# Netty 系列之 Netty 高性能之道

## 1. 背景

## 1.1. 惊人的性能数据

最近一个圈内朋友通过私信告诉我，通过使用 Netty4 + Thrift 压缩二进制编解码技术，他们实现了 10W TPS（1K 的复杂 POJO 对象）的跨节点远程服务调用。相比于传统基于 Java 序列化+BIO（同步阻塞 IO）的通信框架，性能提升了 8 倍多。

事实上，我对这个数据并不感到惊讶，根据我 5 年多的 NIO 编程经验，通过选择合适的 NIO 框架，加上高性能的压缩二进制编解码技术，精心的设计 Reactor 线程模型，达到上述性能指标是完全有可能的。

下面我们就一起来看下 Netty 是如何支持 10W TPS 的跨节点远程服务调用的，在正式开始讲解之前，我们先简单介绍下 Netty。

## 1.2. Netty 基础入门

Netty 是一个高性能、异步事件驱动的 NIO 框架，它提供了对 TCP、UDP 和文件传输的支持，作为一个异步 NIO 框架，Netty 的所有 IO 操作都是异步非阻塞的，通过 Future-Listener 机制，用户可以方便的主动获取或者通过通知机制获得 IO 操作结果。

作为当前最流行的 NIO 框架，Netty 在互联网领域、大数据分布式计算领域、游戏行业、通信行业等获得了广泛的应用，一些业界著名的开源组件也基于 Netty 的 NIO 框架构建。

## 2. Netty 高性能之道

## 2.1. RPC 调用的性能模型分析

### **2.1.1.** 传统 RPC 调用性能差的三宗罪

网络传输方式问题：传统的 RPC 框架或者基于 RMI 等方式的远程服务（过程）调用采用了同步阻塞 IO，当客户端的并发压力或者网络时延增大之后，同步阻塞 IO 会由于频繁的 wait 导致 IO 线程经常性的阻塞，由于线程无法高效的工作，IO 处理能力自然下降。

下面，我们通过 BIO 通信模型图看下 BIO 通信的弊端：

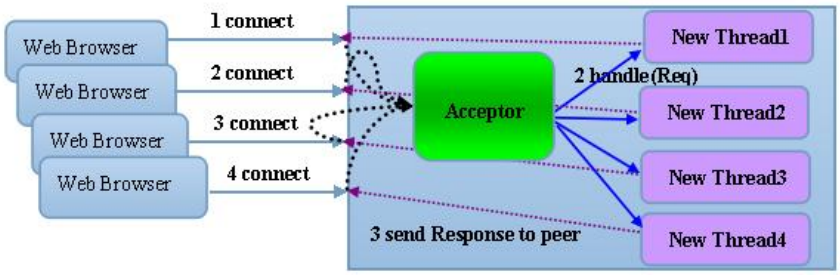


图 2-1 BIO 通信模型图

采用 BIO 通信模型的服务端，通常由一个独立的 Acceptor 线程负责监听客户端的连接，接收到客户端连接之后为客户端连接创建一个新的线程处理请求消息，处理完成之后，返回应答消息给客户端，线程销毁，这就是典型的一请求一应答模型。该架构最大的问题就是不具备弹性伸缩能力，当并发访问量增加后，服务端的线程个数和并发访问数成线性正比，由于线程是 JAVA 虚拟机非常宝贵的系统资源，当线程数膨胀之后，系统的性能急剧下降，随着并发量的继续增加，可能会发生句柄溢出、线程堆栈溢出等问题，并导致服务器最终宕机。

序列化方式问题：Java 序列化存在如下几个典型问题：

1) Java 序列化机制是 Java 内部的一种对象编解码技术，无法跨语言使用；例如对于异构系统之间的对接，Java 序列化后的码流需要能够通过其它语言反序列化成原始对象（副本），目前很难支持；

2) 相比于其它开源的序列化框架，Java 序列化后的码流太大，无论是网络传输还是持久化到磁盘，都会导致额外的资源占用；

3) 序列化性能差（CPU 资源占用高）。

线程模型问题：由于采用同步阻塞 IO，这会导致每个 TCP 连接都占用 1 个线程，由于线程资源是 JVM 虚拟机非常宝贵的资源，当 IO 读写阻塞导致线程无法及时释放时，会导致系统性能急剧下降，严重的甚至会导致虚拟机无法创建新的线程。

