

# Python 多线程与网络通信

作者:向仔州

延时函数.....	2
time.sleep() 延时函数，秒，毫秒级延时.....	2
Python 多线程.....	3
_thread.start_new_thread(线程函数, (元组)).....	3
外部向线程内部传递参数.....	3
线程冲突，多个线程同时访问同一个变量的问题.....	5
Python 多线程互斥锁使用.....	5
threading.Lock().....	5
互斥锁反复上锁问题.....	7
threading.RLock() 使用.....	7
在类中创建多线程，另一种创建线程的方法.....	8
threading.Thread.join() 当前线程执行完之前阻塞下一个线程执行.....	8
多线程执行完成后线程结果统计实现.....	9
信号量.....	11
acquire 函数使用，信号量可以让线程阻塞，和上面一样，这里用 acquire 函数来实现....	13
release 函数使用，释放信号量.....	14
限定线程数量.....	14
threading.Barrier 使用.....	15
线程通信.....	16
threading.Event()使用.....	16
threading.Event().wait()使用.....	16
另一种线程同步 threading.Condition().....	18
线程流水线控制方法，wait, notify 实际应用.....	20
生产者，消费者模型.....	22
线程池.....	24
time.time 获取线程当前时间，一般用于查询线程执行了多少时间，看看代码效率....	24
线程池主要解决线程加速运行的问题.....	24
threadpool 库.....	24
threadpool.ThreadPool(参数) #参数:线程池支持的最大任务数。 .....	24
threadpool.makeRequests(需要开启多线程的函数名，函数相关参数，回调函数) .....	24
pool.putRequest(...) 将多线程函数请求丢进线程池.....	25
线程定时器，定时时间到触发线程执行.....	25
threading.Timer(延时时间单位秒，时间到执行的函数).....	25
threading.start()启动线程.....	25
Python 网络 TCP/UDP 编程.....	26
socket.socket(协议族参数，套接字类型)//返回对象，创建套接字.....	26

对象.close()//关闭套接字.....	26
对象.sendto(byte 类型 string, (address, port)) .....	26
对象.bind(元祖参数).....	26
对象.recv(bufsize) .....	26
对象.recvfrom(接受多少个 UDP 数据).....	27
UDP 双向通信实验.....	27
Python echo 测试, 用来测试你的网络通信代码延时和硬件网络延时...未完成	
Python 多线程解决 UDP 服务器和客户端, 相互之间随时发随时收.....	30
UDP 实现 TFTP 下载器.....	33
下面讲解 tftp 数据格式.....	34
struct.pack 使用.....	37
struct.unpack 使用.....	39
大端和小端问题.....	39
UDP 广播.....	41
setsockopt(level,optname,value) 函数使用.....	42
TCP 协议使用.....	44
socket.listen(参数) //设置电脑链接数量.....	44
socket.connect((ip 地址和端口)).....	45
TCP 实现服务器和客户端相互循环随时发送数据.....	46
TCP 的 listen 到底能链接多少客户端, 怎么理解 listen.....	48
TCP 商用服务器编写.....	50
TCP 三次握手, 四次挥手原理.....	53
单进程服务器商用案例.....	55
setsockopt(SOL_SOCKET,SO_REUSEADDR,1) //服务器端口重启函数使用.....	55
multiprocessing.Process(进程函数, 函数参数).....	57
单进程实现并发服务器, select 和 epoll 使用.....	59
单进程并发服务器逻辑.....	59
setblocking(False) 默认 accept 是阻塞, 但是设置 False 之后 accept 就是非阻塞....	59
select 函数版本并发服务器.....	63
readable, writeable, exceptionable = select.select(rlist, wlist, xlist, time).....	63
select 监听多个套接字.....	64
epoll 使用, 解决 select 轮询效率低问题.....	67
register(fd 套接字文件描述符, eventmask 允许什么情况下触发).....	69

## 延时函数

time.sleep() 延时函数，秒，毫秒级延时

```
import time #延时需要用的python库
```

```
while True:  
    for i in range(5):  
        print("i = %d" %i)  
        time.sleep(1) #延时单位1秒  
    print("结束执行")  
    break
```

```
i = 0  
i = 1  
i = 2  
i = 3  
i = 4  
结束执行
```

```
import time #延时需要用的python库
```

```
while True:  
    for i in range(5):  
        print("i = %d" %i)  
        time.sleep(0.1) #延时单位100毫秒  
    print("结束执行")  
    break
```

## Python 多线程

```
_thread.start_new_thread(线程函数, (元组))
```

```
import time #延时需要用的python库
import _thread #多线程库

def thread1():#线程函数1
    while True:
        print("thread11111")
        time.sleep(1)

def thread2():#线程函数2
    while True:
        print("thread22222")
        time.sleep(1)

_thread.start_new_thread( thread1, () )#创建线程函数1
_thread.start_new_thread( thread2, () )#创建线程函数2

while True: #这是主线程死循环, 不能取消, 取消多线程程序就结束了, 相当于main死循环
    pass
```

```
thread11111
thread22222
thread11111
thread22222
thread11111
thread22222    死循环, thread1 和 thread2 函数同时执行
```

### 外部向线程内部传递参数

```
import time #延时需要用的python库
import _thread #多线程库

def thread1(name):#线程函数1
    while True:
        print("thread11111")
        print(name)
        time.sleep(1)

def thread2(name):#线程函数2
    while True:
        print("thread22222")
        print(name)
        time.sleep(1)

_thread.start_new_thread( thread1, ("线程1",) )#创建线程1, 向线程1传递元组
_thread.start_new_thread( thread2, ("线程2",) )#创建线程2, 向线程2传递元组

while True: #这是主线程死循环, 不能取消, 取消多线程程序就结束了, 相当于main死循环
    pass
```

```
thread22222  
线程2  
thread11111  
线程1  
thread22222  
线程2  
thread11111  
线程1  
thread22222  
线程2
```

name 变量传入了线程的外部元组

```
import time #延时需要用的python库  
import _thread #多线程库  
  
def thread1(name, num): #线程函数1  
    while True:  
        print("thread11111")  
        print(name)  
        print(num)  
        time.sleep(1)  
  
def thread2(name, num): #线程函数2  
    while True:  
        print("thread22222")  
        print(name)  
        print(num)  
        time.sleep(1)
```

线程接受外部两个变量

```
_thread.start_new_thread( thread1, ("线程1", 10000) )#创建线程1，向线程1传递元组  
_thread.start_new_thread( thread2, ("线程2", 20000) )#创建线程2，向线程2传递元组  
  
while True: #这是主线程死循环，不能取消，取消多线程程序就结束了，相当于main死循环  
    pass  
thread11111  
线程1  
10000  
thread22222  
线程2  
20000
```

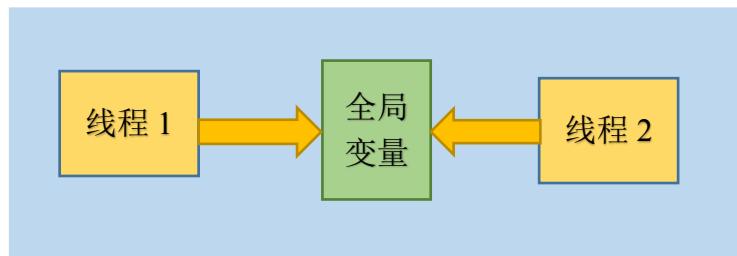
给线程传递字符和数字

所以线程传参可以给元组传递多个元素，看自己设计。

线程冲突，多个线程同时访问同一个变量的问题

```
global_num = 0 #全局变量  
我定义了1个全局  
变量 global_num  
  
def thread1():#线程函数1  
    global global_num #声明外部全局变量  
    while True:  
        global_num = global_num + 1  
        print("thread11111 = %d" %global_num)  
        time.sleep(1)  
  
    def thread2():#线程函数2  
        global global_num #声明外部全局变量  
        while True:  
            global_num = global_num + 1  
            print("thread22222 = %d" %global_num)  
            time.sleep(1)  
  
    _thread.start_new_thread( thread1, () )#创建线程1  
    _thread.start_new_thread( thread2, () )#创建线程2  
  
    while True: #这是主线程死循环，不能取消，取消多线程程序就结束了，相当于main死循环  
        pass
```

```
thread11111 = 3  
thread22222 = 4  
thread22222 = 5  
thread11111 = 6  
thread22222 = 7  
thread11111 = 8
```



两个线程同时访问 global\_num 全局变量，在运行的时候貌似没有出现什么问题，数据有条不紊的增加着，但是随着执行次数越多，就会出现全局变量数据错误，乱变化的情况，这里运行时间太少表现不出来，在我《QT4.7Mutilthread》文档多线程互斥中也讲到了多线程同时访问 1 个变量的问题。

## Python 多线程互斥锁使用

```
import time #延时需要用的python库  
import _thread #多线程库  
import threading #这也是多线程库  
  
global_num = 0 #全局变量  
mutex = threading.Lock() #创建1个锁
```

需要定义一个线程锁变量 `threading.Lock()`

```

def thread1():
    global global_num
    while True:
        if mutex.acquire(): #锁住成功, 如果没有锁住成功线程卡在这里
            global_num = global_num + 1
            print("thread111 = %d" %global_num)
            mutex.release() #释放锁
            time.sleep(1)

def thread2():
    global global_num
    while True:
        global_num = global_num + 1
        print("thread222 = %d" %global_num)
        time.sleep(1)

_thread.start_new_thread(thread1, ())
_thread.start_new_thread(thread2, ())

```

`while True:` #这是主线程死循环, 不能取消, 取消多线程程序就结束了, 相当于main死循环  
`pass`

```

thread222 = 1
thread111 = 2
thread222 = 3
thread111 = 4

```

```

def thread1():
    global global_num
    while True:
        if mutex.acquire(): #锁住成功, 如果没有锁住成功线程卡在这里
            global_num = global_num + 1
            print("thread111 = %d" %global_num)
            mutex.release() #释放锁
            time.sleep(1)

def thread2():
    global global_num
    while True:
        if mutex.acquire(): #锁住成功, 如果没有锁住成功线程卡在这里
            global_num = global_num + 1
            print("thread222 = %d" %global_num)
            time.sleep(1)

```

```

thread111 = 1
thread222 = 2

```

你看卡住了, 所以两个线程访问同 1 个全局变量, 一定要用同一把锁才安全。

线程 1 上锁

线程 1 解锁

因为线程 1 每次访问全局变量上锁后  
解锁了的, 所以线程 2 不会被卡死

如果线程 2 上  
锁之后

线程 2 不解  
锁, 那么用  
同一个变量  
锁的线程 1  
就会卡在  
acquire 这里

## 互斥锁反复上锁问题

```
def thread1():
    global global_num
    while True:
        if mutex.acquire(): #锁住成功, 如果没有锁住成功线程卡在这里
            if mutex.acquire(): #锁住成功, 如果没有锁住成功线程卡在这里
                global_num = global_num + 1
                print("thread111 = %d" %global_num)
                time.sleep(1)
                mutex.release() #释放锁
            mutex.release() #释放锁

def thread2():
    global global_num
    while True:
        global_num = global_num + 1
        print("thread222 = %d" %global_num)
        time.sleep(1)

thread222 = 1
thread222 = 2
thread222 = 3
```

线程 1 被锁住了就只有执行线程 2

第 1 次上锁后, 没有释放该锁, 那么第 2 次上锁线程就卡在这里, 再也无法去执行下面的释放锁的函数了

## threading.RLock() 使用

```
global_num = 0 #全局变量
mutex = threading.RLock() #创建1个RLock锁

def thread1():
    global global_num
    while True:
        if mutex.acquire(): #锁住成功, 如果没有锁住成功线程卡在这里
            if mutex.acquire(): #锁住成功, 如果没有锁住成功线程卡在这里
                global_num = global_num + 1
                print("thread111 = %d" %global_num)
                time.sleep(1)
                mutex.release() #释放锁
            mutex.release() #释放锁

def thread2():
    global global_num
    while True:
        global_num = global_num + 1
        print("thread222 = %d" %global_num)
        time.sleep(1)
```

RLock 可以解决单个线程死锁问题

比如单个线程多次加锁

```
thread111 = 5
thread222 = 6
thread111 = 7
thread222 = 8
thread111 = 9
thread222 = 10
```

你看重复加锁的线程 1 和没加锁的线程 2 正常执行了

## 在类中创建多线程，另一种创建线程的方法

```
import time #延时需要用的python库
import _thread #多线程库
import threading #这也是多线程库

class mythread1(threading.Thread):#类创建线程要继承至父类threading.Thread
    def run(self):#run是父类自带的函数，这里是重写
        while True:
            print("mythread class 11111....")
            time.sleep(1)

class mythread2(threading.Thread):#类创建线程要继承至父类threading.Thread
    def run(self):#run是父类自带的函数，这里是重写
        while True:
            print("mythread class 22222....")
            time.sleep(1)

objmy1 = mythread1() #类创建1个对象就是1个线程
objmy1.start()
objmy2 = mythread2() #为了区分出类线程，用另外个类创建个对象
objmy2.start()

while True: #这是主线程死循环，不能取消，取消多线程程序就结束了，相当于main死循环
    pass
```

用子类继承父类的线程 `threading.Thread`, 然后重新 `run` 函数

然后这个子类就具备了线程的功能，这个子类创建一个对象，就是一个线程。创建两个对象就是两个线程。

```
mythread class 11111....
mythread class 22222....
mythread class 11111....
mythread class 22222....
mythread class 11111....
mythread class 22222....
```

运行正常

`threading.Thread.join()` 当前线程执行完之前阻塞下一个线程执行

```
import time #延时需要用的python库
import _thread #多线程库
import threading #这也是多线程库

class mythread1(threading.Thread):#类创建线程要继承至父类threading.Thread
    def run(self):#run是父类自带的函数，这里是重写
        while True:
            print("mythread class 11111....")
            time.sleep(1)

class mythread2(threading.Thread):#类创建线程要继承至父类threading.Thread
    def run(self):#run是父类自带的函数，这里是重写
        while True:
            print("mythread class 22222....")
            time.sleep(1)
```

```

print("thread1111 执行")
objmy1 = mythread1() #类创建1个对象就是1个线程
objmy1.start()
objmy1.join()#主进程阻塞等待先创建的线程1执行完了才执行下面线程2

print("thread2222 执行")
objmy2 = mythread2() #为了区分出类线程,用另外个类创建个对象
objmy2.start()

while True: #这是主线程死循环,不能取消,取消多线程程序就结束了,相当于main死循环
    pass

```

执行 join 函数, 让主进程阻塞在这里, 等待线程 1 执行完了才能执行线程 2

```

thread1111 执行
mythread class 11111....
mythread class 11111....
mythread class 11111....

```

你看, 只要线程 1 在死循环执行, 线程 2 是无法

开启的, 这个 join 就可以用来做线程控制, 先执行什么线程后又执行什么线程。

多线程执行完成后线程结果统计实现

```

import time #延时需要用的python库
import _thread #多线程库
import threading #这也是多线程库

class mythread1(threading.Thread):#类创建线程要继承至父类threading.Thread
    def run(self):#run是父类自带的函数, 这里是重写
        while True:
            print("mythread class 11111....")
            time.sleep(1)

class mythread2(threading.Thread):#类创建线程要继承至父类threading.Thread
    def run(self):#run是父类自带的函数, 这里是重写
        while True:
            print("mythread class 22222....")
            time.sleep(1)

lst=[] #创建变量当作线程集合
print("thread1111 执行")
objmy1 = mythread1() #类创建1个对象就是1个线程
objmy1.start()
lst.append(objmy1)#加入线程进列表
print("thread2222 执行")
objmy2 = mythread2() #为了区分出类线程, 用另外个类创建个对象
objmy2.start()
lst.append(objmy2)#加入线程进列表

for l in lst:
    l.join()
    print("线程集合")

while True: #这是主线程死循环, 不能取消, 取消多线程程序就结束了, 相当于main死循环
    pass

```

这是死循环

这是死循环

建立个普通列表

把创建的线程放入列表

在列表中的线程执行完之前, 这里会阻塞

```
thread1111 执行  
mythread  class 11111....  
thread2222 执行  
mythread  class 22222....  
mythread  class 11111....  
mythread  class 22222....
```

你看一直没有出现‘线程集合’，因为是死循环，线程不可能结束

```
class mythread1(threading.Thread):#类创建线程要继承至父类threading.Thread  
    def run(self):#run是父类自带的函数，这里是重写  
        for i in range(3):  
            print("mythread  class 11111....")
```

把线程改成  
for 循环 3 次

```
class mythread2(threading.Thread):#类创建线程要继承至父类threading.Thread  
    def run(self):#run是父类自带的函数，这里是重写  
        for i in range(3):  
            print("mythread  class 22222....")  
            time.sleep(1)
```

把线程改成  
for 循环 3 次

```
lst=[] #创建变量当作线程集合
```

```
print("thread1111 执行")  
objmy1 = mythread1() #类创建1个对象就是1个线程  
objmy1.start()  
lst.append(objmy1) #加入线程进列表
```

```
print("thread2222 执行")  
objmy2 = mythread2() #为了区分出类线程，用另外个类创建个对象  
objmy2.start()  
lst.append(objmy2) #加入线程进列表
```

```
for l in lst:  
    l.join()  
    print("线程集合")
```

线程循环 3 次执行完了，这里的 join 就不会阻塞

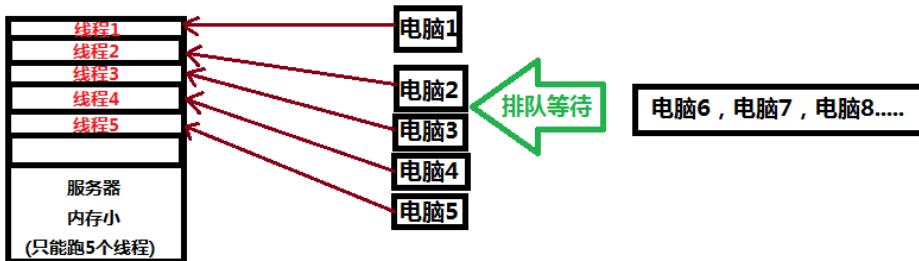
```
while True: #这是主线程死循环，不能取消，取消多线程程序就结束了，相当于main死循环  
    pass
```

```
thread1111 执行  
mythread  class 11111....thread2222 执行  
  
mythread  class 22222....  
mythread  class 22222....  
mythread  class 11111....  
mythread  class 22222....  
mythread  class 11111....  
线程集合  
线程集合
```

这样做好处就是可以汇总多线程的信息进行最后处理

## 信号量

信号量限制线程数量



打个比方    如果是web服务器，  
1个网页请求我分配1  
个线程去处理

这时候其余的电脑只  
有等待前面5台电脑  
处理完了，才能接受  
服务器的数据，  
程，那必须把这5个电  
脑的网页数据处理好

这就是服务器处理请求的能力，叫做服务器压力测试，因为服务器内存不是无限大，所以线程也不是无限的增加，所以得看自己怎么操作

```
import threading
import time
import _thread # _thread.start_new_thread必须要_thread库

def thread1():
    while True:
        print("thread1111...")
        time.sleep(1)

def thread2():
    while True:
        print("thread2222...")
        time.sleep(1)

def thread3():
    while True:
        print("thread3333...")
        time.sleep(1)

def thread4():
    while True:
        print("thread4444...")
        time.sleep(1)

_thread.start_new_thread(thread1, ())
_thread.start_new_thread(thread2, ())
_thread.start_new_thread(thread3, ())
_thread.start_new_thread(thread4, ())
```

常规情况我们开  
4个线程

常规情况我们开  
4个线程

```
while True:
    pass
```

thread1111...
thread2222...
thread3333...
thread4444...
thread2222...
thread1111...
thread3333...
thread4444... 4个线程同时都在循环执行

下面加入信号量

```
import threading
import time
import _thread # _thread.start_new_thread必须要有_thread库

sem = threading.Semaphore(2) # 创建2个信号量

def thread1():
    with sem:
        while True:
            print("thread1111...")
            time.sleep(1)

def thread2():
    with sem:
        while True:
            print("thread2222...")
            time.sleep(1)

def thread3():
    with sem:
        while True:
            print("thread3333...")
            time.sleep(1)

def thread4():
    with sem:
        while True:
            print("thread4444...")
            time.sleep(1)

_thread.start_new_thread(thread1, ())
_thread.start_new_thread(thread2, ())
_thread.start_new_thread(thread3, ())
_thread.start_new_thread(thread4, ())

while True:
    pass
```

我每个线程都消耗 1 个信号量

注意信号量用的 with 语  
法，那么下面的 while 就需  
要 Tab，注意语法格式

thread333...  
thread2222...  
thread2222...  
thread333...  
thread2222...  
thread333...

