**同步**和**异步**关注的是**消息通信机制**

同步：就是在发出一个\*调用\*时，在没有得到结果之前，该\*调用\*就不返回。但是一旦调用返回，就得到返回值了。

异步：\*调用\*在发出之后，这个调用就直接返回了，所以没有返回结果。换句话说，当一个异步过程调用发出后，调用者不会立刻得到结果。而是在\*调用\*发出后，\*被调用者\*通过状态、通知来通知调用者，或通过回调函数处理这个调用。

**阻塞**和**非阻塞**关注的是**程序在等待调用结果**（消息，返回值）**时的状态**.

阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。

非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。

案例：

老张爱喝茶，废话不说，煮开水。

出场人物：老张，水壶两把（普通水壶，简称水壶；会响的水壶，简称响水壶）。

1 老张把水壶放到火上，立等水开。（同步阻塞）

老张觉得自己有点傻

2 老张把水壶放到火上，去客厅看电视，时不时去厨房看看水开没有。（同步非阻塞）

老张还是觉得自己有点傻，于是变高端了，买了把会响笛的那种水壶。水开之后，能大声发出嘀~~~~的噪音。

3 老张把响水壶放到火上，立等水开。（异步阻塞）

老张觉得这样傻等意义不大

4 老张把响水壶放到火上，去客厅看电视，水壶响之前不再去看它了，响了再去拿壶。（异步非阻塞）

**Netty高性能**

**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**。

**Netty高性能之道**

**RPC调用的性能模型分析**

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：

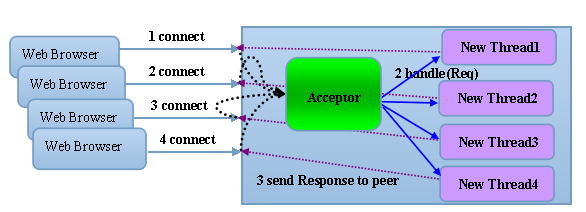


图2-1 BIO通信模型图

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）。

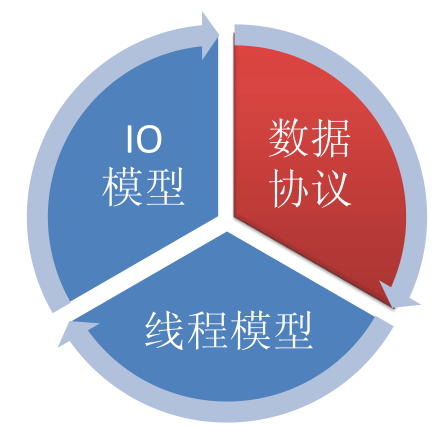
线程模型问题：由于采用同步阻塞IO，这会导致每个TCP连接都占用1个线程，由于线程资源是JVM虚拟机非常宝贵的资源，当IO读写阻塞导致线程无法及时释放时，会导致系统性能急剧下降，严重的甚至会导致虚拟机无法创建新的线程。

**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。

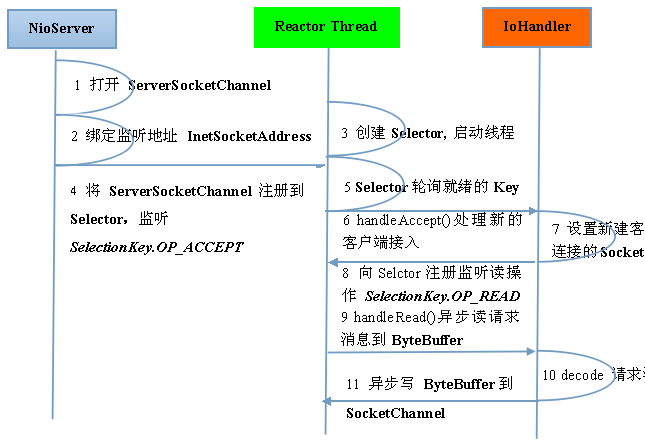
3) **线程**：数据报如何读取？读取之后的编解码在哪个线程进行，编解码后的消息如何派发，Reactor线程模型的不同，对性能的影响也非常大。