### 2.1.2. 高性能的三个主题

1) 传输：用什么样的通道将数据发送给对方，BIO、NIO 或者 AIO，IO 模型在很大程度上决定了框架的性能。

2) 协议：采用什么样的通信协议，HTTP 或者内部私有协议。协议的选择不同，性能模型也不同。相比于公有协议，内部私有协议的性能通常可以被设计的更优。

3) 线程：数据报如何读取？读取之后的编解码在哪个线程进行，编解码后的消息如何派发，Reactor 线程模型的不同，对性能的影响也非常大。


图 2-2 RPC 调用性能三要素

## 2.2. Netty 高性能之道

### 2.2.1. 异步非阻塞通信

在 IO 编程过程中，当需要同时处理多个客户端接入请求时，可以利用多线程或者 IO 多路复用技术进行处理。IO 多路复用技术通过把多个 IO 的阻塞复用到同一个 select 的阻塞上，从而使得系统在单线程的情况下可以同时处理多个客户端请求。与传统的多线程/多进程模型比，I/O 多路复用的最大优势是系统开销小，系统不需要创建新的额外进程或者线程，也不需要维护这些进程和线程的运行，降低了系统的维护工作量，节省了系统资源。

JDK1.4 提供了对非阻塞 IO（NIO）的支持，JDK1.5_update10 版本使用 epoll 替代了传统的 select/poll，极大的提升了 NIO 通信的性能。

JDK NIO 通信模型如下所示：

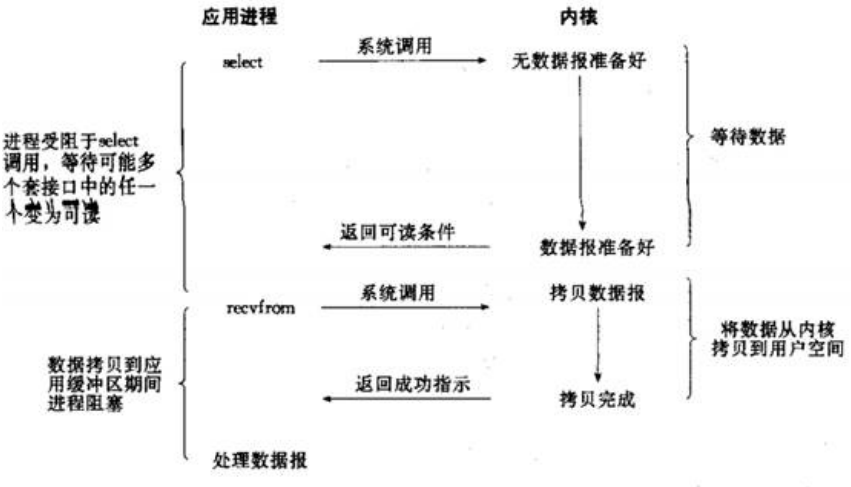


图 2-3 NIO 的多路复用模型图

与 Socket 类和 ServerSocket 类相对应，NIO 也提供了 SocketChannel 和 ServerSocketChannel 两种不同的套接字通道实现。这两种新增的通道都支持阻塞和非阻塞两种模式。阻塞模式使用非常简单，但是性能和可靠性都不好，非阻塞模式正好相反。开发人员一般可以根据自己的需要来选择合适的模式，一般来说，低负载、低并发的应用程序可

