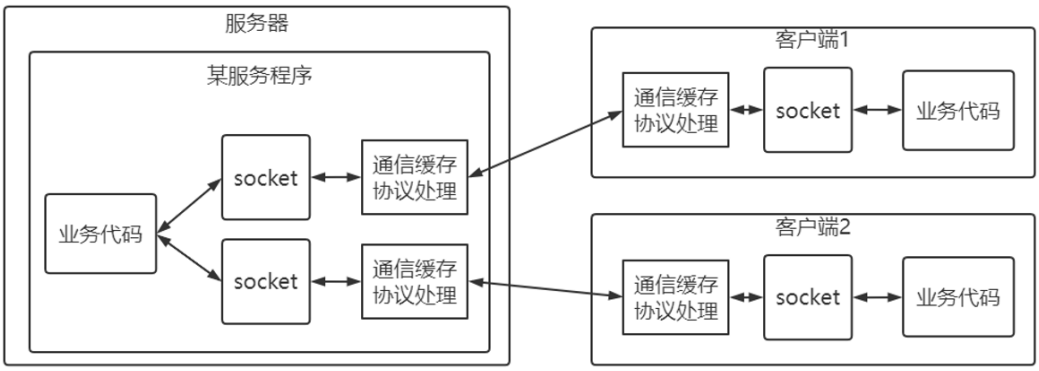
# 网络通信编程基本常识

## 什么是 Socket?

**Socket** 是应用层与 TCP/IP 协议族通信的中间软件抽象层，它是一组接口，一般由操作系统提供。在设计模式中，Socket 其实就是一个门面模式，它把复杂的 TCP/IP 协议处理和通信缓存管理等等都隐藏在 Socket 接口后面，对用户来说，使用一组简单的接口就能进行网络应用编程，让 Socket 去组织数据，以符合指定的协议。主机 A 的应用程序要能和主机 B 的应用程序通信，必须通过 Socket 建立连接。

客户端连接上一个服务端，就会在客户端中产生一个 socket 接口实例，服务端每接受一个客户端连接，就会产生一个 socket 接口实例和客户端的 socket 进行通信，有多个客户端连接自然就有多个 socket 接口实例。




## 短连接

连接->传输数据->关闭连接

传统 HTTP 是无状态的，浏览器和服务器每进行一次 HTTP 操作，就建立一次连接，但任务结束就中断连接。

也可以这样说：短连接是指 SOCKET 连接后发送后接收完数据后马上断开连接。

## 长连接

连接->传输数据->保持连接 -> 传输数据-> 。。。 ->关闭连接。

长连接指建立 SOCKET 连接后不管是否使用都保持连接。

## 什么时候用长连接，短连接?

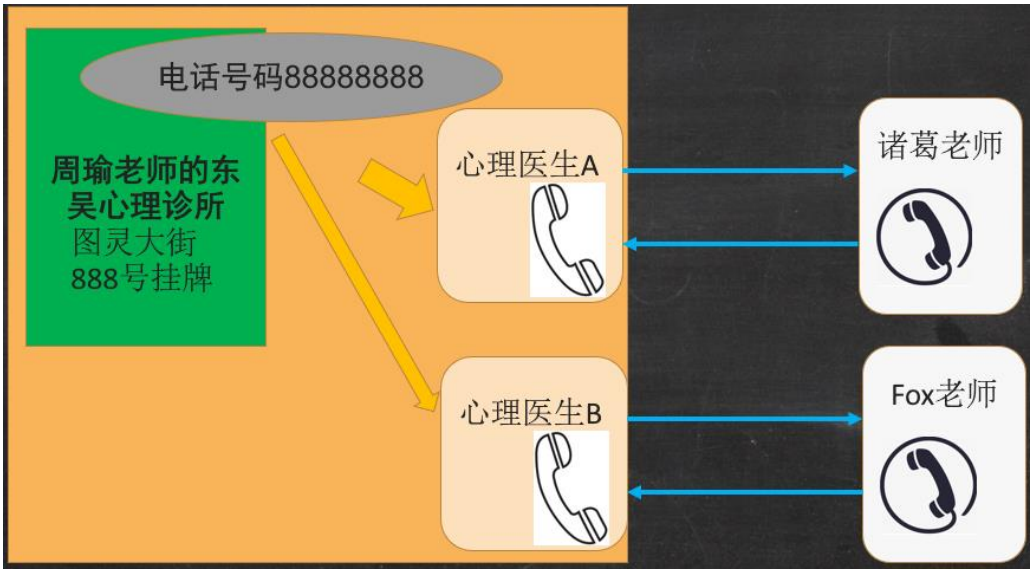
长连接多用于操作频繁，点对点的通讯。每个 TCP 连接都需要三步握手，这需要时间，如果每个操作都是先连接，再操作的话那么处理速度会降低很多，所以每个操作完后都不断开，下次处理时直接发送数据包就 OK 了，不用建立 TCP 连接。例如：数据库的连接用长连接， 如果用短连接频繁的通信会造成 socket 错误，而且频繁的 socket 创建也是对资源的浪费。

而像 WEB 网站的 http 服务按照 Http 协议规范早期一般都用短链接，因为长连接对于服务端来说会耗费一定的资源，而像 WEB 网站这么频繁的成千上万甚至上亿客户端的连接用短连接会更省一些资源。但是现在的 Http 协议，Http1.1，尤其是 Http2、Http3 已经开始向长连接演化。

总之，长连接和短连接的选择要视情况而定。

# 网络编程里通用常识

我们首先来看一个生活中的场景。周瑜老师准备开一个心理咨询中心，嘴上光喊没用，只有到工商局注册“东吴心理诊所”并且在图灵大街 888 号挂牌了，才算正式开张。疫情来了，准备开展电话业务，申请了一个电话号码 88888888。诸葛老师有了心理问题，于是打电话过来，周瑜老师接了电话，但是周瑜老师不懂心理咨询，于是通过内部分机把电话转给请来的心理医生 A 负责接待诸葛老师，心理医生 A 和诸葛老师通过电话进行沟通，模式一般就是一个人说另个一人听，两者进行沟通交流。Fox 老师也来了，周瑜老师接了电话，又把电话转给请来的心理医生 B 负责接待 Fox 老师，心理医生 B 和 Fox 老师也通过电话进行沟通。




上述的场景和网络编程有很大的相似之处。

我们已经知道在通信编程里提供服务的叫服务端，连接服务端使用服务的叫客户端。在开发过程中，如果类的名字有 Server 或者 ServerSocket 的，表示这个类是给服务端容纳网络服务用的，如果类的名字只包含 Socket 的，那么表示这是负责具体的网络读写的。

那么对于服务端来说 ServerSocket 就只是个场所，就像上面的“东吴心理诊所”，它必须要绑定某个 IP 地址，就像“东吴心理诊所”在“图灵大街 888 号挂牌”，同时 ServerSocket 还需要监听某个端口，就像“申请了一个电话号码 88888888”。

有电话进来了，具体和客户端沟通的还是一个一个的 socket，就像“周瑜老师不懂心理咨询，于是通过内部分机把电话转给请来的心理医生 A 负责接待诸葛老师”，所以在通信编程里，ServerSocket 并不负责具体的网络读写，ServerSocket 就只是负责接收客户端连接后，新启一个 socket 来和客户端进行沟通。这一点对所有模式的通信编程都是适用的。

在通信编程里，我们关注的其实也就是三个事情：连接（客户端连接服务器，服务器等待和接收连接）、读网络数据、写网络数据，所有模式的通信编程都是围绕着这三件事情进

行的。服务端提供 IP 和监听端口，客户端通过连接操作想服务端监听的地址发起连接请求，通过三次握手连接，如果连接成功建立，双方就可以通过套接字进行通信。

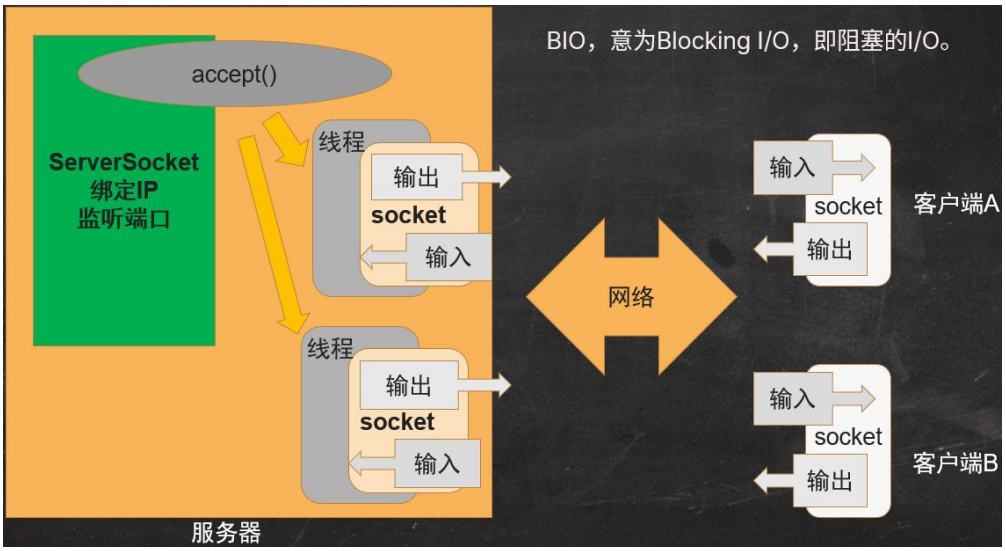
我们后面将学习的 BIO 和 NIO 其实都是处理上面三件事，只是处理的方式不一样。

