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。
同步就是一直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（被调用着通知调用者、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，但我可以玩会手机、看看电视。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，我在旁边看电视、玩手机，饭打好了叫我。

同步IO、异步IO、IO多路复用

在386之前，CPU工作在实模式下，之后，开始支持保护模式，对内存进行了划分。
X86 CPU有4中工作级别：
Ring0级，可以执行特权指令，可以访问所有级别数据，可以访问IO设备等
Ring3级，级别最低，只能访问本级别数据
内核代码运行在Ring0，用户代码运行在Ring3。

现代操作系统采用虚拟存储器，对于32位系统来说，进程对虚拟内存地址的内存寻址空间为4G（2^{32}）。

操作系统中，内核程序独立且运行在较高的特权级别上，它们驻留在被保护的内存空间上，拥有访问硬件设备的所有权限，这部分内存称为内核空间（内核态，最高地址1G）。

普通应用程序在运行在用户空间（用户态）。

应用程序想访问某些硬件资源就需要通过操作系统提供的**系统调用**，系统调用可以使用特权指令运行在内核空间，此时进程陷入内核态运行。系统调用完成，进程将回到用户态执行用户空间代码。

IO两个阶段

IO过程分两阶段：
1、数据准备阶段
2、内核空间复制回用户空间进程缓冲区阶段

发生IO的时候：
1、内核从IO设备读、写数据（淘米，把米放饭锅里煮饭）
2、进程从内核复制数据（盛饭，从内核这个饭锅里面把饭装到碗里来）

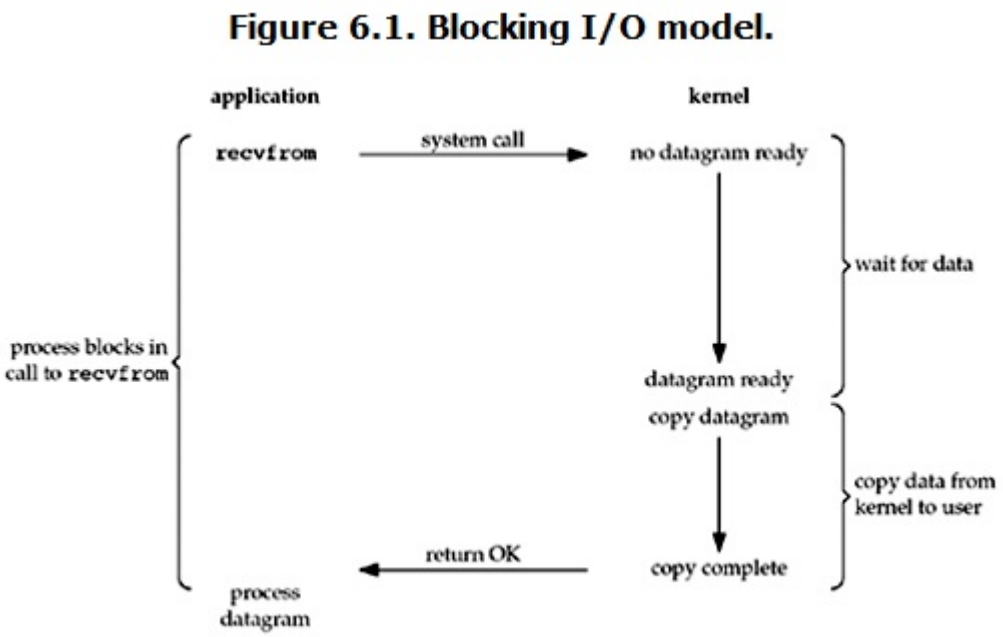
系统调用——read函数

