day30-01（网络基础1）

网络编程

## 网络基础1

由于不同机器上的程序要通讯，所以产生了网络

两种架构：c/s架构和b/s架构

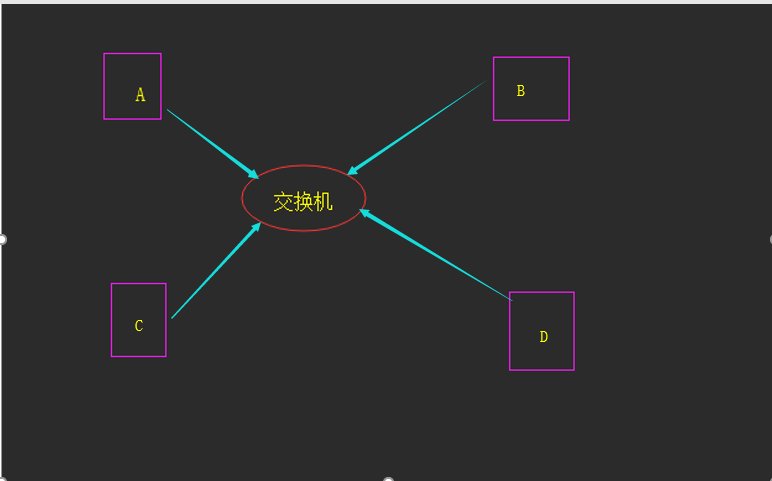
b/s火了是因为方便和统一入口（比如微信的各种小程序和公众号）

b/s和c/s架构的关系

网卡提供了网线的插口（网卡有全球唯一的mac地址）

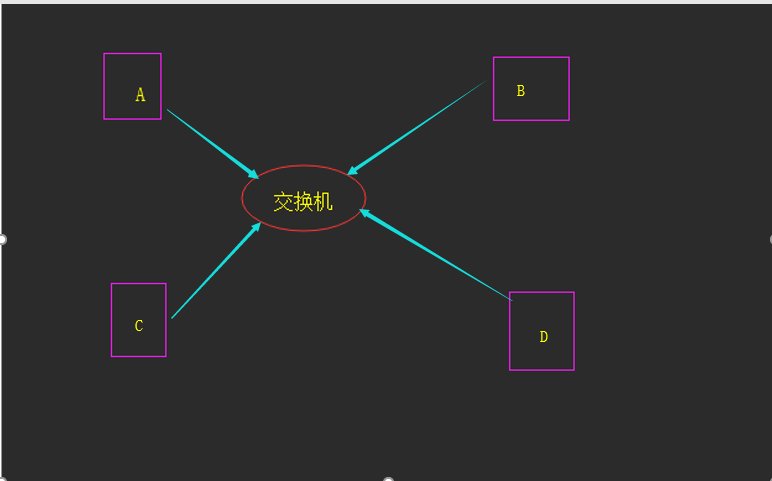
mac地址:每块网卡出厂时都会被烧制上一个世界上唯一的mac地址，长度为48位2进制，通常由12位16进制数表示（前六位是厂商编号，后六位是流水限号）

### 通过ip地址找到mac地址（arp协议）



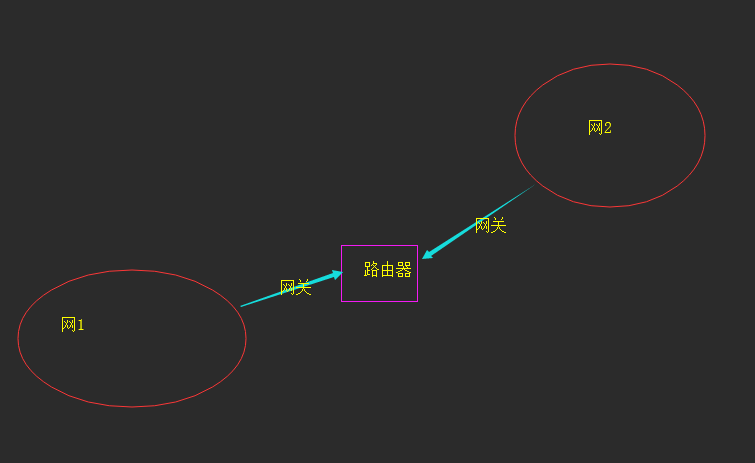
交换机的原理（A,B,C,D四台机器比如我要告诉交换机我要找A,交换机会（以广播的方式）把信息发送给A，B，C，D但是只有A会（以单播的形式）回应我）（这种方式的话，如果机器越多就容易产生广播风暴，所以产生了局域网）

### arp协议（局域网和交换机）



A电脑的数据包找ip为10.0.0.1的地址，A通过交换机以广播的形式告诉所有的电脑，只要是电脑的ip为10.0.0.1就将自己的mac地址发送过来，收到返回信息后将该ip地址和物理地址存入本机ARP缓存中并保留一定时间，下次请求时直接查询ARP缓存以节约资源。

### 广域网和路由器



网关：局域网中的机器想要访问局域网外面的机器需要通过网关

通过ip地址和子网掩码按位与判断是否在同一个网段里

day30-02(网络基础2-1)

总结：ip协议的作用主要有两个：一个是为每台计算机分配IP地址，另一个是确定哪些地址在同一个子网络里

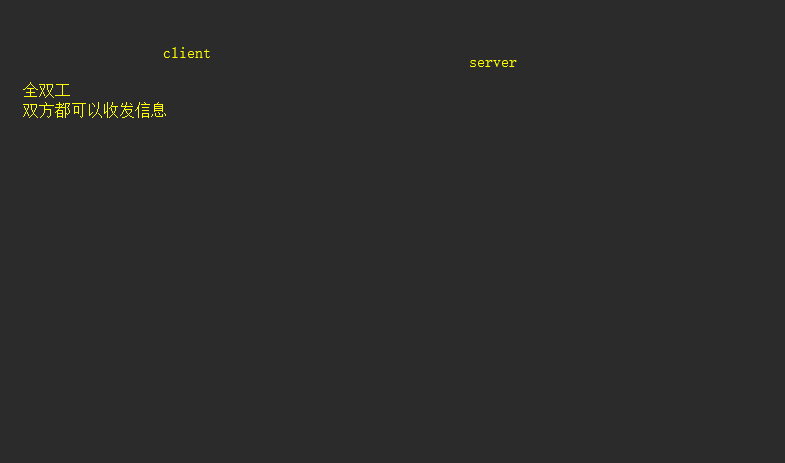
## 网络基础2

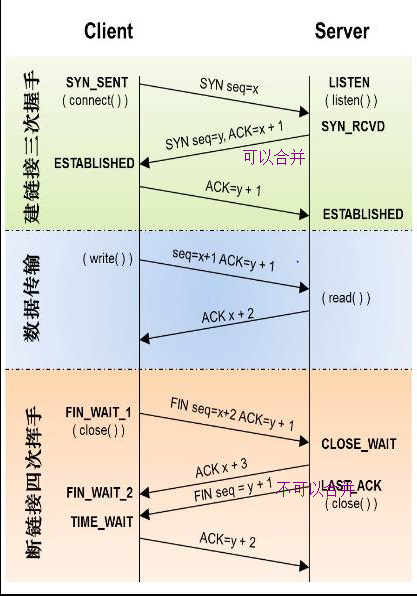
端口用来找到程序（在计算机上每一个需要网络通信的程序，都会开一个端口，在同一时间只会有一个程序占用一个端口，不可能在同一时间，有两个程序占用同一个端口）

