

异步编程

重要概念

同步、异步

阻塞、非阻塞

区别

联系

同步IO、异步IO、IO多路复用

IO两个阶段

IO模型

同步IO

阻塞IO

非阻塞IO

IO多路复用

异步IO

Python 中 IO多路复用

selectors库

练习

异步编程

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。

同步就是我让你打饭，你不打好给我不走开，直到你打饭给了我。

异步就是我让你打饭，你打着，我不等你，但是我会盯着你，你打完，我会过来拿走的。异步并不保证多长时间最终打完饭。

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，还要通过被调用者，使用其它方式通知调用者，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，叫号

异步非阻塞，我要打饭，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

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进程缓冲区阶段

发生IO的时候：

- 1、内核从输入设备读、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

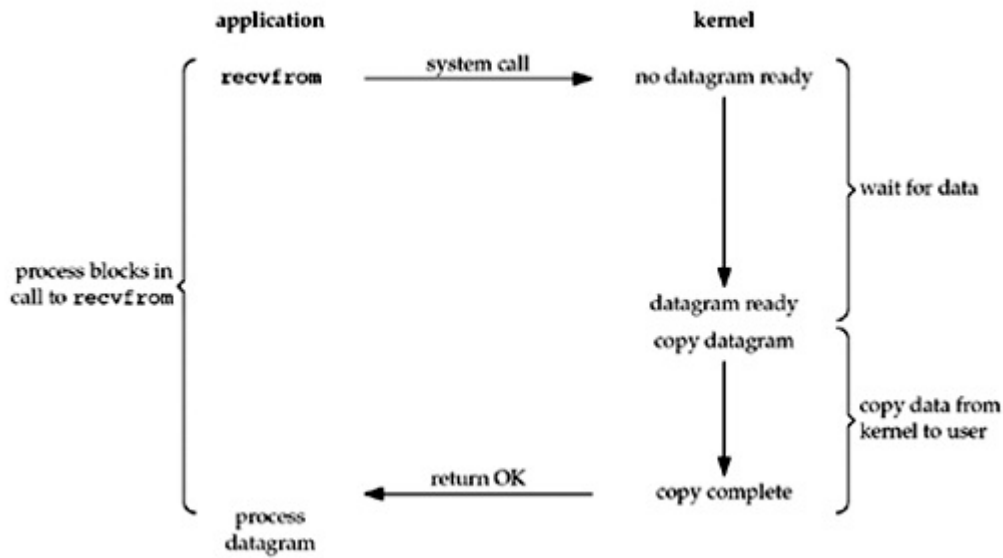
IO模型

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

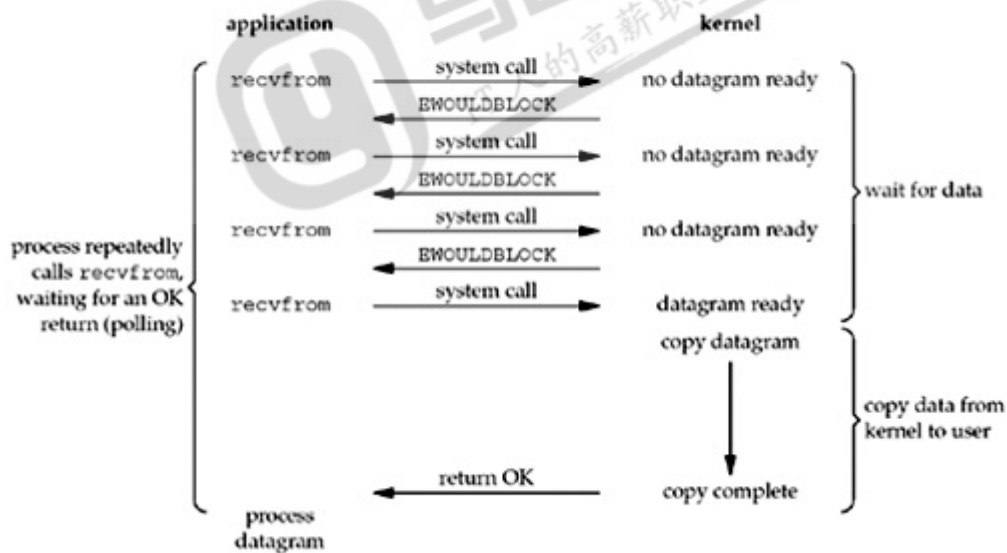
Figure 6.1. Blocking I/O model.



