Python 并发编程-事件驱动模型

一、事件驱动模型介绍

1、传统的编程模式

例如：线性模式大致流程

开始--->代码块 A--->代码块 B--->代码块 C--->代码块 D--->.....--->结束

每一个代码块里是完成各种各样事情的代码，但编程者知道代码块 A,B,C,D...的执行顺序，唯一能够改变这个流程的是数据。输入不同的数据，根据条件语句判断，流程或许就改为 A--->C--->E...--->结束。每一次程序运行顺序或许都不同，但它的控制流程是由输入数据和你编写的程序决定的。如果你知道这个程序当前的运行状态（包括输入数据和程序本身），那你就知道接下来甚至一直到结束它的运行流程。

例如：事件驱动型程序模型大致流程

开始--->初始化--->等待

与上面传统编程模式不同，事件驱动程序在启动之后，就在那等待，等待什么呢？等待被事件触发。传统编程下也有“等待”的时候，比如在代码块 D 中，你定义了一个 input()，需要用户输入数据。但这与下面的等待不同，传统编程的“等待”，比如 input()，你作为程序编写者是知道或者强制用户输入某个东西的，或许是数字，或许是文件名称，如果用户输入错误，你还需要提醒他，并请他重新输入。事件驱动程序的等待则是完全不知道，也不强制用户输入或者干什么。只要某一事件发生，那程序就会做出相应的“反应”。这些事件包括：输入信息、鼠标、敲击键盘上某个键还有系统内部定时器触发。

2、事件驱动模型

通常，我们写服务器处理模型的程序时，有以下几种模型：

（1）每收到一个请求，创建一个新的进程，来处理该请求；

（2）每收到一个请求，创建一个新的线程，来处理该请求；

（3）每收到一个请求，放入一个事件列表，让主进程通过非阻塞 I/O 方式来处理请求

3、第三种就是协程、事件驱动的方式，一般普遍认为第（3）种方式是大多数网络服务器采用的方式

示例：

```
#事件驱动之鼠标点击事件注册

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>

</head>
<body>

<p onclick="fun()">点我呀</p>


<script type="text/javascript">
    function fun() {
          alert('约吗?')
    }
</script>
</body>

</html>
```

执行结果：

在 UI 编程中，常常要对鼠标点击进行相应，首先如何获得鼠标点击呢？

两种方式：

1、创建一个线程循环检测是否有鼠标点击

那么这个方式有以下几个缺点：

1. CPU 资源浪费，可能鼠标点击的频率非常小，但是扫描线程还是会一直循环检测，这会造成很多的 CPU 资源浪费；如果扫描鼠标点击的接口是阻塞的呢？
2. 如果是堵塞的，又会出现下面这样的问题，如果我们不但要扫描鼠标点击，还要扫描键盘是否按下，由于扫描鼠标时被堵塞了，那么可能永远不会去扫描键盘；
3. 如果一个循环需要扫描的设备非常多，这又会引来响应时间的问题；
   所以，该方式是非常不好的。

2、事件驱动模型

目前大部分的 UI 编程都是事件驱动模型，如很多 UI 平台都会提供 onClick()事件，这个事件就代表鼠标按下事件。事件驱动模型大体思路如下：

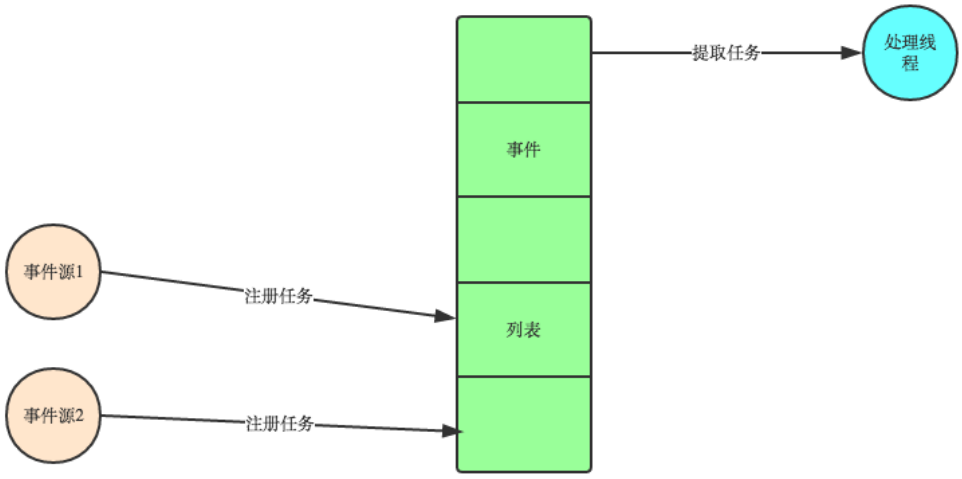
1. 有一个事件（消息）队列；
2. 鼠标按下时，往这个队列中增加一个点击事件（消息）；
3. 有个循环，不断从队列取出事件，根据不同的事件，调用不同的函数，如 onClick()、onKeyDown()等；
4. 事件（消息）一般都各自保存各自的处理函数指针，这样，每个消息都有独立的处理函数；

