## 认识NIO

#### 一．Linux IO模型

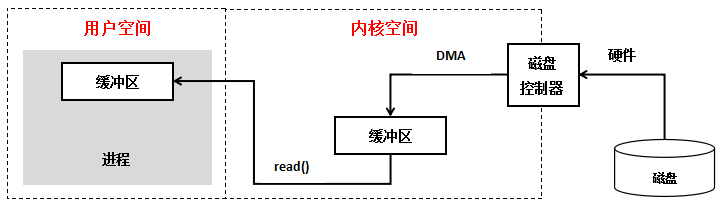
在《Unix网络编程》这本书中将IO模型划分为以下五种：

1. 阻塞式IO模型Blocking IO
2. 非阻塞式IO模型Non-Blocking IO/New IO
3. IO复用
4. 信号驱动式IO模型
5. 异步IO模型Asynchronous IO

其中前四种都是同步模式。

#### 了解Linux IO流程

在Linux中运行的应用程序如果需要进行IO操作，需要涉及到两个空间的概念，用户空间和内核空间。



上图表示的是一个数据读取的过程：

1. DMA先从将磁盘数据拷贝到内核空间，该过程也称为数据准备过程；
2. 应用程序拷贝内核空间数据到用户空间。
3. 写数据的过程与之相反。

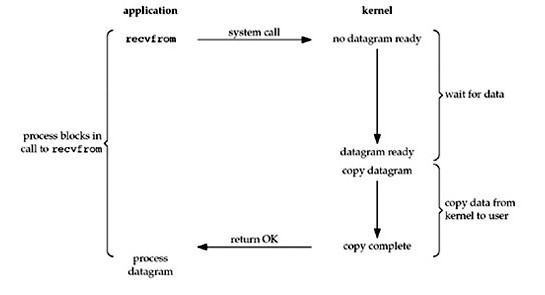
从应用程序角度来看，分为两步：

1. 等待内核将数据准备好（Waiting for the data to be ready）
2. 从内核向进程复制数据（Copying the data from the kernel to the process）

#### 各IO模型执行过程

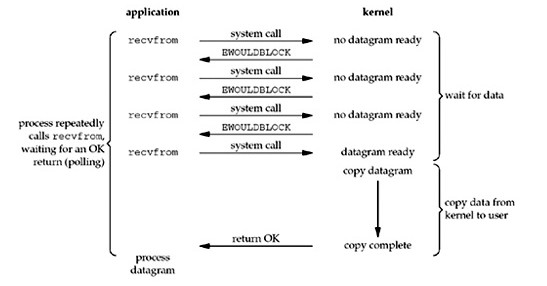
1. 阻塞式IO模型

应用程序发起一次系统调用，即IO请求，询问内核数据是否准备好，发现没有，则进程一直等待数据准备好为止；在接收到系统调用后内核开始准备数据报，准备好后进行内核态到用户态的数据拷贝，拷贝完成后，返回准备就绪信息给调用进程。



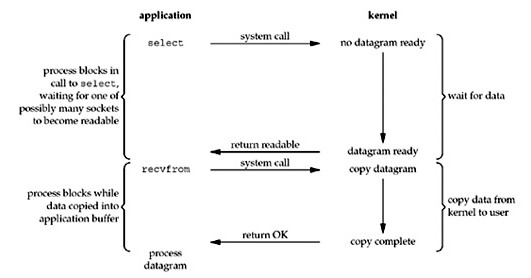
1. 非阻塞式IO模型

应用进程轮询询问内核数据是否准备好。当有数据报准备好时就进行数据报的拷贝操作，当没有准备好时，内核直接返回未准备就绪的信号，不让进程阻塞，并且开始准备数据，等待进程下一次询问。



