一：什么是IO？

大白话：

1. 网络IO本质上也是IO的一种，一般的IO就是数据的输入输出，再简单来说，就是从一个地方，到另一个地方
2. 在计算机中，能够存储数据的，一定是存储介质，而且一般采用的存储结构无非是数组、链表、树。而且我们知道，网络中的数据是以二进制的形式传输的，所以，可以简单认为，数据就是二进制数组中的元素。
3. 那么，就有一个简单的推论，网络IO其实就是数据从一个数组到另一个数组的过程。接下来，就把这个数组具体化。

稍微提高一点专业性

1. 网络IO的硬件基础是网卡，数据到达网卡之后，终点是应用程序的内存，但是，应用程序是无法直接访问网卡的，所以需要操作系统做中转。
2. 所以，上面提到的两个数组，分别是操作系统的IO数组和应用程序内存中的数组
3. 操作系统的IO数组，就是io buffer，程序的数组，就是我们熟悉的字节数组。

小结

1. 网络IO的过程，就是操作系统接收到网卡的数据，缓存到一个buffer中，然后应用程序调用操作系统的函数，从对应的buffer中取出数据。
2. 这么看来，操作系统可以理解为一个较为抽象的消息中间件。

二：一次IO请求中，涉及到什么角色，整个请求的过程是什么？

对于一个网络IO来说，他会涉及到两个系统对象，一个是调用这个IO的process或者thread，另 一个就是系统内核（kernel）。当一个read操作发生的时候，会经历两个阶段：

1：等待数据在内核中准备完毕。

2：将数据从内核拷贝到进程中。

##### 三：常见的IO模型有哪些？

所谓的IO模型，是操作系统提供的IO函数，和具体的语言无关，不过我个人熟悉的是java，就和java结合来说。

常见的IO模型，分别是BIO、NIO、Select、POLL、EPOLL，后三者又统称为多路复用器。

所有的网络模型解决的核心问题都是以下3个问题：

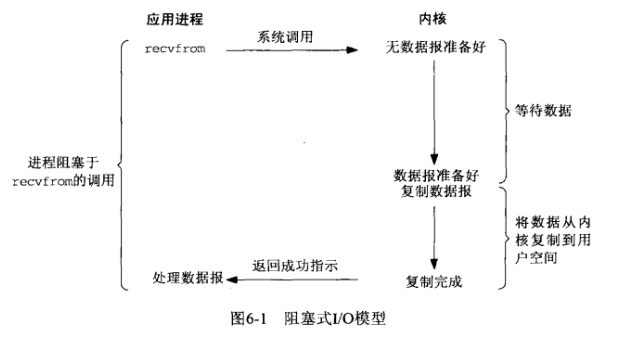
1. 有哪些网络连接？
2. 哪个连接有数据？
3. 如何读取数据？

下面我们说的IO模型都是Linux系统下的IO模型。

四：什么是传统IO模型？传统IO的缺陷在哪里？

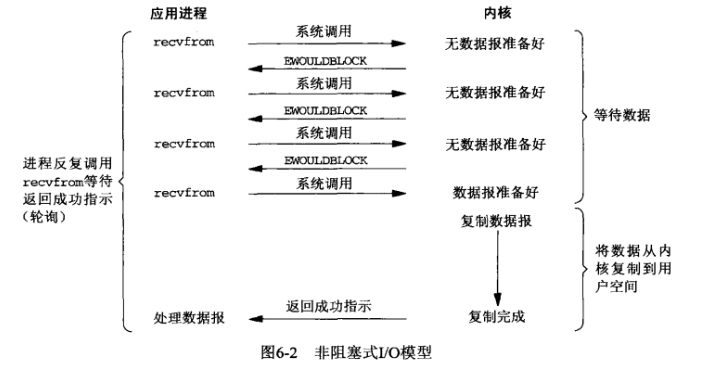
1. 传统IO也就是BIO，无论是获取新的连接还是读取指定连接的数据，调用操作系统的函数都是阻塞的，如果要实现服务多个连接，就必须每个连接建立一个线程异步处理，否则，当建立起一个连接，但是客户端不发送数据，服务端就会被这个客户端占用，无法接受新的连接。

