当客户端和服务器使用TCP协议进行通信时，客户端封装一个请求对象req，将请求对象req序列化成字节数组，然后通过套接字socket将字节数组发送到服务器，服务器通过套接字socket读取到字节数组，再反序列化成请求对象req，进行处理，处理完毕后，生成一个响应对应res，将响应对象res序列化成字节数组，然后通过套接字将自己数组发送给客户端，客户端通过套接字socket读取到自己数组，再反序列化成响应对象。

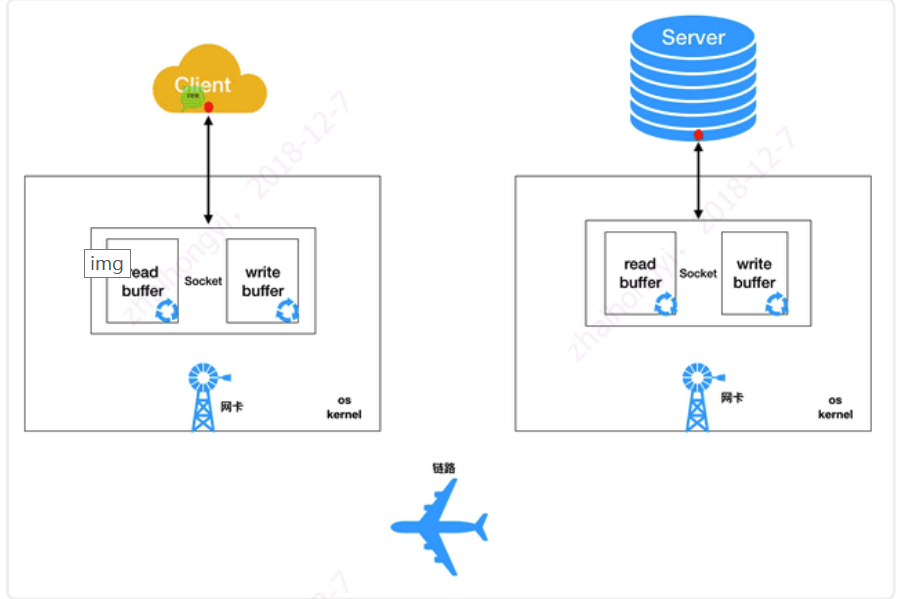
通信框架往往可以将序列化的过程隐藏起来，请求对象req和响应对象res在客户端和服务器之间跑来跑去。

也许你觉得这个过程还是挺简单的，很好理解，但是实际上背后发生的一系列事件超出了你们中大多数人的想象。通信的真实过程要比上面的这张图复杂太多。你也许会问，我们需要了解的那么深入么，直接拿来用不就可以了么？

在互联网技术服务行业工作多年的经验告诉我，如果你对底层机制不了解，你就会不明白为什么对套接字socket的读写会出现各种奇奇乖乖的问题，为什么有时会阻塞，有时又不阻塞，有时候还报错，为什么会有粘包半包问题，NIO具体又是什么，它是什么特别新鲜的技术么？对于这些问题的理解都需要你了解底层机制。

# **细节过程**

为了方便大家对通信底层的理解，我花了些时间做了下面这个动画，它并不能完全覆盖底层细节的全貌，但是对于理解套接字的工作机制已经足够了。请读者仔细观察这个动画，后面的讲解将围绕着这个动画展开。



我们平时用到的套接字其实只是一个引用(一个对象ID)，这个套接字对象实际上是放在操作系统内核中。这个套接字对象内部有两个重要的缓冲结构，一个是读缓冲(read buffer)，一个是写缓冲(write buffer)，它们都是有限大小的数组结构。

当我们对客户端的socket写入字节数组时(序列化后的请求消息对象req)，是将字节数组拷贝到内核区套接字对象的write buffer中，内核网络模块会有单独的线程负责不停地将write buffer的数据拷贝到网卡硬件，网卡硬件再将数据送到网线，经过一些列路由器交换机，最终送达服务器的网卡硬件中。

同样，服务器内核的网络模块也会有单独的线程不停地将收到的数据拷贝到套接字的read buffer中等待用户层来读取。最终服务器的用户进程通过socket引用的read方法将read buffer中的数据拷贝到用户程序内存中进行反序列化成请求对象进行处理。然后服务器将处理后的响应对象走一个相反的流程发送给客户端，这里就不再具体描述。

**阻塞**

我们注意到write buffer空间都是有限的，所以如果应用程序往套接字里写的太快，这个空间是会满的。一旦满了，写操作就会阻塞，直到这个空间有足够的位置腾出来。不过有了NIO(非阻塞IO)，写操作也可以不阻塞，能写多少是多少，通过返回值来确定到底写进去多少，那些没有写进去的内容用户程序会缓存起来，后续会继续重试写入。

同样我们也注意到read buffer的内容可能会是空的。这样套接字的读操作(一般是读一个定长的字节数组)也会阻塞，直到read buffer中有了足够的内容(填充满字节数组)才会返回。有了NIO，就可以有多少读多少，无须阻塞了。读不够的，后续会继续尝试读取。

**ack**

那上面这张图就展现了套接字的全部过程么？显然不是，数据的确认过程(ack)就完全没有展现。比如当写缓冲的内容拷贝到网卡后，是不会立即从写缓冲中将这些拷贝的内容移除的，而要等待对方的ack过来之后才会移除。如果网络状况不好，ack迟迟不过来，写缓冲很快就会满的。

**包头**

内核的网络模块会将缓冲区的消息进行分块传输，如果缓冲区的内容太大，是会被拆分成多个独立的小消息包的。并且还要在每个消息包上附加上一些额外的头信息，比如源网卡地址和目标网卡地址、消息的序号等信息，到了接收端需要对这些消息包进行重新排序组装去头后才会扔进读缓冲中。这些复杂的细节过程就非常难以在动画上予以呈现了。

**速率**

还有个问题那就是如果读缓冲满了怎么办，网卡收到了对方的消息要怎么处理？一般的做法就是丢弃掉不给对方ack，对方如果发现ack迟迟没有来，就会重发消息。那缓冲为什么会满？是因为消息接收方处理的慢而发送方生产的消息太快了，这时候tcp协议就会有个动态窗口调整算法来限制发送方的发送速率，使得收发效率趋于匹配。如果是udp协议的话，消息一丢那就彻底丢了。