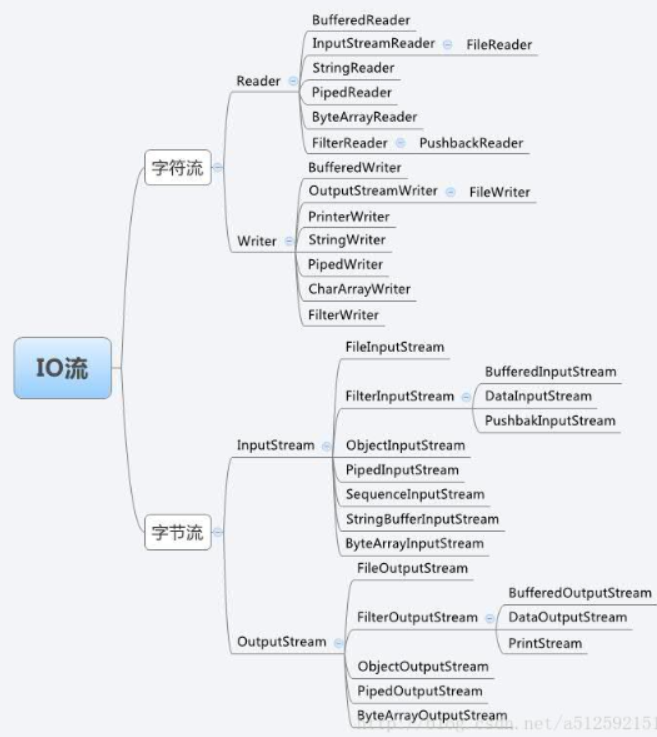
## IO 和NIO相关

I/O问题可以说是当今Web应用中所面临的主要问题之一，因为在当前这个海量数据时代，数据在网络中随处流动。在这个流动过程中都涉及到IO问题，可以说大部分Web应用系统的瓶颈都是IO瓶颈。

### JAVA中的IO基本架构

1. 基于字节操作的io接口: InputStream 和 OutputStream。
2. 基于字符操作的io接口:Writer和Reader。
3. 基于磁盘操作的io接口:File。
4. 基于网络操作的io接口:Socket。

1和2主要是数据传输的格式，后两组主要是数据传输的方式。

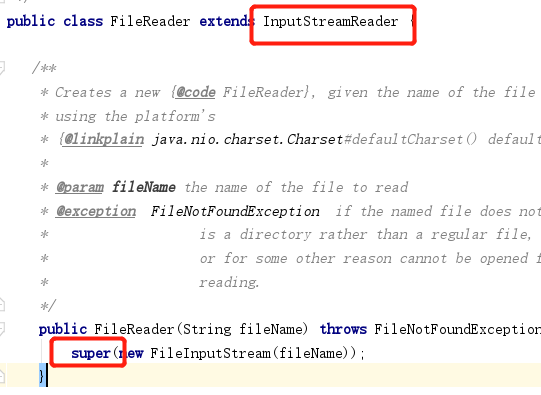
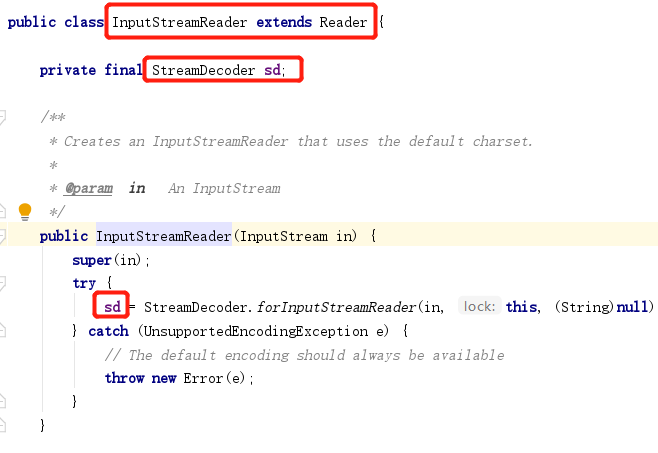


除此之外，还可以分为节点流(低级流)和高级流。

低级流：可以从或向一个特定的地方（节点）读写数据。如FileReader。

高级流：是对一个已存在的流的连接和封装，通过所封装的流的功能调用实现数据读写。如BufferedReader.处理流的构造方法总是要带一个其他的流对象做参数。一个流对象经过其他流的多次包装，称为流的链接。如：InputStreamReader OutputStreamReader BufferedInputStrean BufferedOutputStream BufferedReader BufferedWriter DataInputStream DataOutputStream。

\* 字节与字符的转化接口：数据持久化或网络传输都是以字节进行的，所以必须要有从字符到字节或从字节到字符的转化。InputStreamReader类是从字节到字符的转化桥梁，从Inputtream到Reader的过程要指定编码字符集，否则将采用系统默认的字符集，很可能出现乱码，StreamDecoder正式从你字节到字符的解码的实现类。例如，FileReader继承了InputStreamReader类，实际上是读取文件流，然后通过StreamDecoder解码成char。

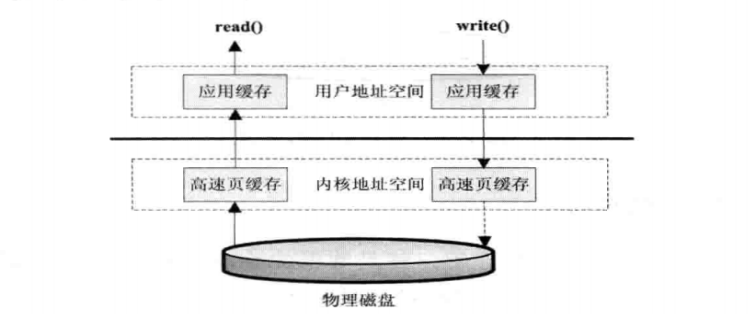
### 磁盘IO工作机制

应用程序要访问物理设备只能通过系统调用的方式来工作，读和写分别是read()和write()两个系统调用。而只要是系统调用就可能存在内核空间地址和用户空间地址切换的问题，这是操作系统为了保护系统本身的运行按权，而将内核程序运行使用的内存空间和用户程序运行的内存空间进行隔离造成的。这样保证了内存程序运行的安全性，但是也必然存在数据可能需要从内存空间想用户空间复制的问题。，为了加速io访问，在内核空间使用缓存机制，也就是将从磁盘读取的文件按照一定的组织方式进行缓存，如果用户访问的是同一段磁盘地址的空间数据，那么操作系统将从内核缓存中直接取出返回给用户程序，这样可以减少io响应时间。

#### 几种访问文件的方式：

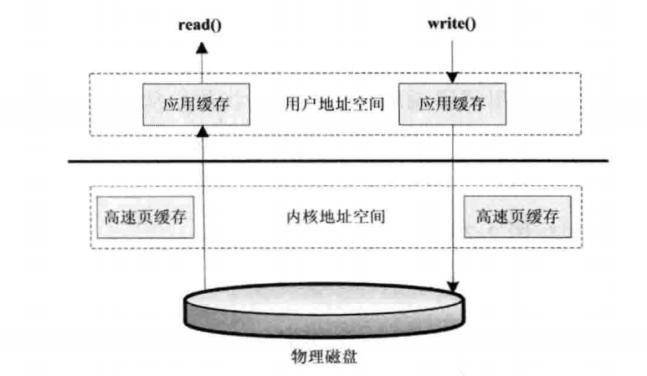
1. 标准访问文件的方式  
    当应用程序调用read()接口时，操作系统检查在内核的高速缓存中有没有需要的数据，如果已经缓存了，那么直接从你缓存中返回，如果没有，则从磁盘读取，然后缓存在操作系统的缓存中。

当应用程序调用write()接口将数据从用户地址空间复制到内核地址空间的缓存中。这时对用户程序来说，写操作就已经完成，至于什么时候写道磁盘由系统决定，除非显式调用sync同步命令。



1. 直接IO的方式，

直接访问磁盘，不经过内核数据缓冲区，减少从内核缓冲区到用户程序缓存的数据复制。



1. 同步访问文件的方式

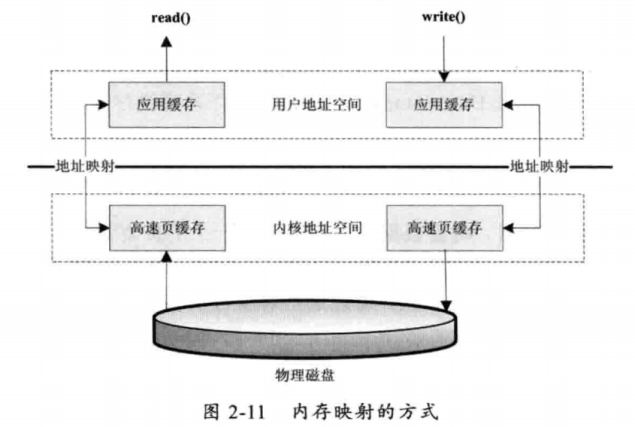
数据的读取和输入都是同步操作的，他与标准访问文件的方式不同，只有当数据被成功写到磁盘时才返回给应用程序成功的标志。此方式只在一些对数据安全性要求比较高的场景中才使用