进程等待（阻塞），直到读写完成。（全程等待）
read/write函数

非阻塞IO

Figure 6.2. Nonblocking I/O model.



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。

read/write

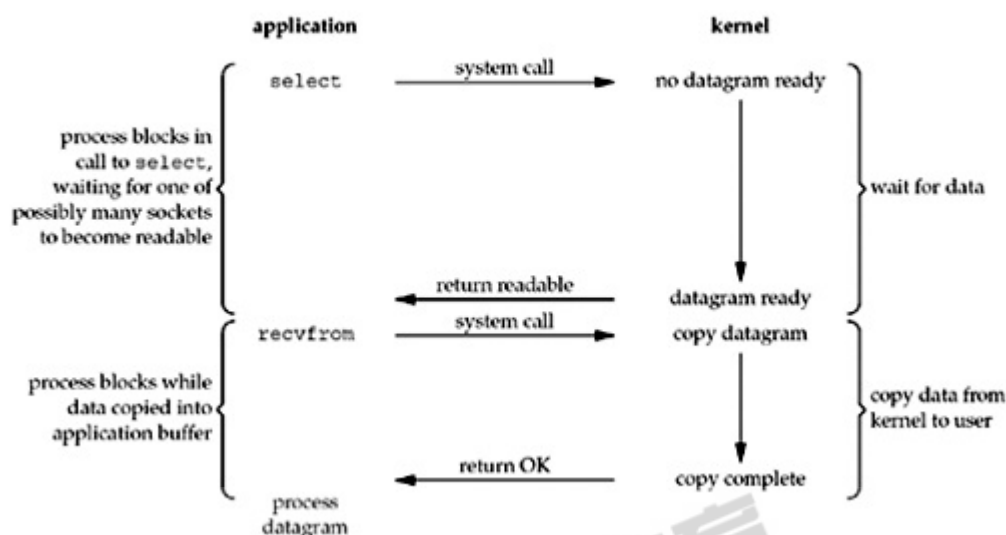
IO多路复用

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。其中一样菜好了，大师傅叫你过来说你点的菜有好了的，你得自己找找看哪一样才好了，请服务员把做好的菜打给你。

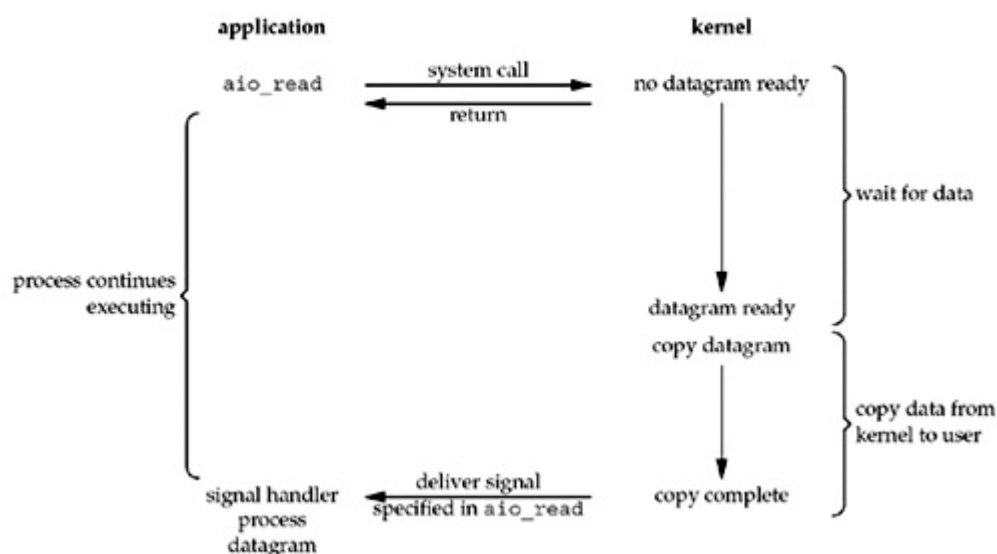
epoll是有菜准备好了，大师傅喊你去几号窗口直接打菜，不用自己找菜了。

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

Figure 6.5. Asynchronous I/O model.



