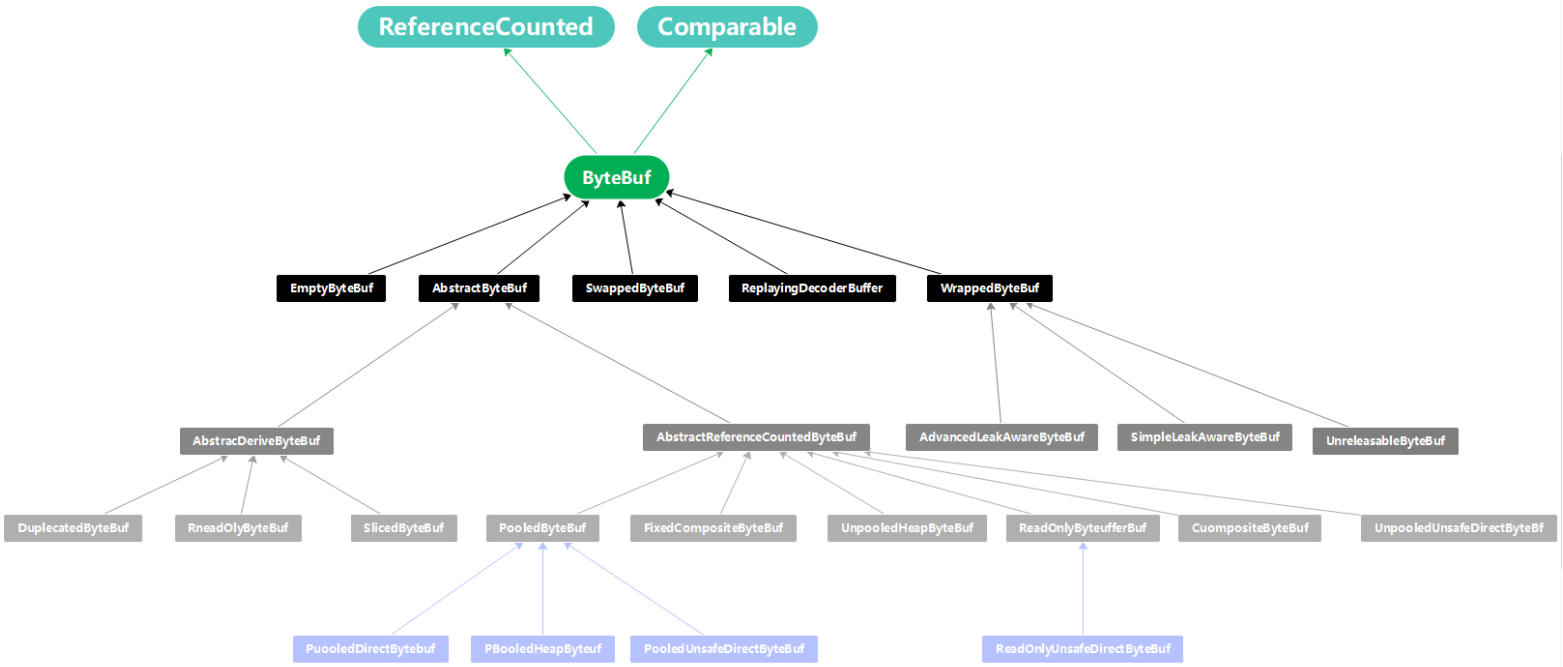
面向对象源码阅读：Netty Btyebuf 模块——核心流程分析

一、 类间关系

首先给出 ByteBuf 的类间关系图如下：



ByteBuf 本身是一个抽象类，它的内部是抽象的函数接口。而它的主要实现是依靠抽象类 AbstractByteBuf。对此在第一部分的报告里已经有所介绍。在 AbstractByteBuf 外，ByteBuf 还有其他的直接子类。

ByteBuf 子类	功能
EmptyByteBuf	构建空的 ByteBuf 对象
ReplayingDecoderBuffer	实现无阻塞解码的 ByteBuf 对象，处理 IO 阻塞时的异常接收情况
SwappedByteBuf	构建具有切换字节顺序功能的 ByteBuf 对象
WrappedByteBuf	提供三个子类用于记录 ByteBuf 对象的一些信息：堆栈信息，对象引用计数器等
AbstractByteBuf	ByteBuf 主要功能的默认实现