1. 异步访问文件的方式

访问数据的线程发出请求之后，线程会接着处理其他事情，而不是阻塞等待，当请求的数据返回后继续处理下边的操作。可以明显提高应用程序的效率，不会改变访问文件的效率。

1. 内存映射的方式

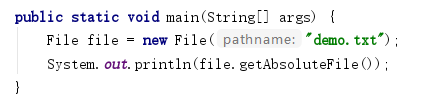
将内存中的某一块区域与磁盘中的文件关联起来，当要访问内存中的一段数据时，转换为访问文件的某一段数据。目的同样是减少数据从内核空间缓存到用户空间缓存的数据复制操作。



#### Java访问磁盘文件

数据在磁盘中的唯一最小描述就是文件，也就是说上层应用程序只能通过文件来操作磁盘上的数据，文件也是操作系统和磁盘驱动器交互的最小单元。在Java中通常的File并不代表一个真实存在的文件对象，创建File时，会返回一个代表这个路径的虚拟对象。

在真正读取这个文件时，才会检查这个文件存不存在。

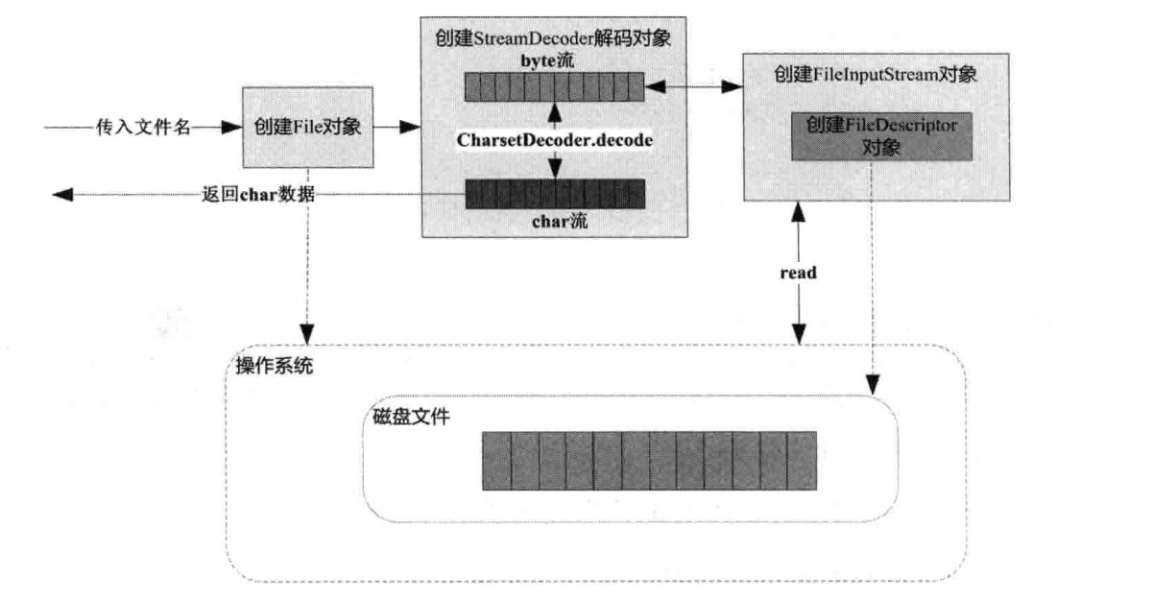


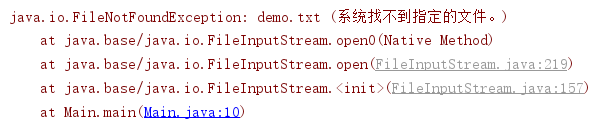
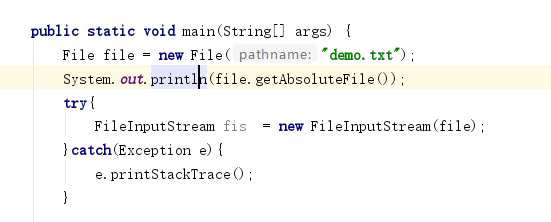
此时，并没有demo.txt文件，但仍能正常使用file.api。

只有在真正读取这个文件时，才会真正检查一个文件存不存在。例如FileInputStream类都是操作一个文件的接口，在创建一个FileInputStream对象时会创建一个FileDescriptor对象，这个对象就是真正代表一个存在的文件对象的描述。

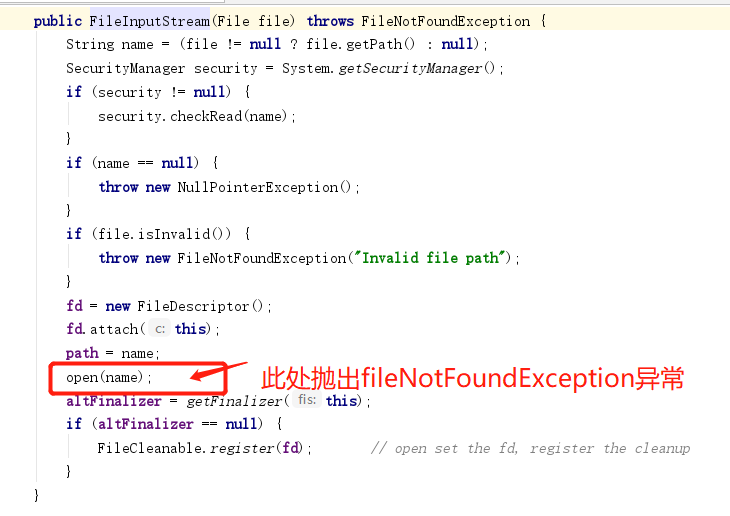
！现在以读取文件的程序为例，介绍如何从磁盘读取一段文本字符。

当传入一个文件路径时，将会根据这个路径创建一个File对象来表示这个文件，再根据File对象创建真正读取文件的操作对象，这是将会创建一个关联真是存在的磁盘文件的文件描述符FileDescriptor，通过这个对象可以直接控制这个磁盘文件，由于我们需要读取的时字符格式，所以需要StreamDecoder类将byte解码为char格式。





此时会报错。



#### Java序列化技术

Java序列化就是将一个对象转化成一串二进制表示的字节数组，通过保存或转移这些字节数据来达到持久化的目的。需要持久化，对象必须继承java.io.Serializable接口，反序列化则是相反的规则，将这个字节数组再重新构造成对象。序列化的数据并不像class文件那样保存类的完整的结构信息。

序列化一些总结：

1. 当父类继承Serializable接口时，所有子类都可以被序列化。
2. 子类实现了Serializable接口，父类没有，父类中的属性不能序列化(不报错数据会丢失)， 但是在子类中属性仍能正确序列化
3. 如果序列化的属性是对象，则这个对象也必须实现Serializable接口，否则会报错。
4. 在反序列化时，如果对象的属性有修改或删减，则修改的部分属性会丢失，但不会报错。
5. 在反序列化时，如果serialVersionUid被修改，则反序列化会失败。

### 网络IO工作机制

#### TCP状态转化

1. TCP建立链接需要3次握手，详情如下：

相关状态解释：

CLOSED：虚拟出来的状态，实际不存在，你在netstat -ant | grep 端口的时候是找不到的；

LISTEN：表示Server大门已开，随时准备有Client前来连线；

SYN\_SENT：只在Client端出现，表示Client发送过SYN了，正在焦急地等待Server的ACK；

SYN\_RCVD：只在Server端出现，表示Server收到Client的SYN了，并且已经发给Client自 己的ACK和SYN了，正在焦急地等待Client的ACK；

ESTABLISHED：在Client端出现表示Client把自己的ACK（第3次握手）发出去了，Client已 经就绪；在Server端出现表示Server已经收到Client的ACK（第3次握手）了，Server 已经就绪；