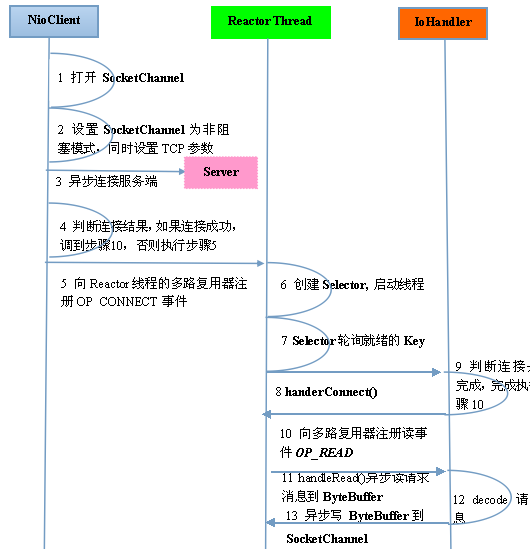
RPC调用性能三要素

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：



**客户端通信序列图**如下：



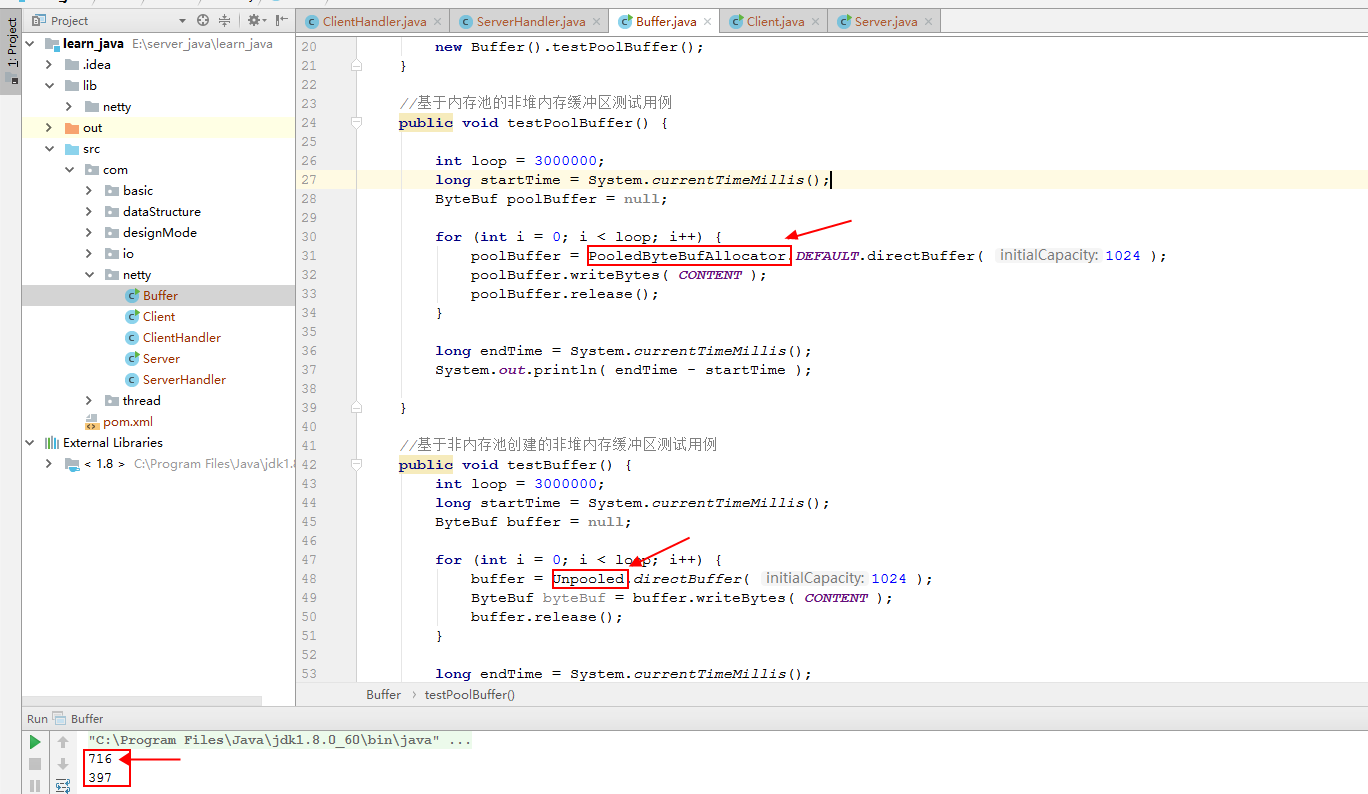
**零拷贝**

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。

**内存池**



测试结果相近，和预想的不一致？正常应该内存池效率高于不使用内存池的情况（20倍的效率提升）。

**案例代码路径：https://github.com/ben201708/learn\_java/tree/master/src/com/netty**

**测试结果如下**

