[**IO与SOCKET IO散记**](http://shift-alt-ctrl.iteye.com/blog/1840402)

**对于File IO**,磁盘驱动器将数据从磁盘中"汲取"到Kernel space(系统内存),此过程将采用DMA(直接内存存取)方式;不消耗CPU周期.对于用户进程,如果想操作数据,**需要将kernel space中的数据buffer复制到user space(即进程空间).**因为进程不具有特权,无法直接操作kernel space以及磁盘,进程操作需要通过OS内核做调度.

**虚拟内存:**目前所有的操作系统都使用了"虚拟内存"策略,一个或多个虚拟内存地址可以对应一个物理内存地址,虚拟内存地址大小可以远大于物理内存大小.因为设备驱动器无法通过DMA访问user space,但可以通过将kernel space以虚拟内存地址的方式映射到物理内存上,让"user space"使用虚拟内存地址的方式操作数据,DMA(只能操作物理内存)将buffer填充,那么它(buffer)将会对kernel与user space都可见;因此,"虚拟内存"可以避免buffer的复制,但是需要kernel与user space共享页片(page segment);buffer必须为磁盘block尺寸的整数倍以及为2的次方.(一般block尺寸为512byte);操作系统将内存地址空间分割为页来管理(page),已更加快捷的定为地址.

因为虚拟内存可以比物理内存更大且需要快读定位,它也需要分页.虚拟内存可以被swap并持久存储在磁盘上,以方便为其他虚拟页提供内存空间..此外,物理内存扮演了cache作用和hotspot,当页被swap到磁盘后,其所占有的物理内存地址也将被移除.

现代CPU都有一个**MMU(内存管理单元)**子系统,此设备逻辑上位于CPU和物理内存之间,它持有将虚拟内存地址翻译(转换)成物理内存地址的映射关系;当CPU引用了一个内存位置,MMU来决定页的位置以及将虚拟页码(page number)转换成物理页码,如果当前没有有效的映射,它将向CPU提交一个页故障(page fault)信号.

页故障导致一个trap(圈套,诱捕),kernel将会逐步校验页,将会调度一个"页码调进"(pagein)操作来将丢失的页重新加入到物理内存(这些页存储在磁盘上),同时也会导致另外一个操作,将其他页转存到磁盘上,为最新加载的数据腾出物理内存空间.一旦页故障解决,MMU将会重建虚拟内存和物理内存的映射,用户进程操作将可以继续;这些工作都是系统所为,用户进程感觉不到内部发生的"故障".

**对于常规的FILE I/O**,用户的read()和write()调用都会引起数据在kernel space和user space之间的复制,这是因为在文件系统与用户buffer之间没有一个一对一的基准;不过有一种特定的I/O操作允许用户进程最大程度上使用基于页的系统I/O优势,**可以完全避免buffer复制,这就是"内存映射IO"(memory-mapping file).**

内存映射IO使文件系统在user space与适用的文件系统页建立虚拟内存映射;这有几个优点:

1. 用户进程犹如操作内存一样操作文件,无须read和write的系统调用
2. 当用户进程接触到内存映射空间时,页故障会自动触发并将数据复制到内存中,如果用户修改了内存映射空间,受影响的页将会被标记为dirty,此后将会被依次更新到磁盘文件.
3. 数据总是页对齐的(page aligned),不存在buffer复制问题
4. 大尺寸文件能够再无需消耗太多内存的情况下被映射.

**文件锁**,File lock分为两种:共享锁/排他锁;多个共享锁可以同时对文件或者文件区域有效;排他锁在同一文件区域不能与其他类型的文件锁重叠.

共享锁的获取是任意的,对于文件read均会获得共享锁,如果一个进程需要更新文件中的区域,它需要获得排他锁,将会阻塞直到其他类型锁(共享锁/排他锁)被释放;一旦获得排他锁,其他read进程或者write进程都将阻塞,直到排他锁释放.

并不是所有的I/O都是基于block的(区别于文件IO),还有stream IO,它基于pipeline模式,流IO中字节是有顺序的,比如priter,网络IO等.非阻塞流IO需要"就绪选择"(readiness selection)的能力,需要操作系统检测流的状态,操作系统会被告知来监视stream集合列表,以返回给进程那些流已经就绪;这个能力就允许一个进程来使用普通代码在单线程模式下处理多个活跃stream;它被网络服务器广泛的用来处理大量网络连接.

