相关资料 IO基本概念 Linux环境 同步异步阻塞非阻塞 同步与异步 阻塞与非阻塞 IO模型Reference Link 阻塞IO模型 非阻塞IO模型 IO复用模型 信号驱动异步IO模型 异步IO模型 总结 AIOBIONIO Java对BIONIOAIO的支持 AIOReference Link1ReferenceLink2 NIOReference Link epollselectpollReference Link LTETepoll select的几大缺点 poll实现 epollreference Link 总结 IOCP ReferenceLinkConcreteRealization 相关资料 IO基本概念 Linux环境 Linux的内核将所有外部设备都可以看做一个文件来操作。那么我们对与外部设备的操作都可以看做对文件进行操作。我们对一个文件的读写，都通过调用内核提供的系统调用；内核给我们返回一个file descriptor（fd,文件描述符）。对一个socket的读写也会有相应的描述符，称为socketfd(socket描述符）。描述符就是一个数字(可以理解为一个索引)，指向内核中一个结构体（文件路径，数据区，等一些属性）。应用程序对文件的读写就通过对描述符的读写完成。 一个基本的IO，它会涉及到两个系统对象，一个是调用这个IO的进程对象，另一个就是系统内核(kernel)。当一个read操作发生时，它会经历两个阶段：

通过read系统调用想内核发起读请求。 内核向硬件发送读指令，并等待读就绪。 内核把将要读取的数据复制到描述符所指向的内核缓存区中。 将数据从内核缓存区拷贝到用户进程空间中。 同步，异步，阻塞，非阻塞 同步与异步 同步和异步关注的是消息通信机制 (synchronous communication/ asynchronous communication) 所谓同步，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。 换句话说，就是由调用者主动等待这个调用的结果。

而异步则是相反，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。

典型的异步编程模型比如Node.js

举个通俗的例子： 你打电话问书店老板有没有《分布式系统》这本书，如果是同步通信机制，书店老板会说，你稍等，”我查一下”，然后开始查啊查，等查好了（可能是5秒，也可能是一天）告诉你结果（返回结果）。 而异步通信机制，书店老板直接告诉你我查一下啊，查好了打电话给你，然后直接挂电话了（不返回结果）。然后查好了，他会主动打电话给你。在这里老板通过“回电”这种方式来回调。

实质：访问数据的方式，同步需要当前线程读写数据，在读写数据的过程中还是会阻塞；异步只需要I/O操作完成的通知，当前进程并不主动读写数据，由操作系统内核完成数据的读写。

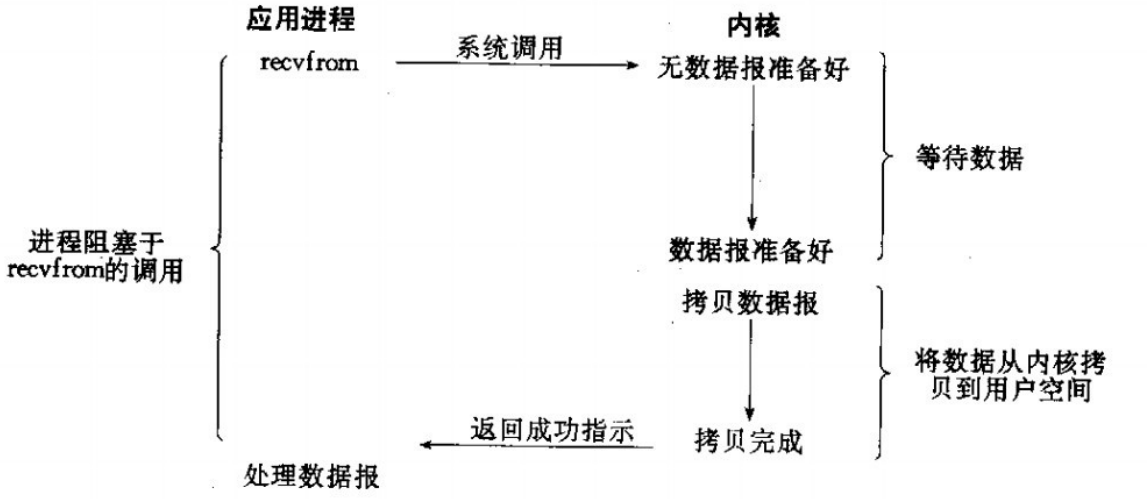
阻塞与非阻塞 阻塞和非阻塞关注的是程序在等待调用结果（消息，返回值）时的状态.