# Java 原生网络编程-BIO

## 原生 JDK 网络编程 BIO

BIO，意为 Blocking I/O，即阻塞的 I/O。

BIO 基本上就是我们上面所说的生活场景的朴素实现。在 BIO 中类 ServerSocket 负责绑定 IP 地址，启动监听端口，等待客户连接；客户端 Socket 类的实例发起连接操作，ServerSocket 接受连接后产生一个新的服务端 socket 实例负责和客户端 socket 实例通过输入和输出流进行通信。




bio 的阻塞，主要体现在两个地方。

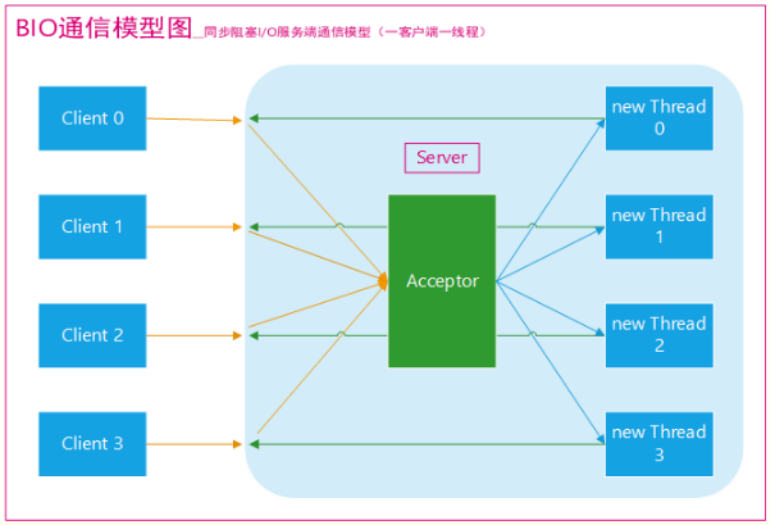
①若一个服务器启动就绪，那么主线程就一直在等待着客户端的连接，这个等待过程中主线程就一直在阻塞。

②在连接建立之后，在读取到 socket 信息之前，线程也是一直在等待，一直处于阻塞的状态下的。

这一点可以通过 cn.tuling.bio 下的 ServerSingle.java 服务端程序看出，启动该程序后，启动一个 Client 程序实例，并让这个 Client 阻塞住，位置就在向服务器输出具体请求之前，再启动一个新的 Client 程序实例，会发现尽管新的 Client 实例连接上了服务器，但是 ServerSingle 服务端程序仿佛无感知一样？为何，因为执行的主线程被阻塞了一直在等待第一个 Client 实例发送消息过来。

所以在 BIO 通信里，我们往往会在服务器的实现上结合线程来处理连接以及和客户端的通信。

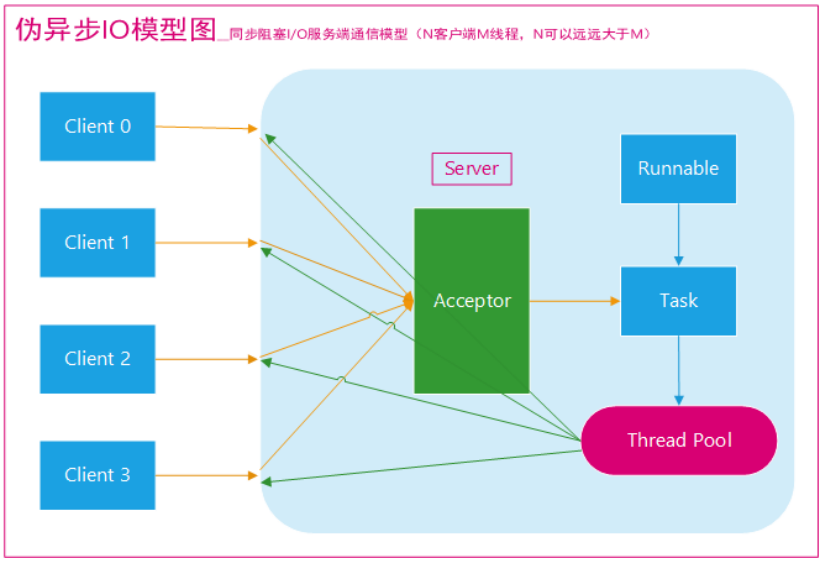
传统 BIO 通信模型：采用 BIO 通信模型的服务端，通常由一个独立的 Acceptor 线程负责监听客户端的连接，它接收到客户端连接请求之后为每个客户端创建一个新的线程进行链路处理，处理完成后，通过输出流返回应答给客户端，线程销毁。即典型的一请求一应答模型，同时数据的读取写入也必须阻塞在一个线程内等待其完成。代码可见 cn.tuling.bio.Server。




该模型最大的问题就是缺乏弹性伸缩能力，当客户端并发访问量增加后，服务端的线程个数和客户端并发访问数呈 1:1 的正比关系，Java 中的线程也是比较宝贵的系统资源，线程数量快速膨胀后，系统的性能将急剧下降，随着访问量的继续增大，系统最终就**死-掉-了**。

为了改进这种一连接一线程的模型，我们可以使用线程池来管理这些线程，实现 1 个或多个线程处理 N 个客户端的模型（但是底层还是使用的同步阻塞 I/O），通常被称为“伪异步 I/O 模型“。

我们知道，如果使用 CachedThreadPool 线程池（不限制线程数量，如果不清楚请参考文首提供的文章），其实除了能自动帮我们管理线程（复用），看起来也就像是 1:1 的客户端：线程数模型，而使用 FixedThreadPool 我们就有效的控制了线程的最大数量，保证了系统有限的资源的控制，实现了 N:M 的伪异步 I/O 模型。代码可见 cn.tuling.bio.ServerPool。




但是，正因为限制了线程数量，如果发生读取数据较慢时（比如数据量大、网络传输慢等），大量并发的情况下，其他接入的消息，只能一直等待，这就是最大的弊端。

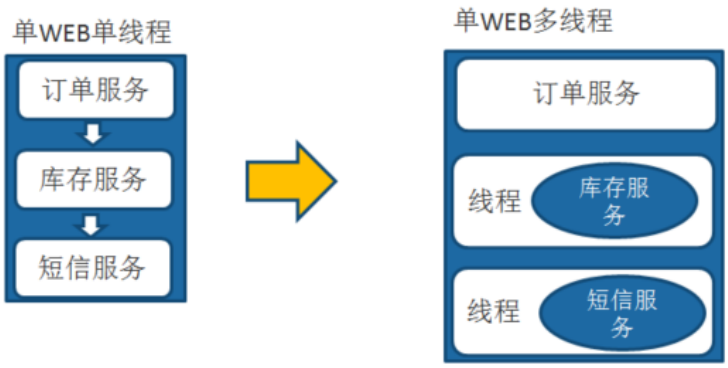
# 附录：BIO 实战-手写 RPC 框架

因为后面的 Dubbo 课程中，周瑜老师会带大家手写实现一个 RPC 框架，所以本节内容不会讲述，从笔记到代码都供大家自行学习和参考。

## *为什么要有 RPC？*

我们最开始开发的时候，一个应用一台机器，将所有功能都写在一起，比如说比较常见的电商场景，服务之间的调用就是我们最熟悉的普通本地方法调用。

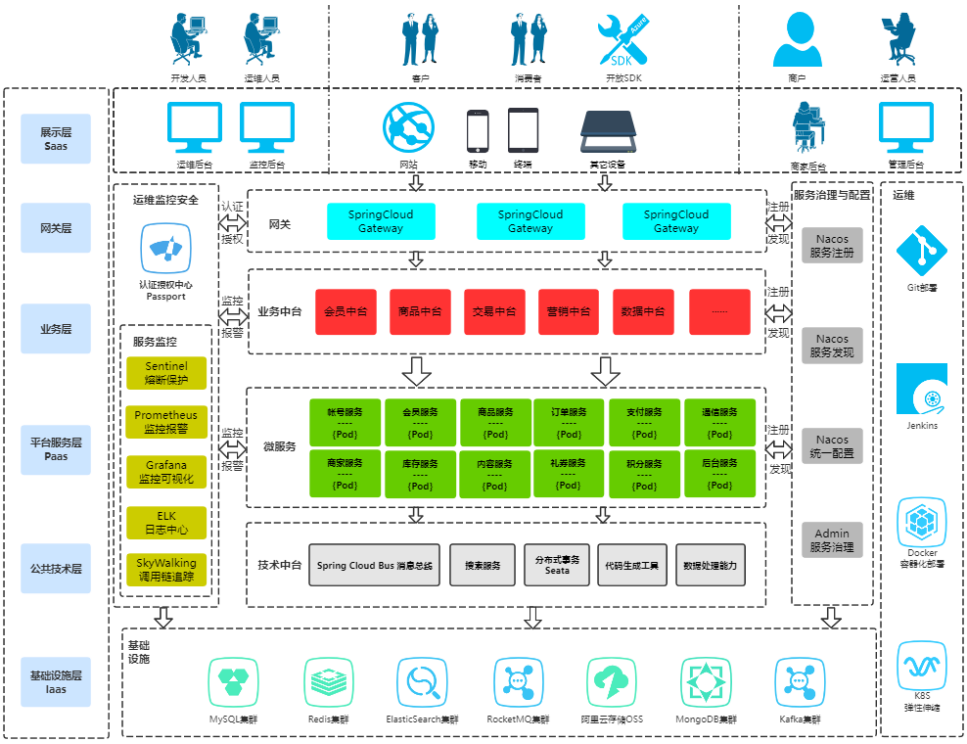
随着我们业务的发展，我们需要提示性能了，我们会怎么做？将不同的业务功能放到线程里来实现异步和提升性能，但本质上还是本地方法调用。




但是业务越来越复杂，业务量越来越大，单个应用或者一台机器的资源是肯定背负不起的，这个时候，我们会怎么做？将核心业务抽取出来，作为独立的服务，放到其他服务器上或者形成集群。这个时候就会请出 RPC，系统变为分布式的架构。

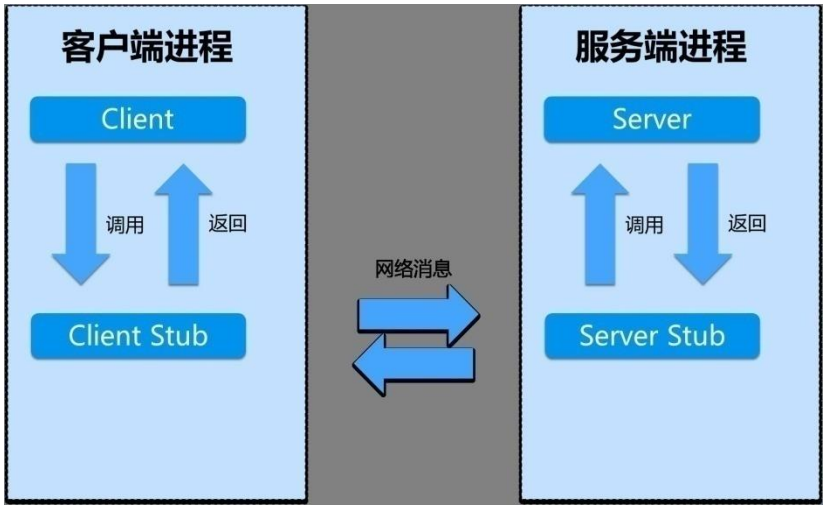
为什么说千万级流量分布式、微服务架构必备的 RPC 框架？和 LocalCall 的代码进行比较，因为引入 rpc 框架对我们现有的代码影响最小，同时又可以帮我们实现架构上的扩展。现在的开源 rpc 框架，有什么？dubbo，grpc 等等

