**IO - 同步，异步，阻塞，非阻塞**

**实际上同步与异步是针对应用程序与内核的交互而言的。**

**同步过程中进程触发IO操作并等待(也就是我们说的阻塞)或者轮询的去查看IO操作(也就是我们说的非阻塞)是否完成。**

**异步过程中进程触发IO操作以后，直接返回，做自己的事情，IO交给内核来处理，完成后内核通知进程IO完成。**

**同步和异步针对应用程序来，关注的是程序中间的协作关系；**

**阻塞与非阻塞更关注的是单个进程的执行状态。**

**同步有阻塞和非阻塞之分，异步没有，它一定是非阻塞的。**

**阻塞、非阻塞、多路IO复用，都是同步IO，异步必定是非阻塞的，所以不存在异步阻塞和异步非阻塞的说法。**

**真正的异步IO需要CPU的深度参与。换句话说，只有用户线程在操作IO的时候根本不去考虑IO的执行全部都交给CPU去完成，而自己只等待一个完成信号的时候，才是真正的异步IO。**

**所以，拉一个子线程去轮询、去循环，或者使用select、poll、epool，都不是异步。**

**同步：执行一个操作之后，进程触发IO操作并等待(也就是我们说的阻塞)或者轮询的去查看IO操作(也就是我们说的非阻塞)是否完成，等待结果，然后才继续执行后续的操作。**

**异步：执行一个操作后，可以去执行其他的操作，然后等待通知再回来执行刚才没执行完的操作。**

**阻塞：进程给CPU传达一个任务之后，一直等待CPU处理完成，然后才执行后面的操作。**

**非阻塞：进程给CPU传达任我后，继续处理后续的操作，隔断时间再来询问之前的操作是否完成。这样的过程其实也叫轮询。**

同步（synchronous） IO和异步（asynchronous） IO，阻塞（blocking） IO和非阻塞（non-blocking）IO分别是什么，到底有什么区别？这个问题其实不同的人给出的答案都可能不同，比如wiki，就认为asynchronous IO和non-blocking IO是一个东西。这其实是因为不同的人的知识背景不同，并且在讨论这个问题的时候上下文(context)也不相同。

1. Stevens在文章中一共比较了五种IO Model：  
       blocking IO  
       non-blocking IO  
       IO multiplexing  
       signal driven IO  
       asynchronous IO  
   由于signal driven IO在实际中并不常用，所以我这只提及剩下的四种IO Model。  
     
   再说一下IO发生时涉及的对象和步骤。  
   对于一个network IO (这里我们以read举例)，它会涉及到两个系统对象，一个是**调用这个IO的process (or thread)，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：

**1. 等待数据准备 (Waiting for the data to be ready)。对于一个套接口上的操作，这一步骤关系到数据从网络到达，并将其复制到内核的某个缓冲区。**

1. **将数据从内核缓冲区复制到进程缓冲区（copying the data from the kernel to the process）。**记住这两点很重要，因为这些IO Model的区别就是在两个阶段上各有不同的情况。
2. 以下是用户级线程和内核级线程的区别：

（1）内核支持线程是OS内核可感知的，而用户级线程是OS内核不可感知的。

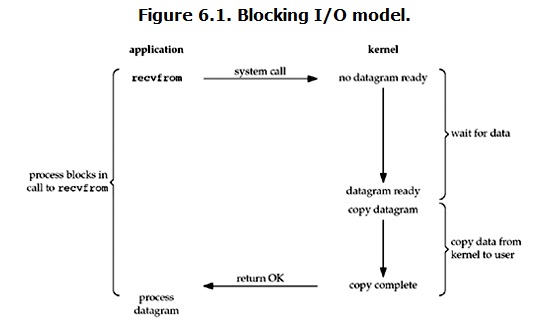
（2）**用户级线程的创建、撤消和调度不需要OS内核的支持，是在语言（如Java）这一级处理的；而内核支持线程的创建、撤消和调度都需OS内核提供支持，而且与进程的创建、撤消和调度大体是相同的。**

（3）用户级线程执行系统调用指令时将导致其所属进程被中断，而内核支持线程执行系统调用指令时，只导致该线程被中断。

（4）在**只有用户级线程的系统内，CPU调度还是以进程为单位**，处于运行状态的进程中的多个线程，由用户程序控制线程的轮换运行；在有**内核支持线程的系统内，CPU调度则以线程为单位**，由OS的线程调度程序负责线程的调度。

（5）用户级线程的程序实体是运行在用户态下的程序，而内核支持线程的程序实体则是可以运行在任何状态下的程序。

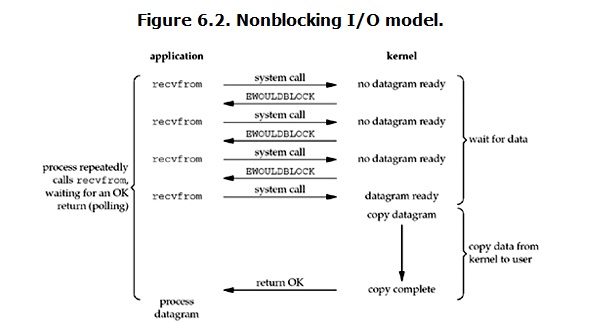
**blocking IO**   
在linux中，默认情况下所有的socket都是blocking，一个典型的读操作流程大概是这样：



当用户进程**调用了recvfrom这个系统**，kernel就开始了IO的第一个阶段：准备数据。对于network io来说，很多时候数据在一开始还没有到达（比如，还没有收到一个完整的UDP包），这个时候**kernel就要等待足够的数据到来**。而在用户进程这边，整个进程会被阻塞。当kernel一直等到数据准备好了，它就会将数据从kernel中拷贝到用户内存，然后kernel返回结果，用户进程才解除block的状态，重新运行起来。  
所以，blocking IO的特点就是在IO执行的两个阶段都被**block**了。