以选择同步阻塞 IO 以降低编程复杂度。但是对于高负载、高并发的网络应用，需要使用 NIO 的非阻塞模式进行开发。

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：

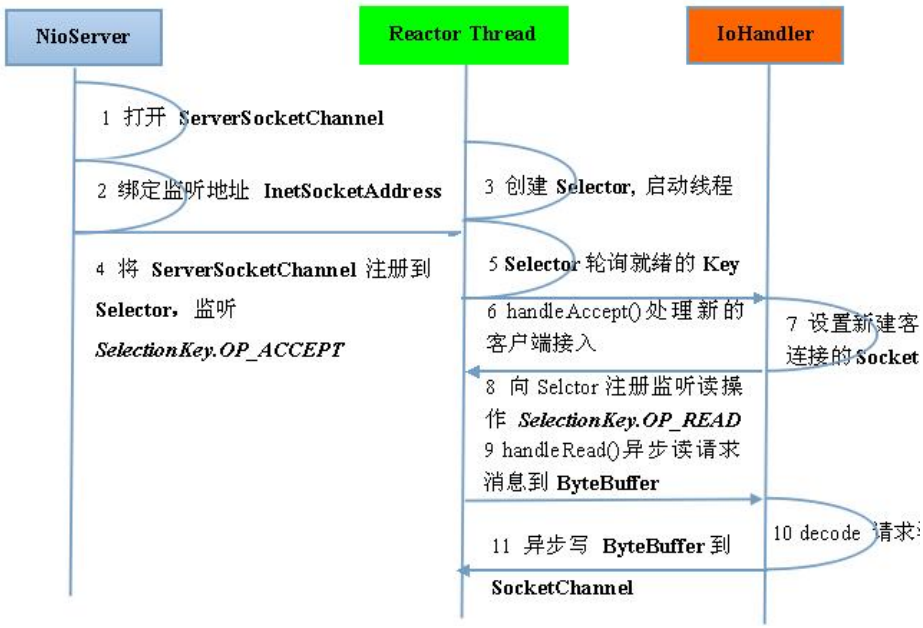


图 2-3 NIO 服务端通信序列图

客户端通信序列图如下：

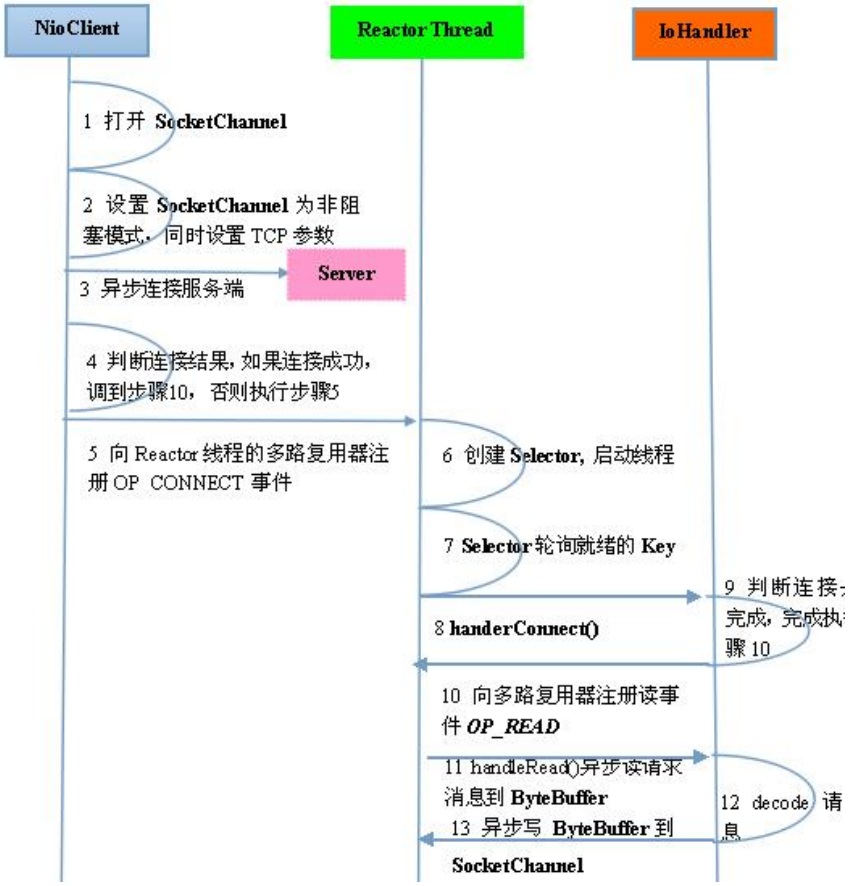


图 2-4 NIO 客户端通信序列图

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：

```
@Override
public void read() {
    final ChannelConfig config = config();
    final ChannelPipeline pipeline = pipeline();
    final ByteBufAllocator allocator = config.getAllocator();
    final int maxMessagesPerRead = config.getMaxMessagesPerRead();
    RecvByteBufAllocator.Handle allocHandle = this.allocHandle;
    if (allocHandle == null) {
        this.allocHandle = allocHandle = config.getRecvByteBufAllocator().newHandle();
    }
    if (!config.isAutoRead()) {
        removeReadOp();
    }

    ByteBuf byteBuf = null;
    int messages = 0;
    boolean close = false;
    try {
        int byteBufCapacity = allocHandle.guess();
        int totalReadAmount = 0;
        do {
            byteBuf = allocator.ioBuffer(byteBufCapacity);
```

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：

```
ByteBuf io.netty.buffer.ByteBufAllocator.ioBuffer(int initialCapacity)

Allocate a ByteBuf, preferably a direct buffer which is suitable for I/O.

Parameters:
      initialCapacity
```

