```
private SelectorTuple openSelector() {  
    final Selector unwrappedSelector;  
    try {  
        unwrappedSelector = provider.openSelector();  
    } catch (IOException e) {  
        throw new ChannelException("failed to open a new selector", e);  
    }  
  
    if (DISABLE_KEYSET_OPTIMIZATION) {  
        return new SelectorTuple(unwrappedSelector);  
    }  
}
```

```
object maybeSelectorImplClass = AccessController.doPrivileged(new  
PrivilegedAction<Object>() {  
    @Override  
    public Object run() {  
        try {  
            return Class.forName(  
                "sun.nio.ch.SelectorImpl",  
                false,  
                PlatformDependent.getSystemClassLoader());  
        } catch (Throwable cause) {  
            return cause;  
        }  
    }  
});
```

如果没有打开selectedKeys优化开关，通过provider.openSelector()创建并打开多路复用器后直接返回。

如果开启了优化开关，需要通过反射的方式从Selector实例中获取selectedKeys和publicSelectedKeys，将上述的两个成员变量设置为可写，通过反射的方式使用Netty构造的selectedKeys包装类selectedKeySet将原JDK的selectedKeys替换。

```
int selectNow() throws IOException {  
    try {  
        return selector.selectNow();  
    } finally {  
        // restore wakeup state if needed  
        if (wakenUp.get()) {  
            selector.wakeup();  
        }  
    }  
}
```

selectNow()方法会立即出发Selector的选择操作，如果有准备就虚的Channel，直接返回就虚的Channel集合，否则返回0.选择完成后，判断用户是否调用了selector.wakeup()，如果调用了，则执行wakeup()操作

## Future和Promise

### Future功能

表 19-1 ChannelFuture 接口列表

返回值	方法名称
ChannelFuture	addListener(GenericFutureListener<? extends Future<? super java.lang.Void>> listener) Adds the specified listener to this future
ChannelFuture	addListeners(GenericFutureListener<? extends Future<? super java.lang.Void>>... listeners) Adds the specified listeners to this future
ChannelFuture	await() Waits for this future to be completed
ChannelFuture	awaitUninterruptibly() Waits for this future to be completed without interruption
Channel	channel() Returns a channel where the I/O operation associated with this future takes place
ChannelFuture	removeListener(GenericFutureListener<? extends Future<? super java.lang.Void>> listener) Removes the specified listener from this future

• 439 •

Netty 权威指南（第 2 版）

续表

返回值	方法名称
ChannelFuture	removeListeners(GenericFutureListener<? extends Future<? super java.lang.Void>>... listeners) Removes the specified listeners from this future
ChannelFuture	sync() Waits for this future until it is done, and rethrows the cause of the failure if this future failed
ChannelFuture	syncUninterruptibly() Waits for this future until it is done, and rethrows the cause of the failure if this future failed

ChannelFuture有两种状态：uncomplete和complete。当开始一个I/O操作时，一个新的ChannelFuture被创建，此时处于uncomplete状态。一旦I/O操作完成后，ChannelFuture就会被设置为complete。此时有可能会有三个结果：操作成功、操作失败、操作被取消。

ChannelFuture可以同时增加一个或者多个GenericFutureListener，也可以用remove方法删除GenericFutureListener。

GenericFutureListener 的接口定义如图 19-4 所示。

```
void operationComplete(F future) throws Exception;
```

图 19-4 GenericFutureListener 接口定义

当I/O操作完成之后，I/O线程会回调ChannelFuture中GenericFutureListener的operationComplete方法，并把ChannelFuture对象当作方法的入参。用户需要做上下文相关的操作，需要将上下文信息保存到对应的ChannelFuture中。

推荐通过 GenericFutureListener 代替 ChannelFuture 的 get 等方法的原因是：当我们进行异步 I/O 操作时，完成的时间是无法预测的，如果不设置超时时间，它会导致调用线程长时间被阻塞，甚至挂死。而设置超时时间，时间又无法精确预测。利用异步通知机制回调 GenericFutureListener 是最佳的解决方案，它的性能最优。

需要注意的是：不要在 ChannelHandler 中调用 ChannelFuture 的 await()方法，这会导致死锁。原因是发起 I/O 操作之后，由 I/O 线程负责异步通知发起 I/O 操作的用户线程，如果 I/O 线程和用户线程是同一个线程，就会导致 I/O 线程等待自己通知操作完成，这就导致了死锁，这跟经典的两个线程互等待死锁不同，属于自己把自己挂死。