当服务越来越多，各种 rpc 之间的调用会越来越复杂，这个时候我们会引入中间件，比如说 MQ、缓存，同时架构上整体往微服务去迁移，引入了各种比如容器技术 docker，DevOps 等等。最终会变为如图所示来应付千万级流量，但是不管怎样，rpc 总是会占有一席之地。




## 什么是RPC?

RPC（Remote Procedure Call ——远程过程调用），它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络的技术。




一次完整的 RPC 同步调用流程：

1）服务消费方（client）以本地调用方式调用客户端存根；

2）什么叫客户端存根？就是远程方法在本地的模拟对象，一样的也有方法名，也有方法参数，client stub 接收到调用后负责将方法名、方法的参数等包装，并将包装后的信息通过网络发送到服务端；

3）服务端收到消息后，交给代理存根在服务器的部分后进行解码为实际的方法名和参数

4） server stub 根据解码结果调用服务器上本地的实际服务；

5）本地服务执行并将结果返回给 server stub；

6）server stub 将返回结果打包成消息并发送至消费方；

7）client stub 接收到消息，并进行解码；

8）服务消费方得到最终结果。

RPC 框架的目标就是要中间步骤都封装起来，让我们进行远程方法调用的时候感觉到就像在本地方法调用一样。

## *RPC 和 HTTP*

rpc 字面意思就是远程过程调用，只是对不同应用间相互调用的一种描述，一种思想。具体怎么调用？实现方式可以是最直接的 tcp 通信，也可以是 http 方式，在很多的消息中间件的技术书籍里，甚至还有使用消息中间件来实现 RPC 调用的，我们知道的 dubbo 是基于 tcp 通信的，gRPC 是 Google 公布的开源软件，基于最新的 HTTP2.0 协议，底层使用到了 Netty 框架的支持。所以总结来说，rpc 和 http 是完全两个不同层级的东西，他们之间并没有什么可比性。

## *实现 RPC 框架*

### 实现 RPC 框架需要解决的那些问题

#### *代理问题*

代理本质上是要解决什么问题？要解决的是被调用的服务本质上是远程的服务，但是调用者不知道也不关心，调用者只要结果，具体的事情由代理的那个对象来负责这件事。既然是远程代理，当然是要用代理模式了。

代理(Proxy)是一种设计模式,即通过代理对象访问目标对象.这样做的好处是:可以在目标对象实现的基础上,增强额外的功能操作,即扩展目标对象的功能。那我们这里额外的功能操作是干什么，通过网络访问远程服务。

jdk 的代理有两种实现方式：静态代理和动态代理。

#### *序列化问题*

序列化问题在计算机里具体是什么？我们的方法调用，有方法名，方法参数，这些可能是字符串，可能是我们自己定义的 java 的类，但是在网络上传输或者保存在硬盘的时候，网络或者硬盘并不认得什么字符串或者 javabean，它只认得二进制的 01 串，怎么办？要进行序列化，网络传输后要进行实际调用，就要把二进制的 01 串变回我们实际的 java 的类，这个叫反序列化。java 里已经为我们提供了相关的机制 Serializable。

#### *通信问题*

我们在用序列化把东西变成了可以在网络上传输的二进制的 01 串，但具体如何通过网络传输？使用 JDK 为我们提供的 BIO。

#### *登记的服务实例化*

登记的服务有可能在我们的系统中就是一个名字，怎么变成实际执行的对象实例，当然是使用反射机制。

反射机制是什么？

反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为 java 语言的反射机制。

反射机制能做什么

反射机制主要提供了以下功能：

•在运行时判断任意一个对象所属的类；

•在运行时构造任意一个类的对象；

•在运行时判断任意一个类所具有的成员变量和方法；

•在运行时调用任意一个对象的方法；

•生成动态代理。

### 成型代码

参见工程 ketang-tl-rpc，其中

rpc-client 包含了 rpc 框架客户端的使用范例

rpc-reg 包含了 rpc 框架的远程注册中心的实现代码

rpc-server-sms 包含了 rpc 框架服务端的使用范例 sms 服务

rpc-server-stock 包含了 rpc 框架服务端的使用范例 stock 服务

rpc-netty-client 包含了基于 Netty 通信框架的 rpc 实现的客户端

rpc-netty-server 包含了基于 Netty 通信框架的 rpc 实现的服务端，采用了本地注册中心模式

# 原生 JDK 网络编程- NIO

## 什么是 NIO?

NIO 库是在 JDK 1.4 中引入的。NIO 弥补了原来的 BIO 的不足，它在标准 Java 代码中提供了高速的、面向块的 I/O。NIO 被称为 no-blocking io 或者 new io 都说得通。

### *和 BIO 的主要区别*

#### 面向流与面向缓冲

Java NIO 和 IO 之间第一个最大的区别是，IO 是面向流的，NIO 是面向缓冲区的。 Java IO 面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。 Java NIO 的缓冲导向方法略有不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所有需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。

### 阻塞与非阻塞 IO

Java IO 的各种流是阻塞的。这意味着，当一个线程调用 read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。

Java NIO 的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取。而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。 非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。线程通常将非阻塞 IO 的空闲时间用于在其它通道上执行 IO 操作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）。

## NIO 之 Reactor 模式

“反应”器名字中”反应“的由来：

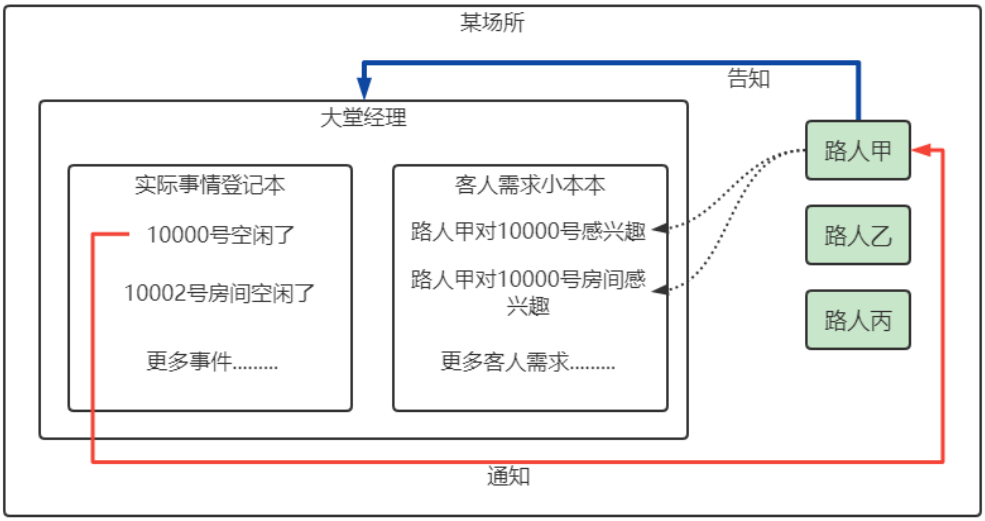
“反应”即“倒置”，“控制逆转”,具体事件处理程序不调用反应器，而向反应器注册一个事件处理器，表示自己对某些事件感兴趣，有时间来了，具体事件处理程序通过事件处理器对某个指定的事件发生做出反应；这种控制逆转又称为“好莱坞法则”（不要调用我，让我来调用你）

例如，路人甲去做男士 SPA，大堂经理负责服务，路人甲现在只对 10000 技师感兴趣，但是路人甲去的比较早，就告诉大堂经理，等 10000 技师上班了或者是空闲了，通知我。等路人甲接到大堂经理通知，做出了反应，把 10000 技师占住了。

然后，路人甲想起上一次的那个 10000 号房间不错，设备舒适，灯光暧昧，又告诉大堂经理，我对 10000 号房间很感兴趣，房间空出来了就告诉我，我现在先和 10000 这个小姐聊下人生，10000 号房间空出来了，路人甲再次接到大堂经理通知，路人甲再次做出了反应。

路人甲就是具体事件处理程序，大堂经理就是所谓的反应器，“10000 技师上班了”和“10000 号房间空闲了”就是事件，路人甲只对这两个事件感兴趣，其他，比如 10001 号技师或者 10002 号房间空闲了也是事件，但是路人甲不感兴趣。

大堂经理不仅仅服务路人甲这个人，他还可以同时服务路人乙、丙……，每个人所感兴趣的事件是不一样的，大堂经理会根据每个人感兴趣的事件通知对应的每个人。



# NIO 三大核心组件

NIO 有三大核心组件：Selector 选择器、Channel 管道、buffer 缓冲区。

## *Selector*

Selector 的英文含义是“选择器”，也可以称为为“轮询代理器”、“事件订阅器”、“channel 容器管理机”都行。

Java NIO 的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器(Selectors)，然后使用一个单独的线程来操作这个选择器，进而“选择”通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。

应用程序将向 Selector 对象注册需要它关注的 Channel，以及具体的某一个 Channel 会对哪些 IO 事件感兴趣。Selector 中也会维护一个“已经注册的 Channel”的容器。

## *Channels*

通道，被建立的一个应用程序和操作系统交互事件、传递内容的渠道（注意是连接到操作系统）。那么既然是和操作系统进行内容的传递，那么说明应用程序可以通过通道读取数据，也可以通过通道向操作系统写数据，而且可以同时进行读写。

- 所有被 Selector（选择器）注册的通道，只能是继承了 SelectableChannel 类的子类。
- ServerSocketChannel：应用服务器程序的监听通道。只有通过这个通道，应用程序才能向操作系统注册支持“多路复用 IO”的端口监听。同时支持 UDP 协议和 TCP 协议。
- ScoketChannel：TCP Socket 套接字的监听通道，一个 Socket 套接字对应了一个客户端 IP：端口 到 服务器 IP：端口的通信连接。