开机后，随机两个线程先执行  
了 with sem;那么  
threading.Semaphore(2)定义的两  
个信号量就被两个线程消耗  
了，这时候必须等这两个线程  
执行完了，其它线程才能执  
行，这里先消耗信号量 sem 的  
是线程 2 和线程 3

如果看起不够直观我们用 for 循环来测试

```
def thread1():
    with sem:
        for x in range(1, 3):
            print("thread1111...")
            time.sleep(1)

def thread2():
    with sem:
        for x in range(1, 3):
            print("thread2222...")
            time.sleep(1)

def thread3():
    with sem:
        for x in range(1, 3):
            print("thread3333...")
            time.sleep(1)

def thread4():
    with sem:
        for x in range(1, 3):
            print("thread4444...")
            time.sleep(1)
```

线程 2, 线程 3, 消耗掉了 sem,  
所以先执行, 线程 2, 线程 3 循环  
结束后, 剩下的线程 1, 线程 4 再  
执行。这就是信号量控制线程执行  
顺序的一种方法, 让 4 个线程不能  
同时执行, 而是按照信号量定义的  
规则来依次执行

```
thread3333...
thread2222...
thread3333...
thread2222...
thread1111...
thread4444...
thread1111...
thread4444...
```

acquire 函数使用, 信号量可以让线程阻塞, 和上面一样, 这里用 acquire 函数来实现

```
import threading
import time
import _thread # _thread.start_new_thread必须要_thread库

sem = threading.Semaphore(2) # 创建2个信号量
#Semaphore(value=1) # value设置是内部维护的计数器的大小, 默认为1.

def thread1():
    while True:
        sem.acquire() #每当调用acquire()时, 内置计数器-1, 直到为0的时候阻塞
        print("thread1111...")
        time.sleep(1)

def thread2():
    while True:
        print("thread2222...")
        time.sleep(1)

_thread.start_new_thread(thread1, ())
_thread.start_new_thread(thread2, ())

while True:
    pass
```

```
thread1111...
thread2222...
thread1111...
thread2222...
thread2222...
thread2222...
thread2222...
```

你看线程 1 将两个信号量使用完了，就阻塞了，后面线程 2 一直在跑

release 函数使用，释放信号量

```
sem = threading.Semaphore(2) # 创建2个信号量
# Semaphore(value=1) # value设置是内部维护的计数器的大小，默认为1.

def thread1():
    while True:
        sem.acquire() # 每当调用acquire()时，内置计数器-1，直到为0的时候阻塞
        print("thread1111...")
        time.sleep(1)

def thread2():
    while True:
        sem.release() #每当调用release()时，内置计数器+1，并让某个线程的acquire()从阻塞变为不阻塞
        print("thread2222...")
        time.sleep(1)

_thread.start_new_thread(thread1, ())
_thread.start_new_thread(thread2, ())
```

我每次线程 2 释放了线程 1 消耗的信号量，线程 1 就不会阻塞了，因为线程 1 信号量永远没有减到 0.

```
thread1111...
thread2222...
thread1111...
thread2222...
thread1111...
thread2222...
thread1111...
thread2222...
```

## 限定线程数量

```
def thread1():
    print(threading.current_thread().name, "start")
    time.sleep(1)
    print(threading.current_thread().name, "end")

for x in range(5):
    threading.Thread(target=thread1).start() # 这也是创建线程的一种方式

while True:
    pass
```

这种传统的让 5 个线程运行

```
Thread-1 start  
Thread-2 start  
Thread-3 start  
Thread-4 start  
Thread-5 start  
Thread-5 end  
Thread-3 end  
Thread-1 end  
Thread-4 end  
Thread-2 end
```

五个线程同时运行

### threading.Barrier 使用

Barrier 类是设置了一个线程数量障碍，当等待的线程到达了这个数量就会唤醒所有的等待线程

```
import threading  
import time  
import _thread # _thread.start_new_thread必须要_thread库  
  
bar = threading.Barrier(2) # 必须有2个以上的线程才能执行  
  
def thread1():  
    print(threading.current_thread().name, "start")  
    time.sleep(1)  
    bar.wait()  
    print(threading.current_thread().name, "end")  
  
  
for x in range(3):  
    threading.Thread(target=thread1).start() # 这也是创建线程的一种方式  
  
while True:  
    pass
```

```
Thread-1 start  
Thread-2 start  
Thread-3 start  
Thread-1 end  
Thread-2 end
```

这里的意思就是必须两个线程为1组才能执行，你创建3个线程，就只有1,2线程是一组，能继续执行，线程3是无法执行的，除非有线程4

## 线程通信

threading.Event() 使用  
threading.Event().wait() 使用

```
import threading  
import time  
import _thread # _thread.start_new_thread 必须要 _thread 库
```

```
def xzzevent():  
    e = threading.Event() # 创建1个事件对象  
  
    def go():  
        e.wait() # 线程会在这里等待阻塞  
        print("go")  
    threading.Thread(target=go).start() # 创建go函数为线程  
    return e  
  
t = xzzevent() # 把创建的e对象给外部t  
print("xxxxxxxx")  
time.sleep(3)  
t.set() # 执行set 才能让go函数执行
```

def xxx():  
 def zzz():  
 /...程序.../  
 这种就叫  
 函数嵌套

e.wait 就是让线程在这  
里等待，外部有某个程序  
间接或者直接使用 e.set，  
那么这里就会停止等待  
向下执行

这种线程夺一下跳一下的方法就是事件触发，有事件线程才执行，没有事件线程就等待阻塞。

XXXXZZZZZZ

go 过了 3 秒之后，t.set()类似 threading.Event().set() 执行后，go 程序才打印

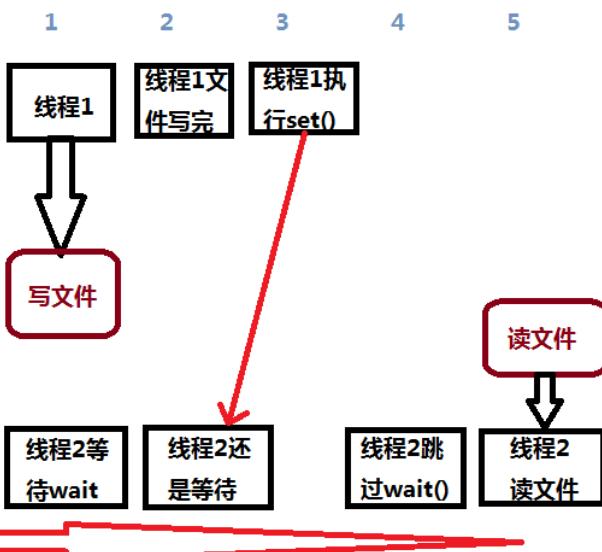
创建个threading.Event()

对象给线程1和线程2

线程1得到  
的是set()

线程2得到  
的是wait()

时间线



这就是一个线程做事件触发，一个线程做事件响应的事件驱动方式

下面说下 `e.wait()` 也就是 `threading.Event().wait()` 的使用次数

```
def xzzevent():
    e = threading.Event() # 创建1个事件对象

    def go():
        for x in range(5):
            e.wait() # 线程会在这里等待阻塞
            print(x, "go")
    threading.Thread(target=go).start() # 创建go函数为线程
    return e
```

```
t = xzzevent() # 把创建的e对象给外部t
```

```
while True:
    print("xxxxxxxx")
    time.sleep(1)
    t.set() # 执行set 才能让go函数执行
```

```
XXXXXXXXX
0 go
1 go
2 go
3 go
4 go
```

我发现 go 程序瞬间就执行完了，不是 1 秒执行 1 次

我想这个线程每 1  
秒 t.set 执行 1 次

```
def xzzevent():
    e = threading.Event() # 创建1个事件对象

    def go():
        for x in range(5):
            e.wait() # 线程会在这里等待阻塞
            print(x, "go")
    threading.Thread(target=go).start() # 创建go函数为线程
    return e
```

```
t = xzzevent() # 把创建的e对象给外部t
```

```
while True:
    print("xxxxxxxx")
    time.sleep(1)
    t.set() # 执行set 才能让go函数执行
```

这就是 wait 被 set 执  
行一次后，wait 就会  
清 0，再也没有阻塞  
线程功能了。

```

def xzzevent():
    e = threading.Event() # 创建1个事件对象

    def go():
        for x in range(5):
            e.wait() # 线程会在这里等待阻塞
            e.clear() # 清除wait(), 让wait()有再次阻塞的功能
            print(x, "go")
    threading.Thread(target=go).start() # 创建go函数为线程
    return e

t = xzzevent() # 把创建的e对象给外部t

while True:
    print("xxxxxxxx")
    time.sleep(1)
    t.set() # 执行set 才能让go函数执行

```

```

xxxxxxxx
xxxxxxxx
0 go
1 go
xxxxxxxx
xxxxxxxx
2 go
xxxxxxxx
3 go
xxxxxxxx
4 go
xxxxxxxx

```

你看主进程和线程每秒执行一次，这样 wait() 就达到每秒阻塞线程 1 次  
另一种线程同步 `threading.Condition()`

```

def thread1():
    for x in range(5):
        print(x, "thread111")
        time.sleep(1)

def thread2():
    for x in range(5):
        print(x, "thread222")
        time.sleep(1)

threading.Thread(target=thread1).start()
threading.Thread(target=thread2).start()

while True:
    pass

```

```

0 thread111
0 thread222
1 thread111
1 thread222
2 thread111
2 thread222
3 thread222
3 thread111
4 thread222
4 thread111

```

这是常规的多线程执行方法，但是我现在想线程 1 先执行完，然后执行线程 2，线程 2 执行完又回来执行线程 1。

```

cond = threading.Condition()

def thread1():
    with cond:# 条件变量，等待通知
        for x in range(10):
            print(x, "thread111")
            time.sleep(1)
            if x == 5:
                cond.wait() # 让线程1不要执行后面的循环，等待在这里

def thread2():
    with cond: # 条件变量，等待通知
        for x in range(5):
            print(x, "thread222")
            time.sleep(1)
    cond.notify() # 通知另外一个使用相同cond条件变量的线程执行

threading.Thread(target=thread1).start()
threading.Thread(target=thread2).start()

while True:
    pass

```

0 thread111  
1 thread111  
2 thread111  
3 thread111  
4 thread111  
5 thread111  
0 thread222  
1 thread222  
2 thread222  
3 thread222  
4 thread222  
6 thread111  
7 thread111  
8 thread111  
9 thread111

with cond:  
/\*\*\*代码段\*\*\*/

这段代码都会被with cond:这个对象管  
也就是代码被threading.Condition管辖

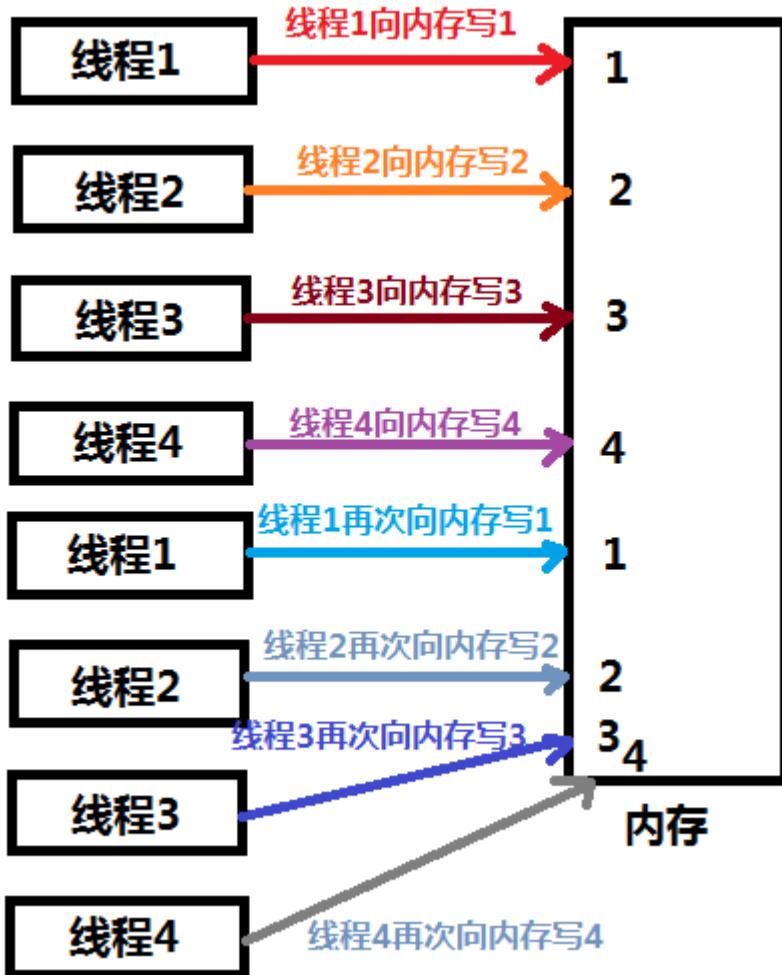
with cond:  
线程1  
当执行到wait()  
线程1等待，让相同对象  
cond的其它线程执行

with cond:  
线程2  
线程2执行  
当线程2执行了  
cond.notify()

with cond:  
线程1  
唤醒线程1继续接着执  
行

这也是线程同步

## 线程流水线控制方法，wait, notify 实际应用



这就是线程流水线，每个线程依次向同一个内存或者文件写数据，而且写数据的顺序必须按照线程的顺序来。

```
def thread1():

    for x in range(10):
        print(x, "thread111")
        time.sleep(1)

def thread2():

    for x in range(5):
        print(x, "thread222")
        time.sleep(1)

threading.Thread(target=thread1).start()
threading.Thread(target=thread2).start()

while True:
    pass
```

```
0 thread111
0 thread222
1 thread111
1 thread222
2 thread222
2 thread111
3 thread111
3 thread222
4 thread222
4 thread111
5 thread111
```

你看我们常规创建线程都是乱序的，有时候线程 1 先执行，有时候线程 2 先执行

```

def thread1():
    with cond:
        for x in range(10):
            print(x, "thread111")
            time.sleep(1)
            cond.wait() # 等待
            cond.notify() # 通知

```

线程 1 先执行, 第一次循环  
打印 0, 然后就等待

通知线程 2 执行

```

def thread2():
    with cond:
        for x in range(5):
            print(x, "thread222")
            time.sleep(1)
            cond.notify() # 通知
            cond.wait() # 等待

```

线程 2 执行第 1 次  
循环也是打印 0

通知线程 1, 线程 2 阻塞

```

threading.Thread(target=thread1).start()
threading.Thread(target=thread2).start()

```

```

0 thread111
0 thread222
1 thread111
1 thread222
2 thread111
2 thread222
3 thread111
3 thread222
4 thread111

```

这样线程 1, 线程 2 都是打印的 0, 我想的是线程 1 先执行打印 0, 那么线程 2 后执行就打印 2, 线程 1 再执行打印 3, 线程 2 打印 4, 这样按照线程流水线控制图的方式执行。

```

def thread1():
    with cond:
        for x in range(10):
            print(x, "thread111")
            time.sleep(1)
            cond.notify() # 通知
            cond.wait() # 等待

```

调换了 wait 和 notify 顺序

```

def thread2():
    with cond:
        for x in range(5):
            print(x, "thread222")
            time.sleep(1)
            cond.wait() # 等待
            cond.notify() # 通知

```

调换了 wait 和 notify 顺序

```

threading.Thread(target=thread2).start()
threading.Thread(target=thread1).start()

```

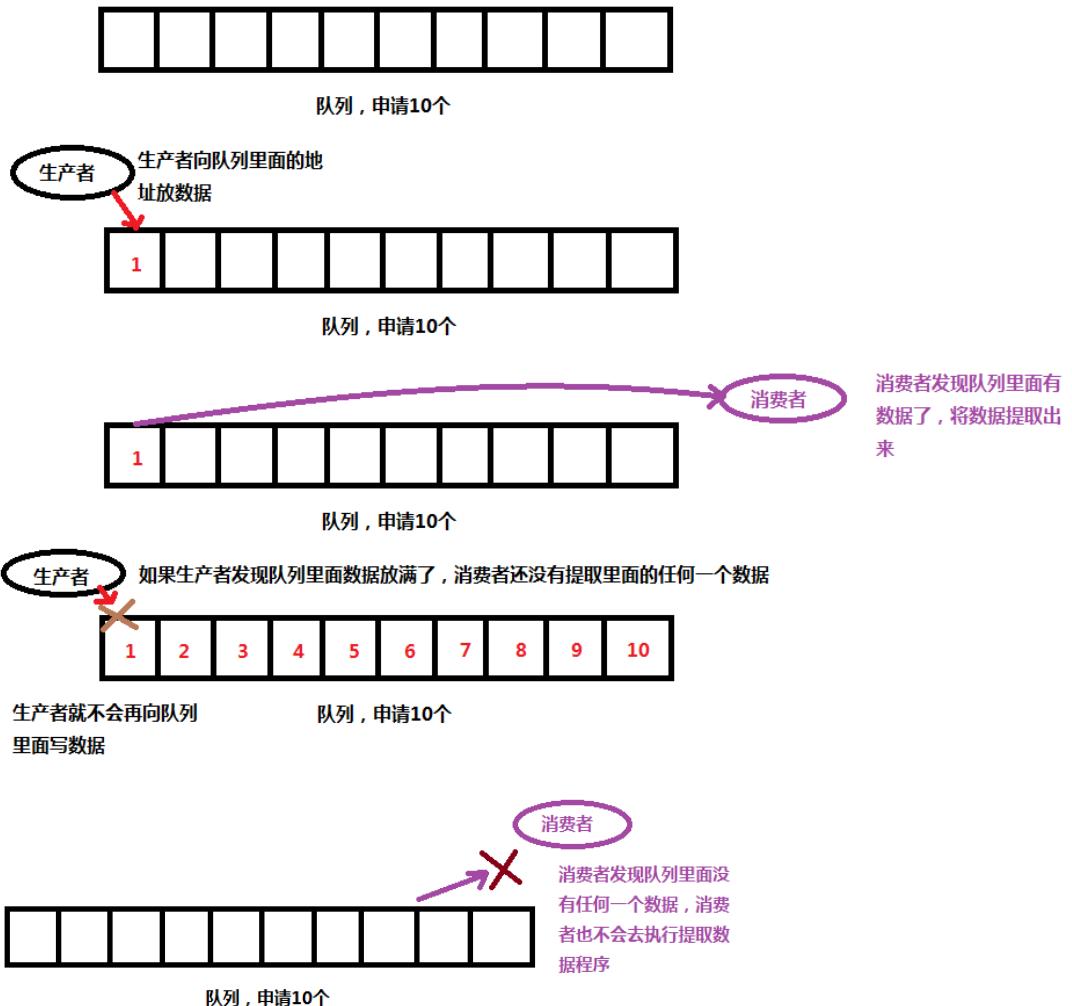
```

0 thread222
0 thread111
1 thread222
1 thread111
2 thread222
2 thread111

```

线程 2 改成先执行

## 生产者，消费者模型



queue 队列库

```

import threading
import time
import queue
import random

myqueue = queue.Queue(10)
# 填入10表示创建队列最大容量，也就是10个队列
# 如果填0就是队列无限大，你内存好大，你队列就可以增加好大

class createThread(threading.Thread): # 这是生产(生成)数据的类
    def __init__(self, index, myqueue):
        #接受初始化传入的全局队列，和给队列定义的index下标
        threading.Thread.__init__(self) # 调用父类初始化函数
        self.index = index # 索引
        self.myqueue = myqueue # 全局队列地址传给局部队列

    def run(self):
        while True:
            time.sleep(3) # 3秒钟创建一个数据
            num = random.randint(100, 200) # 生成100~200之间随机数
            self.myqueue.put("物品编号" + str(self.index) + "物品数据" + str(num))
            #局部队列有了全局队列地址后，可以间接用全局队列的put向全局队列写入数据，

            print("生产者编号" + str(self.index) + "数据" + str(num))
            self.myqueue.task_done() # 完成任务

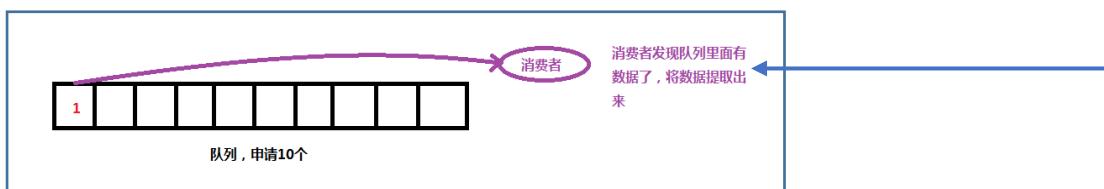
```

```

class comsumption(threading.Thread): # 这是消费类，也就是提取数据类
    def __init__(self, index, myqueue):
        #接受初始化传入的全局队列，和给队列定义的index下标
        threading.Thread.__init__(self) # 调用父类初始化函数
        self.index = index # 索引
        self.myqueue = myqueue # 全局队列地址传给局部队列

    def run(self):
        while True:
            time.sleep(1)
            item = self.myqueue.get() # 抓取全局队列里面所有数据
            if item is None: # 如果item为空None 退出循环
                break
            print("客户索引", self.index, "买到物品", item)
            #获取全局队列里面index和数据
            self.myqueue.task_done() # 完成任务

```



```

createThread(1, myqueue).start() # 创建1个生产者，生产数据
createThread(2, myqueue).start() # 创建2个生产者，生产数据
createThread(3, myqueue).start() # 创建3个生产者，生产数据

```

```

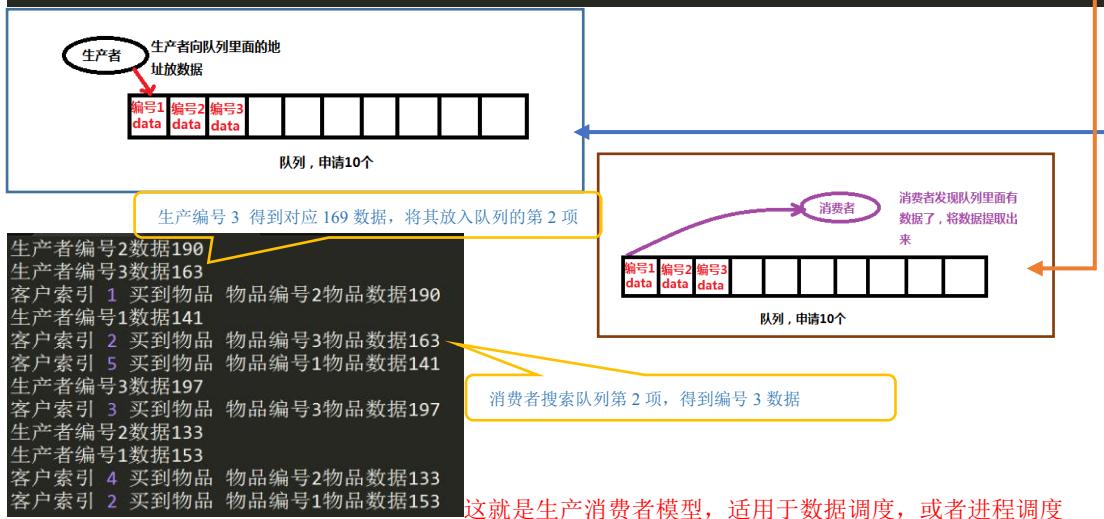
comsumption(1, myqueue).start() # 创建1个消费者，提取数据
comsumption(2, myqueue).start() # 创建2个消费者，提取数据
comsumption(3, myqueue).start() # 创建3个消费者，提取数据
comsumption(4, myqueue).start() # 创建4个消费者，提取数据
comsumption(5, myqueue).start() # 创建5个消费者，提取数据

```

```

while True:
    pass

```



## 线程池

time.time 获取线程当前时间，一般用于查询线程执行了多少时间，看看代码效率

```
import time

namelist = ["show执行1次", "show执行2次", "show执行3次"]

def show(str):
    print("%s" % str)
    time.sleep(2)

start = time.time() # time.time()输出线程执行当前时间

for name in namelist:
    show(name)

end = time.time() # time.time()输出线程执行当前时间
print(end - start)
# 线程开始获取的当前时间减去线程结束获取的当前时间，就是线程执行的时间
```

```
show执行1次
show执行2次
show执行3次
6.000343322753906
```

因为函数延时，这个线程执行了 6 秒

线程池主要解决线程加速运行的问题

先用 pip 安装 threadpool 模块

pip install threadpool

```
C:\Users\WZZO\AppData\Local\Programs\Python\Python36-32>pip install threadpool
  Downloading https://files.pythonhosted.org/packages/6a/92/483af8a4325cd72c131db4e42cc4812505667b784eb180a4fba9348c4a3c/threadpool-1.3.2-py3-none-any.whl
Installing collected packages: threadpool
Successfully installed threadpool-1.3.2
WARNING: You are using pip version 19.2.3, however version 19.3.1 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.
```

threadpool 库

threadpool.ThreadPool(参数) #参数:线程池支持的最大任务数。

threadpool.makeRequests(需要开启多线程的函数名，函数相关参数，回调函数)