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。

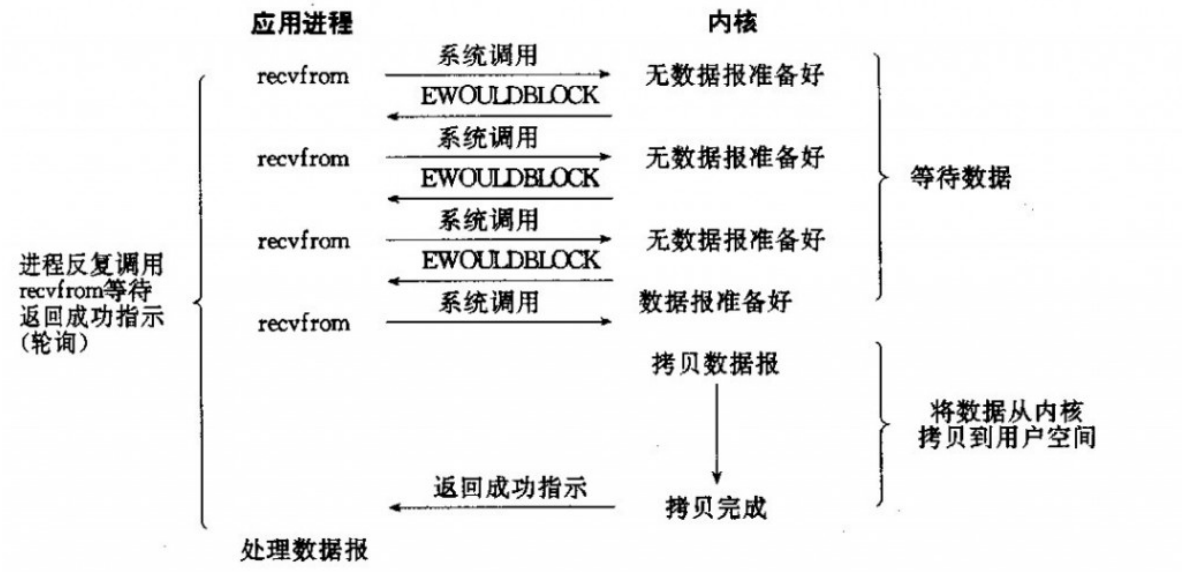
还是上面的例子， 你打电话问书店老板有没有《分布式系统》这本书，你如果是阻塞式调用，你会一直把自己“挂起”，直到得到这本书有没有的结果，如果是非阻塞式调用，你不管老板有没有告诉你，你自己先一边去玩了， 当然你也要偶尔过几分钟check一下老板有没有返回结果。 在这里阻塞与非阻塞与是否同步异步无关。跟老板通过什么方式回答你结果无关。