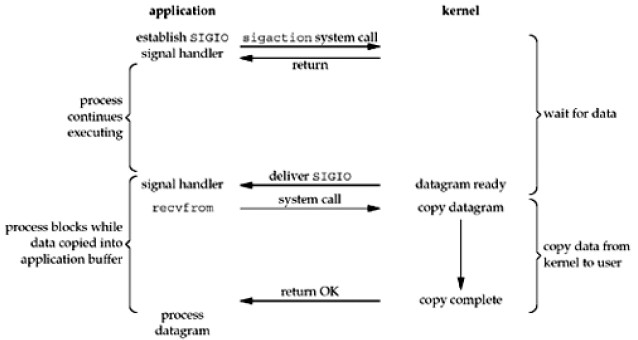
1. IO多路复用模型

IO多路复用模型会通过一个select函数对多个文件描述符（集合）进行循环监听，当某个文件描述符就绪时，就对这个文件描述符的数据进行处理。



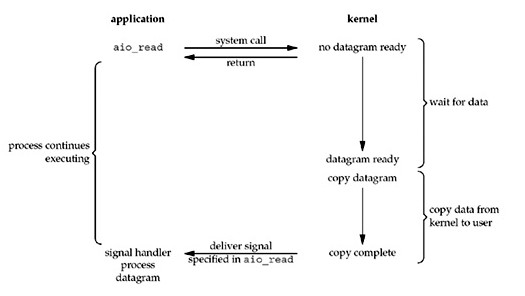
1. 信号驱动式IO模型

应用程序通知内核，当数据准备就绪时给它发送一个SIGIO信号，应用程序会对这个信号进行捕捉，并且调用相应的信号处理函数执行数据拷贝的过程。



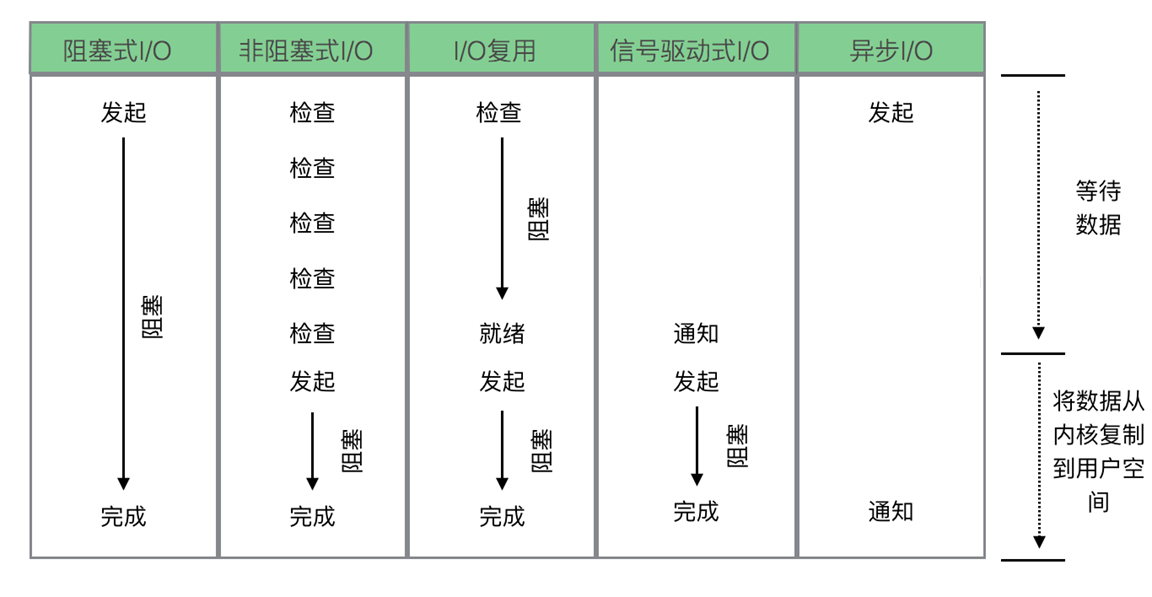
1. 异步IO模型

当应用程序调用aio\_read时，内核一边开始准备数据，另一边将程序控制权返回给应用进程，让应用进程处理其他事情；当内核中有数据报准备就绪时，由内核将数据报拷贝到用户空间，这也是与其他四种模型的最大区别，拷贝完成后返回aio\_read中定义好的函数处理程序。



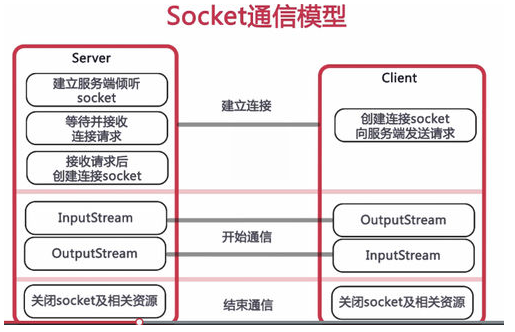
1. 各IO模型对比

从下图中可以看出阻塞程度是：阻塞式IO >非阻塞式IO > IO复用>信号驱动式IO >异步IO。



#### BIO下的Socket处理方式

下图是Socket和ServerSocket通信模型



###### Single Thread Socket Server

public void startServer() throws IOException {  
 final ServerSocket serverSocket = new ServerSocket(8080);  
 System.out.println("Listening for connection on port 8080...");  
 while (!Thread.interrupted()) {  
 final Socket socket = serverSocket.accept();  
 // 1. Read request from the socket of client.  
 // 2. Prepare a response.  
 // 3. Send response to the client.  
 // 4. Close the socket.  
 }  
}

