

## 重要概念

---

### 同步、异步

函数或方法被调用的时候，调用者是否得到**最终结果**的。

直接得到最终结果结果的，就是同步调用；

不直接得到最终结果的，就是异步调用。

### 阻塞、非阻塞

函数或方法调用的时候，是否立刻返回。

立即返回就是非阻塞调用；

不立即返回就是阻塞调用。

### 区别

同步、异步，与阻塞、非阻塞不相关。

同步、异步强调的是，是否得到（最终的）**结果**；

阻塞、非阻塞强调的是时间，是否**等待**。

同步与异步区别在于：调用者是否得到了想要的最终结果。

同步就是一定要执行到返回最终结果；

异步就是直接返回了，但是返回的不是最终结果。调用者不能通过这种调用得到结果，以后可以通过被调用者提供的某种方式（被调用者通知调用者、调用者反复查询、回调），来取回最终结果。

阻塞与非阻塞的区别在于，调用者是否还能干其他事。

阻塞，调用者就只能干等；

非阻塞，调用者可以先去忙会别的，不用一直等。

### 联系

同步阻塞，我啥事不干，就等你打饭打给我。打到饭是结果，而且我啥事不干一直等，同步加阻塞。

同步非阻塞，我等着你打饭给我，饭没好，我不等，但是我无事可做，反复看饭好了没有。打饭是结果，但是我不一直等。

异步阻塞，我要打饭，你说等叫号，并没有返回饭给我，我啥事不干，就干等着饭好了你叫我。例如，取了号什么不干就等叫自己的号。

异步非阻塞，我要打饭，你给我号，你说等叫号，并没有返回饭给我，我去看电视、玩手机，饭打好了叫我。

## 同步IO、异步IO、IO多路复用

---

### IO两个阶段

---

IO过程分两阶段：

- 1、数据准备阶段。从设备读取数据到内核空间的缓冲区（淘米，把米放饭锅里煮饭）
- 2、内核空间复制回用户空间进程缓冲区阶段（盛饭，从内核这个饭锅里面把饭装到碗里来）

