Netty:

关于这部分：一定是建立在对于《Netty 实战》、《Netty 权威指南》、《Unix 网络编程》重点章节都看过之后，才进行理解。对于文章中部分问题，在 Netty 权威指南中都可以找到，或者找相关博客，推荐 Netty 博客：占小狼 Netty 系列（源码级别解析），这个博客+书籍+遇到特例问题自己搜索其他博客补充，即可，通关 Netty，让你校招逼格满满!

BIO 是什么?

概念

- BIO ，全称 Block-IO ，是一种**阻塞 + 同步**的通信模式。
- 是一个比较传统的通信方式，模式简单，使用方便。但并发处理能力低，通信耗时，依赖网速。

原理

- 服务器通过一个 Acceptor 线程，负责监听客户端请求和为每个客户端创建一个新的线程进行链路处理。典型的**一请求一应答模式**。
- 若客户端数量增多，频繁地创建和销毁线程会给服务器打开很大的压力。后改良为用线程池的方式代替新增线程，被称为伪异步 IO 。

示例

- 代码参见 bio 。

小结

BIO 模型中，通过 Socket 和 ServerSocket 实现套接字通道的通信。阻塞，同步，建立连接耗时。

NIO 是什么?

概念

- NIO ，全称 New IO ，也叫 Non-Block IO ，是一种**非阻塞 +** 同步的通信模式。

原理

- NIO 相对于 BIO 来说一大进步。客户端和服务器之间通过 Channel 通信。NIO 可以在 Channel 进行读写操作。这些 Channel 都会被注册在 Selector 多路复用器上。Selector 通过一个线程不停的轮询这些 Channel 。找出已经准备就绪的 Channel 执行 IO 操作。
- NIO 通过一个线程轮询，实现千万个客户端的请求，这就是非阻塞 NIO 的特点。
  - 缓冲区 Buffer ：它是 NIO 与 BIO 的一个重要区别。
    - BIO 是将数据直接写入或读取到流 Stream 对象中。
    - NIO 的数据操作都是在 Buffer 中进行的。Buffer 实际上是一个数组。Buffer 最常见的类型是 ByteBuffer，另外还有 CharBuffer，ShortBuffer，IntBuffer，LongBuffer，FloatBuffer，DoubleBuffer。
  - 通道 Channel ：和流 Stream 不同，通道是双向的。NIO 可以通过 Channel 进行数据的读、写和同时读写操作。
    - 通道分为两大类：一类是网络读写（SelectableChannel），一类是用于文件操作（FileChannel）。我们使用的是前者 SocketChannel 和 ServerSocketChannel，都是 SelectableChannel 的子类。
  - 多路复用器 Selector ：NIO 编程的基础。多路复用器提供选择已经就绪的任务的能力：就是 Selector 会不断地轮询注册在其上的通道（Channel），如果某个通道处于就绪状态，会被 Selector 轮询出来，然后通过 SelectionKey 可以取得就绪的 Channel 集合，从而进行后续的 IO 操作。
    - 服务器端只要提供一个线程负责 Selector 的轮询，就可以接入成千上万个客户端，这就是 JDK NIO 库的巨大进步。

示例

- 代码参见 nio
- **小结**

NIO 模型中通过 SocketChannel 和 ServerSocketChannel 实现套接字通道的通信。非阻塞，同步，避免为每个 TCP 连接创建一个线程。

继续挖掘

可能有胖友对非阻塞和阻塞，同步和异步的定义有点懵逼，

在一些文章中，会将 Java NIO 描述成**异步** IO ，实际是不太正确的： Java NIO 是**同步** IO ，Java AIO (也称为 NIO 2)是**异步** IO。具体原因，推荐阅读文章：

- 《异步和非阻塞一样吗? (内容涉及 BIO, NIO, AIO, Netty)》 。
- 《BIO 与 NIO、AIO 的区别(这个容易理解)》

总结来说，在 **Unix IO 模型**的语境下：

- 同步和异步的区别：数据拷贝阶段是否需要完全由操作系统处理。
- 阻塞和非阻塞操作：是针对发起 IO 请求操作后，是否有立刻返回一个标志信息而不让请求线程等待。

因此，Java NIO 是**同步**且非阻塞的 IO 。

-

AIO 是什么?

