Netty 面试题

1、NIO 的组成？

Buffer：与 Channel 进行交互，数据是从 Channel 读入缓冲区，从缓冲区写入 Channel 中的。

flip 方法 ：反转此缓冲区，将 position 给 limit，然后将 position 置为 0，其实就是切换读写模式。

clear 方法 ：清除此缓冲区，将 position 置为 0，把 capacity 的值给 limit。

rewind 方法 ：重绕此缓冲区，将 position 置为 0。

DirectByteBuffer 可减少一次系统空间到用户空间的拷贝。但 Buffer 创建和销毁的成本更高，不可控，通常会用内存池来提高性能。直接缓冲区主要分配给那些易受基础系统的本机 I/O 操作影响的大型、持久的缓冲区。如果数据量比较小的中小应用情况下，可以考虑使用 heapBuffer，由 JVM 进行管理。

Channel：表示 IO 源与目标打开的连接，是双向的，但不能直接访问数据，只能与 Buffer 进行交互。通过源码可知，FileChannel 的 read 方法和 write 方法都导致数据复制了两次。

Selector 可使一个单独的线程管理多个 Channel，open 方法可创建 Selector，register 方法向多路复用器器注册通道，可以监听的事件类型：读、写、连接、accept。注册事件后会产生一个 SelectionKey：它表示

SelectableChannel 和 Selector 之间的注册关系。

wakeup 方法：使尚未返回的第一个选择操作立即返回。

原因是：注册了新的 channel 或者事件；channel 关闭，取消注册；优先级更高的事件触发（如定时器事件），希望及时处理。

Selector 在 Linux 的实现类是 EPollSelectorImpl，委托给 EPollArrayWrapper 实现，其中三个 native 方法是对 epoll 的封装，而 EPollSelectorImpl、implRegister 方法，通过调用 epoll_ctl 向 epoll 实例中注册事件，还将注册的文件描述符(fd)与 SelectionKey 的对应关系添加到 fdToKey 中，这个 map 维护了文件描述符与 SelectionKey 的映射。

fdToKey 有时会变得非常大，因为注册到 Selector 上的 Channel 非常多（百万连接）；过期或失效的 Channel 没有及时关闭。fdToKey 总是串行读取的，而读取是在 select 方法中进行的，该方法是非线程安全的。

Pipe：两个线程之间的单向数据连接，数据会被写到 sink 通道，从 source 通道读取

NIO 的服务端建立过程：Selector.open()：

打开一个 Selector；ServerSocketChannel.open()：创建服务端的 Channel；

bind()：绑定到某个端口上。并配置非阻塞模式；

register()：注册 Channel 和关注的事件到 Selector 上；

select()轮询拿到已经就绪的事件。

2、Netty 的线程模型？

Netty 通过 Reactor 模型基于多路复用器接收并处理用户请求，内部实现了两个线程池，boss 线程池和 work 线程池，其中** boss 线程池**的线程负责处理请求的 accept 事件，当接收到 accept 事件的请求时，把对应的 socket 封装到一个 NioSocketChannel 中，并交给 work 线程池，其中 work 线程池负责请求的 read 和 write 事件，由对应的 Handler 处理。

单线程模型：所有 I/O 操作都由一个线程完成，即多路复用、事件分发和处理都是在一个 Reactor 线程上完成的。既要接收客户端的连接请求,向服务端发起连接，又要发送/读取请求或应答/响应消息。一个 NIO 线程同时处理成百上千的链路，性能上无法支撑，速度慢，若线程进入死循环，整个程序不可用，对于高负载、大并发的应用场景不合适。

多线程模型：有一个 NIO 线程（Acceptor） 只负责监听服务端，接收客户端的 TCP 连接请求；NIO 线程池负责网络 IO 的操作，即消息的读取、解码、编码和发送；1 个 NIO 线程可以同时处理 N 条链路，但是 1 个链路只对应 1 个 NIO 线程，这是为了防止发生并发操作问题。但在并发百万客户端

连接或需要安全认证时，一个 Acceptor 线程可能会存在性能不足问题。

主从多线程模型：Acceptor 线程用于绑定监听端口，接收客户端连接，将 SocketChannel 从主线程池的 Reactor 线程的多路复用器上移除，重新注册到 Sub 线程池的线程上，用于处理 I/O 的读写等操作，从而保证 mainReactor 只负责接入认证、握手等操作。