系统调用——read函数、recv函数等

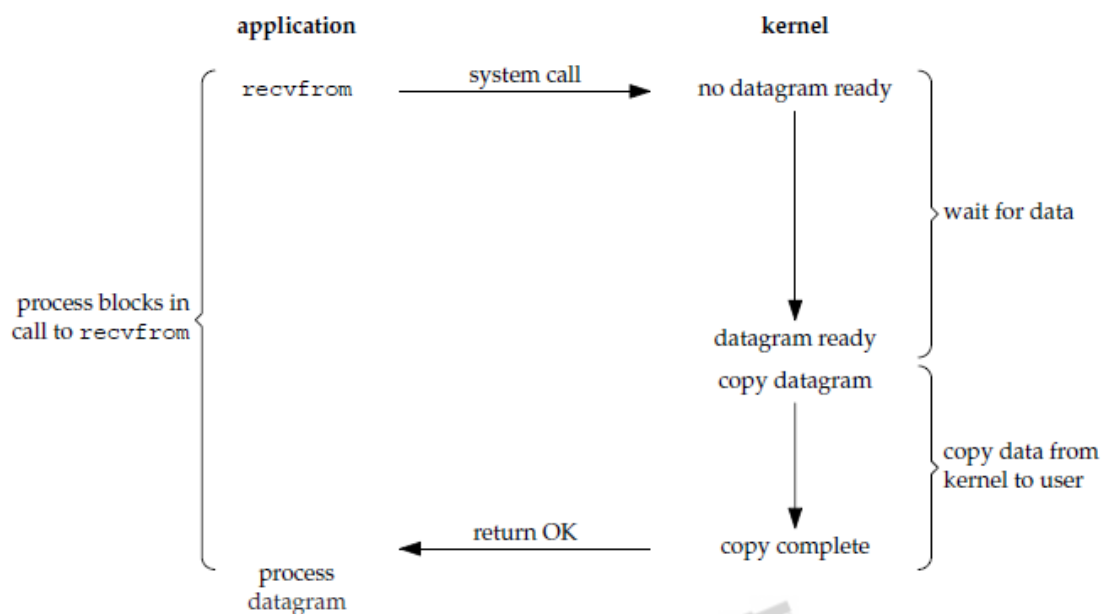
### IO模型

---

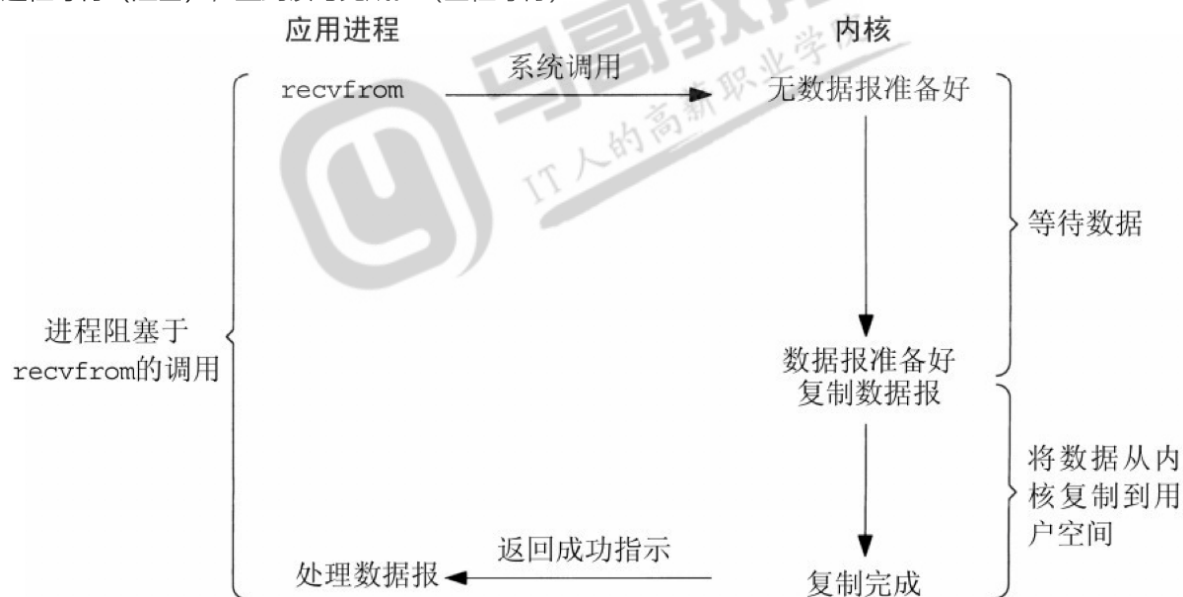
# 同步IO

同步IO模型包括 阻塞IO、非阻塞IO、IO多路复用

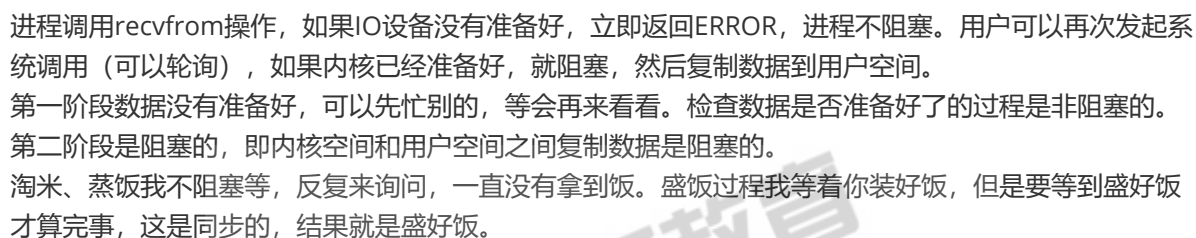
## 阻塞IO



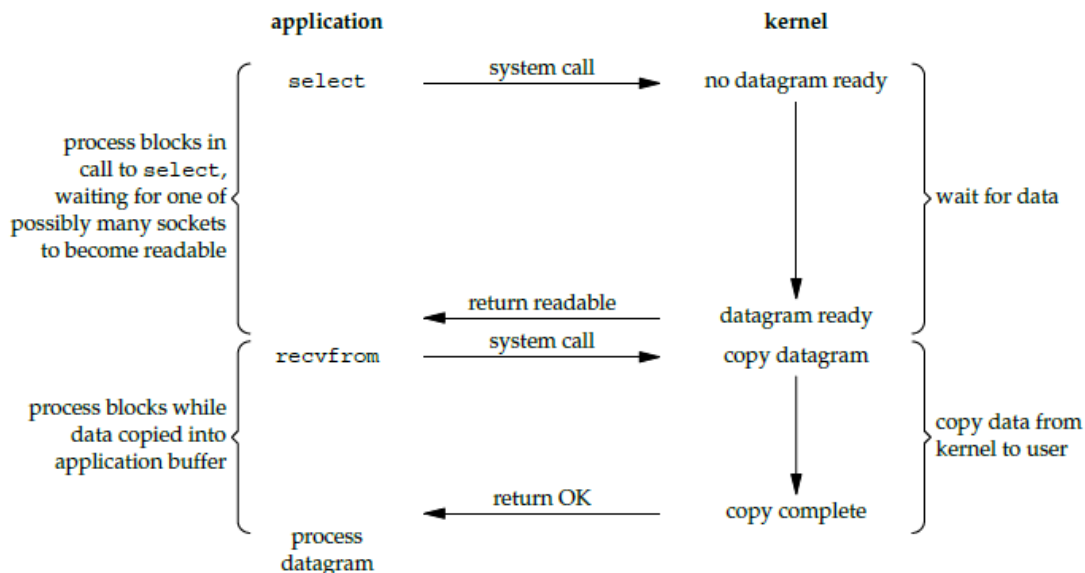
进程等待（阻塞），直到读写完成。（全程等待）



## 非阻塞IO



所谓IO多路复用，就是同时监控多个IO，有一个准备好了，就不需要等了开始处理，提高了同时处理IO的能力。