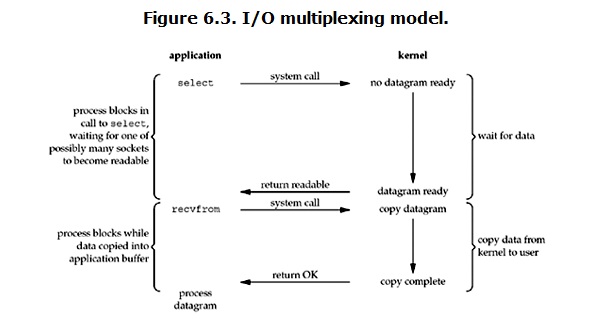
**non-blocking IO**

linux下，可以通过设置socket使其变为non-blocking。当对一个non-blocking socket执行读操作时，流程是这个样子：



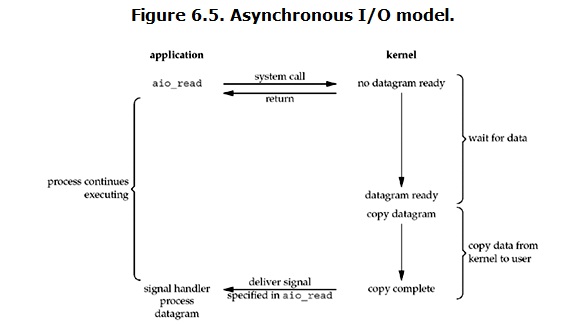
从图中可以看出，当用户进程发出read操作时，如果kernel中的数据还没有准备好，那么它并不会block用户进程，**而是立刻返回一个error。从用户进程角度讲 ，它发起一个read操作后，并不需要等待，而是马上就得到了一个结果。**用户进程判断结果是一个error时，它就知道数据还没有准备好，于是它可以再次发送read操作。一旦kernel中的数据准备好了，并且又再次收到了用户进程的system call，那么它马上就将数据拷贝到了用户内存，然后返回。  
所以，**用户进程其实是需要不断的主动询问kernel数据好了没有。**

**IO multiplexing（IO多路复用）**

**正因为阻塞I/O只能阻塞一个I/O操作，而I/O复用模型能够阻塞多个I/O操作，所以才叫做多路复用。**此模型用到select和poll函数，**这两个函数也会使进程阻塞**，select先阻塞，有活动套接字才返回，但是和阻塞I/O不同的是，这两个函数可以同时阻塞多个I/O操作**，而且可以同时对多个读操作，多个写操作的I/O函数进行检测，直到有数据可读或可写（**就是监听多个socket**）**。select被调用后，进程会被阻塞，内核监视所有select负责的socket，当有任何一个socket的数据准备好了，**select就会返回套接字可读，我们就可以调用recvfrom处理数据。**

**Asynchronous I/O**

linux下的asynchronous IO其实用得很少。先看一下它的流程：



**用户进程发起read操作之后，立刻就可以开始去做其它的事**。而另一方面，从kernel的角度，当它受到一个asynchronous read之后，首先它会立刻返回，所以不会对用户进程产生任何block。然后，**kernel会等待数据准备完成，然后将数据拷贝到用户内存（不用再叫用户进程来recvfrom处理数据了），当这一切都完成之后，kernel会给用户进程发送一个signal，告诉它read操作完成了。**

到目前为止，已经将四个IO Model都介绍完了。现在回过头来回答最初的那几个问题：

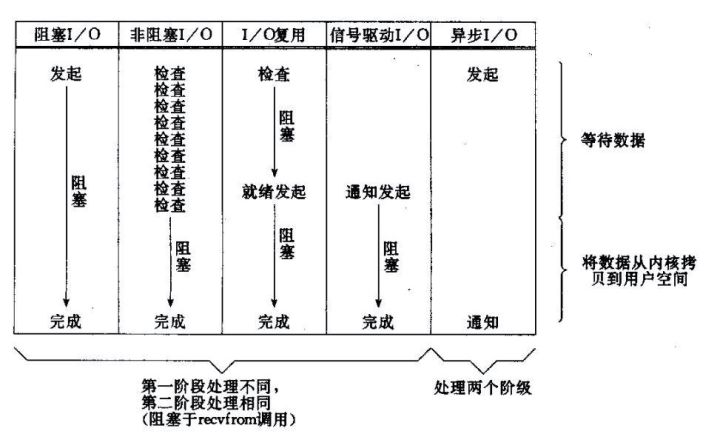
**blocking和non-blocking的区别在哪，**

**synchronous IO和asynchronous IO的区别在哪。**  
先回答最简单的这个：

blocking vs non-blocking:前面的介绍中其实已经很明确的说明了这两者的区别。**调用blocking IO会一直block住对应的进程直到操作完成，而non-blocking IO在kernel还没准备好数据的情况下会立刻返回。**