在 ByteBuf 的这些直接子类中，AbstractByteBuf 和 WrappedByteBuf 又较为复杂，各自有继承的直接子类进行更加细致的实现，这里也给出说明:

WrappedByteBuf 主要起到对 ByteBuf 的装饰作用，可以监控防止内存泄漏的发生，也可对一个或多个内存对象进行包装，形成一个新的 ByteBuf。

WrappedByteBuf 子类	功能
AdvancedLeakAwareByteBuf	记录所有操作的堆栈信息
SimpleLeakAwareByteBuf	记录 order 的堆栈信息
UnreleasableByteBuf	控制对象引用计数器的修改使能

AbstractByteBuf 是 ByteBuf 主要功能的实现，它有两个直接子类:

AbstractByteBuf 子类	功能
AbstractDerivedByteBuf	实现派生 ByteBuf
AbstractReferenceCountedByteBuf	实现对象引用计数器相关操作

AbstractDerivedByteBuf 的子类如下:

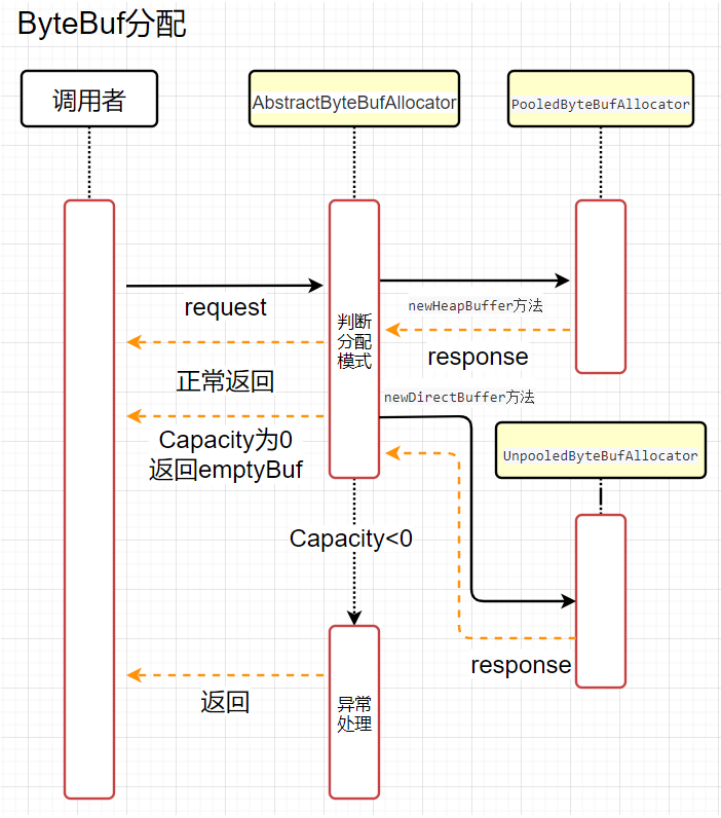
AbstractDerivedByteBuf 子类	功能
DuplicatedByteBuf	创建 ByteBuf 的复制对象，实现同一个缓冲区两套独立读写头的操作方式
ReadOnlyByteBuf	创建 ByteBuf 的只读对象，实现同一个缓冲区两套独立读写头（其中一套的写头被禁用）的操作
SlicedByteBuf	创建 ByteBuf 的一个子区域 ByteBuf 对象，共享部分缓冲区，子区域对象有自己的独立读写头

AbstractReferenceCountedByteBuf 子类	功能
CompositeByteBuf	将多个 ByteBuf 组合在一起，支持动态扩展和合并传输
FixedCompositeByteBuf	将多个 ByteBuf 组合在一起，不支持动态扩展和修改
PooledByteBuf	重用 ByteBuf 对象，提升内存的使用效率
ReadOnlyByteBufferBuf	内部的只读 ByteBuf，操作由 ByteBuffer 实现
UnpooledDirectByteBuf	堆外进行内存分配的非内存池 ByteBuf，操作基于 ByteBuffer
UnpooledHeapByteBuf	堆内存分配非内存池 ByteBuf
UnpooledUnsafeDirectByteBuf	与 UnpooledDirectByteBuf 基本相同，操作基于 PlatformDependent

