1. NIO编程解决的问题
2. 解决线程资源受限：NIO 模型中 selector 的作用，一条连接来了之后，现在不创建一个 while 死循环去监听是否有数据可读了，而是直接把这条连接注册到 selector 上，然后，通过检查这个 selector，就可以批量监测出有数据可读的连接，进而读取数据，实际开发过程中，我们会开多个线程，每个线程都管理着一批连接，相对于 IO 模型中一个线程管理一条连接，消耗的线程资源大幅减少
3. 线程切换效率低下：由于 NIO 模型中线程数量大大降低，线程切换效率因此也大幅度提高
4. IO读写面向流：IO 读写是面向流的，一次性只能从流中读取一个或者多个字节，并且读完之后流无法再读取，你需要自己缓存数据。 而 NIO 的读写是面向 Buffer 的，你可以随意读取里面任何一个字节数据，不需要你自己缓存数据，这一切只需要移动读写指针即可。
5. 使用 Netty 不使用 JDK 原生 NIO 的原因：

（1）使用 JDK 自带的NIO需要了解太多的概念，编程复杂，一不小心 bug 横飞

（2）Netty 底层 IO 模型随意切换，而这一切只需要做微小的改动，改改参数，Netty可以直接从 NIO 模型变身为 IO 模型

（3）Netty 自带的拆包解包，异常检测等机制让你从NIO的繁重细节中脱离出来，让你只需要关心业务逻辑

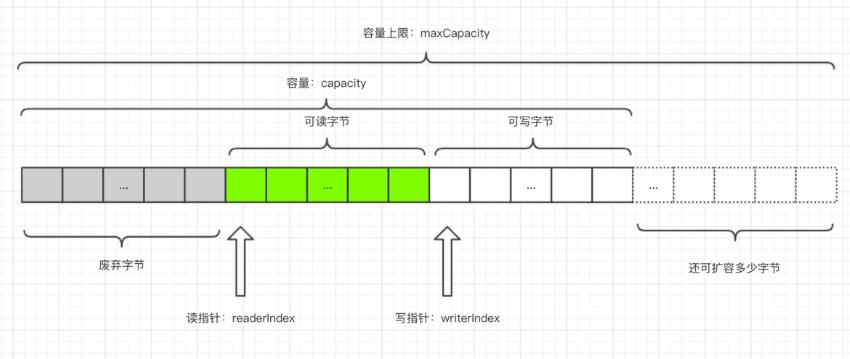
（4）JDK 的 NIO 底层由 epoll 实现，该实现饱受诟病的空轮询 bug 会导致 cpu 飙升 100%，Netty 解决了 JDK 的很多包括空轮询在内的 Bug；

（5）Netty 底层对线程，selector 做了很多细小的优化，精心设计的 reactor 线程模型做到非常高效的并发处理

（6）自带各种协议栈让你处理任何一种通用协议都几乎不用亲自动手；

3、 Netty 服务端启动的流程：创建一个引导类，然后给他指定线程模型，IO模型，连接读写处理逻辑，绑定端口之后，服务端就启动起来了；

4、数据传输载体 ByteBuf：



（1）ByteBuf 是一个字节容器，容器里面的的数据分为三个部分，第一个部分是已经丢弃的字节，这部分数据是无效的；第二部分是可读字节，这部分数据是 ByteBuf 的主体数据， 从 ByteBuf 里面读取的数据都来自这一部分;最后一部分的数据是可写字节，所有写到 ByteBuf 的数据都会写到这一段。最后一部分虚线表示的是该 ByteBuf 最多还能扩容多少容量

（2）以上三段内容是被两个指针给划分出来的，从左到右，依次是读指针（readerIndex）、写指针（writerIndex），然后还有一个变量 capacity，表示 ByteBuf 底层内存的总容量

（3）从 ByteBuf 中每读取一个字节，readerIndex 自增1，ByteBuf 里面总共有 writerIndex-readerIndex 个字节可读, 由此可以推论出当 readerIndex 与 writerIndex 相等的时候，ByteBuf 不可读

（4）写数据是从 writerIndex 指向的部分开始写，每写一个字节，writerIndex 自增1，直到增到 capacity，这个时候，表示 ByteBuf 已经不可写了

（5）ByteBuf 里面其实还有一个参数 maxCapacity，当向 ByteBuf 写数据的时候，如果容量不足，那么这个时候可以进行扩容，直到 capacity 扩容到 maxCapacity，超过 maxCapacity 就会报错；

5、客户端与服务端通信协议：

无论是使用 Netty 还是原始的 Socket 编程，基于 TCP 通信的数据包格式均为二进制，协议指的就是客户端与服务端事先商量好的，每一个二进制数据包中每一段字节分别代表什么含义的规则。