IO模型（Reference Link） 在linux系统下面，根据IO操作的是否被阻塞以及同步异步问题进行分类，可以得到下面五种IO模型

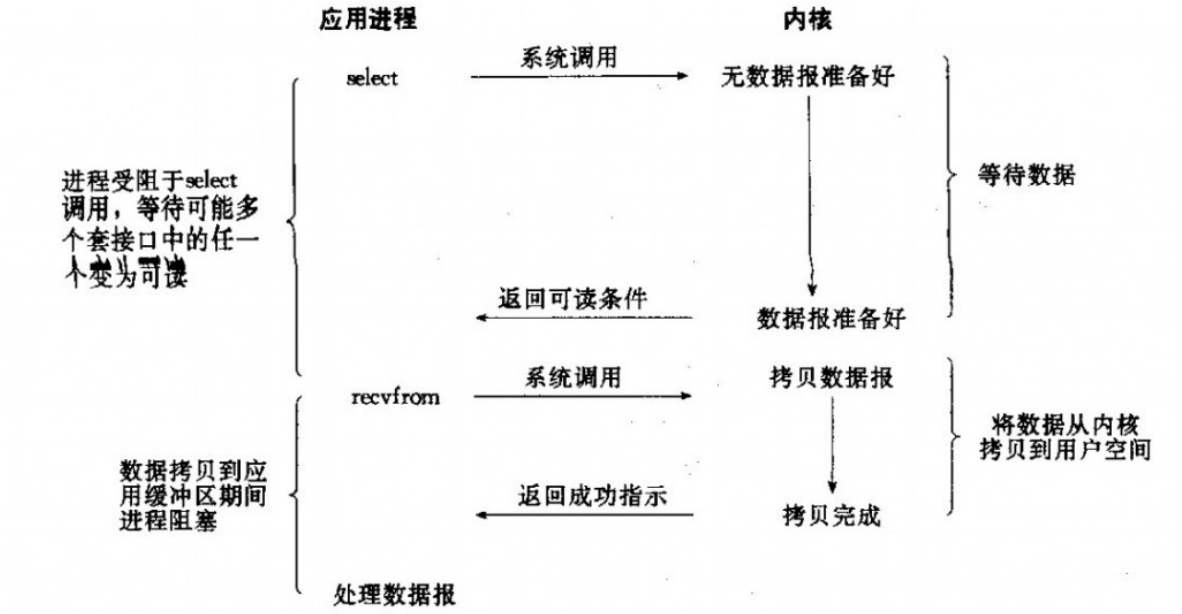
阻塞I/O模型 最常见的I/O模型是阻塞I/O模型，缺省情形下，所有文件操作都是阻塞的。我们以套接口为例来讲解此模型。在进程空间中调用recvfrom，其系统调用直到数据报到达且被拷贝到应用进程的缓冲区中或者发生错误才返回，期间一直在等待。我们就说进程在从调用recvfrom开始到它返回的整段时间内是被阻塞的。



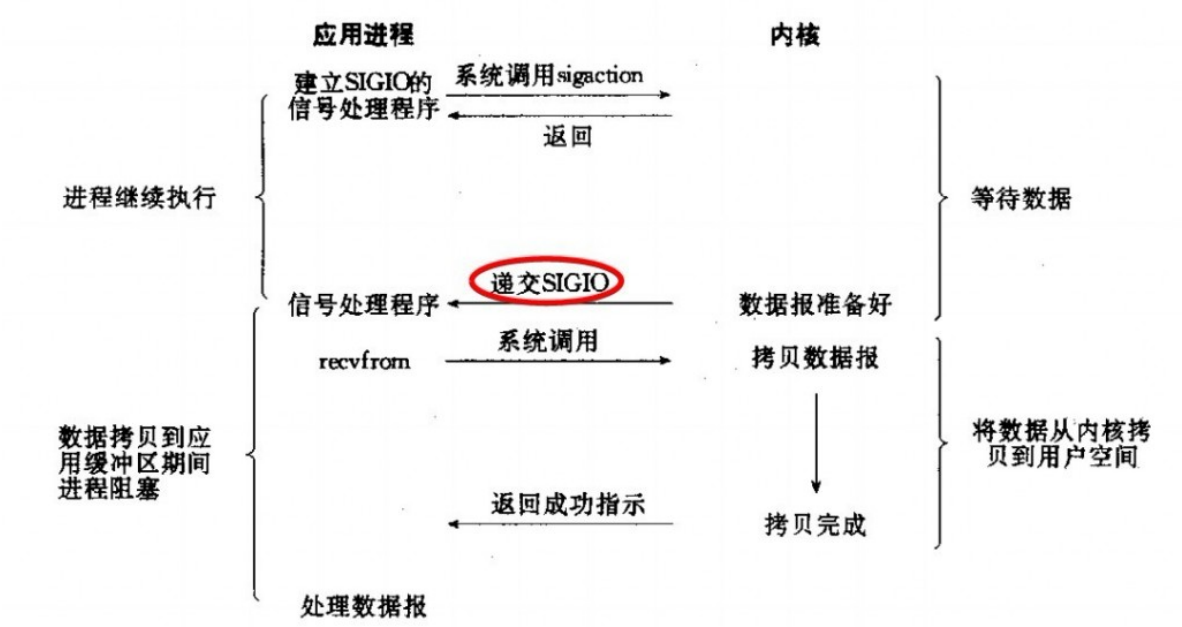
非阻塞I/O模型 进程把一个套接口设置成非阻塞是在通知内核：当所请求的I/O操作不能满足要求时候，不把本进程投入睡眠，而是返回一个错误。也就是说当数据没有到达时并不等待，而是以一个错误返回。