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：

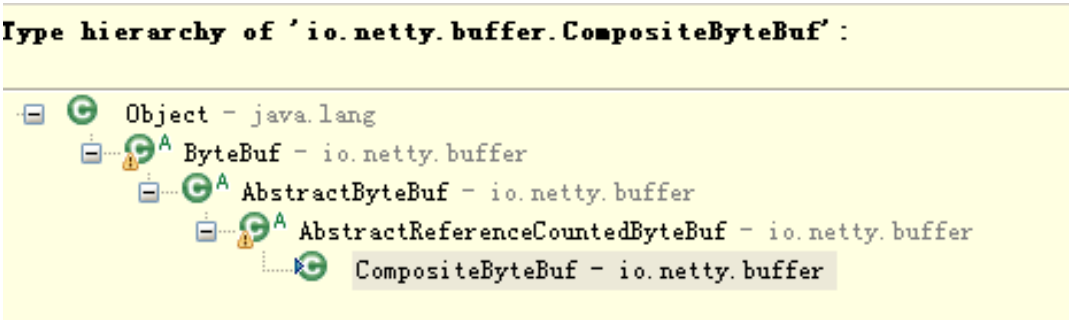


图 2-7 CompositeByteBuf 类继承关系

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：

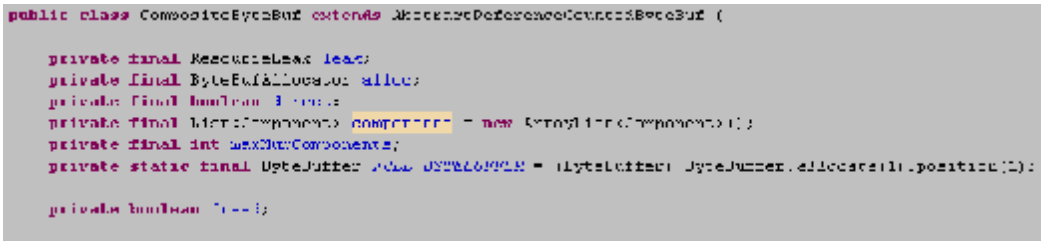

图 2-8 CompositeByteBuf 类定义

添加 ByteBuf，不需要做内存拷贝，相关代码如下：

```
private int addComponent0(int cIndex, ByteBuf buffer) {
    checkComponentIndex(cIndex);

    if (buffer == null) {
        throw new NullPointerException("buffer");
    }

    int readableBytes = buffer.readableBytes();
    if (readableBytes == 0) {
        return cIndex;
    }

    // No need to consolidate - just add a component to the list.
    Component c = new Component(buffer.order(ByteOrder.BIG_ENDIAN).slice());
    if (cIndex == components.size()) {
        components.add(c);
        if (cIndex == 0) {
            c.endOffset = readableBytes;
        } else {
            Component prev = components.get(cIndex - 1);
            c.offset = prev.endOffset;
            c.endOffset = c.offset + readableBytes;
        }
    } else {
        components.add(cIndex, c);
        updateComponentOffsets(cIndex);
    }
    return cIndex;
```

图 2-9 新增 ByteBuf 的“零拷贝”

最后，我们看下文件传输的“零拷贝”：

```
@Override
public long transferTo(WritableByteChannel target, long position) throws IOException {
    long count = this.count - position;
    if (count < 0 || position < 0) {
        throw new IllegalArgumentException(
                "position out of range: " + position +
                " (expected: 0 - " + (this.count - 1) + ')');
    }
    if (count == 0) {
        return 0L;
    }

    long written = file.transferTo(this.position + position, count, target);
    if (written > 0) {
        transfered += written;
    }
    return written;
}
```

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：

**long java.nio.channels.FileChannel.transferTo(long position, long count, WritableByteChannel target) throws IOException**

Transfers bytes from this channel's file to the given writable byte channel.

An attempt is made to read up to `count` bytes starting at the given `position` in this channel's file and write them to the target channel. An invocation of this method may or may not transfer all of the requested bytes; whether or not it does so depends upon the natures and states of the channels. Fewer than the requested number of bytes are transferred if this channel's file contains fewer than `count` bytes starting at the given `position`, or if the target channel is non-blocking and it has fewer than `count` bytes free in its output buffer.

This method does not modify this channel's position. If the given position is greater than the file's current size then no bytes are transferred. If the target channel has a position then bytes are written starting at that position and then the position is incremented by the number of bytes written.