当用户进程调用了recvfrom这个系统调用，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。



BIO的缺陷很明显，IO执行的两个阶段都被block了。

后来，人们对BIO进行了改进，出现了non-blockingIO

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。

但是，non-blockingIO模型下，每次有一个IO连接，用户线程就要新建一个线程去处理IO连接，也就是说，IO连接与线程是以一对应的关系。在这种情况下，IO多路复用模型出现了。

五：IO多路复用的出现解决了什么问题，是怎么解决的？

1. 5.1：初始的IO多路复用：单个process就可以同时处理多个网络连接IO，它的原理在于select函数，这个函数会在kernel准备好数据之后告诉用户线程。  
    当一个IO请求发起时，不是会调用recvfrom函数，而是先调用select函数，调用之后，整个进程就会陷入阻塞状态，而同时，kernel会监视所有select负责的socket，当任何一个socket准备好了之后，select就会返回，这个时候在调用recvfrom操作，数据就一定是准备好的，然后，第二阶段IO开始，数据从kernel拷贝到用户进程。



图3 多路分离函数select

这时你可能会问：那还不是第一阶段会立即堵塞？事实上，是的，甚至，还不如同步阻塞IO，因为这里需要使用两个system call (select 和 recvfrom)，而blocking IO只调用了一个system call (recvfrom)。  
这里需要搞清楚一个概念，这里的阻塞，是process被select函数阻塞，而同步阻塞IO中process是被Socket IO阻塞。  
但是IO多路复用的优势在于：IO多路复用可以由一个process同时处理多个connection（也就是注册在select里面的网络连接或者叫文件描述符）。用户可以在一个线程内同时处理多个socket的IO请求。用户可以在select上注册多个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（我的理解是：一个线程同时监控内核处理多个IO请求）**。（思考：IO多路复用不适用于处理哪种IO呢？）

用户线程使用select函数的伪代码描述为：

{

select(socket);

while(1) {

sockets = select();

for(socket in sockets) {

if(can\_read(socket)) {

read(socket, buffer);

process(buffer);