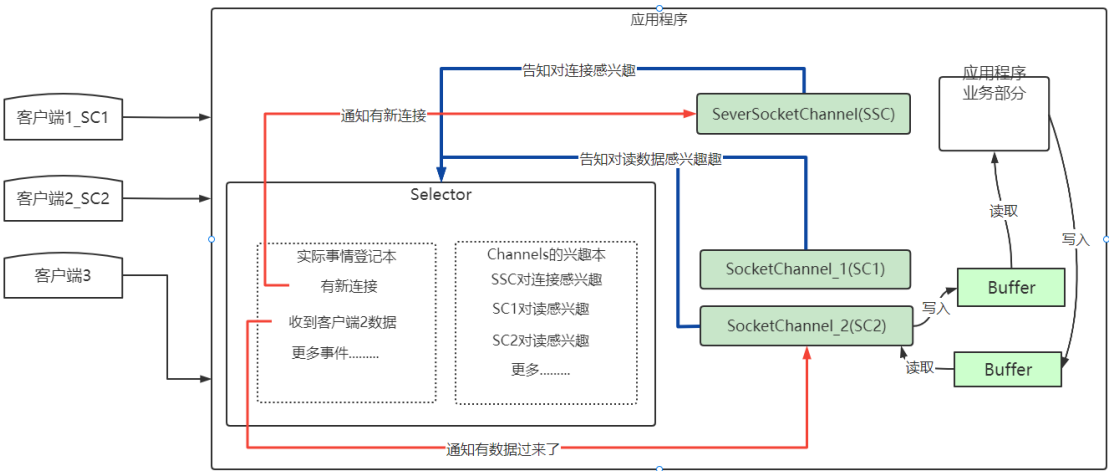
通道中的数据总是要先读到一个 Buffer，或者总是要从一个 Buffer 中写入。

## *buffer 缓冲区*

我们前面说过 JDK NIO 是面向缓冲的。Buffer 就是这个缓冲，用于和 NIO 通道进行交互。数据是从通道读入缓冲区，从缓冲区写入到通道中的。以写为例，应用程序都是将数据写入缓冲，再通过通道把缓冲的数据发送出去，读也是一样，数据总是先从通道读到缓冲，应用程序再读缓冲的数据。

缓冲区本质上是一块可以写入数据，然后可以从中读取数据的内存（其实就是数组）。这块内存被包装成 NIO Buffer 对象，并提供了一组方法，用来方便的访问该块内存。

后面的附录详细讲到其中的 api 等相关内容。




## 实现代码

```
//创建选择器
selector = Selector.open();
//打开监听通道
serverChannel = ServerSocketChannel.open();
//如果为 true，则此通道将被置于阻塞模式；
// 如果为 false，则此通道将被置于非阻塞模式
serverChannel.configureBlocking(false);//开启非阻塞模式
//绑定端口 backlog设为1024
serverChannel.socket()
        .bind(new InetSocketAddress(port), backlog: 1024);
//监听客户端连接请求
serverChannel.register(selector, SelectionKey.OP_ACCEPT);
```

1、

Selector 对象是通过调用静态工厂方法 open()来实例化的，如下：

Selector Selector=Selector.open();

2、

要实现 Selector 管理 Channel，需要将 channel 注册到相应的 Selector 上，如下：

channel.configureBlocking(false);

SelectionKey key= channel.register(selector,SelectionKey.OP_READ);

通过调用通道的 register()方法会将它注册到一个选择器上。与 Selector 一起使用时，Channel 必须处于非阻塞模式下，否则将抛出 IllegalBlockingModeException 异常，这意味着不能将 FileChannel 与 Selector 一起使用，因为 FileChannel 不能切换到非阻塞模式，而套接字通道都可以。另外通道一旦被注册，将不能再回到阻塞状态，此时若调用通道的 configureBlocking(true)将抛出 BlockingModeException 异常。

register()方法的第二个参数是“interest 集合”，表示选择器所关心的通道操作，它实际上是一个表示选择器在检查通道就绪状态时需要关心的操作的比特掩码。比如一个选

择器对通道的read和write操作感兴趣,那么选择器在检查该通道时,只会检查通道的read和write操作是否已经处在就绪状态。

具体的操作类型和通道上能被支持的操作类型前面已经讲述过。

如果Selector对通道的多操作类型感兴趣，可以用“位或”操作符来实现：

int interestSet=SelectionKey.OP_READ|SelectionKey.OP_WRITE;

同时 一个 Channel 仅仅可以被注册到一个 Selector 一次, 如果将 Channel 注册到 Selector 多次, 那么其实就是相当于更新 SelectionKey 的 interest set。

通过SelectionKey可以判断Selector是否对Channel的某种事件感兴趣，比如

int interestSet = selectionKey.interestOps();

boolean isInterestedInAccept = (interestSet & SelectionKey.OP_ACCEPT) == SelectionKey.OP_ACCEPT;

通过SelctionKey对象的readyOps()来获取相关通道已经就绪的操作。它是interest集合的子集，并且表示了interest集合中从上次调用select()以后已经就绪的那些操作。JAVA中定义几个方法用来检查这些操作是否就绪，比如selectionKey.isAcceptable();

同时,通过SelectionKey可以取出这个SelectionKey所关联的Selector和Channel。

如果我们要取消关联关系，怎么办？SelectionKey对象的cancel()方法来取消特定的注册关系。

在实际的应用中，我们还可以为SelectionKey绑定附加对象，在需要的时候取出。

SelectionKey key=channel.register(selector,SelectionKey.OP_READ,theObject);

或selectionKey.attach(theObject);

取出这个附加对象，通过：

Object attachedObj = key.attachment();

3、

在实际运行中,我们通过Selector的select（）方法可以选择已经准备就绪的通道（这些通道包含你感兴趣的的事件）。

下面是Selector几个重载的select()方法：

select():阻塞到至少有一个通道在你注册的事件上就绪了。

select(long timeout)：和select()一样，但最长阻塞事件为timeout毫秒。

selectNow():非阻塞，立刻返回。

select()方法返回的int值表示有多少通道已经就绪,是自上次调用select()方法后有多少通道变成就绪状态。

一旦调用select()方法，并且返回值不为0时，则可以通过调用Selector的selectedKeys()方法来访问已选择键集合。

Set selectedKeys=selector.selectedKeys();

这个时候，循环遍历selectedKeys集中的每个键，并检测各个键所对应的通道的就绪事件，再通过SelectionKey关联的Selector和Channel进行实际的业务处理。

注意每次迭代末尾的 keyIterator.remove()调用。Selector 不会自己从已选择键集中移除 SelectionKey 实例。必须在处理完通道时自己移除，否则的话，下次该通道变成就绪时，Selector 会再次将其放入已选择键集中。

**具体与 NIO 编程相关的代码参见模块 nio 下包 cn.tuling.nio.nio。从服务端的代码我们可以看到，我们仅用了一个线程就处理了多个客户端的的连接。**

# 重要概念 SelectionKey

## *什么是 SelectionKey*

SelectionKey 是一个抽象类,表示 selectableChannel 在 Selector 中注册的标识.每个 Channel 向 Selector 注册时,都将会创建一个 SelectionKey。SelectionKey 将 Channel 与 Selector 建立了关系，并维护了 channel 事件。

可以通过 cancel 方法取消键,取消的键不会立即从 selector 中移除,而是添加到 cancelledKeys 中,在下一次 select 操作时移除它.所以在调用某个 key 时,需要使用 isValid 进行校验.

## *SelectionKey 类型和就绪条件*

在向 Selector 对象注册感兴趣的事件时，JAVA NIO 共定义了四种：OP_READ、OP_WRITE、OP_CONNECT、OP_ACCEPT（定义在 SelectionKey 中），分别对应读、写、请求连接、接受连接等网络 Socket 操作。

| 操作类型 | 就绪条件及说明 |
| --- | --- |
| OP_READ | 当操作系统读缓冲区有数据可读时就绪。并非时刻都有数据可读，所以一般需要注册该操作，仅当有就绪时才发起读操作，有的放矢，避免浪费 CPU。 |
| OP_WRITE | 当操作系统写缓冲区有空闲空间时就绪。一般情况下写缓冲区都有空闲空间，小块数据直接写入即可，没必要注册该操作类型，否则该条件不断就绪浪费 CPU；但如果是写密集型的任务，比如文件下载等，缓冲区很可能满，注册该操作类型就很有必要，同时注意写完后取消注册。 |
| OP_CONNECT | 当 SocketChannel.connect()请求连接成功后就绪。该操作只给客户端使用。 |
| OP_ACCEPT | 当接收到一个客户端连接请求时就绪。该操作只给服务器使用。 |

关于 OP_WRITE 的相关代码可以参见包 cn.tuling.nio.nio.writeable

## *服务端和客户端分别感兴趣的类型*

ServerSocketChannel 和 SocketChannel 可以注册自己感兴趣的操作类型，当对应操作类型的就绪条件满足时 OS 会通知 channel，下表描述各种 Channel 允许注册的操作类型，Y 表示允许注册，N 表示不允许注册，其中服务器 SocketChannel 指由服务器 ServerSocketChannel.accept()返回的对象。

| | OP_READ | OP_WRITE | OP_CONNECT | OP_ACCEPT |
|---|---|---|---|---|
| 服务器 ServerSocketChannel | | | | **Y** |
| 服务器 SocketChannel | **Y** | **Y** | | |
| 客户端 SocketChannel | **Y** | **Y** | **Y** | |

服务器启动 ServerSocketChannel，关注 OP_ACCEPT 事件，

客户端启动 SocketChannel，连接服务器，关注 OP_CONNECT 事件

服务器接受连接，启动一个服务器的 SocketChannel，这个 SocketChannel 可以关注 OP_READ、OP_WRITE 事件，一般连接建立后会直接关注 OP_READ 事件

客户端这边的客户端 SocketChannel 发现连接建立后，可以关注 OP_READ、OP_WRITE 事件，一般是需要客户端需要发送数据了才关注 OP_READ 事件

连接建立后客户端与服务器端开始相互发送消息（读写），根据实际情况来关注 OP_READ、OP_WRITE 事件。

# 附录- Buffer 详解

## *重要属性*

### capacity

作为一个内存块，Buffer 有一个固定的大小值，也叫“capacity”.你只能往里写 capacity 个 byte、long，char 等类型。一旦 Buffer 满了，需要将其清空（通过读数据或者清除数据）才能继续写数据往里写数据。

### position

当你写数据到 Buffer 中时，position 表示当前能写的位置。初始的 position 值为 0.当一个 byte、long 等数据写到 Buffer 后， position 会向前移动到下一个可插入数据的 Buffer 单元。position 最大可为 capacity – 1.