什么是事件驱动模型 ？

目前大部分的 UI 编程都是事件驱动模型，如很多 UI 平台都会提供 onClick()事件，这个事件就代表鼠标按下事件。事件驱动模型大体思路如下：

1. 有一个事件（消息）队列；
2. 鼠标按下时，往这个队列中增加一个点击事件（消息）；

3. 有个循环，不断从队列取出事件，根据不同的事件，调用不同的函数，如onClick()、onKeyDown()等；
4. 事件（消息）一般都各自保存各自的处理函数指针，这样，每个消息都有独立的处理函数；



事件驱动编程是一种编程范式，这里程序的执行流由外部事件来决定。它的特点是包含一个事件循环，当外部事件发生时使用回调机制来触发相应的处理。另外两种常见的编程范式是（单线程）同步以及多线程编程。

需知：每个 cpu 都有其一套可执行的专门指令集，如 SPARC 和 Pentium，其实每个硬件之上都要有一个控制程序，cpu 的指令集就是 cpu 的控制程序。

二、IO 模型准备

在进行解释之前，首先要说明几个概念：

1. 用户空间和内核空间
2. 进程切换
3. 进程的阻塞
4. 文件描述符
5. 缓存 I/O

1、用户空间和内核空间

例如：采用虚拟存储器，对于 32bit 操作系统，它的寻址空间(虚拟存储空间为 4G，即 2 的 32 次方)。

操作系统的核心是内核，独立于普通的应用程序，可以访问受保护的内存空间，也可以访问底层硬件的所有权限。

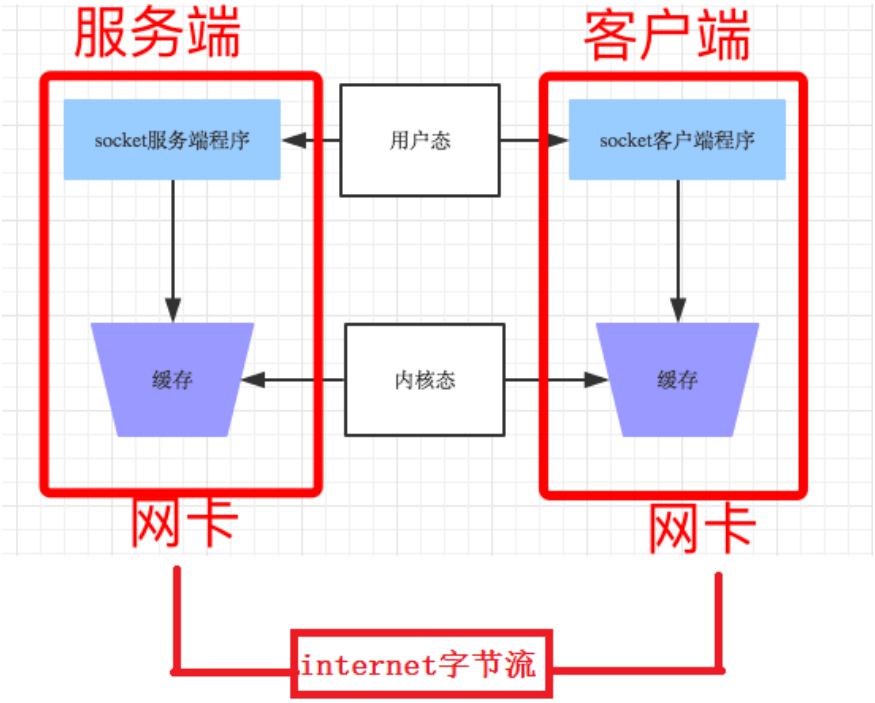
为了保证用户进程不能直接操作内核(kernel),保证内核的安全，操作系统将虚拟空间划分为两部分：一部分为内核空间，另一部分为用户空间。

那么操作系统是如何分配空间的？这里就会涉及到内核态和用户态的两种工作状态。

1G： 0 --->内核态
3G： 1 --->用户态

CPU 的指令集，是通过 0 和 1 决定你是用户态，还是内核态

计算机的两种工作状态： 内核态和用户态



cpu 的两种工作状态：

现在的操作系统都是分时操作系统，分时的根源，来自于硬件层面操作系统内核占用的内存与应用程序占用的内存彼此之间隔离。cpu 通过 psw（程序状态寄存器）中的一个 2 进制位来控制 cpu 本身的工作状态，即内核态与用户态。

内核态：操作系统内核只能运作于 cpu 的内核态，这种状态意味着可以执行 cpu 所有的指令，可以执行 cpu 所有的指令，这也意味着对计算机硬件资源有着完全的控制权限，并且可以控制 cpu 工作状态由内核态转成用户态。

用户态：应用程序只能运作于 cpu 的用户态，这种状态意味着只能执行 cpu 所有的指令的一小部分（或者称为所有指令的一个子集），这一小部分指令对计算机的硬件资源没有访问权限（比如 I/O），并且不能控制由用户态转成内核态。

2、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 上执行的进程，并恢复以前挂起的某个进程的执行，这种行为就被称为进程切换。