}

}

}

}

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。

但是这种IO复用是有问题的。

3.2升级的IO多路复用

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。

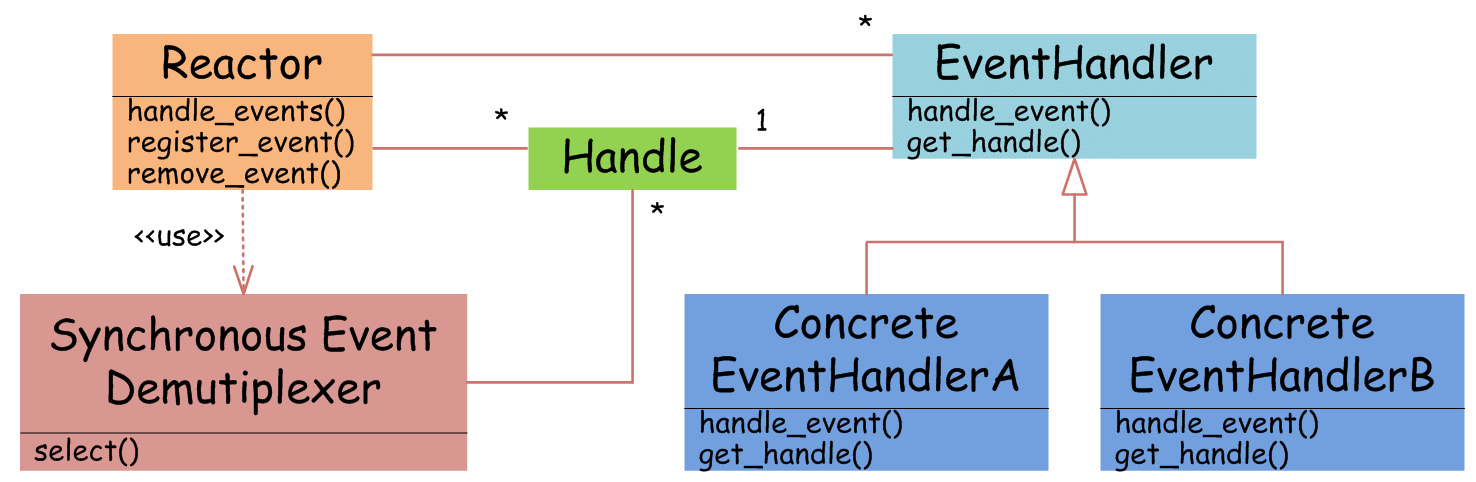


图4 Reactor设计模式

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。



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NONBLOCK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：

void UserEventHandler::handle\_event() {

if(can\_read(socket)) {

read(socket, buffer);

process(buffer);

}

}

{

Reactor.register(new UserEventHandler(socket));

}

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。

Reactor::handle\_events() {

while(1) {

sockets = select();

for(socket in sockets) {

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

}

}

}

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。

也许你会问：如果每次select()函数询问的时候内核都准备好了多个IO连接请求的数据，那这些数据还不是要被这个复用的线程排队处理？

没错，是这样的，但是第二阶段的IO一般来说都特别快，即使是被一个线程排队顺序执行也浪费不了多少时间，当然，一个线程的能力也是有限的。

六：IO多路复用适合应用在什么样的场景？优点和缺陷是什么？

七：进阶1：IO复用模型的核心在于select()函数，那么select()函数可以怎么被优化？请结合代码分析。

1. Select，这是最初级的多路复用器，从NIO到多路复用器，其实就是一个从多次到批量的演进，多路复用器支持一次询问多个文件描述符（fd）（linux中，一切皆为文件，连接也是文件，有对应的文件描述符）。从多次到批量，就能节省大量的运行态切换成本。但是select的问题在于，批量有上限，是有限的批量。但是，它仅仅知道了，有I/O事件发生了，却并不知道是哪那几个流（可能有一个，多个，甚至全部），我们只能无差别轮询所有流，找出能读出数据，或者写入数据的流，对他们进行操作。所以**select具有O(n)的无差别轮询复杂度**，同时处理的流越多，无差别轮询时间就越长。能监听的端口：32位机默认是1024个。64位机默认是2048个.
2. POLL，解决了select的上限问题，一次可以询问任意个数的fd，真正做到了批量。但是，即使减少了运行态切换的成本，针对每次传来的fd，操作系统依然需要逐个遍历，复杂度依然是O(n)，只是每次操作的损耗降低了。所以，poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态， **但是它没有最大连接数的限制**，原因是它是基于链表来存储的.

EPOLL，解决了POLL和Select存在的遍历问题，将复杂度降为O(1)，操作系统提前维护好用户程序对应的fd，每次有数据到达，就把对应的fd放到一个数据结构中存起来，当用户程序需要读取数据时，直接把这些有状态的fd返回，用户程序一次性获取fd，逐个读取即可。用户只要调用一次，操作系统也不需要遍历。这是目前大部分场景下，最高效的模型。**epoll可以理解为event poll**，不同于忙轮询和无差别轮询，epoll会把哪个流发生了怎样的I/O事件通知我们。所以我们说epoll实际上是**事件驱动（每个事件关联上fd）**的，此时我们对这些流的操作都是有意义的。**（复杂度降低到了O(1)）  
下面再详细介绍一下epoll:**epoll有EPOLLLT和EPOLLET两种触发模式，LT是默认的模式，ET是“高速”模式。LT模式下，只要这个fd还有数据可读，每次 epoll\_wait都会返回它的事件，提醒用户程序去操作，而在ET（边缘触发）模式中，它只会提示一次，直到下次再有数据流入之前都不会再提示了，无 论fd中是否还有数据可读。所以在ET模式下，read一个fd的时候一定要把它的buffer读光，也就是说一直读到read的返回值小于请求值，或者 遇到EAGAIN错误。还有一个特点是，epoll使用“事件”的就绪通知方式，通过epoll\_ctl注册fd，一旦该fd就绪，内核就会采用类似callback的回调机制来激活该fd，epoll\_wait便可以收到通知。

