Netty架构按照Reactor模式设计和实现，它的服务端通信序列图如下：

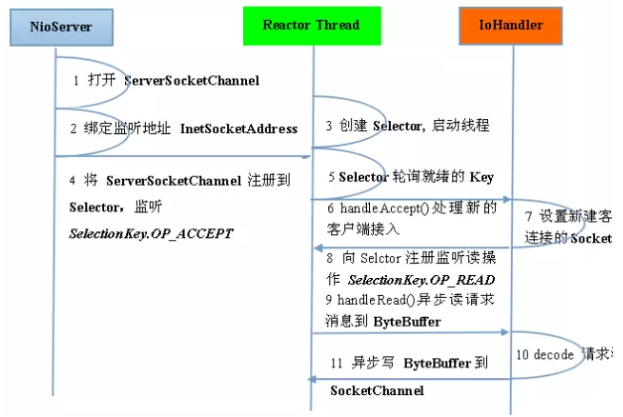


图2-3 NIO服务端通信序列图

客户端通信序列图如下：

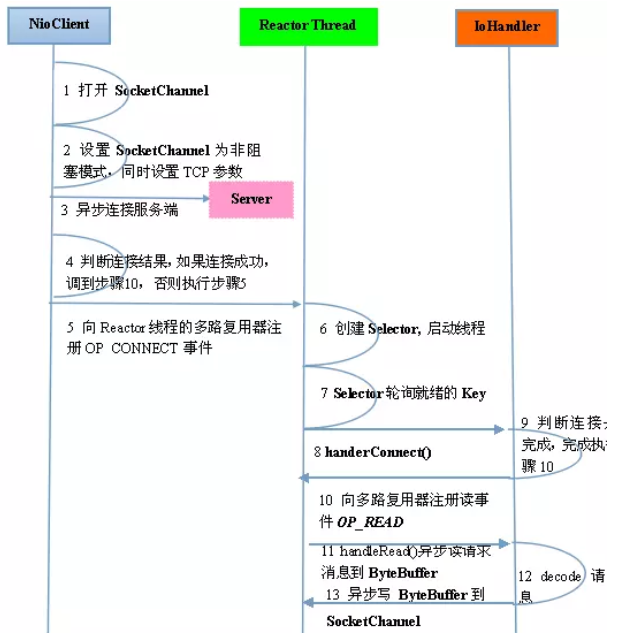


图2-4 NIO客户端通信序列图

Netty的IO线程NioEventLoop由于聚合了多路复用器Selector，可以同时并发处理成百上千个客户端Channel，由于读写操作都是非阻塞的，这就可以充分提升IO线程的运行效率，避免由于频繁IO阻塞导致的线程挂起。另外，由于Netty采用了异步通信模式，一个IO线程可以并发处理N个客户端连接和读写操作，这从根本上解决了传统同步阻塞IO一连接一线程模型，架构的性能、弹性伸缩能力和可靠性都得到了极大的提升。

2.2.2. 零拷贝

很多用户都听说过Netty具有“零拷贝”功能，但是具体体现在哪里又说不清楚，本小节就详细对Netty的“零拷贝”功能进行讲解。

Netty的“零拷贝”主要体现在如下三个方面：

1) 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。如果使用传统的堆内存（HEAP BUFFERS）进行Socket读写，JVM会将堆内存Buffer拷贝一份到直接内存中，然后才写入Socket中。相比于堆外直接内存，消息在发送过程中多了一次缓冲区的内存拷贝。

2) Netty提供了组合Buffer对象，可以聚合多个ByteBuffer对象，用户可以像操作一个Buffer那样方便的对组合Buffer进行操作，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。

3) Netty的文件传输采用了transferTo方法，它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。

下面，我们对上述三种“零拷贝”进行说明，先看Netty 接收Buffer的创建：

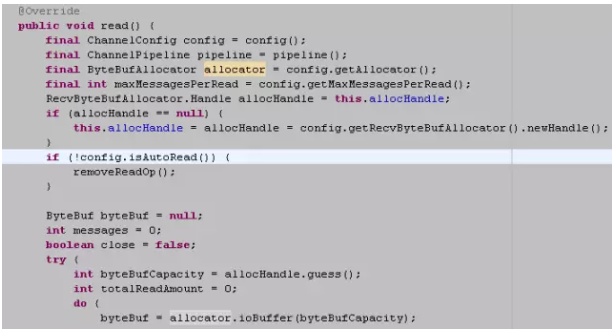


图2-5 异步消息读取“零拷贝”

