JAVA IO

Java 的 I/O 大概可以分成以下几类：

- 磁盘操作：File
- 字节操作：InputStream 和 OutputStream
- 字符操作：Reader 和 Writer
- 对象操作：Serializable
- 网络操作：Socket
- 新的输入/输出：NIO

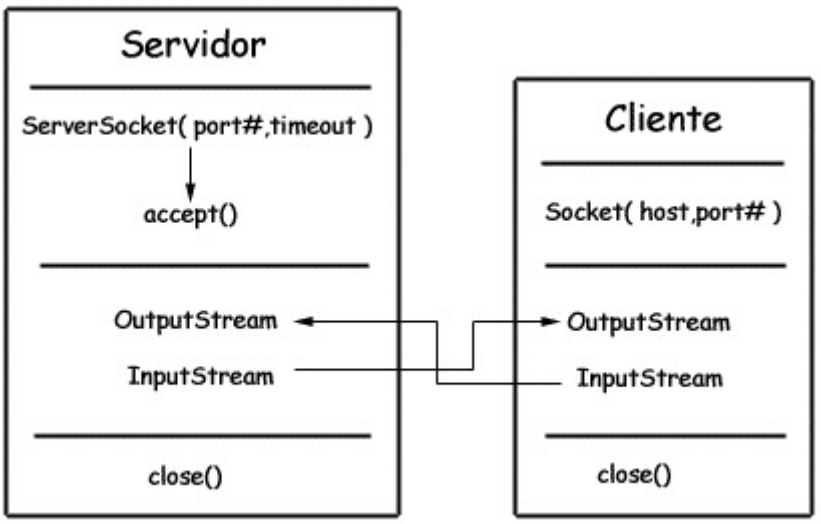
网络操作

Java 中的网络支持：

- InetAddress：用于表示网络上的硬件资源，即 IP 地址；
- URL：统一资源定位符；
- Sockets：使用 TCP 协议实现网络通信；
- Datagram：使用 UDP 协议实现网络通信。

Sockets

- ServerSocket：服务器端类
- Socket：客户端类
- 服务器和客户端通过 InputStream 和 OutputStream 进行输入输出。



流与块：

新的输入/输出 (NIO) 库是在 JDK 1.4 中引入的。NIO 弥补了原来的 I/O 的不足，提供了高速的、面向块的 I/O。

I/O 与 NIO 最重要的区别是数据打包和传输的方式，I/O 以流的方式处理数据，而 NIO 以块的方式处理数据。

- 面向流的 I/O 一次处理一个字节数据：一个输入流产生一个字节数据，一个输出流消费一个字节数据。为流式数据创建过滤器非常容易，链接几个过滤器，以便每个过滤器只负责复杂处理机制的一部分。不利的一面是，面向流的 I/O 通常相当慢。
- 面向块的 I/O 一次处理一个数据块，按块处理数据比按流处理数据要快得多。但是面向块的 I/O 缺少一些面向流的 I/O 所具有的优雅性和简单性。

I/O 包和 NIO 已经很好地集成了，java.io.* 已经以 NIO 为基础重新实现了，所以现在它可以利用 NIO 的一些特性。例如，java.io.* 包中的一些类包含以块的形式读写数据的方法，这使得即使在面向流的系统中，处理速度也会更快。

通道与缓冲区

通道：是对原 I/O 包中的流的模拟，可以通过它读取和写入数据。通道与流的不同之处在于，流只能在一个方向上移动，(一个流必须是 InputStream 或者 OutputStream 的子类)，而通道是双向的，可以用于读、写或者同时用于读写。

通道包括以下类型：

- FileChannel：从文件中读写数据；
- DatagramChannel：通过 UDP 读写网络中数据；
- SocketChannel：通过 TCP 读写网络中数据；
- ServerSocketChannel：可以监听新进来的 TCP 连接，对每一个新进来的连接都会创建一个 SocketChannel。

缓冲区：发送给一个通道的所有数据都必须首先放到缓冲区中，同样地从通道中读取的任何数据都要先读到缓冲区中。也就是说，不会直接对通道进行读写数据，而是要先经过缓冲区。缓冲区实质上是一个数组，但它不仅仅是一个数组，缓冲区提供了对数据的结构化访问，而且还可以跟踪系统的读/写进程。
缓冲区包括以下类型：