**epoll为什么要有EPOLLET触发模式？**

如果采用EPOLLLT模式的话，系统中一旦有大量你不需要读写的就绪文件描述符，它们每次调用epoll\_wait都会返回，这样会大大降低处理程序检索自己关心的就绪文件描述符的效率.。而采用EPOLLET这种边沿触发模式的话，当被监控的文件描述符上有可读写事件发生时，epoll\_wait()会通知处理程序去读写。如果这次没有把数据全部读写完(如读写缓冲区太小)，那么下次调用epoll\_wait()时，它不会通知你，也就是它只会通知你一次，直到该文件描述符上出现第二次可读写事件才会通知你！！！**这种模式比水平触发效率高，系统不会充斥大量你不关心的就绪文件描述符**

**epoll的优点：**

1、**没有最大并发连接的限制，能打开的FD的上限远大于1024（1G的内存上能监听约10万个端口）**；  
**2、效率提升，不是轮询的方式，不会随着FD数目的增加效率下降。只有活跃可用的FD才会调用callback函数；**  
**即Epoll最大的优点就在于它只管你“活跃”的连接，而跟连接总数无关，因此在实际的网络环境中，Epoll的效率就会远远高于select和poll。**

3、 内存拷贝，利用mmap()文件映射内存加速与内核空间的消息传递；即epoll使用mmap减少复制开销。

接下来我们来讨论一下select到epoll具体的升级策略：

**select的几大缺点：**

**（1）每次调用select，都需要把fd集合从用户态拷贝到内核态，这个开销在fd很多时会很大**

**（2）同时每次调用select都需要在内核遍历传递进来的所有fd，这个开销在fd很多时也很大**

**（3）select支持的文件描述符数量太小了，默认是1024**

**2 poll实现**

　　poll的实现和select非常相似，只是描述fd集合的方式不同，poll使用pollfd结构而不是select的fd\_set结构，其他的都差不多,管理多个描述符也是进行轮询，根据描述符的状态进行处理，**但是poll没有最大文件描述符数量的限制**。poll和select同样存在一个缺点就是，包含大量文件描述符的数组被整体复制于用户态和内核的地址空间之间，而不论这些文件描述符是否就绪，它的开销随着文件描述符数量的增加而线性增大。

**3、epoll**

　　epoll既然是对select和poll的改进，就应该能避免上述的三个缺点。那epoll都是怎么解决的呢？在此之前，我们先看一下epoll和select和poll的调用接口上的不同，select和poll都只提供了一个函数——select或者poll函数。而epoll提供了三个函数，epoll\_create,epoll\_ctl和epoll\_wait，epoll\_create是创建一个epoll句柄；epoll\_ctl是注册要监听的事件类型；epoll\_wait则是等待事件的产生。

　　对于第一个缺点，epoll的解决方案在epoll\_ctl函数中。每次注册新的事件到epoll句柄中时（在epoll\_ctl中指定EPOLL\_CTL\_ADD），会把所有的fd拷贝进内核，而不是在epoll\_wait的时候重复拷贝。epoll保证了每个fd在整个过程中只会拷贝一次。