每循环读取一次消息，就通过ByteBufAllocator的ioBuffer方法获取ByteBuf对象，下面继续看它的接口定义：

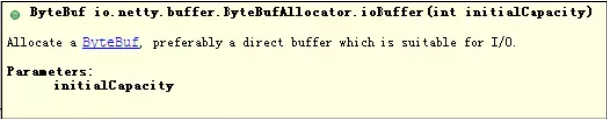


图2-6 ByteBufAllocator 通过ioBuffer分配堆外内存

当进行Socket IO读写的时候，为了避免从堆内存拷贝一份副本到直接内存，Netty的ByteBuf分配器直接创建非堆内存避免缓冲区的二次拷贝，通过“零拷贝”来提升读写性能。

下面我们继续看第二种“零拷贝”的实现CompositeByteBuf，它对外将多个ByteBuf封装成一个ByteBuf，对外提供统一封装后的ByteBuf接口，它的类定义如下：



图2-7 CompositeByteBuf类继承关系

通过继承关系我们可以看出CompositeByteBuf实际就是个ByteBuf的包装器，它将多个ByteBuf组合成一个集合，然后对外提供统一的ByteBuf接口，相关定义如下：

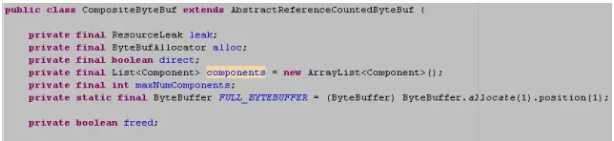


图2-8 CompositeByteBuf类定义

添加ByteBuf，不需要做内存拷贝，相关代码如下：

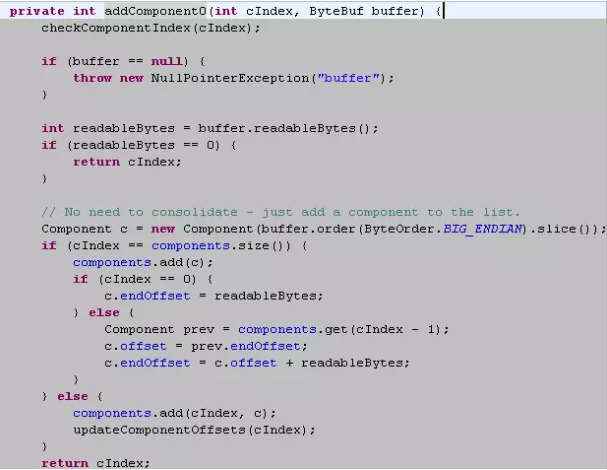


图2-9 新增ByteBuf的“零拷贝”

最后，我们看下文件传输的“零拷贝”：

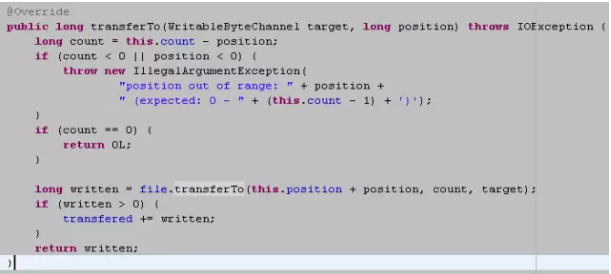


图2-10 文件传输“零拷贝”

Netty文件传输DefaultFileRegion通过transferTo方法将文件发送到目标Channel中，下面重点看FileChannel的transferTo方法，它的API DOC说明如下：

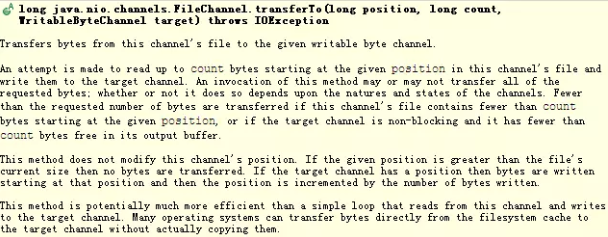


图2-11 文件传输 “零拷贝”