以select为例，将关注的IO操作告诉select函数并调用，进程阻塞，内核“监视”select关注的文件描述符fd，被关注的任何一个fd对应的IO准备好了数据，select返回。再使用read将数据复制到用户进程。

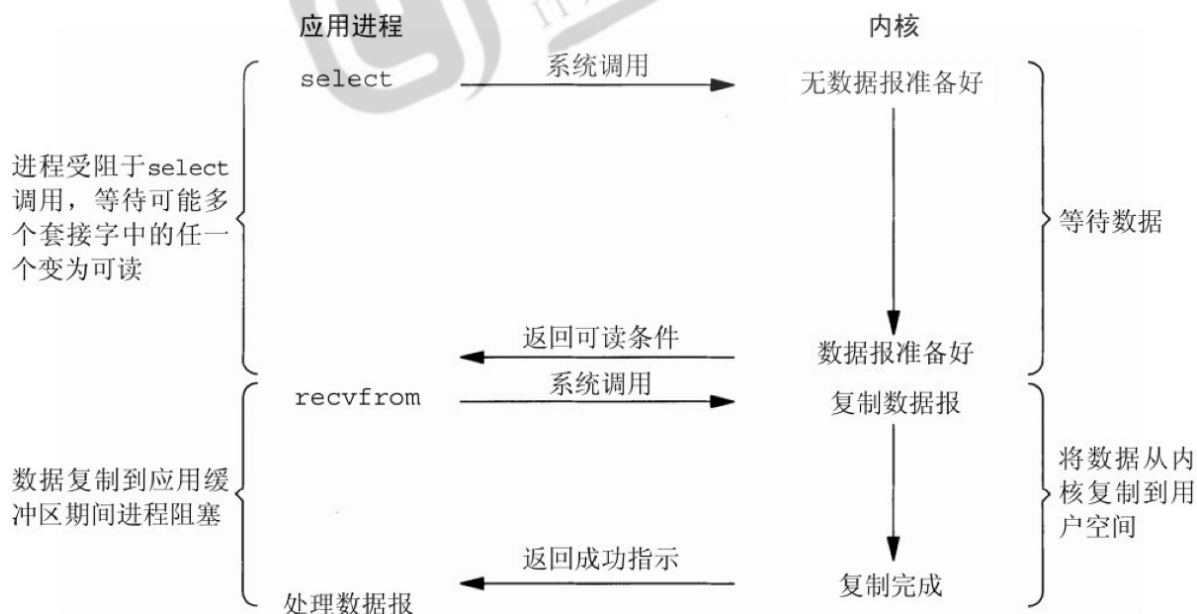
select举例：

食堂供应很多菜（众多的IO），你需要吃某三菜一汤，大师傅（操作系统）说要现做，需要你等，好多人都在等菜，谁先好不知道，你只好等待大师傅叫。你的其中一样菜好了，大师傅叫你，说你点的菜有好了的，你得自己遍历找找看哪一样才好了，请服务员把做好的菜打给你。