- ByteBuffer
- CharBuffer
- ShortBuffer
- IntBuffer
- LongBuffer
- FloatBuffer
- DoubleBuffer

缓冲区状态变量：

- capacity：最大容量；
- position：当前已经读写的字节数；
- limit：还可以读写的字节数。

文件 NIO 实例

```
public static void fastCopy(String src, String dist) throws IOException
{
FileInputStream fin = new FileInputStream(src); /* 获取源文件的输入字节流 */
FileChannel fcin = fin.getChannel(); /* 获取输入字节流的文件通道 */
```

```
FileOutputStream fout = new FileOutputStream(dist); /* 获取目标文件的输出字节流 */
FileChannel fcout = fout.getChannel(); /* 获取输出字节流的通道 */
ByteBuffer buffer = ByteBuffer.allocateDirect(1024); /* 为缓冲区分配 1024 个字节 */
while (true) {
int r = fcin.read(buffer); /* 从输入通道中读取数据到缓冲区中 */
if (r == -1) { /* read() 返回 -1 表示 EOF */
break;
}
buffer.flip(); /* 切换读写 */
fcout.write(buffer); /* 把缓冲区的内容写入输出文件中 */
buffer.clear(); /* 清空缓冲区 */
}
}
```

选择器

NIO 常常被叫做非阻塞 IO，主要是因为 NIO 在网络通信中的非阻塞特性被广泛使用。

NIO 实现了 IO 多路复用中的 Reactor 模型，一个线程 Thread 使用一个选择器 Selector 通过轮询的方式去监听多个通道 Channel 上的事件，从而让一个线程就可以处理多个事件。通过配置监听的通道 Channel 为非阻塞，那么当 Channel 上的 IO 事件还未到达时，就不会进入阻塞状态一直等待，而是继续轮询其它 Channel，找到 IO 事件已经到达的 Channel 执行。因为创建和切换线程的开销很大，因此使用一个线程来处理多个事件而不是一个线程处理一个事件具有更好的性能。应该注意的是，只有套接字 Channel 才能配置为非阻塞，而 FileChannel 不能为 FileChannel 配置非阻塞也没有意义。

IO

用户空间和内核空间：现在操作系统都是采用虚拟存储器，那么对32位操作系统而言，它的寻址空间（虚拟存储空间）为4G（2的32次方）。操作系统的核心是内核，独立于普通的应用程序，可以访问受保护的内存空间，也有访问底

层硬件设备的所有权限。为了保证用户进程不能直接操作内核（kernel），保证内核的安全，操心系统将虚拟空间划分为两部分，一部分为内核空间，一部分为用户空间。针对linux操作系统而言，将最高的1G字节（从虚拟地址0xC0000000到0xFFFFFFFF），供内核使用，称为内核空间，而将较低的3G字节（从虚拟地址0x00000000到0xBFFFFFFF），供各个进程使用，称为用户空间。

进程的阻塞：正在执行的进程，由于期待的某些事件未发生，如请求系统资源失败、等待某种操作的完成、新数据尚未到达或无新工作做等，则由系统自动执行阻塞原语(Block)，使自己由运行状态变为阻塞状态。可见，进程的阻塞是进程自身的一种主动行为，也因此只有处于运行态的进程（获得CPU），才可能将其转为阻塞状态。当进程进入阻塞状态，是不占用CPU资源的。

文件描述符：是一个用于表述指向文件的引用的抽象化概念。文件描述符在形式上是一个非负整数。实际上，它是一个索引值，指向内核为每一个进程所维护的该进程打开文件的记录表。当程序打开一个现有文件或者创建一个新文件时，内核向进程返回一个文件描述符。但是这一概念往往只适用于unix、linux这样的操作系统。

缓存IO: 缓存 I/O 又被称作标准 I/O，大多数文件系统的默认 I/O 操作都是缓存 I/O。在 Linux 的缓存 I/O 机制中，操作系统会将 I/O 的数据缓存在文件系统的页缓存（ page cache ）中，也就是说，数据会先被拷贝到操作系统内核的缓冲区中，然后才会从操作系统内核的缓冲区拷贝到应用程序的地址空间。缺点就是数据在传输过程中需要在应用程序地址空间和内核进行多次数据拷贝操作，这些数据拷贝操作所带来的 CPU 以及内存开销是非常大的。