#创建了要开启多线程的函数，以及函数相关参数和回调函数，其中回调函数可以不写，  
default 是无，也就是说 makeRequests 只需要 2 个参数，函数名和相关参数就可以运行

pool.putRequest(...) 将多线程函数请求丢进线程池

```
import threadpool # 线程池库
import time

namelist = ["show执行1次", "show执行2次", "show执行3次"]

def show(str):
    print("%s" % str)
    time.sleep(2)

start = time.time() # time.time()输出线程执行当前时间

pool = threadpool.ThreadPool(10) # 申请线程池中的线程个数, 10个线程
request = threadpool.makeRequests(show, namelist)
#threadpool.makeRequests创建需要线程池处理的任务,传入线程函数名, 传入列表参数
#返回多个任务, 也就把show函数根据namelist列表分成多个show任务, 然后返回

for name in request: #循环每一个show任务, 获取出请求进行执行
    pool.putRequest(name) #threadpool.putRequest将创建的多个任务put到线程池中
                           #压入线程池开始执行

end = time.time() # time.time()输出线程执行当前时间
print(end - start)
# 线程开始获取的当前时间减去线程结束获取的当前时间, 就是线程执行的时间
```

```
0.0019998550415039062
show执行1次
show执行2次
show执行3次
```

你看就算函数有延时也会被忽略, 直接加速执行

## 线程定时器, 定时时间到触发线程执行

threading.Timer(延时时间单位秒, 时间到执行的函数)

threading.start()启动线程

```
import threading
import time

def timego():
    print("定时线程执行")

timethread = threading.Timer(5, timego) # 定时时间, 要执行的函数
timethread.start() # 启动线程定时器

i = 0
while True:
    time.sleep(1)
    i = i + 1
    print("主程序运行 = %d" % i)
```

```
主程序运行 = 1
主程序运行 = 2
主程序运行 = 3
主程序运行 = 4
定时线程执行
主程序运行 = 5
主程序运行 = 6
主程序运行 = 7
主程序运行 = 8
主程序运行 = 9
主程序运行 = 10
主程序运行 = 11
主程序运行 = 12
```

你看主程序执行 5 秒后, 定时线程就是执行了, 然后主程序继续执行下去, 定时线程因为执行一次后就消亡了, 不会再次执行。你可以给定时线程加死循环, 让它一直和主线程并行执行。

## Python 网络 TCP/UDP 编程

socket.socket(协议族参数, 套接字类型) //返回对象, 创建套接字

协议族参数:AF\_INET //socket 用于多台电脑网络通信

AF\_UNIX //socket 用于一台电脑进程间通信

套接字类型:SOCK\_STREAM //主要用于 TCP 协议

SOCK\_DGRAM //主要用于 UDP 协议

对象.close()//关闭套接字

对象.sendto(byte 类型 string, (address, port))

Byte 类型 string: 就是你发的数据不管是字符串还是数字必须用 encode 转换成 byte 类型

Address: 就是你发往的服务器(或另一台电脑 IP 地址), 元祖

port: 就是你发往的服务器(或另一台电脑端口号), 元祖

客户端代码

```
import socket

udp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) #创建socket

data = "abcdefg123456"
udp.sendto(data.encode("utf-8"), ("192.168.127.128", 8888)) #发送数据

udp.close() #关闭udp
```

下面写个服务端来验证

socket 和客户端一样

对象.bind(元祖参数)

元祖参数: 填入本服务器地址和开放的端口, 这样客户端可以填入这个地址和端口找到本服务器

对象.recv(bufsize) //接受 TCP 套接字的数据。数据以字符串形式返回, bufsize 指定要接收的最大数据量。

flag 提供有关消息的其他信息, 通常可以忽略

服务端代码

```
import socket

udpserver = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
udpserver.bind(("192.168.127.128", 8888))
while True:
    data=udpserver.recv(1024) #一次接收1024字节
    print(data.decode('utf-8'))
```

返回的数据一定要解码, 因为你客户端发数据也是 utf8 编码的

如果服务端不想接受数据了, 记得一定要执行 close

```
root@ubuntu:/home/xzz/pytest# python3 udp.py
abcdefg123456
```

服务器收到客户端发来的字符串

如果我服务端不对数据进行解码呢?

```
import socket

udpserver = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
udpserver.bind(("192.168.127.128", 8888))
while True:
    data=udpserver.recv(1024) #一次接收1024字节
    print(data)
```

服务端没有解码

```
root@ubuntu:/home/xzz/pytest# python3 udp.py
b'abcdefg123456'
```

你看我收到客户端发来的 b 表示的二进制字符串, 这种二级制字符串没法进行运算处理。

对象.recvfrom(接受多少个 UDP 数据) //接收 UDP 数据, 与 recv()类似, 但返回值是 (data,address) 元组。其中 data 是包含接收数据字符串, address 是发送数据的套接字地址。

```
import socket

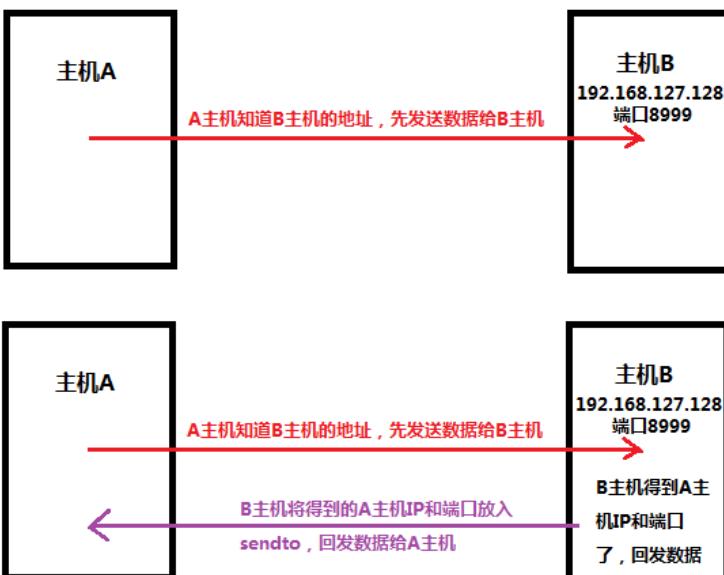
udpserver = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
udpserver.bind(("192.168.127.128",8888))
while True:
    addr,data=udpserver.recvfrom(1024) #一次接收1024字节
    print(data)
    print(addr)
udpserver.close()
```

如果服务端不想接受数据了, 记得一定要执行 close

```
root@ubuntu:/home/xzz/pytest# python3 udp.py
('192.168.127.1', 60682)
b'abcdefg123456'
```

执行客户端, 服务端收到(地址端口)和数据

## UDP 双向通信实验



### 客户端代码

```
import socket

udp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) # 创建socket

data = "abcdefg123456"
udp.sendto(data.encode("utf-8"), ("192.168.127.128", 8999)) # 发送数据给A主机
receiveData, receiveAddr = udp.recvfrom(1024) # 接受A主机返回来的数据
print(receiveData.decode('utf-8')) # 显示A主机返回来的数据
print(receiveAddr) # 显示A主机返回来的A主机IP和地址
udp.close() # 关闭udp
```

```
root@ubuntu:/home/xzz/pytest# python3 udp.py
```

先执行服务端程序，等待客户端发数据来触发  
callback Data to B  
('192.168.127.128', 8999)  
[Finished in 0.3s]

```
root@ubuntu:/home/xzz/pytest# python3 udp.py
```

服务端也得到客户端发来的数据和 IP 端口

这就是客户端和服务端 UDP 双向通信，但是感觉有点像客户端触发一下数据发向服务端，服务端才返回一下数据给客户端，有一点绕一下跳一下的逻辑。这倒是像 HTTP 协议。

如果客户端不触发数据，服务端也不会返回数据给客户端

下面实现一个类似 MQTT 协议的双向触发逻辑，就是服务器可以随时发数据给客户端，客户端也可以随时发数据给服务端。

### 服务端不变

```
import socket

udpserver = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

udpserver.bind(("192.168.127.128", 8999))

while True:
    data, addr = udpserver.recvfrom(1024) # 因为不知道对方A主机的IP地址，只有先接受数据获取到A主机IP和端口
    print(addr) # addr是返回的ip地址和端口的元组
    print(data.decode('utf-8')) # data返回的是字符串byte类型，需要转码成str输出

    udpserver.sendto("callback Data to B".encode('utf-8'), addr) # 发回数据给A主机

udpserver.close()
```

### 客户端加入死循环

```
import socket

udp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) # 创建socket

data = "abcdefg123456"
while True:
    udp.sendto(data.encode("utf-8"), ("192.168.127.128", 8999)) # 发送数据给A主机
    receiveData, receiveAddr = udp.recvfrom(1024) # 接受A主机返回来的数据
    print(receiveData.decode('utf-8')) # 显示A主机返回来的数据
    print(receiveAddr) # 显示A主机返回来的A主机IP和地址
    udp.close() # 关闭udp
```

客户端只要加入死循环，不停的等待接受服务器的数据，就和服务端一样不停的等待接受客户端数据，这样就做到类似 MQTT 不需要触发就能相互随时访问，但是还有一个问题，如果我随时发送数据貌似不行，因为我发送函数在接受函数后面，下次循环接受函数就把进程阻塞了，后面多线程服务器会讲。

客户端不停的接受

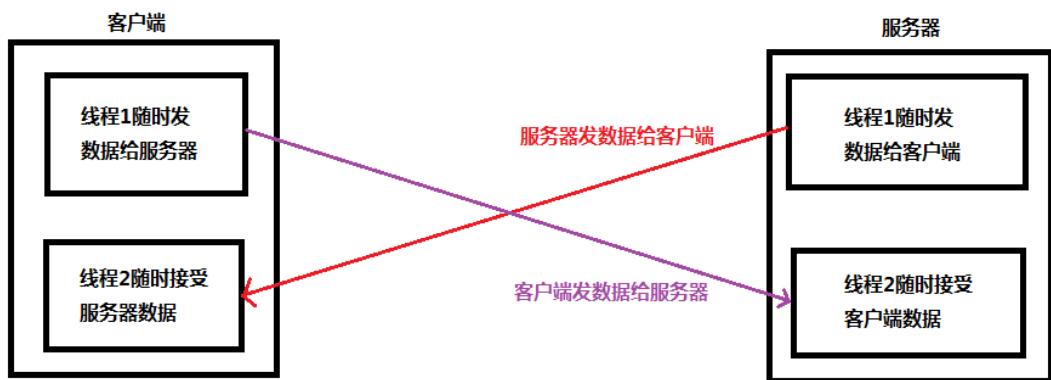
服务端不停接受

```
abcdefg123456
('192.168.127.1', 56248)
```

但是这有一个问题，代码没错，但是这样服务端就只能将进程卡死处理一个客户端的数据，其它客户端就管不了了，其实这在后面服务器架构改进会介绍，这里不介绍处理多客户端方法。

Python echo 测试，用来测试你的网络通信代码延时和硬件网络延时

## Python 多线程解决 UDP 服务器和客户端，相互之间随时发随时收



服务端程序

```
import socket
import threading

udpserver = None
#这是全局变量，因为我不知道这个变量会接受什么类型的数据，我定义None，#None可以接受数据后再让变量有属性

def recvData():#接受客户端随时发来的数据
    while True:
        data,addr=udpserver.recvfrom(1024)
        print(data.decode('utf-8'))

def sendData():#随时发送数据给客户端
    while True:
        string = input("---- please input")
        udpserver.sendto(string.encode('utf-8'),("192.168.1.5",8889))

def main():
    global udpserver #全局变量使用
    udpserver = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
    udpserver.bind(("192.168.127.128",8999))#这里服务端也需要绑定

    threading.Thread(target=sendData).start() #启动发送线程
    threading.Thread(target=recvData).start() #启动接受线程

    while True: #死循环一定不要忘
        pass

    udpserver.close() #注意程序执行完要关闭socket

if __name__ == "__main__":
    main()
```

## 客户端程序

```
import threading
import socket
import time

xudpserver = None
#这是全局变量，因为我不知道这个变量会接受什么类型的数据，我定义None,
#None可以接受数据后再让变量有属性

def recvdata():#接受服务器来的数据

    while True:
        xdata,xaddr = xudpserver.recvfrom(1024)#如果多线程接受数据一定要进行绑定bind
        print(xdata.decode('utf-8'))

def senddata():#发送给服务器的数据

    while True:
        xstring = input("please input")
        xudpserver.sendto(xstring.encode('utf-8'),("192.168.127.128",8999))

def main():
    global xudpserver #全局变量使用

    xudpserver = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
    xudpserver.bind(("","8889"))
    #因为我是和虚拟机服务器通信，有可能ip冲突，
    #导致一边只能发一边只能收，所以用""实现动态ip
    #如何客户端服务器都是实际的主机，那么要双方都要绑定IP和端口
    #这样recvfrom才能执行成功

    threading.Thread(target=senddata).start()#启动发送线程
    threading.Thread(target=recvdata).start()#启动接受线程

    while True: #死循环一定不要忘
        pass

    xudpserver.close()#注意程序执行完要关闭socket

if __name__ == "__main__":
    main()
```

这里一定要注意如果不 bind 会报错  
File "udpclient.py", line 12, in recvdata  
 xdata,xaddr = xudpserver.recvfrom(1024)  
OSError: [WinError 10022] 提供了一个无效的参数。  
因为是多线程，发送和服务分离了函数的，不能像前面半双工那样 sendto 之后直接 recv

please input--->>>>>>send data to service 客户端向服务器发数据

<--- please input--->>>>>>send data to service 服务器收到

<<<<-----service data to client 服务器向客户端发数据

please input<<<<-----service data to client 客户端收到服务器数据

```
please inputaaaaaaaaaaaaaaa  
please inputbbbbbbbbbbbbbbbbbb  
please inputcccccccccccccc
```

客户端随时向服务器发 3 条数据

```
aaaaaaaaaaaaaaaaaa  
bbbbbbbbbbbbbbbbbb  
cccccccccccccc
```

服务器连续收到客户端 3 条数据

```
<--- please input111111111111111111111111  
<--- please input2222222222222222222222  
<--- please input333333333333333333333333
```

服务器随时发 3 条数据给客户端

```
111111111111111111111111  
2222222222222222222222  
3333333333333333333333
```

客户端收到服务器 3 条数据

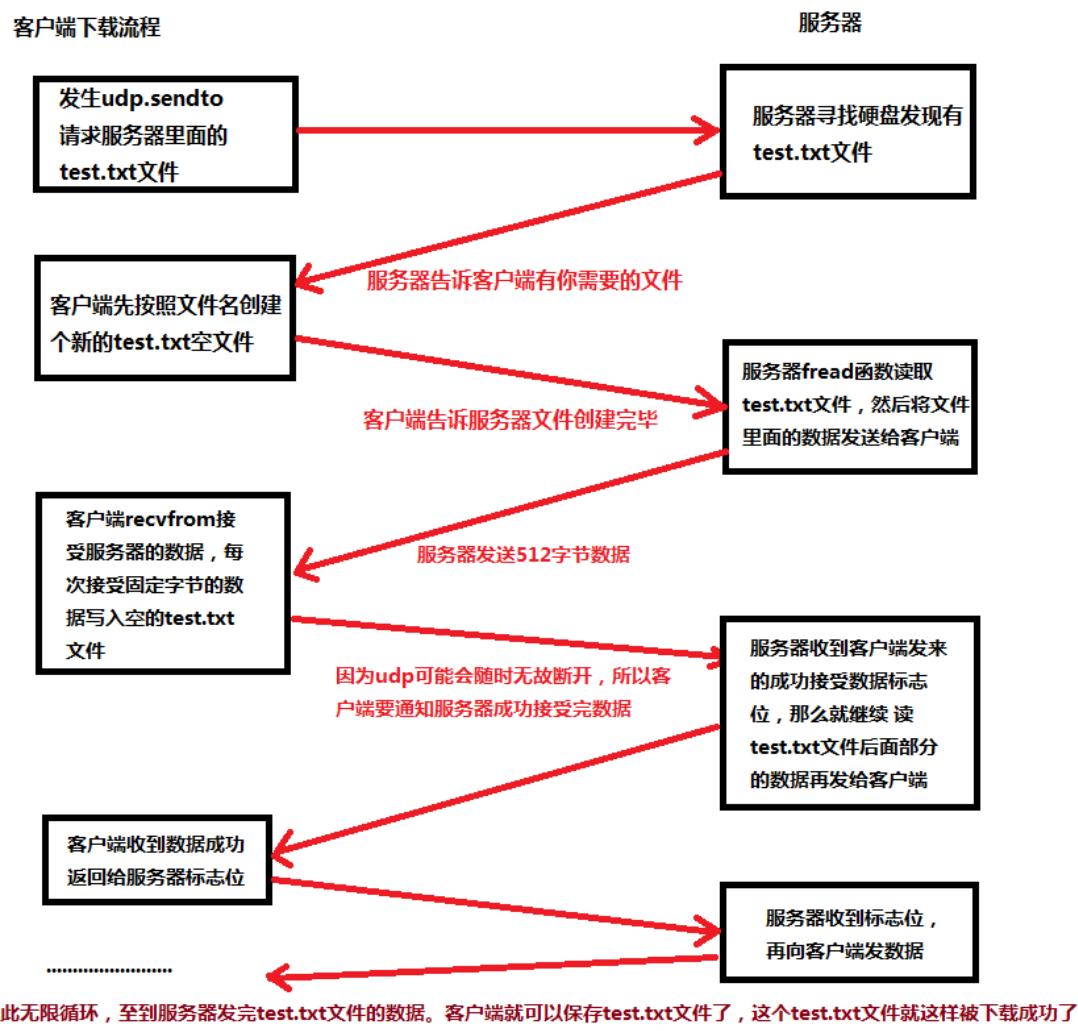
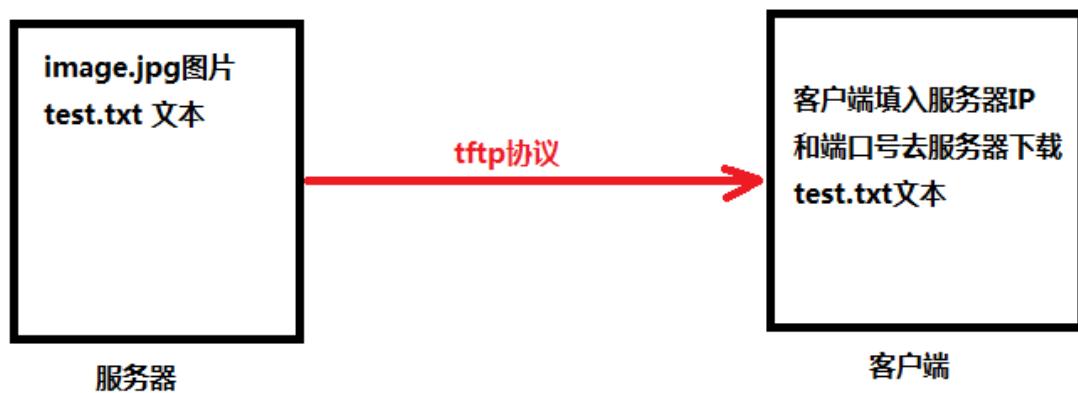
这就是 UDP 全双工通信，服务器客户端可以随意随时发数据，不需要请求再来响应这种繁琐的操作。

我们再来看看无需 bind 的程序，为什么运行不会报错

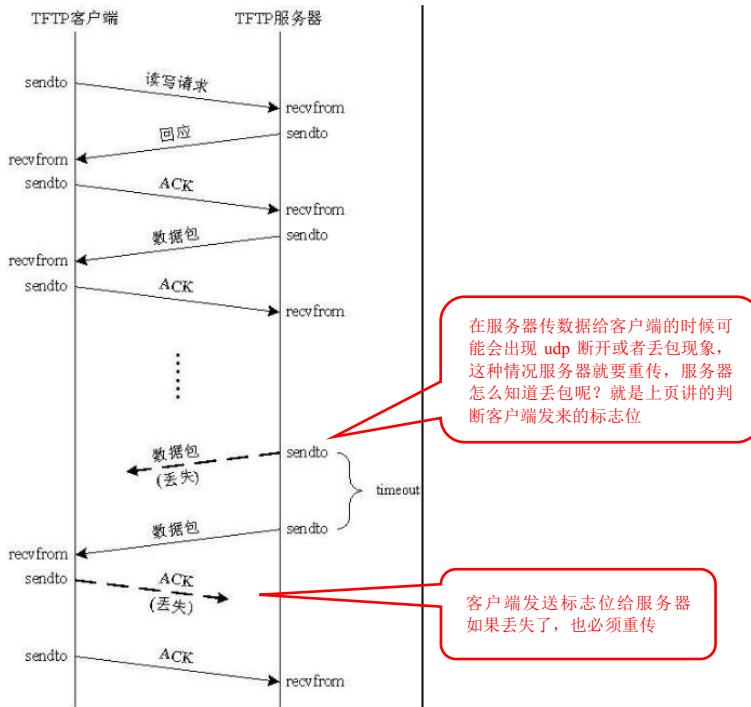
```
import socket  
  
udp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)#创建socket  
  
data = "abcdefg123456"  
while True:  
    udp.sendto(data.encode("utf-8"), ("192.168.127.128", 8999))#发送数据给A主机  
    receiveData, receiveAddr = udp.recvfrom(1024)#接受A主机返回来的数据  
    print(receiveData.decode('utf-8'))#显示A主机返回来的数据  
    print(receiveAddr)#显示A主机返回来的A主机IP和地址  
    udp.close()#关闭udp
```

像这种 sendto 之后马上 recvfrom，而且是在同一个函数，同一个线程里面的 socket 就可以不需要 bind。

## UDP 实现 TFTP 下载器



但是每一次发送和接受的数据包都有格式要求，这个格式就是 tftp 格式  
下面将 tftp 流程在详细描述一下



### 下面讲解 tftp 数据格式

读请求 Read request (RRQ)	操作码 1(Read)	文件名	0	模式	0
	2 Bytes	n Bytes String	1B	n Bytes String	1B
写请求 Write request (WRQ)	操作码 2(Write)	文件名	0	模式	0
	2 Bytes	n Bytes String	1B	n Bytes String	1B
数据包 (DATA)	操作码 3(Data)	块编号	数据		
	2 Bytes	2 Bytes	512 Bytes Data		
ACK	操作码 4(ACK)	块编号			
	2 Bytes	2 Bytes			
ERROR	操作码 5(ERR)	差错码	差错信息	0	
	2 Bytes	2 Bytes	n Bytes String	1B	https://blog.csruyoung2415

这是客户端发给---->服务器的一包数据

操作码 2 字节: 操作码写 1 表示下载, 写 2 表示上传

文件名: 就是写入你要下载的文件的名称(如 test.txt)

0: 就是默认写数字 0

模式: 写 octet, tftp 有很多种格式, 我们这里用 octet 格式做数据包

0: 就是默认写 0

将这包数据发到服务器 69 端口, 69 端口是大家公认的 tftp 下载端口。80 端口是 http 端口  
客户端下载流程 服务器

发生udp.sendto  
请求服务器里面的  
test.txt文件

服务器寻找硬盘发现有  
test.txt文件

就是这步的数据包

读请求 Read request (RRQ)	操作码 1(Read)	文件名	0	模式	0
	2 Bytes	n Bytes String	1B	n Bytes String	1B
写请求 Write request (WRQ)	操作码 2(Write)	文件名	0	模式	0
	2 Bytes	n Bytes String	1B	n Bytes String	1B
数据包 (DATA)	操作码 3(Data)	块编号	数据		
	2 Bytes	2 Bytes	512 Bytes Data		
ACK	操作码 4(ACK)	块编号			
	2 Bytes	2 Bytes			
ERROR	操作码 5(ERR)	差错码	差错信息	0	
	2 Bytes	2 Bytes	n Bytes String	1B	https://blog.csdn.net/lyyoung2415

