NIO

1. NIO原理



* 1. linux下的5种IO
     1. 基本概念

用户空间与内核空间

1、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)。

2、操作系统的核心是内核（内核也是应用程序），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。

3、为了保证用户进程不能直接操作内核(kernel)，保证内核的安全，操作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0×00000000到0xBFFFFFFF)，供各个进程使用，称为用户空间

进程切换

为了控制进程的执行，**内核必须有能力挂起正在CPU上运行的进程，并恢复以前挂起的某个进程的执行**。这种行为被称为**进程切换**。

从一个进程的运行转到另一个进程上运行，这个过程中经过下面这些变化：

1、保存处理机上下文（包括程序计数器和其他寄存器）。

2、更新PCB（进程控制块）信息。

3、把进程的PCB移入相应的队列，如就绪、在某事件阻塞等队列。

4、选择另一个进程执行，并更新其PCB。

5、更新内存管理的数据结构。

6、恢复处理机上下文

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

1、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。

2、文件描述符在形式上是一个**非负整数**。

3、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。

4、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。在程序设计中，一些涉及底层的程序编写往往会围绕着文件描述符展开。

但是文件描述符这一概念往往只适用于UNIX、Linux这样的操作系统。

同步与异步

**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。

**异步**，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。

阻塞与非阻塞

**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。

IO复用

复用也就是共用的意思。

这样理解还是有些抽象，为此，咱们来理解下复用在通信领域的使用，在通信领域中为了充分利用网络连接的物理介质，往往在同一条网络链路上采用**时分复用或频分复用**的技术使其在同一链路上传输多路信号；

到这里我们就基本上理解了复用的含义，即公用某个“介质”来尽可能多的做同一类(性质)的事，那IO复用的“介质”是什么呢?为此我们首先来看看服务器编程的模型：

客户端发来的请求服务端会产生一个进程来对其进行服务，每当来一个客户请求就产生一个进程来服务，然而进程不可能无限制的产生，因此为了解决大量客户端访问的问题，引入了IO复用技术，即：一个进程可以同时对多个客户请求进行服务。也就是说**IO复用的“介质”是进程**(准确的说复用的是select和poll，因为进程也是靠调用select和poll来实现的)，复用一个进程(select和poll)来对多个IO进行服务，虽然客户端发来的IO是并发的但是IO所需的读写数据多数情况下是没有准备好的，因此就可以利用一个函数(select和poll)来监听IO所需的这些数据的状态，一旦IO有数据可以进行读写了，进程就来对这样的IO进行服务。

select

* **1、select函数原型**

int select(int nfds, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, struct timeval \*timeout);

**int nfds：**fd\_set集合中最大描述符值加1;

其中，fd\_set是一个位数组，其大小限制为\_\_FD\_SETSIZE(1024)，位数组的每一位代表其对应的描述符是否需要被检查;

**fd\_set readfds:**

**fd\_set \*writefds:**

**fd\_set \*exceptfds:**

第二三四参数表示需要关注读、写、错误事件的文件描述符位数组；这些参数既是输入参数也是输出参数，可能会被内核修改**用于标示哪些描述符上发生了关注的事件**，所以每次调用select前都需要重新初始化fdset

**struct timeval \*timeout:超时时间；**

* **2、select函数的调用步骤：**

**(1)、使用copy\_from\_user从用户空间拷贝fdset（文件描述符集合）到内核空间；**

**(2)、注册回调函数\_\_pollwait；**

**(3)、遍历fdset中的所有fd（文件描述符），调用其对应的poll方法(对于socket，这个poll方法是sock\_poll，sock\_poll根据情况会调用到tcp\_poll,udp\_poll或者datagram\_poll)；**

**(4)、以tcp\_poll为例，其核心实现就是\_\_pollwait，也就是上面注册的回调函数。**

**\_\_pollwait的主要工作就是把current(当前进程)挂到设备的等待队列中，不同的设备有不同的等待队列，对于tcp\_poll 来说，其等待队列是sk->sk\_sleep(注意把进程挂到等待队列中并不代表进程已经睡眠了)。在设备收到一条消息(网络设备)或填写完文件数 据(磁盘设备)后，会唤醒设备等待队列上睡眠的进程，这时current便被唤醒了。**

**-------poll方法返回时会返回一个描述读写操作是否就绪的mask掩码，根据这个mask掩码给fd\_set赋值。**

**-------如果遍历完所有的fd，还没有返回一个可读写的mask掩码，则会调用schedule\_timeout使调用select的进程(也就是 current)进入睡眠。当设备驱动发生自身资源可读写后，会唤醒其等待队列上睡眠的进程。如果超过一定的超时时间(schedule\_timeout 指定)，还是没人唤醒，则调用select的进程会重新被唤醒获得CPU，进而重新遍历fd，判断有没有就绪的fd。**