epoll是有菜准备好了，大师傅喊你去几号窗口直接打菜，不用自己找菜了。

一般情况下，select最多能监听1024个fd（可以修改），但是由于select采用轮询的方式，当管理的IO多了，每次都要遍历全部fd，效率低下。

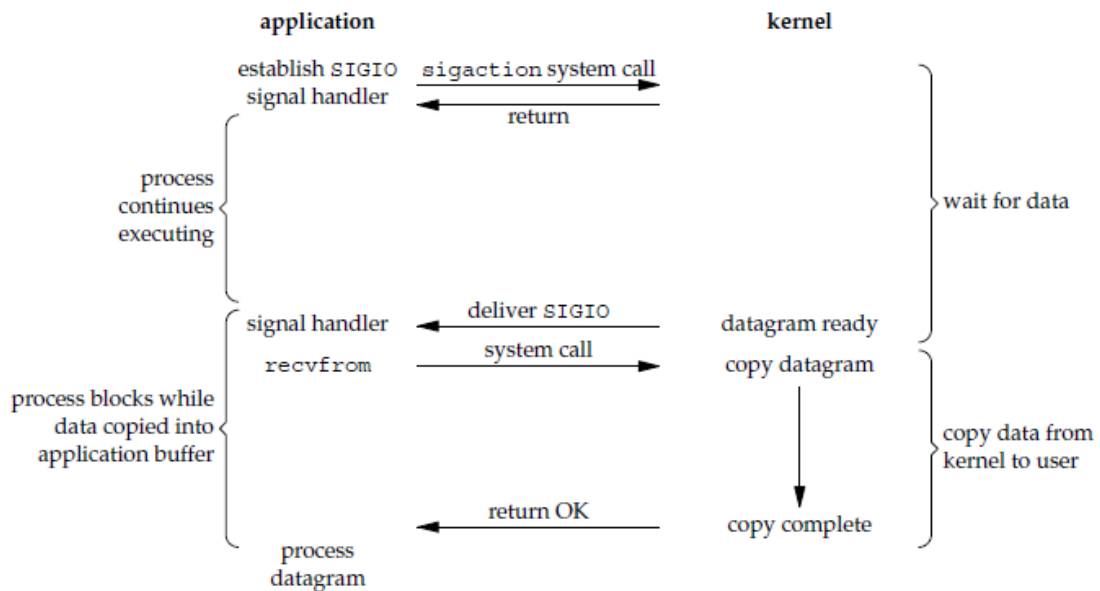
epoll没有管理的fd的上限，且是回调机制，不需遍历，效率很高。



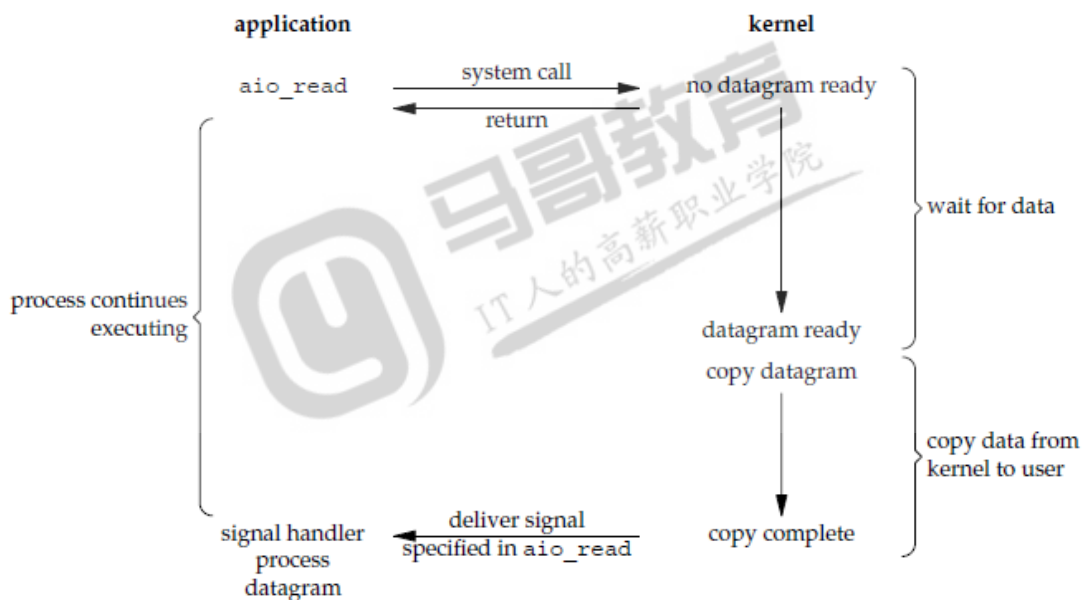