总结：进程切换是很消耗资源的。

3、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。

4、文件描述符 fd

文件描述符（File descriptor）是计算机科学中的一个术语，是一个用于表述指向文件的引用的抽象化概念。
文件描述符在形式上是一个非负整数。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。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。

5、缓存
I/O

缓存 I/O 又被称作标准 I/O，大多数文件系统的默认 I/O 操作都是缓存 I/O。在 Linux 的缓存 I/O 机制中，操作系统会将 I/O 的数据缓存在文件系统的页缓存（ page cache ）中，也就是说，数据会先被拷贝到操作系统内核的缓冲区中，然后才会从操作系统内核的缓冲区拷贝到应用程序的地址空间。用户空间没法直接访问内核空间的，内核态到用户态的数据拷贝。

缓存 I/O 的缺点：

数据在传输过程中需要在应用程序地址空间和内核进行多次数据拷贝操作，这些数据拷贝操作所带来的 CPU 以及内存开销是非常大的。

本文讨论的背景是 Linux 环境下的 network IO。

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，

1、一个是调用这个 IO 的 process (or thread)，

2、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)

2、将数据从内核拷贝到进程中 (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。

常见的几种 IO 模型：

- blocking IO （阻塞 IO）
- nonblocking IO （非阻塞 IO）
- IO multiplexing （IO 多路复用）
- signal driven IO （信号驱动式 IO）
- asynchronous IO （异步 IO）

一、不常用的 IO 模型

1、信号驱动 IO 模型（Signal-driven IO）

使用信号，让内核在描述符就绪时发送 SIGIO 信号通知应用程序，称这种模型为信号驱动式 I/O（signal-driven I/O）。

原理图：

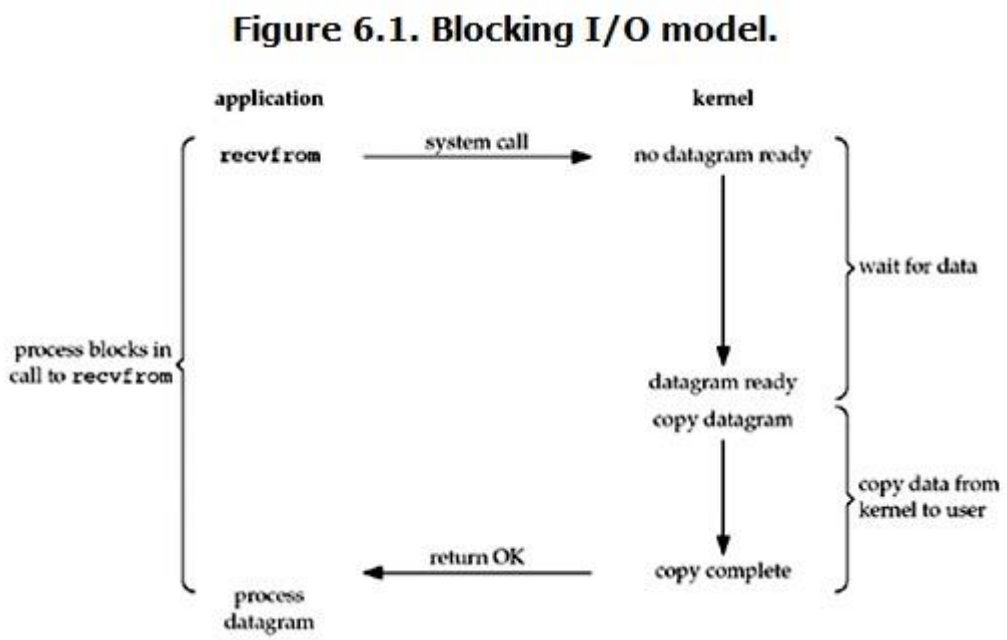
首先开启套接字的信号驱动式 I/O 功能，并通过 sigaction 系统调用安装一个信号处理函数。该系统调用将立即返回，我们的进程继续工作，也就是说进程没有被阻塞。当数据报准备好读取时，内核就为该进程产生一个 SIGIO 信号。随后就可以在信号处理函数中调用 recvfrom 读取数据报，并通知主循环数据已经准备好待处理，也可以立即通知主循环，让它读取数据报。

无论如何处理 SIGIO 信号，这种模型的优势在于等待数据报到达期间进程不被阻塞。主循环可以继续执行 ，只要等到来自信号处理函数的通知：既可以是数据已准备好被处理，也可以是数据报已准备好被读取。