简而言之，ByteBuf 本身实现 ReferenceCounted 和 Comparable，而它自身主要由 AbstractByteBuf 类实现，AbstractByteBuf 又由子类 DuplicatedByteBuf、ReadOnlyByteBuf、SlicedByteBuf 实现其主要功能。

二、 时序图

ByteBuf 提供了丰富的使用模式和功能，因此很难对其所有的工作流程都做出列举，这里给出 ByteBuf 典型的使用两种 allocator 的分配流程：

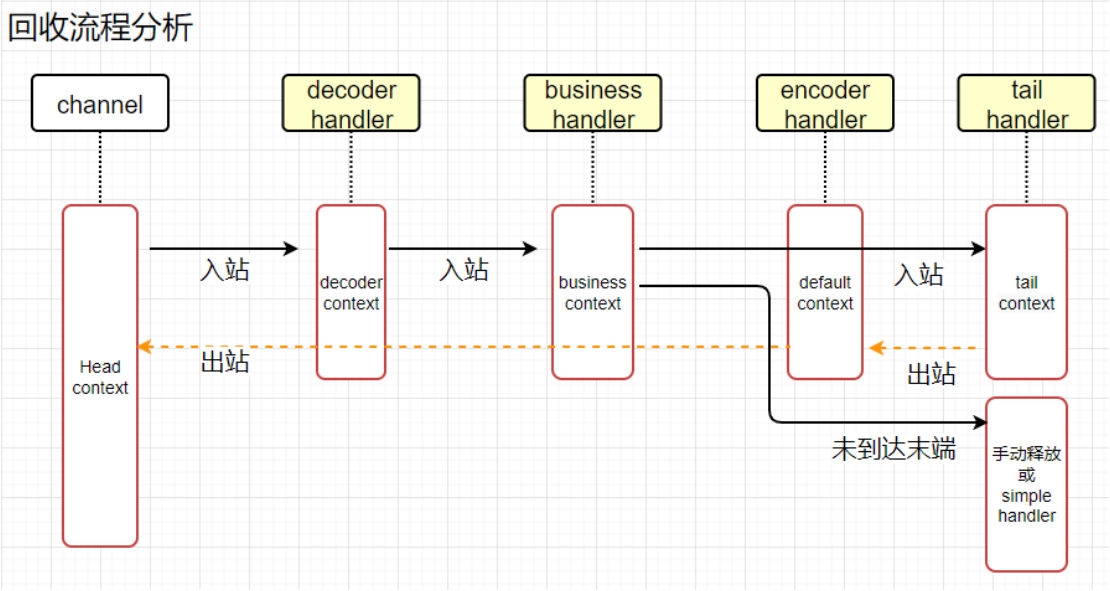


可以看出，在需要 ByteBuf 进行空间分配时，ByteBuf 会使用 AbstractByteBuf 中的方法，AbstractByteBuf 根据 Capicity 的值，判断选择的处理方法。如果 Capicity 值正常，那么根据调用者的模式和系统的情况选择 Direct 或者 Heap

方法进行空间分配，最后返回给调用者。

对于 ByteBuf 的回收，Netty 默认会在处理流水线的最后添加一个 tail handler 完成 ByteBuf 的自动 release。但是如果在使用过程中，向下传递的是新值，那么传入的老值所依赖的 ByteBuf 需要手动释放。相似的，在执行过程中，调用者使用上述流程创建了新的 ByteBuf，但是没有向下继续传递，那么该 ByteBuf 需要手动释放，或者使用 simplehandler 进行释放。ByteBuf 的释放主要出现在非池化的情况中，其方法在类中的依赖情况和上述的分配过程是高度相似的。

对此也可以给出 Tail handler 的回收流程：



关于 ByteBuf 的读写及复制过程，在第一部分的建模过程中已经给出介绍，其使用读写分开的 index 进行操作，在此不再复述。