这是服务器端回发数据---->给客户端

操作码: 写 3 表示服务器返回的这包数据是有 512 字节数据的。写 5 表示服务器没有文件, 这包数据没有 512 字节数据。

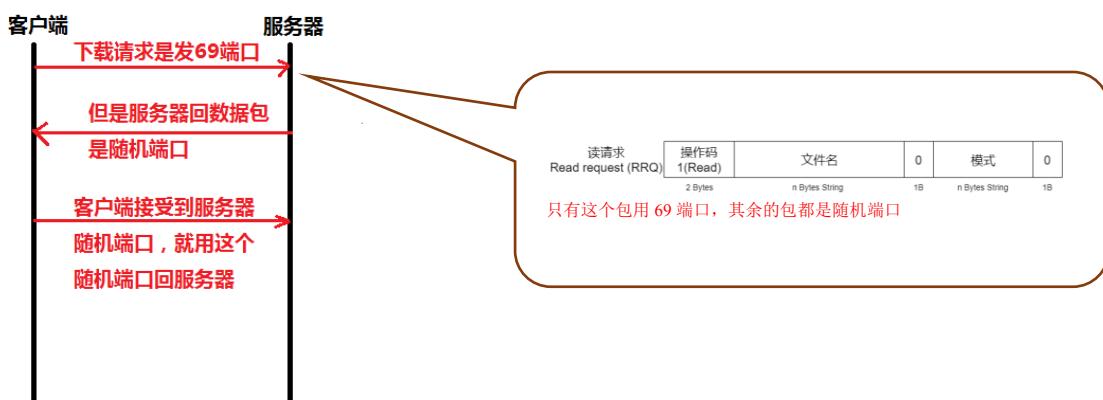
块编号 2 字节: 就是服务器将这 512 个字节的数据做一个编号, 让客户端知道这个编号对应这次发来的 512 字节数据

数据: 就是固定的 512 字节 txt 文件部分数据

客户端向---->服务器发送回复确认 ACK, 表示收到了 512 字节数据包成功

操作码: 写 4  
块编号: 就是刚才接受服务器 512 字节数据包的编号。把这个编号发回给服务器, 服务才可以确认成功发送这帧数据包

有个地方要注意



所以要记住, 只有客户端下载请求包是用69端口发, 后面服务器和客户端双向数据交互都是随机端口, 不要搞错了, 不要以为后面的双向交互还是69端口, 这样理解是错误的

tftp 程序如何知道服务器的文件数据发送完了



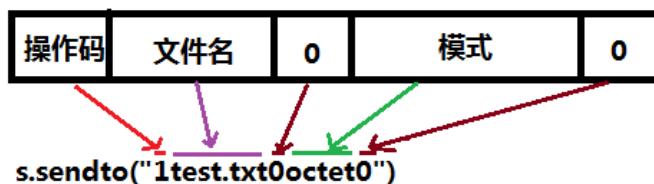
如果客户端接受到最后一包数据正好是516怎么办？

那么客户端会让服务器再发送一次，这次客户端接受到的数据全部都是0了，证明接受结束

也就是服务器发生一个长度为 0 的数据包给客户端，客户端就知道数据发完了。

我怎么保证我发的数据，操作码只占用 2 个字节？

s = socket(...)



1是操作码，我这样发送数据，怎么能保证1只占用2字节，万一 '1' 被编码成1位呢？而不是1个字节呢？岂不是会出问题

test.txt文件也是，英文1个字符占用1字节，万一我是中文字符呢？那1个字符岂不是占用2字节

所以我们需要**struct.pack**来封包，把没有占用到1字节的自动填充成1字节，如果是写的1个位的数在操作码，我们就把它自动变成2字节的数

案例：

```
buf = struct.pack("!H8sb5sb",1,test.txt,0,"octet",0)

'!' 表示我发送的是网络数据(网络数据是大端)
'H' 表示我发送的'1'这个字符占用2字节
'8s' 表示我的字符串是8个字节，每个占用1字节，就是8字节
'b' 表示占用1字节
'5s' 表示我的字符串是5个字节，和8s一个逻辑
'0' 表示占位符，表示后面的数据包长度不足516字节，系统会自动把1做成占用2字节的数据1

发送的数据包就成这样了 [ 1 | test.txt | 0 | octet | 0 ]
```

后面几页的大小端章节会讲

1.先确认 linux 服务器的 ip 地址和 tftp 端口号

```
# /etc/default/tftpd-hpa
TFTP_USERNAME="tftp"
TFTP_DIRECTORY="/tftpboot"
TFTP_ADDRESS="0.0.0.0:69"
TFTP_OPTIONS="-l -c -s"
```

inet addr:192.168.127.128

Linux 服务器:端口号 69, ip 地址 192.168.127.128

2.确认放在 tftp 服务器里面的被下载文件

test.txt 要下载的文件是 test.txt

xxxxzzzzabcdefg123456 这是文件内容

3.客户端实现下载

```
import struct #struct.pack封包库
import socket

file_name_byte_array = "test.txt".encode('gb2312')
#组包, octet 代表TFTP协议的一种模式
sendData = struct.pack('!H'+str(len(file_name_byte_array))+'sb5sb',
                      1, file_name_byte_array, 0, b'octet', 0)

udpSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
udpSocket.sendto(sendData, ("192.168.127.128", 69))

newFile = open("test.txt", 'wb')
while True:
    #等待接收数据
    recvInfo = udpSocket.recvfrom(1024) #1024表示本次接受的最大字节数
    print(recvInfo)
```

这是封包函数, 把请求包封装在 sendData 变量里面

读请求 1(RRQ) 操作码 文件名 0 模式 0

等待服务器把文件的数据发过来

在python\_base基础文档讲过struct.pack和struct.unpack函数的, 这里重复一下 ←

struct.pack(fmt, v1, v2, v3, ....) 这是格式

struct.pack('!H'+str(len(file\_name\_byte\_array))+'sb5sb', 1, file\_name\_byte\_array, 0, b'octet', 0)

解析下就是:

fmt='!H'+str(len(file\_name\_byte\_array))+'sb5sb'  
↑  
! =network(网络大端) str(就是将字符串计 s=char[]  
H=unsigned short 2字节 算出来的长度数字加 b=signed char  
在这里), 那么字符串  
必须这么长,

fmt就组成了2字节+文件名+1B字符+5个字符+1B字符的数据包

既然格式这么定了, 那么v1,v2, v3....传入进来的数据顺序也必须符合这个fmt格式

v1=1  
v2=file\_name\_byte\_array  
v3=0  
v4=b'octet' (二进制字符串)  
v5=0

结合起来就是 1, file\_name\_byte\_array, 0, b'octet', 0

b'\x00\x01test.txt\x00octet\x00'

所以 senddata 就是

(b'\x00\x03\x00\x01xxxxzzzzabcdefg123456\n', ('192.168.127.128', 33983))  
(b'\x00\x03\x00\x01xxxxzzzzabcdefg123456\n', ('192.168.127.128', 33983))

数据返回成功, 确实是 test.txt 文件里面的字符。第一帧数据包发送, 返回解析成功了。

下面我们下载有大量数据的文件

```
import struct #struct.pack封包库
import socket

file_name_byte_array = "test.txt".encode('gb2312')
#组包，octet 代表TFTP协议的一种模式
sendData = struct.pack('!H'+str(len(file_name_byte_array))+'sb5sb',
                      1, file_name_byte_array, 0, b'octet', 0)
print(sendData)
udpSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
udpSocket.sendto(sendData, ("192.168.127.128", 69))

newFile = open("test.txt", 'wb')
while True:
    #等待接收数据
    recvInfo = udpSocket.recvfrom(1024) #1024表示本次接受的最大字节数
    transPort = recvInfo[1][1] #服务器传回来用的端口号
    data = recvInfo[0] #TFTP数据包的字节流
    len_data = len(data)#计算数据包长度
    result = struct.unpack("!H", data[:2]) #解包
    opcode = result[0] #获取操作码
    if opcode == 3: #如果操作码是3，说明是DATA包
        result = struct.unpack('!H'+str(len_data-4)+'s', data[2:len_data])
        block = result[0] #获取块编号
        fileStream = result[1] #文件字节流
        newFile.write(fileStream) #字节流写入文件
        #向服务器发送一个确认包
        ackInfo = struct.pack('!HH', 4, block)
        udpSocket.sendto(ackInfo, ("192.168.127.128", 69))
        if len(fileStream) < 512:
            break
    elif opcode == 5: #如果操作码是5，说明是ERROR包
        result = struct.unpack('!H'+str(len_data-5)+'s', data[2:len_data-1])
        print('传输出现异常！')
        print(result[1].decode('gb2312')) #输出错误信息
        break
newFile.close()
udpSocket.close()
print("下载完成")
```



这是服务器端回发数据---->给客户端

操作码: 写 3 表示服务器返回的这包数据是有 512 字节数据的。写 5 表示服务器没有文件，这包数据没有 512 字节数据。

块编号 2 字节: 就是服务器将这 512 个字节的数据做一个编号，让客户端知道这个编号对应这次发来的 512 字节数据

数据: 就是固定的 512 字节 txt 文件部分数据



这就是整个 tftp 下载文件的安全传输过程，但是我对 len(fileStream)<512 这段表示怀疑。

struct.unpack 使用

假如封包的内容是 `data = struct.pack("!H8sb5sb", 1, "test.txt", 0, "octet", 0)`

那么解包就是 `yuanzu = struct.unpack("!H", data[:2])`

将`data[0 : 2]`也就是`data`的0字节和1字节数据取出来，然后用H反转换一下，得到两个字节的数据，但是这个数据值就是'1'

'!'感叹号是因为你封包是按照"!"网络大小端封包的，所以解包也要这样解

一般用到unpack的地方如下：

`recvdata=recvfrom(...)` 比如客户端接受了服务器数据，需要解析头



`yuanzu = struct.unpack("!HH", recvdata[:4])`

取出`recvdata`前4个字节，0字节和1字节是操作码，2字节3字节是块编号，所以用两个H来解码，因为1个H代表2字节

`yuanzu = (操作码, 块编号)`

## 大端和小端问题

比如有个数是 `0x11ff`

在大端模式的电脑内存下



也就是数据的高字节放在内存低位地址，数据低字节放在内存高位地址，这就是大端，不符合人类理解习惯，所以大端的数据要反过来读

在小端模式的电脑内存下



低地址存放数据的低位，高地址存放数据的高位，符合人类阅读顺序，所以小端顺着读就是

个人电脑  
一般是小端

大型服务器  
是大端

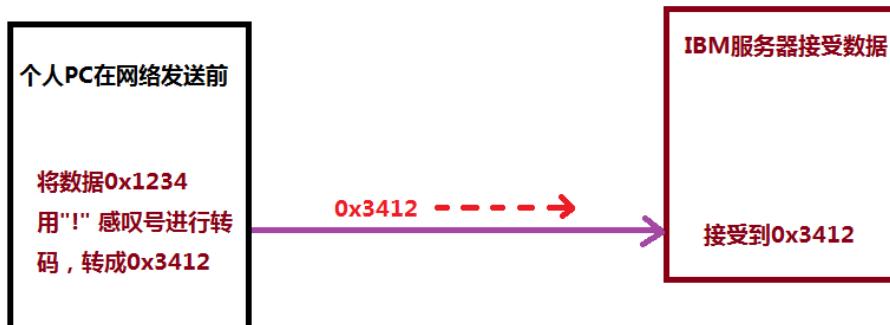
在网络传输中



然后IBM取数据的时候  
将接受的0x1234变成0x3412取出来

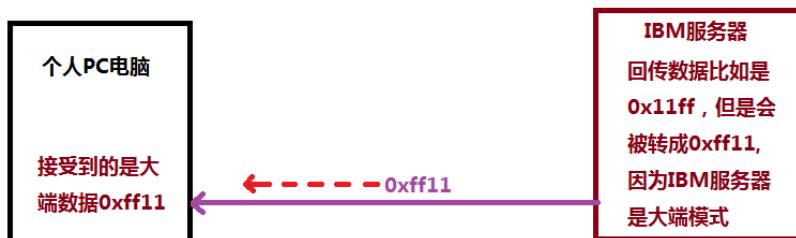
这样服务器读取的数据就不是客户端想发送的

在网络传输中



然后IBM取数据的时候  
还是按照大端提取，  
将0x3412转成  
0x1234再提取

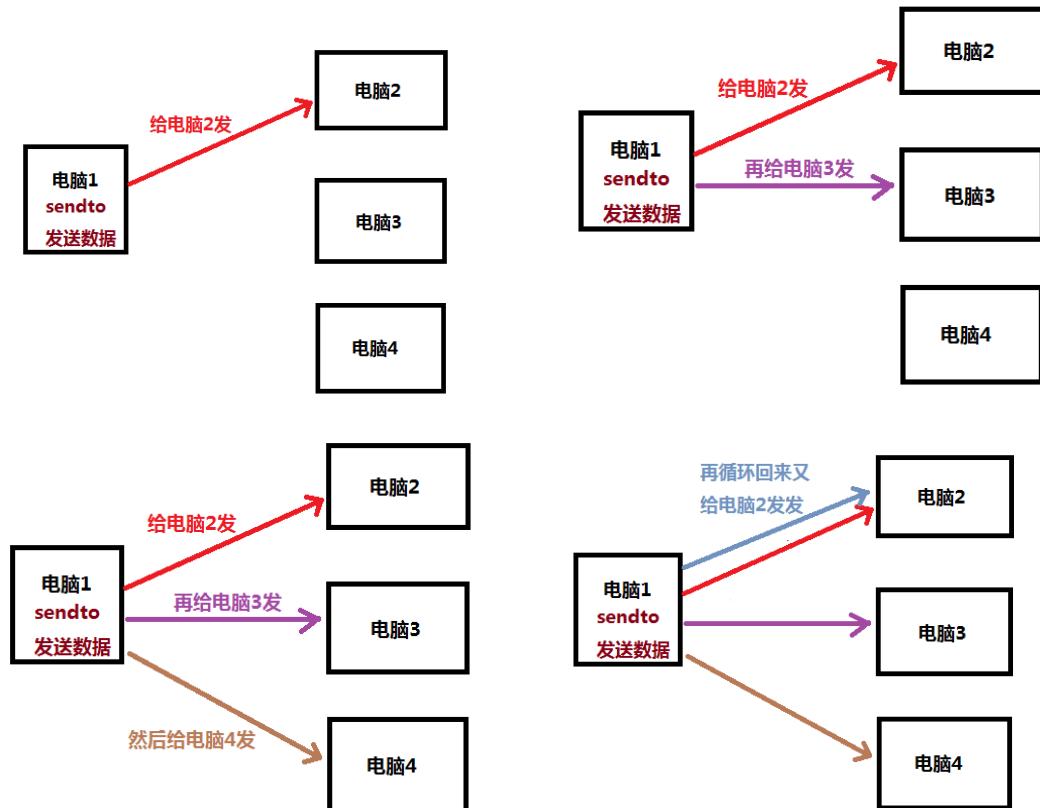
发现了吗，IBM服务器解析的数据0x1234就是客户端发送的0x1234，这就对了



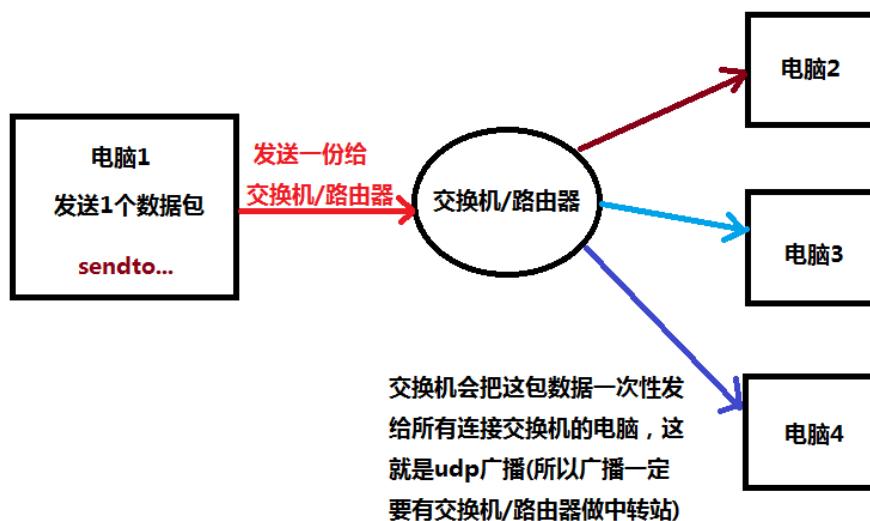
所以在struct.unpack解码的时候要使用"!"感叹号，将0xff11转码成0x11ff才正确

## UDP 广播

udp 广播发送数据流程猜想



如此循环下去。确认是这样发的吗？当然不是，这样理解是错误的。



所以 udp 通信分为单播，多播，广播。

单播就是两台电脑通信

多播就是一台电脑发数据给指定的几台局域网电脑，局域网其它没指定的电脑收不到数据。

广播就是一台电脑发数据给所有局域网的电脑

如何设置本台电脑是广播发送，如果设置其它电脑为广播接收？

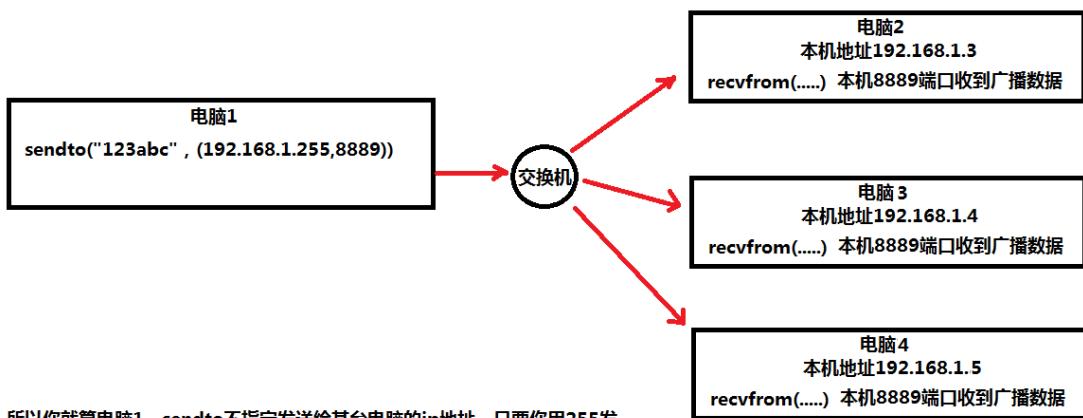
```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM); //创建套接字  
s.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1); //设置这个套接字具有广播功能
```

```
s.sendto("12345abcde", ('<broadcast>', 8889)); //发送广播数据
```

<broadcast> 就是 192.168.1.255，当然你也可以写 192.168.1.255，只是我觉得写 broadcast 更能说明是发送的广播

所以平时我们 sendto 发送的是指定 ip 地址，就是 192.168.1.1~254，像这种地址就是每台电脑的 ip 地址，我指定 192.168.1.1 就表示发给 192.168.1.1 地址的电脑，这种是我们前面说的针对性的发送数据，不是广播

广播就是要 sendto 发送 192.168.1.255，这个 255 就是表示该发送的数据包为广播数据，所有电脑都可以接受



所以你就算电脑1，sendto不指定发送给某台电脑的ip地址，只要你用255发送，其余的电脑都能收到这包数据

这里注意，一定要先设置 setsockopt(...) 这个函数，否则你写 192.168.1.255 是发不了广播数据的，程序会出 bug。

setsockopt(level,optname,value) 函数使用

level 参数： 1. 一般都设置 socket.SOL\_SOCKET 这是用来访问套接字接口选项  
2. socket.SOL\_TCP 访问 TCP 层选项

只要 level 参数确定了，那么 optname 参数能选的选项都是根据 SOL\_SOCKET 来的，如果 level 选择 SOL\_TCP，那么 optname 又是另外一组选项。

optname 参数： 选项(在 SOL\_SOCKET 情况下)  
SOL\_SOCKET SO\_REUSEADDR  
SOL\_SOCKET SO\_KEEPALIVE  
SOL\_SOCKET SO\_LINGER  
SOL\_SOCKET SO\_BROADCAST //我选择广播  
SOL\_SOCKET SO\_OOBINLINE  
SOL\_SOCKET SO\_SNDBUF  
SOL\_SOCKET SO\_RCVBUF  
SOL\_SOCKET SO\_TYPE  
SOL\_SOCKET SO\_ERROR

value 参数： 默认写 1

客户端程序，向外发广播

```
import socket
import sys

s = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
s.setsockopt(socket.SOL_SOCKET,socket.SO_BROADCAST,1) #只要是广播，这句就必须设置
print("发送广播")
s.sendto("12345abcd".encode('utf-8'),('<broadcast>',8999))

#while True:
#    (buff,address) = s.recvfrom(1024)
#    print("receive %s : %s" %(address,buff))

s.close()
```

服务端代码

```
import socket
import threading

def main():
    udpserver = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
    udpserver.setsockopt(socket.SOL_SOCKET,socket.SO_BROADCAST, 1)
    udpserver.bind(('',8999))#这里服务端也需要绑定
    while True: #死循环一定不要忘
        (buff,address) = udpserver.recvfrom(1024)
        print("receive broadcast %s : %s" %(address,buff.decode('utf-8')))
    udpserver.close() #注意程序执行完要关闭socket

if __name__ == "__main__":
    main()
```

服务端接受广播数据也需要设置 setsockopt，只要局域网的每台电脑都需要接受广播数据，那么都需要去设置 setsockopt

服务端不需要绑定 ip 地址，因为接受的是广播，所以不管绑不绑 IP 都要接受广播数据

接受广播数据还是用 recvfrom

经过我 ubuntu 虚拟机和 windows 客户端测试，发现不得行，不知道是不是虚拟机问题，或者广播这台电脑的 MAC 地址要设置成 FFFF.FFFF.FFFF，有机会还是用机房的电脑实测。

## TCP 协议使用

UDP的过程是:



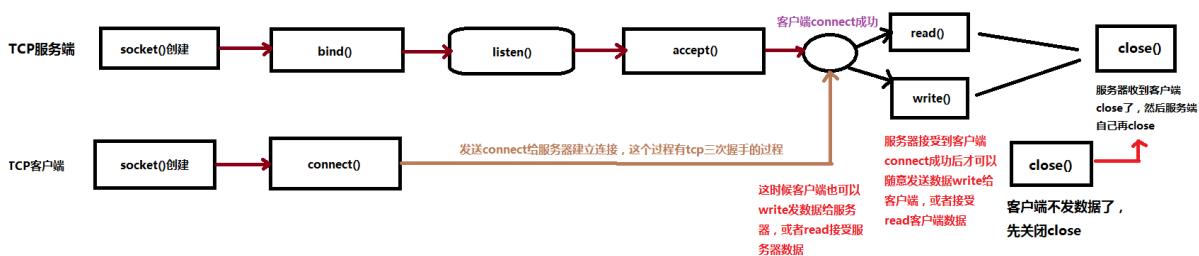
如果是接收:



tcp 链接和 udp 不一样的地方是服务端和客户端双方都必须先 connect 成功之后才能相互 write(sendto)发送数据和 read(recvfrom)接受数据



下面是详细流程



socket.listen(参数) //设置电脑链接数量

tcp 服务端程序编写

```
import socket
from socket import * #tcp头文件不能用socket，要用from socket import *
#因为套接字创建不是用socket.socket，而是直接socket

tcpserver = socket(AF_INET,SOCK_STREAM) #tcp记住是SOCK_STREAM

tcpserver.bind(("192.168.127.128",8888))#因为是服务器，绑定本机IP和端口

tcpserver.listen(5) #listen(参数)

newsocket,addr = tcpserver.accept()#每次客户端执行一次connect，服务器accept才会取消阻塞

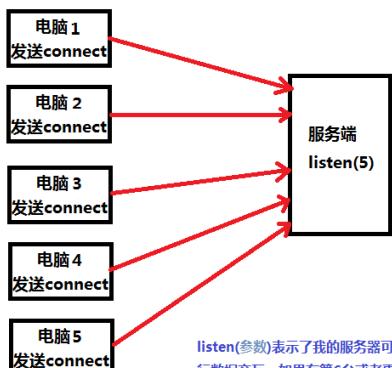
data = newsocket.recv(1024)#接受客户端数据

print("receive : %s : %s" %(str(addr),data))

newsocket.close()#数据接受完成后，如果你要关闭socket，一定要先关闭连接的socket
tcpserver.close()#然后再关闭总的socket
```

listen详解

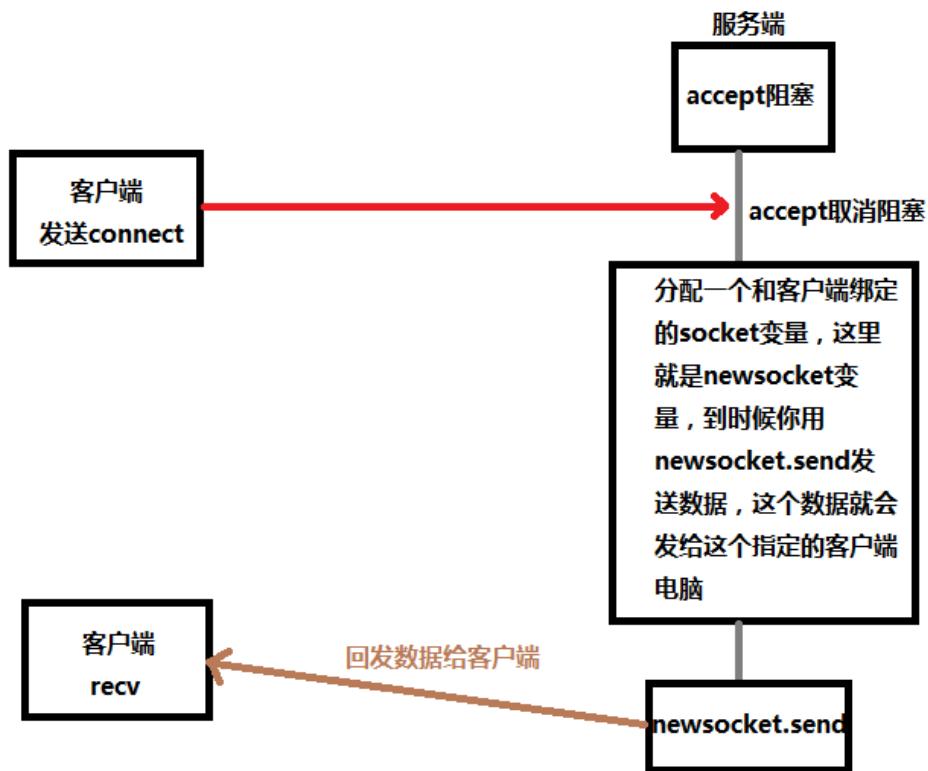
比如listen(5)



listen(参数)表示了我的服务器可以同时连接5台电脑发过来的connect，然后进行数据交互，如果有第6台或者更多的电脑，就只有等待，无法进行数据交互，除非连接connect成功的电脑有一台自动close，就可以放出资源来connect连接另外一台电脑，当然你可以把listen设置得很大来连接更多的电脑

### accept()详解

( connect连接上来的电脑分配一个socket , 连接上来的电脑ip和端口 ) = accept()  
客户端没有执行connect连接上来 , accept就阻塞  
客户端执行connect , accept就会返回客户端的ip端口 , 顺便分配一个socket变量 , 这个变量只代表连接上来的这一台电脑



socket.connect((ip 地址和端口))

tcp 客户端程序编写

```
from socket import * #这里要用from socket import的方式, 服务端程序讲过

tcpsocket = socket(AF_INET,SOCK_STREAM)#记住是SOCK_STREAM

tcpsocket.connect(("192.168.127.128",8888))#链接哪台服务器, 执行connect, 服务器的accept才不会阻塞

tcpsocket.send("abcd1234".encode('utf-8'))#发送数据给服务器
print("客户端tcp发送数据")

tcpsocket.close()#关闭tcp链接
```

客户端你理解下就是了, 主要就是要执行 connect, 这个 connect 执行很慢, 里面有 3 次循环握手程序, 但是你人感觉不出来。所以为什么有些数据要求传输快的要用 udp, 但是 udp 可能会丢数据, 看你怎么选择。

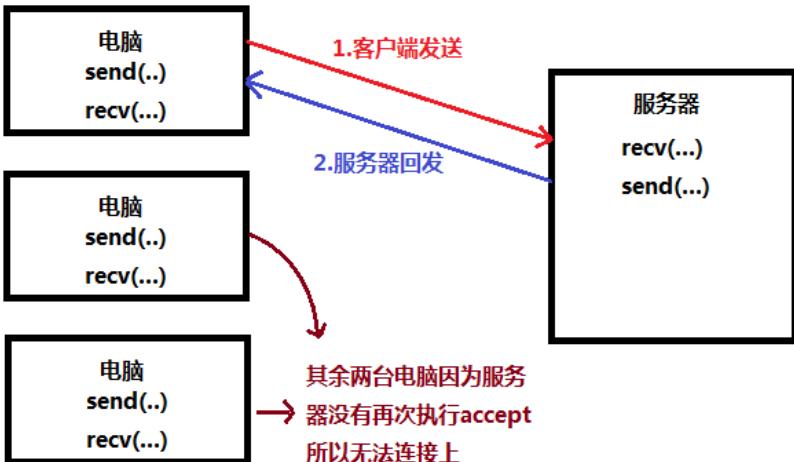
```
root@ubuntu:/home/xzz/pytest# python3 tcp.py
```

等待客户端发数据

```
E:\vscodetest>python -u "e:\vscodetest\pytest\tcp.py"
客户端tcp发送数据
```

```
root@ubuntu:/home/xzz/pytest# python3 tcp.py
receive : ('192.168.127.1', 55902) : b'abcd1234' 服务端收到数据
```

## TCP 实现服务器和客户端相互循环随时发送数据



服务端

```
tcpserver = socket(AF_INET,SOCK_STREAM) #tcp记住是SOCK_STREAM
tcpserver.bind(("192.168.127.128",8888))#因为是服务器，绑定本机IP和端口
tcpserver.listen(5)

while True:
    newsocket,addr = tcpserver.accept()#每次客户端执行一次connect，服务器accept才会取消阻塞
    while True:
        data = newsocket.recv(1024)
        print("receive : %s : %s" %(str(addr),data))
        newsocket.send("echo tcp server data".encode('utf-8'))
    newsocket.close()
tcpserver.close()#然后再关闭总的socket
```

客户端

```
from socket import * #这里要用from socket import的方式，服务端程序讲过
import time

tcpsocket = socket(AF_INET,SOCK_STREAM)#记住是SOCK_STREAM

tcpsocket.connect(("192.168.127.128",8888))#链接哪台服务器，执行connect，服务器的accept才不会阻塞
while True:
    tcpsocket.send("abcd1234".encode('utf-8'))#发送数据给服务器
    print("客户端tcp发送数据")
    data = tcpsocket.recv(1024)
    print("%s" %(data.decode('utf-8')))
    time.sleep(1)

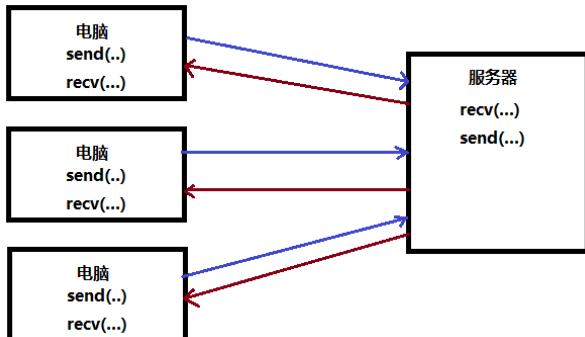
tcpsocket.close()#关闭tcp链接
```

客户端链接上服务器之后也不去关掉这次的tcp链接，那么服务器端口就一直被这台电脑用着

服务器接受

```
E:\vscodetest>python -u "e:\vscodetest\pytest\tcp.py"
客户端tcp发送数据
echo tcp server data
客户端tcp发送数据
echo tcp server data
客户端tcp发送数据
```

客户端也发送接受，但是确实实现了双向传输



现在改进成每台电脑都可以随时和服务器通信  
服务端程序

```
tcpserver = socket(AF_INET,SOCK_STREAM) #tcp记住是SOCK_STREAM
tcpserver.bind(("192.168.127.128",8888))#因为是服务器，绑定本机IP和端口
tcpserver.listen(5)

while True:
    newsocket,addr = tcpserver.accept()#每次客户端执行一次connect，服务器accept才会取消阻塞
    while True:
        data = newsocket.recv(1024)
        if len(data) >0 :
            print("receive : %s : %s" %(str(addr),data))
            newsocket.send("echo tcp server data".encode('utf-8'))
        else:
            print("client disconnet")
            break
    newsocket.close()
tcpserver.close()#然后再关闭总的socket
```

我在死循环加入 data 的接受数据判断，如果发现这次接受的数据 <0，就证明客户端已经断开

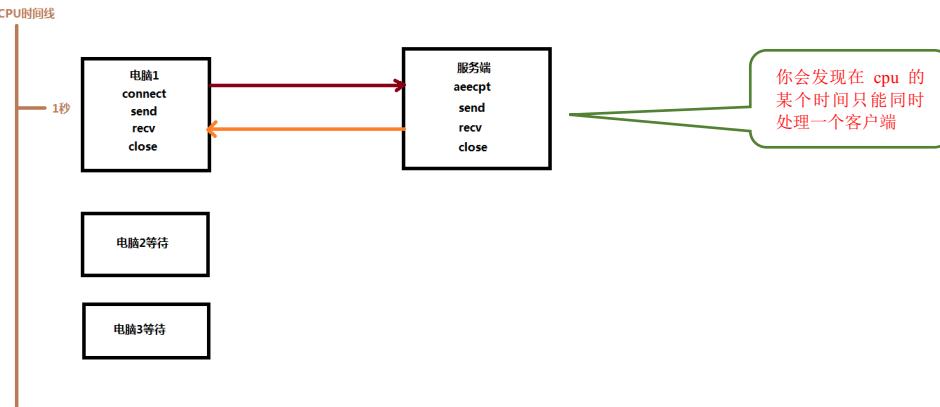
客户端断开，就会跳出循环，然后程序跑到大循环的 accept 处等待新的客户端链接，当然也可能再次链接的是老的客户端，但是也当做新的来链接

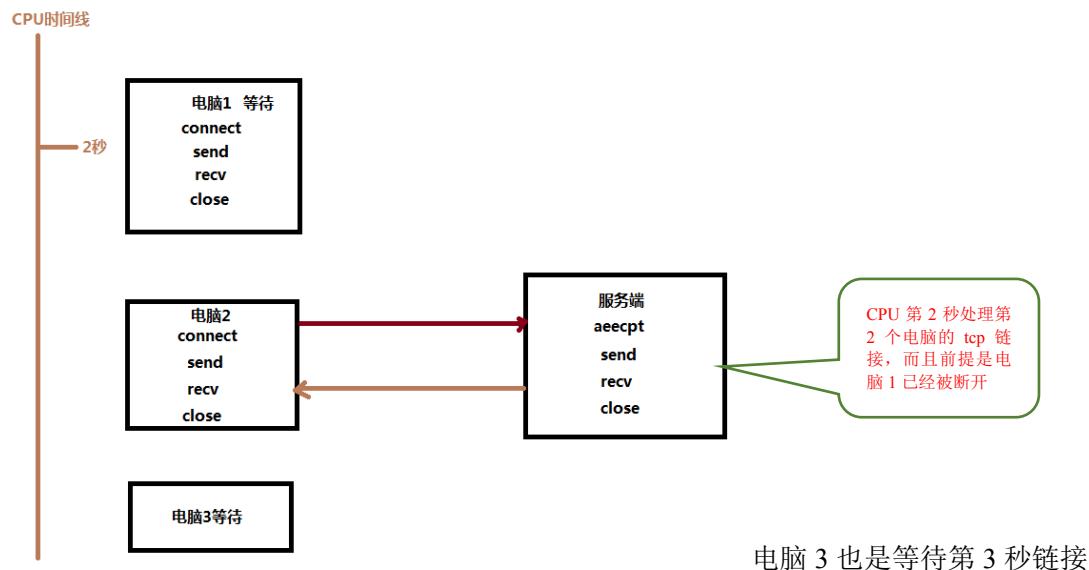
断开客户端之后，一定不要忘记关闭这次创建的 socket，不然下次 accept 连接不上，服务器认为上次的链接还没有断开

客户端程序不变

```
root@ubuntu:/home/xzz/pytest# python3 tcp.py
receive : ('192.168.127.1', 58960) : b'abcd1234'
client disconnet
receive : ('192.168.127.1', 58973) : b'abcd1234'
client disconnet
```

服务器收到客户端数据了，而且客户端断开之后也能判断出来





这就是常规的tcp服务器做法，循环链接多个客户端，循环处理客户端数据，因为CPU速度太快，人类感觉不出来。但是现在又有并发服务器的做法，其实就是在服务器每次收到accept请求，然后将返回的新SOCKET放入一个新的线程/或者进程，这样一个线程/进程代表一个电脑链接过来的SOCKET，做到并行处理。

## TCP 的 listen 到底能链接多少客户端，怎么理解 listen

listen 就是一个队列

服务端代码

```
import socket
from socket import * #tcp头文件不能用socket，要用from socket import *
#因为套接字创建不是用socket.socket，而是直接socket
import time

tcpserver = socket(AF_INET,SOCK_STREAM) #tcp记住是SOCK_STREAM
tcpserver.bind(("192.168.127.128",8888))#因为是服务器，绑定本机IP和端口
tcpserver.listen(5)

while True:
    newsocket,addr = tcpserver.accept()
    print(addr)
    time.sleep(1)
```

客户端代码

```
from socket import * #这里要用from socket import的方式，服务端程序讲过
import time

for i in range(10):#我要链接10次服务器
    s = socket(AF_INET,SOCK_STREAM)
    s.connect(("192.168.127.128",8888))
    print("链接上服务器次数 %d" %i)
```

我服务端 1 秒读取一次 addr，那么客户端会不会是 1 秒链接一次服务端呢？客户端没有延时哦

```
root@ubuntu:/home/xzz/pytest# python3 tcp2.py
```

服务端启动

```
E:\vscodetest>python  
链接上服务器次数 0  
链接上服务器次数 1  
链接上服务器次数 2  
链接上服务器次数 3  
链接上服务器次数 4  
链接上服务器次数 5  
链接上服务器次数 6  
链接上服务器次数 7  
链接上服务器次数 8
```

我发现客户端一次性就链接上了 8 个节点到服务器

```
E:\vscodetest>python  
链接上服务器次数 0  
链接上服务器次数 1  
链接上服务器次数 2  
链接上服务器次数 3  
链接上服务器次数 4  
链接上服务器次数 5  
链接上服务器次数 6  
链接上服务器次数 7  
链接上服务器次数 8  
链接上服务器次数 9
```

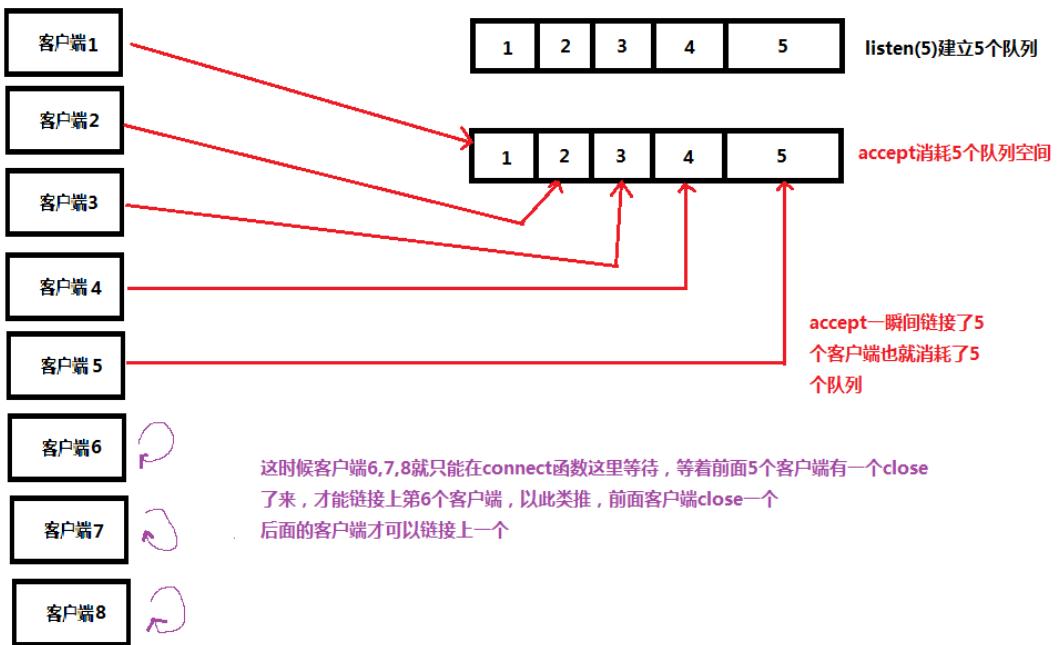
过一会客户端才完成第 9 个节点链接，这是为什么？

```
tcpserver.listen(5)  
  
while True:  
  
    newsocket,addr = tcpserver.accept()  
    print(addr)  
    time.sleep(1)
```

在以前这里 listen 写几，客户端一瞬间就能连上几个，比如写 5，那么客户端瞬间就能链接上 5 个，现在瞬间链接 8 个是因为 linux 底层做了优化，所以这里写几没有意义，写上就是了

所以你会发现这里并不是 accept 循环一次链接一次，而是 accept 根据 listen 的值，一次性会自动连接很多个客户端，然后这里的 newsocket 会自动生成很多个客户端对应的对象。然后在延时循环下一轮。

所以服务端加延时是不对的，服务端就应该按照最快速度运行  
这里只是让你看清楚这个 listen 链接现象



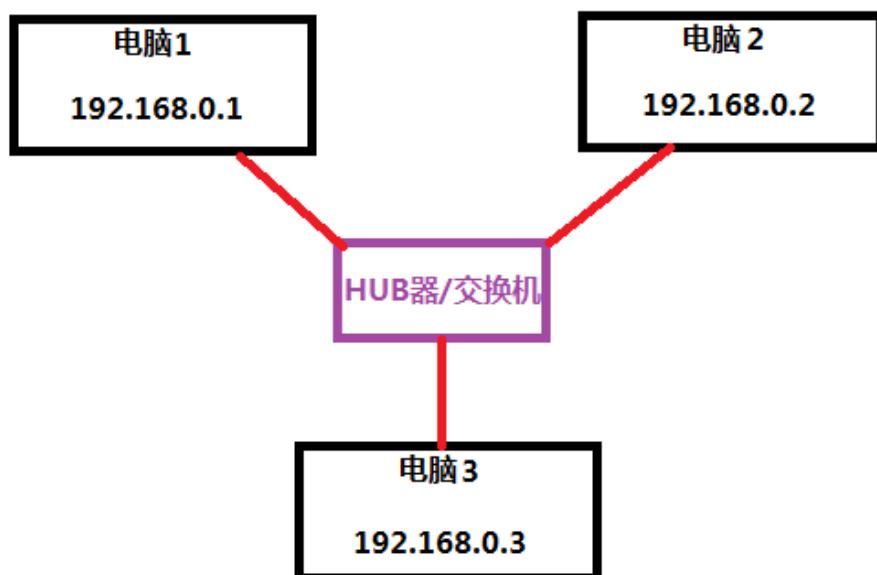
所以 listen 的值是越大越好，但是 linux 系统已经帮你分配了最合理的最大值，所以你 listen 随便写个就是。