端口的范围：0-65535，自己可以用的是8000之后的端口（之前的给系统用）

### TCP协议和UDP协议

#### TCP协议

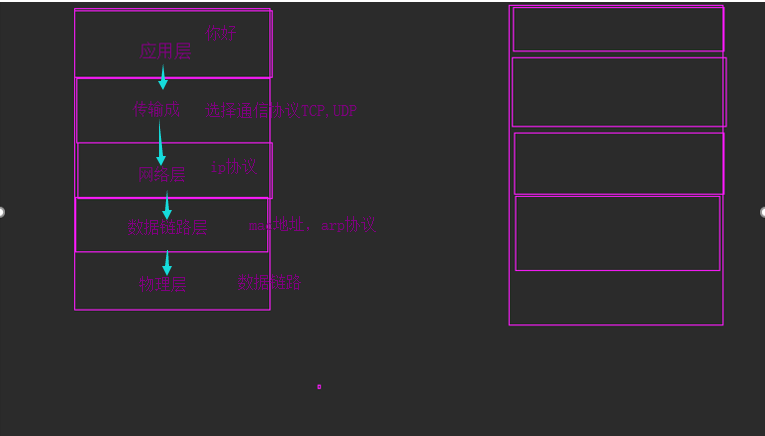




#### UDP协议（快）

qq采用的udp协议。他的确认机制是通过程序完成的不是协议完成的。

## 互联网协议



### 每层运行常见物理设备

传输层：四层交换机，四层路由器

网络层：路由器，三层交换机

数据链路层：网桥，以太网交换机，网卡

物理层：中继器，集线器，双绞线

### 每层运行常见协议

应用层：http（https花钱，它是安全的，买证书需要花钱）,ftp(文件传输),smtp（邮件协议）

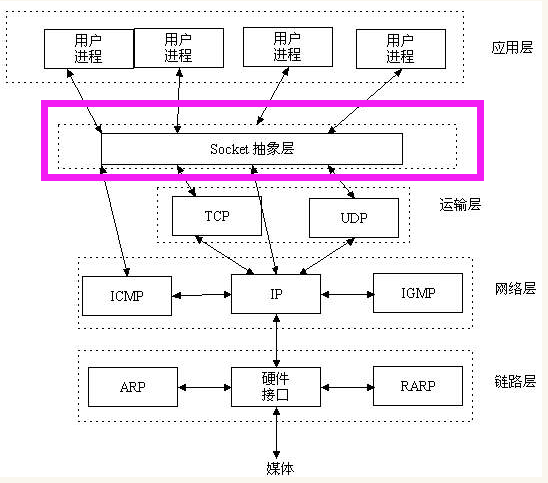
传输层：tcp和udp

网络层：ip

数据链路层：arp

day30-03(tcp协议socket初识)

## socket(套接字)



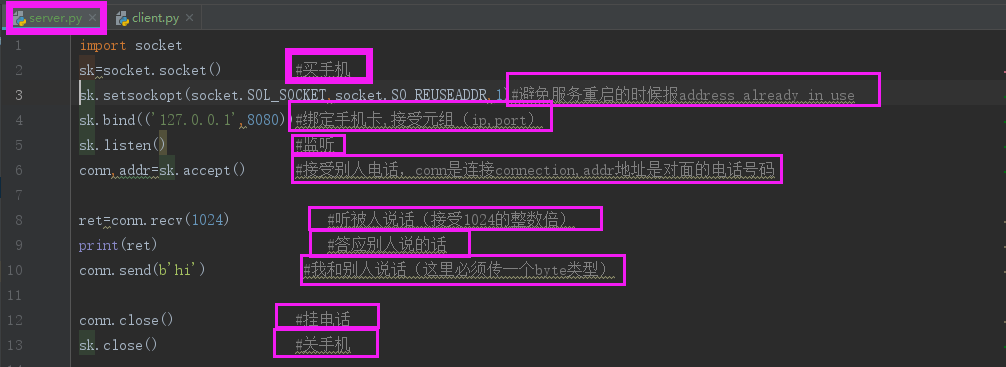
socket是应用层与TCP/IP协议族通信的中间软件抽象层，它是一组接口。在设计模式中，Socket其实就是一个门面模式，它把复杂的TCP/IP协议族隐藏在Socket接口后面，对用户来说，一组简单的接口就是全部，让Socket去组织数据，以符合指定的协议。