3、Netty 核心组件有哪些？分别有什么作用？

Channel

Channel 接口是 Netty 对网络操作抽象类，它除了包括基本的 I/O 操作，如 bind()、connect()、read()、write() 等。
比较常用的 Channel 接口实现类是 NioServerSocketChannel（服务端）和 NioSocketChannel（客户端），这两个 Channel 可以和 BIO 编程模型中的 ServerSocket 以及 Socket 两个概念对应上。Netty 的 Channel 接口所提供的 API，大大地降低了直接使用 Socket 类的复杂性。

EventLoop

EventLoop（事件循环）接口可以说是 Netty 中最核心的概念了，
EventLoop 的主要作用实际就是负责监听网络事件并调用事件处理器进行相关 I/O 操作的处理。

那 Channel 和 EventLoop 有什么联系呢？

Channel 为 Netty 网络操作(读写等操作)抽象类，EventLoop 负责处理注册到其上的 Channel 处理 I/O 操作，两者配合参与 I/O 操作。

ChannelFuture

Netty 是异步非阻塞的，所有的 I/O 操作都为异步的。

因此，我们不能立刻得到操作是否执行成功，但是，可以通过 ChannelFuture 接口的 addListener() 方法注册一个 ChannelFutureListener，当操作执行成功或者失败时，监听就会自动触发返回结果。

并且，还可以通过 ChannelFuture 的 channel() 方法获取关联的 Channel

```
public interface ChannelFuture extends Future<Void> {
    Channel channel();

    ChannelFuture addListener(GenericFutureListener<? extends Future<? super Void>>
var1);
     ......

    ChannelFuture sync() throws InterruptedException;
}
```

另外，还可以通过 ChannelFuture 接口的 sync()方法让异步的操作变成同步的。

ChannelHandler 和 ChannelPipeline

下面这段代码使用过 Netty 的小伙伴应该不会陌生，我们指定了序列化编解码器以及自定义的 ChannelHandler 处理消息。

```
b.group(eventLoopGroup)
        .handler(new ChannelInitializer<SocketChannel>() {
            @Override
            protected void initChannel(SocketChannel ch) {
                ch.pipeline().addLast(new NettyKryoDecoder(kryoSerializer,
RpcResponse.class));
                ch.pipeline().addLast(new NettyKryoEncoder(kryoSerializer,
RpcRequest.class));
                ch.pipeline().addLast(new KryoClientHandler());
            }
        });
```

ChannelHandler 是消息的具体处理器。他负责处理读写操作、客户端连接等事情。

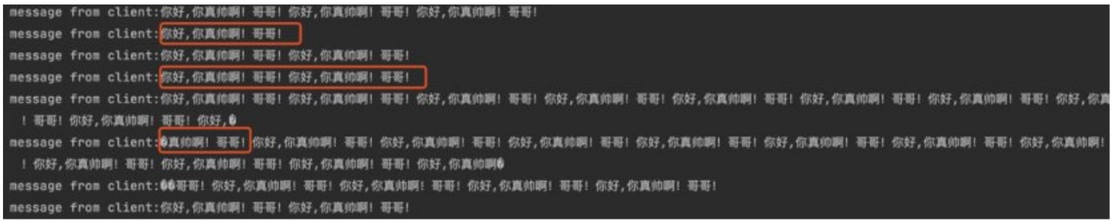
ChannelPipeline 为 ChannelHandler 的链，提供了一个容器并定义了用于沿着链传播入站和出站事件流的 API 。当 Channel 被创建时，它会被自动地分配到它专属的 ChannelPipeline。

可以在 ChannelPipeline 上通过 addLast() 方法添加一个或者多个 ChannelHandler ，因为一个数据或者事件可能会被多个 Handler 处理。当一个 ChannelHandler 处理完之后就将数据交给下一个 ChannelHandler 。

4、什么是 TCP 粘包/拆包?有什么解决办法呢？

TCP 粘包/拆包 就是你基于 TCP 发送数据的时候，出现了多个字符串“粘”在了一起或者一个字符串被“拆”开的问题。比如你多次发送：“你好,你真帅

啊！哥哥！” ，但是客户端接收到的可能是下面这样的：



解决方法如下：

使用 Netty 自带的解码器

LineBasedFrameDecoder：发送端发送数据包的时候，每个数据包之间以换行符作为分隔，LineBasedFrameDecoder 的工作原理是它依次遍历 ByteBuf 中的可读字节，判断是否有换行符，然后进行相应的截取。

DelimiterBasedFrameDecoder：可以自定义分隔符解码器，LineBasedFrameDecoder 实际上是一种特殊的 DelimiterBasedFrameDecoder 解码器。