所以这里要求服务器处理完了一个客户端的 socket(变量 newsocket)，就要记得及时 close，不然其他电脑在客户端等着就不好了。

## TCP 商用服务器编写

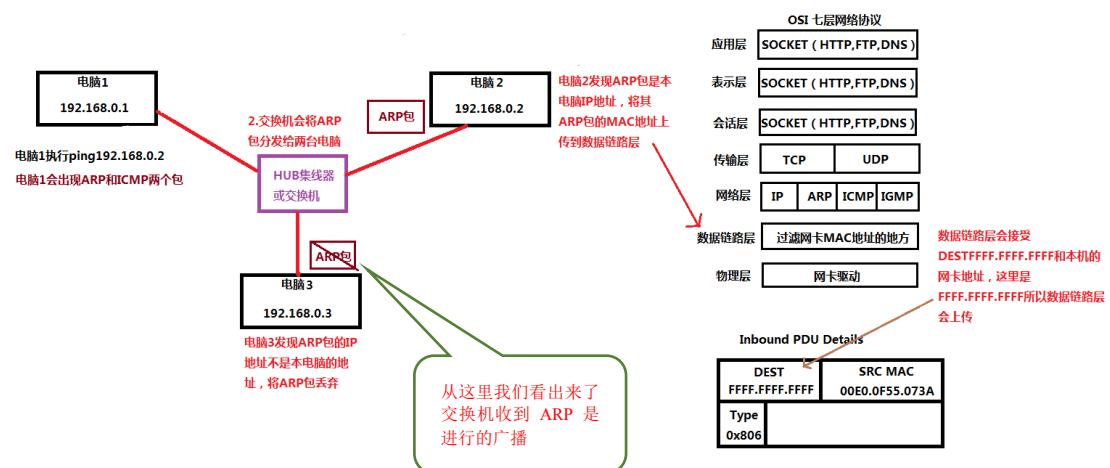
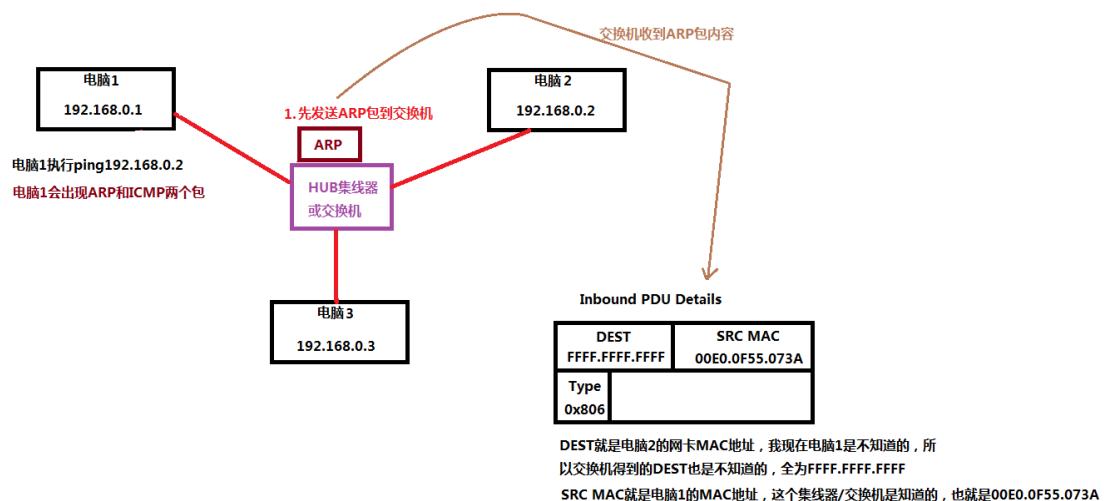
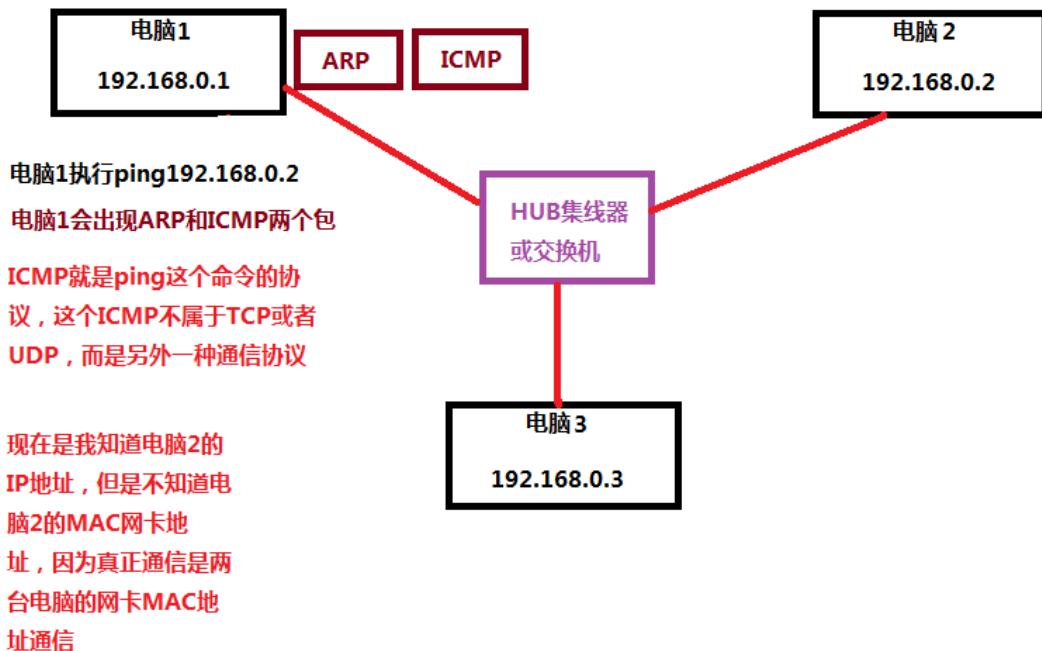
在编写商用服务器之间，先研究下网络链接理论

1.多台电脑怎么通信的？

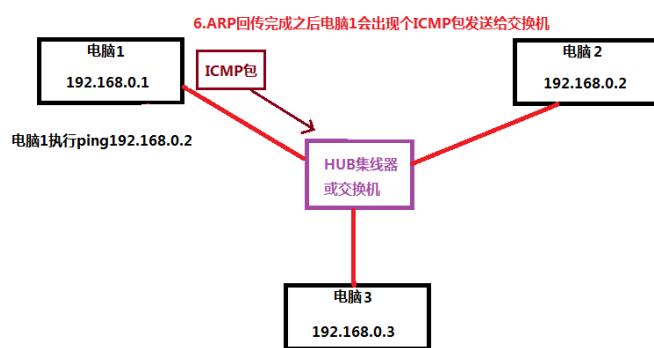
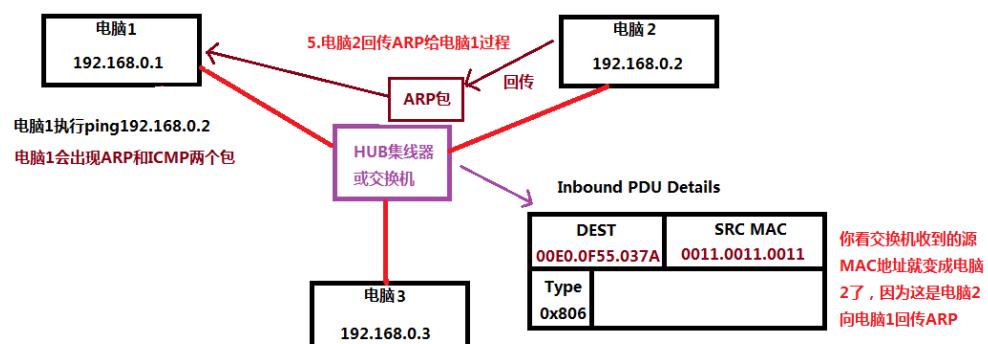
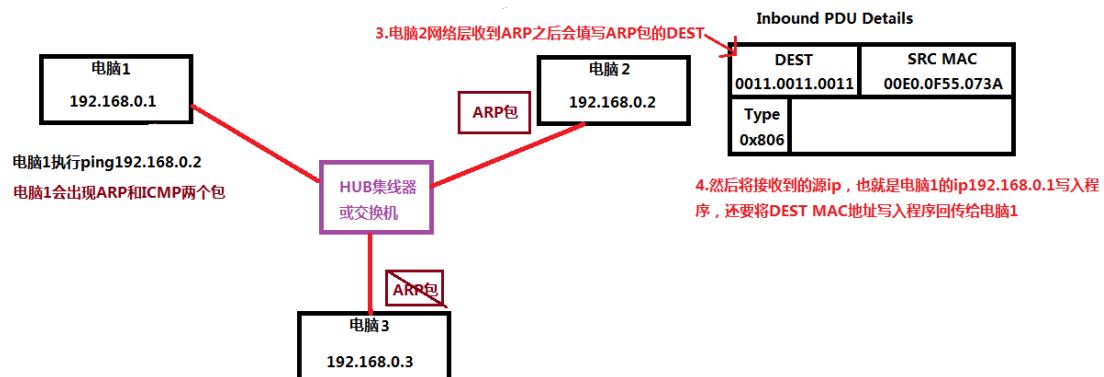
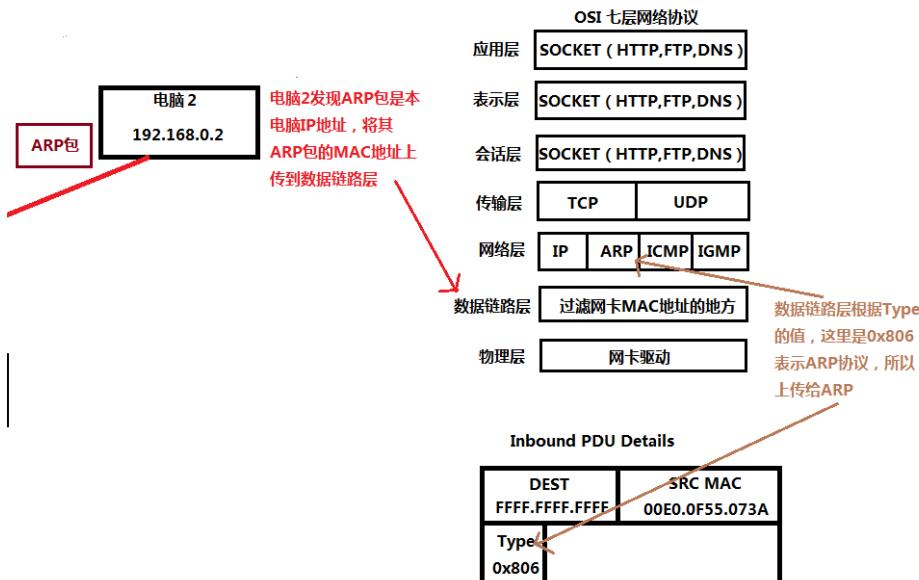


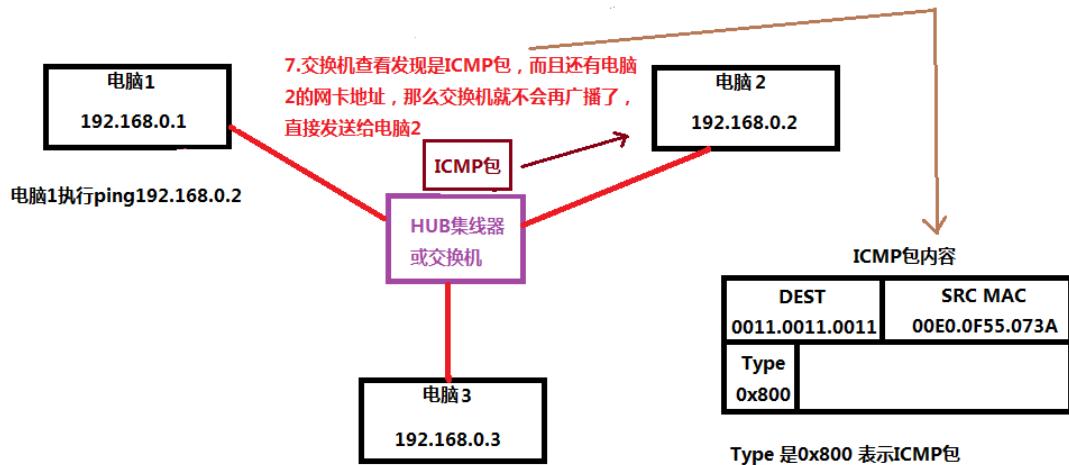
使用192.168.0.1去ping192.168.0.2或者192.168.0.3都能ping通，这是因为电脑1,2,3都在同一个网段，也就是都在192.168.0.....这个ip的第3位0就是网段

2. 电脑1去ping电脑2通信过程是怎么样的？

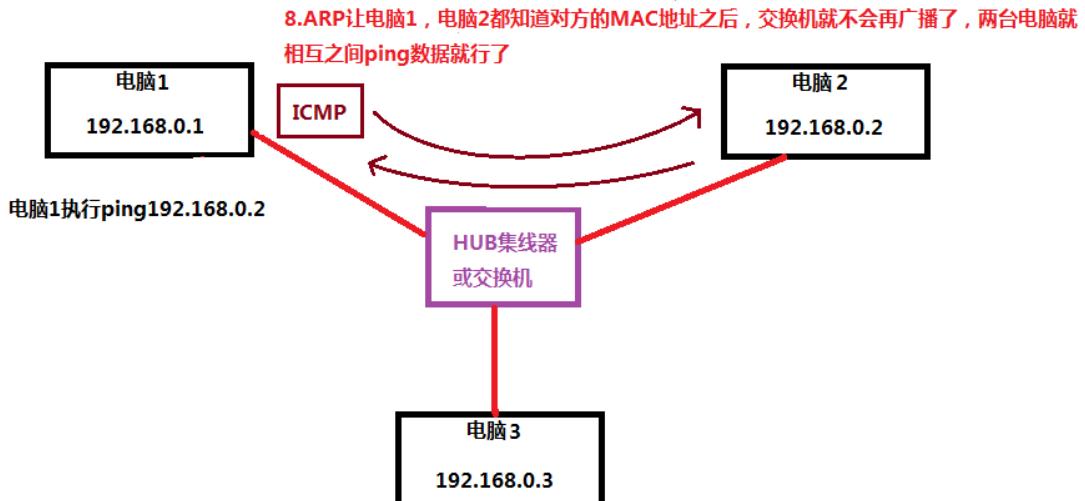


数据链路层会上传给谁呢？





现在我思考后发现，前面的 UDP 广播传输失败会不会是没有设置网卡 MAC 地址为 FFFF.FFFF.FFFF 的原因，因为本机的 MAC 为 FFFF.FFFF.FFFF 才代表为广播。



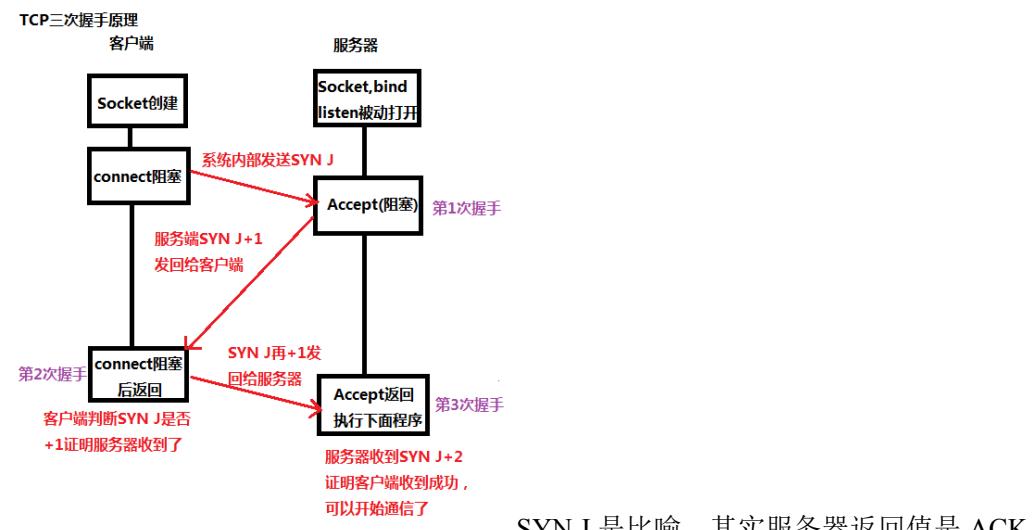
这就是网络通信的基本流程

需要知道对方电脑的 MAC 地址就用 ARP

有了 ARP 才可以用 ICMP 实现 ping 的功能，所以 ping 的实现其实是用 ICMP 协议。ARP 是不管什么协议都要用 ARP 先获取对方的 MAC 地址，才能实现你自己的网络协议。

ARP 就是根据 IP 找 MAC 地址，RARP 就是根据 MAC 地址找 IP

TCP 三次握手，四次挥手原理



对三次握手进行更深入的理解



这就是TCP包格式

SRC PORT:1028	DEST PORT: 80
SEQUENCE NUM:0	
ACK NUM:0	第1次发的包ACK为0
SYN	第1次发的包SYN为空



这就是TCP包格式

SRC PORT:1028	DEST PORT: 80
SEQUENCE NUM:0	
ACK NUM: 1	第2次服务器将ACK+1
SYN+ACK	所以第2次握手的包就包含了 SYN第1次客户端发的+ACK 服务器回复的

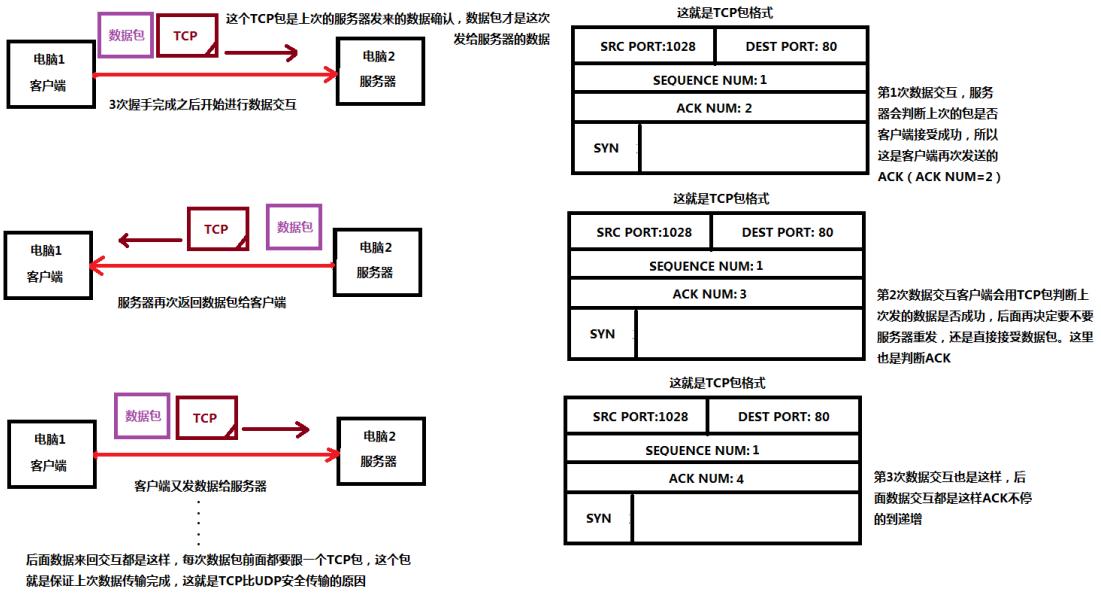


这就是TCP包格式

SRC PORT:1028	DEST PORT: 80
SEQUENCE NUM: 1	第3次TCP包的序列就会+1，服务器回判断 这个变量确定客户端收到第2次的ACK了
ACK NUM: 1	第3次服务器返回的ACK+1不变
SYN	这里回到SYN表示客户端发的同步

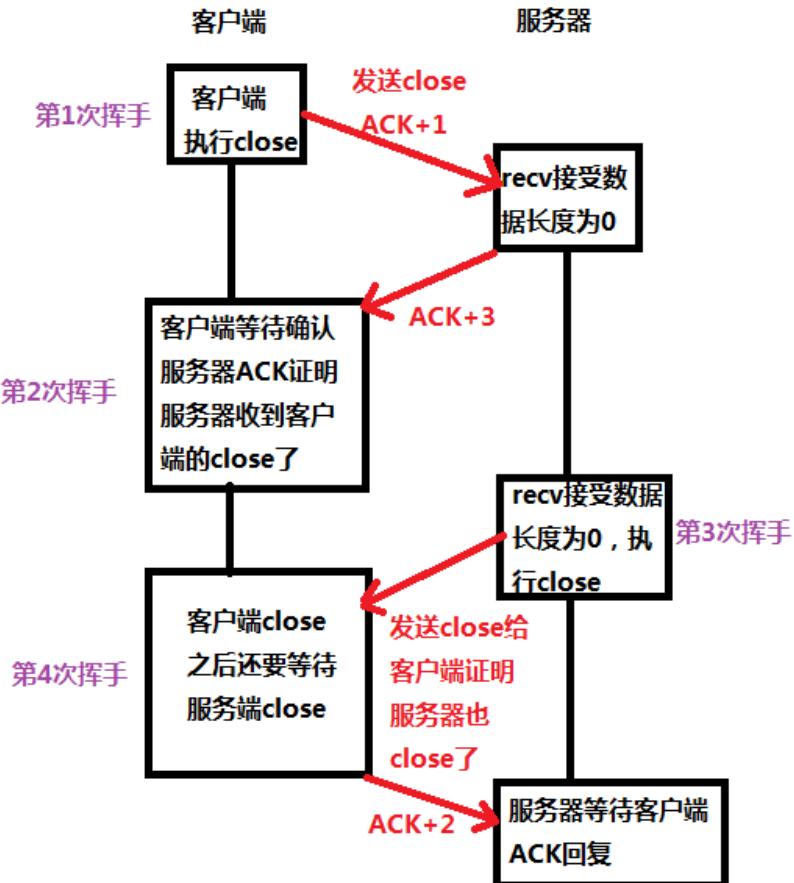
所有第3次握手只要TCP包包含SEQUENCE为1, ACK位1, 服务器就认为3次握手成功开始数据包通信。数据包可以是HTTP协议, FTP协议, MQTT协议...等等各种协议。

## TCP 三次握手成功之后就接着进行数据包交互发送



## TCP 四次挥手原理

### TCP四次挥手逻辑



## 单进程服务器商用案例

setsockopt(SOL\_SOCKET,SO\_REUSEADDR,1) //服务器端口重启函数使用

```
import socket
from socket import * #tcp头文件不能用socket, 要用from socket import *
#因为套接字创建不是用socket.socket, 而是直接socket
import time

tcpserver = socket(AF_INET,SOCK_STREAM) #tcp记住是SOCK_STREAM
tcpserver.setsockopt(SOL_SOCKET,SO_REUSEADDR,1)#比如我服务器程序突然结束了, 重启服务器时不会导致服务器指定的端口无法使用
tcpserver.bind(('',8888))#IP地址 写引号 '' 的好处是我换一台服务器, 我服务器程序不用从新输入ip
tcpserver.listen(5)

while True:
    newsocket,addr = tcpserver.accept()

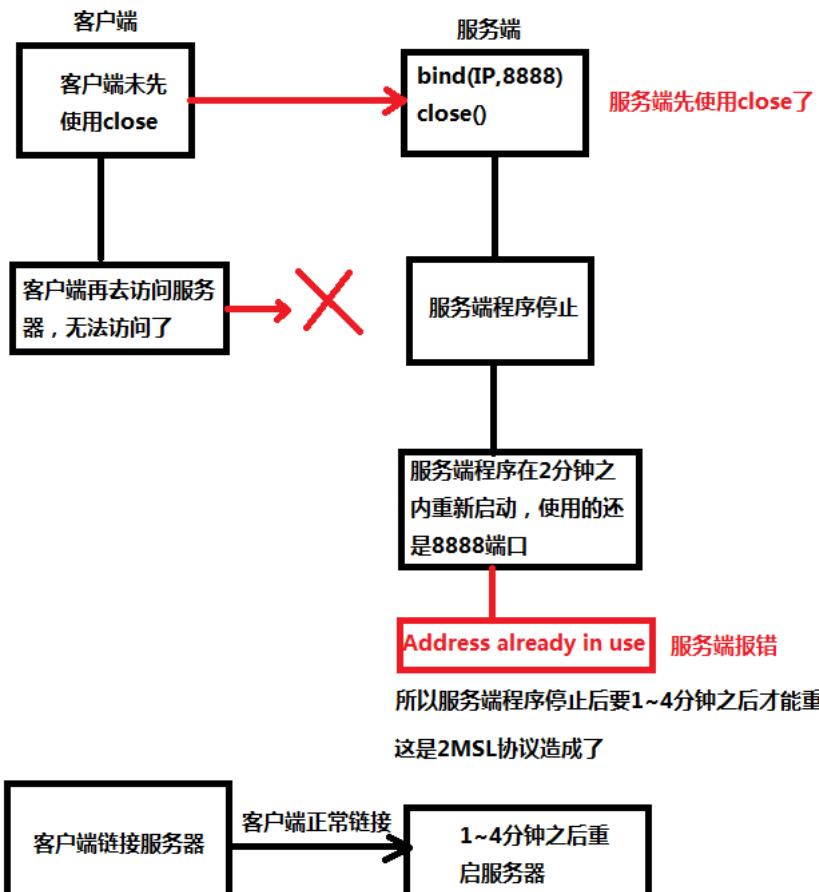
    try:
        while True:
            recvdata = newsocket.recv(1024)
            if len(recvdata) > 0: #接受数据>0 表示客户端有数据, 如果<0表示客户端先执行close了
                print("addr = %s data =%s" %(addr,recvdata))
            else:
                print("client close")
                break
    finally:
        newsocket.close()
tcpserver.close()
```