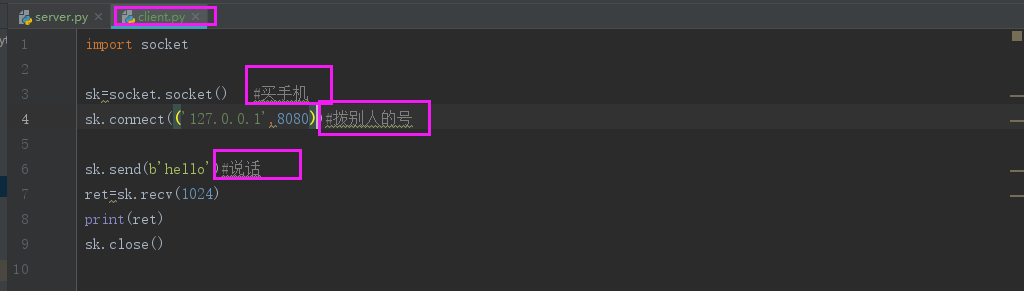
### 套接字分类

基于文件类型的套接字家族 AF\_UNIX

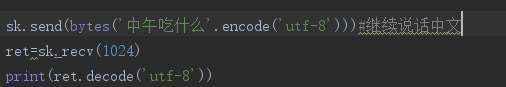
基于网络类型的套接字家族 AF\_INET

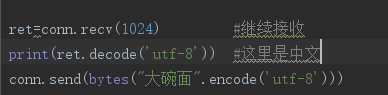
### 基于tcp协议的socket代码



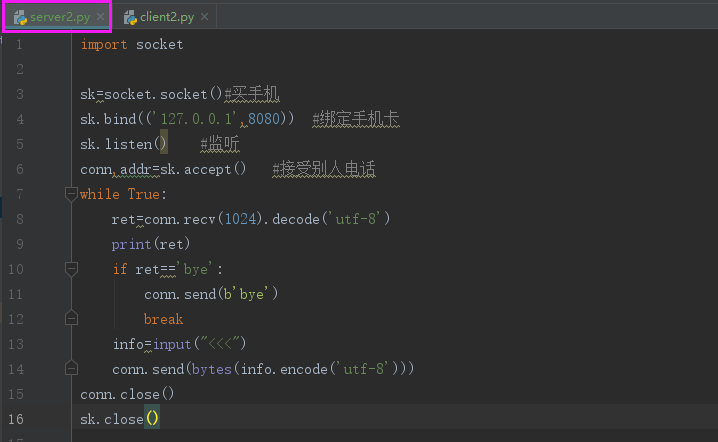


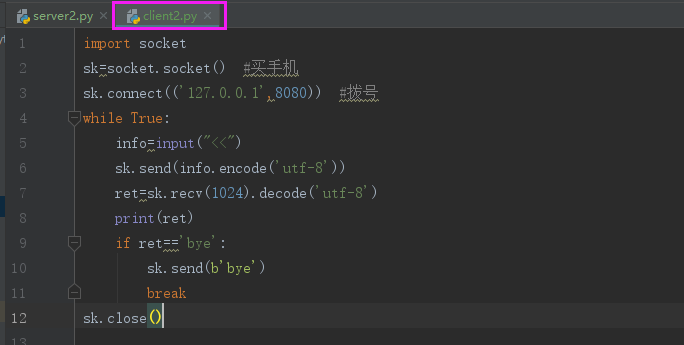
接受1024字节不足这么多按这么多接受超过的话不接除非下面还有接受





#### 无限聊天





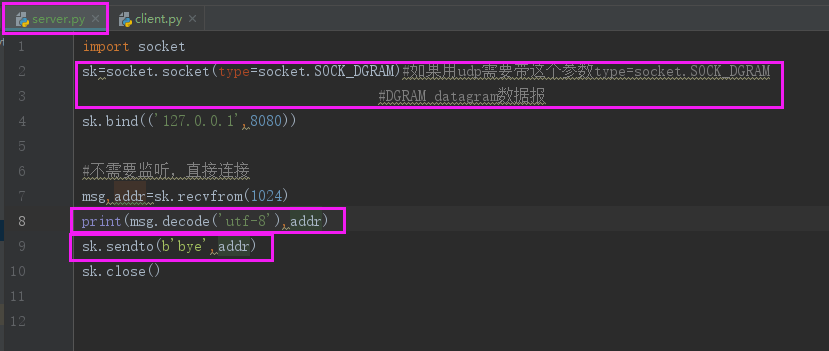
day31-01(复习和认识tcp的长连接)

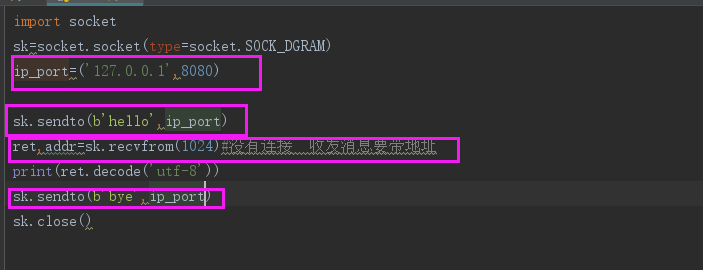
#### TCP的长连接

server端在接受client1的请求时建立了长连接，但是允许client2来连接，但是client必须等client连接中断，才和client2简历链接（长连接在任何时刻只能同一个人聊天）

day31-02（基于udp的socket服务）

### 基于UDP协议的socket代码





UDP的发送需要带地址

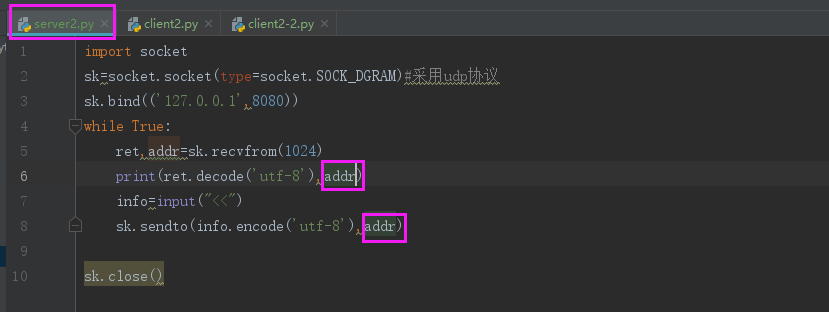
udp的server不需要监听也不需要建立连接

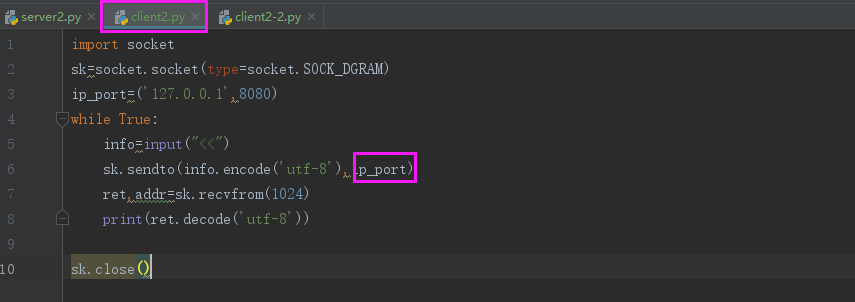
在启动服务之后只能被动的等待客户端发送消息过来

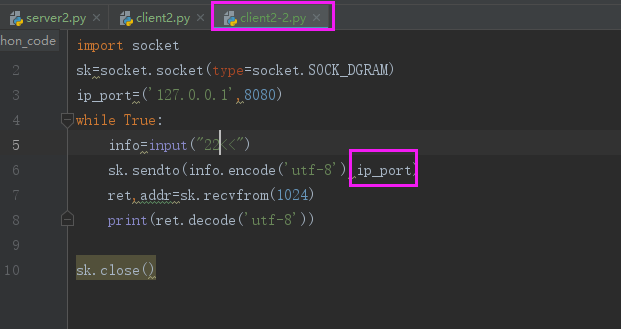
客户端发送消息的同时还会自带服务端的地址信息

消息恢复的时候，不仅需要发送消息，还需要把对方的地址填写上

#### qq聊天







一个udp的服务可以同时与多个人聊天

#### 作业时间服务

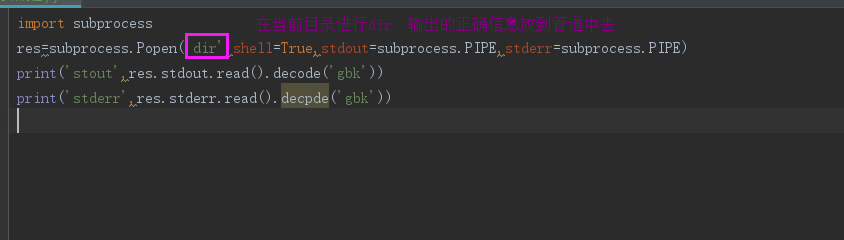
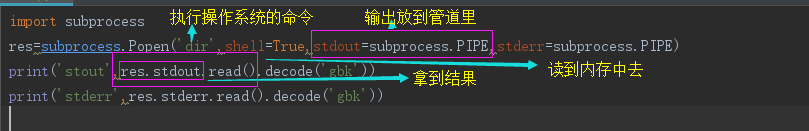
server:接收信息 时间格式 将我的时间转换成接受到的格式 发回给客户端

client：发送时间格式

day31-03(黏包现象)