在说明synchronous IO和asynchronous IO的区别之前，需要先给出两者的定义。Stevens给出的定义（其实是POSIX的定义）是这样子的：（同步和异步的定义先给出）  
 两者的区别就在于synchronous IO（同步IO）做**”IO operation”**的时候会将process阻塞。按照这个定义，之前所述的blocking IO，non-blocking IO，IO multiplexing都属于synchronous IO。有人可能会说，non-blocking IO并没有被block啊。这里有个非常“狡猾”的地方，定义中所指的”IO operation”是指真实的IO操作，就是例子中的**recvfrom这个system call**。non-blocking IO在执行recvfrom这个system call的时候，如果kernel的数据没有准备好，这时候不会block进程。但是，当kernel中数据准备好的时候，recvfrom会将数据从kernel拷贝到用户内存中，这个时候进程是被block了，在这段时间内，进程是被block的**（到最后用户进程都是要通过系统回调来做这个操作的）**。而asynchronous IO则不一样，当进程发起IO 操作之后，就直接返回再也不理睬了，直到kernel发送一个信号，告诉进程说IO完成。在这整个过程中，进程完全没有被block。

各个IO Model的比较如图所示：



经过上面的介绍，会发现**non-blocking IO和asynchronous IO的区别还是很明显的。**在non-blocking IO中，**虽然进程大部分时间都不会被block，但是它仍然要求进程去主动的check，并且当数据准备完成以后，也需要进程主动的再次调用recvfrom来将数据拷贝到用户内存（此时就是同步了）。**

**而asynchronous IO则完全不同。它就像是用户进程将整个IO操作交给了他人（kernel）完成，然后他人做完后发信号通知。在此期间，用户进程不需要去检查IO操作的状态，也不需要主动的去拷贝数据（此时就是异步了）。**

**高性能IO模型浅析**

服务器端编程经常需要构造高性能的IO模型，常见的IO模型有四种：

（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](http://lib.csdn.net/base/javase" \o "Java SE知识库" \t "_blank)的NIO（New IO）库。

（3）IO多路复用（IO Multiplexing）：**即经典的Reactor设计模式，Java中的Selector和[Linux](http://lib.csdn.net/base/linux" \o "Linux知识库" \t "_blank)中的epoll都是这种模型。**

（4）异步IO（Asynchronous IO）：即经典的Proactor设计模式，也称为异步非阻塞IO。

为了方便描述，我们统一使用IO的读操作作为示例。

**一、同步阻塞IO**

同步阻塞IO模型是最简单的IO模型，用户线程在内核进行IO操作时被阻塞。

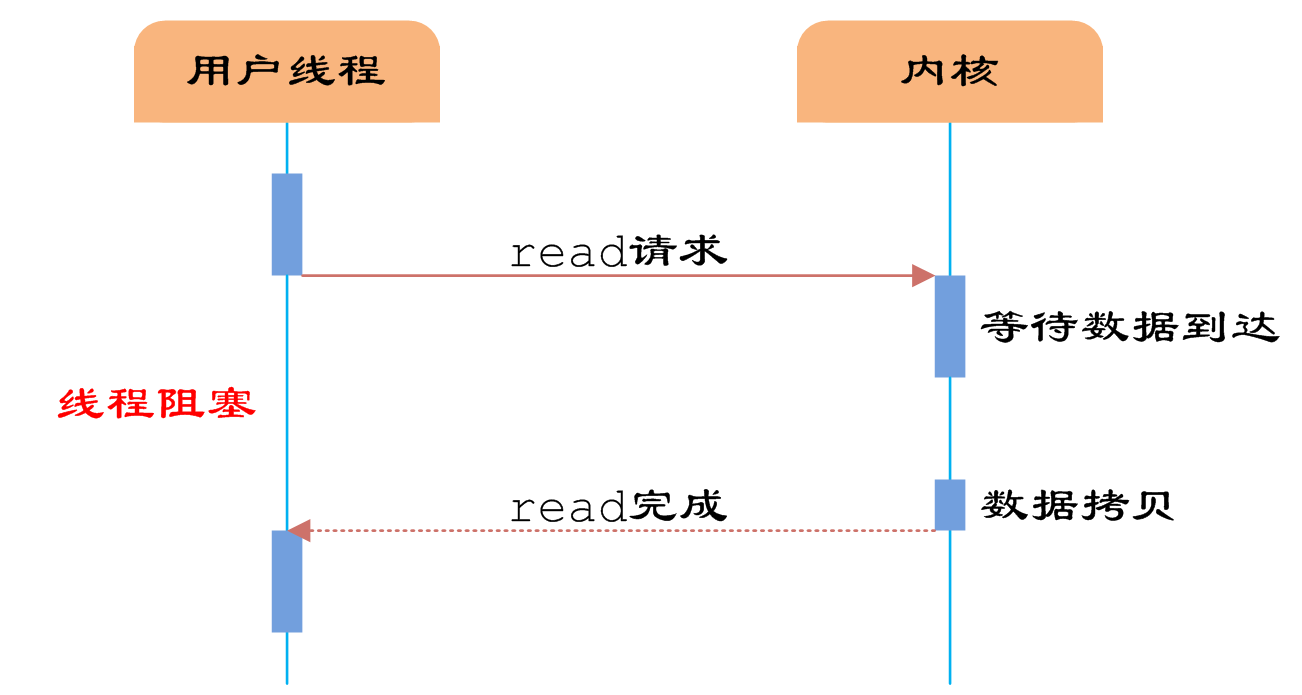


图1 同步阻塞IO

如图1所示，**用户线程通过系统调用read发起IO读操作，由用户空间转到内核空间。内核等到数据包到达后，然后将接收的数据拷贝到用户空间，完成read操作。**

用户线程使用同步阻塞IO模型的伪代码描述为：

[复制代码](javascript:void(0);)

{

read(socket, buffer);

process(buffer);

}

[复制代码](javascript:void(0);)

即用户需要等待read将socket中的数据读取到buffer后，才继续处理接收的数据。整个IO请求的过程中，用户线程是被阻塞的，这导致用户在发起IO请求时，不能做任何事情，对CPU的资源利用率不够。