比如我队列收到 5 个客户端电脑, 我把电脑 1 取出来建立新 newsocket

然后电脑 1 客户端一直在给我发数据, 我就无法退出这个循环执行 finally

这时候无法再次循环执行 accept 取出队列的电脑 2 客户端建立 newsocket, 那么电脑 2, 3, 4, 5 客户端就一直在 connect 那儿等待, 这种电脑 1 死着服务器不放的程序就不对。下面用多进程/多线程来解决。

SO\_REUSEADDR参数一般是针对服务端使用



为了解决服务器不能马上重启同一端口程序的问题，

我们使用 setsockopt(SOL\_SOCKET,SO\_REUSEADDR,1) 来解决

在 bind 之前使用 setsockopt(...,SO\_REUSEADDR, 1) 这样就可以在服务器关闭之后不需要等待 2 分钟以上，直接马上重启服务器。

其实如果服务器程序每次重启使用不同的端口就不会有 2MSL 协议造成的问题，但是服务器程序一般都是固定端口。

用多进程解决多台客户端链接服务器的问题  
multiprocessing.Process(进程函数, 函数参数)  
需要导入 import multiprocessing 库

### 服务端

```
import socket
from socket import * #tcp头文件不能用socket, 要用from socket import *
#因为套接字创建不是用socket.socket, 而是直接socket
import time
import multiprocessing

def Multclient(newsocket,Addr):
    while True:
        recvdata = newsocket.recv(1024)
        if len(recvdata) > 0: #接受数据>0 表示客户端有数据, 如果<0表示客户端先执行close了
            print("addr = %s data %s" %(Addr,recvdata))
        else:
            print("client close")
            break
    newsocket.close()

def main():
    tcpserver = socket(AF_INET,SOCK_STREAM) #tcp记住是SOCK_STREAM
    tcpserver.setsockopt(SOL_SOCKET,SO_REUSEADDR,1) #比如我服务器程序突然结束了, 重启服务器时不会导致服务器指定的端口无法使用
    tcpserver.bind(('',8888)) #IP地址'写引号'的好处是我换一台服务器, 我服务器程序不用从新输入ip
    tcpserver.listen(5)

    try:
        while True:
            newsocket,addr = tcpserver.accept() #每次循环到这里就等待新客户端链接上来
            xclinet =multiprocessing.Process(target = Multclient,args = (newsocket,addr)) #给每次链接的新客户端创建一个对应的进程
            xclinet.start() #启动该客户端进程
            newsocket.close()
    finally:
        tcpserver.close()

if __name__ == '__main__':
    main()
```

2.子进程会去自动处理这个链接上来的客户端数据, 那么主进程可以继续去等待新客户端链接, 这就是并发服务器

3.因为是复制的一份新的 newsocket 客户端对象, 所以子进程处理完客户端对象一定要 close 这份复制的 newsocket

4.既然说 newsocket 是复制给子进程的, 那么主进程这个 newsocket 变量就必须 close 掉, 然后循环到 accept 创建新 newsocket, 如此循环。

5.这里用 finally 是因为在执行 try 死循环的时候很有可能认为突然的 ctrl+c 关闭程序, 这时候 try 会发现有异常, 从而在 ctrl+c 的时候去执行 finally 关闭 tcp 的 socket

这就是多进程的优势, 创建一个进程为一个客户端服务, 再创建一个进程为另一个客户端服务。

### 客户端程序

```
for i in range(10): #我要链接10次服务器
    s = socket(AF_INET,SOCK_STREAM)
    s.connect(("192.168.127.128",8888))
    s.send("abcdefg12345".encode('utf-8'))
    print("链接上服务器次数 %d" %i)
    s.close()
```

```
root@ubuntu:/home/xzz/pytest# python3 tcp3.py
addr = ('192.168.127.1', 54705) data b'abcdefg12345'
addr = ('192.168.127.1', 54706) data b'abcdefg12345'
'client close
client close
addr = ('192.168.127.1', 54707) data b'abcdefg12345'
client close
addr = ('192.168.127.1', 54708) data b'abcdefg12345'
client close
addr = ('192.168.127.1', 54709) data b'abcdefg12345'
client close
addr = ('192.168.127.1', 54710) data b'abcdefg12345'
client close
addr = ('192.168.127.1', 54711) data b'abcdefg12345'
client close
addr = ('192.168.127.1', 54712) data b'abcdefg12345'
client close
addr = ('192.168.127.1', 54713) data b'abcdefg12345'
client close
addr = ('192.168.127.1', 54714) data b'abcdefg12345'
client close
```

从这里就看出来了, 客户端在服务器的链接是并行处理的, 所以多进程同时执行和同时关闭的可能性很大, 所以这里连续了两个 close, 也证明了客户端程序是被并行处理的。

## 多线程解决客户端并发问题

服务端

```
import socket
from socket import * #tcp头文件不能用socket, 要用from socket import *
#因为套接字创建不是用socket.socket, 而是直接socket
import time
import _thread

def MultClient(newsocket,Addr):
    while True:
        recvdata = newsocket.recv(1024)
        if len(recvdata) > 0: #接受数据>0 表示客户端有数据, 如果<0表示客户端先执行close了
            print("addr = %s data =%s" %(Addr,recvdata))
        else:
            print("client close")
            break

    newsocket.close()

def main():
    tcpserver = socket(AF_INET,SOCK_STREAM) #tcp记住是SOCK_STREAM
    tcpserver.setsockopt(SOL_SOCKET,SO_REUSEADDR,1) #比如我服务器程序突然结束了, 重启服务器时不会导致服务器指定的端口无法使用
    tcpserver.bind(('','8888')) #IP地址写引号 '' 的好处是我换一台服务器, 我服务器程序不用从新输入ip
    tcpserver.listen(5)

    try:
        while True:
            newsocket,addr = tcpserver.accept() #每次循环到这里就等待新客户端链接上来
            xclinet = _thread.start_new_thread(MultClient,(newsocket,addr)) #给每次链接的新客户端创建一个对应的线程
            #newsocket.close()
    finally:
        tcpserver.close()

if __name__ == "__main__":
    main()
```

这里没变

这里也没变

唯一变化的就是把进程改成了线程函数启动

但是注意, 在线程中所有变量都在同一个进程, 所以这里不能像进程那样关闭掉客户端的套接字, 因为进程是复制了一个客户端套接字给子进程的, 相当于子进程是新创建的独立内存空间来存放这个客户端套接字, 如果主进程死了, 子进程还是单独运行的, 在我的《linux应用编程》文档中介绍过, 所以线程的所有变量都在一个进程的内存中, 所以不能在这里关闭客户端 socket

客户端

```
for i in range(10): #我要链接10次服务器
    s = socket(AF_INET,SOCK_STREAM)
    s.connect(("192.168.127.128",8888))
    s.send("abcdefg12345".encode('utf-8'))
    print("链接上服务器次数 %d" %i)
    s.close()
```

```
root@ubuntu:/home/xzz/pytest# python3 tcp3.py
addr = ('192.168.127.1', 54935) data b'abcdefg12345'
client close
addr = ('192.168.127.1', 54936) data b'abcdefg12345'
client close
addr = ('192.168.127.1', 54937) data b'abcdefg12345'
client close
addr = ('192.168.127.1', 54938) data b'abcdefg12345'
client close
addr = ('192.168.127.1', 54939) data b'abcdefg12345'
client close
addr = ('192.168.127.1', 54940) data b'abcdefg12345'
client close
addr = ('192.168.127.1', 54941) data b'abcdefg12345'
client close
addr = ('192.168.127.1', 54942) data b'abcdefg12345'
client close
addr = ('192.168.127.1', 54943) data b'abcdefg12345'
client close
addr = ('192.168.127.1', 54944) data b'abcdefg12345'
client close
```

## 单进程实现并发服务器，select 和 epoll 使用

### 单进程并发服务器逻辑

setblocking(False) 默认 accept 是阻塞，但是设置 False 之后 accept 就是非阻塞

```
import socket
from socket import * #tcp头文件不能用socket, 要用from socket import *
#因为套接字创建不是用socket.socket, 而是直接socket
import time

tcpserver = socket(AF_INET,SOCK_STREAM) #tcp记住是SOCK_STREAM
tcpserver.bind(("192.168.127.128",8888))#因为是服务器, 绑定本机IP和端口
tcpserver.listen(5)
tcpserver.setblocking(False) #设置accept为非阻塞状态
要在 accept 之前设置非阻塞

while True:
    newsocket,addr = tcpserver.accept()
    print(addr)
    time.sleep(1)
```

因为 accept 是非阻塞状态, 所以就算客户端没有 connect 链接服务器, 程序也会继续向下执行

```
root@ubuntu:/home/xzz/pytest# python3 tcp4.py
Traceback (most recent call last):
  File "tcp4.py", line 16, in <module>
    newsocket,addr = tcpserver.accept()
  File "/usr/lib/python3.5/socket.py", line 195, in accept
    fd, addr = self._accept()
BlockingIOError: [Errno 11] Resource temporarily unavailable
```

我们发现运行报错 BlockingIOError, 这就是 accept 因为是非阻塞, 在执行 accept 之后如果没有客户端链接就会报出异常, 我们就是要利用这个异常来做并发服务。

### 服务器修改之后

```
import socket
from socket import * #tcp头文件不能用socket, 要用from socket import *
#因为套接字创建不是用socket.socket, 而是直接socket
import time

tcpserver = socket(AF_INET,SOCK_STREAM) #tcp记住是SOCK_STREAM
tcpserver.bind(("192.168.127.128",8888))#因为是服务器, 绑定本机IP和端口
tcpserver.listen(5)
tcpserver.setblocking(False) #设置accept为非阻塞状态

while True:
    try:
        newsocket,addr = tcpserver.accept()
        print(addr)
        time.sleep(1)
    except:
        pass
    else:
        print("connect new client %s" %str(addr))
```

1. 使用 try 来让异常发生时不执行这句代码, 这样就不会报错

2. 异常发生时执行 except, except 下我什么都不做, 也就是没有客户端链接, 我就什么都不做。

3. 有客户端链接我就执行 else

```
root@ubuntu:/home/xzz/pytest# python3 tcp4.py
```

你看服务器启动后没有客户端链接, 就循环执行 accept, 发现异常就跳到 except 执行, 什么都不做所以没有答应。但是实际是在循环执行, accept 没有阻塞。

客户端发送链接服务器

```
for i in range(10):#我要链接10次服务器
    s = socket(AF_INET,SOCK_STREAM)
    s.connect(("192.168.127.128",8888))
    s.send("abcdefg12345".encode('utf-8'))
    print("链接上服务器次数 %d" %i)
    s.close()
```

```
root@ubuntu:/home/xzz/pytest# python3 tcp4.py
('192.168.127.1', 52760)
connect new client ('192.168.127.1', 52760)
('192.168.127.1', 52761)
connect new client ('192.168.127.1', 52761)
('192.168.127.1', 52762)
connect new client ('192.168.127.1', 52762)
('192.168.127.1', 52763)
```

你看服务端收到客户端 connect 就执行 else 程序

再来分析下服务端代码

```
tcpserver = socket(AF_INET,SOCK_STREAM) #tcp记住是SOCK_STREAM
tcpserver.bind(("192.168.127.128",8888))#因为是服务器，绑定本机IP和端口
tcpserver.listen(5)

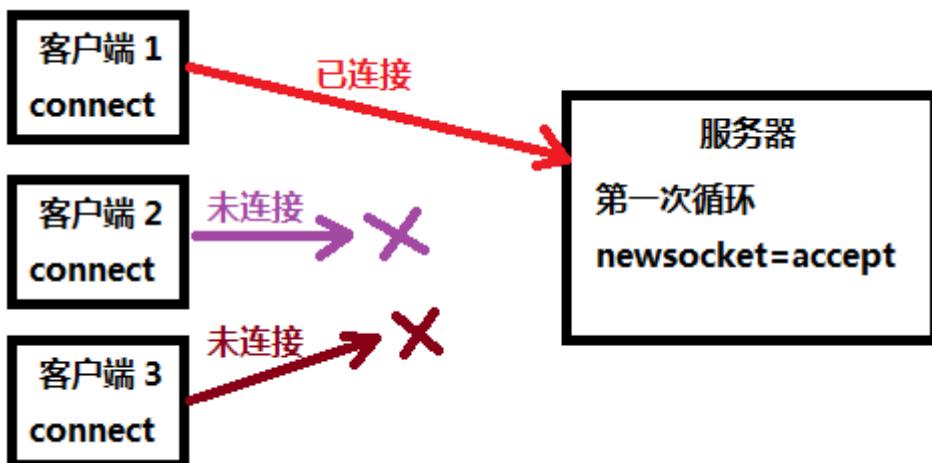
tcpserver.setblocking(False) #设置accept为非阻塞状态

while True:
    try:
        newsocket,addr = tcpserver.accept()
        print(addr)
        time.sleep(1)
    except:
        pass
    else:
        print("connect new client %s" %str(addr))
```

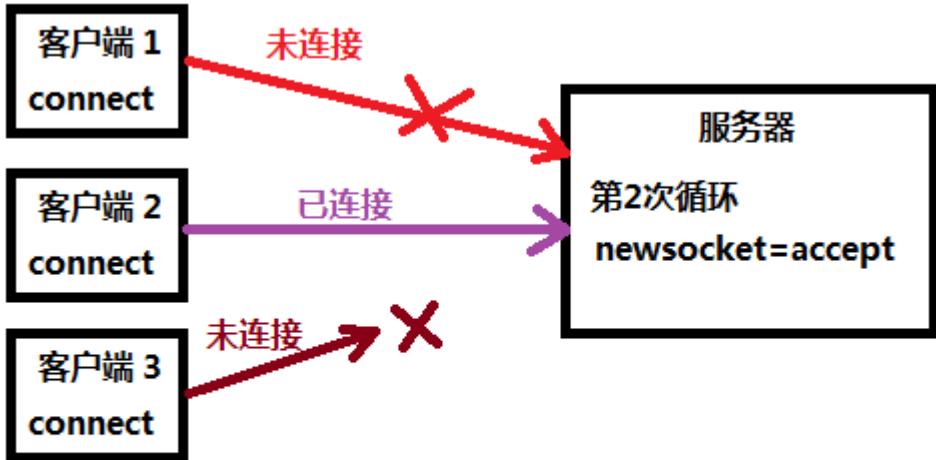
实际就是每次循环回来 accept 发现有一新的客户端链接，就会创建新的 newsocket，所以这个 newsocket 变量，每次循环回来都是新的

这里处理的 addr 和 newsocket 都是这次链接的客户端对象，而不是上次循环的客户端对象

这种循环方式有个问题，就是每次都是新的 newsocket，这样就会覆盖上一次的 newsocket，会出现怎么现象呢？



第 1 次客户端 1 链接上服务器很正常



我发现第2次服务器循环accept，客户端1并没有执行close()，  
怎么连接客户端2的时候把客户端1又断开了，不对啊

```
tcpserver = socket(AF_INET,SOCK_STREAM) #tcp记住是SOCK_STREAM
tcpserver.bind(("192.168.127.128",8888))#因为是服务器，绑定本机IP和端口
tcpserver.listen(5)
tcpserver.setblocking(False) #设置accept为非阻塞状态

while True:
    try:
        newsocket,addr = tcpserver.accept()
        print(addr)
        time.sleep(1)
    except:
        pass
    else:
        print("connect new client %s" %str(addr))
```

这是因为第2次循环 accept 创建了新的对象，把上一次客户端 1 的 newsocket 给覆盖了，所以现在 newsocket 是客户端 2 的对象

如果客户端 3 链接上服务器，就会把客户端 2 断开，所以就成了客户端 1,2 都被断开，客户端 3 链接上的现象。为了解决这个问题，你必须每次循环，都把新的 newsocket 放入列表保存。

```
tcpserver = socket(AF_INET,SOCK_STREAM) #tcp记住是SOCK_STREAM
tcpserver.bind(("192.168.127.128",8888))#因为是服务器，绑定本机IP和端口
tcpserver.listen(5)
tcpserver.setblocking(False) #设置accept为非阻塞状态
clientList = []
```

保存新的客户端列表

我将新客户端 1 放入列表，下次  
服务器接受客户端 2 就不会覆盖  
客户端 1 的套接字对象

这里有个问题，接受客户端数据的时候  
会阻塞，导致 for 循环卡在这里

```
for ClientSocket,ClientAddr in clientList:
    data = ClientSocket.recv(1024)
    print(data)#新客户端的数据处理
    ClientSocket.close()#数据处理后没有必要留着新客户端套接字，所以将其关闭
```

```

tcpserver = socket(AF_INET,SOCK_STREAM) #tcp记住是SOCK_STREAM
tcpserver.bind(("192.168.127.128",8888))#因为是服务器，绑定本机IP和端口
tcpserver.listen(5)
tcpserver.setblocking(False) #设置accept为非阻塞状态
clientList = []

while True:
    try:
        newsocket,addr = tcpserver.accept()
        print(addr)
        time.sleep(1)
    except:
        pass
    else:
        print("connect new client %s" %str(addr))
        newsocket.setblocking(False)
        clientList.append((newsocket,addr)) #保存新客户端的元祖变量

    for ClientSocket,ClientAddr in clientList:
        data = ClientSocket.recv(1024)
        print(data)#新客户端的数据处理
        ClientSocket.close()#数据处理后没有必要留着新客户端套接字，所以将其关闭

```

在新客户端套接字放入列表之前，给客户端套接字修改成非阻塞状态

```

root@ubuntu:/home/xzz/pytest# python3 tcp4.py
('192.168.127.1', 53271)
connect new client ('192.168.127.1', 53271)
b'abcdefg12345'
('192.168.127.1', 53272)
connect new client ('192.168.127.1', 53272)
Traceback (most recent call last):
  File "tcp4.py", line 29, in <module>
    data = ClientSocket.recv(1024)
OSErrror: [Errno 9] Bad file descriptor

```

发现接受两次数据后就报异常了

这其实和前面的 accept 一样的，这里不过是 recv 执行时发现客户端没有发数据过来，所以报异常，我们也是利用这个异常配合 try 来解决。

```

while True:
    try:
        newsocket,addr = tcpserver.accept()
        print(addr)
        time.sleep(1)
    except:
        pass
    else:
        print("connect new client %s" %str(addr))
        newsocket.setblocking(False)
        clientList.append((newsocket,addr)) #保存新客户端的元祖变量

    for ClientSocket,ClientAddr in clientList:
        try:
            data = ClientSocket.recv(1024)
        except:
            pass
        else:
            print(data)#新客户端的数据处理
            ClientSocket.close()#数据处理后没有必要留着新客户端套接字，所以将其关闭

```

加入 try，问题得到解决

```

root@ubuntu:/home/xzz/pytest# python3 tcp4.py
('192.168.127.1', 53337)
^C('192.168.127.1', 53338)
connect new client ('192.168.127.1', 53338)
b'abcdefg12345'
('192.168.127.1', 53339)
connect new client ('192.168.127.1', 53339)
b'abcdefg12345'
('192.168.127.1', 53340)
connect new client ('192.168.127.1', 53340)
b'abcdefg12345'
('192.168.127.1', 53341)

```

你看数据正常了。

```

while True:
    try:
        newsocket,addr = tcpserver.accept()
        print(addr)
        time.sleep(1)
    except:
        pass
    else:
        print("connect new client %s" %str(addr))
        newsocket.setblocking(False)
        clientList.append((newsocket,addr)) #保存新客户端的元祖变量

    for ClientSocket,ClientAddr in clientList:
        try:
            data = ClientSocket.recv(1024)
        except:
            pass
        else:
            print(data)#新客户端的数据处理
            ClientSocket.close()#数据处理后没有必要留着新客户端套接字，所以将其关闭
            clientList.remove((ClientSocket,ClientAddr))

```

为了节省内存可以把不用的套接字变量删除了

这就是单进程用非阻塞方式实现并发服务器的思路。但是我们最好用 select 和 epoll 这种成熟的库比较安全。

## select 函数版本并发服务器

readable, writeable, exceptionable = select.select(rlist, wlist, xlist, time)

rlist: 是我们要监听的可读套接字列表  
wlist: 是我们要监听的可写套接字列表  
xlist: 是我们需要监听异常的套接字列表  
time: 阻塞时间限制

readable: 如果形参 input 套接字满足了可读条件就返回数据给 readable 变量

writeable: 如果形参 output 套接字满足了可写条件就返回数据给 writeable 变量

如果套接字有异常, 就返回给 exceptionable 变量

需要使用 import select 库

```

import socket
from socket import * #tcp头文件不能用socket, 要用from socket import *
import time
import select

tcpserver = socket(AF_INET,SOCK_STREAM) #tcp记住是SOCK_STREAM
tcpserver.bind(("192.168.127.128",8888))#因为是服务器, 绑定本机IP和端口
tcpserver.listen(5)

while True:
    read,write,exp = select.select([tcpserver],[],[])
    for i in read:
        newsocket,addr = i.accept()
        print(addr)

```

现在我就只监听 socket 接收的链接功能, 看看有没有客户端链接, 这是列表的一种使用方法, 当然你也可以在外部定义列表, 这里列表只有一个成员 tcpserver 套接字, 所以现在 select 只监听 tcpserver 套接字是否有客户端链接过来

如果有客户端链接过来, 就跳过阻塞, 不然程序就阻塞到这里

跳过阻塞, 就会返回列表里面的套接字给 read, 这里把 read 变量里的套接字获取出来, 用 accept 分配 newsocket 来指定这个链接的客户端

```

root@ubuntu:/home/xzz/pytest# python3 tcp5.py
('192.168.127.1', 56044)
('192.168.127.1', 56045)
('192.168.127.1', 56046)
('192.168.127.1', 56047)
('192.168.127.1', 56048)