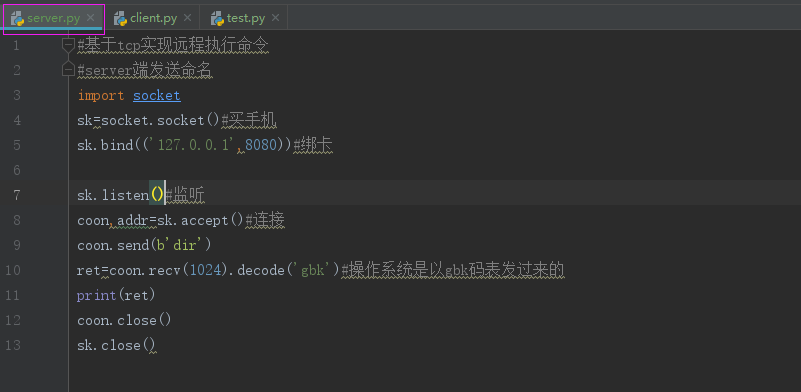
## 黏包现象（接受的消息多了或者少了）

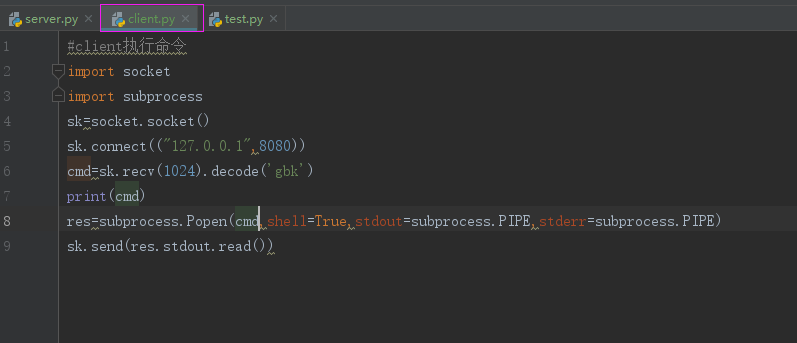
### subprocess模块



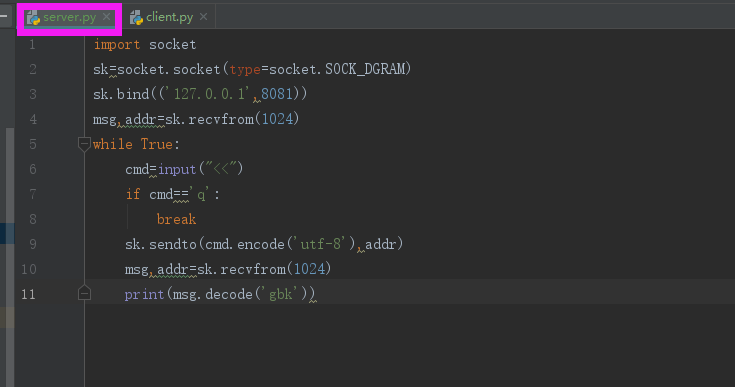
### 基于tcp实现远程执行命令（会出现黏包现象）

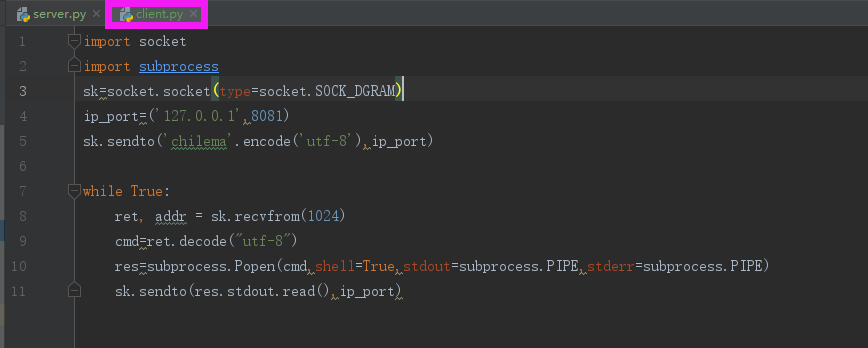
在server端下发命令在client端执行命令





### 基于udp实现远程执行命令（只能在拿到客户端的地址后才能发送请求）





### 两者优缺点

tcp出现黏包，但不会丢包

udp不会出项黏包，但是会丢包（qq发的消息太长，让你以文件的形式发过去）

### 黏包原因（tcp连续发的两个包时间间隔短）

#### TCP协议的数据传递

tcp协议的拆包机制

当发送端缓冲区的长度大于网卡的MTU(Maximum Transmission Unit的缩写，意思是网络上传输的最大数据包)。MTU的单位是字节，大部分网络设备的MTU都是1500，如果本机的MTU比网关的MTU大，大的数据包就会被拆开来传送，这样会产生很多的数据包碎片，增加丢包率，降低网络速度。

#### 面向流的通信特点和Nagle算法

TCP是面向连接的，面向流的，提供高可可靠性服务。收发两段都要有一成对的socket，因此，发送端为了将多个发往接收端的包，更有效的发送到对方，使用了优化算法（Nagle算法），将多次间隔较小且数据量小的数据，合并成一个大的数据块，然后进行封包。

这样接收端，就难于分辨出来了，必须提供科学的拆包机制。即面向流的机制是无消息保护边界的。

对于空消息，tcp是基于数据流的，于是收发的消息不能为空，这就需要客户端和服务端都田间空消息处理机制，防止程序卡住，而udp是基于数据包报的，即便是你输入的是空内容，也可以被发送。

### 补充说明

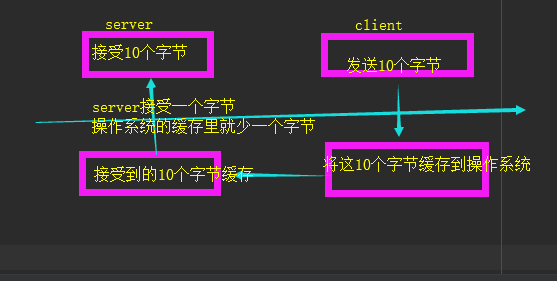
用UDP协议发送时，用sendto函数最大能发送数据的长度为65535-IP头(20)-UDP头（20）=65507字节。用sendto函数发送数据时，如果发送数据长度大于该值，则函数会返回错误（丢弃这个包，不进行发送）