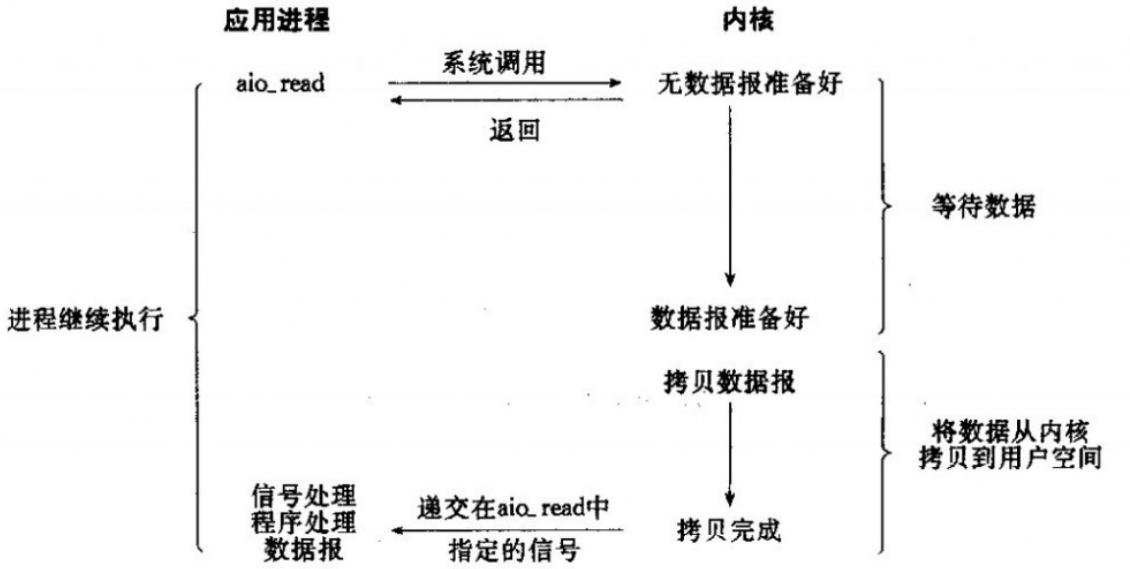
I/O复用模型 linux提供select/poll，进程通过将一个或多个fd传递给select或poll系统调用，阻塞在select;这样select/poll可以帮我们侦测许多fd是否就绪。但是select/poll是顺序扫描fd是否就绪，而且支持的fd数量有限。linux还提供了一个epoll系统调用，epoll是基于事件驱动方式，而不是顺序扫描,当有fd就绪时，立即回调函数rollback；