**JAVA NIO部分拾遗:**

* buffer的equals方法只会在remaining中所有byte都相等的情况下才返回true.
* channel.close()方法将会导致线程阻塞,直到底层IO结束,即使在非阻塞模式下也一样.
* FileChannel不能使用非阻塞模式,也将无法使用selector.FileOutputStream将强制获取排他锁,FileInpytStream将获得共享锁.
* FileLock为API级别锁,它将无法再多线程环境中使用,对同一个fileChannel调用tryLock,也只能有一个线程获得成功,每次调用tryLock都将触发一次本地调用;只要有进程获得锁都不行,即使同一个JVM的不同线程(甚至是同一个线程的连续调用)都不行.
* 非阻塞IO,通常被server端使用已在长连接高并发环境中使用而获益,client端也可以使用.
* Socket channels是线程安全的,多线程无需特殊的同步操作仍可以安全的使用它们,在任何时间只能有一个read或者write被执行.不过需要注意,因为socket数据操作基于流的,不是基于packet,它能保证byte发送的循序是按照read/write的调用顺序,但是不保证数据是组织良好的.在NIO模式下,并发的调用read/write,有可能单个线程获得的数据是"组织不良好"的,如果确保数据的成帧性,需要特殊的同步过程,或者队列化数据包.
* register()方法定义在SelectableChannel类中,channels向selectors注册;一个selector维护了channels列表;一个channel可以注册多个selector,但是不知道当前注册的是哪个selector对象;一般场景中,我们期望一个channel注册一个selector.(否则可能多个selector触发几乎一样的事件).

**Java代码  [收藏代码](javascript:void())**

1. //网络通讯协议栈示意图:
2. application
3. |
4. Socket
5. |
6. UDP/TCP
7. |
8. IP
9. |
10. HOST------>Channel + byteBuffer -->路由节点--->Channel + buffer-->HOST

**网络协议:**

其中IP协议处于网络层,UDP/TCP处于传输层协议.

**IP协议**提供了一种数据报服务:每组分组报文都有网络独立处理和分发,就像信封或者包裹通过邮政系统发送一样,IP报文必须包括一个保存村其目的地址字段(address);IP协议只是一个"best effort"(尽力而为)的协议,它试图分发每一个分组报文,但是在网络传输过程中,偶尔也会发生丢失报文/使报文顺序被打乱或者重新发送报文的情况.

**TCP/UDP协议**的共同功能就是寻址,IP协议只能将分组报文发送到不同的主机,很明显,还需要更细颗粒度的寻址将报文发送到主机的不同应用程序(端口);TCP/UDP协议也成为端对端协议.TCP协议能够检测和回复IP层提供的主机到主机信道中可能发生的报文丢失/重发以及其他错误,TCP提供了可信赖的字节流信道,应用程序无需再对错误进行处理.UDP协议并不尝试对IP层产生的错误进行修复,它仅仅简单的扩展了IP协议的best-effort数据报服务,因此UDP协议的应用程序需要自己处理数据丢失或者重发的错误等.同一个TCP(不包括UDP)的一次write操作输出的数据,将有可能在网络发送时分成多个chunks,并且每个chunks会单独对路由重排之后一次发送,所有对于socket端read操作,可能先后收到不同的chunks,此后再对数据进行整理.

shutdownInput()将会导致TCP流的输入端关闭,已经接受的但尚未被read的数据将会被丢弃.在阻塞模式下read方法至少得到一个字节后返回,并非读取所有的数据(buffer满)才会返回.

**如果TCP socket输出流发送了大量的数据,但是对等端不进行read操作,将可能导致输出流阻塞,甚至带来致命问题.**

UDP协议基于IP协议,并增加对port寻址,会对传输的数据进行简单的检测并丢弃损坏/尺寸过载的数据.UDP协议保留了packet消息的边界,每次DatagramSocket的receive操作最多只能接收一个数据包.

TCP和UDP在数据发送机制上有些不同:TCP Socket的输出流上调用write方法返回后,此时数据已经被复制到sendBuffer中,此后数据被发送,可能已经到达;对于UDP而言,它无需数据恢复机制,因此不需要对发送的数据进行缓冲区复制,而是直接发送,当send()方法返回后,数据已经正在被发送或者已经达到.