对于 AIO 来说，基本理解即可。

概念

- AIO ，全称 Asynchronous IO ，也叫 NIO2 ，是一种**非阻塞 + 异步**的通信模式。在 NIO 的基础上，引入了新的异步通道的概念，并提供了异步文件通道和异步套接字通道的实现。
- 原理：

- AIO 并没有采用 NIO 的多路复用器，而是使用异步通道的概念。其 read，write 方法的返回类型，都是 Future 对象。而 Future 模型是异步的，其核心思想是：**去主函数等待时间**。

示例

- 代码参见 aio

小结

AIO 模型中通过 AsynchronousSocketChannel 和 AsynchronousServerSocketChannel 实现套接字通道的通信。非阻塞，异步。

BIO、NIO 有什么区别?

- 线程模型不同
  - BIO：一个连接一个线程，客户端有连接请求时服务器端就需要启动一个线程进行处理。所以，线程开销大。可改良为用线程池的方式代替新创建线程，被称为伪异步 IO 。
  - NIO：一个请求一个线程，但客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有新的 I/O 请求时，才启动一个线程进行处理。可改良为一个线程处理多个请求，基于 多 Reactor 模型。
- BIO 是面向流(Stream)的，而 NIO 是面向缓冲区(Buffer)的。
- BIO 的各种操作是阻塞的，而 NIO 的各种操作是非阻塞的。
- BIO 的 Socket 是单向的，而 NIO 的 Channel 是双向的。

有一点要注意，虽然图中说 NIO 的性能一般，但是在绝大多数我们日常业务场景，NIO 和 AIO 的性能差距实际没这么大。在 Netty5 中，基于 AIO 改造和支持，最后发现，性能并没有想象中这么强悍，

所以 Netty5 被废弃，而是继续保持 Netty4 为主版本，使用 NIO 为主。

什么是 Netty ？

> Netty 是一款提供异步的、事件驱动的网络应用程序框架和工具，用以快速开发高性能、高可靠性的网络服务器和客户端程序。
>
> 也就是说，Netty 是一个基于 NIO 的客户、服务器端编程框架。使用 Netty 可以确保你快速和简单地开发出一个网络应用，例如实现了某种协议的客户，服务端应用。Netty 相当简化和流线化了网络应用的编程开发过程，例如，TCP 和 UDP 的 socket 服务开发。
>
> （以上摘自百度百科）。

Netty 具有如下特性(摘自《Netty in Action》)

分类	Netty 的特性
设计	1. 统一的 API ， 支持多种传输类型(阻塞和非阻塞的) 2. 简单 报套接字(UDP)支持 4. 连接逻辑组件(ChannelHander 中顺序 可以被多个 ChannelPipeLine 复用)
易于使用	1. 详实的 Javadoc 和大量的示例集 2. 不需要超过 JDK 1.6+
性能	拥有比 Java 的核心 API 更高的吞吐量以及更低的延迟(得益于池化
健壮	1. 不会因为慢速、快速或者超载的连接而导致 OutOfMemoryError <b

分类	Netty 的特性
性	公平读 / 写比率
安全性	完整的 SSL/TLS 以及 StartTLs 支持，可用于受限环境下，如 Appl
社区驱动	发布快速而且频繁

为什么选择 Netty ？

- **使用简单**：API 使用简单，开发门槛低。
- **功能强大**：预置了多种编解码功能，支持多种主流协议。
- **定制能力强**：可以通过 ChannelHandler 对通信框架进行灵活的扩展。
- **性能高**：通过与其它业界主流的 NIO 框架对比，Netty 的综合性能最优。
- **成熟稳定**：Netty 修复了已经发现的所有 JDK NIO BUG，业务开发人员不需要再为 NIO 的 BUG 而烦恼。
- **社区活跃**：版本迭代周期短，发现的 BUG 可以被及时修复，同时，更多的新功能会被加入。
- **案例丰富**：经历了大规模的商业应用考验，质量已经得到验证。在互联网、大数据、网络游戏、企业应用、电信软件等众多行业得到成功商用，证明了它可以完全满足不同行业的商业应用。

实际上，这个也是我们做技术选型的一些参考点，不仅仅适用于 Netty ，也同样适用于其他技术栈。当然，

为什么说 Netty 使用简单?