**二、同步非阻塞IO**

同步非阻塞IO是在同步阻塞IO的基础上，将socket设置为NONBLOCK。这样做用户线程可以在发起IO请求后可以立即返回。

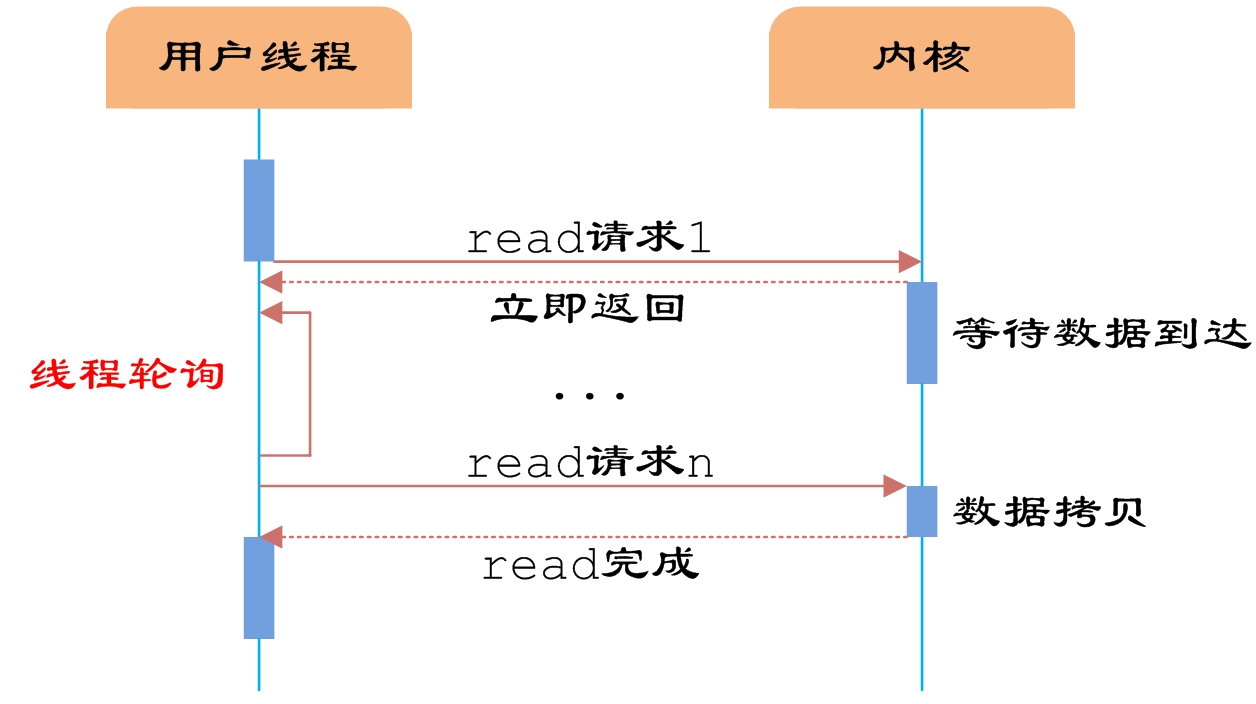


图2 同步非阻塞IO

如图2所示，由于socket是非阻塞的方式，因此用户线程发起IO请求时立即返回。但并未读取到任何数据，用户线程需要不断地发起IO请求，直到数据到达后，才真正读取到数据，继续执行。

用户线程使用同步非阻塞IO模型的伪代码描述为：

[复制代码](javascript:void(0);)

{

while(read(socket, buffer) != SUCCESS)

;

process(buffer);

}

[复制代码](javascript:void(0);)

即用户需要不断地调用read，尝试读取socket中的数据，直到读取成功后，才继续处理接收的数据。整个IO请求的过程中，虽然用户线程每次发起IO请求后可以立即返回，但是为了等到数据，仍需要不断地轮询、重复请求，消耗了大量的CPU的资源。一般很少直接使用这种模型，而是在其他IO模型中使用非阻塞IO这一特性。

**三、IO多路复用**

IO多路复用模型是建立在内核提供的**多路分离函数select基础之上的**，使用select函数可以避免同步非阻塞IO模型中轮询等待的问题。



图3 多路分离函数select

如图3所示，用户首先将需要进行IO操作的socket添加到select中，然后阻塞等待select系统调用返回。当数据到达时，socket被激活，select函数返回。用户线程正式发起read请求，读取数据并继续执行。

从流程上来看，使用select函数进行IO请求和同步阻塞模型没有太大的区别，甚至还多了添加监视socket，以及调用select函数的额外操作，效率更差。但是，**使用select以后最大的优势是用户可以在一个线程内同时处理多个socket的IO请求。用户可以注册多个socket，然后不断地调用select读取被激活的socket，即可达到在同一个线程内同时处理多个IO请求的目的。**而在同步阻塞模型中，必须通过多线程的方式才能达到这个目的。

用户线程使用select函数的伪代码描述为：

[复制代码](javascript:void(0);)

{

select(socket);

while(1) {

sockets = select();

for(socket in sockets) {

if(can\_read(socket)) {

read(socket, buffer);

process(buffer);

}

}

}

}

[复制代码](javascript:void(0);)

其中while循环前将socket添加到select监视中，然后在while内一直调用select获取被激活的socket，一旦socket可读，便调用read函数将socket中的数据读取出来。

  然而，使用select函数的优点并不仅限于此。虽然上述方式允许单线程内处理多个IO请求，但是每个IO请求的过程还是阻塞的**（在select函数上阻塞）**，平均时间甚至比同步阻塞IO模型还要长。