```

收到 5 个客户端的链接

我们不是要用 select 做非阻塞并发服务器吗？如果 select 阻塞还有什么意义，下面解决阻塞问题

```
tcpserver = socket(AF_INET,SOCK_STREAM) #tcp记住是SOCK_STREAM
tcpserver.bind(("192.168.127.128",8888))#因为是服务器，绑定本机IP和端口
tcpserver.listen(5)

while True:
    read,write,exp = select.select([tcpserver],[],[],3)

    for i in read:
        newsocket,addr = i.accept()
        print(addr)
        print("=====")
```

前面不是说这里有个 time 参数吗，就是指定 select 阻塞几秒后，就算没有接收到客户端数据，也跳过阻塞，这里是 3 秒跳过阻塞

这里打印====号就是证明实现了延时非阻塞

```
root@ubuntu:/home/xzz/pytest# python3 tcp5.py
=====
服务器启动时阻塞的
root@ubuntu:/home/xzz/pytest# python3 tcp5.py
=====
=====
=====
三秒之后就不阻塞了
```

```
root@ubuntu:/home/xzz/pytest# python3 tcp5.py
=====
('192.168.127.1', 56171)
=====
('192.168.127.1', 56172)
=====
('192.168.127.1', 56173)
=====
('192.168.127.1', 56174)
=====
('192.168.127.1', 56175)
=====
```

如果我在定时阻塞的情况下，有客户端发数据，select 也会马上跳过阻塞，不受 3 秒阻塞延时影响

这就解决了并发服务器的第一步，单进程非阻塞

### select 监听多个套接字

```
tcpserver = socket(AF_INET,SOCK_STREAM) #tcp记住是SOCK_STREAM
tcpserver.bind(("192.168.127.128",8888))#因为是服务器，绑定本机IP和端口
tcpserver.listen(5)

readin = [tcpserver,]

while True:
    read,write,exp = select.select(readin,[],[],3)

    for i in read:
        newsocket,addr = i.accept()
        print(addr)
        readin.append(newsocket)
```

初步理解是将每个新客户端的 newsocket 放入监听列表

```

root@ubuntu:/home/xzz/pytest# python3 tcp5.py
('192.168.127.1', 57623)
Traceback (most recent call last):
  File "tcp5.py", line 21, in <module>
    newsocket,addr = i.accept()
  File "/usr/lib/python3.5/socket.py", line 195, in accept
    fd, addr = self._accept()
OSErrror: [Errno 22] Invalid argument
root@ubuntu:/home/xzz/pytest#

```

这是为什么呢？

下面进行代码修改，对 select 机制进行深入分析

```

tcpserver = socket(AF_INET,SOCK_STREAM) #tcp记住是SOCK_STREAM
tcpserver.bind(("192.168.127.128",8888))#因为是服务器，绑定本机IP和端口
tcpserver.listen(5)

readin = [tcpserver,]

while True:
    read,write,exp = select.select(readin,[],[],3)
    for i in read:
        if i == tcpserver:
            newsocket,addr = i.accept()
            print(addr)
            print(tcpserver)#打印链接上来的客户端
            readin.append(newsocket)
        else:
            data = i.recv(1024)
            print(data)
            time.sleep(1)

```

```

tcpserver = socket(AF_INET,SOCK_STREAM) #tcp记住是SOCK_STREAM
tcpserver.bind(("192.168.127.128",8888))#因为是服务器，绑定本机IP和端口
tcpserver.listen(5)

readin = [tcpserver,]

while True:
    read,write,exp = select.select(readin,[],[],3)
    for i in read:
        if i == tcpserver:
            newsocket,addr = i.accept()
            print(addr)
            print(tcpserver)#打印链接上来的客户端
            readin.append(newsocket)
        else:
            data = i.recv(1024)
            print(data)
            time.sleep(1)

```

```

tcpserver = socket(AF_INET,SOCK_STREAM) #tcp记住是SOCK_STREAM
tcpserver.bind(("192.168.127.128",8888))#因为是服务器，绑定本机IP和端口
tcpserver.listen(5)

readin = [tcpserver,]

while True:
    read,write,exp = select.select(readin,[],[],3)
    for i in read:
        if i == tcpserver:
            newsocket,addr = i.accept()
            print(addr)
            print(tcpserver)#打印链接上来的客户端
            readin.append(newsocket)
        else:
            data = i.recv(1024)
            print(data)
            time.sleep(1)

```

第3次循环，发现客户端2还是没有链接上，这时候列表还是处于监听状态[tcpserver, 客户端1]，但是突然客户端1发数据过来了，记住是数据，而不是 connect，所以列表中的 [tcpserver, 客户端1] 客户端1 被触发了，把客户端1返回给 read

因为 read 赋值给 i 的是客户端1的数据套接字，而不是 connect 套接字，所以如果不执行，直接执行 else，i 现在就是客户端1的数据套接字变量，所以用 recv 取出的是客户端1发来的数据

```

tcpserver = socket(AF_INET,SOCK_STREAM) #tcp记住是SOCK_STREAM
tcpserver.bind(("192.168.127.128",8888))#因为是服务器，绑定本机IP和端口
tcpserver.listen(5)

readin = [tcpserver,]

while True:
    read,write,exp = select.select(readin,[],[],3)
    for i in read:
        if i == tcpserver:
            newsocket,addr = i.accept()
            print(addr)
            print(tcpserver)#打印链接上来的客户端
            readin.append(newsocket)
        else:
            data = i.recv(1024)
            print(data)
            time.sleep(1)

```

第5次循环，突然有客户端2链接上来了，返回客户端2的套接字给 read

将客户端2放入监听列表 [tcpserver, 客户端1, 客户端2]，这下 select 就监听三个 socket 了，tcpserver, 客户端1, 客户端2

如法炮制，这时候如果有数据来了，或者是新客户端链接上来，select的监听列表会 [tcpserver, 客户端1, 客户端2, ...] 查询，是客户端的 connect 吗？如果是就返回 tcpserver。如果是数据，select 又会确定是客户端1的数据还是客户端2的数据。所以 select 每次循环都是返回列表里面的东西，只是看是数据还是链接。

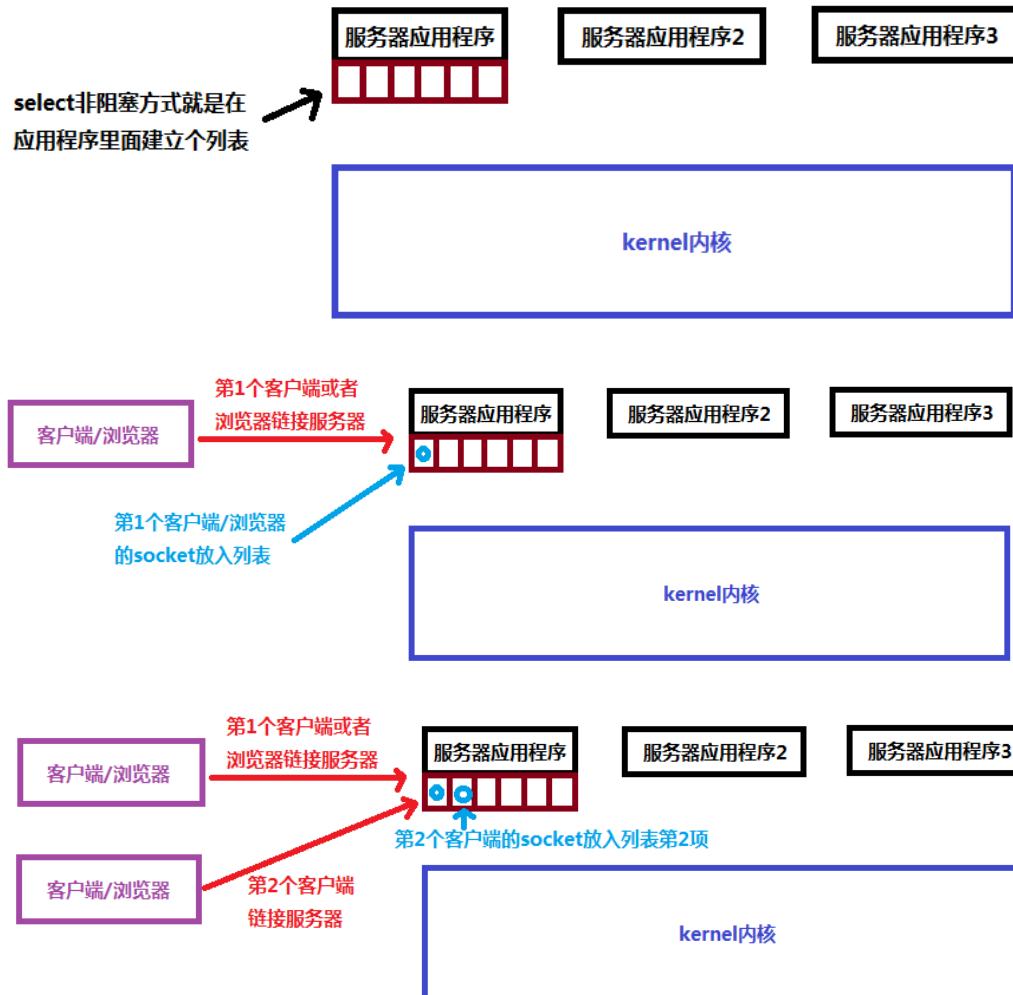
为什么没有给 newsocket 在执行完客户端服务之后加入 close()？

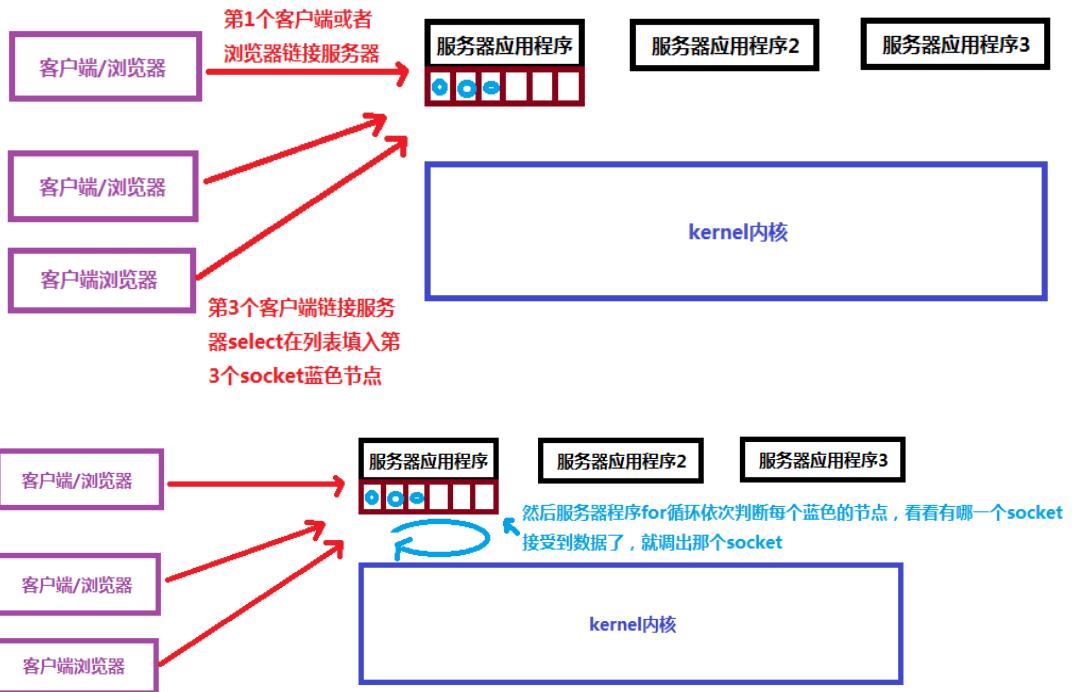
这是因为 select 自带 close(),一旦客户端执行 close(),那么 select 就会去列表里面删除该客户端的元素。

```
root@ubuntu:/home/xzz/pytest# python3 tcp5.py
('192.168.127.1', 58299)
<socket.socket fd=3, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('192.168.127.128', 8888)>
b'abcdefg12345'
b''
('192.168.127.1', 58301)
<socket.socket fd=3, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('192.168.127.128', 8888)>
b''
('192.168.127.1', 58303)
<socket.socket fd=3, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('192.168.127.128', 8888)>
b''
b'abcdefg12345'
('192.168.127.1', 58304)
<socket.socket fd=3, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('192.168.127.128', 8888)>
b''
b''
b'abcdefg12345'
('192.168.127.1', 58305)
<socket.socket fd=3, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('192.168.127.128', 8888)>
b''
b''
b'abcdefg12345'
b''
b''
b''
b'abcdefg12345'
b''
b''
b''
b''
b''
```

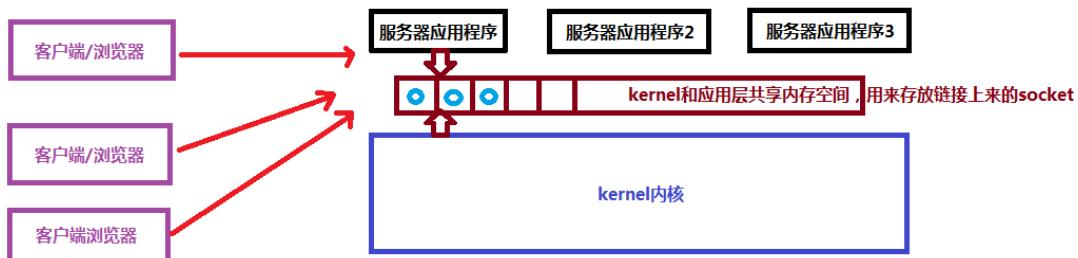
你看链接上和数据接受顺序有点乱，但是最终都是5个客户端链接，5个客户端数据，就是不知道数据对应客户端关系是否一致，还得考证

epoll 使用，解决 select 轮询效率低问题



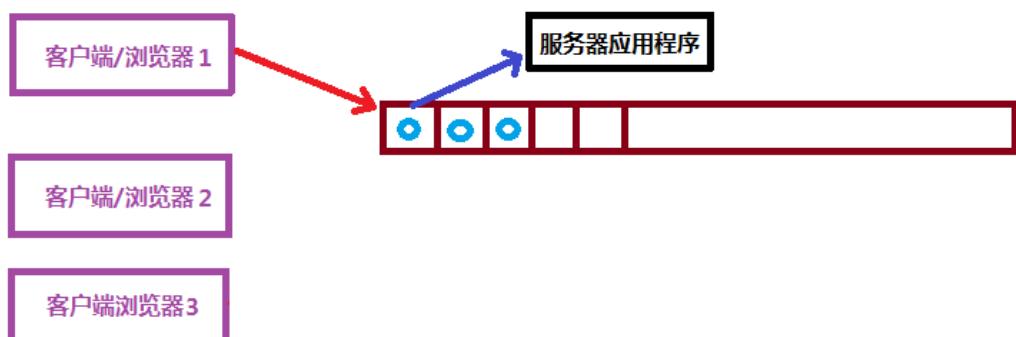


这种轮询方式效率就很低，下面用 epoll 来改善 select 非阻塞轮询效率低的情况。



kernel管理共享内存列表，在这种情况下就是某一个客户端发来数据，直接触发列表某项socket执行

比如客户端1发数据给服务器，服务器epoll建立的列表里面代表客户端1的  
socket触发给应用程序，类似单片机中断机制，也类似消息触发机制





忘了说一点



`register(fd 套接字文件描述符, eventmask 允许什么情况下触发)`

`fd:` 表示写入共享内存的描述符是哪一个？

`eventmask:` `select.EPOLLIN` 设置描述符为可读  
`select.EPOLLOUT` 设置描述符为可写

```

import socket
from socket import *
import select #epoll是封装在select库的

tcpserver = socket(AF_INET,SOCK_STREAM)
tcpserver.bind(("192.168.127.128",8888))#因为是服务器，绑定本机IP和端口
tcpserver.listen(10)
tcpserver.setblocking(False)

client_list = []
fd_dict = dict()

epoll = select.epoll() #创建应用层和kernel内核的共享内存
epoll.register(tcpserver.fileno(),select.EPOLLIN);#将tcp的监听描述符放入epoll共享内存列表，然后设置该监听节点为select.EPOLLIN 接受数据模式
while True:
    epoll_list = epoll.poll(); #epoll变量的poll默认会阻塞，直到内核检测到数据到来，触发应用层取消阻塞
    #返回是个列表[...], 如果os触发，可以一次返回多个客户端链接的socket给应用层，这些socket就放在返回的列表中。这些列表的每项由一个元祖[(),(),()]
    #fd:得到的是这次客户端连接对应的文件描述符
    #event:是指这个文件描述符到底是什么事件，是接收数据吗还是发送
    for fd,events in epoll_list:
        if fd == tcpserver.fileno():#确定现在触发的套接字是我设置的监听套接字，对比得到的文件描述符来确定是不是我监听套接字
            #如果是监听套接字被触发，证明有新的tcp链接进来
            newsocket,client_addr = tcpserver.accept()
            print(newsocket)
            print(client_addr)
            epoll.register(newsocket.fileno(),select.EPOLLIN);
            fd_dict[newsocket.fileno()] = newsocket
        elif events == select.EPOLLIN:
            recvdata = fd_dict[fd].recv(1024).decode("utf-8")
            if len(recvdata) > 0:
                print("====%s====" %recvdata)
            else:
                epoll.unregister(fd)
                fd_dict[fd].close()
                print("----close----")

```

这样监听描述符才能接受客户端发来的tcp链接或者是tcp数据



如果是一台客户端 1 电脑链接到我的 epoll 服务器，那么 `epoll.epoll` 就返回两个列表，一个列表是监听套接字，一个列表是客户端 1 的 socket



如果是两台电脑，客户端 1，客户端 2 同时链接上 epoll，那么 `epoll.epoll` 就返回 3 个列表，以此类推

```

while True:
    epoll_list = epoll.poll(); #epoll变量的poll默认会阻塞,直到内核检测到数据到来,触发应用层取消阻塞
    #返回是个列表[....],如果os触发,可以一次返回多个客户端连接的socket给应用层,这些socket就放在返回的列表中。这些列表的每项
    #fd:得到的是这次客户端链接对应的文件描述符
    #event:是指这个文件描述符到底是什么事件,是接收数据吗还是发送
    for fd,events in epoll_list:
        if fd == tcpserver.fileno():#确定现在触发的套接字是我设置的监听套接字,对比得到的文件描述符来确定是不是我监听套接字
            #如果是监听套接字被触发,证明有新的tcp连接进来
            newsocket,client_addr = tcpserver.accept()
            print(newsocket)
            print(client_addr)
            epoll.register(newsocket.fileno(),select.EPOLLIN);
            fd_dict[newsocket.fileno()] = newsocket
        elif events == select.EPOLIN:
            recvdata = fd_dict[fd].recv(1024).decode("utf-8")
            if len(recvdata) > 0:
                print("=====%s====" %recvdata)
            else:
                epoll.unregister(fd)
            fd_dict[fd].close()
            print("----close----")

```

然后我使用 for 循环去获取共享内存 epoll\_list 列表变量里面的数据  


如果 if 成功证明监听套接字收到客户端发来的链接请求  

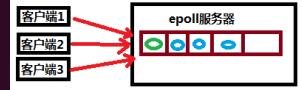

applet会把这次链接的客户端套接字获取到返回成元祖,所以我把这次链接的客户端socket写入共享内存  


然后设置这次链接客户端对应的 socket 为接受数据

```

while True:
    epoll_list = epoll.poll(); #epoll变量的poll默认会阻塞,直到内核检测到数据到来,触发应用层取消阻塞
    #返回是个列表[....],如果os触发,可以一次返回多个客户端连接的socket给应用层,这些socket就放在返回的列表中。这些列表的每项
    #fd:得到的是这次客户端链接对应的文件描述符
    #event:是指这个文件描述符到底是什么事件,是接收数据吗还是发送
    for fd,events in epoll_list:
        if fd == tcpserver.fileno():#确定现在触发的套接字是我设置的监听套接字,对比得到的文件描述符来确定是不是我监听套接字
            #如果是监听套接字被触发,证明有新的tcp连接进来
            newsocket,client_addr = tcpserver.accept()
            print(newsocket)
            print(client_addr)
            epoll.register(newsocket.fileno(),select.EPOLLIN);
            fd_dict[newsocket.fileno()] = newsocket
        elif events == select.EPOLIN:
            recvdata = fd_dict[fd].recv(1024).decode("utf-8")
            if len(recvdata) > 0:
                print("=====%s====" %recvdata)
            else:
                epoll.unregister(fd)
            fd_dict[fd].close()
            print("----close----")

```

如果没有客户端发数据,只有客户端在连接,那么 if 之后就返回 for 循环开头。再次提取共享内存的第 2 个列表或者后面的列表看看是 tcp 链接还是 tcp 数据,如果是 tcp 数据就执行 elif  


```

while True:
    epoll_list = epoll.poll(); #epoll变量的poll默认会阻塞,直到内核检测到数据到来,触发应用层取消阻塞
    #返回是个列表[....],如果os触发,可以一次返回多个客户端连接的socket给应用层,这些socket就放在返回的列表中。这些列表的每项
    #fd:得到的是这次客户端链接对应的文件描述符
    #event:是指这个文件描述符到底是什么事件,是接收数据吗还是发送
    for fd,events in epoll_list:
        if fd == tcpserver.fileno():#确定现在触发的套接字是我设置的监听套接字,对比得到的文件描述符来确定是不是我监听套接字
            #如果是监听套接字被触发,证明有新的tcp连接进来
            newsocket,client_addr = tcpserver.accept()
            print(newsocket)
            print(client_addr)
            epoll.register(newsocket.fileno(),select.EPOLLIN);
            fd_dict[newsocket.fileno()] = newsocket
        elif events == select.EPOLIN:
            recvdata = fd_dict[fd].recv(1024).decode("utf-8")
            if len(recvdata) > 0:
                print("=====%s====" %recvdata)
            else:
                epoll.unregister(fd)
            fd_dict[fd].close()
            print("----close----")

```

for 循环后,我 if 判断共享内存列表 epoll\_list 的后面节点不是监听套接字,而是客户端事件,那么执行 elif  


判断这次列表节点的客户端事件是接收事件还是发送事件,如果是 select.EPOLIN,就是客户端发送数据过来了,用 recv 去接收。

这就是 epoll 基本流程,更多细节需要百度查阅资料。

## 客户端程序

```
from socket import * #这里要用from socket import的方式，服务端程序讲过
import time

tcpsocket = socket(AF_INET,SOCK_STREAM)#记住是SOCK_STREAM

tcpsocket.connect(("192.168.127.128",8888))#链接哪台服务器，执行connect，服务器的accept才不会阻塞
while True:
    tcpsocket.send("abcd1234".encode('utf-8'))#发送数据给服务器
    print("客户端tcp发送数据")
    data = tcpsocket.recv(1024)
    print("%s" %(data.decode('utf-8')))
    time.sleep(1)

tcpsocket.close()#关闭tcp链接
```

```
root@ubuntuv:/home/xzz/pytest# python3 xepoll.py
<socket.socket fd=5, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('192.168.127.128', 8888), raddr=('192.168.127.1', 53741)>
('192.168.127.1', 53741)
====abcd1234==
```

经过测试 epoll 服务器成功收到客户端发来的数据。