IO模式

当一个read操作发生时，它会经历两个阶段：

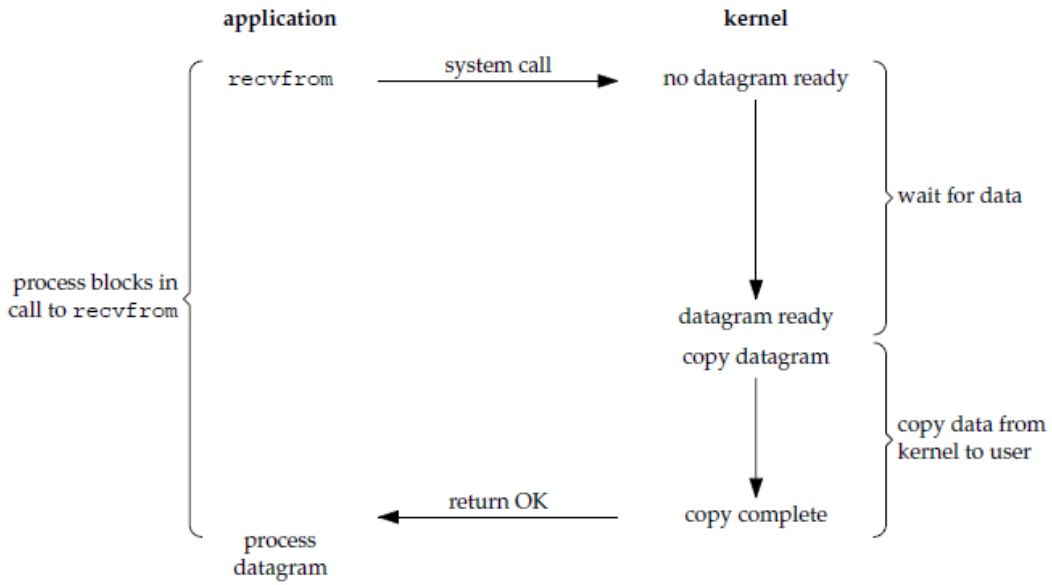
- 等待数据准备：
- 将数据从内核拷贝到进程中

正式因为这两个阶段，linux系统产生了下面五种网络模式的方案。

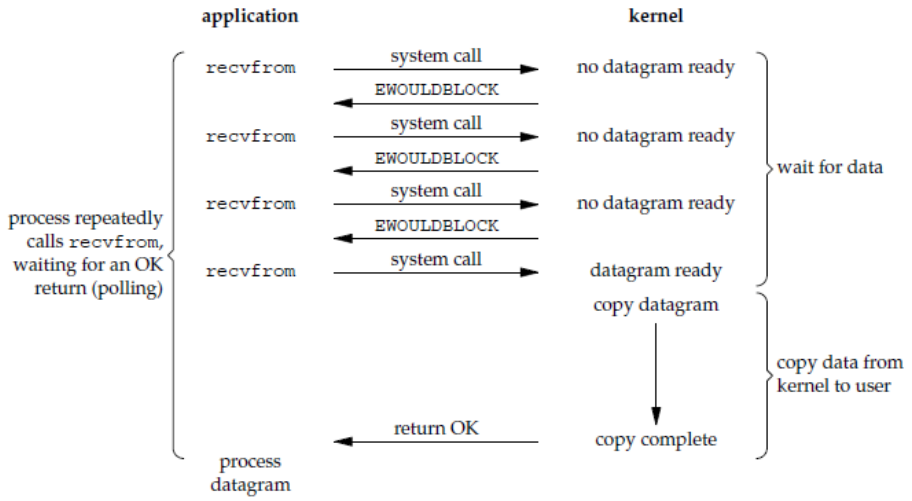
- - 阻塞 I/O（blocking IO）

- - 非阻塞 I/O（nonblocking IO）
- - I/O 多路复用（ IO multiplexing）
- - 信号驱动 I/O（ signal driven IO）
- - 异步 I/O（asynchronous IO）