对于很多操作系统它直接将文件缓冲区的内容发送到目标Channel中，而不需要通过拷贝的方式，这是一种更加高效的传输方式，它实现了文件传输的“零拷贝”。

2.2.3. 内存池

随着JVM虚拟机和JIT即时编译技术的发展，对象的分配和回收是个非常轻量级的工作。但是对于缓冲区Buffer，情况却稍有不同，特别是对于堆外直接内存的分配和回收，是一件耗时的操作。为了尽量重用缓冲区，Netty提供了基于内存池的缓冲区重用机制。下面我们一起看下Netty ByteBuf的实现：

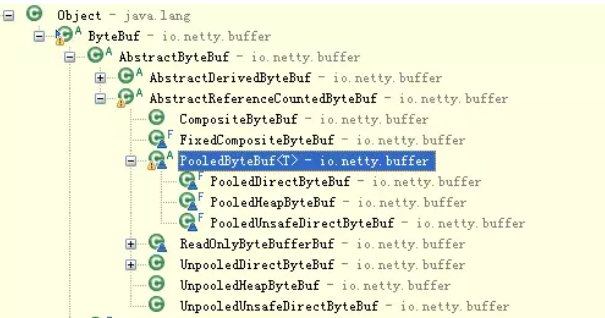


图2-12 内存池ByteBuf

Netty提供了多种内存管理策略，通过在启动辅助类中配置相关参数，可以实现差异化的定制。

下面通过性能测试，我们看下基于内存池循环利用的ByteBuf和普通ByteBuf的性能差异。

用例一，使用内存池分配器创建直接内存缓冲区：

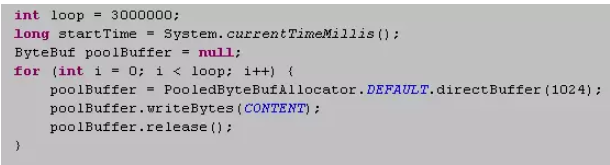


图2-13 基于内存池的非堆内存缓冲区测试用例

用例二，使用非堆内存分配器创建的直接内存缓冲区：

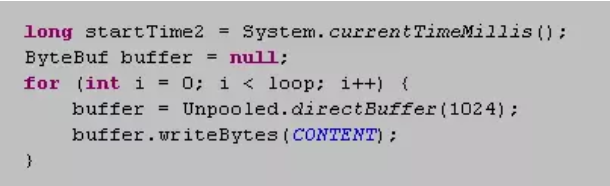


图2-14 基于非内存池创建的非堆内存缓冲区测试用例

各执行300万次，性能对比结果如下所示：

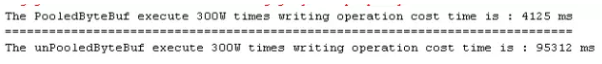


图2-15 内存池和非内存池缓冲区写入性能对比

性能测试表明，采用内存池的ByteBuf相比于朝生夕灭的ByteBuf，性能高23倍左右（性能数据与使用场景强相关）。

下面我们一起简单分析下Netty内存池的内存分配：

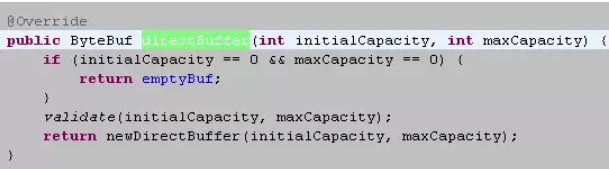


图2-16 AbstractByteBufAllocator的缓冲区分配

继续看newDirectBuffer方法，我们发现它是一个抽象方法，由AbstractByteBufAllocator的子类负责具体实现，代码如下：

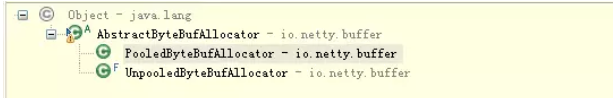


图2-17 newDirectBuffer的不同实现

代码跳转到PooledByteBufAllocator的newDirectBuffer方法，从Cache中获取内存区域PoolArena，调用它的allocate方法进行内存分配：