IO模型

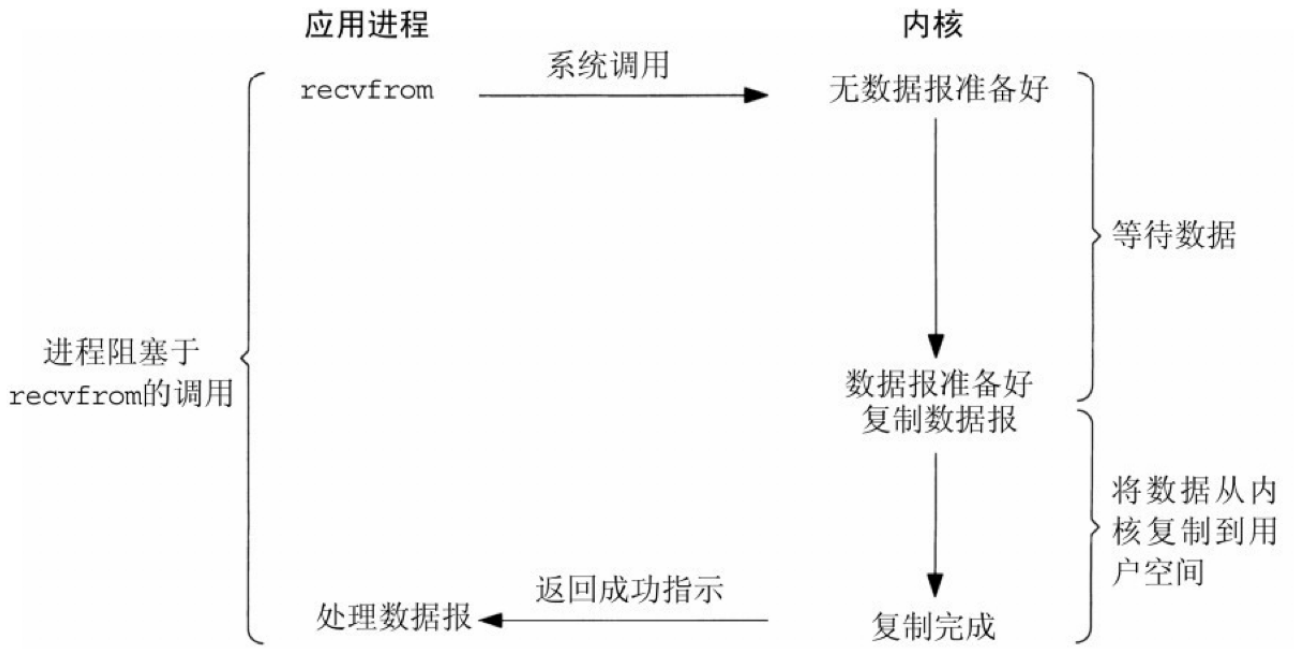
同步IO

同步IO模型包括 阻塞IO、非阻塞IO、IO多路复用

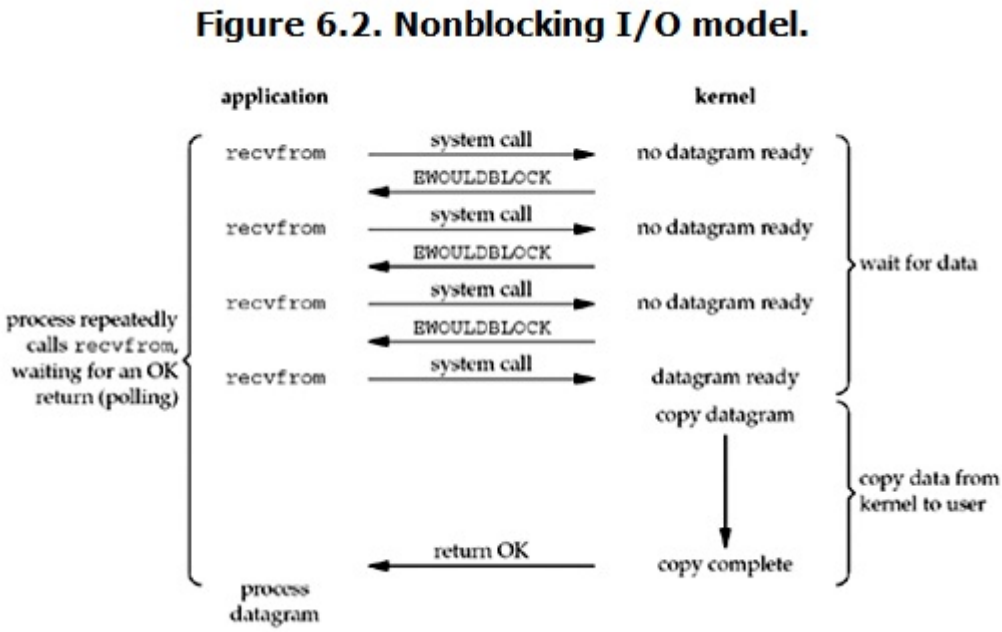
阻塞IO



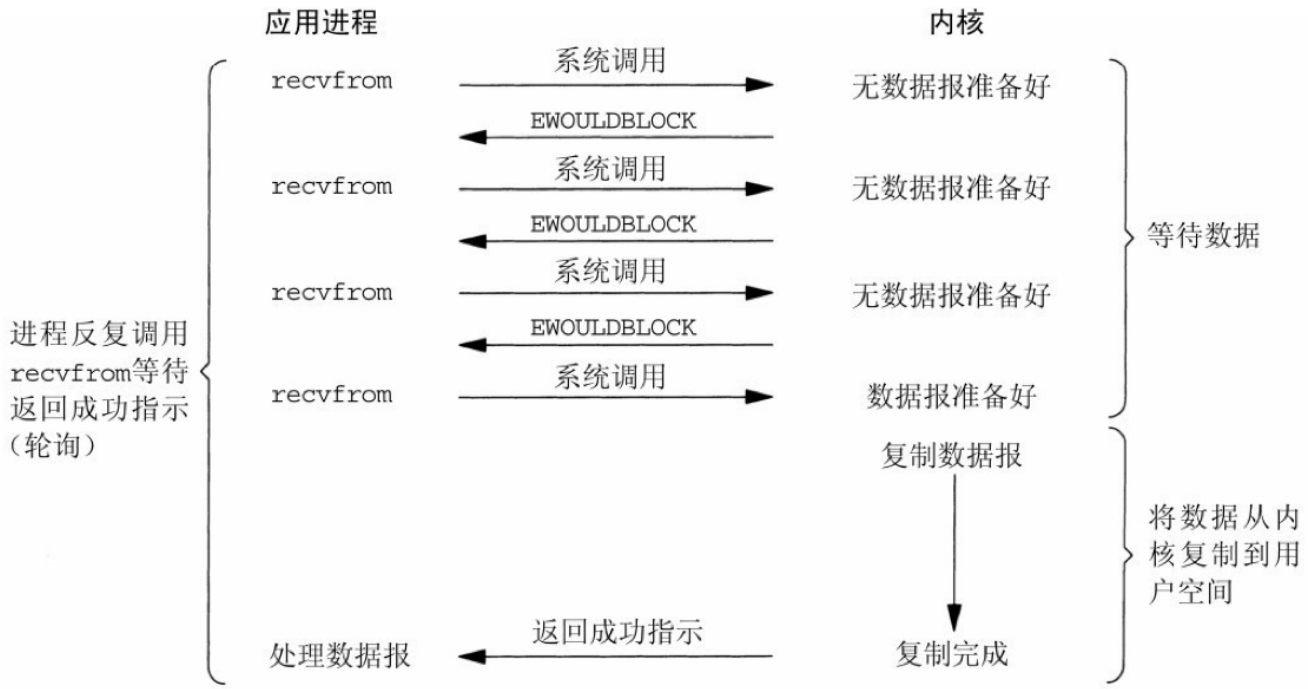
进程等待（阻塞），直到读写完成。（全程等待）



非阻塞IO



进程调用read操作，如果IO设备没有准备好，立即返回ERROR，进程不阻塞。用户可以再次发起系统调用，如果内核已经准备好，就阻塞，然后复制数据到用户空间。
第一阶段数据没有准备好，就先忙别的，等会再来看看。检查数据是否准备好了的过程是非阻塞的。
第二阶段是阻塞的，即内核空间和用户空间之间复制数据是阻塞的。
淘米、蒸饭我不等，我去玩会，盛饭过程我等着你装好饭，但是要等到盛好饭才算完事，这是同步的，结果就是盛好饭。