阻塞IO:在linux中，默认情况下所有的socket都是blocking，一个典型的读操作流程大概是这样：当用户进程调用了recvfrom这个系统调用，kernel就开始了IO的第一个阶段：准备数据（对于网络IO来说，很多时候数据在一开始还没有到达。比如还没有收到一个完整的UDP包。这个时候kernel就要等待足够的数据到来）这个过程需要等待，也就是说数据被拷贝到操作系统内核的缓冲区是需要一个过程的。而在用户进程这边，整个进程会被阻塞（当然，是进程自己选择的阻塞）。当kernel一直等到数据准备好了，它就会将数据从kernel中拷贝到用户内存，然后kernel返回结果，用户进程才重新运行起来。



非阻塞IO: 当用户进程发出read操作时，如果kernel中的数据还没有准备好，那么它并不会block用户进程，而是立刻返回一个error。从用户进程角度讲 ，它发起一个read操作后，并不需要等待，而是马上就得到了一个结果。用户进程判断结果是一个error时，它就知道数据还没有准备好，于是它可以再次发送read操作。一旦kernel中的数据准备好了，并且又再次收到了用户进程的system call，那么它马上就将数据拷贝到了用户内存，然后返回。所以，非阻塞IO的特点就是用户进程需要不断主动询问kernel数据好了没有。

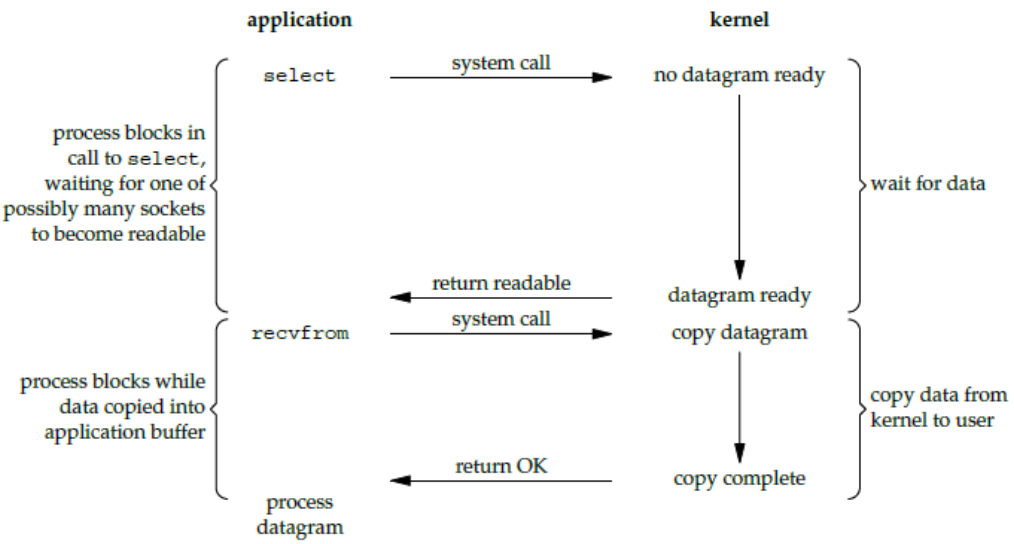


IO多路复用：IO multiplexing，也称这种IO方式为事件驱动IO(event driven IO)。非阻塞IO（non-blocking IO）模式需要用户自己去轮询查看是否数据准备好，如果准备好则阻塞调用kernel进行copy。多路复用IO就是解决这种轮询问题，linux内部提供了select/poll/epoll来完成IO复用。

使用 select 或者 poll 等待数据，并且可以等待多个套接字中的任何一个变为可读，这一过程会被阻塞，当某一个套接字可读时返回。之后再使用 recvfrom 把数据从内核复制到进程中。如果一个 Web 服务器没有 I/O 复用，那么每一个 Socket 连接都需要创建一个线程去处理。如果同时有几万个连接，那么就需要创建相同数量的线程。并且相比于多进程和多线程技术，I/O 复用不需要进程线程创建和切换的开销，系统开销更小。