当读取数据时，也是从某个特定位置读。当将 Buffer 从写模式切换到读模式，position 会被重置为 0. 当从 Buffer 的 position 处读取数据时，position 向前移动到下一个可读的位置。

### limit

在写模式下，Buffer 的 limit 表示你最多能往 Buffer 里写多少数据。 写模式下，limit 等于 Buffer 的 capacity。

当切换 Buffer 到读模式时， limit 表示你最多能读到多少数据。因此，当切换 Buffer 到读模式时，limit 会被设置成写模式下的 position 值。换句话说，你能读到之前写入的所有数据（limit 被设置成已写数据的数量，这个值在写模式下就是 position）

## *Buffer 的分配*

要想获得一个 Buffer 对象首先要进行分配。 每一个 Buffer 类都有 **allocate** 方法(可以在堆上分配，也可以在直接内存上分配)。

分配 48 字节 capacity 的 ByteBuffer 的例子:ByteBuffer buf = ByteBuffer.allocate(48);

分配一个可存储 1024 个字符的 CharBuffer：CharBuffer buf = CharBuffer.allocate(1024);

**wrap 方法**：把一个 byte 数组或 byte 数组的一部分包装成 ByteBuffer：

ByteBuffer wrap(byte [] array)

ByteBuffer wrap(byte [] array, int offset, int length)

## *直接内存*

HeapByteBuffer 与 DirectByteBuffer，在原理上，前者可以看出分配的 buffer 是在 heap 区域的，其实真正 flush 到远程的时候会先拷贝到直接内存，再做下一步操作；在 NIO 的框架下，很多框架会采用 DirectByteBuffer 来操作，这样分配的内存不再是在 java heap 上，经过性能测试，可以得到非常快速的网络交互，在大量的网络交互下，一般速度会比 HeapByteBuffer 要快速好几倍。

直接内存（Direct Memory）并不是虚拟机运行时数据区的一部分，也不是 Java 虚拟机规范中定义的内存区域，但是这部分内存也被频繁地使用，而且也可能导致 OutOfMemoryError 异常出现。

NIO 可以使用 Native 函数库直接分配堆外内存，然后通过一个存储在 Java 堆里面的 DirectByteBuffer 对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在 Java 堆和 Native 堆中来回复制数据。

### 直接内存（堆外内存）与堆内存比较

直接内存申请空间耗费更高的性能，当频繁申请到一定量时尤为明显

直接内存 IO 读写的性能要优于普通的堆内存，在多次读写操作的情况下差异明显

## *Buffer 的读写*

### 向 Buffer 中写数据

**写数据到 Buffer 有两种方式：**

- **读取 Channel 写到 Buffer。**
- **通过 Buffer 的 put()方法写到 Buffer 里。**

从 Channel 写到 Buffer 的例子

**int** bytesRead = inChannel.read(buf); //read into buffer.

通过 put 方法写 Buffer 的例子：

buf.put(127);

put 方法有很多版本，允许你以不同的方式把数据写入到 Buffer 中。例如， 写到一个指定的位置，或者把一个字节数组写入到 Buffer。在比如：

put(byte b)　相对写，向 position 的位置写入一个 byte，并将 postion+1，为下次读写作准备。

### flip()方法

flip 方法将 Buffer 从写模式切换到读模式。调用 flip()方法会将 position 设回 0，并将 limit 设置成之前 position 的值。

换句话说，position 现在用于标记读的位置，limit 表示之前写进了多少个 byte、char 等 —— 现在能读取多少个 byte、char 等。

### 从 Buffer 中读取数据

**从 Buffer 中读取数据有两种方式：**

1. **从 Buffer 读取数据写入到 Channel。**
2. **使用 get()方法从 Buffer 中读取数据。**

从 Buffer 读取数据到 Channel 的例子：

```
int bytesWritten = inChannel.write(buf);
```

使用 get()方法从 Buffer 中读取数据的例子

```
byte aByte = buf.get();
```

get 方法有很多版本，允许你以不同的方式从 Buffer 中读取数据。例如，从指定 position 读取，或者从 Buffer 中读取数据到字节数组，再比如

get()属于相对读，从 position 位置读取一个 byte，并将 position+1，为下次读写作准备;

### 使用 Buffer 读写数据常见步骤

1. 写入数据到 Buffer
2. 调用 flip()方法
3. 从 Buffer 中读取数据
4. 调用 clear()方法或者 compact()方法，准备下一次的写入

当向 buffer 写入数据时，buffer 会记录下写了多少数据。一旦要读取数据，需要通过 flip()方法将 Buffer 从写模式切换到读模式。在读模式下，可以读取之前写入到 buffer 的所有数据。

一旦读完了所有的数据，就需要清空缓冲区，让它可以再次被写入。有两种方式能清空缓冲区：调用 clear()或 compact()方法。clear()方法会清空整个缓冲区。compact()方法只会清除已经读过的数据。

### 其他常用操作

**绝对读写**

put(int index, byte b)　绝对写，向 byteBuffer 底层的 bytes 中下标为 index 的位置插入 byte b，不改变 position 的值。

get(int index)属于绝对读，读取 byteBuffer 底层的 bytes 中下标为 index 的 byte，不改变 position。

更多 Buffer 实现的细节参考 JavaDoc。

**rewind()方法**

Buffer.rewind()将 position 设回 0，所以你可以重读 Buffer 中的所有数据。limit 保持不变，仍然表示能从 Buffer 中读取多少个元素（byte、char 等）。

**clear()与 compact()方法**

一旦读完 Buffer 中的数据，需要让 Buffer 准备好再次被写入。可以通过 clear()或 compact()方法来完成。

如果调用的是 clear()方法，position 将被设回 0，limit 被设置成 capacity 的值。换句话说，Buffer 被清空了。Buffer 中的数据并未清除，只是这些标记告诉我们可以从哪里开始往 Buffer 里写数据。

如果 Buffer 中有一些未读的数据，调用 clear()方法，数据将“被遗忘”，意味着不再有任何标记会告诉你哪些数据被读过，哪些还没有。

如果 Buffer 中仍有未读的数据，且后续还需要这些数据，但是此时想要先先写些数据，那么使用 compact()方法。

compact()方法将所有未读的数据拷贝到 Buffer 起始处。然后将 position 设到最后一个未读元素正后面。limit 属性依然像 clear()方法一样，设置成 capacity。现在 Buffer 准备好写数据了，但是不会覆盖未读的数据。

**mark()与 reset()方法**

通过调用 Buffer.mark()方法，可以标记 Buffer 中的一个特定 position。之后可以通过调用 Buffer.reset()方法恢复到这个 position。例如：

```
buffer.mark();//call buffer.get() a couple of times, e.g. during parsing.

buffer.reset(); //set position back to mark.
```

**equals()与 compareTo()方法**

可以使用 equals()和 compareTo()方法两个 Buffer。

**equals()**

当满足下列条件时，表示两个 Buffer 相等：

1. 有相同的类型（byte、char、int 等）。
2. Buffer 中剩余的 byte、char 等的个数相等。
3. Buffer 中所有剩余的 byte、char 等都相同。

如你所见，equals 只是比较 Buffer 的一部分，不是每一个在它里面的元素都比较。实际上，它只比较 Buffer 中的剩余元素。

**compareTo()方法**

compareTo()方法比较两个 Buffer 的剩余元素(byte、char 等)， 如果满足下列条件，则认为一个 Buffer“小于”另一个 Buffer：

1. 第一个不相等的元素小于另一个 Buffer 中对应的元素 。
2. 所有元素都相等，但第一个 Buffer 比另一个先耗尽(第一个 Buffer 的元素个数比另一个少)。

## *Buffer 方法总结*

| | |
|---|---|
| limit(), limit(10)等 | 其中读取和设置这 4 个属性的方法的命名和 jQuery 中的 val(),val(10)类似，一个负责 get，一个负责 set |
| reset() | 把 position 设置成 mark 的值，相当于之前做过一个标记，现在要退回到之前标记的地方 |
| clear() | position = 0;limit = capacity;mark = -1；有点初始化的味道，但是并不影响底层 byte 数组的内容 |
| flip() | limit = position;position = 0;mark = -1；翻转，也就是让 flip 之后的 position 到 limit 这块区域变成之前的 0 到 position 这块，翻转就是将一个处于存数据状态的缓 |

| | |
|---|---|
| | 冲区变为一个处于准备取数据的状态 |
| rewind() | 把 position 设为 0，mark 设为-1，不改变 limit 的值 |
| remaining() | return limit - position;返回 limit 和 position 之间相对位置差 |
| hasRemaining() | return position < limit 返回是否还有未读内容 |
| compact() | 把从 position 到 limit 中的内容移到 0 到 limit-position 的区域内，position 和 limit 的取值也分别变成 limit-position、capacity。如果先将 positon 设置到 limit，再 compact，那么相当于 clear() |
| get() | 相对读，从 position 位置读取一个 byte，并将 position+1，为下次读写作准备 |
| get(int index) | 绝对读，读取 byteBuffer 底层的 bytes 中下标为 index 的 byte，不改变 position |
| get(byte[] dst, int offset, int length) | 从 position 位置开始相对读，读 length 个 byte，并写入 dst 下标从 offset 到 offset+length 的区域 |
| put(byte b) | 相对写，向 position 的位置写入一个 byte，并将 postion+1，为下次读写作准备 |
| put(int index, byte b) | 绝对写，向 byteBuffer 底层的 bytes 中下标为 index 的位置插入 byte b，不改变 position |
| put(ByteBuffer src) | 用相对写，把 src 中可读的部分（也就是 position 到 limit）写入此 byteBuffer |
| put(byte[] src, int offset, int length) | 从 src 数组中的 offset 到 offset+length 区域读取数据并使用相对写写入此 byteBuffer |

Buffer 相关的代码参见模块 nio 下包 cn.tuling.nio.buffer

# Reactor 模式类型

## 单线程 Reactor 模式流程：

① 服务器端的 Reactor 是一个线程对象，该线程会启动事件循环，并使用 Selector(选择器)来实现 IO 的多路复用。注册一个 Acceptor 事件处理器到 Reactor 中，Acceptor 事件处理器所关注的事件是 ACCEPT 事件，这样 Reactor 会监听客户端向服务器端发起的连接请求事件(ACCEPT 事件)。

② 客户端向服务器端发起一个连接请求，Reactor 监听到了该 ACCEPT 事件的发生并将该 ACCEPT 事件派发给相应的 Acceptor 处理器来进行处理。Acceptor 处理器通过 accept()方