FixedLengthFrameDecoder: 固定长度解码器，它能够按照指定的长度对消息进行相应的拆包。

自定义序列化编解码器

在 Java 中自带的有实现 Serializable 接口来实现序列化，但由于它性能、安全性等原因一般情况下是不会被使用到的。

通常情况下，我们使用 Protostuff、Hessian2、json 序列方式比较多，另外还有一些序列化性能非常好的序列化方式也是很好的选择：

- 专门针对 Java 语言的：Kryo，FST 等等。
- 跨语言的：Protostuff（基于 protobuf 发展而来），ProtoBuf，Thrift，

Avro，MsgPack 等等。

5、Netty 的使用场景

- 构建高性能、低时延的各种 Java 中间件，Netty 主要作为基础通信框架提供高性能、低时延的通信服务。例如：RocketMQ，分布式消息队列。Dubbo，服务调用框架。Spring WebFlux，基于响应式的 Web 框架。
- 公有或者私有协议栈的基础通信框架，例如可以基于 Netty 构建异步、高性能的 WebSocket、Protobuf 等协议的支持。
- 各领域应用，例如大数据、游戏等，Netty 作为高性能的通信框架用于内部各模块的数据分发、传输和汇总等，实现模块之间高性能通信。

6、如何选择序列化协议？

具体场景

对于公司间的系统调用，如果性能要求在 100ms 以上的服务，基于 XML 的 SOAP 协议是一个值得考虑的方案。

基于 Web browser 的 Ajax，以及 Mobile app 与服务端之间的通讯，JSON 协议是首选。对于性能要求不太高，或者以动态类型语言为主，或者传输数据载荷很小的的运用场景，JSON 也是非常不错的选择。

对于调试环境比较恶劣的场景，采用 JSON 或 XML 能够极大的提高调试效率，降低系统开发成本。

当对性能和简洁性有极高要求的场景，Protobuf，Thrift，Avro 之间具有一定的竞争关系。

对于 T 级别的数据的持久化应用场景，Protobuf 和 Avro 是首要选择。如果持久化后的数据存储在 hadoop 子项目里，Avro 会是更好的选择。

对于持久层非 Hadoop 项目，以静态类型语言为主的应用场景，Protobuf 会更符合静态类型语言工程师的开发习惯。由于 Avro 的设计理念偏向于动态类型语言，对于动态语言为主的应用场景，Avro 是更好的选择。

如果需要提供一个完整的 RPC 解决方案，Thrift 是一个好的选择。

如果序列化之后需要支持不同的传输层协议，或者需要跨防火墙访问的高性能场景，Protobuf 可以优先考虑。

protobuf 的数据类型有多种：bool、double、float、int32、int64、string、bytes、enum、message。protobuf 的限定符：required: 必须赋值，不能为空、optional:字段可以赋值，也可以不赋值、repeated: 该字段可以重复任意次数（包括 0 次）、枚举；只能用指定的常量集中的一个值作为其值；

protobuf 的基本规则：每个消息中必须至少留有一个 required 类型的字段、包含 0 个或多个 optional 类型的字段；repeated 表示的字段可以包含 0 个或多个数据；[1,15]之内的标识号在编码的时候会占用一个字节（常用），

[16,2047]之内的标识号则占用 2 个字节，标识号一定不能重复、使用消息类型，也可以将消息嵌套任意多层，可用嵌套消息类型来代替组。

protobuf 的消息升级原则：不要更改任何已有的字段的数值标识；不能移除已经存在的 required 字段，optional 和 repeated 类型的字段可以被移除，但要保留标号不能被重用。新添加的字段必须是 optional 或 repeated。因为旧版本程序无法读取或写入新增的 required 限定符的字段。

编译器为每一个消息类型生成了一个.java 文件，以及一个特殊的 Builder 类（该类是用来创建消息类接口的）。如：UserProto.User.Builder builder = UserProto.User.newBuilder();builder.build()；

Netty 中的使用：ProtobufVarint32FrameDecoder 是用于处理半包消息的解码类；ProtobufDecoder(UserProto.User.getDefaultInstance())这是创建的 UserProto.java 文件中的解码类；ProtobufVarint32LengthFieldPrepender 对 protobuf 协议的消息头上加上一个长度为 32 的整形字段，用于标志这个消息的长度的类；ProtobufEncoder 是编码类

7、TCP 粘包/拆包的原因及解决方法？

TCP 是以流的方式来处理数据，一个完整的包可能会被 TCP 拆分成多个包进行发送，也可能把小的封装成一个大的数据包发送。

TCP 粘包/分包的原因：

应用程序写入的字节大小大于套接字发送缓冲区的大小，会发生拆包现象，而应用程序写入数据小于套接字缓冲区大小，网卡将应用多次写入的数据发送到网络上，这将会发生粘包现象；