## 信号驱动IO

进程在IO访问时，先通过sigaction系统调用，提交一个信号处理函数，立即返回。进程不阻塞。

当内核准备好数据后，产生一个SIGIO信号并投递给信号处理函数。可以在此函数中调用recvfrom函数操作数据从内核空间复制到用户空间，这段过程进程阻塞。



## 异步IO

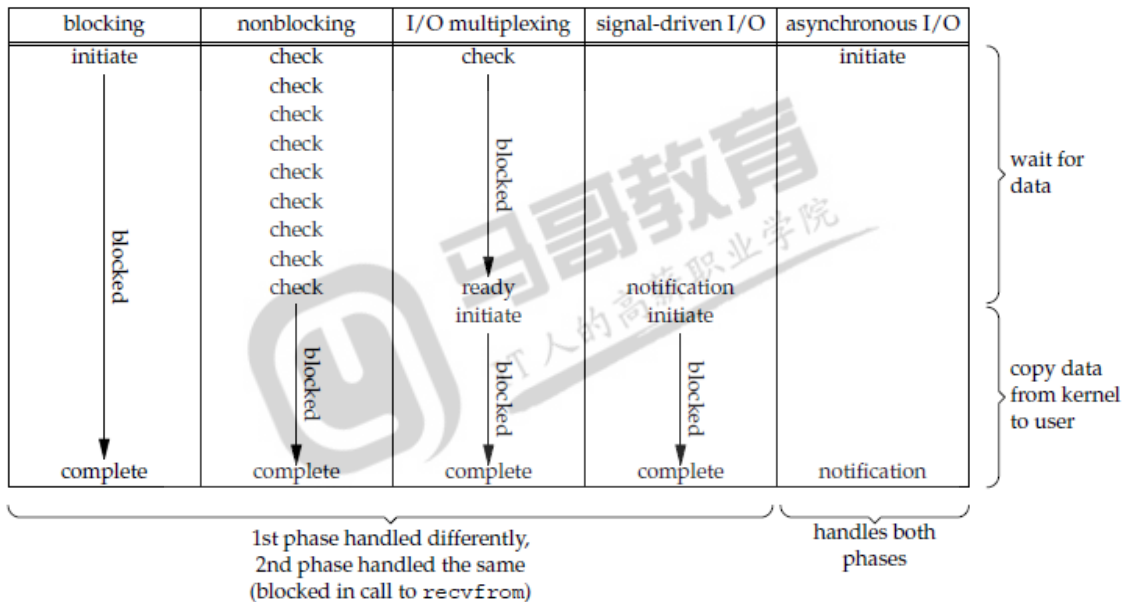
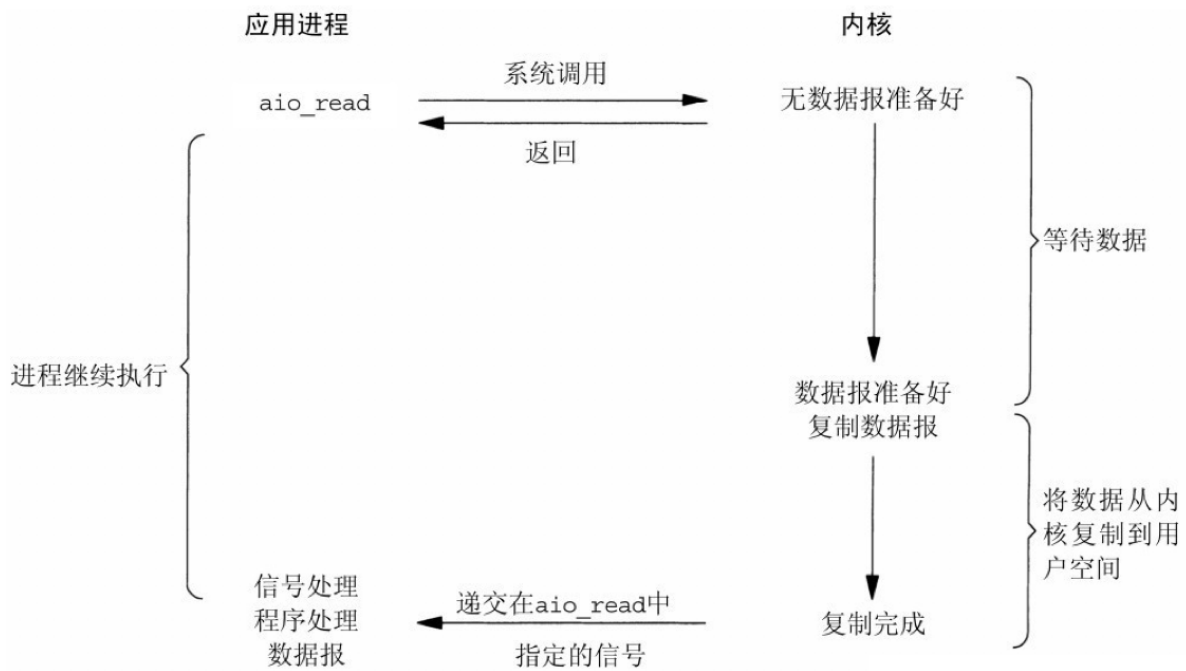