Java NIO 的步骤如下：

1. 创建 ServerSocketChannel 。
   - 绑定监听端口，并配置为非阻塞模式。

2. 创建 Selector，将之前创建的 ServerSocketChannel 注册到 Selector 上，监听 SelectionKey.OP_ACCEPT 。
    - 循环执行 Selector#select() 方法，轮询就绪的 Channel。
3. 轮询就绪的 Channel 时，如果是处于 OP_ACCEPT 状态，说明是新的客户端接入，调用 ServerSocketChannel#accept() 方法，接收新的客户端。
    - 设置新接入的 SocketChannel 为非阻塞模式，并注册到 Selector 上，监听 OP_READ 。
4. 如果轮询的 Channel 状态是 OP_READ ，说明有新的就绪数据包需要读取，则构造 ByteBuffer 对象，读取数据。
    - 这里，解码数据包的过程，需要我们自己编写。

注意噢，上述步骤还是最简的 Java NIO 启动步骤，不包括多 Reactor 多线程模型

Netty **的步骤**如下：

1. 创建 NIO 线程组 EventLoopGroup 和 ServerBootstrap。
    - 设置 ServerBootstrap 的属性：线程组、SO_BACKLOG 选项，设置 NioServerSocketChannel 为 Channel
    - 设置业务处理 Handler 和 编解码器 Codec 。
    - 绑定端口，启动服务器程序。
2. 在业务处理 Handler 中，处理客户端发送的数据，并给出响应。

简化了哪些步骤呢？

1. 无需关心 OP_ACCEPT、OP_READ、OP_WRITE 等等 IO **操作**，Netty 已经封装，对我们在使用是透明无感的。
2. 使用 boss 和 worker EventLoopGroup ，Netty 直接提供**多** Reactor **多线程模型**。
3. 在 Netty 中，我们看到有使用一个解码器 FixedLengthFrameDecoder，可以用于处理定长消息的问题，能够解决 TCP **粘包拆包**问题，十分方便。如果使用 Java NIO ，需要我们自行实现解码器。

说说业务中 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。
  - HDFS ，分布式文件系统。
- 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。

说说 Netty 如何实现高性能?

1.

线程模型：更加优雅的 Reactor 模式实现、灵活的线程模型、利用 EventLoop 等创新性的机制，可以非常高效地管理成百上千的 Channel 。

2.

内存池设计：使用池化的 Direct Buffer 等技术，在提高 IO 性能的同时，减少了对象的创建和销毁。并且，内吃吃的内部实现是用一颗二叉查找树，更好的管理内存分配情况。

3.

内存零拷贝：使用 Direct Buffer ，可以使用 Zero-Copy 机制。

Zero-Copy ，在操作数据时，不需要将数据 Buffer 从一个内存区域拷贝到另一个内存区域。因为少了一次内存的拷贝，因此 CPU 的效率就得到的提升。

4.

协议支持：提供对 Protobuf 等高性能序列化协议支持。

使用更多本地代码。例如：

- 直接利用 JNI 调用 Open SSL 等方式，获得比 Java 内建 SSL 引擎更好的性能。
- 利用 JNI 提供了 Native Socket Transport ，在使用 Epoll edge-triggered 的情况下，可以有一定的性能提升。

其它：

5.

- 利用反射等技术直接操纵 SelectionKey ，使用数组而不是 Java 容器等。
- 实现 FastThreadLocal 类，当请求频繁时，带来更好的性能。
- ....