TCP是基于链接的，即在发送实际数据之前，TCP的双端必须经过三次握手并建立链接(双方互通，并感知对方的address)；UPD是无连接的，尽管在API级别具有connect/bind方法，这些方法只不过是在本地创建了Socket句柄，并持有远端的address，只需要在发送数据时知道远端地址即可。

**编码:**

ASCII码将英文字母,数字,标点符号以及一些特殊符号(不可打印符号,\n等)映射到-~127的整数(一个字节);java中使用了unicode的国际标准编码字符来表示char和string,unicode字符集将世界上大部分语言和字符映射到0~65535(short类型,16位)之间;

unicode包含了ASCII码,且映射值和ASCII码一致,unicode对ASCII码提供了兼容性.UTF-8是unicode编码的分支,8位为一个字节,中文在UTF-8中占用3个字节,ASCII一个字节,其中中文编码为负值,UTF-8遇到负值则连续读入3个字节;GBK为双字节编码,任何字符均为双字节(为了兼容ASCII码,其在GBK中仍未一个字节).

**流操作:**

BufferedInput/BufferedOutputStream是基于内部的buffer操作,任何对数据的read或者write都首先填充buffer,当buffer满或者调用flush或者close(),才会导致数据的传输.它能够提高网络通讯的性能.因为Socket通讯中,其inputStream(实为SocketInputStream)和outputStream可以被java.io中其他stream进行包装,来提高程序的性能或者可操作性.

Buffered\*Steam:基于buffer,提高性能.

Data\*Stream:基于特定字节长度的数据类型转换,可操作性强.

Checked\*Stream:基于数据校验操作,例如CheckSum等.

Ciper\*Stream:数据加密

GZIP\*Stream:数据压缩

Object\*Stream:java序列化,提供了数据操作,一般我们不会再网络IO中使用.

**DatagramSocket数据的边界有协议设定**,接受者或者发送者无需特殊关注.**对于TCP而言,协议没有设定数据边界,因此读取数据时需要对数据的边界进行特殊处理,即成帧技术**,成帧技术大概有2中方式,一个是在流中填充自定义的结束定界符来标记一个数据包的结束,一种是使用meta操作,即首先声明和发送数据帧的长度和校验信息,此后再发送实际数据,读取方只读取指定长度的数据即可.

基于meta方式是比较常见的,比如发送数据的顺序如:[MAGIC-HEADER][CHECKSUM][DATA-LENGTH][DATA...]

基于TCP的多线程编程模式对于server端收益很大,一般我们采取一个socket连接由单独线程处理,或者加入线程池.Socket通信中具有阻塞特点的操作:read,write,accept.如果信道中没有数据时read将会阻塞,如果缓冲区已满,write操作将会阻塞,对于accpet当serverSocket端没有侦听到连接时将会阻塞.

**广播与多播:**

通信中有2中类型的一对多服务:广播(broadcast)和多播(multicast).对于广播,本地网络中所有的主机都会收到数据副本.对于多播,只有侦听多播组的主机才会收到数据.这些都需要UDP协议的支持.

广播UDP数据报文和单播报文相似,唯一的区别就是其使用的是广播地址而非单播地址,IPV6没有明确提供广播地址,然而一个特殊的全节点的本地连接范围的多播地址,FFO2::1,发送给此地址的消息都会多播到一个连接上的所有节点.IPV4的本地广播地址为:

255.255.255.255,消息发送到在同一广播网络上的每个主机.本地广播不会被路由器转发,而且大部分路由器都不支持广播消息转发.

本地广播功能还是非常有用,它通常用于在网络游戏中处于同一本地网络的玩家之间交换状态信息.

多播:与广播一样,多播与单播之间的主要区别是地址的形式,一个多播地址指示了一组接受者,IPV4中多播地址范围为224.0.0.0~239.255.255.255.IPV6中所有以FF开头的地址.对于多播,为了避免packet在网络中不断转发,可以设置TTL,当TTL被路由器转发一次将会递减,当TTL为0时,packet将不再被转发.在设置TTL时最好计算一下路由节点的个数,已确保消息能够被接收到.可以通过setLoopbackMode(boolean)来设置多播的消息是否会回传给自己.setBroadcast(boolean on) 方法可以指定当前datagramSocket是否接受多播和广播数据.

