# 三种IO介绍

主要有三种IO模型：BIO、NIO、AIO

**BIO、NIO、AIO代码demo：（网上找）**

## BIO介绍

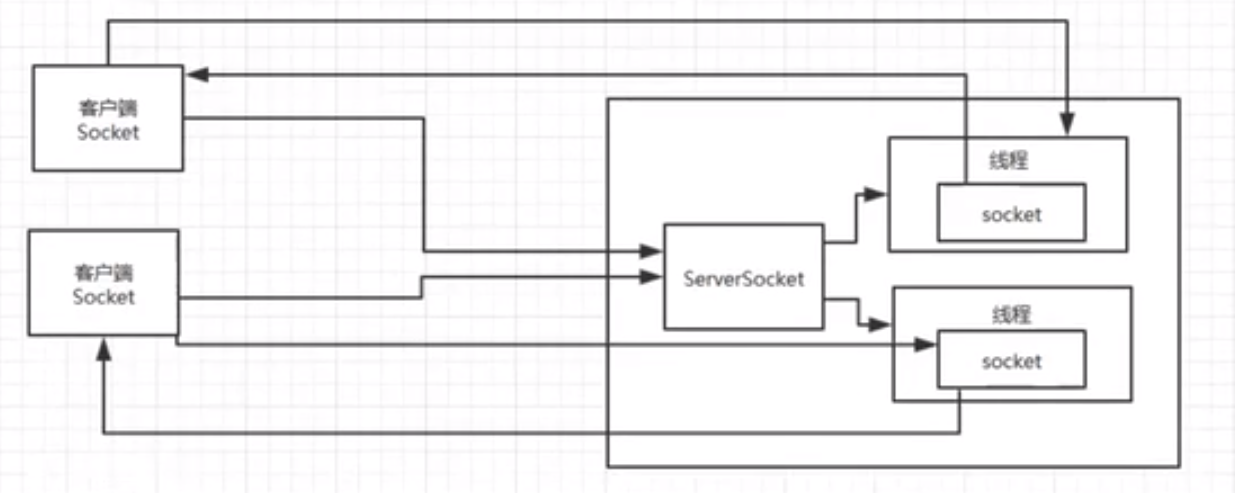
BIO是最传统的网络通信协议，同步阻塞式IO。

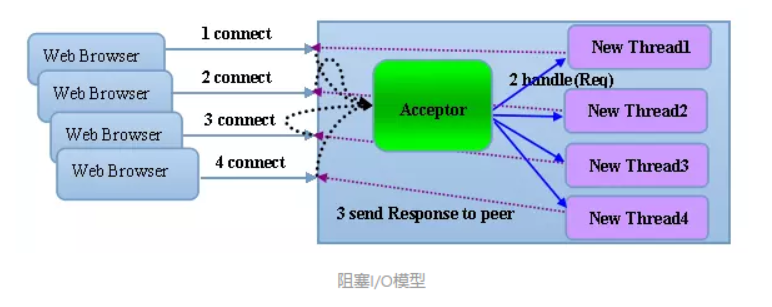
工作原理：首先服务端创建一个ServerSocket，然后客户端用一个Socket去连接该ServerSocket，然后ServerSocket接收到一个Socket的连接请求就创建一个Socket和一个线程去跟客户端Socket进行通信。

客户端和服务端的socket进行同步阻塞式的通信，客户端socket发送一个请求，服务端socket进行处理后返回响应，响应必须是等处理完后才会返回，在此之前啥事也干不了，这就是同步。