## Promise介绍

Promise时可写的Future，Future自身并没有写操作相关的接口。Netty通过Promise对Future进行扩展，用于设置I/O操作的结果。

<code>Promise&lt;V&gt;</code>	<code>setFailure(java.lang.Throwable cause)</code> Marks this future as a failure and notifies all listeners.
<code>Promise&lt;V&gt;</code>	<code>setSuccess(V result)</code> Marks this future as a success and notifies all listeners.
<code>boolean</code>	<code>setUncancellable()</code> Make this future impossible to cancel.
<code>Promise&lt;V&gt;</code>	<code>sync()</code> Waits for this future until it is done, and rethrows the cause of the failure if this future failed.
<code>Promise&lt;V&gt;</code>	<code>syncUninterruptibly()</code> Waits for this future until it is done, and rethrows the cause of the failure if this future failed.
<code>boolean</code>	<code>tryFailure(java.lang.Throwable cause)</code> Marks this future as a failure and notifies all listeners.
<code>boolean</code>	<code>trySuccess(V result)</code> Marks this future as a success and notifies all listeners.

图 19-13 Promise 写操作相关的接口定义

Netty发起I/O操作的时候，会创建一个新的Promise对象。

当I/O操作发生异常或者完成时，设置Promise的结果。

# Netty高级特性

## Netty架构

Netty采用了典型的三层网络架构

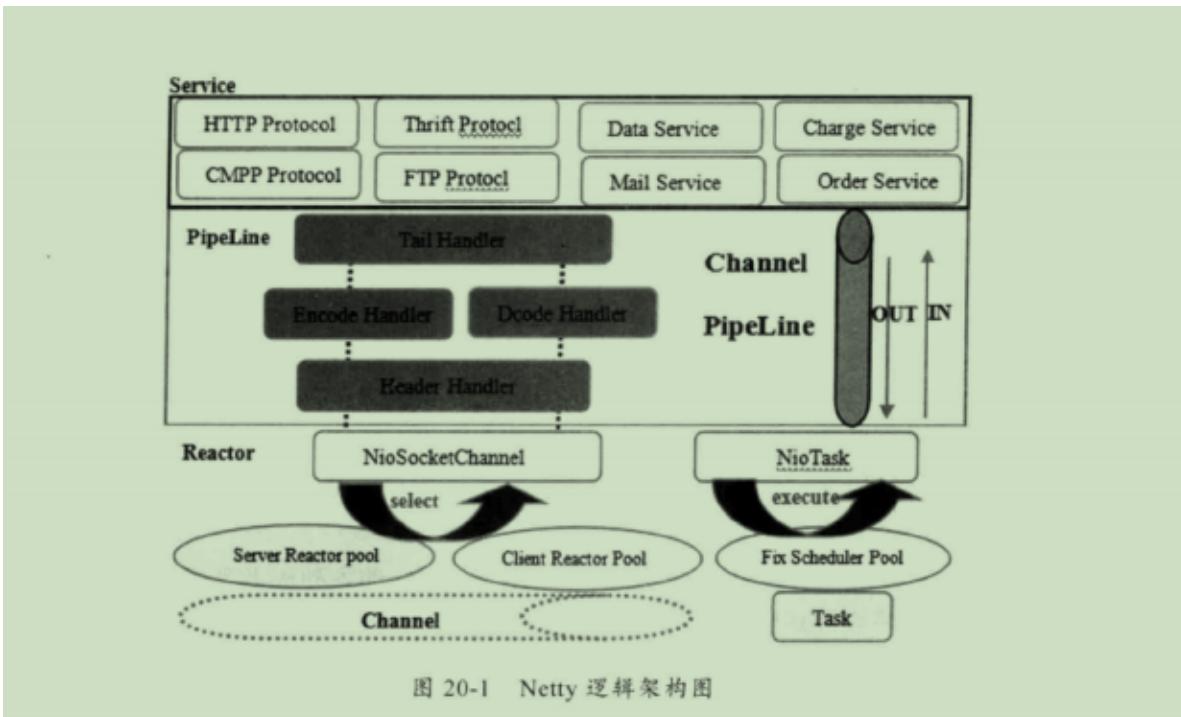


图 20-1 Netty 逻辑架构图

## Reactor通信调度层：

它由一系列辅助类完成，包括Reactor线程NioEventLoop及其父类，  
NioSocketChannel/NioServerSocketChannel及其父类，ByteBuf及其衍生出来的各类Buffer，Unsafe  
及其衍生出来的各种内部类。

该层主要职责就是监听网络的读写和连接操作，负责将网络层的数据读取到内存缓冲区中，然后触发各类网络事件。例如连接创建、连接激活、读事件、写事件等，将这些事件触发到pipeline中，由Pipeline管理的职责链来进行后续的处理。

## 职责链ChannelPipeline

负责事件在职责链中的有序传播，同时负责动态的编排职责链，职责链可以选择监听和处理自己关心的事件，它可以拦截处理向前后传播的事件。不同应用的Handler节点功能也不同。