二、常用的四种 IO 模型：

1、 blocking IO（阻塞 IO 模型）

原理图：



示例：一收一发程序会进入死循环

server.py

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#Author: nulige

import socket

sk=socket.socket()

sk.bind(("127.0.0.1",8080))

sk.listen(5)

while 1:
    conn,addr=sk.accept()
```

```
15
16      while 1:
17          conn.send("hello client".encode("utf8"))
18          data=conn.recv(1024)
19          print(data.decode("utf8"))
```

client.py

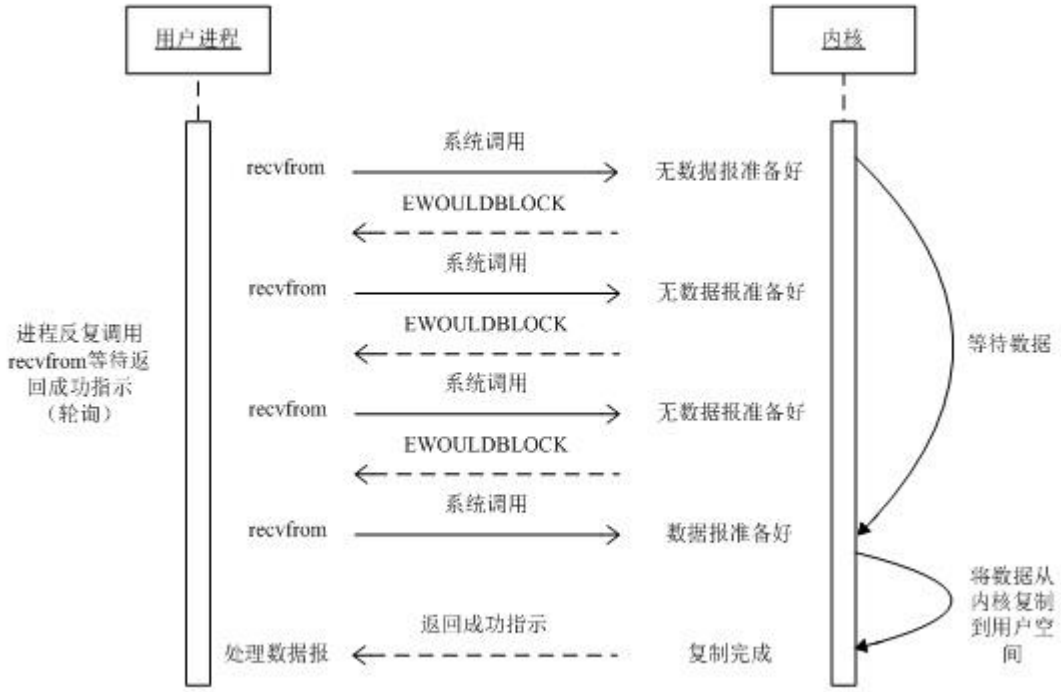
```
 1 #!/usr/bin/env python
 2 # -*- coding:utf-8 -*-
 3 #Author: nulige
 4
 5 import socket
 6
 7 sk=socket.socket()
 8
 9 sk.connect(("127.0.0.1",8080))
10
11 while 1:
12     data=sk.recv(1024)
13     print(data.decode("utf8"))
14     sk.send(b"hello server")
```

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。

所以，blocking IO 的特点就是在 IO 执行的两个阶段都被 block 了。

2、non-blocking IO(非阻塞 IO)

原理图：



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。

注意：

在网络 IO 时候，非阻塞 IO 也会进行 recvform 系统调用，检查数据是否准备好，与阻塞 IO 不一样，”非阻塞将大的整片时间的阻塞分成 N 多的小的阻塞, 所以进程不断地有机会 ‘被’ CPU 光顾”。即每次 recvform 系统调用之间，cpu 的权限还在进程手中，这段时间是可以做其他事情的，

也就是说非阻塞的 recvform 系统调用调用之后，进程并没有被阻塞，内核马上返回给进程，如果数据还没准备好，此时会返回一个 error。进程在返回之后，可以干点别的事情，然后再发起 recvform 系统调用。重复上面

的过程，循环往复的进行 recvform 系统调用。这个过程通常被称之为轮询。轮询检查内核数据，直到数据准备好，再拷贝数据到进程，进行数据处理。需要注意，拷贝数据整个过程，进程仍然是属于阻塞的状态。

示例：

服务端：

```
import time
import socket
sk = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
sk.bind(('127.0.0.1',6667))
sk.listen(5)
sk.setblocking(False)  #设置成非阻塞状态
while True:
    try:
        print ('waiting client connection .......')
        connection,address = sk.accept()   # 进程主动轮询
        print("+++",address)
        client_messge = connection.recv(1024)
        print(str(client_messge,'utf8'))
        connection.close()
    except Exception as e:  #捕捉错误
        print (e)
        time.sleep(4)  #每 4 秒打印一个捕捉到的错误
```

客户端：

```
import time
import socket
sk = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
```