+

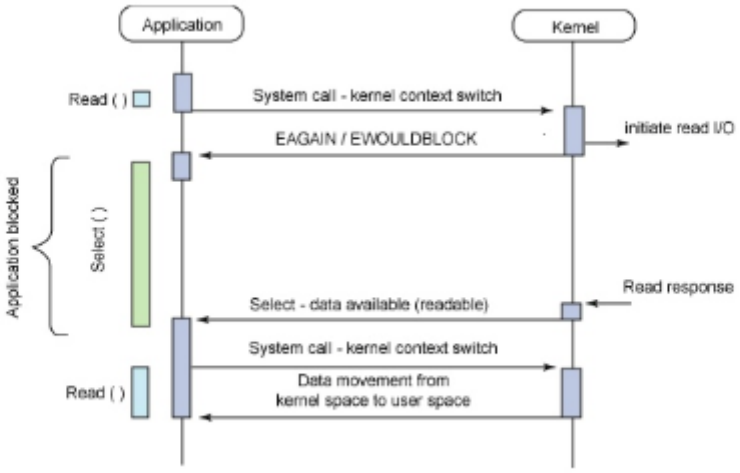
select/poll/epoll的好处就在于单个process就可以同时处理多个网络连接的IO。它的基本原理就是select/poll/epoll这个function会不断的轮询所负责的所有socket，当某个socket有数据到达了，就通知用户进程。它的流程如图：



在IO multiplexing Model中，实际中，对于每一个socket，一般都设置成为non-blocking，但是，如上图所示，整个用户的process其实是一直被block的。只不过process是被select这个函数block，而不是被socket IO给block。

select

select的调用过程如下所示：



select负责管理多个FD文件描述符，kernel就会轮询检查所有select负责的fd，看是否有一个FD的数据已准备好。select会返回kernel数据准备就绪的FD， FD调用read操作让kernel完成数据的拷贝。 select解决了非阻塞状态下用户进程需要自己轮询的问题，同时可以用一个线程管理多个用户进程的读写操作。

所以，I/O 多路复用的特点是通过一种机制一个进程能同时等待多个文件描述符，而这些文件描述符（套接字描述符）其中的任意一个进入读就绪状态，select()函数就可以返回。

select目前几乎在所有的平台上支持，其良好跨平台支持也是它的一个优点。

select的缺点：

- 单个进程能够监视的文件描述符的数量存在最大限制，通常是1024，当然可以更改数量。
- 对socket进行扫描时是线性扫描，即采用轮询的方法，效率较低。
- 内核/用户空间内存拷贝问题。每次调用select，都需要把fd集合从用户态拷贝到内核态，这个开销在fd(客户端套接字)很多时会很大。

poll

poll本质上和select没有区别，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，如果设备就绪则在设备等待队列中加入一项并继续遍历，如果遍历完所有fd后没有发现就绪设备，则挂起当前进程，直到设备就绪或者主动超时，被唤醒后它又要再次遍历fd。这个过程经历了多次无谓的遍历。

它没有最大连接数的限制，原因是它是基于链表来存储的，但是同样有一个缺点：

+

- 大量的fd的数组被整体复制于用户态和内核地址空间之间，而不管这样的复制是不是有意义。
- poll还有一个特点是“水平触发”，如果报告了fd后，没有被处理，那么下次poll时会再次报告该fd。

Select与Poll的比较：

功能：select 和 poll 的功能基本相同，不过在一些实现细节上有所不同。

- select 会修改描述符，而 poll 不会；
- select 的描述符类型使用数组实现，FD_SETSIZE 大小默认为 1024，因此默认只能监听 1024 个描述符。如果要监听更多描述符的话，需要修

改 FD_SETSIZE 之后重新编译；而 poll 的描述符类型使用链表实现，没有描述符的数量的限制；

- poll 提供了更多的事件类型，并且对描述符的重复利用上比 select 高。
- 如果一个线程对某个描述符调用了 select 或者 poll，另一个线程关闭了该描述符，会导致调用结果不确定。

速度：select 和 poll 速度都比较慢。

- select 和 poll 每次调用都需要将全部描述符从应用进程缓冲区复制到内核缓冲区。
- select 和 poll 的返回结果中没有声明哪些描述符已经准备好，所以如果返回值大于 0 时，应用进程都需要使用轮询的方式来找到 I/O 完成的描述符。

可移植性：几乎所有的系统都支持 select，但是只有比较新的系统支持 poll。

epoll

在linux 没有实现epoll事件驱动机制之前，我们一般选择用select或者poll等IO多路复用的方法来实现并发服务程序，epoll是在2.6内核中提出的，是之前的select和poll的增强版本。相对于select和poll来说，epoll更加灵活，没有描述符限制。epoll使用一个文件描述符管理多个描述符，将用户关系的文件描述符的事件存放到内核的一个事件表中，这样在用户空间和内核空间的copy只需一次。