This method is potentially much more efficient than a simple loop that reads from this channel and writes to the target channel. Many operating systems can transfer bytes directly from the filesystem cache to the target channel without actually copying them.

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：

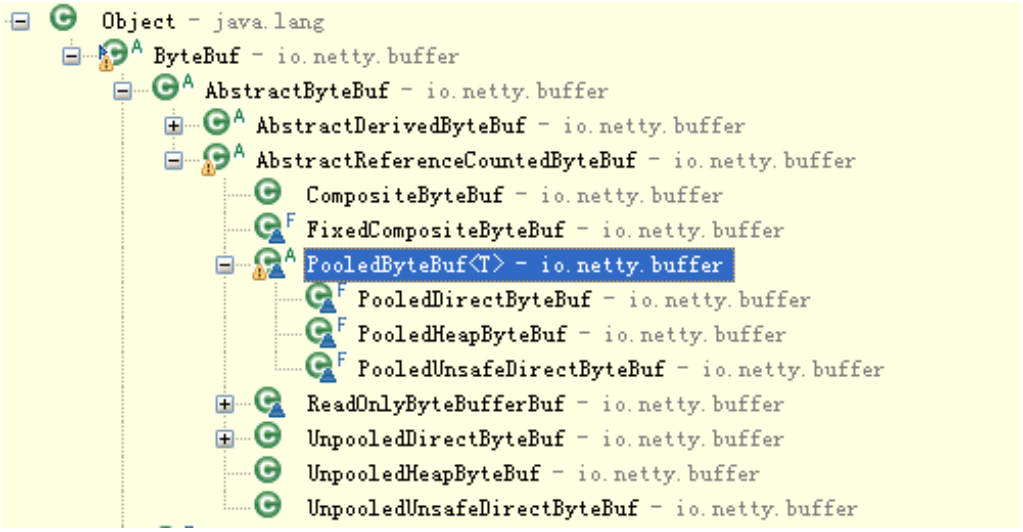


图 2-12 内存池 ByteBuf

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：

```
int loop = 3000000;
long startTime = System.currentTimeMillis();
ByteBuf poolBuffer = null;
for (int i = 0; i < loop; i++) {
    poolBuffer = PooledByteBufAllocator.DEFAULT.directBuffer(1024);
    poolBuffer.writeBytes(CONTENT);
    poolBuffer.release();
}
```

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：

```
long startTime2 = System.currentTimeMillis();
ByteBuf buffer = null;
for (int i = 0; i < loop; i++) {
    buffer = Unpooled.directBuffer(1024);
    buffer.writeBytes(CONTENT);
}
```

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：

```
The PooledByteBuf execute 300W times writing operation cost time is : 4125 ms
================================================================================
The unPooledByteBuf execute 300W times writing operation cost time is : 95312 ms
```

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：

```
@Override
public ByteBuf directBuffer(int initialCapacity, int maxCapacity) {
    if (initialCapacity == 0 && maxCapacity == 0) {
        return emptyBuf;
    }
    validate(initialCapacity, maxCapacity);
    return newDirectBuffer(initialCapacity, maxCapacity);
}
```