```
4
5 while True:
6     sk.connect(('127.0.0.1',6667))
7     print("hello")
8     sk.sendall(bytes("hello","utf8"))
9     time.sleep(2)
10    break
```

缺点：

1、发送了太多系统调用数据

2、数据处理不及时

3、IO multiplexing（IO 多路复用）

IO multiplexing 这个词可能有点陌生，但是如果我说 select，epoll，大概就都能明白了。有些地方也称这种 IO 方式为 event driven IO。我们都知道，select/epoll 的好处就在于单个 process 就可以同时处理多个网络连接的 IO。它的基本原理就是 select/epoll 这个 function 会不断的轮询所负责的所有 socket，当某个 socket 有数据到达了，就通知用户进程。

IO 多路复用的三种方式：

1、select--->效率最低，但有最大描述符限制，在 linux 为 1024。

2、poll ---->和 select 一样，但没有最大描述符限制。

3、epoll --->效率最高，没有最大描述符限制，支持水平触发与边缘触发。

IO 多路复用的优势：同时可以监听多个连接，用的是单线程，利用空闲时间实现并发。

select、poll、epoll三者的区别	
select	select最早于1983年出现在4.2BSD中，它通过一个select()系统调用来监视多个文件描述符的数组，当select()返回后，该数组中就绪的文件描述符便会被内核修改标志位，使得进程可以获得这些文件描述符从而进行后续的读写操作。 select目前几乎在所有的平台上支持 select的一个缺点在于单个进程能够监视的文件描述符的数量存在最大限制，在Linux上一般为1024，不过可以通过修改宏定义甚至重新编译内核的方式提升这一限制。 另外，select()所维护的存储大量文件描述符的数据结构，随着文件描述符数量的增大，其复制的开销也线性增长。同时，由于网络响应时间的延迟使得大量TCP连接处于非活跃状态，但调用select()会对所有socket进行一次线性扫描，所以这也浪费了一定的开销。
poll	poll和select在本质上没有多大差别，但是poll没有最大文件描述符数量的限制。 一般也不用它，相当于过渡阶段。
epoll	poll和select在本质上没有多大差别，但是poll没有最大文件描述符数量的限制。 一般也不用它，相当于过渡阶段 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I/O就绪通知方法。windows不支持。 没有最大文件描述符数量的限制。 比如100个连接，有两个活跃了，epoll会告诉用户这两个两个活跃了，直接取就ok了，而select是循环一遍。 （了解）epoll可以同时支持水平触发和边缘触发（Edge Triggered，只告诉进程哪些文件描述符刚刚变为就绪状态，它只说一遍，如果我们没有采取行动，那么它将不会再次告知，这种方式称为边缘触发），理论上边缘触发的性能要更高一些，但是代码实现相当复杂。 另一个本质的改进在于epoll采用基于事件的就绪通知方式。在select/poll中，进程只有在调用一定的方法后，内核才对所有监视的文件描述符进行扫描，而epoll事先通过epoll_ctl()来注册一个文件描述符，一旦基于某个文件描述符就绪时，内核会采用类似callback的回调机制，迅速激活这个文件描述符，当进程调用epoll_wait()时便得到通知。 所以市面常见的所谓异步IO，比如nginx、Tornado等，我们叫它异步IO，实际上是IO多路复用。

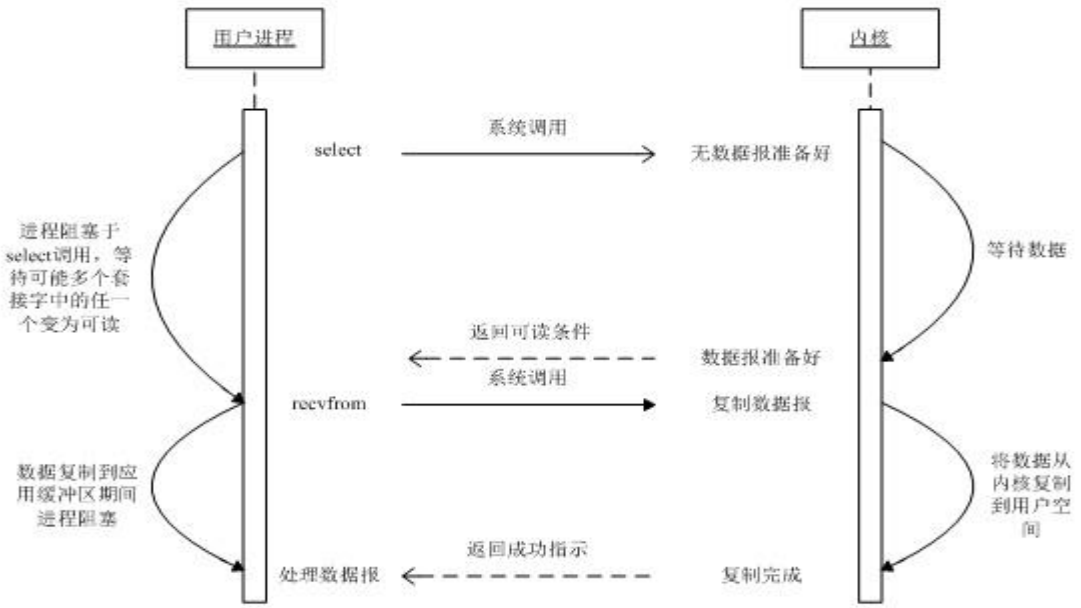
注意：

Linux 系统： select、poll、epoll

Windows 系统：select

Mac 系统：select、poll

原理图：



当用户进程调用了 select，那么整个进程会被 block，而同时，kernel 会“监视”所有 select 负责的 socket，当任何一个 socket 中的数据准备好了，select 就会返回。这个时候用户进程再调用 read 操作，将数据从 kernel 拷贝到用户进程。

这个图和 blocking IO 的图其实并没有太大的不同，事实上，还更差一些。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)。但是，用 select 的优势在于它可以同时处理多个 connection。（多说一句。所以，如果处理的连接数不是很高的话，使用 select/epoll 的 web server 不一定比使用 multi-threading + blocking IO 的 web server 性能更好，可能延迟还更大。select/epoll 的优势并不是对于单个连接能处理得更快，而是在于能处理更多的连接。）

在 IO multiplexing Model 中，实际中，对于每一个 socket，一般都设置成为 non-blocking，但是，如上图所示，整个用户的 process 其实是一直被 block 的。只不过 process 是被 select 这个函数 block，而不是被 socket IO 给 block。

注意 1：select 函数返回结果中如果有文件可读了，那么进程就可以通过调用 accept()或 recv()来让 kernel 将位于内核中准备到的数据 copy 到用户区。

注意 2：select 的优势在于可以处理多个连接，不适用于单个连接

示例：

server.py