注：只有在Client和Server同时为ESTABLISHED时，即同时就绪时才可以进行数据传输。

Client端口状态转换：

1. 应用层调用connect，发送SYN到对端，等待对端的ACK和SYN；

2. 等待对端的ACK和SYN到来，接收到ACK和SYN后，发出自己的ACK，状态进入ESTABLISHED；

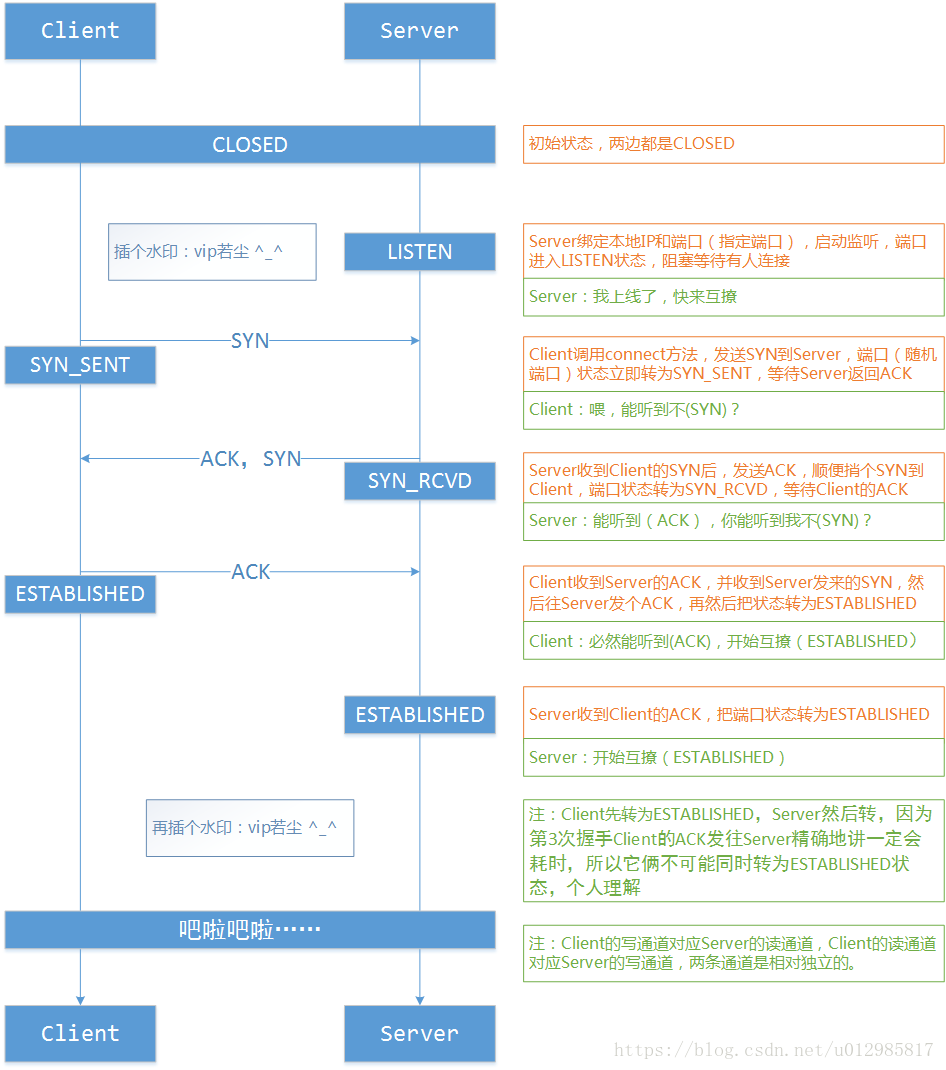
3. 等待对端的ACK和SYN期间，端口状态一直为SYN\_SENT，超时后进入CLOSED；

Server端口状态转化：

1. 应用层启动侦听，端口进入LISTEN状态；

2. 接收到Client发来的SYN，发送自己的ACK和SYN，进入SYN\_RCVD状态，等待Client的ACK；

3. 等待Client的ACK到来，接收到ACK后，进入ESTABLISHED；等待超时，进入CLOSED；



2 Tcp断开链接：

此过程需要进行4次挥手

状态解释：

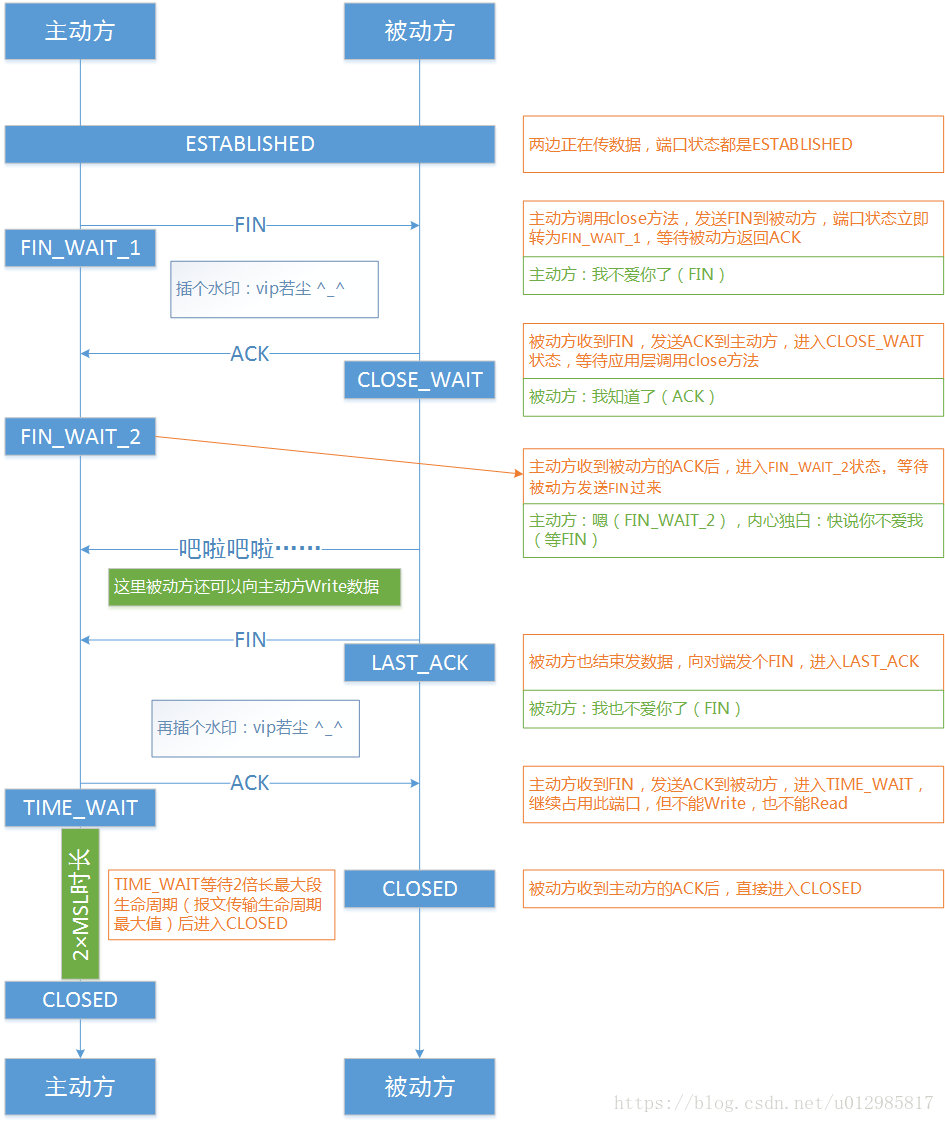
FIN\_WAIT\_1：仅出现在主动方，表示主动方想要断开连接，已经关闭了写通道，并向对端发送了FIN，等待对方的ACK到来

CLOSE\_WAIT：仅出现在被动方，表示被动方收到FIN后，已经回复ACK，正在等自己的应用层调用close方法关闭写通道，在CLOSE\_WAIT状态下，自己只能发送数据，但不能接收数据；

FIN\_WAIT\_2：仅出现在主动方，表示主动方已收到对端的ACK，等待对端的FIN，此时无法再发送数据，但是可以接收数据；

LAST\_ACK：仅出现在被动方，表示被动方缓冲区数据已经发送完毕，并且已经发送FIN到对端，等待对端的ACK；如果应用层写的比较垃圾，没有调用close关闭socket，则会一直停留在CLOSE\_WAIT；

TIME\_WAIT：仅出现在主动方，表示主动方已经发出ACK了，本次通讯完事了，双方都不能再读写了，但是主动方不确定对方能不能收到最后一个ACK，为了保证这个端口释放后，不被后来的连接马上使用被当成是新连接，通俗地讲，为了不乱套，这个状态会一直等待，等多久呢， 等2×MSL（Maximum Segment Lifetime）个时间，这个MSL是操作系统配置的，有默认参数，可以改。



注：TIME\_WAIT:这个状态比较复杂。有3个状态可以转化为此状态

1. 由FIN-WAIT-2转换到TIME-WAIT，具体情况时，在双方不同是发起FIN的情况下，主动关闭的一方完成在自身发起的关闭请求后，接受到被动关闭乙方的fin后进入的此状态。(正常情况)
2. 有一种ClOSING状态：两边同事发起关闭请求，由FIN-WAIT-1进入此状态，具体动作是接收到FIN请求，同事响应一个ACK。正常情况下，当你发送FIN报文后，按理来说是应该先收到（或同时收到）对方的ACK报文，再收到对方的FIN报文。但是CLOSING状态表示你发送FIN报文后，并没有收到对方的ACK报文，反而却也收到了对方的FIN报文。