用TCP协议发送时，由于tcp是数据流协议，因此不存在包大小的限制（暂时不考虑缓冲区的大小），这是指在用send函数时，数据长度参数不受限制。而实际上，所制定的这段数据并不会一次性发送出去，如果这段数据比较长，会被分段发送，如果比较短，可能会等待和下一次数据一起发送

### 会发生黏包的两种原因

情况一发送方有缓存机制

情况二接收方有缓存机制



最根本的原因是不知道发送方发送多少数据

day32-01(复习)

## 复习

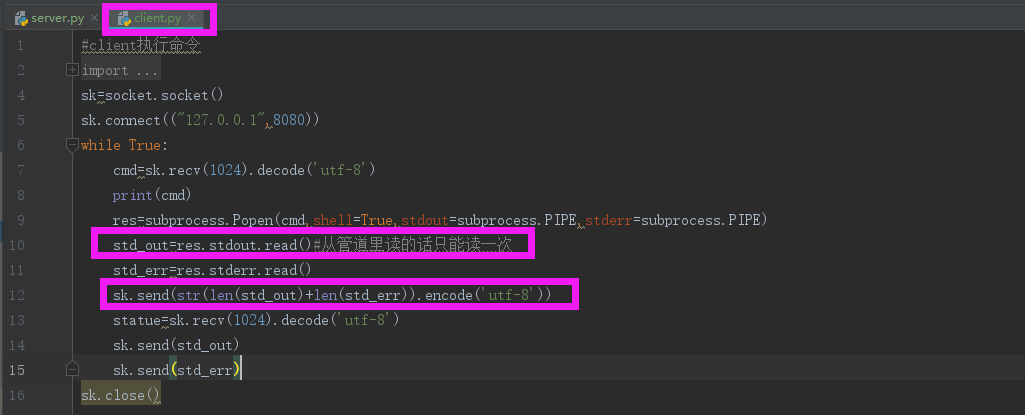
多个send小的数据连载一起，会发生黏包现象，是tcp协议内部的优化算法造成的。

day32-02(解决黏包问题的方式)

## 解决黏包问题的方式一

首先发送以下这个数据到底有多大





### 好处：

确定了我到底要接受多大的数据

要在文件中配置一个配置项，就是每一次recv的大小 buffer=4096

打我们要发送大数据的时候，要明确的告诉接收方要发送多大的数据，以便接收方能够准确的接受所有数据

多用在文件的传输过程中

大文件的传输一定是按照字节读，每一次读固定的字节

传输的过程中一边读一边传一边收一边写

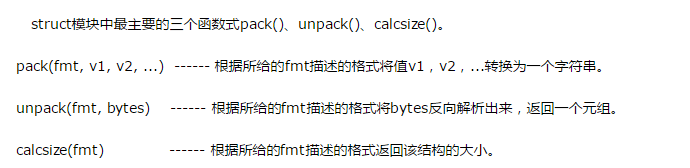
### 坏处：

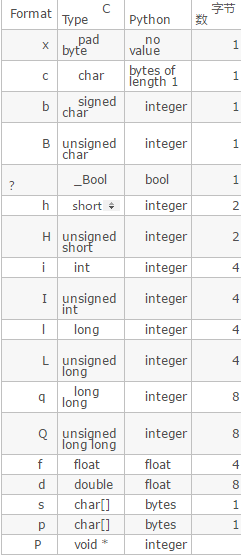
多了一次交互

send sendto在超过一定范围的时候，都会报错

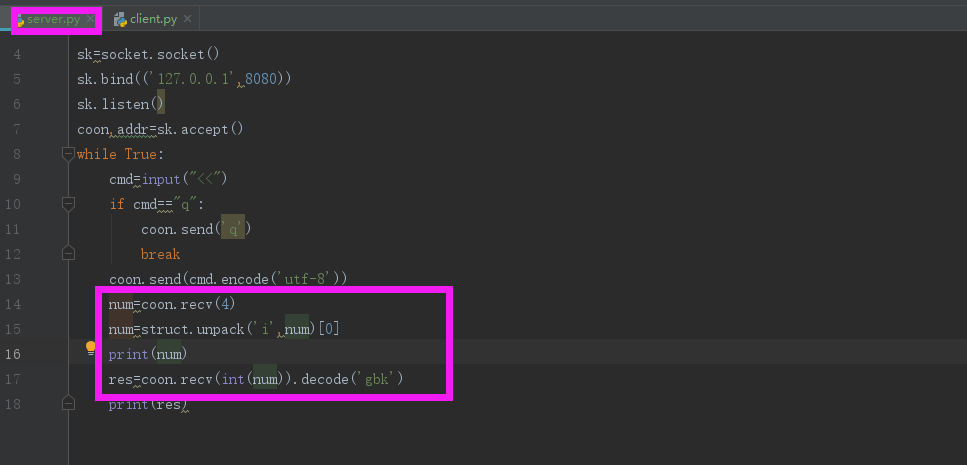
day23—03(struct模块解决黏包问题方式)

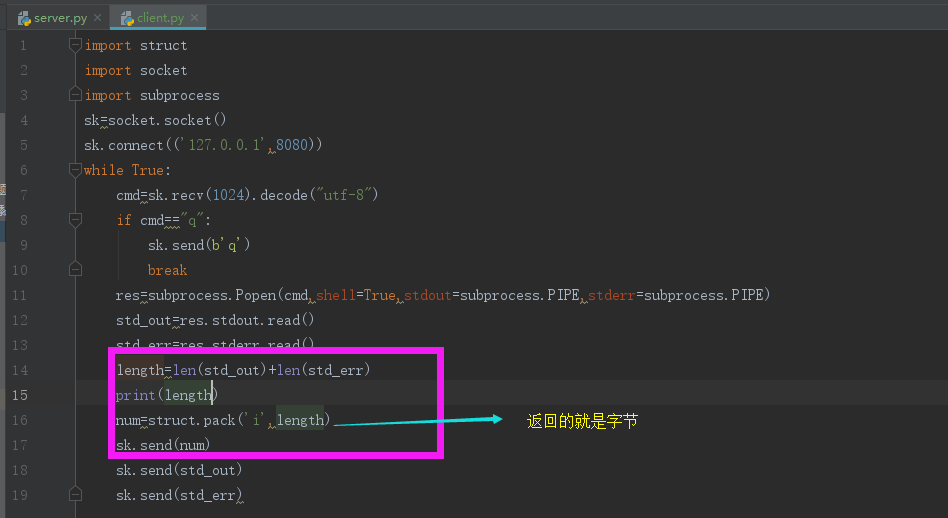
## struct模块解决黏包问题方式2（将一个数转成字节）





### 代码





day32-01(struct模块补充)