```
#server.py

import socket
import select
sk=socket.socket()
sk.bind(("127.0.0.1",9904))
sk.listen(5)

while True:
    # sk.accept() #文件描述符
    r,w,e=select.select([sk,],[],[],5)  #输入列表，输出列表，错误列表,5：是监听 5 秒
```

```
    for i in r:   #[sk,]
        conn,add=i.accept()
        print(conn)
        print("hello")
    print('>>>>>>')
```

client.py

```
import socket

sk=socket.socket()

sk.connect(("127.0.0.1",9904))

while 1:
    inp=input(">>").strip()
    sk.send(inp.encode("utf8"))
    data=sk.recv(1024)
    print(data.decode("utf8"))
```

IO 多路复用中的两种触发方式：

水平触发：如果文件描述符已经就绪可以非阻塞的执行 IO 操作了，此时会触发通知. 允许在任意时刻重复检测 IO 的状态， 没有必要每次描述符就绪后尽可能多的执行 IO. select，poll 就属于水平触发。

边缘触发：如果文件描述符自上次状态改变后有新的 IO 活动到来，此时会触发通知. 在收到一个 IO 事件通知后要尽可能 多的执行 IO 操作，因为如果在一次通知中没有执行完 IO 那么就需要等到下一次新的 IO 活动到来才能获取到就绪的描述 符. 信号驱动式 IO 就属于边缘触发。

epoll：即可以采用水平触发，也可以采用边缘触发。

1、水平触发

只有高电平或低电平的时候才触发

1-----高电平---触发

0-----低电平---不触发

示例：

server 服务端

```
#水平触发
import socket
import select
sk=socket.socket()
sk.bind(("127.0.0.1",9904))
sk.listen(5)

while True:
    r,w,e=select.select([sk,],[],[],5)  #input 输入列表，output 输出列表，er
ron 错误列表,5: 是监听 5 秒
    for i in r:   #[sk,]
        print("hello")

    print('>>>>>>')
```

client 客户端

```
import socket

sk=socket.socket()

```

```
sk.connect(("127.0.0.1",9904))

while 1:
    inp=input(">>").strip()
    sk.send(inp.encode("utf8"))
    data=sk.recv(1024)
    print(data.decode("utf8"))
```

2、边缘触发

1---------高电平--------触发

0---------低电平--------触发

IO 多路复用优势：同时可以监听多个连接

示例：select 可以监控多个对象

服务端

```
#优势
import socket
import select
sk=socket.socket()
sk.bind(("127.0.0.1",9904))
sk.listen(5)
inp=[sk,]

while True:
    r,w,e=select.select(inp,[],[],5)  #[sk,conn]，5 是每隔几秒监听一次

```

```
    for i in r:   #[sk,]
        conn,add=i.accept()  #发送系统调用
        print(conn)
        print("hello")
        inp.append(conn)
        # conn.recv(1024)
    print('>>>>>>')
```

客户端：

```
import socket

sk=socket.socket()

sk.connect(("127.0.0.1",9904))

while 1:
    inp=input(">>").strip()
    sk.send(inp.encode("utf8"))
    data=sk.recv(1024)
    print(data.decode("utf8"))
```

多了一个判断，用 select 方式实现的并发

示例：实现并发聊天功能（select+IO 多路复用，实现并发）

服务端：

```python
import socket
import select
sk=socket.socket()
sk.bind(("127.0.0.1",8801))
sk.listen(5)
inputs=[sk,]
while True:  #监听 sk 和 conn
    r,w,e=select.select(inputs,[],[],5) #conn 发生变化,sk 不变化就走 else
    print(len(r))
    #判断 sk or conn 谁发生了变化
    for obj in r:
        if obj==sk:
            conn,add=obj.accept()
            print(conn)
            inputs.append(conn)
        else:
            data_byte=obj.recv(1024)
            print(str(data_byte,'utf8'))
            inp=input('回答%s 号客户>>>'%inputs.index(obj))
            obj.sendall(bytes(inp,'utf8'))

    print('>>',r)
```

客户端:

```python
import socket
sk=socket.socket()
sk.connect(('127.0.0.1',8801))
```

```
4

5 while True:

6     inp=input(">>>>")

7     sk.sendall(bytes(inp,"utf8"))

8     data=sk.recv(1024)

9     print(str(data,'utf8'))
```

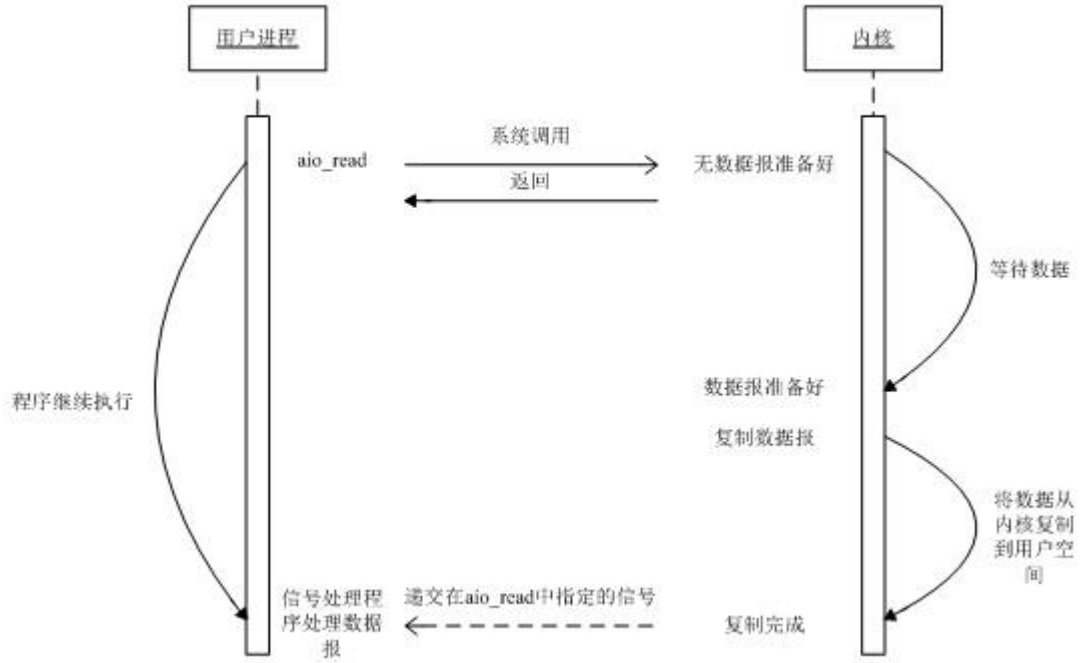
执行结果：

先运行服务端，再运行多个客户端，就可以聊天啦。（可以接收多个客户端消息）

```
 1 #server

 2 >> [<socket.socket fd=276, family=AddressFamily.AF_INET, type=SocketKind.
SOCK_STREAM, proto=0, laddr=('127.0.0.1', 8801)>]

 3 1

 4 hello

 5 回答 1 号客户>>>word

 6 >> [<socket.socket fd=344, family=AddressFamily.AF_INET, type=SocketKind.
SOCK_STREAM, proto=0, laddr=('127.0.0.1', 8801), raddr=('127.0.0.1', 5438
8)>]

 7 1

 8

 9 #clinet

10 >>>>hello

11 word
```

View Code

4、Asynchronous I/O（异步 IO）



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。

异步最大特点：全程无阻塞

synchronous IO（同步 IO)和 asynchronous IO(异步 IO)的区别：

- **A synchronous I/O operation causes the requesting process to be blocked until that I/O operationcompletes;**
- **An asynchronous I/O operation does not cause the requesting process to be blocked;**

两者的区别就在于 synchronous IO 做”IO operation”的时候会将 process 阻塞。（有一丁点阻塞，都是同步 IO）按照这个定义，之前所述的 blocking IO，non-blocking IO，IO multiplexing 都属于 synchronous IO（同步 IO）。

同步 IO：包括 blocking IO、non-blocking、select、poll、epoll（故:epool 只是伪异步而已）（有阻塞）

异步 IO：包括:asynchronous （无阻塞）

五种 IO 模型比较：

图 6.6　五个 I/O 模型的比较

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。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。

5、selectors 模块应用

python 封装好的模块：selectors

selectors 模块：会选择一个最优的操作系统实现方式

示例：

select_module.py

```
1 import selectors
```

```python
import socket

sel = selectors.DefaultSelector()

def accept(sock, mask):
    conn, addr = sock.accept()  # Should be ready
    print('accepted', conn, 'from', addr)
    conn.setblocking(False)  #设置成非阻塞
    sel.register(conn, selectors.EVENT_READ, read) #conn 绑定的是 read

def read(conn, mask):
    try:
        data = conn.recv(1000)  # Should be ready
        if not data:
            raise Exception
        print('echoing', repr(data), 'to', conn)
        conn.send(data)  # Hope it won't block
    except Exception as e:
        print('closing', conn)
        sel.unregister(conn)  #解除注册
        conn.close()

sock = socket.socket()
sock.bind(('localhost', 8090))
sock.listen(100)
sock.setblocking(False)
#注册
sel.register(sock, selectors.EVENT_READ, accept)
print("server....")

```