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。

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

selectors库

3.4版本提供这个库，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，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。

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

```
# 使用举例
import selectors
import threading
import socket
import logging

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

# 回调函数，自己定义形参
```

```

def accept(sock:socket.socket, mask):
    """mask: 事件掩码的或值 """
    conn, raddr = sock.accept()
    conn.setblocking(False) # 不阻塞
    # 监视conn这个文件对象
    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 is {}".format(data.decode())
    conn.send(msg.encode())

# 构造缺省性能最优selector
selector = selectors.DefaultSelector()

# 创建Tcp Server
sock = socket.socket()
sock.bind(('0.0.0.0', 9999))
sock.listen()
logging.info(sock)

sock.setblocking(False) # 非阻塞

# 注册文件对象sock关注读事件, 返回SelectorKey
# 将sock、关注事件、data都绑定到key实例属性上
key = selector.register(sock, selectors.EVENT_READ, accept)
logging.info(key)

e = threading.Event()

def select(e):
    while not e.is_set():
        # 开始监视, 等到有文件对象监控事件产生, 返回(key, mask)元组
        events = selector.select()
        print('-'*30)
        for key, mask in events:
            logging.info(key)
            logging.info(mask)
            callback = key.data # 回调函数
            callback(key.fileobj, mask)

threading.Thread(target=select, args=(e,), name='select').start()

def main():
    while not e.is_set():
        cmd = input('>>')
        if cmd.strip() == 'quit':
            e.set()
            fobjs = []
            logging.info('{}'.format(list(selector.get_map().items())))

```

```

        for fd, key in selector.get_map().items(): # 返回注册的项
            print(fd, key)
            print(key.fileobj)
            fobjjs.append(key.fileobj)

    for fobj in fobjjs:
        selector.unregister(fobj)
        fobj.close() # 关闭socket
    selector.close()

if __name__ == '__main__':
    main()

```

练习

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 socket
import threading
import datetime
import logging
import selectors

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): # 启动服务
        self.sock = socket.socket()
        self.addr = (ip, port)

        self.event = threading.Event()
        self.selector = selectors.DefaultSelector() # 创建selector

    def start(self): # 启动监听
        self.sock.bind(self.addr) # 绑定
        self.sock.listen() # 监听
        self.sock.setblocking(False) # 不阻塞
        # 注册
        self.selector.register(self.sock, selectors.EVENT_READ, self.accept)

        threading.Thread(target=self.select, name='selector', daemon=True).start()

    def select(self): # 阻塞
        while not self.event.is_set():
            # 开始监视，等到有文件对象监控事件产生，返回(key, mask)元组
            events = self.selector.select()

```



```

        print('-' * 30)
        for key, mask in events:
            logging.info(key)
            logging.info(mask)
            callback = key.data # 回调函数
            callback(key.fileobj)

    def accept(self, sock:socket.socket): # 多人连接
        conn, addr = sock.accept() # 阻塞
        conn.setblocking(False) # 非阻塞
        # 注册, 监视每一个连接的socket对象
        self.selector.register(conn, selectors.EVENT_READ, self.recv)

    def recv(self, sock:socket.socket): # 接收客户端数据
        data = sock.recv(1024) # 阻塞到数据到来
        if data == b'': # 客户端主动断开, 注销并关闭socket
            self.selector.unregister(sock)
            sock.close()
            return

        msg = "{:%Y/%m/%d %H:%M:%S} {}:{}\n{}\n".format(datetime.datetime.now(),
*sock.getpeername(), data.decode())
        logging.info(msg)
        msg = msg.encode()
        # 群发
        for key in self.selector.get_map().values():
            if key.data == self.recv: # 排除self.accept
                key.fileobj.send(msg)

    def stop(self): # 停止服务
        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()

def main():
    cs = ChatServer()
    cs.start()

    while True:
        cmd = input('>>').strip()
        if cmd == 'quit':
            cs.stop()
            threading.Event().wait(3)
            break
        logging.info(threading.enumerate())

if __name__ == '__main__':

```

```
main()
```

基本完成功能，但是退出机制、异常处理没有加，这个和以前的处理方式一样，请自行完成。