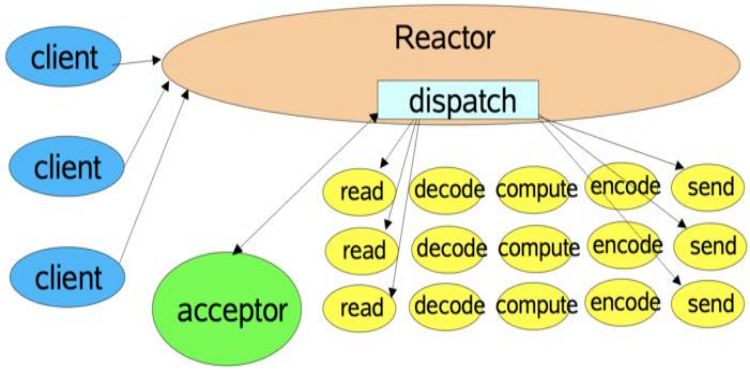
法得到与这个客户端对应的连接(SocketChannel)，然后将该连接所关注的 READ 事件以及对应的 READ 事件处理器注册到 Reactor 中，这样一来 Reactor 就会监听该连接的 READ 事件了。

③ 当 Reactor 监听到有读或者写事件发生时，将相关的事件派发给对应的处理器进行处理。比如，读处理器会通过 SocketChannel 的 read()方法读取数据，此时 read()操作可以直接读取到数据，而不会堵塞与等待可读的数据到来。

④ 每当处理完所有就绪的感兴趣的 I/O 事件后，Reactor 线程会再次执行 select()阻塞等待新的事件就绪并将其分派给对应处理器进行处理。

注意，Reactor 的单线程模式的单线程主要是针对于 I/O 操作而言，也就是所有的 I/O 的 accept()、read()、write()以及 connect()操作都在一个线程上完成的。

但在目前的单线程 Reactor 模式中，不仅 I/O 操作在该 Reactor 线程上，连非 I/O 的业务操作也在该线程上进行处理了，这可能会大大延迟 I/O 请求的响应。所以我们应该将非 I/O 的业务逻辑操作从 Reactor 线程上卸载，以此来加速 Reactor 线程对 I/O 请求的响应。




## 单线程 Reactor，工作者线程池

与单线程 Reactor 模式不同的是，添加了一个工作者线程池，并将非 I/O 操作从 Reactor 线程中移出转交给工作者线程池来执行。这样能够提高 Reactor 线程的 I/O 响应，不至于因为一些耗时的业务逻辑而延迟对后面 I/O 请求的处理。

使用线程池的优势：

① 通过重用现有的线程而不是创建新线程，可以在处理多个请求时分摊在线程创建和销毁过程产生的巨大开销。

② 另一个额外的好处是，当请求到达时，工作线程通常已经存在，因此不会由于等待创建线程而延迟任务的执行，从而提高了响应性。

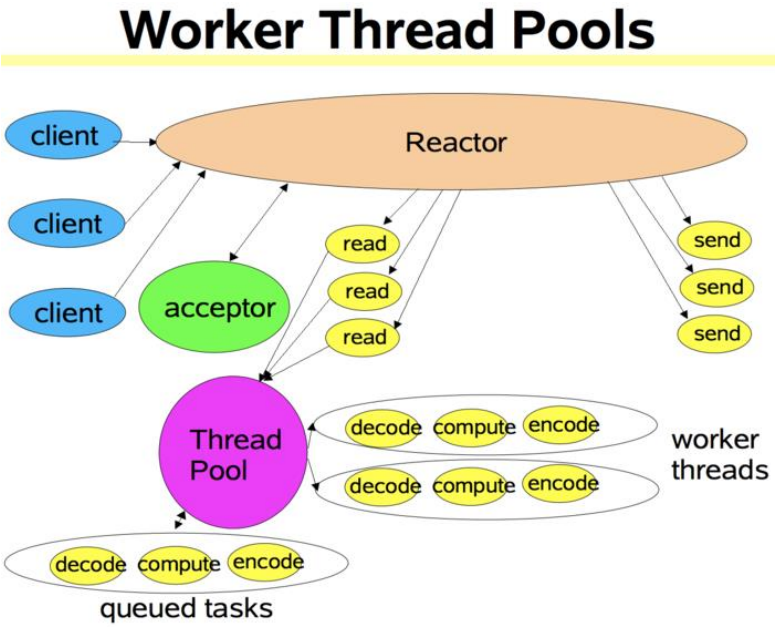
③ 通过适当调整线程池的大小，可以创建足够多的线程以便使处理器保持忙碌状态。同时还可以防止过多线程相互竞争资源而使应用程序耗尽内存或失败。

改进的版本中，所以的 I/O 操作依旧由一个 Reactor 来完成，包括 I/O 的 accept()、read()、write()以及 connect()操作。

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、大并发或大数据量的应用场景却不合适，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，成为系统的性能瓶颈；




## 多线程主从 Reactor 模式

Reactor 线程池中的每一 Reactor 线程都会有自己的 Selector、线程和分发的事件循环逻辑。

mainReactor 可以只有一个，但 subReactor 一般会有多个。mainReactor 线程主要负责接收客户端的连接请求，然后将接收到的 SocketChannel 传递给 subReactor，由 subReactor 来完成和客户端的通信。

流程：

① 注册一个 Acceptor 事件处理器到 mainReactor 中，Acceptor 事件处理器所关注的事件是 ACCEPT 事件，这样 mainReactor 会监听客户端向服务器端发起的连接请求事件(ACCEPT 事件)。启动 mainReactor 的事件循环。

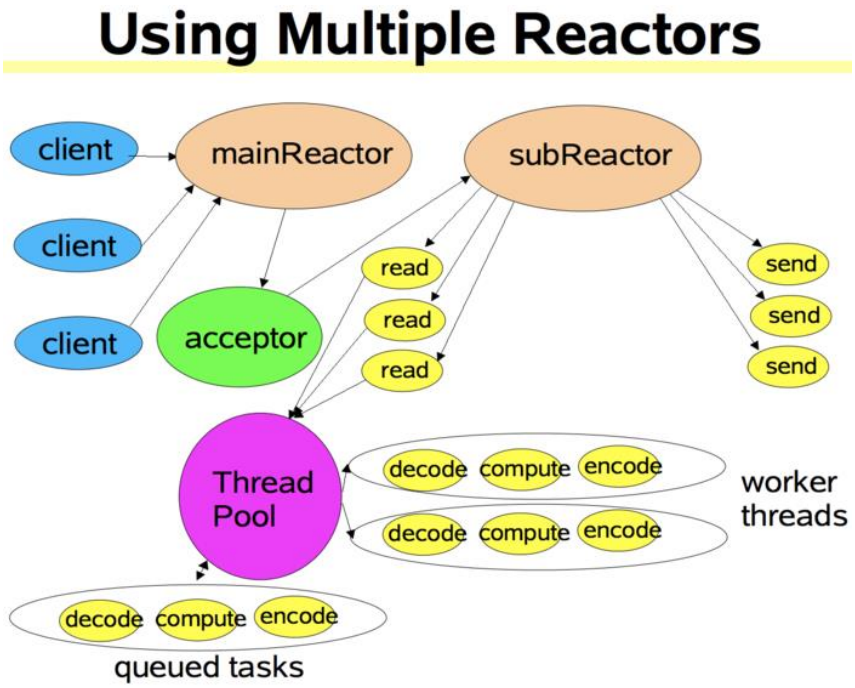
② 客户端向服务器端发起一个连接请求，mainReactor 监听到了该 ACCEPT 事件并将该 ACCEPT 事件派发给 Acceptor 处理器来进行处理。Acceptor 处理器通过 accept()方法得到与这个客户端对应的连接(SocketChannel)，然后将这个 SocketChannel 传递给 subReactor 线程池。

③ subReactor 线程池分配一个 subReactor 线程给这个 SocketChannel，即，将 SocketChannel 关注的 READ 事件以及对应的 READ 事件处理器注册到 subReactor 线程中。当然你也注册 WRITE 事件以及 WRITE 事件处理器到 subReactor 线程中以完成 I/O 写操作。Reactor 线程池中的每一 Reactor 线程都会有自己的 Selector、线程和分发的循环逻辑。

④ 当有 I/O 事件就绪时，相关的 subReactor 就将事件派发给响应的处理器处理。注意，这里 subReactor 线程只负责完成 I/O 的 read()操作，在读取到数据后将业务逻辑的处理放入到线程池中完成，若完成业务逻辑后需要返回数据给客户端，则相关的 I/O 的 write 操作还是会被提交回 subReactor 线程来完成。

注意，所以的 I/O 操作(包括，I/O 的 accept()、read()、write()以及 connect()操作)依旧还是在 Reactor 线程(mainReactor 线程 或 subReactor 线程)中完成的。Thread Pool(线程池)仅用来处理非 I/O 操作的逻辑。

多 Reactor 线程模式将“接受客户端的连接请求”和“与该客户端的通信”分在了两个 Reactor 线程来完成。mainReactor 完成接收客户端连接请求的操作，它不负责与客户端的通信，而是将建立好的连接转交给 subReactor 线程来完成与客户端的通信，这样一来就不会因为 read()数据量太大而导致后面的客户端连接请求得不到即时处理的情况。并且多 Reactor 线程模式在海量的客户端并发请求的情况下，还可以通过实现 subReactor 线程池来将海量的连接分发给多个 subReactor 线程，在多核的操作系统中这能大大提升应用的负载和吞吐量。




## 和观察者模式的区别

**观察者模式：**

也可以称为为 发布-订阅 模式，主要适用于多个对象依赖某一个对象的状态并，当某对象状态发生改变时，要通知其他依赖对象做出更新。是一种一对多的关系。当然，如果依赖的对象只有一个时，也是一种特殊的一对一关系。通常，观察者模式适用于消息事件处理，监听者监听到事件时通知事件处理者对事件进行处理（这一点上面有点像是回调，容易与反应器模式和前摄器模式的回调搞混淆）。