**DatagramSocket本身线程安全,几乎所有的方法都具有同步.**

SendBufferSize和ReceiveBufferSize的设定只是一个建议值,实际大小可能会有差异.ServerSocket如果需要对accept之后的Socket设置ReceiveBufferSize,那么它应该在accpt之前就对ServerSocket设置,那么此后accpet返回的Socket就会"继承"这个参数,因为Socket被accept之后,就可以立即接受数据,如果在accept之后设置有可能不会生效..对于sendBufferSize的设置,因为socket可以控制send的时机,只要在send之前设置就可.

**setSoTimeout(int)此值将会影响read和accept阻塞的时间.**

对于HTTP协议而言,是由服务器发起关闭连接,客户端首先向server发送"GET"请求,然后服务器返回响应头部信息,此后跟着请求文件内容,当数据发送完毕后,server端关闭socket连接,因为客户端不能确定文件的大小,只能通过关闭socket来指示文件的结束.

**SOCKET通讯中,使用多线程可以让server收益,但是如果socket都是长连接,线程池(多线程)仍然限制了并发处理的能力,**因为很多时候连接处于空闲状态.同时,在多线程环境中,如果socket需要访问共享数据(例如数据发送队列),对共享数据的同步仍会带来一定的瓶颈,因为同步锁让并发再次串行,而且还存在线程上下文切换的开支问题.**所以我们需要NIO.**

**Buffer:**

直接缓冲区没有后援数组（并不是有JAVA buffer保存）,调用array()或者arrayOffset()方法将会抛出UnsupportedOperationException异常.对于直接缓冲区,调用allocateDirect()方法并不能保证保证一定能够分配成功,有可能平台或者JVM不支持,最终仍然会得到heapBuffer;所以方法返回后需要调用isDirect()检测;直接缓冲区的分配和销毁需要消耗更多的系统资源,因为其后援数组在JVM之外,对它的管理需要与系统直接交互,只有当很多IO操作上长时间使用时,才考虑分配直接缓冲区;直接缓冲区可以明显的提高系统性能.

需要注意对于byteBuffer,其put方法仍然受到limit的限制,如果在执行put时,数据超过了limit,将抛出异常.

dunplicate()方法也会复制原数组的postion和limit和mark.asReadOnlyBuffer()也会"继承"原数组的position/limit和mark信息.对于asCharBuffer/asLongBuffer方法将不继承position,但是其limit和capacity会按照字节比例而缩小.

**Socket注意事项:**

Socket通讯仍然有死锁的可能:

1) socket建立连接后,双方都read,而不去write将会导致死锁,即互相等待.此外双端都write而不read同样造成死锁.

2) socket两端都在单线程模式下,进行大数据的read和write,也将导致死锁,比如我们设置sendBufferSize和receiveBufferSize都很小,但是一端write一个较大尺寸的数据,另一端read之后直接write回来,也会导致死锁,因为bufferSize在满载之后,会导致一端write阻塞,但是另一端read之后写入也遇到了相同的问题(因为另一端已经阻塞,此时它的write的数据无法被read,将阻塞).

**对于这种问题,可以考虑将read和write在两个线程中执行,即可避免（但通常不这么做）.**

sendBufferSize和receiveBufferSize对性能优化有很大帮助.通常何时的bufferSize可以在满足要求的情况下提高网络传输效率.较小的bufferSize将有更大的几率造成阻塞,以及导致更多的网络传输次数.

关闭TCP连接的最后微妙之处在于TIME-WAIT状态,执行close之后,连接不会立即关闭,而是进入了time-wait状态,直到双端都传输--接受数据完成之后,转换为close状态.进入time-wait之后,数据的发送就是"尽力而为",有可能网络故障或者主机故障导致数据不能完全发送.time-wait状态并不释放本地端口,所以此时在端口上重新注册socket,将被拒绝.**可以通过设置reused选项来避免这个问题**,但是它更多的时候是适用于server端.

多路复用的一个秘密就是,同一个机器的不同socket可以有相同的本地端口和本地地址,serverSocket就是如此.通过serverSocket的accept方法接受的新socket将和serverSocket使用相同的本地端口,显然,要确定传入的分组报文应该分配到哪个socket上(解调多路复用)不仅仅是查看分组报文的目的地址和端口.

**一对socket,需要localPort + localAddress + remotePort + remoteAddress四个参数来决定解调复用.**所以对于serverSocket端持有的socket,需要remotePort和remoteAddress做为区分数据路由.