```
while True:
    events = sel.select() #监听[sock,conn1,conn2]
    print("events",events)
    #拿到 2 个元素，一个 key,一个 mask
    for key, mask in events:
        # print("key",key)
        # print("mask",mask)
        callback = key.data  #绑定的是 read 函数
        # print("callback",callback)
        callback(key.fileobj, mask)  #key.fileobj=sock,conn1,conn2
```

client.py

```
import socket

sk=socket.socket()

sk.connect(("127.0.0.1",8090))
while 1:
    inp=input(">>>")
    sk.send(inp.encode("utf8")) #发送内容
    data=sk.recv(1024)  #接收信息
    print(data.decode("utf8"))  #打印出来
```

执行结果：

先运行 select_module.py，再运行 clinet.py

```
#server
```

```
2

3 server....

4 events [(SelectorKey(fileobj=<socket.socket fd=312, family=AddressFamil
y.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 809
0)>, fd=312, events=1, data=<function accept at 0x01512F60>), 1)]

5 accepted <socket.socket fd=376, family=AddressFamily.AF_INET, type=Socket
Kind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 8090), raddr=('127.0.0.1', 5
7638)> from ('127.0.0.1', 57638)

6 events [(SelectorKey(fileobj=<socket.socket fd=376, family=AddressFamil
y.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 8090),
 raddr=('127.0.0.1', 57638)>, fd=376, events=1, data=<function read at 0x015
C26A8>), 1)]

7 echoing b'hello' to <socket.socket fd=376, family=AddressFamily.AF_INET,
type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 8090), raddr=('12
7.0.0.1', 57638)>

8 events [(SelectorKey(fileobj=<socket.socket fd=312, family=AddressFamil
y.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 809
0)>, fd=312, events=1, data=<function accept at 0x01512F60>), 1)]

9 accepted <socket.socket fd=324, family=AddressFamily.AF_INET, type=Socket
Kind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 8090), raddr=('127.0.0.1', 5
7675)> from ('127.0.0.1', 57675)

10 events [(SelectorKey(fileobj=<socket.socket fd=324, family=AddressFamil
y.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 8090),
 raddr=('127.0.0.1', 57675)>, fd=324, events=1, data=<function read at 0x015
C26A8>), 1)]

11 echoing b'uuuu' to <socket.socket fd=324, family=AddressFamily.AF_INET, t
ype=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 8090), raddr=('12
7.0.0.1', 57675)>

12 events [(SelectorKey(fileobj=<socket.socket fd=324, family=AddressFamil
y.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 8090),
 raddr=('127.0.0.1', 57675)>, fd=324, events=1, data=<function read at 0x015
C26A8>), 1)]

13 closing <socket.socket fd=324, family=AddressFamily.AF_INET, type=Socket
Kind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 8090), raddr=('127.0.0.1', 5
7675)>

14 events [(SelectorKey(fileobj=<socket.socket fd=312, family=AddressFamil
y.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 809
0)>, fd=312, events=1, data=<function accept at 0x01512F60>), 1)]
```

```
15 accepted <socket.socket fd=324, family=AddressFamily.AF_INET, type=Socke
tKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 8090), raddr=('127.0.0.1',
57876)> from ('127.0.0.1', 57876)

16 events [(SelectorKey(fileobj=<socket.socket fd=324, family=AddressFamil
y.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 8090),
 raddr=('127.0.0.1', 57876)>, fd=324, events=1, data=<function read at 0x015
C26A8>), 1)]

17 echoing b'welcome' to <socket.socket fd=324, family=AddressFamily.AF_INE
T, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 8090), raddr=
('127.0.0.1', 57876)>

18

19 #clinet (启动两个 client)

20 >>>hello

21 hello

22

23 >>>welcome

24 welcome
```

View Code

6、I/O 多路复用的应用场景

（1）当客户处理多个描述字时（一般是交互式输入和网络套接口），必须使用 I/O 复用。

（2）当一个客户同时处理多个套接口时，而这种情况是可能的，但很少出现。

（3）如果一个 TCP 服务器既要处理监听套接口，又要处理已连接套接口，一般也要用到 I/O 复用。

（4）如果一个服务器即要处理 TCP，又要处理 UDP，一般要使用 I/O 复用。

（5）如果一个服务器要处理多个服务或多个协议，一般要使用 I/O 复用。

’’’与多进程和多线程技术相比，I/O 多路复用技术的最大优势是系统开销小，系统不必创建进程/线程，也不必维护这些进程/线程，从而大大减小了系统的开销。’’’

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；【阻塞】
B 的鱼竿有个功能，能够显示是否有鱼上钩（这个显示功能一直去判断鱼是否

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；【非阻塞】
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；【同步】
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（消息回掉机制，主动告知）。【异步】

作业：

1、使用 IO 多路复用，做一个 ftp 的上传和下载作业

要求：实现多用户操作，可以同时上传和下载（不能用 socketserver），有显示进度条。