**(5)、把fd\_set从内核空间拷贝到用户空间。**

poll

* 函数原型：

int poll(struct pollfd \*fds, nfds\_t nfds, int timeout);

**1）struct pollfd \*fds：**

struct pollfd{

　　int fd; //文件描述符

　　short events; //请求的事件，需要监听的事件

　　short revents; //返回的事件，已经发生的事件

　　};

events和revents是通过对代表各种事件的标志进行逻辑或运算构建而成的。

**2）第二个参数nfds：**

要监视的描述符的数目。

**3）最后一个参数timeout：**

是一个用毫秒表示的时间，是指定poll在返回前没有接收事件时应该等待的时间。

如果它的值为-1，poll就永远都不会超时。

如果整数值为32个比特，那么最大的超时周期大约是30分钟。

epoll

直到Linux2.6才出现了由内核直接支持的实现方法，那就是epoll，被公认为Linux2.6下性能最好的**多路IO就绪通知方法**，

* 函数原型：

int epoll\_wait(int epfd, struct epoll\_event \*events, int maxevents, int timeout);

总结

(1)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。**

(2)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。

* + 1. IO模型

同步IO模型

阻塞IO模型（同步阻塞）

工作原理：

应用程序(application)为了执行这个read操作，会调用相应的一个system call（系统调用），将系统控制权交给kernel，**然后**就进行等待(这其实就是被阻塞了)。kernel开始执行这个system call，执行完毕后会向应用程序返回响应，应用程序得到响应后，就不再阻塞，并进行后面的工作。

非阻塞IO模型

在linux下，应用程序可以通过设置文件描述符的属性O\_NONBLOCK，IO操作可以立即返回，但是并不保证IO操作成功。

也就是说，当应用程序设置了O\_NONBLOCK之后，执行write操作，调用相应的system call，这个system call会从内核中立即返回。但是在这个返回的时间点，数据可能还没有被真正的写入到指定的地方。也就是说，kernel只是很快的返回了这个 system call(只有立马返回，应用程序才不会被这个IO操作blocking)，但是这个system call具体要执行的事情(写数据)可能并没有完成。

同步非阻塞

应用程序主动地循环地去问kernel，写操作是否完成；

IO多路复用（异步阻塞IO）

原理：

应用程序要执行read操作，因此调用一个system call，这个system call被传递给了kernel。但在应用程序这边，它调用system call之后，并不等待kernel的返回结果而是立即返回，虽然立即返回的调用函数是一个异步的方式，但应用程序会被像select()、poll和epoll等具有复用多个文件描述符的函数阻塞住，一直等到这个system call有结果返回了，再通知应用程序。

也就是说，“在这种模型中，IO函数是非阻塞的，使用阻塞 select、poll、epoll系统调用来确定一个 或多个IO 描述符何时能操作。”所以，从IO操作的实际效果来看，异步阻塞IO和第一种同步阻塞IO是一样的，**应用程序都是一直等到IO操作成功之后(数据已经被写入或者读取)，才开始进行下面的工作。**

不同点在于**异步阻塞IO**用一个select函数可以为多个描述符提供通知，提高了并发性。举个例子：假如有一万个并发的read请求，但是网络上仍然没有数据，此时这一万个read会同时各自阻塞，现在用select、poll、epoll这样的函数来专门负责阻塞同时监听这一万个请求的状态，一旦有数据到达了就负责通知，这样就将之前一万个的各自为战的等待与阻塞转为一个专门的函数来负责与管理。

与此同时，异步阻塞IO和第二种同步非阻塞IO的区别在于：

**同步非阻塞IO**是需要应用程序主动地循环去询问是否有操作数据可操作；

**异步阻塞IO**是通过像select和poll等这样的IO多路复用函数来同时检测多个事件句柄来告知应用程序是否可以有数据操作。

信号驱动（异步非阻塞）

应用程序提交read请求的system call，然后，kernel开始处理相应的IO操作，而同时，应用程序并不等kernel返回响应，就会开始执行其他的处理操作(应用程序没有被IO操作所阻塞)。当kernel执行完毕，返回read的响应，就会产生一个信号或执行一个基于线程的回调函数来完成这次 IO 处理过程。

异步IO模型

异步IO的工作机制是：

告知内核启动某个操作，并让内核在整个操作（读或者写）完成后通知我们，这种模型与信号驱动的IO区别在于，信号驱动IO是由内核通知我们何时可以启动一个IO操作，这个IO操作由用户自定义的信号函数来实现；而异步IO模型是由内核告知我们IO操作何时完成。