另外，推荐阅读白衣大大的两篇文章：

1. 《Netty 高性能编程备忘录(上)》
2. 《Netty 高性能编程备忘录（下）》

下面三连问！

Netty 是一个高性能的、高可靠的、可扩展的异步通信框架，那么高性能、高可靠、可扩展设计体现在哪里呢？

Netty 的高性能如何体现？

这个问题，和「说说 Netty 如何实现高性能？」问题，会有点重叠。

没事，反正理解就好，也背不下来。哈哈哈哈。

性能是设计出来的，而不是测试出来的。那么，Netty 的架构设计是如何实现高性能的呢？

1. **线程模型**：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 模式下服务端无法平滑处理客户端线性增长的问题。
2. **堆外内存**：TCP 接收和发送缓冲区采用直接内存代替堆内存，避免了内存复制，提升了 I/O 读取和写入性能。
3. **内存池设计**：支持通过内存池的方式循环利用 ByteBuf，避免了频繁创建和销毁 ByteBuf 带来的性能消耗。
4. **参数配置**：可配置的 I/O 线程数目和 TCP 参数等，为不同用户提供定制化的调优参数，满足不同的性能场景。
5. **队列优化**：采用环形数组缓冲区，实现无锁化并发编程，代替传统的线程安全容器或锁。
6. **并发能力**：合理使用线程安全容器、原子类等，提升系统的并发能力。
7. **降低锁竞争**：关键资源的使用采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 资源消耗问题。
8. **内存泄露检测**：通过引用计数器及时地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，减少频繁 GC 带来的时延增大和 CPU 损耗。

Netty 的高可靠如何体现？

1. **链路有效性检测**：由于长连接不需要每次发送消息都创建链路，也不需要在消息完成交互时关闭链路，因此相对于短连接性能更高。为了保证长连接的链路有效性，往往需要通过心跳机制周期性地进行链路检测。使用心跳机制的原因是，避免在系统空闲时因网络闪断而断开连接，之后又遇到海量业务冲击导致消息积压无法处理。为了解决这个问题，需要周期性地对链路进行有效性检测，一旦发现问题，可以及时关闭链路，重建 TCP 连接。为了支持心跳，Netty 提供了两种链路空闲检测机制：
   - 读空闲超时机制：连续 T 周期没有消息可读时，发送心跳消息，进行链路检测。如果连续 N 个周期没有读取到心跳消息，可以主动关闭链路，重建连接。
   - 写空闲超时机制：连续 T 周期没有消息需要发送时，发送心跳消息，进行链路检测。如果连续 N 个周期没有读取对方发回的心跳消息，可以主动关闭链路，重建连接。

2. **内存保护机制**：Netty 提供多种机制对内存进行保护，包括以下几个方面：
    - 通过对象引用计数器对 ByteBuf 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。
    - 可设置的内存容量上限，包括 ByteBuf、线程池线程数等，避免异常请求耗光内存。
3. **优雅停机**：优雅停机功能指的是当系统推出时，JVM 通过注册的 Shutdown Hook 拦截到退出信号量，然后执行推出操作，释放相关模块的资源占用，将缓冲区的消息处理完成或清空，将待刷新的数据持久化到磁盘和数据库中，等到资源回收和缓冲区消息处理完成之后，再退出。

Netty 的可扩展如何体现？

可定制、易扩展。

- **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抽象类，便于用户自定义实现。
- **提供大量的工厂类**：通过重载这些工厂类，可以按需创建出用户需要的对象。
- **提供大量系统参数**：供用户按需设置，增强系统的场景定制性。

简单介绍 Netty 的核心组件？

Netty 有如下六个核心组件：

- Bootstrap & ServerBootstrap
- Channel
- ChannelFuture
- EventLoop & EventLoopGroup
- ChannelHandler
- ChannelPipeline
-