## struct模块补充

ret=struct.pack('i',4096000)#b'\x00\x80>\x00'

这里>也是一个字节可以用长度检验

day32-05(struct模块定制报头的理论)

## struct模块定制报头的理论

所有的报文都有报头

我自己定制报文：

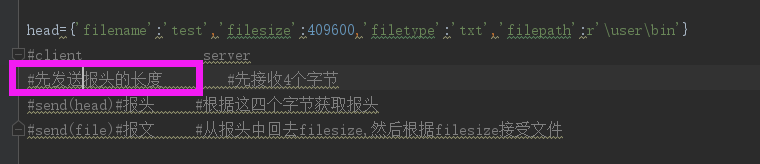
传输文件的时候

文件的名字

文件的大小

文件的类型

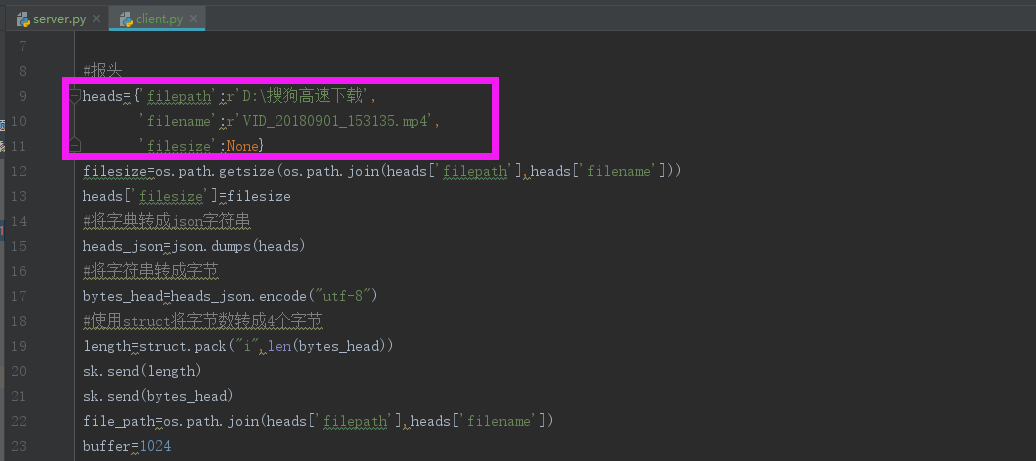
存储的路径

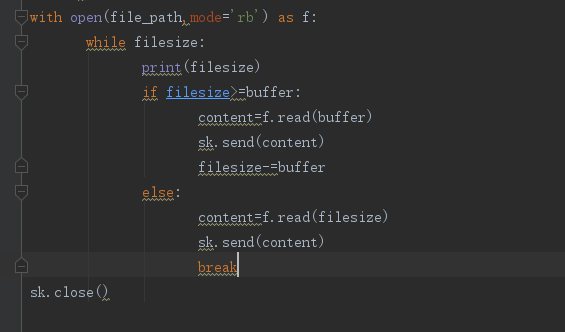


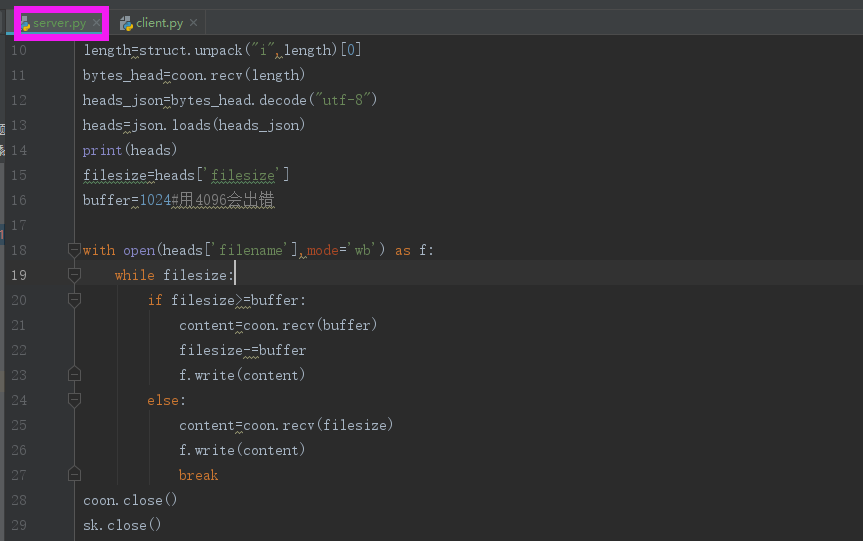
网络中处处有协议，协议就是一堆报头和报文

day32-06(struct模块定制报头ftp实践)

## struct模块定制报头ftp实践







day32-07(关于ftp文件传输的报错)

## 关于ftp文件传输的报错

读的速度比写的速度快，当buffer=4096时长时间在操作系统积累，操作系统积累的越多，就会断开连接。

day33-01(复习)

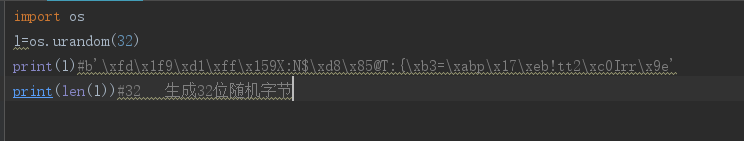
## 复习

解决黏包问题

在传输大量数据之前先告诉接收端要发送的数据大小

如果想更漂亮的解决问题，可以通过struct模块

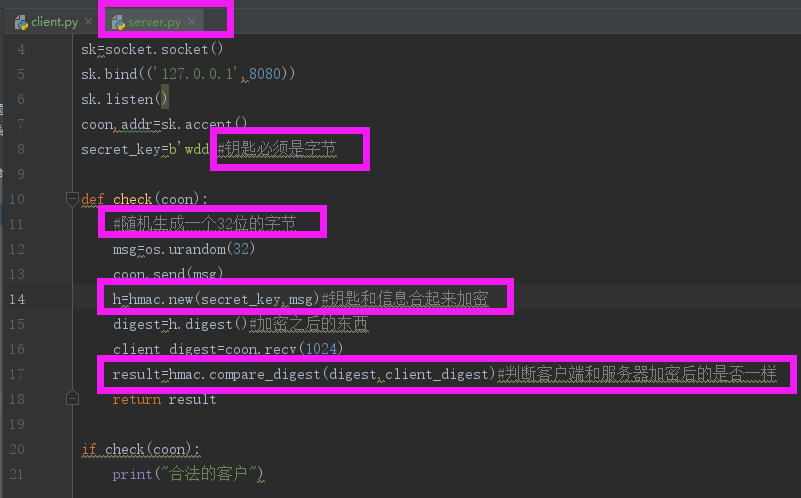
## 验证客户端的合法性

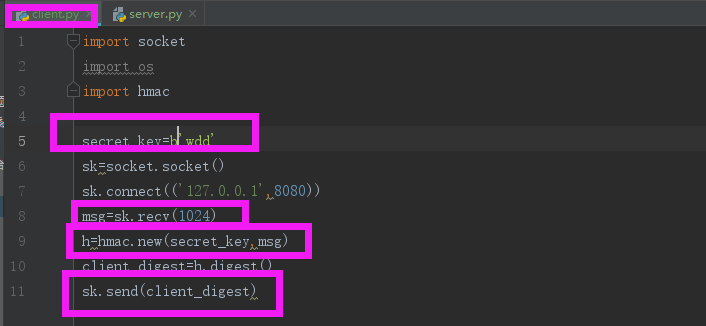


client-🡪server 只需要ip和 port就能连接了

监测客户端是否合法，不依赖登录验证

原理：client有个钥匙，server有个要是，随机发送字节，client和server同时用钥匙加密





day33-02(hmac的检验客户端合法性和socketserver)