从CLOSING转换到TIME-WAIT，具体情况是：在双方同时发起关闭，都做了发起FIN的请求，同时接收到了FIN并作了ACK的情况下。由CLOSING状态进入此状态。

1. 由FIN-WAIT-1转换到此状态，具体情况是：同时接受到FIN(对方发起)和ACK(本身发起的FIN回应)，它与CLOSING转换到TIME-WAIT的区别在于本身发起的FIN回应的ACK先于对方的FIN请求到达。而由CLOSING转化则是FIN先到达。

#### 影响网络传输的因素

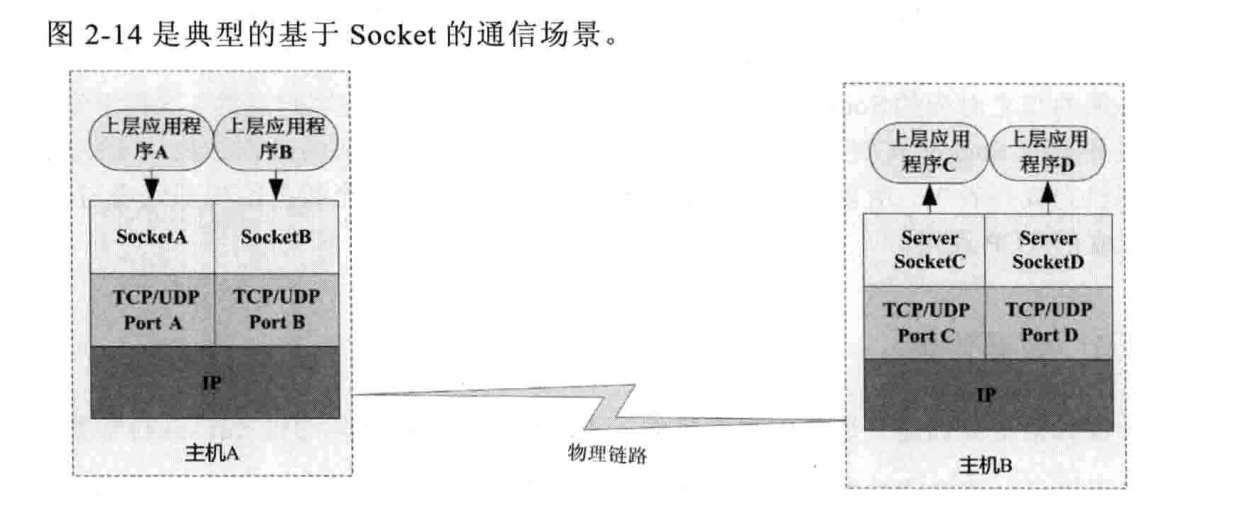
1.网络带宽:所谓带宽就是一条物理链路在1s内能够传输的最大比特数，注意这里是比特(bit

)而不是字节数，也就是b/s。网络带宽肯定是影响数据传输的一个关键环节。

1. 传输距离。
2. TCP拥塞控制，tcp传输是一个“停-等-停-等”的协议，传输方和接收方的步调要一致，要达到步调一致，就要通过拥塞控制来调节，tcp在传输时会设定一个”窗口”(BDP，Bandwidth Delay Product)，这个窗口的大小是由带宽和RTT（Round-Trip Time,数据在两端的来回时间，也就是响应时间）决定的，计算的公式是 带宽(b/s)\*RTT(s)。通过这个值可以得出理论上的最有TCP缓冲区的大小。

#### Java Socket的工作机制

Socket没有对应到具体的实体，他描述计算机之间完成相互通信的一种抽象功能。



主机A的应用程序要能和主机B的应用程序通信，必须通过Socket建立链接，而建立Socket连接必须由底层TCP/IP来建立TCP链接。建立TCP链接需要底层IP来寻址网络中的住居。我们知道网络层使用的IP可以帮助我们根据IP地址来找到目标主机，但是在一台主机上可能运行着多个应用程序，如何才能与指定的应用程序通信就要通过TCP或UDP的地址也就是端口号来指定，这样就可以通过一个Socket实力来唯一代表一个主机上的应用程序的通信链路了。

Socket建立通信链路：

当客户端要与服务端通信时，客户端首先要创建一个Socket实例，操作系统将为这个Socket实例分配一个没有被使用的本地端口号，并创建一个包含本地地址，远程地址和端口号的套接字数据结构，这个数据结构将一直保存在系统中直到这个连接关闭。在创建Socket实例的构造函数正确返回之前，将要进行TCP的3次握手协议，TCP握手协议完成后，Socket实例对象将完成，否则抛出IOException错误。与之对应的服务端将创建一个ServerSocket实例，创建ServerSocket比较简单，只要指定的端口号没有被占用，一般实例创建都会成功。同时操作系统也会为ServerSocket实例创建一个底层数据结构，在这个数据结构中包含指定舰艇的端口号和包含监听地址的通配符，通常情况下是”\*”, 即舰艇所有地址。之后当调用accept()方法时，将进入阻塞状态，等待客户端的请求。当一个新的请求到来时，将为这个链接创建一个新的套接字数据结构，该套接字数据的信息包含的地址和端口信息正是请求源地址和端口。这个新创建的数据结构将会关联到ServerSocket实例的一个未完成的连接数据结构列表中。注意，这时服务端的与之对应的Socket实例并没有完成创建，而要等到与客户端的3次握手完成后，这个服务端的Socket实例才会返回，并将这个Socket实例对应的数据结构从未完成的列表中移到已完成列表中。所以与ServerSocket所关联的列表中每个数据结构都代表与一个客户端建立的TCP连接。

通过Socket传输数据：

每个Socket实例都有一个InputStream和OutputStrem,并通过这两个对象来交换数据。同时我们也知道网络IO都是以字节流传输的，当创建Socket对象时，操作系统将会为InputStream和OutputStream分别分配一定大小的缓存区，数据的写入和读取都是通过这个缓存区完成的，写入端将数据写到OutputStream对应的SendQ队列中，如果这时RecvQ已经满了，那么OutputStream的write方法将会阻塞，直到RecvQ队列有足够的空间容纳SendQ发送的数据。特别值得注意的是，这个缓存区的大小写入端的速度和读取端的速度非常影响这个链接的传输效率，由于可能会发生阻塞，所以网络IO和磁盘IO不同的是数据的写入和读取还要有一个协调的过程，如果在两边同时传送数据可能会产生死锁。

#### Java中的IO模型

Io的两个阶段: 查看数据是否就绪； 进行数据拷贝（内核将数据拷贝到用户线程）

1. 同步阻塞IO（Blocking IO）：即传统的IO模型。
2. 同步非阻塞IO（Non-blocking IO）：默认创建的socket都是阻塞的，非阻塞IO要求socket被设置为NONBLOCK。注意这里所说的NIO并非Java的NIO（New IO）库。
3. 信号驱动IO模型：当用户线程发起一个IO请求操作，会给对应的socket注册一个信号函数，然后用户线程会继续执行，当内核数据就绪时会发送一个信号给用户线程，用户线程接收到信号之后，便在信号函数中调用IO读写操作来进行实际的IO请求操作。
4. IO多路复用（IO Multiplexing）：即经典的Reactor设计模式，Java中的Selector和Linux中的epoll都是这种模型,多路复用IO模型中，会有一个线程不断去轮询多个socket的状态，只有当socket真正有读写事件时，才真正调用实际的IO读写操作。因为在多路复用IO模型中，只需要使用一个线程就可以管理多个socket，系统不需要建立新的进程或者线程，也不必维护这些线程和进程，并且只有在真正有socket读写事件进行时，才会使用IO资源，所以它大大减少了资源占用,java NIO就是此模型。