　　对于第二个缺点，epoll的解决方案不像select或poll一样每次都把current轮流加入fd对应的设备等待队列中，而只在epoll\_ctl时把current挂一遍（这一遍必不可少）并为每个fd指定一个回调函数，当设备就绪，唤醒等待队列上的等待者时，就会调用这个回调函数，而这个回调函数会把就绪的fd加入一个就绪链表）。epoll\_wait的工作实际上就是在这个就绪链表中查看有没有就绪的fd（利用schedule\_timeout()实现睡一会，判断一会的效果，和select实现中的第7步是类似的）。

　　对于第三个缺点，epoll没有这个限制，它所支持的FD上限是最大可以打开文件的数目，这个数字一般远大于2048,举个例子,在1GB内存的机器上大约是10万左右，具体数目可以cat /proc/sys/fs/file-max察看,一般来说这个数目和系统内存关系很大。

**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。

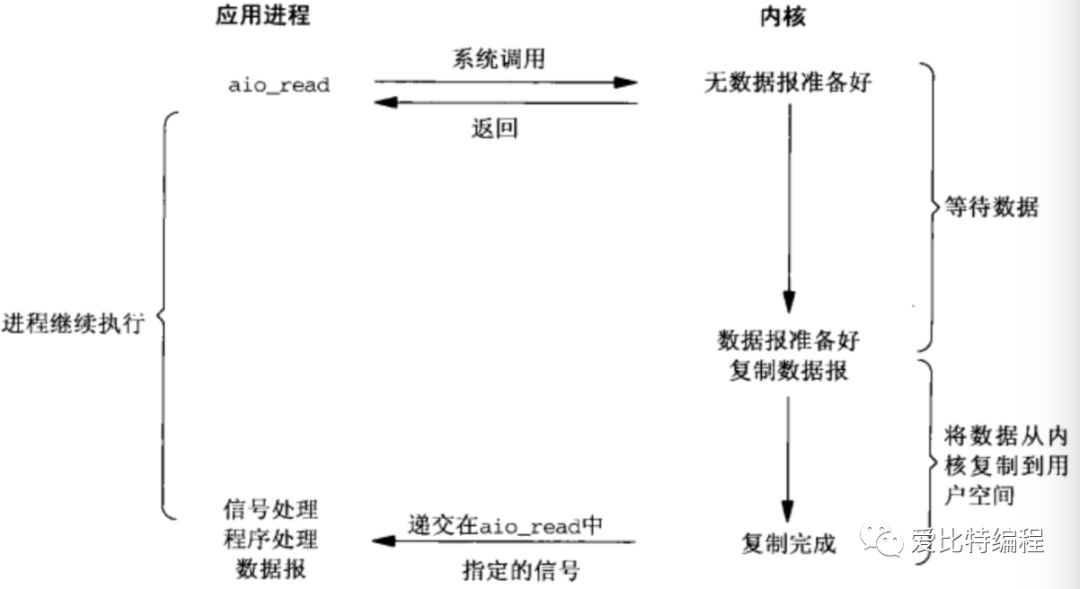
（2）select，poll每次调用都要把fd集合从用户态往内核态拷贝一次，并且要把current往设备等待队列中挂一次，而epoll只要一次拷贝，而且把current往等待队列上挂也只挂一次（在epoll\_wait的开始，注意这里的等待队列并不是设备等待队列，只是一个epoll内部定义的等待队列）。这也能节省不少的开销。

八：进阶2：AsynchronousIO复用模型是怎么回事？

异步I/O(asynchronous I/O或者AIO)，数据准备通知和数据拷贝两个阶段都在内核态完成，两个阶段都不会阻塞，真正的异步I/O。

进程调用read/readfrom时，内核立刻返回，进程不会阻塞，进程可以去干其他的事情，当内核通知进程数据已经完成后，进程直接可以处理数据，不需要再拷贝数据，因为内核已经将数据从内核态拷贝到用户态，进程可以直接处理数据。

异步I/O模型



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，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。