这个坑在于，每次一个客户端接入，都是要在服务端创建一个线程来服务这个客户端的，这会导致服务端的线程数量过大，而导致服务端程序的负载过高，最后宕机。

优化：可以搞一个线程池来处理请求，但是高并发请求时，还是可能会导致各种排队和延时，因为没那么多线程来处理。





BIO的两处阻塞：

调用ServerSocket.accept()方法时会一直阻塞，直到客户端连接才会返回。

调用socket.getInputStream().read()方法是阻塞的，它会一直等到数据到来是（或超时）才会返回。

优化：解决read()处阻塞，可以创建新线程去处理read()方法。

优化后缺点：当客户端多时，会创建大量的处理线程，且每个线程都要占用栈控件和一些CPU事件，且太多线程会导致频繁的上下文切换。

## NIO介绍

### IO与NIO区别

JDK1.4中引入NIO，这是一种同步非阻塞IO，基于Reactor模型。

IO是面向流编程，而NIO是面向块（缓冲区）编程，即数据读写必须经过缓冲区。

BIO的服务端接收到客户端的socket连接后，会创建一个socket和一个线程去跟客户端socket通信；而NIO的服务端接收到客户端socket连接后，只创建一个连接并注册到select，等selector轮询channel有读写事件时，再创建线程去进行IO操作。

原因是一个客户端不是时时刻刻都要发送请求的，没必要死耗着一个线程不放。所以NIO的优化思想是一个请求一个线程，只有某个客户端发送了一个请求时，才会启动一个线程来处理。

### NIO核心组件

NIO有三个核心组件：缓冲区(Buffer)、通道(Channel)、选择器(Selector)

缓冲区(Buffer)：将数据写入Buffer中，然后从Buffer中读取数据。

双向通道(Channel)：NIO都是通过Channel进行数据传输和数据读写的。

选择器(Selector)：selector会不断轮询注册的channel，如果某个channel上发送了读写事件，selector就会将这些channel获取出来，我们通过SelectionKey获取有读写事件的channel，就可以进行IO操作了。一个Selector就使用一个线程，去轮询成千上万个channel，这就意味着你的服务端可以接入成千上万个客户端。

#### Buffer(缓存区)

Buffer是一个抽象类，提供相应的方法来操作这个缓冲区。

子类：ByteBuffer, CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer, LongBuffer, ShortBuffer。ByteBuffer和CharBuffer是核心类

ByteBuffer有一个子类 MappedByteBuffer-》能够将文件直接映射到内存中，那么这样我们就可以像访问内存一样访问文件，非常方便

获取ByteBuffer：

static ByteBuffer allocate(int capacity); //分配一个字节缓冲区

static ByteBuffer allocateDirect(int capacity); //分配一个直接字节缓冲区

二者区别：

1、创建普通Buffer成本低，读写的效率不高

2、创建直接Buffer成本高，所以我们一般用在Buffer生存周期较长的时候使用

3、只有ByteBuffer才能够创建直接Buffer，其他的Buffer对象是不能够创建。但可以使用相关api转换成其它Buffer，如asIntBuffer()。

Buffer的4个重要参数：

关系：0 < mark < postion < limit < capacity

capacity：buffer的大小/容量。不可以为负数，一旦创建就不能改变

position：当前读/写的位置。写操作到缓冲区时，position表示当前待写入的位置，position最大为capacity-1；从缓冲区读数据时，position表示从当前位置读取。

mark：标记索引，该索引能够用于下次读取或者写入，它只能够在0-position之间

limit：信息末尾的位置。limit后面的数据既不可读，也不可写。写操作，表示能够从buffer里写多少数据。

Buffer相关方法：

flip():将写模式切换为读模式， 将limit的值改为postion的值，同时将postion归0。特点: 就是为下一次数据的读取做好准备

clear(): 将读模式切换为写模式，将limit改为capacity的值，同时将postion归0。特点: 就是为下一次数据的写入做好准备

put(): 相对读取，向Buffer中存储数据

get(): 相对读取，从Buffer中获取数据

mark(): 设置标记位

reset(): 重置

hasRemaining(): 判断当前位置和limit之间是否还有元素可处理

绝对读取: get(index) 不会影响position的位置

相对读取: put() get() 会影响，每次读取一次，指针后移

#### Channel(双向通道)

3个常用方法：

read():将Channel中的数据读取到Buffer中

write():向Buffer中写入数据

map(mode, position, size):将channel中的数据全部或者部分映射到Buffer中。

// 内存映射读取文件

MappedByteBuffer mappBuffer = inChannel.map(MapMode.READ\_ONLY, 0, srcFile.length());