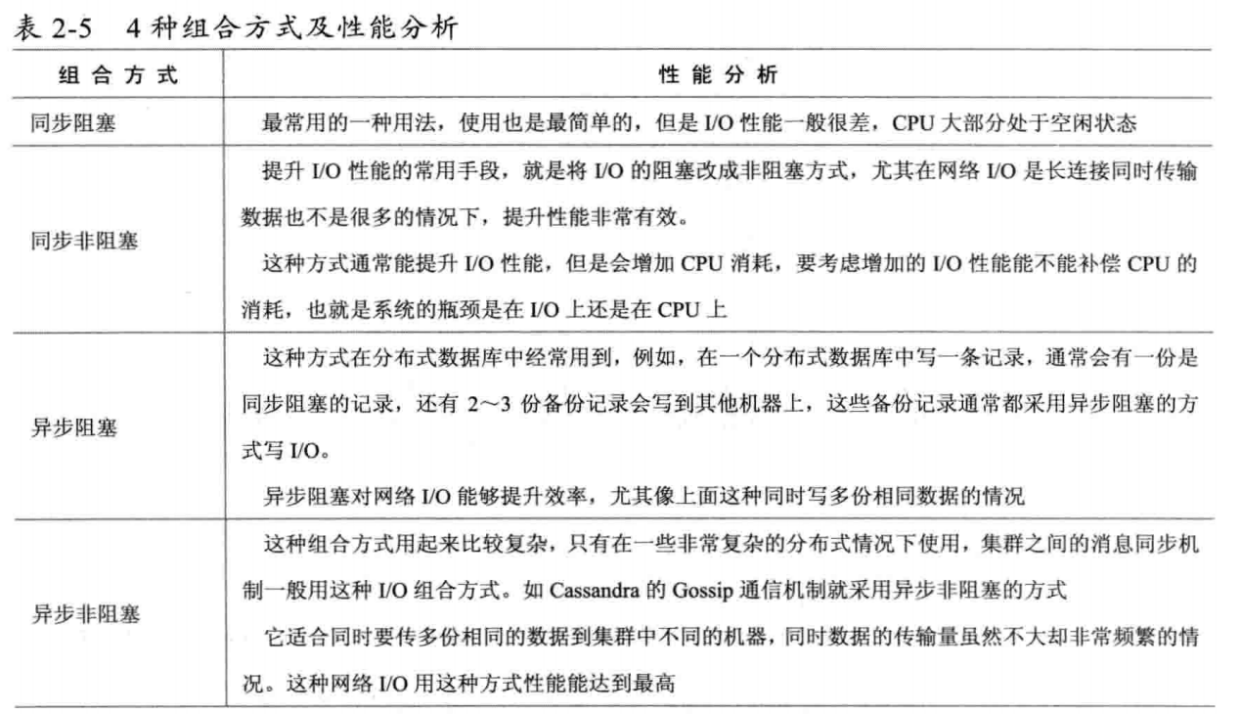
多线程+ 阻塞IO 达到类似的效果，但是由于在多线程 + 阻塞IO 中，每个socket对应一个线程，这样会造成很大的资源占用，并且尤其是对于长连接来说，线程的资源一直不会释放，如果后面陆续有很多连接的话，就会造成性能上的瓶颈。而多路复用IO模式，通过一个线程就可以管理多个socket，只有当socket真正有读写事件发生才会占用资源来进行实际的读写操作。因此，多路复用IO比较适合连接数比较多的情况。另外多路复用IO为何比非阻塞IO模型的效率高是因为在非阻塞IO中，不断地询问socket状态时通过用户线程去进行的，而在多路复用IO中，轮询每个socket状态是内核在进行的，这个效率要比用户线程要高的多。不过要注意的是，多路复用IO模型是通过轮询的方式来检测是否有事件到达，并且对到达的事件逐一进行响应。因此对于多路复用IO模型来说，一旦事件响应体很大，那么就会导致后续的事件迟迟得不到处理，并且会影响新的事件轮询。

1. 异步IO（Asynchronous IO）：即经典的Proactor设计模式，也称为异步非阻塞IO。异步IO模型才是最理想的IO模型，在异步IO模型中，当用户线程发起read操作之后，立刻就可以开始去做其它的事。而另一方面，从内核的角度，当它受到一个asynchronous read之后，它会立刻返回，说明read请求已经成功发起了，因此不会对用户线程产生任何block。然后，内核会等待数据准备完成，然后将数据拷贝到用户线程，当这一切都完成之后，内核会给用户线程发送一个信号，告诉它read操作完成了。也就说用户线程完全不需要实际的整个IO操作是如何进行的，只需要先发起一个请求，当接收内核返回的成功信号时表示IO操作已经完成，可以直接去使用数据了。也就说在异步IO模型中，IO操作的两个阶段都不会阻塞用户线程，这两个阶段都是由内核自动完成

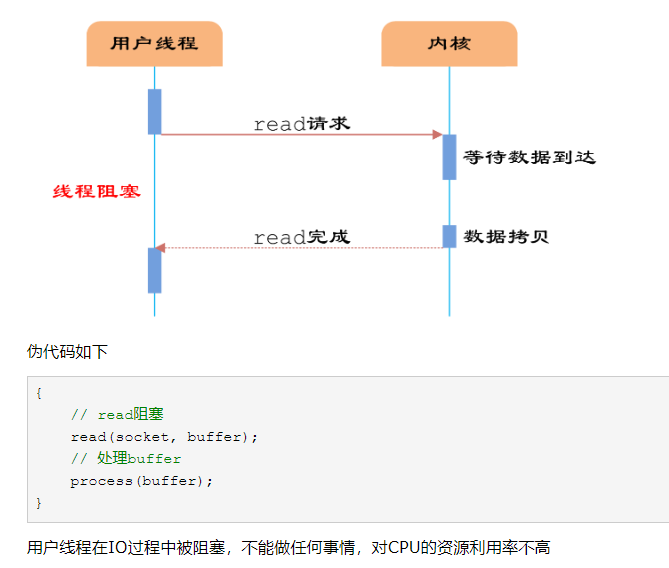
同步和异步的概念描述的是用户线程与内核的交互方式：同步是指用户线程发起IO请求后需要等待或者轮询内核IO操作完成后才能继续执行；而异步是指用户线程发起IO请求后仍继续执行，当内核IO操作完成后会通知用户线程，或者调用用户线程注册的回调函数。同步IO和异步IO的关键区别反映在数据拷贝阶段是由用户线程完成还是内核完成。所以说异步IO必须要有操作系统的底层支持。同步IO数据拷贝由用户线程完成，异步则为内核线程完成。

注意同步IO和异步IO与阻塞IO和非阻塞IO是不同的两组概念。

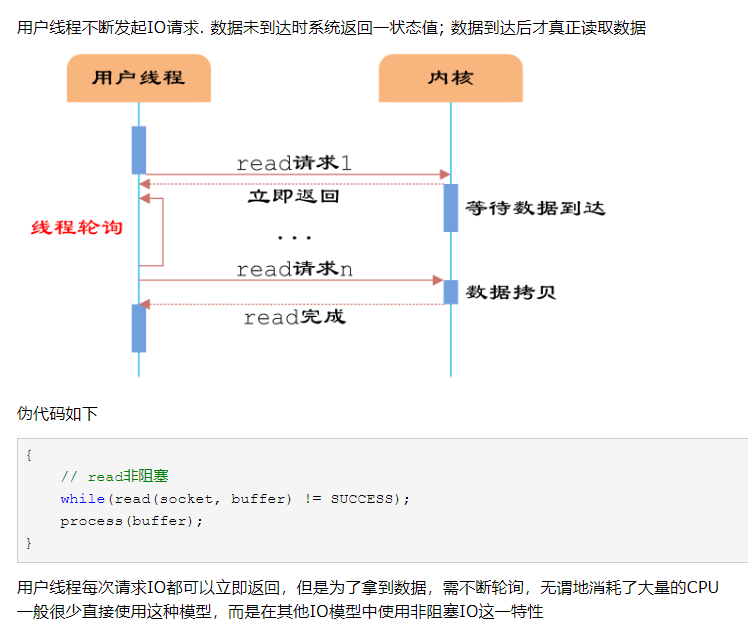
阻塞和非阻塞的概念描述的是用户线程调用内核IO操作的方式：阻塞是指IO操作需要彻底完成后才返回到用户空间；而非阻塞是指IO操作被调用后立即返回给用户一个状态值，无需等到IO操作彻底完成。阻塞与非阻塞主要是从CPU的消耗上来说的，阻塞就是CPU停下来等待一个慢的操作完成以后，CPU才接着完成其他工作。非阻塞就是在这个慢的操作执行时，CPU去做其他工作，等着这个慢的操作完成时，CP再接着完成后续的操作。阻塞IO和非阻塞IO是反映在IO操作的第一个阶段，在查看数据是否就绪时是如何处理的