**如果用户线程只注册自己感兴趣的socket或者IO请求，然后去做自己的事情，等到数据到来时再进行处理，则可以提高CPU的利用率。**

IO多路复用模型使用了**Reactor(加入个事件分发器)**设计模式实现了这一机制。

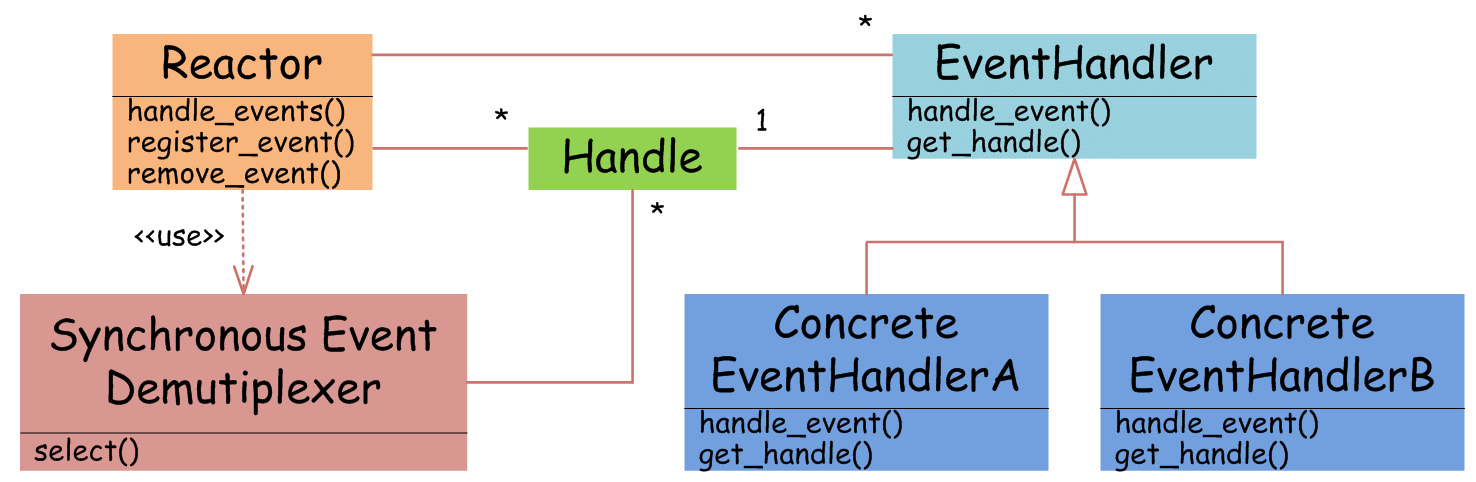


图4 Reactor设计模式

如图4所示，

EventHandler抽象类（接口）表示IO事件处理器，它拥有IO文件句柄Handle（通过get\_handle获取），以及对Handle的操作handle\_event（读/写等）。

继承于EventHandler的子类可以对事件处理器的行为进行定制。

**Reactor类**用于管理EventHandler（注册、删除等），并使用handle\_events实现事件循环，不断调用同步事件多路分离器**（一般是内核）**的多路分离函数select，只要某个文件句柄被激活（可读/写等），select就返回（阻塞），handle\_events就会调用与文件句柄关联的事件处理器的handle\_event进行相关操作。

**Synchronous Event Demultiplexer** ：**同步事件分离器，阻塞等待Handles中的事件发生**



图5 IO多路复用

如图5所示，通过Reactor的方式，可以将用户线程轮询IO操作状态的工作统一交给handle\_events事件循环进行处理。**用户线程注册事件处理器之后可以继续执行做其他的工作（异步）**，而Reactor线程负责**调用内核（事件分离器）的select函数检查socket状态**。当有socket被激活时，**则通知相应的用户线程（或执行用户线程的回调函数），执行handle\_event进行数据读取、处理的工作。**

由于select函数是阻塞的，因此多路IO复用模型也被称为异步阻塞IO模型。注意，这里的所说的阻塞是指select函数执行时线程被阻塞，而不是指socket。

**一般在使用IO多路复用模型时，socket都是设置为NON-BLOCK的，不过这并不会产生影响，因为用户发起IO请求时，数据已经到达了，用户线程一定不会被阻塞。**

用户线程使用IO多路复用模型的伪代码描述为：

[复制代码](javascript:void(0);)

void UserEventHandler::handle\_event() {

if(can\_read(socket)) {

read(socket, buffer);

process(buffer);

}

}

{

Reactor.register(new UserEventHandler(socket));

}

[复制代码](javascript:void(0);)

用户需要重写EventHandler的handle\_event函数进行读取数据、处理数据的工作，用户线程只需要将自己的EventHandler注册到Reactor即可。Reactor中handle\_events事件循环的伪代码大致如下。

[复制代码](javascript:void(0);)

Reactor::handle\_events() {

while(1) {

sockets = select();

for(socket in sockets) {

get\_event\_handler(socket).handle\_event();

}

}

}

[复制代码](javascript:void(0);)

事件循环不断地调用select获取被激活的socket，然后根据获取socket对应的EventHandler，执行器handle\_event函数即可。

IO多路复用是最常使用的IO模型，但是其异步程度还不够“彻底”，因为它**使用了会阻塞线程的select系统调用**。因此IO多路复用只能称为异步阻塞IO，而非真正的异步IO。

**四、异步IO**

“真正”的异步IO需要[操作系统](http://lib.csdn.net/base/operatingsystem" \o "操作系统知识库" \t "_blank)更强的支持。