## 业务逻辑编排层 (Service ChannelHandler)

业务逻辑编排层通常有两类：

1. 纯粹的业务逻辑编排
2. 其他应用层的协议插件

## 关键架构

### 高性能

Netty高性能的实现

1. 采用异步非阻塞的I/O类库，基于Reactor模式实现，解决了传统同步阻塞I/O模式下一个服务端无法平滑的处理线性增长的客户端问题。
2. TCP接收和发送缓冲区使用直接内存代替堆内存，避免内存复制，提升I/O读取和写入的性能。
3. 支持通过内存池的方式循环利用ByteBuf，避免频繁创建销毁ByteBuf带来的性能消耗
4. 可配置的I/O线程数、TCP参数等，为不同场景提供不同的调优参数
5. 采用环形数组缓冲区实现无锁化并发编程
6. 合理地使用线程安全容器、原子类
7. 关键资源的处理使用单线程串行化的方式，避免多线程并发访问带来的锁竞争和额外的CPU资源消耗

8. 通过引用计数器及时的申请释放不再被引用的对象，细粒度的内存管理降低了GC的频率，减少了频繁GC带来的延时增大和CPU消耗

## 可靠性

链路有效性检测

Netty的两种链路空闲检测机制

- 读空闲超时机制：当连续周期T没有消息可读时，触发超时Handler，用户可以基于读空闲超时发送心跳消息，进行链路检测：如果连续N个周期仍然没有读取到心跳消息，主动关闭链路
- 写空闲超时机制：当连续周期T没有消息要发送时，触发超时Handler，用户可以基于写空闲超时发送心跳消息，进行链路检测：如果连续N个周期仍然没有收到对方的心跳消息，主动关闭链路

Netty提供了空闲状态检测事件通知机制，用户可以订阅空闲超时事件、写空闲超时事件、读或者写超时事件、接收到对应的空闲事件之后，灵活的进行处理

内存保护机制

1. 通过对对象引用计数器对Netty的ByteBuf等内置对象进行细粒度的内存申请和释放，对非法的对象引用进行检测和保护
2. 通过内存池来重用ByteBuf，节省内存
3. 可设置的内存容量上限，包括ByteBuf、线程池线程数等

优雅停机

相比于 Netty 的早期版本，Netty 5.0 版本的优雅退出功能做得更加完善。优雅停机功能指的是当系统退出时，JVM 通过注册的 Shutdown Hook 拦截到退出信号量，然后执行退出操作，释放相关模块的资源占用，将缓冲区的消息处理完成或者清空，将待刷新的数据持久化到磁盘或者数据库中，等到资源回收和缓冲区消息处理完成之后，再退出。

优雅停机往往需要设置个最大超时时间  $T$ ，如果达到  $T$  后系统仍然没有退出，则通过 Kill - 9 pid 强杀当前的进程。

Netty 所有涉及到资源回收和释放的地方都增加了优雅退出的方法，它们的相关接口如表 20-1 所示。

表 20-1 Netty 重要资源的优雅退出方法

EventExecutorGroup.shutdownGracefully()	NIO 线程优雅退出
EventExecutorGroup.shutdownGracefully(long quietPeriod, long timeout, TimeUnit unit)	NIO 线程优雅退出，支持设置超时时间
Channel.close()	Channel 的关闭
Unsafe.close(ChannelPromise promise)	Unsafe 的关闭操作，可以设置可写的 Future
Unsafe.closeForcibly()	Unsafe 的强制关闭操作

• 459 •

Netty 权威指南（第 2 版）

续表

ChannelPipeline.close()	ChannelPipeline 的关闭
ChannelPipeline.close(ChannelPromise promise)	ChannelPipeline 的关闭，可以设置可写的 Future
ChannelHandler.close(ChannelHandlerContext ctx, ChannelPromise promise)	ChannelHandler 的关闭

### 20.2.3 可定制性

Netty 的可定制性主要体现在以下几点。

- ◎ 责任链模式：ChannelPipeline 基于责任链模式开发，便于业务逻辑的拦截、定制和扩展。
- ◎ 基于接口的开发：关键的类库都提供了接口或者抽象类，如果 Netty 自身的实现无法满足用户的需求，可以由用户自定义实现相关接口。
- ◎ 提供了大量工厂类，通过重载这些工厂类可以按需创建出用户实现的对象。
- ◎ 提供了大量的系统参数供用户按需设置，增强系统的场景定制性。

### 20.2.4 可扩展性

基于 Netty 的基础 NIO 框架，可以方便地进行应用层协议定制，例如 HTTP 协议栈、Thrift 协议栈、FTP 协议栈等。这些扩展不需要修改 Netty 的源码，直接基于 Netty 的二进制类库即可实现协议的扩展和定制。

目前，业界存在大量的基于 Netty 框架开发的协议，例如基于 Netty 的 HTTP 协议、Dubbo 协议、RocketMQ 内部私有协议等。