1. 同步阻塞IO



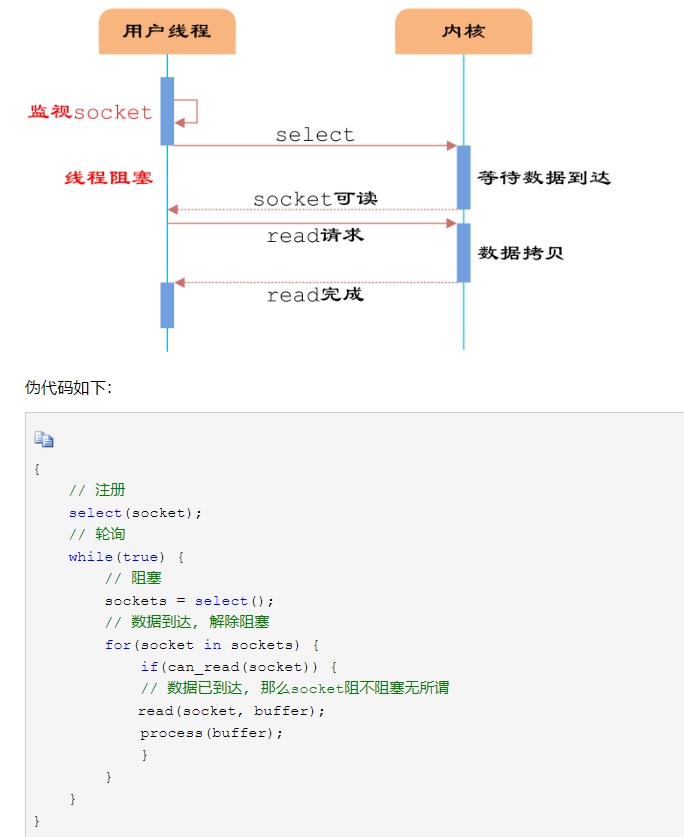
1. 同步非阻塞



1. IO多路复用

IO多路复用建立再内核提供的阻塞函数select上。

用户先将需要进行IO操作的socket添加到select中，然后等待阻塞函数select返回。当数据到达后，socket被激活，select返回，用户线程就能接着发起read请求。



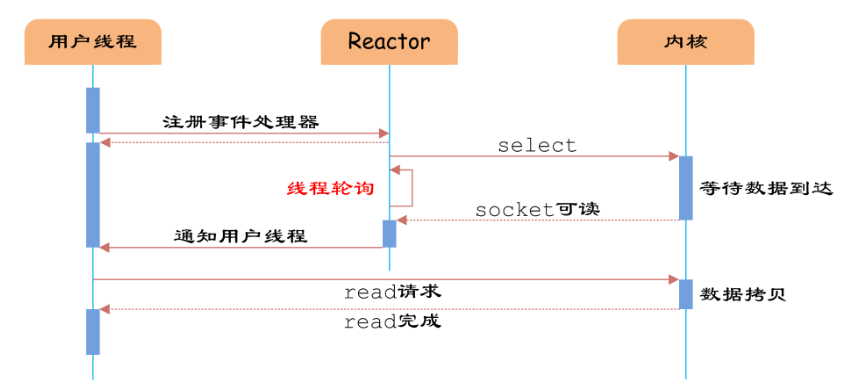
看起来和加了循环的同步阻塞IO差不多？

实际上, 我们可以给select注册多个socket, 然后不断调用select读取被激活的socket，实现在同一线程内同时处理多个IO请求的效果.

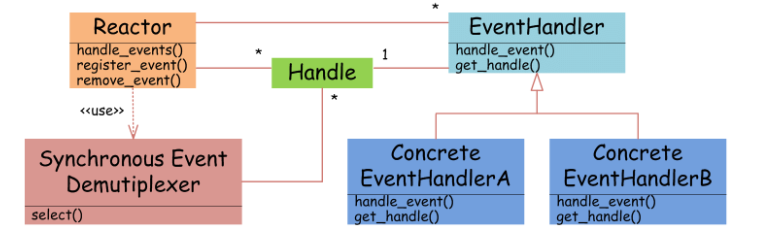
至此, 同步阻塞(阻塞在select) / 同步非阻塞(IO没有阻塞) {不知道该怎么称呼}完成

更进一步, 我们把select轮询抽出来放在一个线程里, 用户线程向其注册相关socket或IO请求，等到数据到达时通知用户线程，则可以提高用户线程的CPU利用率.

这样, 便实现了异步方式



这其实就是Reactor模式，如下图：



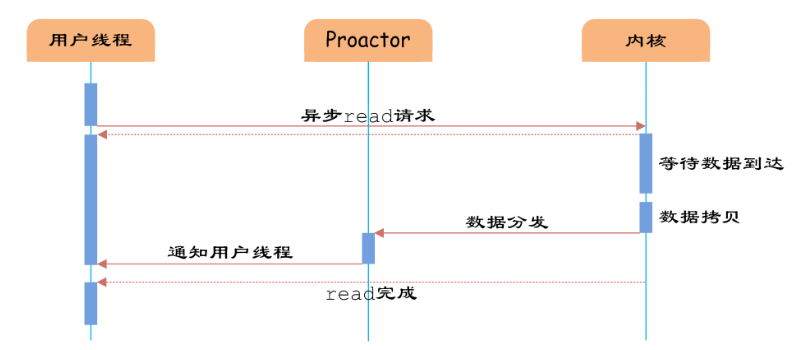
EventHandler抽象类表示IO事件处理器  
   get\_handle方法获得文件句柄Handle  
   handle\_event方法实现对Handle的操作  
 可继承EventHandler对事件处理器的行为进行定制

Reactor类管理EventHandler的注册、删除. handle\_events方法实现了事件循环, 其不断调用阻塞函数select, 只要某个文件句柄被激活（可读/写等），select就从阻塞中返回, handle\_events接着调用与文件句柄关联的事件处理器的handle\_event进行相关操作。handler\_events的伪代码如下

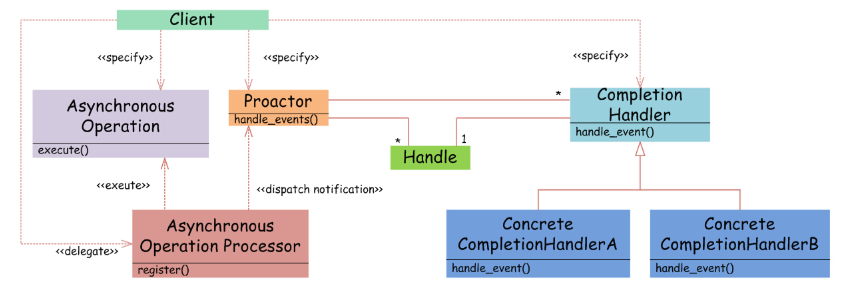


1. 异步IO

真正的异步IO需要操作系统更强的支持。  
IO多路复用模型中，数据到达内核后通知用户线程，用户线程负责从内核空间拷贝数据;  
而在异步IO模型中，当用户线程收到通知时，数据已经被操作系统从内核拷贝到用户指定的缓冲区内，用户线程直接使用即可。



异步IO模型使用了Proactor设计模式实现了这一机制。



Reactor模式中，用户线程向Reactor对象注册事件对应的事件处理器，然后事件触发时Reactor调用事件处理函数。  
Proactor模式中，用户线程将AsynchronousOperation（读/写等）、Proactor以及操作完成时的CompletionHandler注册到AsynchronousOperationProcessor。  
AsynchronousOperationProcessor使用Facade模式提供了一组异步API（读/写等）供用户调用. 当用户线程调用异步API后，便继续执行下一步代码. 而此时AsynchronousOperationProcessor会开启独立的内核线程执行异步操作。  
当read请求的数据到达时，由内核负责读取socket中的数据，并写入用户指定的缓冲区中。  
异步IO完成时，AsynchronousOperationProcessor将Proactor和CompletionHandler取出，并将IO操作结果和CompletionHandler分发给Proactor，Proactor通知用户线程(即回调先前注册的事件完成处理类的函数handle\_event)。  
Proactor一般被实现为单例，以便于集中分发操作完成事件。



相比于IO多路复用，异步IO并不常用，因为目前操作系统对异步IO的支持并不完善，IO多路复用也基本够用. 有很多做法是用IO多路复用模型模拟异步IO（IO事件触发时不直接通知用户线程，而是将数据读写完毕后放到用户指定的缓冲区中）。  
JDK7已经支持了AIO, netty采用过又放弃了, 据说是性能并没有多路复用好.