在IO多路复用模型中，**事件循环将文件句柄的状态事件通知给用户线程，由用户线程自行读取数据、处理数据。**

而在异步IO模型中，当用户线程收到通知时，**数据已经被内核读取完毕，并放在了用户线程指定的缓冲区内，内核在IO完成后通知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。

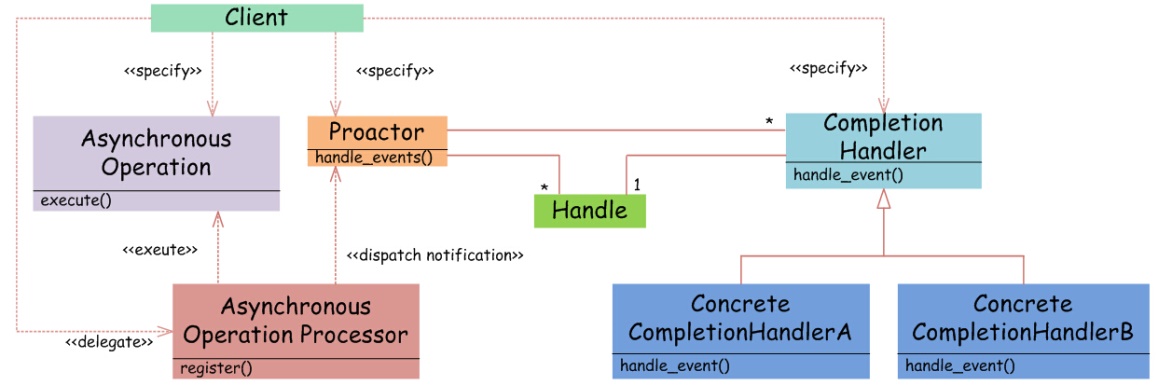


图6 Proactor设计模式

如图6，Proactor模式和Reactor模式在结构上比较相似，不过在用户（Client）使用方式上差别较大。

Reactor模式中，**用户线程通过向Reactor对象注册感兴趣的事件监听，然后事件触发时调用事件处理函数。**

而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操作API（读/写等）供用户使用，当用户线程调用异步API后，便继续执行自己的任务。AsynchronousOperationProcessor 会开启独立的内核线程执行异步操作，实现真正的异步。当异步IO操作完成时，AsynchronousOperationProcessor将用户线程与AsynchronousOperation一起注册的Proactor和CompletionHandler取出，然后将CompletionHandler与IO操作的结果数据一起转发给Proactor，**Proactor负责回调每一个异步操作的事件完成处理函数handle\_event**。虽然Proactor模式中每个异步操作都可以绑定一个Proactor对象，但是一般在操作系统中，Proactor被实现为Singleton模式，以便于集中化分发操作完成事件。

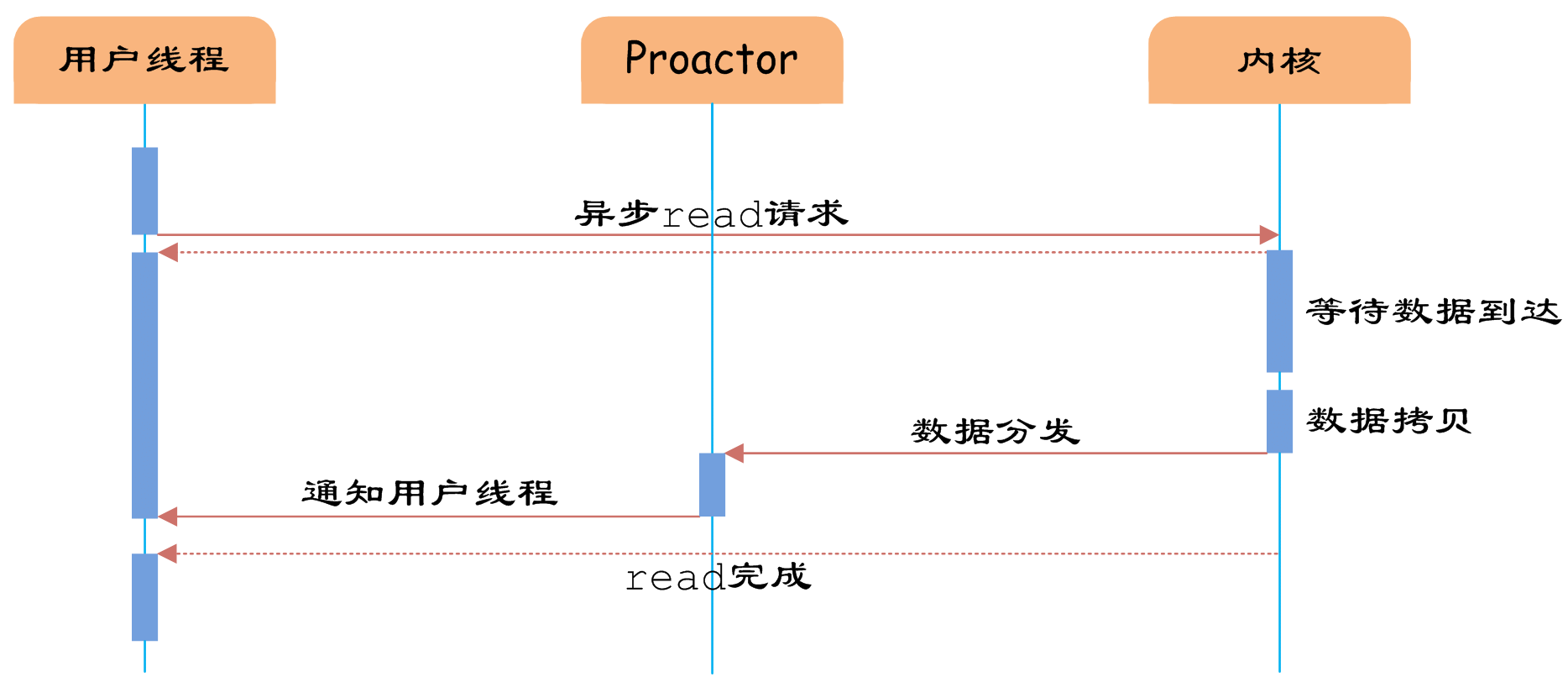


图7 异步IO

如图7所示，异步IO模型中，用户线程直接使用内核提供的异步IO API发起read请求，且发起后立即返回，继续执行用户线程代码。不过此时**用户线程已经将调用的AsynchronousOperation和CompletionHandler注册到内核**，