#### Selector(选择器)

一个组件，可以检测多个NIO Channel，看看读或写事件是否就绪。

多个Channel以事件的方式注册到同一个Selector，从而达到一个线程处理多个请求。

当你调用Selector的select()或者selectNow()方法它只会返回有数据读取的SelectableChannel的实例。

### NIO工作原理

每次客户端建立好连接，服务端就会创建一个对应的Channel并注册到selector上，然后selector轮询这些所有Channel，若channel有请求过来时，它才创建一个工作线程去处理这个请求，处理后得到的结果也写入到该channel中，然后返回给客户端socket。

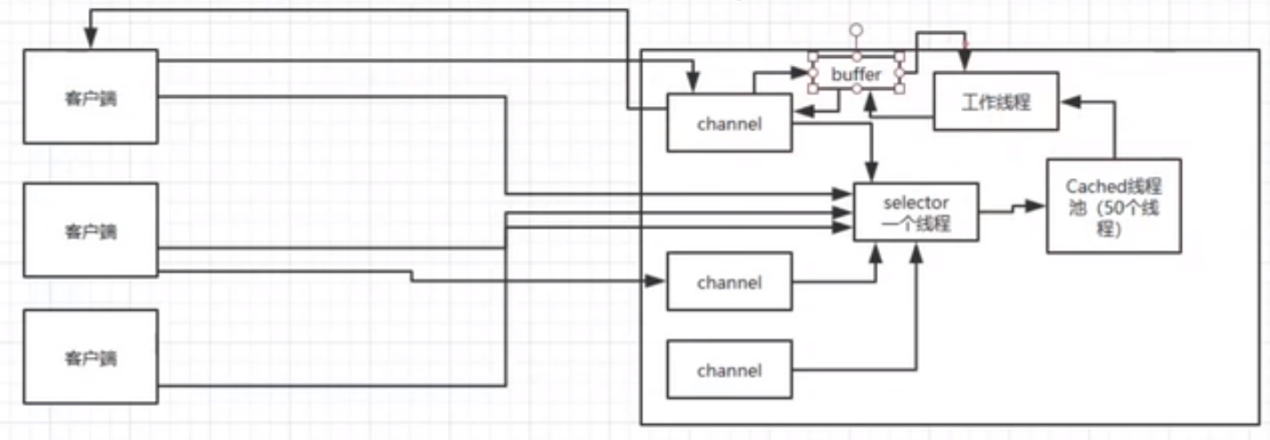
### 为什么NIO是同步非阻塞的呢？

问：为什么NIO是同步非阻塞的呢？

非阻塞理解：因为无论多少客户端都可以接入服务端，客户端接入并不会耗费一个线程只会创建一个连接然后注册到selector上，一个selector线程不断的轮询所有的socket连接，发现有事件了就会通知你，然后你就启动一个线程去处理该请求即可。

但是这个处理过程中，你还是要先读取数据，处理，再返回的，这是个同步的过程。

同步理解：工作线程处理这个请求是同步进行的。向channel进行读写操作都是同步进行的。从channel里读取数据，是工作线程自己去干这个事情，数据没读完，你就卡在那里；然后往channel里写数据，也是你自己去干的事情，数据没写完，你就卡在那里。



## AIO介绍

AIO是基于Proactor模型的，就是异步非阻塞模型。

每个连接发送过来的请求，都会绑定一个buffer，然后通知操作系统去异步完成读，此时你的程序是会去干别的事情，等操作系统完成数据读取之后，就会回调你的接口，给你操作系统异步读完的数据。然后你对这个数据进行处理，接着把结果往回写。写的时候也是给操作系统一个buffer，让操作系统自己获取数据去完成写操作，写完之后再回来通知你。