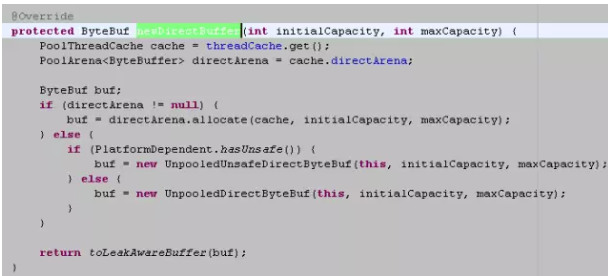


图2-18 PooledByteBufAllocator的内存分配

PoolArena的allocate方法如下：

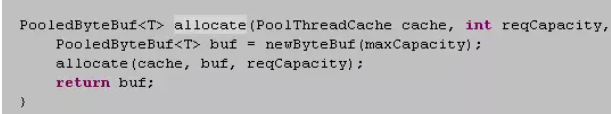


图2-18 PoolArena的缓冲区分配

我们重点分析newByteBuf的实现，它同样是个抽象方法，由子类DirectArena和HeapArena来实现不同类型的缓冲区分配，由于测试用例使用的是堆外内存，

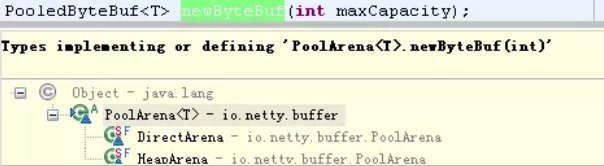


图2-19 PoolArena的newByteBuf抽象方法

因此重点分析DirectArena的实现：如果没有开启使用sun的unsafe，则

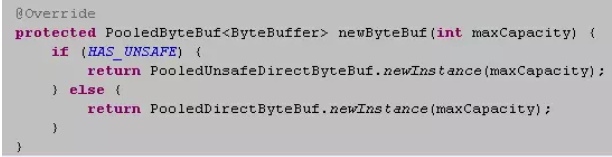


图2-20 DirectArena的newByteBuf方法实现

执行PooledDirectByteBuf的newInstance方法，代码如下：

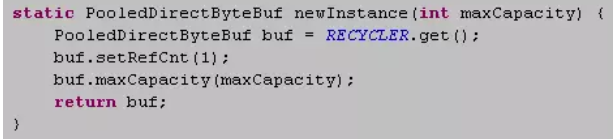


图2-21 PooledDirectByteBuf的newInstance方法实现

通过RECYCLER的get方法循环使用ByteBuf对象，如果是非内存池实现，则直接创建一个新的ByteBuf对象。从缓冲池中获取ByteBuf之后，调用AbstractReferenceCountedByteBuf的setRefCnt方法设置引用计数器，用于对象的引用计数和内存回收（类似JVM垃圾回收机制）。

2.2.4. 高效的Reactor线程模型

常用的Reactor线程模型有三种，分别如下：

1) Reactor单线程模型；

2) Reactor多线程模型；

3) 主从Reactor多线程模型

Reactor单线程模型，指的是所有的IO操作都在同一个NIO线程上面完成，NIO线程的职责如下：

1) 作为NIO服务端，接收客户端的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者应答消息。

Reactor单线程模型示意图如下所示：

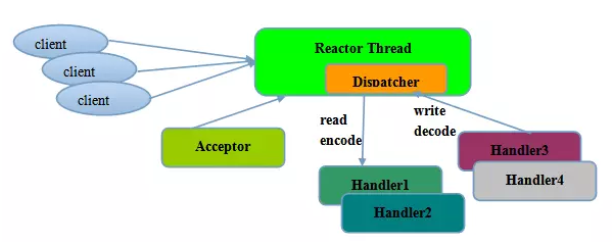


图2-22 Reactor单线程模型

由于Reactor模式使用的是异步非阻塞IO，所有的IO操作都不会导致阻塞，理论上一个线程可以独立处理所有IO相关的操作。从架构层面看，一个NIO线程确实可以完成其承担的职责。例如，通过Acceptor接收客户端的TCP连接请求消息，链路建立成功之后，通过Dispatch将对应的ByteBuffer派发到指定的Handler上进行消息解码。用户Handler可以通过NIO线程将消息发送给客户端。