然后**操作系统开启独立的内核线程去处理IO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。最后内核将read的数据和用户线程注册的CompletionHandler分发给内部Proactor，Proactor将IO完成的信息通知给用户线程（一般通过调用用户线程注册的完成事件处理函数），完成异步IO。

用户线程使用异步IO模型的伪代码描述为：

[复制代码](javascript:void(0);)

void UserCompletionHandler::handle\_event(buffer) {

process(buffer);

}

{

aio\_read(socket, new UserCompletionHandler);

}

[复制代码](javascript:void(0);)

用户需要重写CompletionHandler的handle\_event函数进行处理数据的工作，参数buffer表示Proactor已经准备好的数据，用户线程直接调用内核提供的异步IO API，并将重写的CompletionHandler注册即可。

相比于IO多路复用模型，异步IO并不十分常用，**不少高性能并发服务程序使用IO多路复用模型+多线程任务处理的**[**架构**](http://lib.csdn.net/base/architecture)**基本可以满足需求。**况且目前操作系统对异步IO的支持并非特别完善，更多的是采用IO多路复用模型模拟异步IO的方式（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）。Java7之后已经支持了异步IO，感兴趣的读者可以尝试使用。

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。  
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，**

**阻塞IO和非阻塞IO的区别在于第一步，**

**发起IO请求线程是否会被阻塞，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。  
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。**

**当一个read操作发生时，会经过如下流程：  
通过read系统调用向内核发起读请求。  
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IO和异步IO是针对应用程序和内核的交互而言的。  
同步IO指的是**用户进程触发I/O操作并等待或者轮询的去查看I/O操作是否就绪。**  
异步IO是指用户进程触发I/O操作以后就立即返回，继续开始做自己的事情，而当I/O操作已经完成的时候会得到I/O完成的通知。

同步IO在**读取数据时，都要在同步流程中进行一次内核态到用户态的数据拷贝。**  
异步IO需要操作系统更强的支持，操作系统开启独立的内核线程去处理IO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，最后内核通知用户线程。**异步IO接受到内核通知时，已经完成了内核态到用户态的数据拷贝。**

2.1、阻塞I/O

应用程序调用一个IO函数，导致应用程序阻塞，等待数据准备好。 如果数据没有准备好，一直等待….数据准备好了，从内核拷贝到用户空间,IO函数返回成功指示。

**阻塞I/O模型图：**在调用recv()/recvfrom（）函数时，发生在内核中等待数据和复制数据的过程。

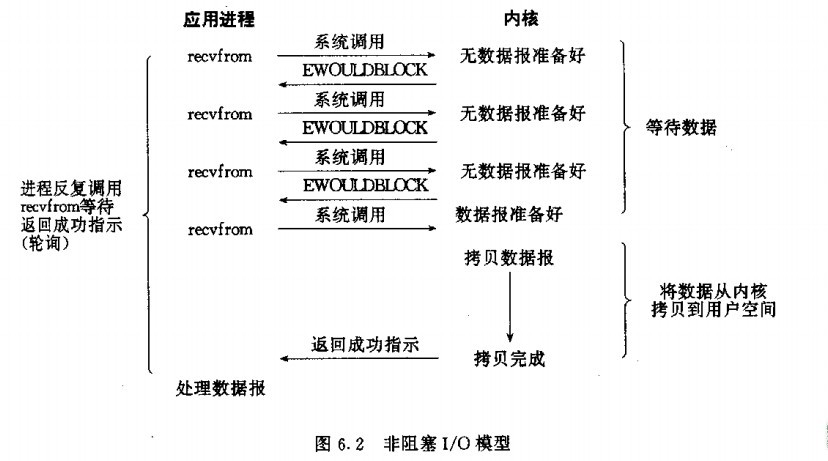


当调用recv()函数时，系统首先查是否有准备好的数据。如果数据没有准备好，那么系统就处于等待状态。当数据准备好后，将数据从系统缓冲区复制到用户空间，然后该函数返回。在套接应用程序中，当调用recv()函数时，未必用户空间就已经存在数据，那么此时recv()函数就会处于等待状态。