工作线程的读数据和写数据：

读取数据时，工作线程会给操作系统提供一个空的buffer，然后就可以干别的事情了，也就是说把读数据的事情交给操作系统去干了。操作系统内核，将数据放入buffer后，就回调你的一个接口，告诉你以读取完毕。

写数据也是一样，把放了数据的buffer交给操作系统内核去处理，你就可以去干别的事情了，操作系统完成了数据的写之后，就会回调你，告诉你说，你交给我的数据，我都给你写回到客户端去了。

## BIO、NIO、AIO总结

1.BIO：

同步阻塞的IO模型，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务端就会启动一个线程去处理。

缺点：若连接不做任何事情会造成不必要的线程开销，可使用线程池机制改善。

BIO方式适用于连接数目比较小且固定的架构。

2.NIO：

描述：同步非阻塞的IO模型，服务器实现模式为一个请求一个线程，即客户端发送的连接都会注册到多路复用器上，多路复用器轮询到连接有IO请求时才启动一个线程进行处理。

NIO也是IO多路复用的基础，是解决高并发与大量连接、IO处理问题的有效方式。

NIO适用于连接数多且连接比较短（轻架构）的架构。比如聊天服务器，并发局限于应用中。

3.AIO：

描述：异步非阻塞的IO模型，服务器实现模式为一个有效请求一个线程，即客户端的IO请求都是由OS先完成了，再通知服务器应用区启动线程去处理。

AIO适用于连接数多且连接比较长（重架构）的架构。比如相册服务器，充分调用OS参与并发操作。

# IO模型

## 概念

阻塞、非阻塞（等待数据全部读取成功再返回，还是读取为空马上返回然后下次再读）

同步、异步（用户缓存主动去读取内核缓存，还是内核缓存读取磁盘成功后通知用户缓存）

NIO是同步非阻塞模型，也是IO多路复用基础。

Reactor模式基于同步IO，Proactor模式基于异步IO

IO模型：

IO的各种流是阻塞的，这意味着当一个线程调用read或write方法时，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做其他事情。

所有IO都分为两个阶段：等待就绪和操作。例如：读函数分为等待系统可读和真正的读；写函数分为等待网卡可写和真正的写。

等待就绪的阻塞是不使用CPU的，是在“空等”；真正的读写操作的阻塞是使用CPU的，而且这个过程非常快，属于memory copy，带宽通常在1GB/s级别以上，基本不耗时。

以socket.read()为例子：

BIO，若TCP RecvBuffer里没有数据，函数会一直阻塞，直到收到数据，返回读到的数据。

NIO，若TCP RecvBuffer有数据，就把数据从网卡读到内存，并且返回给用户；反之则直接返回0，永远不会阻塞。

AIO，不但等待就绪是非阻塞的，就连数据从网卡到内存的过程也是异步的。

## 4种IO模型：

4种IO模型：

1）同步阻塞IO(Blocking IO)：即传统的IO模型

2）同步非阻塞IO(Non-blocking IO)：默认创建的socket都是阻塞的，非阻塞IO要求socket被设置为Non-block。

3）IO多路复用(IO Nultiplexing)：即经典的Reactor设计模式，有时也称为异步阻塞IO，Java中的selector和Linux中的epoll都是这种模型