基本原理：epoll支持水平触发和边缘触发，最大的特点在于边缘触发，它只告诉进程哪些fd刚刚变为就绪态，并且只会通知一次。还有一个特点是，epoll使用“事件”的就绪通知方式，通过epoll_ctl注册fd，一旦该fd就绪，内核就会采用类似callback的回调机制来激活该fd，epoll_wait便可以收到通知。

epoll的优点：

- 没有最大并发连接的限制，能打开的FD的上限远大于1024（1G的内存上能监听约10万个端口）。

- 效率提升，不是轮询的方式，不会随着FD数目的增加效率下降。只有活跃可用的FD才会调用callback函数；即Epoll最大的优点就在于它只管你“活跃”的连接，而跟连接总数无关，因此在实际的网络环境中，Epoll的效率就会远远高于select和poll。
- 内存拷贝，利用mmap()文件映射内存加速与内核空间的消息传递；即epoll使用mmap减少复制开销。

三者的应用场景：

很容易产生一种错觉认为只要用 epoll 就可以了，select 和 poll 都已经过时了，其实它们都有各自的使用场景。

- select 应用场景：select 的 timeout 参数精度为 1ns，而 poll 和 epoll 为 1ms，因此 select 更加适用于实时要求更高的场景，比如核反应堆的控制。select 可移植性更好，几乎被所有主流平台所支持。
- poll 应用场景：poll 没有最大描述符数量的限制，如果平台支持并且对实时性要求不高，应该使用 poll 而不是 select。需要同时监控小于 1000 个描述符，就没有必要使用 epoll，因为这个应用场景下并不能体现 epoll 的优势。需要监控的描述符状态变化多，而且都是非常短暂的，也没有必要使用 epoll。因为 epoll 中的所有描述符都存储在内核中，造成每次需要对描述符的状态改变都需要通过 epoll_ctl() 进行系统调用，频繁系统调用降低效率。并且epoll 的描述符存储在内核，不容易调试。
- epoll 应用场景：只需要运行在 Linux 平台上，并且有非常大量的描述符需要同时轮询，而且这些连接最好是长连接。

异步IO

用户进程发起read操作之后，立刻就可以开始去做其它的事。而另一方面，从kernel的角度，当它受到一个asynchronous read之后，首先它会立刻返回，所以不会对用户进程产生任何block。然后，kernel会等待数据准备完成，然后将数据拷贝到用户内存，当这一切都完成之后，kernel会给用户进程发送一个signal，告诉它read操作完成了。

+

小结：

阻塞和非阻塞：阻塞和非阻塞是一种调用机制，用来描述进程处理调用的方式。在IO中两者的区别主要体现在I/O未准备好时，用户线程是否可以做其他事情。比如网络读操作，根据是否需要等待kernel数据准备好。阻塞是等待某个事件的就绪/发生，当前线程会被挂起，一直处于等待消息通知，不能执行其他业务。阻塞通信意味着通信方法在尝试访问套接字或者读写数据时阻塞了对套接字的访问。以网络读操作为例，用户线程在socket中调用recv函数时，如果缓冲区中没有数据，则需要一直阻塞等待服务端发来的数据，这时候线程会挂起等待。非阻塞和阻塞的概念相对应，指在不能立刻得到结果之前，该函数不会阻塞当前线程，而会立刻返回。非阻塞IO是用户线程不会一直阻塞待待IO绪，通过不断轮询的方式来查看就绪状态。。

同步和异步：同步和异步是一种通信机制，涉及到调用方和被调用方，关注的是IO操作结果的获知方式，主要区别在于IO结果未返回时用户线程是否可以做其他事情：

+

- 同步是调用方需要保持等待直到IO操作完成，进而通过返回获得结果；
- 异步则调用方在IO操作的执行过程中不需要保持等待，而是在操作完成后被动的接受（通过消息或回调）被调用方推送的结果。

同步和异步的区别也在于在进行整个IO操作的时候会用户进程是否会阻塞等待结果，linux中IO模型中blocking IO，non-blocking IO，IO multiplexing都属于synchronous IO。