对于一些小容量应用场景，可以使用单线程模型。但是对于高负载、大并发的应用却不合适，主要原因如下：

1) 一个NIO线程同时处理成百上千的链路，性能上无法支撑，即便NIO线程的CPU负荷达到100%，也无法满足海量消息的编码、解码、读取和发送；

2) 当NIO线程负载过重之后，处理速度将变慢，这会导致大量客户端连接超时，超时之后往往会进行重发，这更加重了NIO线程的负载，最终会导致大量消息积压和处理超时，NIO线程会成为系统的性能瓶颈；

3) 可靠性问题：一旦NIO线程意外跑飞，或者进入死循环，会导致整个系统通信模块不可用，不能接收和处理外部消息，造成节点故障。

为了解决这些问题，演进出了Reactor多线程模型，下面我们一起学习下Reactor多线程模型。

Rector多线程模型与单线程模型最大的区别就是有一组NIO线程处理IO操作，它的原理图如下：

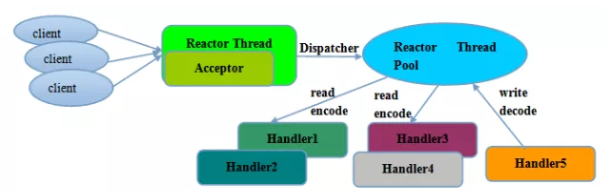


图2-23 Reactor多线程模型

Reactor多线程模型的特点：

1) 有专门一个NIO线程-Acceptor线程用于监听服务端，接收客户端的TCP连接请求；

2) 网络IO操作-读、写等由一个NIO线程池负责，线程池可以采用标准的JDK线程池实现，它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、解码、编码和发送；

3) 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，防止发生并发操作问题。

在绝大多数场景下，Reactor多线程模型都可以满足性能需求；但是，在极特殊应用场景中，一个NIO线程负责监听和处理所有的客户端连接可能会存在性能问题。例如百万客户端并发连接，或者服务端需要对客户端的握手消息进行安全认证，认证本身非常损耗性能。在这类场景下，单独一个Acceptor线程可能会存在性能不足问题，为了解决性能问题，产生了第三种Reactor线程模型-主从Reactor多线程模型。

主从Reactor线程模型的特点是：服务端用于接收客户端连接的不再是个1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TCP连接请求处理完成后（可能包含接入认证等），将新创建的SocketChannel注册到IO线程池（sub reactor线程池）的某个IO线程上，由它负责SocketChannel的读写和编解码工作。Acceptor线程池仅仅只用于客户端的登陆、握手和安全认证，一旦链路建立成功，就将链路注册到后端subReactor线程池的IO线程上，由IO线程负责后续的IO操作。

它的线程模型如下图所示：

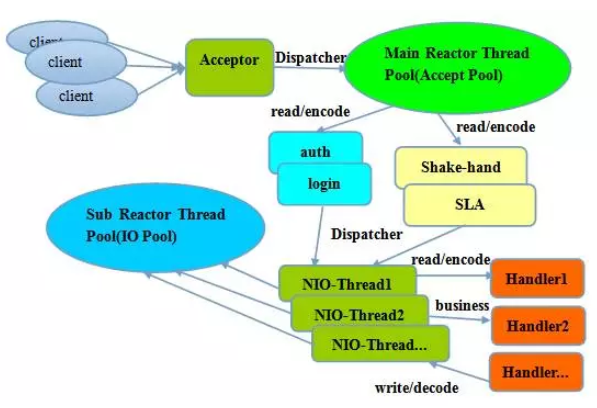


图2-24 Reactor主从多线程模型

利用主从NIO线程模型，可以解决1个服务端监听线程无法有效处理所有客户端连接的性能不足问题。因此，在Netty的官方demo中，推荐使用该线程模型。

事实上，Netty的线程模型并非固定不变，通过在启动辅助类中创建不同的EventLoopGroup实例并通过适当的参数配置，就可以支持上述三种Reactor线程模型。正是因为Netty 对Reactor线程模型的支持提供了灵活的定制能力，所以可以满足不同业务场景的性能诉求。