### NIO工作机制

Java.nio全称java non-blocking IO 为所有的原始类型（boolean类型除外）提供缓存支持的数据容器，使用它可以提供非阻塞式的高伸缩性网络

nio和bio的区别：本质就是阻塞与非阻塞的区别，Java IO 的各种流都是阻塞的，这意味着，当一个线程进行流处理(如read()和write())时，无论是否有数据，该线程会一直被阻塞，直到流通信结束。在此期间线程不能干其他的事情，就算当前没有数据，线程依然保持等待状态。这样无疑会浪费大量的资源。而在NIO的非阻塞模式下，线程发送数据与接收数据都是通过“通道”进行的，线程只需要去询问通道是否有数据需要处理，有则处理，无则立即返回不会进行等待。线程通常将非阻塞IO的空闲时间用于处理其他通道上的IO事件，使用一个单独的线程就可以管理多个输入和输出通道。

那么NIO是怎么实现非阻塞的呢？其实原理很简单，NIO是面向块的，先把数据搬运过来，存放到一个缓冲区中，线程过一段时间来缓冲区看看，有没有数据，这个样线程就不需要始终关注IO了。

（BIO为同步阻塞模型，NIO为同步非阻塞模型。NIO没有实现异步，在JDK1.7后，升级了NIO库包，支持异步非阻塞通信模型，即AIO）

#### NIO相关概念

1. Buffer: Buffer是一个对象，它用来存放即将发送的数据和即将到来的数据。Buffer是NIO的核心思想，他与普通流IO的区别是，普通流IO直接把数据写入或读取到Stream对象中，而NIO是先把读写数据交给Buffer，后再用流处理。Buffer实际上就是一个数组，通常是字节数组，这个数组提供了访问数据的读写等操作属性，如位置(position)，容量(capacity)，上限(limit)等概念。

Buffer类型：

ByteBuffer, CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer, LongBuffer, ShortBuffer

1. Channel

Channel(通道)，与Stream(流)的不用之处在于通道是双向的，流只能在一个方向上操作(一个流必须是InputStream或者OutputStream的子类)，而通道可以用于读，写或者二者同时进行，最关键的是可以和多路复用器结合起来，提供状态位，多路复用器可识别Channel所处的状态。

通道分为两大类:用于网络读写的SelectableChannel和用于文件操作的FileChannel。

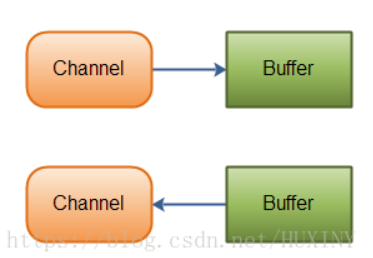
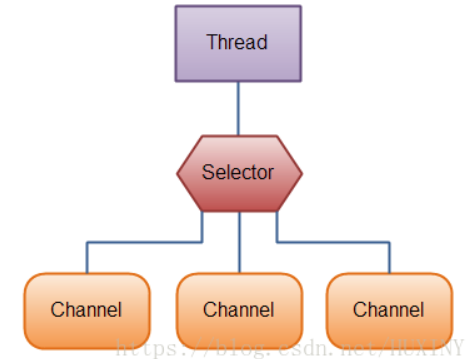
1. Selector

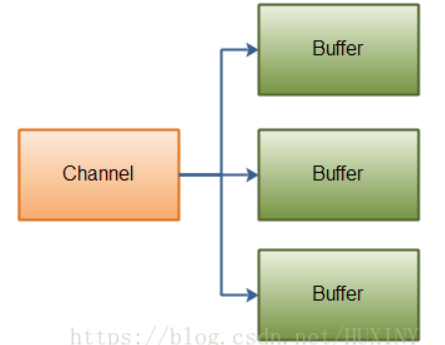
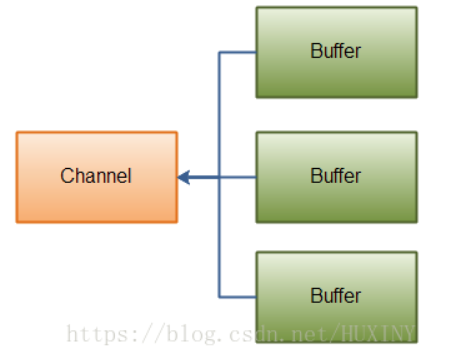
NIO的编程基础，Selector提供选择已经就绪的任务的能力，简单说，就是Selector会不断轮询注册再Selector上的通道(Channel)，如果这个通道发生了读写操作，这个通道就会处于就绪状态，会被Selector察觉到，然后通过SelectionKey可以取出就绪的Channel集合，从而进行IO操作。

一个Selector可以负责成千上万的通道，没有上限。这也是JDK使用了epoll代替传统的Select实现。获得链接句柄没有限制。意味着我们只需要一个线程负责Selector的轮询，就可以接

入成百上千的客户端。

原理图：

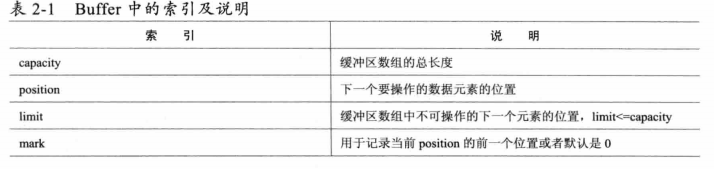
 

#### NIO使用demo详见StudyProject.nio包

#### Buffer工作方式

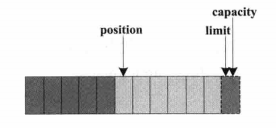
可以把Buffer简单的理解为一组基本数据类型的元素列表，他通过几个变量来保存这个数据的当前位置状态。



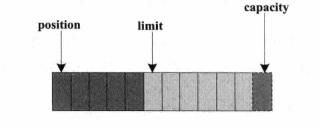
实际操作数据是他们的关系如下图：



我们通过ByteBuffer.allocate(11)方法创建了一个11个byte的数组缓冲区，初始状态如上图所示,position位置为0，capacity和limit默认都是数组长度。当我们写入5个字节时，位置变化如下图所示。

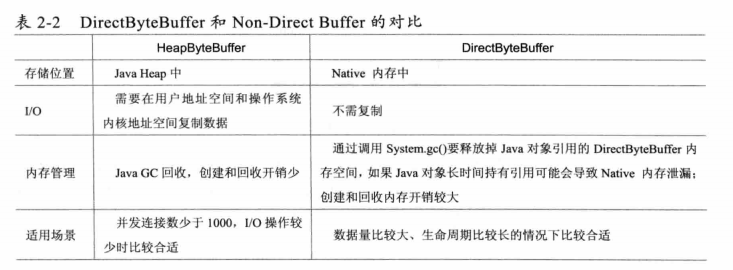


这是我们需要将缓冲区的5个字节数据写入Channel通信信道，所以我们调用byteBuffer.flip()方法，数组的状态发生如下图所示：



这时底层操作系统就可以从缓冲区中正确读取这5个字节并发送出去了。在下一次写数据之前我们再调一下clear()方法，缓冲区的索引状态又回到初始位置。

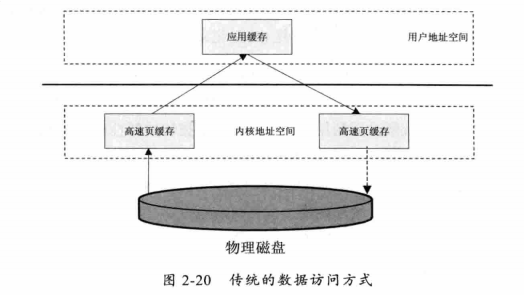
再说明一下mark,当我们调用mark方法时，他将记录当前position的前一个位置。当我们调用reset时，position将恢复mark记录下来的值。

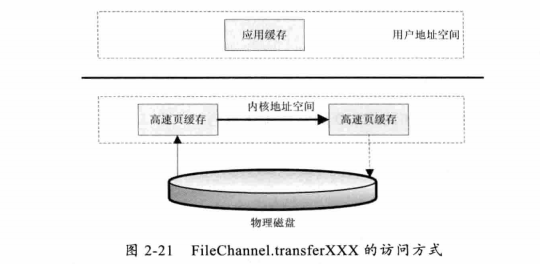
通过Channel获取的I/O数据首先要经过操作系统的Socket缓冲区，再将数据复制到Buffer中，这个操作系统缓冲区就是底层的TCP所关联的RecvQ或者SendQ队列，从操作系统缓冲区到用户缓冲区复制数据比较耗性能，Buffer提供了另外一种直接操作操作系统缓冲区的方式，即ByteBuffer.allocateDirector(size),这个方法返回的DirectByteBuffer就是与底层存储空间关联的缓冲区，它通过Native代码操作非JVM堆的内存空间，每次创建或者释放都调用一次System.gc()。注意，再使用DirectByteBuffer时可能会引起JVM内存泄漏问题。

#### NIO的数据访问方式

NIO提供了比传统文件访问方式更好的办法：FileChannel.transferTo,FileChannel.transferFrom,FileChannel.map

transFerXXX: 与传统的访问文件方式相比可以减少数据从内核到用户空间的复制，数据直接在内核空间中移动，在Linux中使用sendfile系统调用。





FileChannel.map:将文件按照一定大小块映射为内存区域，当程序访问这个内存区域时将直接操作这个文件数据，这种方式省去了数据从内核空间向用户空间复制的损耗。这种方式是和对大文件的只读性操作，如大文件的MD5校验。但这种方式是和操作系统的底层I/O实现相关的

### IO调优

#### 磁盘IO优化

1性能检测。我们的应用程序通常都需要访问磁盘来读取数据，而磁盘I/O通常都很耗时，要判断I/O是否是一个瓶颈，有一些参数指标可以参考。

我们可以压力测试应用程序，看系统的IOwait指标是否正常，例如，测试机器有四个CPU，那么理想的I/Owait参数应该不超过25%，如果超过25%，I/O很可能成为应用程序的新能瓶颈。在linux操作系统下可以通过iostat命令查看。