信号驱动异步I/O模型 首先开启套接口信号驱动I/O功能, 并通过系统调用sigaction安装一个信号处理函数（此系统调用立即返回，进程继续工作，它是非阻塞的）。当数据报准备好被读时，就为该进程生成一个SIGIO信号。随即可以在信号处理程序中调用recvfrom来读数据报，并通知主循环数据已准备好被处理中。也可以通知主循环，让它来读数据报。



异步I/O模型 告知内核启动某个操作，并让内核在整个操作完成后(包括将数据从内核拷贝到用户自己的缓冲区)通知我们。这种模型与信号驱动模型的主要区别是：信号驱动I/O：由内核通知我们何时可以启动一个I/O操作；异步I/O模型：由内核通知我们I/O操作何时完成。



总结 前四种都是同步IO，在内核数据copy到用户空间时都是阻塞的。 最后一种是异步IO，通过API把IO操作交由操作系统处理，当前进程不关心具体IO的实现，通过回调函数，或者信号量通知当前进程直接对IO返回结果进行处理。

首先一个IO操作其实分成了两个步骤：发起IO请求和实际的IO操作，同步IO和异步IO的区别就在于第二个步骤是否阻塞，如果实际的IO读写阻塞请求进程，那么就是同步IO，因此阻塞IO、非阻塞IO、IO复用、信号驱动IO都是同步IO，如果不阻塞，而是操作系统帮你做完IO操作再将结果返回给你，那么就是异步IO。阻塞IO和非阻塞IO的区别在于第一步，发起IO请求是否会被阻塞，如果阻塞直到完成那么就是传统的阻塞IO，如果不阻塞，那么就是非阻塞IO。

AIO，BIO，NIO AIO异步非阻塞IO，AIO方式适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。

NIO同步非阻塞IO，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。

BIO同步阻塞IO，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。

Java对BIO、NIO、AIO的支持： 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，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。（底层是epoll） 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。

AIO（Reference Link1，ReferenceLink2） 一般来说，服务器端的I/O主要有两种情况：一是来自网络的I/O；二是对文件(设备)的I/O。Windows的异步I/O模型能很好的适用于这两种情况。而Linux针对前者提供了epoll模型，针对后者提供了AIO模型(关于是否把两者统一起来争论了很久)。

NIO（Reference Link） epoll,select/poll(Reference Link) 都是IO复用，I/O多路复用就通过一种机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。但select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。 epoll的效率更高，优化了select的轮询操作，通过callback事件响应方式。 epoll除了提供select/poll那种IO事件的水平触发（Level Triggered）外，还提供了边缘触发（Edge Triggered），这就使得用户空间程序有可能缓存IO状态，减少epoll_wait/epoll_pwait的调用，提高应用程序效率。

LT&&ET（epoll） LT（level triggered）是缺省的工作方式，并且同时支持block和no-block socket.在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的fd进行IO操作。如果你不作任何操作，内核还是会继续通知你的，所以，这种模式编程出错误可能性要小一点。传统的select/poll都是这种模型的代表。 ET （edge-triggered）是高速工作方式，只支持no-block socket。在这种模式下，当描述符从未就绪变为就绪时，内核通过epoll告诉你。然后它会假设你知道文件描述符已经就绪，并且不会再为那个文件描述符发送更多的就绪通知，直到你做了某些操作导致那个文件描述符不再为就绪状态了（比如，你在发送，接收或者接收请求，或者发送接收的数据少于一定量时导致了一个EWOULDBLOCK 错误）。但是请注意，如果一直不对这个fd作IO操作（从而导致它再次变成未就绪），内核不会发送更多的通知（only once），不过在TCP协议中，ET模式的加速效用仍需要更多的benchmark确认。 ET和LT的区别就在这里体现，LT事件不会丢弃，而是只要读buffer里面有数据可以让用户读，则不断的通知你。而ET则只在事件发生之时通知。可以简单理解为LT是水平触发，而ET则为边缘触发。LT模式只要有事件未处理就会触发，而ET则只在高低电平变换时（即状态从1到0或者0到1）触发。[1]

select的几大缺点： （1）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很多时会很大

（2）同时每次调用select都需要在内核遍历传递进来的所有fd，这个开销在fd很多时也很大

（3）select支持的文件描述符数量太小了，默认是1024

poll实现 poll的实现和select非常相似，只是描述fd集合的方式不同，poll使用pollfd结构而不是select的fd_set结构.