进程发起异步IO请求，立即返回。内核完成IO的两个阶段，内核给进程发一个信号。

举例，来打饭，跟大师傅说饭好了叫你，饭菜准备好了，窗口服务员把饭盛好了打电话叫你。两阶段都是异步的。在整个过程中，进程都可以忙别的，等好了才过来。

举例，今天不想出去到饭店吃饭了，点外卖，饭菜在饭店做好了（第一阶段），快递员从饭店送到你家门口（第二阶段）。

Linux的aio的系统调用，内核从版本2.6开始支持



前4个都是同步IO，因为核心操作`recv`函数调用时，进程阻塞直到拿到最终结果为止。

而异步IO进程全程不阻塞。

## Python 中 IO多路复用

- IO多路复用
  - 大多数操作系统都支持`select`和`poll`，`poll`是对`select`的升级
  - Linux系统内核2.5+ 支持`epoll`
  - BSD、Mac支持`kqueue`
  - Solaris实现了`/dev/poll`
  - Windows的IOCP

Python的`select`库实现了`select`、`poll`系统调用，这个基本上操作系统都支持。对Linux内核2.5+支持了`epoll`。

开发中的选择

- 1、完全跨平台，使用select、poll。但是性能较差
- 2、针对不同操作系统自行选择支持的技术，这样做会提高IO处理的性能

select维护一个文件描述符数据结构，单个进程使用有上限，通常是1024，线性扫描这个数据结构。效率低。

pool和select的区别是内部数据结构使用链表，没有这个最大限制，但是依然是线性遍历才知道哪个设备就绪了。

epoll使用事件通知机制，使用回调机制提高效率。

select/poll还要从内核空间复制消息到用户空间，而epoll通过内核空间和用户空间共享一块内存来减少复制。

## selectors库

3.4版本提供selectors库，高级IO复用库。

类层次结构：

BaseSelector

++ SelectSelector	实现select
++ PollSelector	实现poll
++ EpollSelector	实现epoll
++ DevpollSelector	实现devpoll
++ KqueueSelector	实现kqueue

selectors.DefaultSelector返回当前平台最有效、性能最高的实现。

但是，由于没有实现Windows下的IOCP，所以，Windows下只能退化为select。

```
# 在selectors模块源码最下面有如下代码
# Choose the best implementation, roughly:
#   epoll|kqueue|devpoll > poll > select.
# select() also can't accept a FD > FD_SETSIZE (usually around 1024)
if 'KqueueSelector' in globals():
    DefaultSelector = KqueueSelector
elif 'EpollSelector' in globals():
    DefaultSelector = EpollSelector
elif 'DevpollSelector' in globals():
    DefaultSelector = DevpollSelector
elif 'PollSelector' in globals():
    DefaultSelector = PollSelector
else:
    DefaultSelector = SelectSelector
```