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：

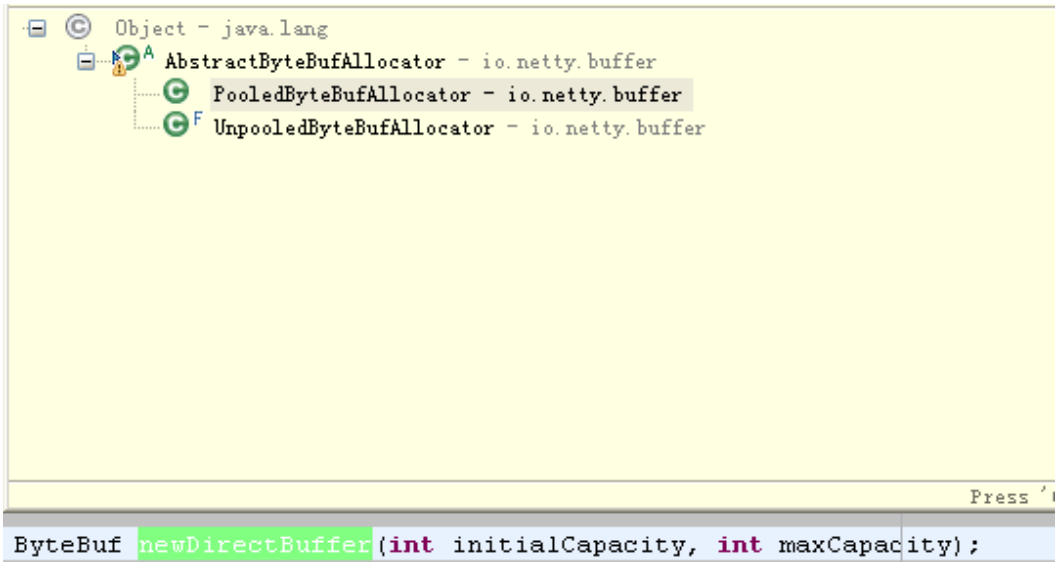


图 2-17 newDirectBuffer 的不同实现

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：

```
@Override
protected ByteBuf newDirectBuffer(int initialCapacity, int maxCapacity) {
    PoolThreadCache cache = threadCache.get();
    PoolArena<ByteBuffer> directArena = cache.directArena;

    ByteBuf buf;
    if (directArena != null) {
        buf = directArena.allocate(cache, initialCapacity, maxCapacity);
    } else {
        if (PlatformDependent.hasUnsafe()) {
            buf = new UnpooledUnsafeDirectByteBuf(this, initialCapacity, maxCapacity);
        } else {
            buf = new UnpooledDirectByteBuf(this, initialCapacity, maxCapacity);
        }
    }

    return toLeakAwareBuffer(buf);
}
```

图 2-18 PooledByteBufAllocator 的内存分配

PoolArena 的 allocate 方法如下：

```
PooledByteBuf<T> allocate(PoolThreadCache cache, int reqCapacity,
    PooledByteBuf<T> buf = newByteBuf(maxCapacity);
    allocate(cache, buf, reqCapacity);
    return buf;
}
```

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，

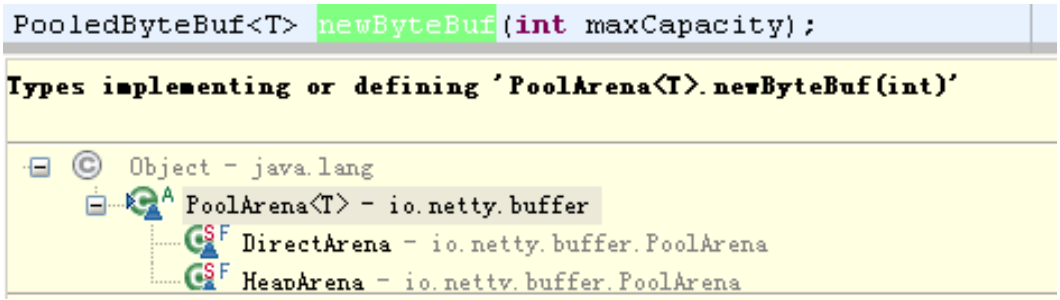


图 2-19 PoolArena 的 newByteBuf 抽象方法

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

```
@Override
protected PooledByteBuf<ByteBuffer> newByteBuf(int maxCapacity) {
    if (HAS_UNSAFE) {
        return PooledUnsafeDirectByteBuf.newInstance(maxCapacity);
    } else {
        return PooledDirectByteBuf.newInstance(maxCapacity);
    }
}
```

图 2-20 DirectArena 的 newByteBuf 方法实现

执行 PooledDirectByteBuf 的 newInstance 方法，代码如下：

```
static PooledDirectByteBuf newInstance(int maxCapacity) {
    PooledDirectByteBuf buf = RECYCLER.get();
    buf.setRefCnt(1);
    buf.maxCapacity(maxCapacity);
    return buf;
}
```