IO模型总结

从理论上说，阻塞IO、IO复用和信号驱动的IO都是同步IO模型。因为在这三种模型中，IO的读写操作都是在IO事件发生之后由**应用程序来完成。**

而POSIX规范所定义的异步IO模型则不同：

对异步IO而言，用户可以直接对IO执行读写操作，这些操作告诉内核用户读写缓冲区的位置，以及IO操作完成后内核通知应用程序的方式。异步IO读写操作总是立即返回，而不论IO是否阻塞的，因为真主的读写操作已经由内核接管。也就是说，**同步IO模型要求用户代码自行执行IO操作(将数据从内核缓冲区读入用户缓冲区，或将数据从用户缓冲区写入内核缓冲区)，而异步IO机制则是由内核来执行IO操作(数据在内核缓冲区和用户缓冲区之间的移动是由内核在后台完成的)**。

你可以这样认为，同步IO向应用程序通知的是IO就绪事件，而异步IO向应用程序通知的是IO完成事件。linux环境下，aio.h头文件中定义的函数提供了对异步IO的支持。

* 1. 问题的提出

在传统的架构中，对于客户端的每一次请求，服务器都会创建一个新的线程或者利用线程池复用去处理用户的一个请求，然后返回给用户结果，这样做在**高并发的**情况下会存在非常严重的性能问题：

1、对于用户的每一次请求都创建一个新的线程是需要一定内存的；

2、同时线程之间频繁的上下文切换也是一个很大的开销。

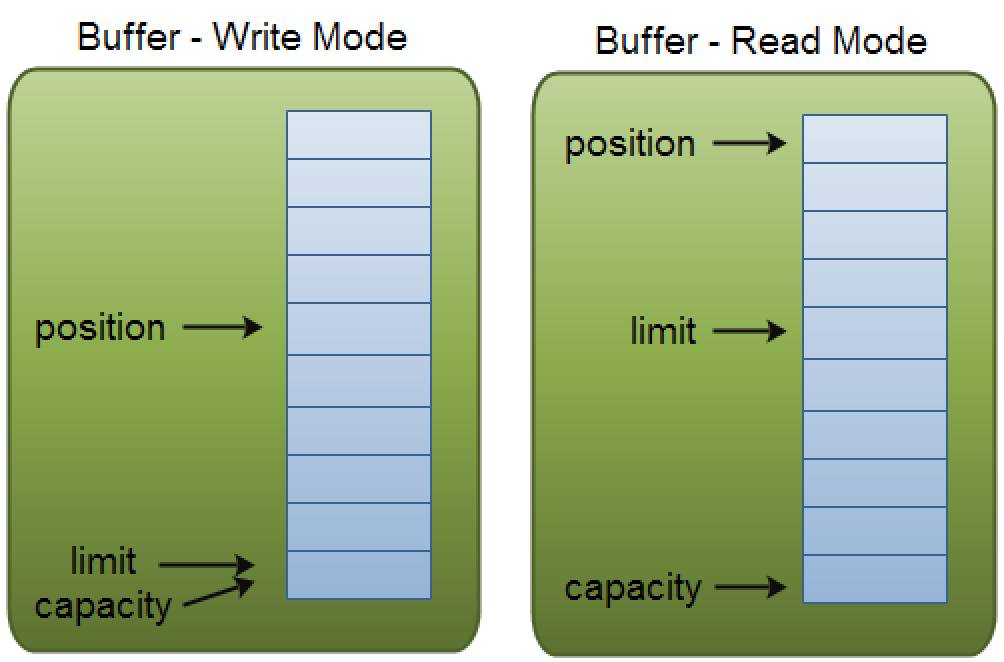
* 1. 基本概念
     1. Buffer（缓冲区）
* **对于缓冲区有四个重要的属性：**

capacity，缓冲区的容量，也就是我们上面指定的n。

position，当前指针指向的位置。

mark，前一个位置，这里我们下面再解释。

limit，最大能读取或者写入的位置。



* **put方法：**

byte变量x放到缓冲区中去，同时position加1。

* **get方法：**

从缓冲区中读取一个字节，同时position加一;

* **flip方法：**

如果我们想将buffer从**写数据的情况变成读数据**的情况，可以直接使用flip方法：

* **mark和reset**

mark是记住当前的位置用的，也就是保存position的值。

回到之前的位置可以调用reset会将mark的值重新赋给position，（前提是在这之前调用了mark方法）；

* + 1. Channel（通道）

当我们读取数据的时候，会先从buffer加载到channel；通过channel.write(buffer)则可以将buffer中的数据写入到到channel中

写入数据的时候，会先入到channel然后通过channel转移到buffer中去；通过channel.read(buffer)可以将channel中的数据写入到buffer中；