## socketserver

为啥要有socketserver:socket起一个tcp服务，在同一个时间，只能与一个通信，socketserver可以与多个

### socketserver源码分析（重点）

看源码的要点

第一 多个类中之间的继承关系要先整理

第二 每一个类中有哪些方法，要大致列处理啊

所有的self对象调用要清楚的了解，到底是谁的对象

所有的方法调用要退回到最子类的类中开始寻找，逐级向上

## ftp作业分析

### 需求分析

1. 多用户同时登录 socketserver
2. 用户登陆，加密认证 hashlib
3. 上传/下载文件，保证文件一致性 md5验证
4. 不同用户家目录不同，且只能访问自己的家目录
5. 对用户进行磁盘配额，不同用户配额不同
6. 用户登录server后，可在家目录权限下切换子目录
7. 查看当前目录下文件，新建文件夹
8. 删除文件和空文件夹
9. 承诺股份使用面向对象知识
10. 传输过程镇南关实现进度条
11. 支持断点续传

### 问题：

加密 client 还是server？ server端加密，客户端可以看到源码

ftp 是不是谁都能用 ？

所有的目录切换放到server端

day34-01

## 复习

交换机：广播，单播，组播（一个局域网内的计算机分组）

物理层：网卡，双绞线

数据链路层：arp 交换机（最高的网络层协议在数据链路层）

网络层：ip

传输层:tcp，ip

解释我在家为啥能上网：

我家的路由器是一个三层交换机（既有路由功能，有又交换机的功能）

## send()和sendall()的区别

#sk.sendall("hello world")  
  
buffer="hello world\n"  
while buffer:  
 bytes=sk.send(buffer)  
 buffer=[bytes:]

上面两段代码等价

sendall是尝试将所有的string发送出去，发送不了就失败，

send是将string可能分段发送出去

day34-01(ftp1)

## ftp1

### 需求分析

1. 多用户同时登录 socketserver
2. 用户登陆，加密认证 hashlib
3. 上传/下载文件，保证文件一致性 md5验证
4. 不同用户家目录不同，且只能访问自己的家目录
5. 对用户进行磁盘配额，不同用户配额不同
6. 用户登录server后，可在家目录权限下切换子目录
7. 查看当前目录下文件，新建文件夹
8. 删除文件和空文件夹
9. 承诺股份使用面向对象知识
10. 传输过程镇南关实现进度条
11. 支持断点续传

看项目代码

day35-01(操作系统的发展史)

## 操作系统的发展史（理论部分）

手工—》批处理（所有的计算任务集中在一个纸袋上）—》脱机处理（专门的读卡机）--》脱机批处理（cpu和输入输出可以并行工作）--》（多道程序系统）--》分时系统

day32-02(进程概念)

## 进程概念

进程---只有在运行当中的程序才称为进程

为什么要引入进程（为了隔离资源，每个程序资源不能共享）

### 进程调度策略

先来先服务

短作业优先算法

时间片轮转算法

多级反馈算法

进程的并行（同时执行，资源够用，四核的cpu）与并发（交替执行）

### 同步

调度之后还一定要等到结果（我叫小王吃饭，小王在敲代码，我等到小王敲完代码在一起去吃法）

同步交互：指发送一个请求,需要等待返回,然后才能够发送下一个请求，有个等待过程；

### 异步

我只管调度，不管结果（我叫小王吃饭，小王在敲代码，我没登他直接去吃饭）

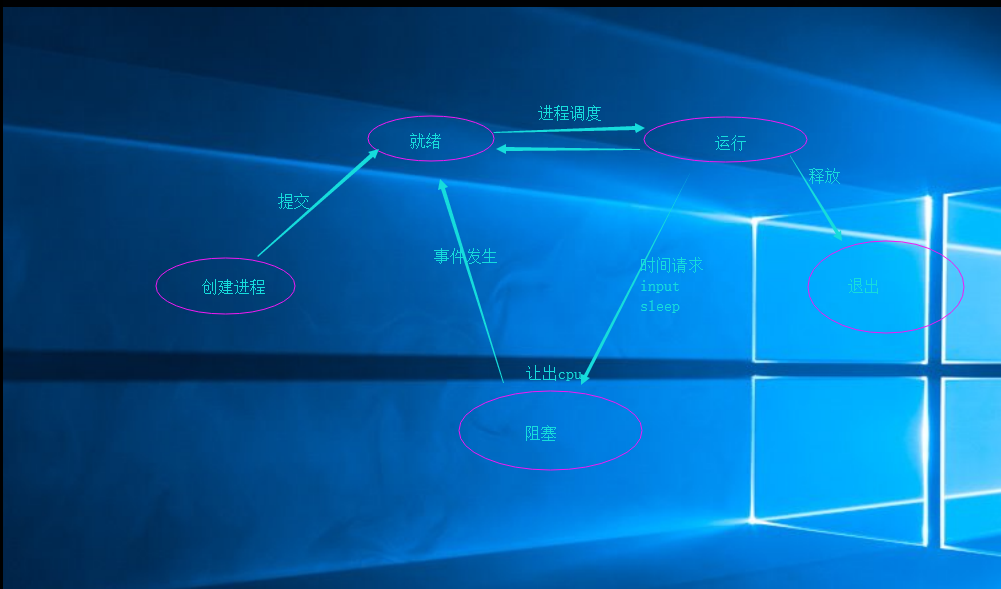
异步交互：指发送一个请求,不需要等待返回,随时可以再发送下一个请求，即不需要等待。

### 阻塞

receive input

### 非阻塞

### 进程交互图



### 同步阻塞形式

效率最低。比如说你排队取号，因为要等结果，但是你阻塞了，所以你现在除了排队外什么也干不了

### 异步阻塞形式

如果在银行等待办理业务的人采用的是异步的方式去等待消息被接触（通知），也就是领了一张小纸条，加入在这段时间里他不能离开银行做其它的事情（他必须在这等通知），那么显然，这个人被阻塞在了这个等待的操作上面。