什么是 Reactor 模型？

直接看《Netty 权威指南》这本书，认真仔细读，这是一个高频面试题。

请介绍 Netty 的线程模型？

直接看《Netty 权威指南》这本书，认真仔细读什么是业务线程池？

问题

在「什么是 Reactor 模型？」问题中，无论是那种类型的 Reactor 模型，都需要在 Reactor 所在的线程中，进行读写操作。那么此时就会有一个问题，如果我们读取到数据，需要进行业务逻辑处理，并且这个业务逻辑需要对数据库、缓存等等进行操作，会有什么问题呢？假设这个数据库操作需要 5 ms ，那就意味着这个 Reactor 线程在这 5 ms 无法进行注册在这个 Reactor 的 Channel 进行读写操作。也就是说，多个 Channel 的所有读写操作都变成了串行。势必，这样的效率会非常非常非常的低。

解决

那么怎么解决呢？创建业务线程池，将读取到的数据，提交到业务线程池中进行处理。这样，Reactor 的 Channel 就不会被阻塞，而 Channel 的所有读写操作都变成了并行了。

TCP 粘包 / 拆包的原因？应该这么解决？

概念

TCP 是以流的方式来处理数据，所以会导致粘包 / 拆包。

- 拆包：一个完整的包可能会被 TCP 拆分成多个包进行发送。
- 粘包：也可能把小的封装成一个大的数据包发送。

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。
- 待发送数据大于 MSS（最大报文长度），TCP 在传输前将进行**拆包**。
- 以太网帧的 payload（净荷）大于 MTU（默认为 1500 字节）进行 IP 分片**拆包**。
- 接收数据端的应用层没有及时读取接收缓冲区中的数据，将发生**粘包**。

解决

在 Netty 中，提供了多个 Decoder 解析类，如下：

- ① FixedLengthFrameDecoder ，基于**固定长度**消息进行粘包拆包处理的。
- ② LengthFieldBasedFrameDecoder ，基于**消息头指定消息长度**进行粘包拆包处理的。
- ③ LineBasedFrameDecoder ，基于**换行**来进行消息粘包拆包处理的。
- ④ DelimiterBasedFrameDecoder ，基于**指定消息边界方式**进行粘包拆包处理的。

实际上，上述四个 FrameDecoder 实现可以进行规整：

- ① 是 ② 的特例，**固定长度**是**消息头指定消息长度**的一种形式。
- ③ 是 ④ 的特例，**换行**是于**指定消息边界方式**的一种形式。

了解哪几种序列化协议？

概念

- 序列化（编码），是将对象序列化为二进制形式（字节数组），主要用于网络传输、数据持久化等。

- 反序列化（解码），则是将从网络、磁盘等读取的字节数组还原成原始对象，主要用于网络传输对象的解码，以便完成远程调用。

选型

在选择序列化协议的选择，主要考虑以下三个因素：

- 序列化后的**字节大小**。更少的字节数，可以减少网络带宽、磁盘的占用。
- 序列化的**性能**。对 CPU、内存资源占用情况。
- 是否支持**跨语言**。例如，异构系统的对接和开发语言切换。

方案

1. 【重点】Java 默认提供的序列化
   - 无法跨语言；序列化后的字节大小太大；序列化的性能差。
2. 【重点】XML 。
   - 优点：人机可读性好，可指定元素或特性的名称。
   - 缺点：序列化数据只包含数据本身以及类的结构，不包括类型标识和程序集信息；只能序列化公共属性和字段；不能序列化方法；文件庞大，文件格式复杂，传输占带宽。
   - 适用场景：当做配置文件存储数据，实时数据转换。
3. 【重点】JSON ，是一种轻量级的数据交换格式。
   - 优点：兼容性高、数据格式比较简单，易于读写、序列化后数据较小，可扩展性好，兼容性好。与 XML 相比，其协议比较简单，解析速度比较快。
   - 缺点：数据的描述性比 XML 差、不适合性能要求为 ms 级别的情况、额外空间开销比较大。
   - 适用场景（可替代 XML ）：跨防火墙访问、可调式性要求高、基于 Restful API 请求、传输数据量相对小，实时性要求相对低（例如秒级别）的服务。
4. 【了解】Thrift ，不仅是序列化协议，还是一个 RPC 框架。
   - 优点：序列化后的体积小，速度快、支持多种语言和丰富的数据类型、对于数据字段的增删具有较强的兼容性、支持二进制压缩编码。
   - 缺点：使用者较少、跨防火墙访问时，不安全、不具有可读性，调试代码时相对困难、不能与其他传输层协议共同使用（例如 HTTP）、无法支持向持久层直接读写数据，即不适合做数据持久化序列化协议。
   - 适用场景：分布式系统的 RPC 解决方案。
5. 【了解】Avro ，Hadoop 的一个子项目，解决了 JSON 的冗长和没有 IDL 的问题。
   - 优点：支持丰富的数据类型、简单的动态语言结合功能、具有自我描述属性、提高了数据解析速度、快速可压缩的二进制数据形式、可以实现远程过程调用 RPC、支持跨编程语言实现。