事件注册

```
class SelectSelector(_BaseSelectorImpl):
    """Select-based selector."""
    def register(fileobj, events, data=None) -> SelectorKey: pass
```

- 为selector注册一个文件对象，监视它的IO事件。返回SelectorKey对象。
- fileobj 被监视文件对象，例如socket对象
- events 事件，该文件对象必须等待的事件
- data 可选的与此文件对象相关联的不透明数据，例如，关联用来存储每个客户端的会话ID，关联方法。通过这个参数在关注的事件产生后让selector干什么事。



Event常量	含义
EVENT_READ	可读 0b01, 内核已经准备好输入设备, 可以开始读了
EVENT_WRITE	可写 0b10, 内核准备好了, 可以往里写了

**selectors.SelectorKey** 有4个属性:

1. fileobj 注册的文件对象
2. fd 文件描述符
3. events 等待上面的文件描述符的文件对象的事件
4. data 注册时关联的数据

## 练习: IO多路复用TCP Server

完成一个TCP Server, 能够接受客户端请求并回应客户端消息。

```
import selectors
import threading
import socket
import logging
import time

FORMAT = "%(asctime)s %(threadName)s %(thread)d %(message)s"
logging.basicConfig(format=FORMAT, level=logging.INFO)

# 构建本系统最优Selector
selector = selectors.DefaultSelector()

sock = socket.socket() # TCP Server
sock.bind(('127.0.0.1', 9999))
sock.listen()
logging.info(sock)

sock.setblocking(False) # 注意: 建议非阻塞

# 回调函数, sock的读事件
# 形参自定义
def accept(sock:socket.socket, mask):
    """mask:事件的掩码"""
    conn, raddr = sock.accept()
    conn.setblocking(False) # 非阻塞

    logging.info('new client socket {} in accept.'.format(conn))

# 注册sock的被关注事件, 返回SelectorKey对象
# key记录了fileobj, fileobj的fd, events, data
key = selector.register(sock, selectors.EVENT_READ, accept)
logging.info(key)

# 开始循环
while True:
    # 监听注册的对象的事件, 发生被关注事件则返回events
```



```

events = selector.select()
print(events) # [(key, mask)]
# 表示那个关注的对象的某事件发生了
for key, mask in events:
    # key.data => accept; key.fileobj => sock
    callback = key.data
    callback(key.fileobj, mask)

```

上面的代码完成了Server socket的读事件的监听。注意，select()方法会阻塞到监控的对象的等待的事件有发生（监听的读或者写就绪）。

```

import selectors
import threading
import socket
import logging
import time

FORMAT = "%(asctime)s %(threadName)s %(thread)d %(message)s"
logging.basicConfig(format=FORMAT, level=logging.INFO)

# 构建本系统最优Selector
selector = selectors.DefaultSelector()

sock = socket.socket() # TCP Server
sock.bind(('127.0.0.1', 9999))
sock.listen()
logging.info(sock)

sock.setblocking(False) # 非阻塞

# 回调函数，sock的读事件
# 形参自定义
def accept(sock:socket.socket, mask):
    """mask:事件的掩码"""
    conn, raddr = sock.accept()
    conn.setblocking(False) # 非阻塞

    logging.info('new client socket {} in accept.'.format(conn))

    key = selector.register(conn, selectors.EVENT_READ, read)
    logging.info(key)

# 回调函数
def read(conn:socket.socket, mask):
    data = conn.recv(1024)
    msg = "Your msg = {} ~~~~".format(data.decode())
    logging.info(msg)
    conn.send(msg.encode())

# 注册sock的被关注事件，返回SelectorKey对象
# key记录了fileobj, fileobj的fd, events, data
key = selector.register(sock, selectors.EVENT_READ, accept)
logging.info(key)

```

```
# 开始循环
while True:
    # 监听注册的对象的事件，发生被关注事件则返回events
    events = selector.select()
    print(events) # [(key, mask)]
    # 表示那个关注的对象的某事件发生了
    for key, mask in events:
        # key.data => accept; key.fileobj => sock
        callback = key.data
        callback(key.fileobj, mask)
```

## 作业

- 使用IO多路复用改写群聊软件
  - 将ChatServer改写成IO多路复用的方式
  - 提示：还需要多线程吗？
- 自己实现事件驱动HTTPServer
  - 可以监听指定端口，例如80
  - 使用浏览器访问，可以返回一个网页