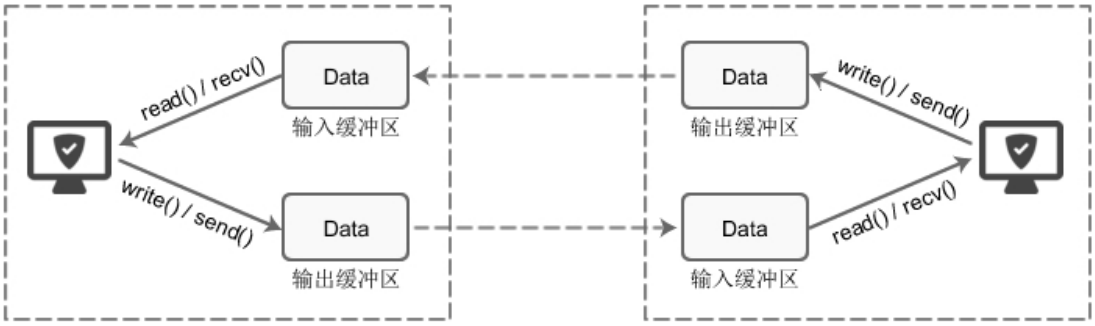
**Reactor 模式：**

reactor 模式，即反应器模式，是一种高效的异步 IO 模式，特征是回调，当 IO 完成时，回调对应的函数进行处理。这种模式并非是真正的异步，而是运用了异步的思想，当 IO 事件触发时，通知应用程序作出 IO 处理。模式本身并不调用系统的异步 IO 函数。

reactor 模式与观察者模式有点像。不过，观察者模式与单个事件源关联，而反应器模式则与多个事件源关联 。当一个主体发生改变时，所有依属体都得到通知。

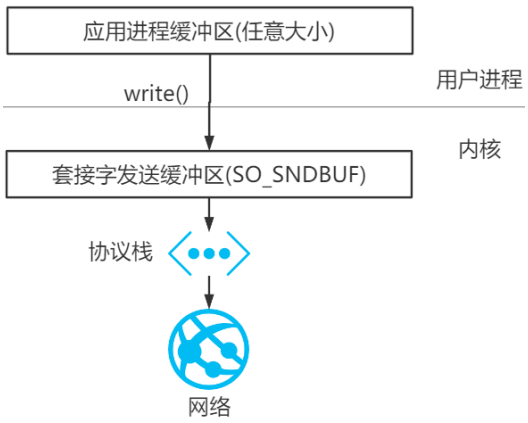
# 3、直接内存深入辨析

在所有的网络通信和应用程序中，每个 TCP 的 Socket 的内核中都有一个发送缓冲区(SO_SNDBUF)和一个接收缓冲区(SO_RECVBUF)，可以使用相关套接字选项来更改该缓冲区大小。




当某个应用进程调用 write 时，内核从该应用进程的缓冲区中复制所有数据到所写套接字的发送缓冲区。如果该套接字的发送缓冲区容不下该应用进程的所有数据(或是应用进程的缓冲区大于套接字的发送缓冲区，或是套接字的发送缓冲区中已有其他数据)，假设该套接字是阻塞的，则该应用进程将被投入睡眠。

内核将不从 write 系统调用返回，直到应用进程缓冲区中的所有数据都复制到套接字发送缓冲区。因此，从写一个 TCP 套接字的 write 调用成功返回仅仅表示我们可以重新使用原来的应用进程缓冲区，并不表明对端的 TCP 或应用进程已接收到数据。




Java 程序自然也要遵守上述的规则。但在 Java 中存在着堆、垃圾回收等特性，所以在实际的 IO 中，在 JVM 内部的存在着这样一种机制：

在 IO 读写上，如果是使用堆内存，JDK 会先创建一个 DirectBuffer，再去执行真正的写操作。这是因为，当我们把一个地址通过 JNI 传递给底层的 C 库的时候，有一个基本的要求，就是这个地址上的内容不能失效。然而，在 GC 管理下的对象是会在 Java 堆中移动的。也就是说，有可能我把一个地址传给底层的 write，但是这段内存却因为 GC 整理内存而失效了。所以必须要把待发送的数据放到一个 GC 管不着的地方。这就是调用 native 方法之前，数据一定要在堆外内存的原因。

可见，站在网络通信的角度 DirectBuffer 并没有节省什么内存拷贝，只是 Java 网络通信里因为 HeapBuffer 必须多做一次拷贝，使用 DirectBuffer 就会少一次内存拷贝。相比没有使用堆内存的 Java 程序，使用直接内存的 Java 程序当然更快一点。

从垃圾回收的角度而言，直接内存不受 GC(新生代的 Minor GC) 影响，只有当执行老年代的 Full GC 时候才会顺便回收直接内存，整理内存的压力也比数据放到 HeapBuffer 要小。

## 堆外内存的优点和缺点

堆外内存相比于堆内内存有几个优势：

1 减少了垃圾回收的工作，因为垃圾回收会暂停其他的工作（可能使用多线程或者时间片的方式，根本感觉不到）

2 加快了复制的速度。因为堆内在 flush 到远程时，会先复制到直接内存（非堆内存），然后在发送；而堆外内存相当于省略掉了这个工作。

而福之祸所依，自然也有不好的一面：

1 堆外内存难以控制，如果内存泄漏，那么很难排查

2 堆外内存相对来说，不适合存储很复杂的对象。一般简单的对象或者扁平化的比较适合。

# 零拷贝

## 什么是零拷贝?

零拷贝(英语: Zero-copy) 技术是指计算机执行操作时，CPU 不需要先将数据从某处内存复制到另一个特定区域。这种技术通常用于通过网络传输文件时节省 CPU 周期和内存带宽。

➢零拷贝技术可以减少数据拷贝和共享总线操作的次数，消除传输数据在存储器之间不必要的中间拷贝次数，从而有效地提高数据传输效率

➢零拷贝技术减少了用户进程地址空间和内核地址空间之间因为上:下文切换而带来的开销

可以看出没有说不需要拷贝，只是说减少冗余[不必要]的拷贝。

下面这些组件、框架中均使用了零拷贝技术：Kafka、Netty、Rocketmq、Nginx、Apache。

## Linux 的 I/O 机制与 DMA

在早期计算机中，用户进程需要读取磁盘数据，需要 CPU 中断和 CPU 参与，因此效率比较低，发起 IO 请求，每次的 IO 中断，都带来 CPU 的上下文切换。因此出现了——DMA。

DMA(Direct Memory Access，直接内存存取) 是所有现代电脑的重要特色，它允许不同速度的硬件装置来沟通，而不需要依赖于 CPU 的大量中断负载。

DMA 控制器，接管了数据读写请求，减少 CPU 的负担。这样一来，CPU 能高效工作了。现代硬盘基本都支持 DMA。

实际因此 IO 读取，涉及两个过程：

1、DMA 等待数据准备好，把磁盘数据读取到操作系统内核缓冲区；

2、用户进程，将内核缓冲区的数据 copy 到用户空间。

## 传统数据传送机制

比如：读取文件，再用 socket 发送出去，实际经过四次 copy。

伪码实现如下：

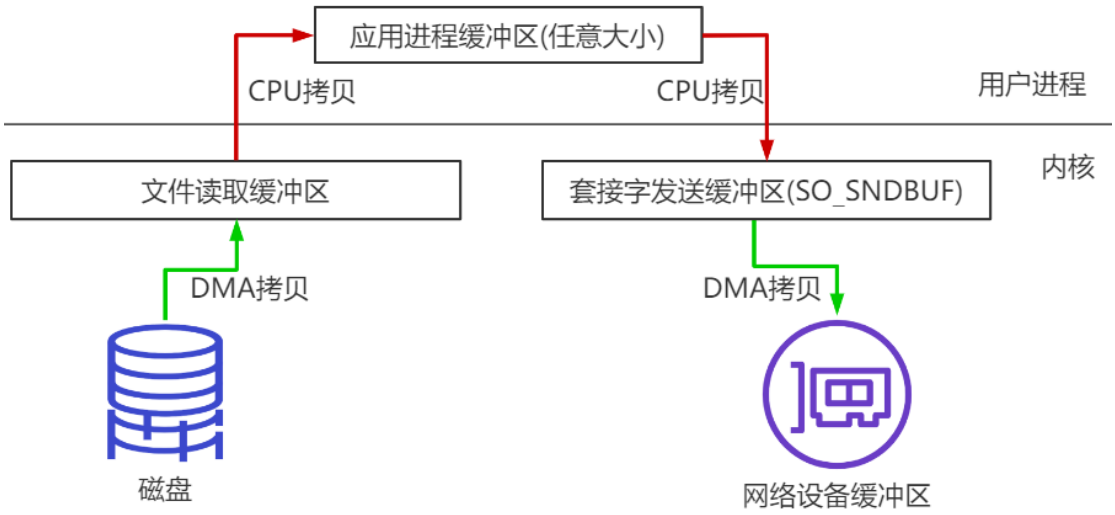
```
buffer = File.read()
Socket.send(buffer)
```

1、第一次：将磁盘文件，读取到操作系统内核缓冲区；

2、第二次：将内核缓冲区的数据，copy 到应用程序的 buffer；

3、第三步：将 application 应用程序 buffer 中的数据，copy 到 socket 网络发送缓冲区(属于操作系统内核的缓冲区)；

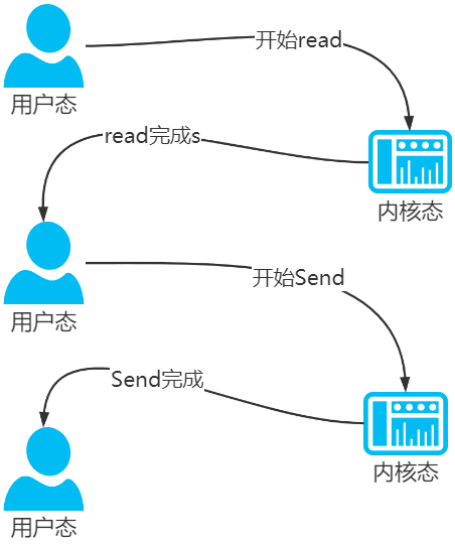
4、第四次：将 socket buffer 的数据，copy 到网卡，由网卡进行网络传输。




分析上述的过程，虽然引入 DMA 来接管 CPU 的中断请求，但四次 copy 是存在“不必要的拷贝”的。实际上并不需要第二个和第三个数据副本。应用程序除了缓存数据并将其传输回套接字缓冲区之外什么都不做。相反，数据可以直接从读缓冲区传输到套接字缓冲区。