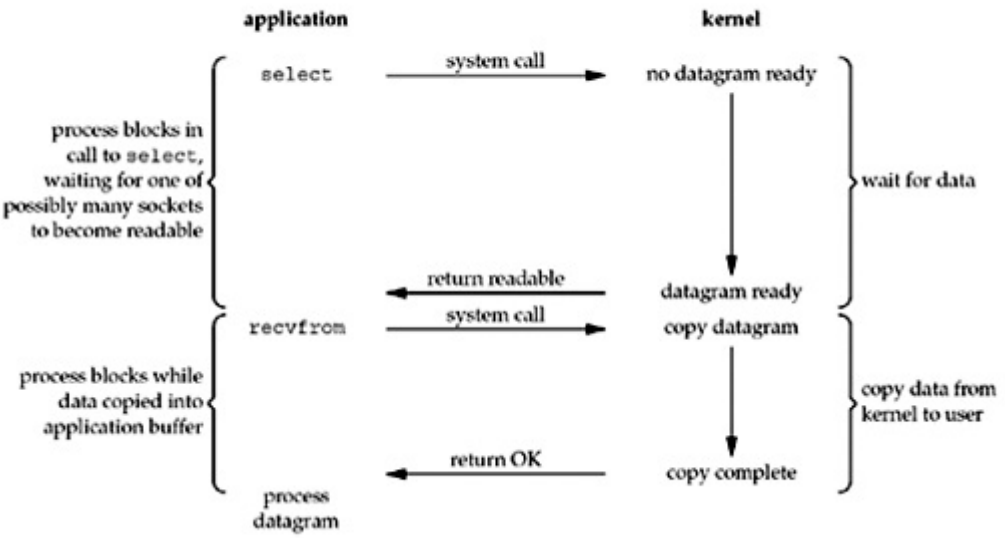


IO多路复用

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。

select几乎所有操作系统平台都支持，poll是对的select的升级。
epoll，Linux系统内核2.5+开始支持，对select和poll的增强，在监视的基础上，增加回调机制。BSD、Mac平台有kqueue，Windows有iocp。

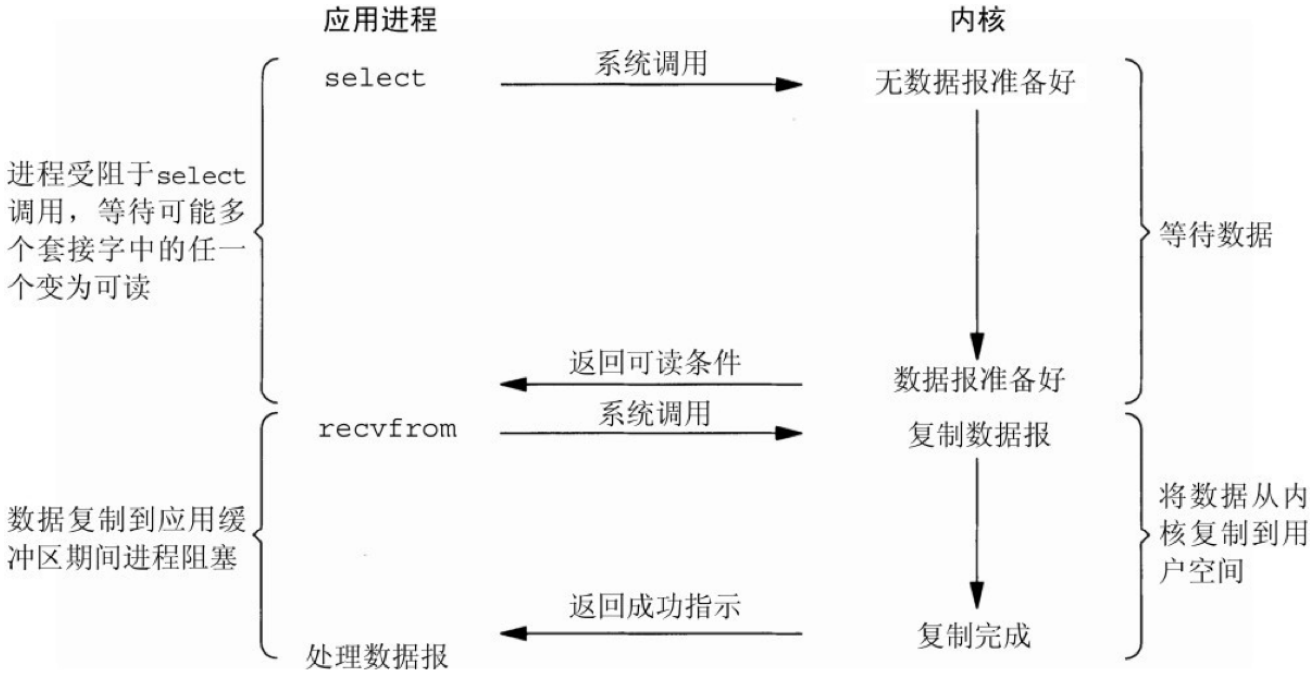
Figure 6.3. I/O multiplexing model.