还有另外一个参数，就是IOPS，即要查看应用程序需要的最低IOPS是多少，磁盘的IOPS能不能达到要求。每个磁盘的IOPS通常在一个范围内，这和存储在磁盘上的数据块的大小和访问方式也有关，但主要是由磁盘的转速决定的，转速越高，IOPS也越高。

现在为了提升磁盘I/O的性能，通常采用一种叫RAID的技术，就是，将不同的磁盘组合起来以提高IO性能，目前有多种RAID技术，每种RAID技术对IO性能的提升会有不同，可以用一个RAID银子来代表。磁盘的读写吞吐量可以通过iostat命令来获取。于是可以计算出一个理论的IOPS值，计算公式如下：

（磁盘数\*每块磁盘的IOPS）/（磁盘读的吞吐量+RAID因子\*磁盘写的吞吐量）=IOPS

2提升IO性能。

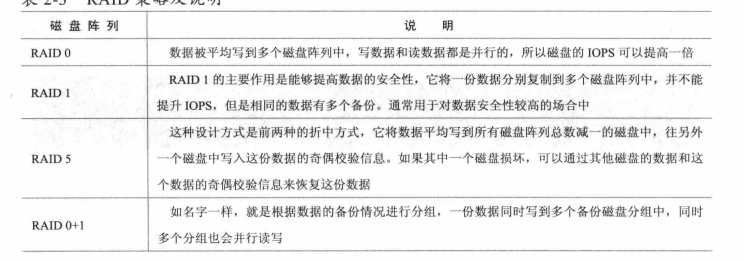
通常提升IO细嫩那个的方法有：

增加缓存，减少磁盘访问次数

优化磁盘的管理系统，设计最优的磁盘方式策略，以及磁盘的寻址策略，这是在底层操作系统层面考虑的。

设计合理的磁盘存储数据块，以及访问这些数据块的策略。这是在引用层面考虑的，例如我们可以给存放的数据设计索引，通过寻址索引来加快和减少磁盘的访问量，还可以采用异步和非阻塞的方式加快磁盘的访问速度。

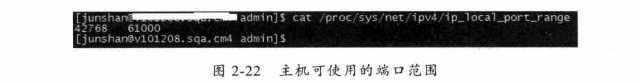
应用合理的RAID策略提升磁盘IO，RAID策略及说明如下图所示。



#### TCP网络参数调优

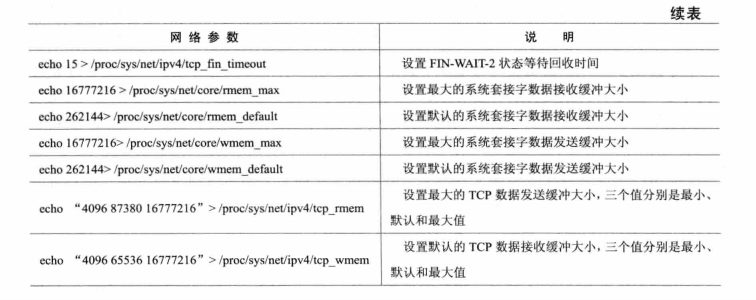
要能够建立一个TCP连接，必须知道对方的IP和一个未被使用的端口号，由于32位操作系统的端口号通常由2个字节表示，也就是至于2的16次方=65535个，所以一台主机能够同时建立的连接数是有限的，当然操作系统还有一些端口0-1024是受保护的，如80，22端口，这些端口都不能被随意占用。

在linux中可以通过查看/proc/sys/net/ipv4/ip\_local\_port\_range文件来知道当前这个主机可以使用的端口范围。

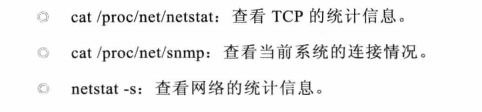


如果可以分配的端口号偏少，在遇到大量并发请求时就会成为瓶颈，由于端口有限导致大量请求等待建立连接，这样性能就会压不上去。另外如果发现有大量的TIME\_WAIT的话，可以设置/proc/sys/net/ipv4/tcp\_fin\_timeout为更小的值来快速释放请求。





注意，以上设置都是临时性的，系统重新启动后就会丢失，另外，Linux还提供了一些工具可用于查看当前的TCP统计信息，如下图



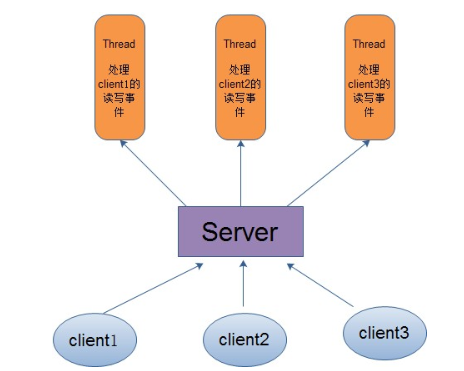
#### 网络IO优化

1. 减少网络交互的次数。需要在网络交互的两端设置缓存，如oracle的jdbc驱动程序就提供了对查询的sql结果的缓存，在客户端和数据库端都有，可以有效地减少对数据库的访问。还有合并访问请求。
2. 减少网络传输数据量的大小。通常是将数据压缩后再传输，如在HTTP请求中，通常Web服务器将请求的Web页面gzip压缩后再传给浏览器。还有就是通过设计简单的协议，尽量通过读取协议头来获取有用的价值信息，如在设计代理程序时，4层代理和7层代理都是在尽量避免要读取整个通信数据来取得需要的信息。
3. 尽量减少编码
4. 根据应用场景设计适合的交互方式。

### 相关IO设计模式

#### 传统

传统的互联网模式中，有两种比较经典的模式，一种是多线程，一种是线程池。对于多线程模式，也就说来了client，服务器就会新建一个线程来处理该client的读写时间，如下图所示。



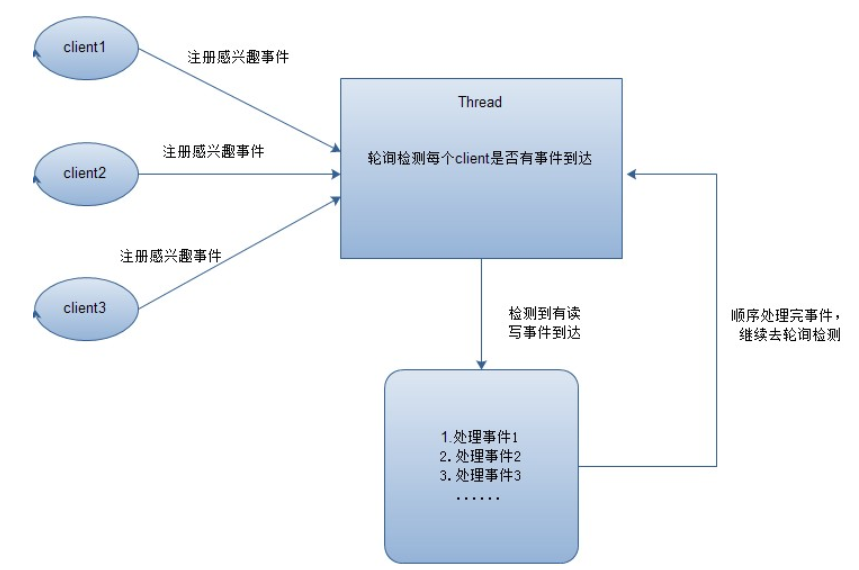
这种模式虽然处理起来简单方便，但是由于服务器为每个client的连接都采用一个线程去处理，使得资源占用非常大。因此，当连接数量达到上限时，再有用户请求连接，直接会导致资源瓶颈，严重的可能会直接导致服务器崩溃。

因此为了解决这种一个线程对应一个客户端模式带来的问题，提出了采用线程池的方式，也就说创建一个固定大小的线程池，来一个客户端，就从线程池取一个空闲线程来处理，当客户端处理完读写操作之后，就交出对线程的占用。因此这样就避免为每一个客户端都要创建线程带来的资源浪费，使得线程可以重用。

但是线程池也有它的弊端，如果连接大多是长连接，因此可能会导致在一段时间内，线程池中的线程都被占用，那么当再有用户请求连接时，由于没有可用的空闲线程来处理，就会导致客户端连接失败，从而影响用户体验。因此，线程池比较适合大量的短连接应用。

#### Reactor和Proactor

Reactor(反应器)模式中，会先对每个client注册感兴趣的事件，然后有一个线程专门去轮询每个client是否有事件发生，当有事件发生时，便顺序处理每个事件，当所有事件处理完之后，便再转去继续轮询。



在Proactor模式中，当检测到有事件发生时，会新起一个异步操作，然后交由内核线程去处理，当内核线程完成IO操作之后。发送一个通知告知操作已完成，异步IO模型采用的就是Proactor模式

### IO相关设计模式

适配器模式

装饰器模式

详见设计模式总结