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：

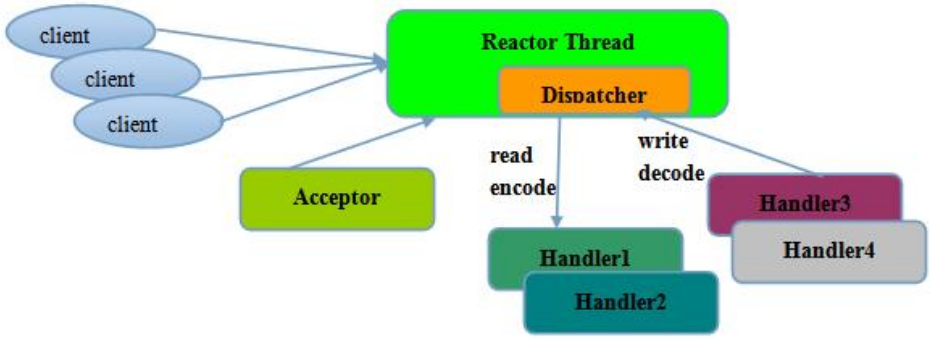


图 2-22 Reactor 单线程模型

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：

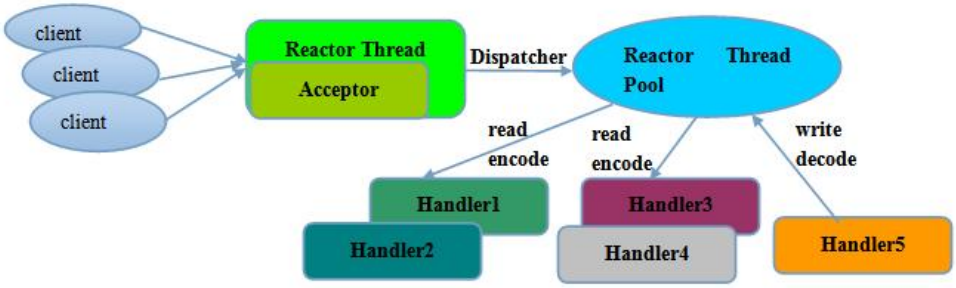


图 2-23 Reactor 多线程模型

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：

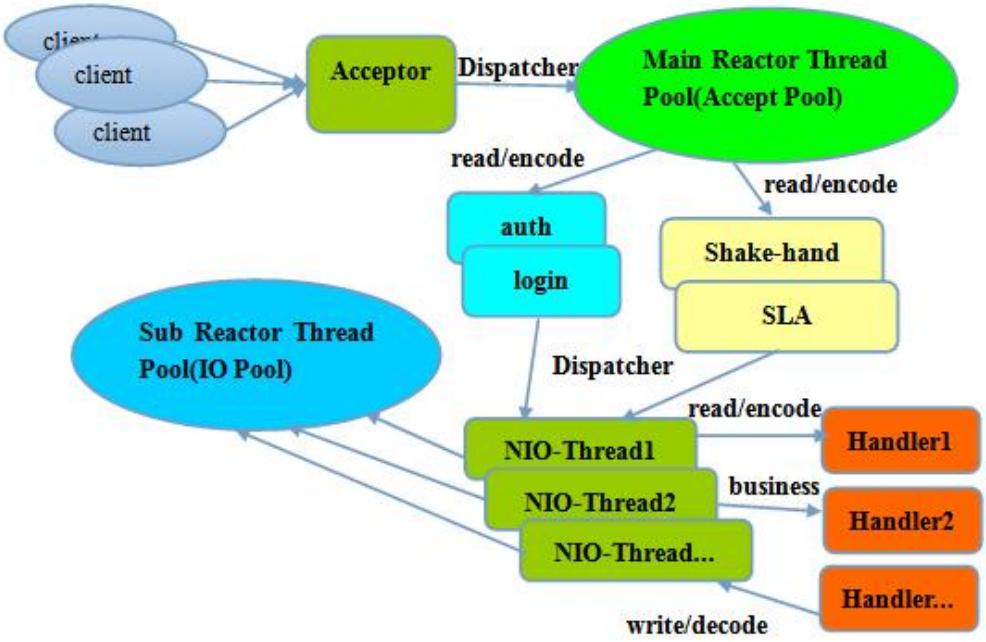


图 2-24 Reactor 主从多线程模型

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：

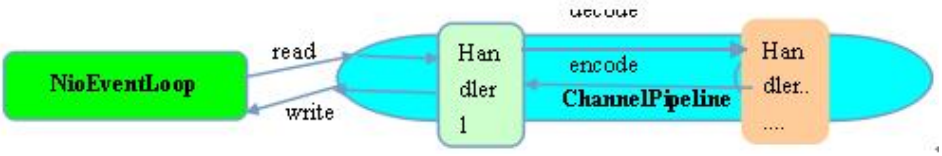

图 2-25 Netty 串行化工作原理图

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。