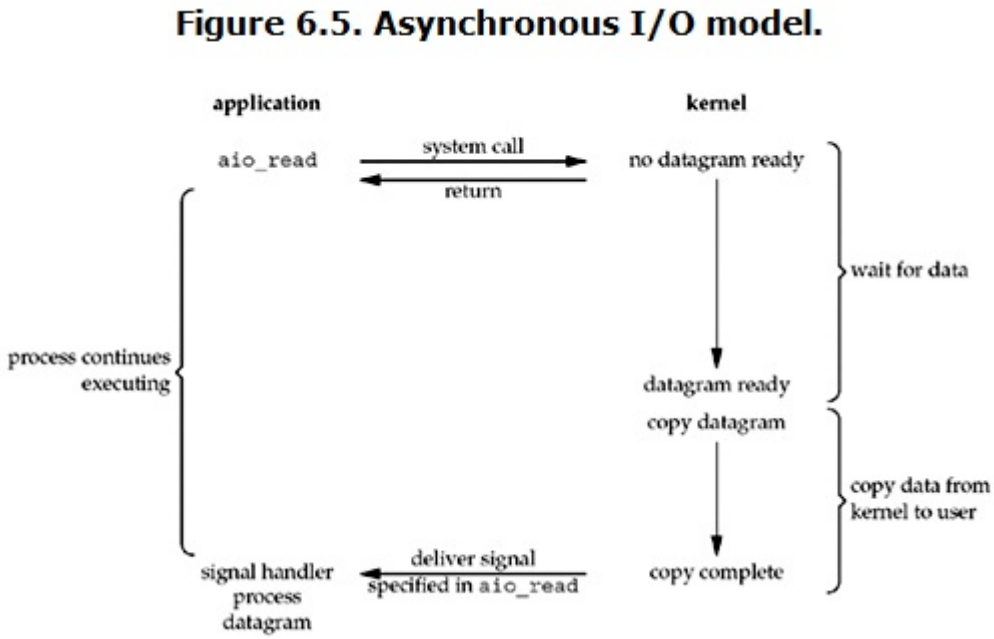
以select为例，将关注的IO操作告诉select函数并调用，进程阻塞，内核“监视”select关注的文件描述符fd，被关注的任何一个fd对应的IO准备好了数据，select返回。再使用read将数据复制到用户进程。

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，需要等，你只好等待大师傅叫。其中一样菜好了，大师傅叫你，说你点的菜有好的了，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。

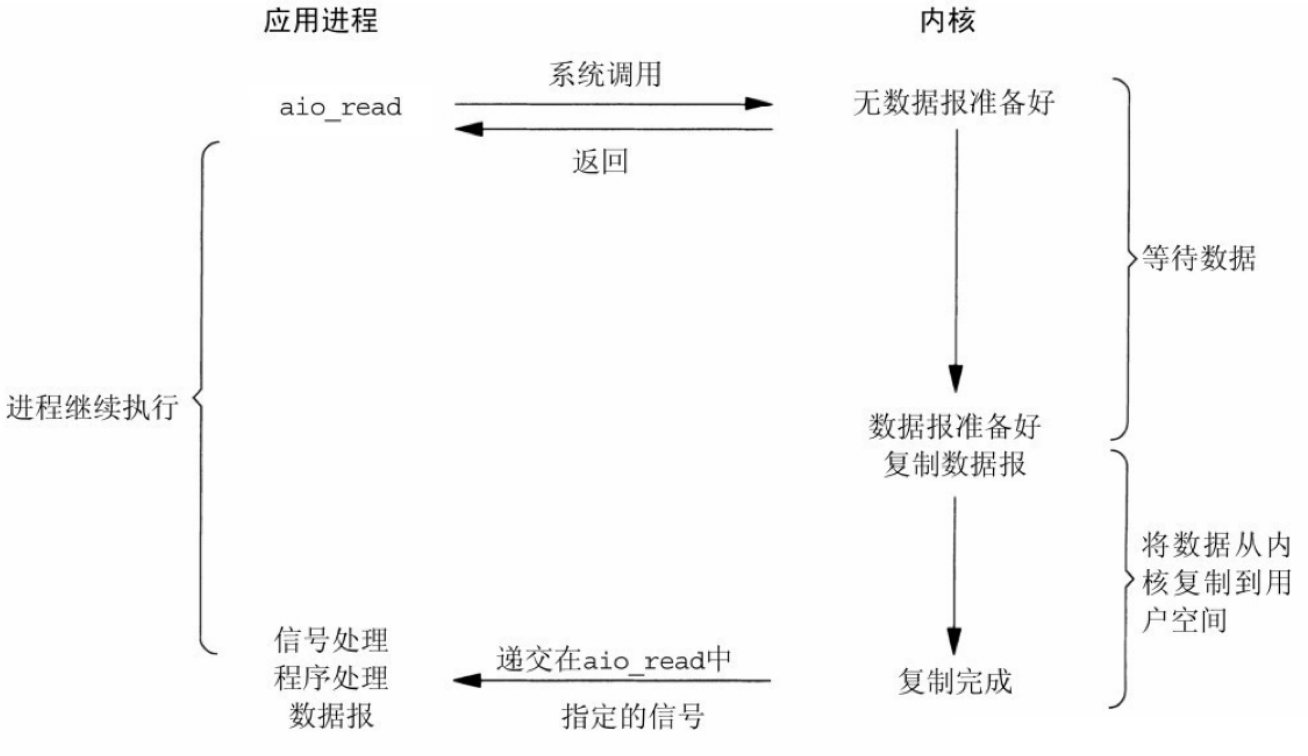
一般情况下，select最多能监听1024个fd（可以修改，但不建议改），但是由于select采用轮询的方式，当管理的IO多了，每次都要遍历全部fd，效率低下。
epoll没有管理的fd的上限，且是回调机制，不需遍历，效率很高。