对于non-blocking IO并没有被block啊的问题，这里有个非常“狡猾”的地方，定义中所指的”IO operation”是指真实的IO操作，就是例子中的recvfrom这个system call。non-blocking IO在执行recvfrom这个system call的时候，如果kernel的数据没有准备好，这时候不会block进程。但是，当kernel中数据准备好的时候，recvfrom会将数据从kernel拷贝到用户内存中，这个时候进程是被block了，在这段时间内，进程是被block的。

Java IO 分类

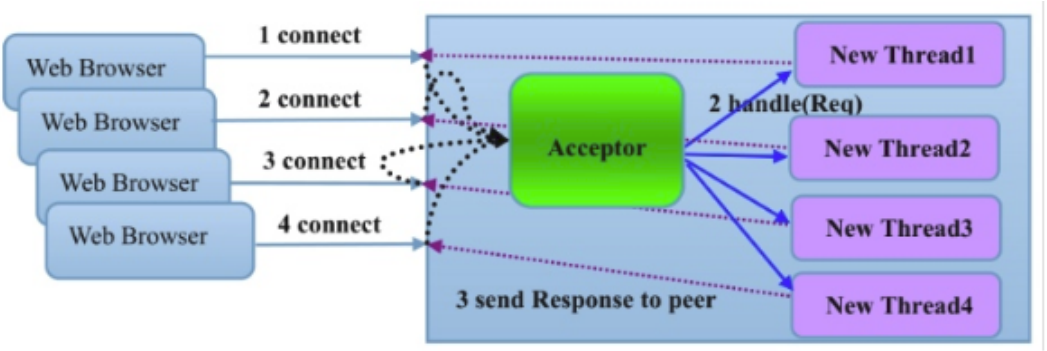
- **Java BIO：** 同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。

+

- **Java NIO：** 同步非阻塞，服务器实现模式为一个请求一个线程，即当一个连接创建后，不需要对应一个线程，这个连接会被注册到多路复用器上面，所以所有的连接只需要一个线程就可以搞定，当这个线程中的多路复用器进行轮询的时候，发现连接上有请求的话，才开启一个线程进行处理，也就是一个请求一个线程模式。BIO与NIO一个比较重要的不同，是我们使用BIO的时候往往会引入多线程，每个连接一个单独的线程；而NIO则是使用单线程或者只使用少量的多线程，每个连接共用一个线程。
- **Java AIO(NIO.2)：** 异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理。

JAVA BIO

在JDK 1.4推出Java NIO之前，基于Java的所有Socket通信都采用了同步阻塞模式（BIO），这种一请求一应答的通信模型简化了上层的应用开发，但是在性能和可靠性方面却存在着巨大的瓶颈。当并发访问量增大、响应时间延迟增大之后，采用Java BIO开发的服务端软件只有通过硬件的不断扩容来满足高并发和低时延，它极大地增加了企业的成本，并且随着集群规模的不断膨胀，系统的可维护性也面临巨大的挑战，只能通过采购性能更高的硬件服务器来解决问题，这会导致恶性循环,传统采用BIO的Java Web服务器如下所示（典型的如Tomcat的BIO模式):



采用该线程模型的服务器调度特点如下：

- 服务端监听线程Acceptor负责客户端连接的接入，每当有新的客户端接入，就会创建一个新的I/O线程负责处理Socket
- 客户端请求消息的读取和应答的发送，都有I/O线程负责
- 除了I/O读写操作，默认情况下业务的逻辑处理，例如DB操作等，也都在I/O线程处理
- I/O操作采用同步阻塞操作，读写没有完成，I/O线程会同步阻塞

BIO线程模型主要存在如下三个问题：

- 性能问题：一连接一线程模型导致服务端的并发接入数和系统吞吐量受到极大限制
- 可靠性问题：由于I/O操作采用同步阻塞模式，当网络拥塞或者通信对端处理缓慢会导致I/O线程被挂住，阻塞时间无法预测
- 可维护性问题：I/O线程数无法有效控制、资源无法有效共享（多线程并发问题），系统可维护性差

BIO、NIO、AIO适用场景分析

- BIO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择，但程序直观简单易理解。
- NIO方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4开始支持。

- AIO方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂，JDK7开始支持。

Java NIO和IO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。
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，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
选择器（Selectors）：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，然后使用一个单独的线程来“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。