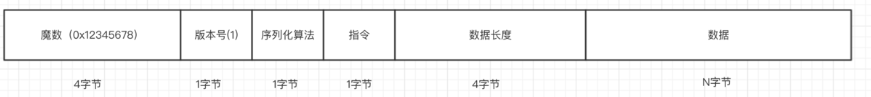
（1）首先，客户端把一个 Java 对象按照通信协议转换成二进制数据包。

（2）然后通过网络，把这段二进制数据包发送到服务端，数据的传输过程由 TCP/IP 协议负责数据的传输，与我们的应用层无关。

（3）服务端接受到数据之后，按照协议取出二进制数据包中的相应字段，包装成 Java 对象，交给应用逻辑处理。

（4）服务端处理完之后，如果需要吐出响应给客户端，那么按照相同的流程进行。

6、通信协议的设计



（1）首先，第一个字段是魔数，通常情况下为固定的几个字节（我们这边规定为4个字节）。 为什么需要这个字段，而且还是一个固定的数？假设我们在服务器上开了一个端口，比如 80 端口，如果没有这个魔数，任何数据包传递到服务器，服务器都会根据自定义协议来进行处理，包括不符合自定义协议规范的数据包。例如，我们直接通过 http://服务器ip 来访问服务器（默认为 80 端口）， 服务端收到的是一个标准的 HTTP 协议数据包，但是它仍然会按照事先约定好的协议来处理 HTTP 协议，显然，这是会解析出错的。而有了这个魔数之后，服务端首先取出前面四个字节进行比对，能够在第一时间识别出这个数据包并非是遵循自定义协议的，也就是无效数据包，为了安全考虑可以直接关闭连接以节省资源。在 Java 的字节码的二进制文件中，开头的 4 个字节为0xcafebabe 用来标识这是个字节码文件，亦是异曲同工之妙。

（2）接下来一个字节为版本号，通常情况下是预留字段，用于协议升级的时候用到，有点类似 TCP 协议中的一个字段标识是 IPV4 协议还是 IPV6 协议，大多数情况下，这个字段是用不到的，不过为了协议能够支持升级，我们还是先留着。

（3）第三部分，序列化算法表示如何把 Java 对象转换二进制数据以及二进制数据如何转换回 Java 对象，比如 Java 自带的序列化，json，hessian 等序列化方式。

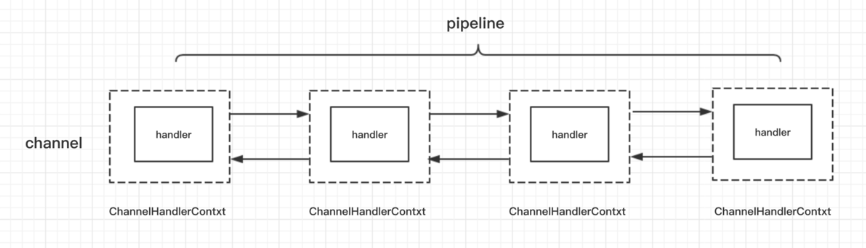
（4）第四部分的字段表示指令，关于指令相关的介绍，我们在前面已经讨论过，服务端或者客户端每收到一种指令都会有相应的处理逻辑，这里，我们用一个字节来表示，最高支持256种指令，对于我们这个 IM 系统来说已经完全足够了。

（5）接下来的字段为数据部分的长度，占四个字节。

（6）最后一个部分为数据内容，每一种指令对应的数据是不一样的，比如登录的时候需要用户名密码，收消息的时候需要用户标识和具体消息内容等等。

注意：Netty 的 ByteBuf 分配器来创建ByteBuf，ioBuffer() 方法会返回适配 io 读写相关的内存，它会尽可能创建一个直接内存，直接内存可以理解为不受 jvm 堆管理的内存空间，写到 IO 缓冲区的效果更高。

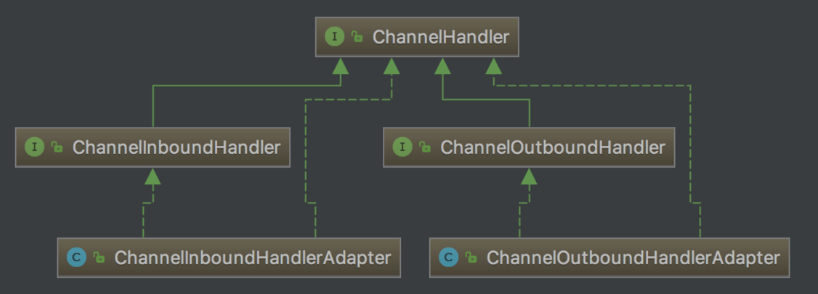
7、pipeline 与 channelHandler 的构成



在 Netty 整个框架里面，一条连接对应着一个 Channel，这条 Channel 所有的处理逻辑都在一个叫做 ChannelPipeline 的对象里面，ChannelPipeline 是一个双向链表结构，他和 Channel 之间是一对一的关系。