- 缺点：对于习惯于静态类型语言的用户不直观。
- 适用场景：在 Hadoop 中做 Hive、Pig 和 MapReduce 的持久化数据格式。

6. 【重点】Protobuf ，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。
   - 优点：序列化后码流小，性能高、结构化数据存储格式（XML JSON 等）、通过标识字段的顺序，可以实现协议的前向兼容、结构化的文档更容易管理和维护。
   - 缺点：需要依赖于工具生成代码、支持的语言相对较少，官方只支持 Java 、C++、python。
   - 适用场景：对性能要求高的 RPC 调用、具有良好的跨防火墙的访问属性、适合应用层对象的持久化。
7. 其它
   - 【重点】Protostuff ，基于 Protobuf 协议，但不需要配置 proto 文件，直接导包即可。
     - 目前，微博 RPC 框架 Motan 在使用它。
   - 【了解】Jboss Marshaling ，可以直接序列化 Java 类， 无须实 java.io.Serializable 接口。
   - 【了解】Message Pack ，一个高效的二进制序列化格式。
   - 【重点】Hessian ，采用二进制协议的轻量级 remoting on http 服务。
     - 目前，阿里 RPC 框架 Dubbo 的**默认**序列化协议。
   - 【重要】kryo ，是一个快速高效的 Java 对象图形序列化框架，主要特点是性能、高效和易用。该项目用来序列化对象到文件、数据库或者网络。
     - 目前，阿里 RPC 框架 Dubbo 的可选序列化协议。
   - 【重要】FST ，fast-serialization 是重新实现的 Java 快速对象序列化的开发包。序列化速度更快（2-10 倍）、体积更小，而且兼容 JDK 原生的序列化。要求 JDK 1.7 支持。
     - 目前，阿里 RPC 框架 Dubbo 的可选序列化协议。

Netty 的零拷贝实现？

Netty 的零拷贝实现，是体现在多方面的，主要如下：

1. 【重点】Netty 的接收和发送 ByteBuffer 采用堆外直接内存 Direct Buffer 。

- 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；使用堆内内存会多了一次内存拷贝，JVM 会将堆内存 Buffer 拷贝一份到直接内存中，然后才写入 Socket 中。
- Netty 创建的 ByteBuffer 类型，由 ChannelConfig 配置。而 ChannelConfig 配置的 ByteBufAllocator 默认创建 Direct Buffer 类型。

2. 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，可将 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，即 CompositeByteBuf 只是逻辑上是一个整体。
3. 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。
4. 通过 wrap 方法，我们可以将 byte[] 数组、ByteBuf、ByteBuffer 等包装成一个 Netty ByteBuf 对象，进而避免了拷贝操作。

原生的 NIO 存在 Epoll Bug 是什么？Netty 是怎么解决的？

Java NIO Epoll BUG

Java NIO Epoll 会导致 Selector 空轮询，最终导致 CPU 100% 。

官方声称在 JDK 1.6 版本的 update18 修复了该问题，但是直到 JDK 1.7 版本该问题仍旧存在，只不过该 BUG 发生概率降低了一些而已，它并没有得到根本性解决。