###### Multiple Thread Socket Server

public void startServer() throws IOException {  
 final ServerSocket serverSocket = new ServerSocket(8080);  
 System.out.println("Listening for connection on port 8080...");  
 while (!Thread.interrupted()) {  
 final Socket socket = serverSocket.accept();  
 new Thread(() -> {  
 // 1. Read request from the socket of client.  
 // 2. Prepare a response.  
 // 3. Send response to the client.  
 // 4. Close the socket.  
 }).start();  
 }  
}

###### Thread Pool Socket Server

public void startServer() throws IOException {  
 final ServerSocket serverSocket = new ServerSocket(8080);  
 // 为了代码简洁，这里直接通过工具类创建一个线程池  
 ExecutorService executor = Executors.newFixedThreadPool(10);  
 System.out.println("Listening for connection on port 8080...");  
 while (!Thread.interrupted()) {  
 final Socket socket = serverSocket.accept();  
 executor.execute(() -> {  
 // 1. Read request from the socket of client.  
 // 2. Prepare a response.  
 // 3. Send response to the client.  
 // 4. Close the socket.  
 });  
 }  
}

#### Java NIO

Java IO和Java NIO的区别

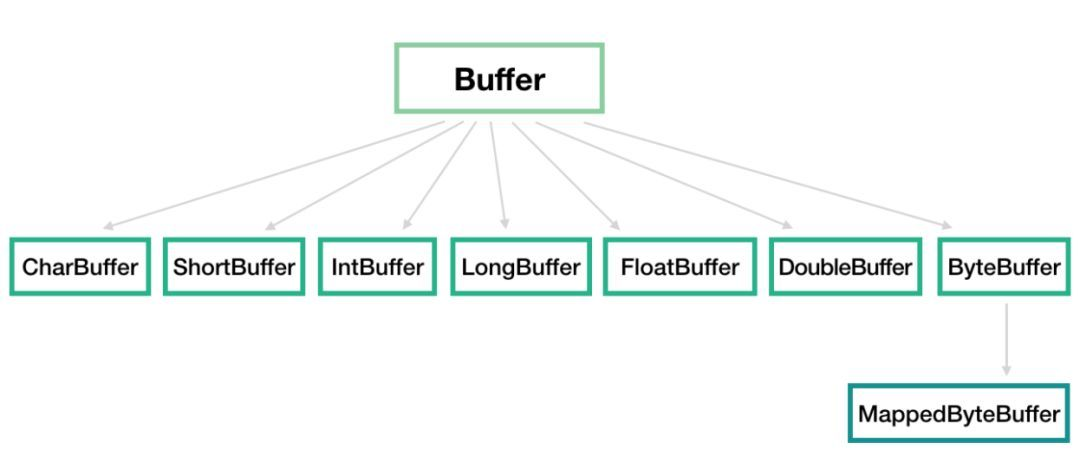


Java NIO三个关键对象

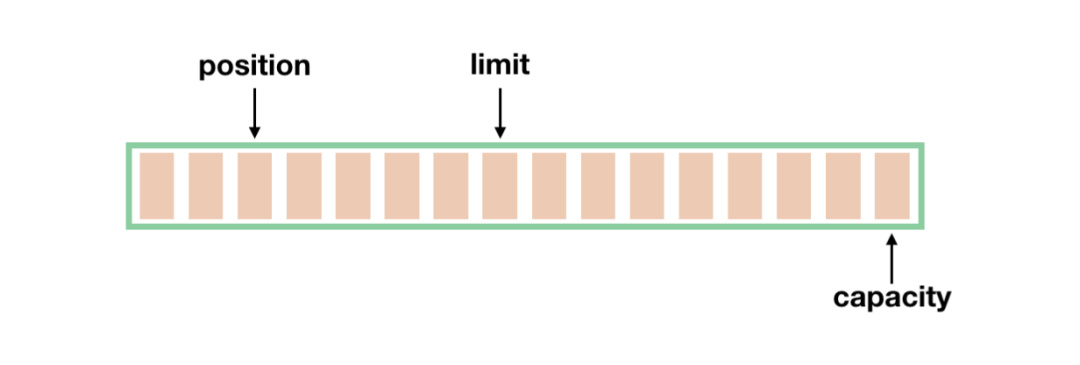
* Buffer
* Channel
* Selector

###### Java NIO Buffer

1. 一个Buffer本质上是内存的一个内存块，允许对这块内存进行数据读写操作，在Java NIO中定义了以下几种Buffer实现：



1. Java NIO的Buffer主要有三个核心属性：



（1）Capacity：

缓冲区容量，一旦设定就不可更改，比如capacity为1024的IntBuffer，代表其最大可以存放1024个int类型的数据。