显然，第二次和第三次数据 copy 其实在这种场景下没有什么帮助反而带来开销，这也正是零拷贝出现的背景和意义。

同时，read 和 send 都属于系统调用，每次调用都牵涉到两次上下文切换：




总结下，传统的数据传送所消耗的成本：4 次拷贝，4 次上下文切换。

4 次拷贝，其中两次是 DMA copy，两次是 CPU copy。

## Linux 支持的(常见)零拷贝

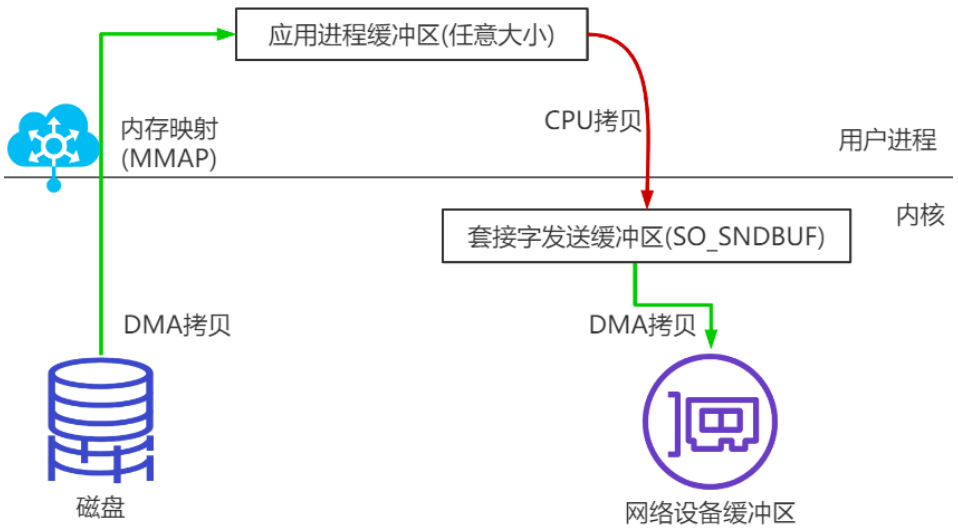
目的：减少 IO 流程中不必要的拷贝，当然零拷贝需要 OS 支持，也就是需要 kernel 暴露 api。

### *mmap 内存映射*

硬盘上文件的位置和应用程序缓冲区(application buffers)进行映射（建立一种一一对应关系），由于 mmap()将文件直接映射到用户空间，所以实际文件读取时根据这个映射关系，直接将文件从硬盘拷贝到用户空间，只进行了一次数据拷贝，不再有文件内容从硬盘拷贝到内核空间的一个缓冲区。

mmap 内存映射将会经历：3 次拷贝: 1 次 cpu copy，2 次 DMA copy；

以及 4 次上下文切换，调用 mmap 函数 2 次，write 函数 2 次。



## *sendfile*

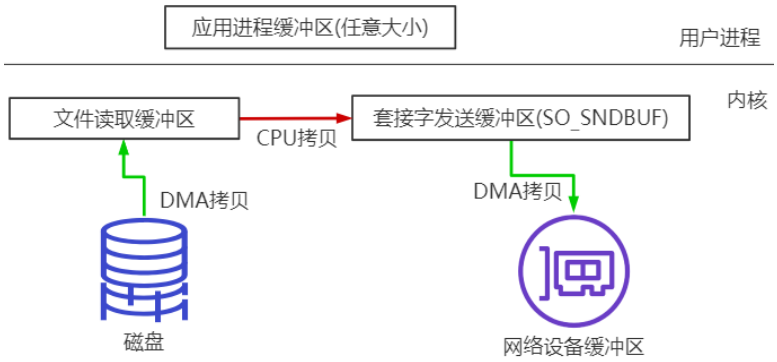
linux 2.1 支持的 sendfile

当调用 sendfile()时，DMA 将磁盘数据复制到 kernel buffer，然后将内核中的 kernel buffer 直接拷贝到 socket buffer；但是数据并未被真正复制到 socket 关联的缓冲区内。取而代之的是，只有记录数据位置和长度的描述符被加入到 socket 缓冲区中。DMA 模块将数据直接从内核缓冲区传递给协议引擎，从而消除了遗留的最后一次复制。但是要注意，这个需要 DMA 硬件设备支持，如果不支持，CPU 就必须介入进行拷贝。

一旦数据全都拷贝到 socket buffer，sendfile()系统调用将会 return、代表数据转化的完成。socket buffer 里的数据就能在网络传输了。

sendfile 会经历：3（2，如果硬件设备支持）次拷贝，1（0，，如果硬件设备支持）次 CPU copy， 2 次 DMA copy；

以及 2 次上下文切换




## *splice*

Linux 从 2.6.17 支持 splice

数据从磁盘读取到 OS 内核缓冲区后，在内核缓冲区直接可将其转成内核空间其他数据 buffer，而不需要拷贝到用户空间。

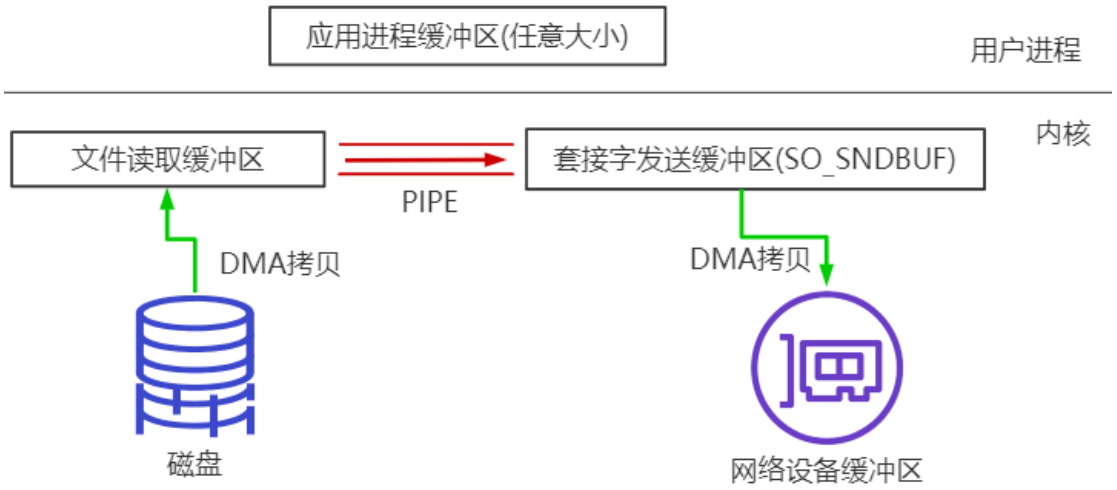
如下图所示，从磁盘读取到内核 buffer 后，在内核空间直接与 socket buffer 建立 pipe 管道。

和 sendfile()不同的是，splice()不需要硬件支持。

注意 splice 和 sendfile 的不同，sendfile 是 DMA 硬件设备不支持的情况下将磁盘数据加载到 kernel buffer 后，需要一次 CPU copy，拷贝到 socket buffer。而 splice 是更进一步，连这个 CPU copy 也不需要了，直接将两个内核空间的 buffer 进行 pipe。

splice 会经历 2 次拷贝: 0 次 cpu copy 2 次 DMA copy；

以及 2 次上下文切换




### *总结 Linux 中零拷贝*

最早的零拷贝定义，来源于

*Linux 2.4 内核新增 sendfile 系统调用，提供了零拷贝。磁盘数据通过 DMA 拷贝到内核态 Buffer 后，直接通过 DMA 拷贝到 NIO Buffer(socket buffer)，无需 CPU 拷贝。这也是零拷贝这一说法的来源。这是真正操作系统 意义上的零拷贝(也就是狭义零拷贝)。*

随着发展，零拷贝的概念得到了延伸，就是目前的减少不必要的数据拷贝都算作零拷贝的范畴。

## Java 生态圈中的零拷贝

Linux 提供的零拷贝技术 Java 并不是全支持，支持 2 种(内存映射 mmap、sendfile)；

### *NIO 提供的内存映射 MappedByteBuffer*

NIO 中的 FileChannel.map()方法其实就是采用了操作系统中的内存映射方式，底层就是调用 Linux mmap()实现的。

将内核缓冲区的内存和用户缓冲区的内存做了一个地址映射。这种方式适合读取大文件，同时也能对文件内容进行更改，但是如果其后要通过 SocketChannel 发送，还是需要 CPU 进行数据的拷贝。

### *NIO 提供的 sendfile*

Java NIO 中提供的 FileChannel 拥有 transferTo 和 transferFrom 两个方法，可直接把 FileChannel 中的数据拷贝到另外一个 Channel，或者直接把另外一个 Channel 中的数据拷贝到 FileChannel。该接口常被用于高效的网络 / 文件的数据传输和大文件拷贝。在操作系统支持的情况下，通过该方法传输数据并不需要将源数据从内核态拷贝到用户态，再从用户态拷贝到目标通道的内核态，同时也避免了两次用户态和内核态间的上下文切换，也即使用了“零拷贝”，所以其性能一般高于 Java IO 中提供的方法。

## Kafka 中的零拷贝

Kafka 两个重要过程都使用了零拷贝技术，且都是操作系统层面的狭义零拷贝，一是 Producer 生产的数据存到 broker，二是 Consumer 从 broker 读取数据。

Producer 生产的数据持久化到 broker，broker 里采用 mmap 文件映射，实现顺序的快速写入；

Customer 从 broker 读取数据，broker 里采用 sendfile，将磁盘文件读到 OS 内核缓冲区后，直接转到 socket buffer 进行网络发送。

## Netty 的零拷贝实现

Netty 的零拷贝主要包含三个方面：

在网络通信上，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。

在缓存操作上，Netty 提供了 CompositeByteBuf 类，它可以将多个 ByteBuf 合并为一个逻辑上的 ByteBuf，避免了各个 ByteBuf 之间的拷贝。

通过 wrap 操作，我们可以将 byte[]数组、ByteBuf、 ByteBuffer 等包装成一个 Netty ByteBuf 对象，进而避免了拷贝操作。

ByteBuf 支持 slice 操作，因此可以将 ByteBuf 分解为多个共享同一个存储区域的 ByteBuf，避免了内存的拷贝。

在文件传输上，Netty 的通过 FileRegion 包装的 FileChannel.tranferTo 实现文件传输，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。

本文档分享地址：

http://note.youdao.com/noteshare?id=8ef33654f746921ad769ad9fe91a4c8f&sub=6737F2A0C19B4DD6893C25A50A823417