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



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
  - Linux 2.5+ 支持epoll
  - BSD、Mac支持kqueue
  - 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。部分实现了epoll。它是底层的IO多路复用模块。

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。

epool使用事件通知机制，使用回调机制提高效率。

select/pool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selects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
```

`abstractmethod register(fileobj, events, data=None)`
为selector注册一个文件对象，监视它的IO事件。返回**SelectKey**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输出设备，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) # 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）。

```python
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)
```

实战：IO多路复用群聊软件

将ChatServer改写成IO多路复用的方式

不需要启动多线程来执行socket的accept、recv方法了

```
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)

class ChatServer:
    def __init__(self, ip='127.0.0.1', port=9999):
        self.sock = socket.socket()
        self.addr = ip, port
        self.event = threading.Event()

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

    def start(self):
        self.sock.bind(self.addr)
        self.sock.listen()
        self.sock.setblocking(False)

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

        # 事件监听循环
        threading.Thread(target=self.select, name='selelct', daemon=True).start()

    def select(self):
        # 开始循环
        while not self.event.is_set():
            # 监听注册的对象的事件，发生被关注事件则返回events
            events = self.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)
```

```
    # 回调函数，sock的读事件
    # 形参自定义
    def accept(self, 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 = self.selector.register(conn, selectors.EVENT_READ, self.recv)
        logging.info(key)

    # 回调函数
    def recv(self, conn:socket.socket, mask):
        data = conn.recv(1024)
        data = data.strip()

        if data == b'quit' or data == b'':
            self.selector.unregister(conn)
            conn.close()
            return

        msg = "Your msg = {} ~~~~".format(data.decode()).encode()
        logging.info(msg)

        for key in self.selector.get_map().values():
            print(self.recv) # 当前绑定的
            print(key.data) # 注册时注入的绑定的对象
            print(self.recv is key.data) # 是否一致！！！
            print(self.recv == key.data) # 是否一致?
            if key.data == self.recv:
                key.fileobj.send(msg)

    def stop(self): # 关闭关注的文件对象，关闭selector
        self.event.set()
        fobjs = []
        for fd, key in self.selector.get_map().items():
            fobjs.append(key.fileobj)
        for fobj in fobjs:
            self.selector.unregister(fobj)
            fobj.close()
        self.selector.close()



if __name__ == '__main__':
    cs = ChatServer()
    cs.start()

    while True:
        cmd = input('>>')
        if cmd.strip() == 'quit':
```

```
            logging.info('quit')
            cs.stop()
            break
        print(threading.enumerate())
```

本例只完成基本功能，其他功能如有需要，请自行完成。

特别注意key.data == self.recv