四种Channel

FileChannel从**文件**中读写数据。

DatagramChannel以**UDP的形式从网络中**读写数据。

SocketChannel以**TCP的形式从网络中**读写数据。

ServerSocketChannel允许你监听TCP连接;

实例：

* **服务端代码：**

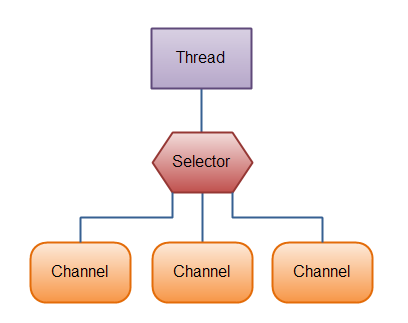
public class NIOServer {  
 public static void main(String[] args) throws IOException {  
 ServerSocketChannel ssc = ServerSocketChannel.*open*();  
 ssc.socket().bind(new InetSocketAddress("127.0.0.1",5000));  
 SocketChannel socketChannel = ssc.accept();  
  
 ByteBuffer readBuffer = ByteBuffer.*allocate*(128);  
 socketChannel.read(readBuffer);  
  
 readBuffer.flip();  
 while(readBuffer.hasRemaining()){  
 System.*out*.println((char)readBuffer.get());  
 }  
  
 socketChannel.close();  
 ssc.close();  
 }  
}

* **客户端代码：**

public class NIOClient {  
 public static void main(String[] args) throws IOException {  
 SocketChannel socketChannel = null;  
 socketChannel = SocketChannel.*open*();  
 socketChannel.connect(new InetSocketAddress("127.0.0.1",5000));  
 ByteBuffer writeBuffer = ByteBuffer.*allocate*(128);  
  
 writeBuffer.put("hello word".getBytes());  
  
 writeBuffer.flip();  
  
 socketChannel.write(writeBuffer);  
  
 socketChannel.close();  
 }  
}

* + 1. Selector

通过使用selector，我们可以通过**一个**线程来同时管理多个channel，省去了创建线程以及线程之间进行上下文切换的开销。



api

* **创建一个selector**

通过调用selector类的静态方法**open**我们就可以创建一个selector对象：

Selector selector = Selector.open();

* **注册channel**

为了保证selector能够监听多个channel，我们需要将channel注册到selector当中：

channel.configureBlocking(false);

SelectionKey key = channel.register(selector, SelectionKey.OP\_READ);

我们可以监听四种事件：

SelectionKey.OP\_CONNECT：当客户端的尝试连接到服务器

SelectionKey.OP\_ACCEPT：当服务器接受来自客户端的请求

SelectionKey.OP\_READ：当服务器可以从channel中读取数据

SelectionKey.OP\_WRITE：当服务器可以向channel中写入数据

* **通过SelectionKey获取注册的对应的Channel**

对SelectorKey调用channel方法可以得到key对应的channel:

Channel channel = key.channel();

而key自身感兴趣的监听事件也可以通过interestOps来获得：

int interestSet = selectionKey.interestOps();

* **获取所有已经注册的Channel**

对selector调用selectedKeys()方法我们可以得到注册的所有key：

Set<SelectionKey> selectedKeys = selector.selectedKeys();

实例：

客户端：

1. 基础教程



查看文档: [18763.html](http://www.importnew.com/18763.html)

1. scatter/gather(分散/聚集)



查看文档: [18857.html](http://www.importnew.com/18857.html)

**应用场景：**

经常用于需要将传输的数据分开处理的场合，

例如传输一个由消息头和消息体组成的消息，你可能会将消息体和消息头分散到不同的buffer中，这样你可以方便的处理消息头和消息体。

**工作方式：**

**读取channel：**

将多个buffer放入一个buffer数组中，将这个数组作为channel对象调用read函数的参数传入，当将channel中的数据填满第一个buffer时再向第二个里边写入数据。

**缺点：**

不能接受字节数不确定的内容；

**写入channel：**

将需要写入channel的buffer放入buffer数组中，将这个数组作为参数传入channel的write方法中，按照buffer在数组中的顺序将数组中的内容写入channel中，但是仅仅写入position之前的数据才会写入；

**优点**：

可以动态的写入长度不同的数据；

1. 通道之间的数据传输（transforTo/transforFrom）



**transforTo:**

将FileChannel中的数据传递到其他的channel中；

**transforFrom：**

将其他channel中的数据传输到FileChannel中；

从上边的两个函数的用法可知，只有在两个需要进行数据传输的channel中至少含有一个FileChannel时才可以使用；

1. Selector（监听者）



Selector采用的是监听者设计模式：

Selector为监听者，用来监听Channel的一些事件：

连接就绪；

接收就绪；

读就绪；

写就绪；

或者其中的一些事件的组合；