Netty 解决方案

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，则判断触发了 Epoll 死循环 Bug 。

此时，Netty **重建** Selector 来解决。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 上取消注册，然后重新注册到新的 Selector 上，最后将原来的 Selector 关闭。

什么是 Netty 空闲检测？

在 Netty 中，提供了 IdleStateHandler 类，正如其名，空闲状态处理器，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 。

IdleStateHandler 目前提供三种类型的心跳检测，通过构造方法来设置。代码如下：

```
// IdleStateHandler.java


public IdleStateHandler(
            int readerIdleTimeSeconds,
            int writerIdleTimeSeconds,
            int allIdleTimeSeconds) {
      this(readerIdleTimeSeconds, writerIdleTimeSeconds,
allIdleTimeSeconds,
              TimeUnit.SECONDS);
}
```

- readerIdleTimeSeconds 参数：为读超时时间，即测试端一定时间内未接受到被测试端消息。
- writerIdleTimeSeconds 参数：为写超时时间，即测试端一定时间内向被测试端发送消息。
- allIdleTimeSeconds 参数：为读或写超时时间。

另外，我们会在网络上看到类似《IdleStateHandler 心跳机制》这样标题的文章，实际上空闲检测和心跳机制是**两件事**。

- 只是说，因为我们使用 IdleStateHandler 的目的，就是检测到连接处于空闲，通过心跳判断其是否还是**有效的连接**。
- 虽然说，TCP 协议层提供了 Keeplive 机制，但是该机制默认的心跳时间是 2 小时，依赖操作系统实现不够灵活。因而，我们才在应用层上，自己实现心跳机制。

具体的，我们来看看下面的问题

Netty 如何实现重连？

- 客户端，通过 IdleStateHandler 实现定时检测是否空闲，例如说 15 秒。
  - 如果空闲，则向服务端发起心跳。
  - 如果多次心跳失败，则关闭和服务端的连接，然后重新发起连接。
- 服务端，通过 IdleStateHandler 实现定时检测客户端是否空闲，例如说 90 秒。
  - 如果检测到空闲，则关闭客户端。
  - 注意，如果接收到客户端的心跳请求，要反馈一个心跳响应给客户端。通过这样的方式，使客户端知道自己心跳成功。

Netty 自己实现的 ByteBuf 有什么优点？

如下是 《Netty 实战》 对它的**优点总**结：

- A01. 它可以被用户自定义的**缓冲区类型**扩展
- A02. 通过内置的符合缓冲区类型实现了透明的**零拷贝**
- A03. 容量可以**按需增长**
- A04. 在读和写这两种模式之间切换不需要调用 #flip() 方法
- A05. 读和写使用了**不同的索引**
- A06. 支持方法的**链式**调用
- A07. 支持引用计数

- Ao8. 支持**池化**

Netty 为什么要实现内存管理?

在 Netty 中,IO 读写必定是非常频繁的操作,而考虑到更高效的网络传输性能,Direct ByteBuffer 必然是最合适的选择。但是 Direct ByteBuffer 的申请和释放是高成本的操作,那么进行**池化**管理,多次重用是比较有效的方式。但是,不同于一般于我们常见的对象池、连接池等**池化**的案例,ByteBuffer 是有**大小**一说。又但是,申请多大的 Direct ByteBuffer 进行池化又会是一个大问题,太大会浪费内存,太小又会出现频繁的扩容和内存复制!!!所以呢,就需要有一个合适的内存管理算法,解决**高效分配内存**的同时又解决**内存碎片化**的问题。