epoll（reference Link） epoll既然是对select和poll的改进，就应该能避免上述的三个缺点。那epoll都是怎么解决的呢？在此之前，我们先看一下epoll和select和poll的调用接口上的不同，select和poll都只提供了一个函数——select或者poll函数。而epoll提供了三个函数，epoll_create,epoll_ctl和epoll_wait，epoll_create是创建一个epoll句柄；epoll_ctl是注册要监听的事件类型；epoll_wait则是等待事件的产生。

对于第一个缺点，epoll的解决方案在epoll_ctl函数中。每次注册新的事件到epoll句柄中时（在epoll_ctl中指定EPOLL_CTL_ADD），会把所有的fd拷贝进内核，而不是在epoll_wait的时候重复拷贝。epoll保证了每个fd在整个过程中只会拷贝一次。

对于第二个缺点，epoll的解决方案不像select或poll一样每次都把current轮流加入fd对应的设备等待队列中，而只在epoll_ctl时把current挂一遍（这一遍必不可少）并为每个fd指定一个回调函数，当设备就绪，唤醒等待队列上的等待者时，就会调用这个回调函数，而这个回调函数会把就绪的fd加入一个就绪链表）。epoll_wait的工作实际上就是在这个就绪链表中查看有没有就绪的fd（利用schedule_timeout()实现睡一会，判断一会的效果，和select实现中的第7步是类似的）。

对于第三个缺点，epoll没有这个限制，它所支持的FD上限是最大可以打开文件的数目，这个数字一般远大于2048,举个例子,在1GB内存的机器上大约是10万左右，具体数目可以cat /proc/sys/fs/file-max察看,一般来说这个数目和系统内存关系很大。

使用mmap加速内核与用户空间的消息传递。无论是select,poll还是epoll都需要内核把FD消息通知给用户空间，如何避免不必要的内存拷贝就很重要，在这点上，epoll是通过内核于用户空间mmap同一块内存实现的。

总结 select，poll实现需要自己不断轮询所有fd集合，直到设备就绪，期间可能要睡眠和唤醒多次交替。而epoll其实也需要调用epoll_wait不断轮询就绪链表，期间也可能多次睡眠和唤醒交替，但是它是设备就绪时，调用回调函数，把就绪fd放入就绪链表中，并唤醒在epoll_wait中进入睡眠的进程。虽然都要睡眠和交替，但是select和poll在“醒着”的时候要遍历整个fd集合，而epoll在“醒着”的时候只要判断一下就绪链表是否为空就行了，这节省了大量的CPU时间。这就是回调机制带来的性能提升。

select，poll每次调用都要把fd集合从用户态往内核态拷贝一次，并且要把current往设备等待队列中挂一次，而epoll只要一次拷贝，而且把current往等待队列上挂也只挂一次（在epoll_wait的开始，注意这里的等待队列并不是设备等待队列，只是一个epoll内部定义的等待队列）。这也能节省不少的开销。

IOCP （ReferenceLink|ConcreteRealization） I/O完成端口,是Windows环境下的异步IO处理模型。IOCP采用了线程池+队列+重叠结构的内核机制完成任务。需要说明的是IOCP其实不仅可以接受套接字对象句柄，还可以接受文件对象句柄等。

IOCP本质是一种线程池的模型，当然这个线程池的核心工作就是去调用IO操作完成时的回调函数。是WINDOWS系统的一个内核对象。通过此对象，应用程序可以获得异步IO的完成通知。