下面我们一起看下不同序列化&反序列化框架序列化后的字节数组对比：

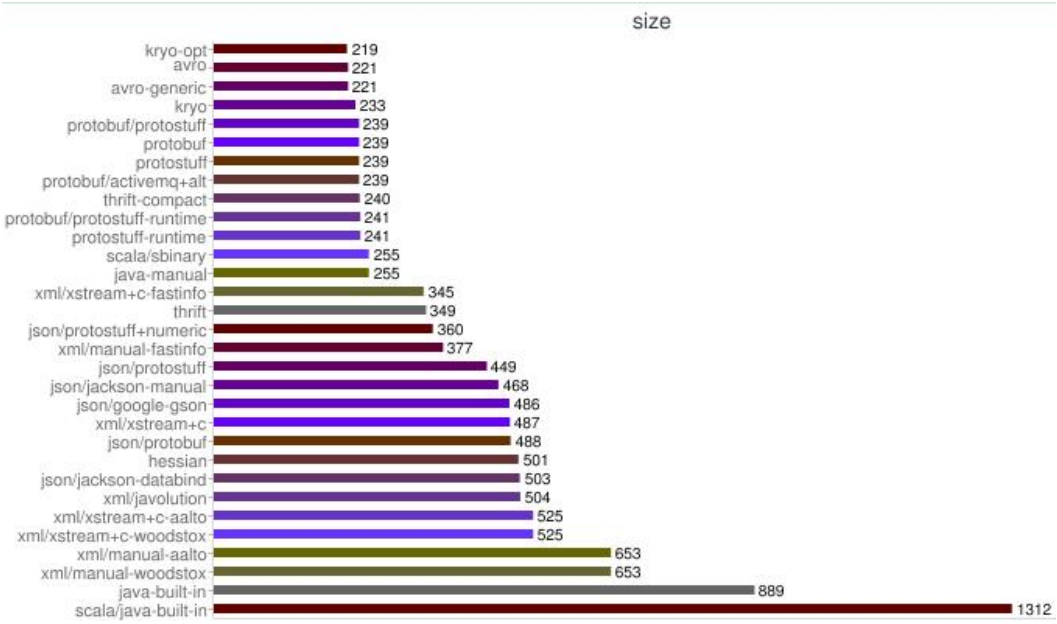


图 2-26 各序列化框架序列化码流大小对比

从上图可以看出，Protobuf 序列化后的码流只有 Java 序列化的 1/4 左右。正是由于 Java 原生序列化性能表现太差，才催生出了各种高性能的开源序列化技术和框架（性能差只是其中的一个原因，还有跨语言、IDL 定义等其它因素）。

### 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：

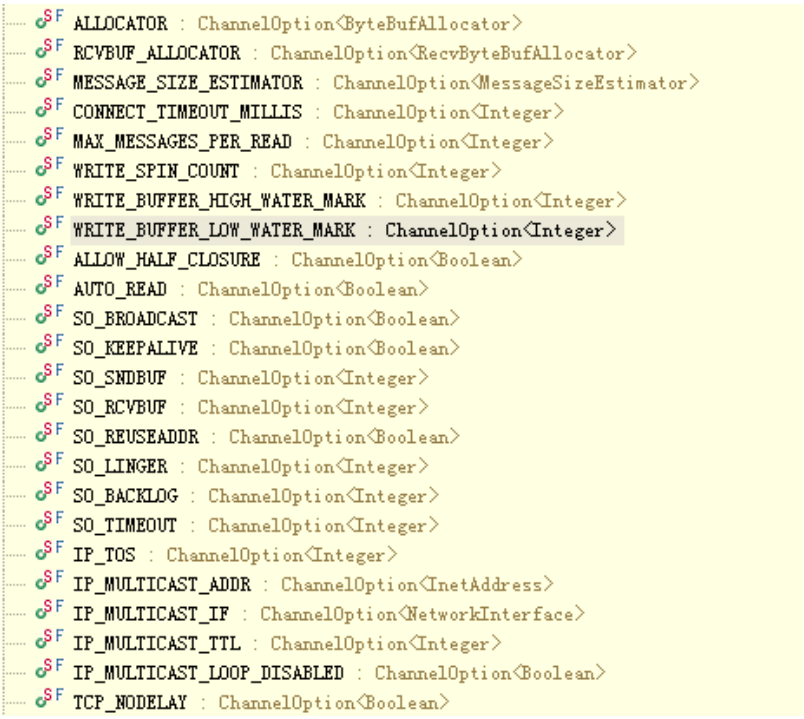


图 2-27 Netty 的 TCP 参数配置定义

## 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。

# 3. 作者简介

李林锋，2007 年毕业于东北大学，2008 年进入华为公司从事高性能通信软件的设计和开发工作，有 6 年 NIO 设计和开发经验，精通 Netty、Mina 等 NIO 框架。Netty 中国社区创始人，《Netty 权威指南》作者。