2.2、非阻塞I/O

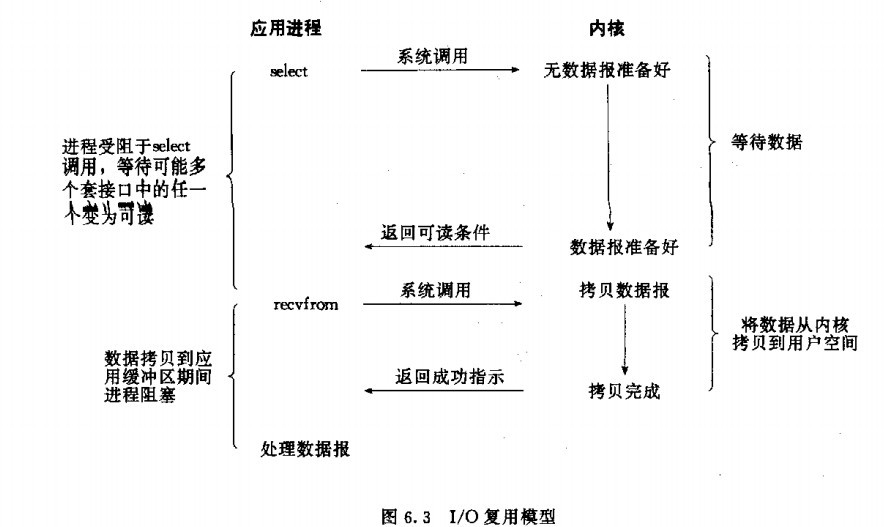
非阻塞IO通过进程反复调用IO函数（多次系统调用，并马上返回）；在数据拷贝的过程中，进程是阻塞的

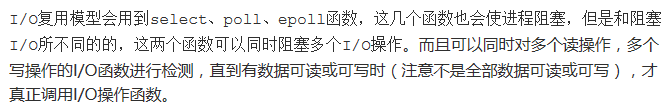
我们把一个SOCKET接口设置为非阻塞就是告诉内核，当所请求的I/O操作无法完成时，不要将进程睡眠，而是返回一个错误。这样我们的I/O操作函数将不断的测试数据是否已经准备好，如果没有准备好，继续测试，直到数据准备好为止。在这个不断测试的过程中，会大量的占用CPU的时间。



2.3、IO复用

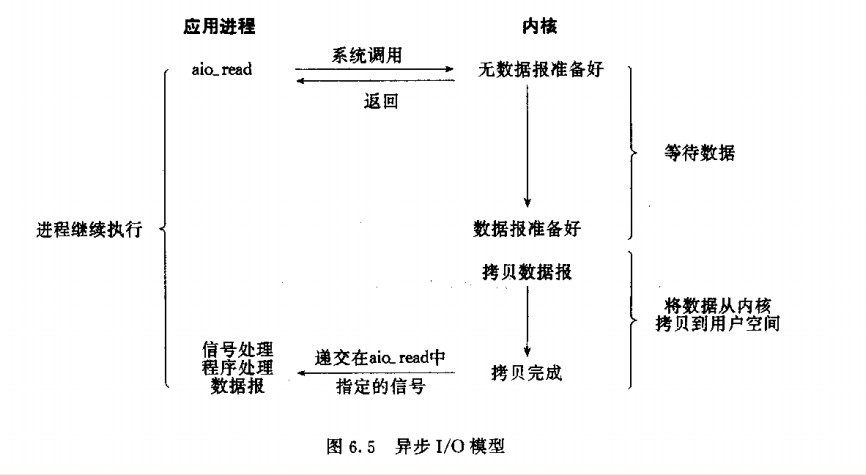
主要是select和epoll；对一个IO端口，两次调用，两次返回，比阻塞IO并没有什么优越性；关键是能实现同时对多个IO端口进行监听； I/O复用模型会用到select、poll、epoll函数，这几个函数也会使进程阻塞，但是和阻塞I/O所不同的的，这两个函数可以同时阻塞多个I/O操作。而且可以同时对多个读操作，多个写操作的I/O函数进行检测，直到有数据可读或可写时，才真正调用I/O操作函数。



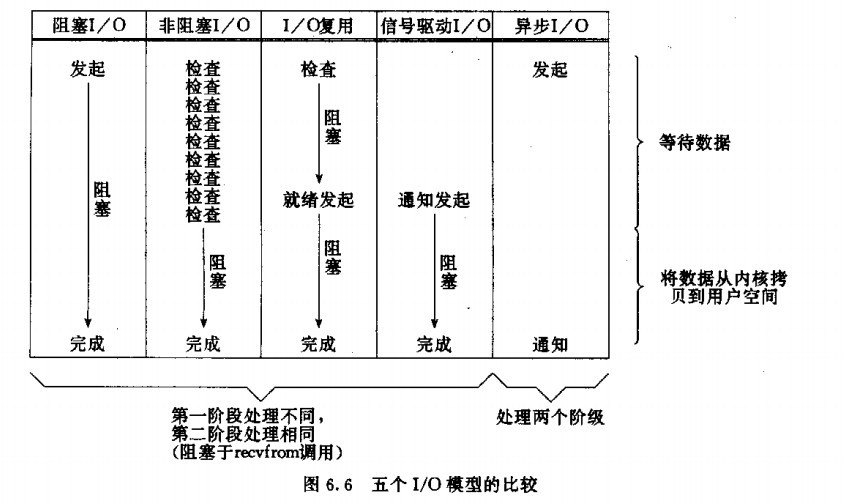


 2.4、异步IO

当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者的输入输出操作



最后，总结比较下五种IO模型：



最后，再举几个不是很恰当的例子来说明这四个IO Model:  
有A，B，C，D四个人在钓鱼：  
A用的是最老式的鱼竿，所以呢，得一直守着，等到鱼上钩了再拉杆；  
B的鱼竿有个功能，能够显示是否有鱼上钩，所以呢，B就和旁边的MM聊天，隔会再看看有没有鱼上钩，有的话就迅速拉杆；  
C用的鱼竿和B差不多，但他想了一个好办法，就是同时放好几根鱼竿，然后守在旁边，一旦有显示说鱼上钩了，它就将对应的鱼竿拉起来；  
D是个有钱人，干脆雇了一个人帮他钓鱼，一旦那个人把鱼钓上来了，就给D发个短信。