ChannelPipeline 里面每个节点都是一个 ChannelHandlerContext 对象，这个对象能够拿到和 Channel 相关的所有的上下文信息，然后这个对象包着一个重要的对象，那就是逻辑处理器 ChannelHandler。

channelHandler 的分类



1. 粘包半包现象：

尽管我们在应用层面使用了 Netty，但是对于操作系统来说，只认 TCP 协议，尽管我们的应用层是按照 ByteBuf 为 单位来发送数据，但是到了底层操作系统仍然是按照字节流发送数据，因此，数据到了服务端，也是按照字节流的方式读入，然后到了 Netty 应用层面，重新拼装成 ByteBuf，而这里的 ByteBuf 与客户端按顺序发送的 ByteBuf 可能是不对等的。因此，我们需要在客户端根据自定义协议来组装我们应用层的数据包，然后在服务端根据我们的应用层的协议来组装数据包，这个过程通常在服务端称为拆包，而在客户端称为粘包。

Netty 自带的拆包器：

1. 固定长度的拆包器 FixedLengthFrameDecoder

如果你的应用层协议非常简单，每个数据包的长度都是固定的，比如 100，那么只需要把这个拆包器加到 pipeline 中，Netty 会把一个个长度为 100 的数据包 (ByteBuf) 传递到下一个 channelHandler。

2. 行拆包器 LineBasedFrameDecoder

从字面意思来看，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，接收端通过 LineBasedFrameDecoder 将粘过的 ByteBuf 拆分成一个个完整的应用层数据包。

3. 分隔符拆包器 DelimiterBasedFrameDecoder

DelimiterBasedFrameDecoder 是行拆包器的通用版本，只不过我们可以自定义分隔符。

4. 基于长度域拆包器 LengthFieldBasedFrameDecoder

最后一种拆包器是最通用的一种拆包器，只要你的自定义协议中包含长度域字段，均可以使用这个拆包器来实现应用层拆包。由于上面三种拆包器比较简单，读者可以自行写出 demo，接下来，我们就结合我们小册的自定义协议，来学习一下如何使用基于长度域的拆包器来拆解我们的数据包。

1. channelHandler的生命周期

（1）handlerAdded() ：指的是当检测到新连接之后，调用 ch.pipeline().addLast(new LifeCyCleTestHandler()); 之后的回调，表示在当前的 channel 中，已经成功添加了一个 handler 处理器。

（2）channelRegistered()：这个回调方法，表示当前的 channel 的所有的逻辑处理已经和某个 NIO 线程建立了绑定关系，类似我们在[Netty 是什么？](https://juejin.im/book/5b4bc28bf265da0f60130116/section/5b4bc28b5188251b1f224ee5)这小节中 BIO 编程中，accept 到新的连接，然后创建一个线程来处理这条连接的读写，只不过 Netty 里面是使用了线程池的方式，只需要从线程池里面去抓一个线程绑定在这个 channel 上即可，这里的 NIO 线程通常指的是 NioEventLoop,不理解没关系，后面我们还会讲到。

（3）channelActive()：当 channel 的所有的业务逻辑链准备完毕（也就是说 channel 的 pipeline 中已经添加完所有的 handler）以及绑定好一个 NIO 线程之后，这条连接算是真正激活了，接下来就会回调到此方法。

（4）channelRead()：客户端向服务端发来数据，每次都会回调此方法，表示有数据可读。

channelReadComplete()：服务端每次读完一次完整的数据之后，回调该方法，表示数据读取完毕。

（5） channelReadComplete()：我们可以在之前调用 writeAndFlush() 的地方都调用 write() 方法，然后在这个方面里面调用 ctx.channel().flush() 方法，相当于一个批量刷新的机制，当然，如果你对性能要求没那么高，writeAndFlush() 足矣。

1. 连接假死：在某一端（服务端或者客户端）看来，底层的 TCP 连接已经断开了，但是应用程序并没有捕获到，因此会认为这条连接仍然是存在的，从 TCP 层面来说，只有收到四次握手数据包或者一个 RST 数据包，连接的状态才表示已断开。

问题：（1）对于服务端来说，因为每条连接都会耗费 cpu 和内存资源，大量假死的连接会逐渐耗光服务器的资源，最终导致性能逐渐下降，程序奔溃。

（2）对于客户端来说，连接假死会造成发送数据超时，影响用户体验。

原因：（1）应用程序出现线程堵塞，无法进行数据的读写。

（2）客户端或者服务端网络相关的设备出现故障，比如网卡，机房故障。

（3）公网丢包。公网环境相对内网而言，非常容易出现丢包，网络抖动等现象，如果在一段时间内用户接入的网络连续出现丢包现象，那么对客户端来说数据一直发送不出去，而服务端也是一直收不到客户端来的数据，连接就一直耗着。

服务端在一段时间内没有收到客户端的数据，这个现象产生的原因可以分为以下两种：

（1）连接假死。

（2）非假死状态下确实没有发送数据。

解决：空闲检测，指的是每隔一段时间，检测这段时间内是否有数据读写，Netty 自带的 IdleStateHandler 就可以实现这个功能；客户端定时发心跳，

11、耗时操作的处理与统计

对于耗时的操作，不要直接在 NIO 线程里做，比如，不要在 channelRead0() 方法里做一些访问数据库或者网络相关的逻辑，要扔到自定义线程池里面去做，然后要注意这个时候，writeAndFlush() 的执行是异步的，需要通过添加监听回调的方式来判断是否执行完毕，进而进行延时的统计。