官方的说法

> Netty 4.x 增加了 Pooled Buffer,实现了高性能的 buffer 池,分配策略则是结合了 buddy allocation 和 slab allocation 的 **jemalloc** 变种,代码在 io.netty.buffer.PoolArena 中。
>
> 官方说提供了以下优势:
>
> - 频繁分配、释放 buffer 时减少了 GC 压力。
> - 在初始化新 buffer 时减少内存带宽消耗(初始化时不可避免的要给 buffer 数组赋初始值)。
> - 及时的释放 direct buffer 。

hushi55 **大佬的理解**

> C/C++ 和 java 中有个围城,城里的想出来,城外的想进去!**
>
> 这个围城就是自动内存管理!
>
> **Netty 4 buffer 介绍**
>
> Netty4 带来一个与众不同的特点是其 ByteBuf 的实现,相比之下,通过维护两个独立的读写指针, 要比 io.netty.buffer.ByteBuf 简单不少,也会更高效一些。不过,Netty 的 ByteBuf 带给我们的最大不同,就是他不再基于传统 JVM 的 GC 模式,相反,它采用了类似于 C++ 中的 malloc/free 的机制,需要开发人员来手动管理回收与释放。从手动内存管理上

升到 GC，是一个历史的巨大进步， 不过，在 20 年后，居然有曲线的回归到了手动内存管理模式，正印证了马克思哲学观： **社会总是在螺旋式前进的，没有永远的最好。**

① GC 内存管理分析

的确，就内存管理而言，GC 带给我们的价值是不言而喻的，不仅大大的降低了程序员的心智包袱， 而且，也极大的减少了内存管理带来的 Crash 困扰，为函数式编程（大量的临时对象）、脚本语言编程带来了春天。 并且，高效的 GC 算法也让大部分情况下程序可以有更高的执行效率。 不过，也有很多的情况，可能是手工内存管理更为合适的。譬如：

- 对于类似于业务逻辑相对简单，譬如网络路由转发型应用（很多 erlang 应用其实是这种类型）， 但是 QPS 非常高，比如 1M 级，在这种情况下，在每次处理中即便产生 1K 的垃圾，都会导致频繁的 GC 产生。 在这种模式下，erlang 的按进程回收模式，或者是 C/C++ 的手工回收机制，效率更高。
- Cache 型应用，由于对象的存在周期太长，GC 基本上就变得没有价值。

所以，理论上，尴尬的 GC 实际上比较适合于处理介于这 2 者之间的情况： 对象分配的频繁程度相比数据处理的时间要少得多的，但又是相对短暂的，典型的，对于 OLTP 型的服务，处理能力在 1K QPS 量级，每个请求的对象分配在 10K-50K 量级， 能够在 5-10s 的时间内进行一 次 younger GC ，每次 GC 的时间可以控制在 10ms 水平上， 这类的应用，实在是太适合 GC 行的模式了，而且结合 Java 高效的分代 GC ，简直就是一个理想搭配。

② 影响

Netty 4 引入了手工内存的模式，我觉得这是一大创新，这种模式甚至于会延展， 应用到 Cache 应用中。实际上，结合 JVM 的诸多优秀特性，如果用 Java 来实现一个 Redis 型 Cache、 或者 In-memory SQL Engine，或者是一个 Mongo DB，我觉得相比 C/C++ 而言，都要更简单很多。 实际上，JVM 也已经提供了打通这种技术的机制，就是 Direct Memory 和 Unsafe 对象。基于这个基础，我们可以像 C 语言一样直接操作内存。实际上，Netty4 的 ByteBuf 也是基于这个基础的。

Netty 如何实现内存管理？

Netty 内存管理机制，基于 Jemalloc 算法。

- 首先会预申请一大块内存 Arena，Arena 由许多 Chunk 组成，而每个 Chunk 默认由 2048 个 page 组成。
- Chunk 通过 AVL 树的形式组织 Page，每个叶子节点表示一个 Page，而中间节点表示内存区域，节点自己记录它在整个 Arena 中的偏移地址。当区域被分配出去后，中间节点上的标记位会被标记，这样就表示这个中间节点以下的所有节点都已被分配了。大于 8k 的内存分配在 PoolChunkList 中，而 PoolSubpage 用于分配小于 8k 的内存，它会把一个 page 分割成多段，进行内存分配。