进行 MSS 大小的 TCP 分段，当 TCP 报文长度-TCP 头部长度>MSS 的时候将发生拆包以太网帧的 payload（净荷）大于 MTU（1500 字节）进行 ip 分片。

解决方法

消息定长：FixedLengthFrameDecoder 类

包尾增加特殊字符分割：行分隔符类：LineBasedFrameDecoder 或自定义分隔符类

DelimiterBasedFrameDecoder：将消息分为消息头和消息体：LengthFieldBasedFrameDecoder 类。分为有头部的拆包与粘包、长度字段在前且有头部的拆包与粘包、多扩展头部的拆包与粘包。

8、Netty 的零拷贝实现？

Netty 的接收和发送 ByteBuffer 采用 DIRECT BUFFERS，使用堆外直接内存进行 Socket 读写，不需要进行字节缓冲区的二次拷贝。堆内存多了一次内存拷贝，JVM 会将堆内存 Buffer 拷贝一份到直接内存中，然后才写入

Socket 中。ByteBuffer 由 ChannelConfig 分配，而 ChannelConfig 创建 ByteBufAllocator 默认使用 Direct Buffer

CompositeByteBuf 类可以将多个 ByteBuf 合并为一个逻辑上的 ByteBuf, 避免了传统通过内存拷贝的方式将几个小 Buffer 合并成一个大的 Buffer。addComponents 方法将 header 与 body 合并为一个逻辑上的 ByteBuf, 这两个 ByteBuf 在 CompositeByteBuf 内部都是单独存在的, CompositeByteBuf 只是逻辑上是一个整体。

通过 FileRegion 包装的 FileChannel.tranferTo 方法 实现文件传输, 可以直接将文件缓冲区的数据发送到目标 Channel，避免了传统通过循环 write 方式导致的内存拷贝问题。

通过 wrap 方法, 我们可以将 byte[] 数组、ByteBuf、ByteBuffer 等包装成一个 NettyByteBuf 对象, 进而避免了拷贝操作。

Selector BUG：若 Selector 的轮询结果为空，也没有 wakeup 或新消息处理，则发生空轮询，CPU 使用率 100%，

Netty 的解决办法：对 Selector 的 select 操作周期进行统计，每完成一次空的 select 操作进行一次计数，若在某个周期内连续发生 N 次空轮询，则触发了 epoll 死循环 bug。重建 Selector，判断是否是其他线程发起的重建请求，若不是则将原 SocketChannel 从旧的 Selector 上去除注册，重新注册到新的 Selector 上，并将原来的 Selector 关闭。

9、Netty 的线程模型？

Netty 通过 Reactor 模型基于多路复用器接收并处理用户请求，内部实现了两个线程池，boss 线程池和 work 线程池，其中 boss 线程池的线程负责处理请求的 accept 事件，当接收到 accept 事件的请求时，把对应的 socket 封装到一个 NioSocketChannel 中，并交给 work 线程池，其中 work 线程池负责请求的 read 和 write 事件，由对应的 Handler 处理。

单线程模型：

所有 I/O 操作都由一个线程完成，即多路复用、事件分发和处理都是在一个 Reactor 线程上完成的。既要接收客户端的连接请求,向服务端发起连接，又要发送/读取请求或应答/响应消息。一个 NIO 线程同时处理成百上千的链路，性能上无法支撑，速度慢，若线程进入死循环，整个程序不可用，对于高负载、大并发的应用场景不合适。

多线程模型：

有一个 NIO 线程（Acceptor） 只负责监听服务端，接收客户端的 TCP 连接请求；NIO 线程池负责网络 IO 的操作，即消息的读取、解码、编码和发送；1 个 NIO 线程可以同时处理 N 条链路，但是 1 个链路只对应 1 个 NIO 线程，这是为了防止发生并发操作问题。但在并发百万客户端连接或需要安全认证时，一个 Acceptor 线程可能会存在性能不足问题。

主从多线程模型：

Acceptor 线程用于绑定监听端口，接收客户端连接，将 SocketChannel 从主线程池的 Reactor 线程的多路复用器上移除，重新注册到 Sub 线程池的线程上，用于处理 I/O 的读写等操作，从而保证 mainReactor 只负责接入认证、握手等操作。

10、Netty 空闲检测

IdleStateHandler，用于检测连接的读写是否处于空闲状态。如果是，则会触发 IdleStateEvent 。