（2）Position：

记录下一个可操作（可读/可写）地址。

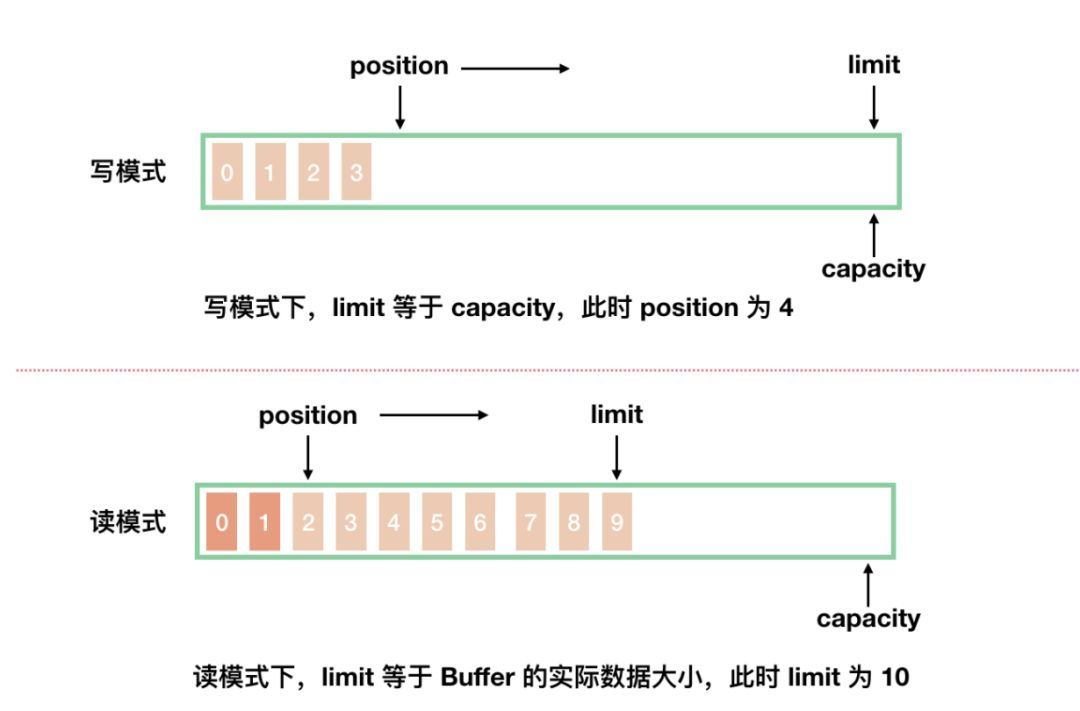
（3）Limit：

Java Buffer是通过一个指针来管理读写模式操作的，从写操作模式切换到读操作模式，position都会归零，这些可以保证从头开始读写，读写模式切换必须使用flip方法。

在初始化后默认是写操作模式，此时limit代表的是最大能写入数据，即limit = capacity；

最大能写入的数据，初始状态下limit = capacity。

写操作结束后，flip切换到读模式下，此时limit等于Buffer中实际写入数据的大小，比如在写模式下写入了10个int类型的数据，那么此时limit=10。



1. Mark：

标记当前position位置，在执行reset方法后将pisition恢复到标记位置，mark的位置必须小于等于position，它们之间的关系：0<=mark<=position<=limit<=capacity

1. ByteBuffer实现

Java NIO中ByteBuffer有两种具体实现Direct ByteBuffer和Heap ByteBuffer，其中HeapByteBuffer也被认为是Non-Direct ByteBuffer。

|  |  |  |
| --- | --- | --- |
|  | DirectByteBuffer | HeapByteBuffer |
| 创建开销 | 大 | 小 |
| 存储位置 | Native heap | JVM heap  维护一个字节数组byte[] |
| 数据拷贝 | 无需临时缓冲区做拷贝 | 先拷贝到DirectByteBuffer类型的临时缓冲区，并且这个缓冲区具有缓存功能。 |
| GC影响 | 每次创建或者释放时都调用一次System.gc() | 无 |

1. Buffer操作方法

|  |  |
| --- | --- |
| 方法名称 | 作用 |
| allocate | 创建一个Heap ByteBuffer类型的缓冲区 |
| allocateDirect | 创建一个Direct ByteBuffer类型的缓冲区 |
| wrap | 通过外部传入一个数组创建Heap ByteBuffer类型的缓冲区 |
| flip | 切换读写操作模式 |
| put | 将单个字节写入缓冲区position位置 |
| get | 从缓冲区中读取单个字节 |
| mark | 标记当前position |
| reset | 恢复position到mark标记的位置 |
| rewind | 将position设回0，所以你可以重读Buffer中的所有数据。limit保持不变。 |
| clear | 清空缓存区，这里的清空并不是清除缓冲区数据，而是position被重置为0，limit被重置为capacity。 |
| compact | Compact与clear不同，compact是将所有未读的数据拷贝到Buffer起始处。然后将position设到最后一个未读数据下一位。limit属性依然像clear()方法一样设置成capacity。 |

###### Channel

###### Selector