4）异步IO(Asynchronous IO)：即经典的Proactor设计模式，也称为异步非阻塞IO。

## IO多路复用

IO多路复用的场景是我们需要设计一个高性能的网络服务器。该网络服务器能同时跟多个客户端连接，并且能处理这些客户端传上来的请求。

文件描述符，简称fd

如何设计一个IO多路复用的网络服务器？

设计1：我们可以写一个多线程的程序，每个传上来的请求都是一个线程。但是多线程会存在上下文切换。

设计2：如何用单线程来处理大量客户端连接请求？引出select/poll/epoll

## select、poll、epoll介绍

select、poll、epoll都是IO多路复用中的模型，一般用户程序基于这个去开发自己的IO复用模型。如NIO的非阻塞模型，就是采用了IO多路复用的方式，是基于epoll实现的。

1、select通过数据来存储。通过轮询数组，才能感知哪个socket来了数据。缺点是数组长度只能是1024，且需要不断在内核空间和用户空间之间拷贝数组。

2、poll基于链表存储。它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态。本质上跟select没有区别，轮询链表，且需要拷贝数据，但它没有最大连接数的限制。

1和2两种方法适合在请求总数比较少，并且活跃请求数较多的情况。1和2都需要在内核空间和用户空间之间不断拷贝数据。

3、epoll基于红黑树存储。它在内核空间开辟该数据结构，树节点中存放的是一个socket描述符以及用户程序感兴趣的事件类型。同时epoll还会维护一个链表，用于存储已经就绪的socket描述符节点。

select模型：

NIO最主要的就是实现了对异步操作的支持。其中一种通过把一个套接字通道(SocketChannel)注册到一个选择器(Selector)中，不时调用选择器的select方法就能返回满足的选择键(SelectionKey)，键中包含了SOCKET事件信息，这就是select模型。

select和epoll区别？

1）select有最大并发数限制，默认最大文件句柄数1024，可修改。epoll没有最大文件句柄数限制，仅受系统中进程能打开的最大文件句柄数限制。

2）select每次都要线性扫描fd\_set集合。epoll只在集合不为空才轮询。

3）select存在内核空间和用户空间的内存拷贝问题。epoll中减少内存拷贝，map将用户空间的一块地址和内核空间的一块地址同时映射到相同的一块物理内存地址。

问：Epoll导致的selector空轮询？

描述：Java NIO Epoll 会导致 Selector 空轮询，最终导致 CPU 100%。

官方声称在JDK1.6版本的update18修复了该问题，但是直到JDK1.7版本该问题仍旧存在，只不过该BUG发生概率降低了一些而已，它并没有得到根本性解决。

Netty的解决方案：对 Selector 的 select 操作周期进行统计，每完成一次空的 select 操作进行一次计数，若在某个周期内连续发生 N 次空轮询，则判断触发了 Epoll 死循环 Bug。然后，Netty 重建 Selector 来解决。判断是否是其他线程发起的重建请求，若不是则将原 SocketChannel 从旧的 Selector 上取消注册，然后重新注册到新的 Selector 上，最后将原来的 Selector 关闭。

## 为什么epoll性能高于select和poll？

严格的说，这个说法是不准确的，epoll的性能并不总是优于select.

首先需要了解下select和epoll的基本原理。

假设我们现在有100个socket连接，select的做法是，每隔一段时间轮询这100个连接，判断是否有网络事件，如果有，则进行处理。

epoll的做法是，建立一个链表，然后告诉操作系统，那100个连接如果哪个发生了网络事件，就给我放到这个链表里，然后epoll每隔一段时间就检查下这个链表是否有元素，有的话就进行处理。

对比不难发现，epoll相对于select，大大降低了空轮询的次数，提高了轮询的效率，同时，由于select轮询时，需要将所有连接的fd从内核拷贝到用户空间，增加了io开销，epoll在这方面用mmap进行了优化。

不过，不能简单的说谁更好。

比如，这100个连接，如果网络活动十分频繁，那么select每次轮询时，只有很少或者根本没有空轮询，那select的无用功就很少，相反，epoll由于多了一步操作，性能反而更差。

因此，select适用于业务时间较短的短链接，比如一般的http服务。epoll适用于连接时间较长但是网络活动不频繁的场景，比如聊天室。