2.2.5. 无锁化的串行设计理念

在大多数场景下，并行多线程处理可以提升系统的并发性能。但是，如果对于共享资源的并发访问处理不当，会带来严重的锁竞争，这最终会导致性能的下降。为了尽可能的避免锁竞争带来的性能损耗，可以通过串行化设计，即消息的处理尽可能在同一个线程内完成，期间不进行线程切换，这样就避免了多线程竞争和同步锁。

为了尽可能提升性能，Netty采用了串行无锁化设计，在IO线程内部进行串行操作，避免多线程竞争导致的性能下降。表面上看，串行化设计似乎CPU利用率不高，并发程度不够。但是，通过调整NIO线程池的线程参数，可以同时启动多个串行化的线程并行运行，这种局部无锁化的串行线程设计相比一个队列-多个工作线程模型性能更优。

Netty的串行化设计工作原理图如下：

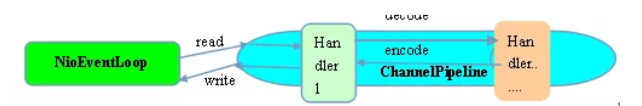


图2-25 Netty串行化工作原理图

Netty的NioEventLoop读取到消息之后，直接调用ChannelPipeline的fireChannelRead(Object msg)，只要用户不主动切换线程，一直会由NioEventLoop调用到用户的Handler，期间不进行线程切换，这种串行化处理方式避免了多线程操作导致的锁的竞争，从性能角度看是最优的。

2.2.6. 高效的并发编程

Netty的高效并发编程主要体现在如下几点：

1) volatile的大量、正确使用;

2) CAS和原子类的广泛使用；

3) 线程安全容器的使用；

4) 通过读写锁提升并发性能。

如果大家想了解Netty高效并发编程的细节，可以阅读之前我在微博分享的《多线程并发编程在 Netty 中的应用分析》，在这篇文章中对Netty的多线程技巧和应用进行了详细的介绍和分析。

2.2.7. 高性能的序列化框架

影响序列化性能的关键因素总结如下：

1) 序列化后的码流大小（网络带宽的占用）；

2) 序列化&反序列化的性能（CPU资源占用）；

3) 是否支持跨语言（异构系统的对接和开发语言切换）。

Netty默认提供了对Google Protobuf的支持，通过扩展Netty的编解码接口，用户可以实现其它的高性能序列化框架，例如Thrift的压缩二进制编解码框架。

2.2.8. 灵活的TCP参数配置能力

合理设置TCP参数在某些场景下对于性能的提升可以起到显著的效果，例如SO\_RCVBUF和SO\_SNDBUF。如果设置不当，对性能的影响是非常大的。下面我们总结下对性能影响比较大的几个配置项：

1) SO\_RCVBUF和SO\_SNDBUF：通常建议值为128K或者256K；

2) SO\_TCPNODELAY：NAGLE算法通过将缓冲区内的小封包自动相连，组成较大的封包，阻止大量小封包的发送阻塞网络，从而提高网络应用效率。但是对于时延敏感的应用场景需要关闭该优化算法；

3) 软中断：如果Linux内核版本支持RPS（2.6.35以上版本），开启RPS后可以实现软中断，提升网络吞吐量。RPS根据数据包的源地址，目的地址以及目的和源端口，计算出一个hash值，然后根据这个hash值来选择软中断运行的cpu，从上层来看，也就是说将每个连接和cpu绑定，并通过这个hash值，来均衡软中断在多个cpu上，提升网络并行处理性能。

Netty在启动辅助类中可以灵活的配置TCP参数，满足不同的用户场景。相关配置接口定义如下：



图2-27 Netty的TCP参数配置定义

2.3. 总结

通过对Netty的架构和性能模型进行分析，我们发现Netty架构的高性能是被精心设计和实现的，得益于高质量的架构和代码，Netty支持10W TPS的跨节点服务调用并不是件十分困难的事情。