异步操作是可以被阻塞住的，只不过它不是处理消息时阻塞，而是在等待消息通知被阻塞

### 同步非阻塞形式

实际上是效率低下的

想象一下你一边打电话一边还需要抬头看到队伍排到你了没，如果把打电话和观察排队的位置看成是程序的两个操作的话，这个程序需要在这两种不同的行为之间来回切换，效率可想而知是底下的。

### 异步非阻塞形式

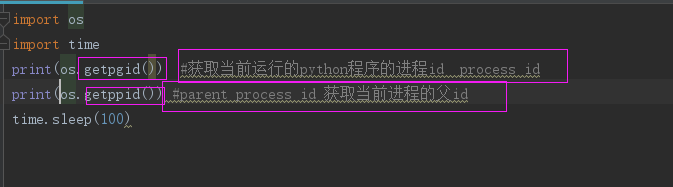
效率更高

因为打电话是（等待着）的事情，而通知你则是柜台（消息触发机制）的事情，程序没有在两种不同的操作中来回切换。

比如说，这个人突然发觉自己烟瘾犯了，需要出去抽根烟，于是他告诉大堂经理说，排到我这个号码的时候麻烦到外面通知我一下，那么他就没有被阻塞在这个等待的操作上面，自然这个就是异步+非阻塞的方式了

### 总结：

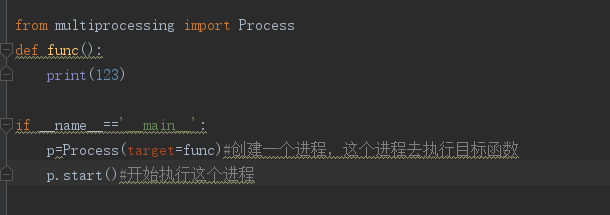
很多人把同步和阻塞混淆，是因为很多时候同步操作会以阻塞的形式表现出来，同样的，很多人也会把异步和非阻塞混淆，因为异步操作一般都不会在真正的IO操作处被阻塞



python中创建进程来替我做事情 multiprocessing模块（这是一个综合的管理进程的包）

day35-03(子进程的开启)

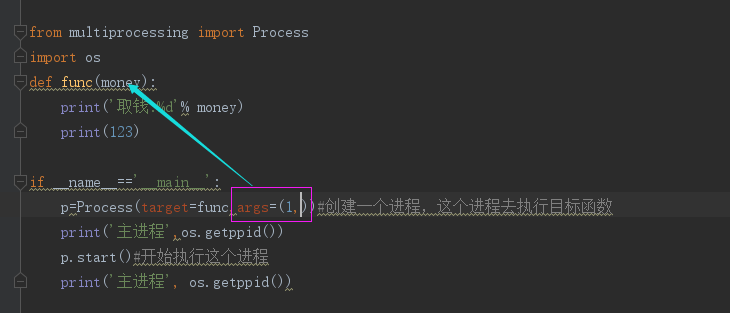
### python中子进程的开启



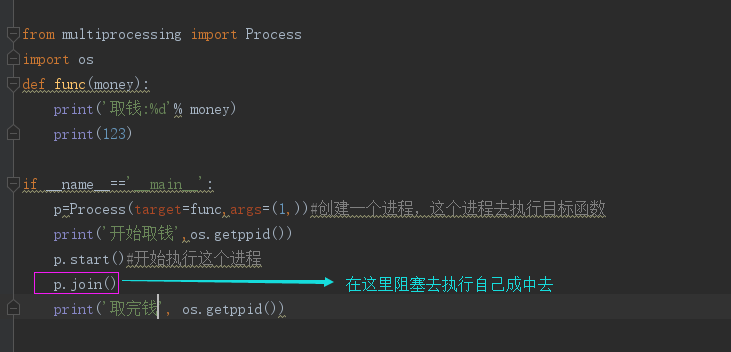
#### 细节（打印顺序）



在同一个进程中，程序确实是从上到下依次进行的。



### 将子进程加入到主进程

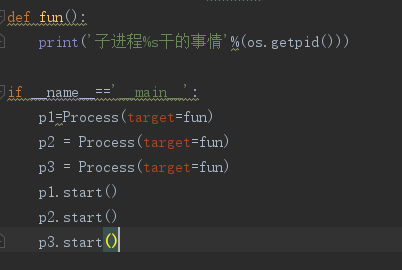


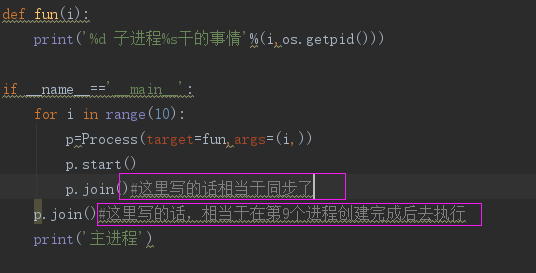
#### windows下的坑

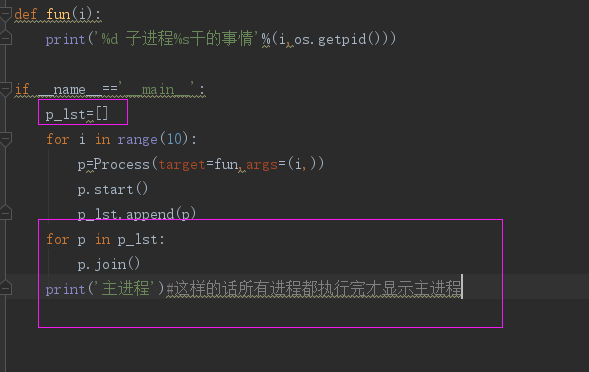
在windows系统中创建进程这句话放到if \_\_name\_\_=’\_\_main\_\_’中去

因为创建进程这个代码会把整个文件在执行一次，如果把创建进程代码放在if外面，会形成递归

#### 开启多个子线程

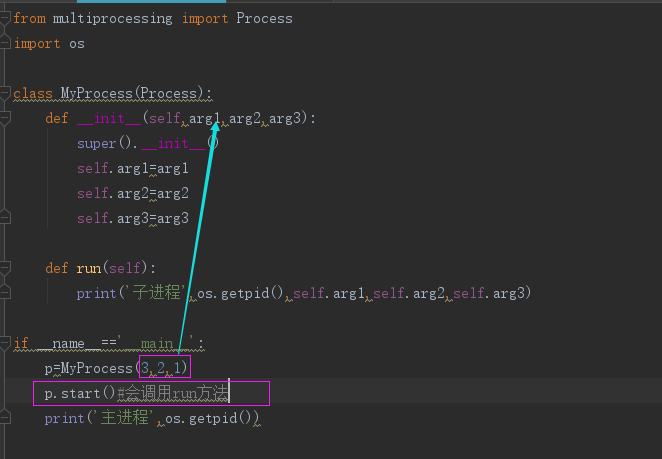






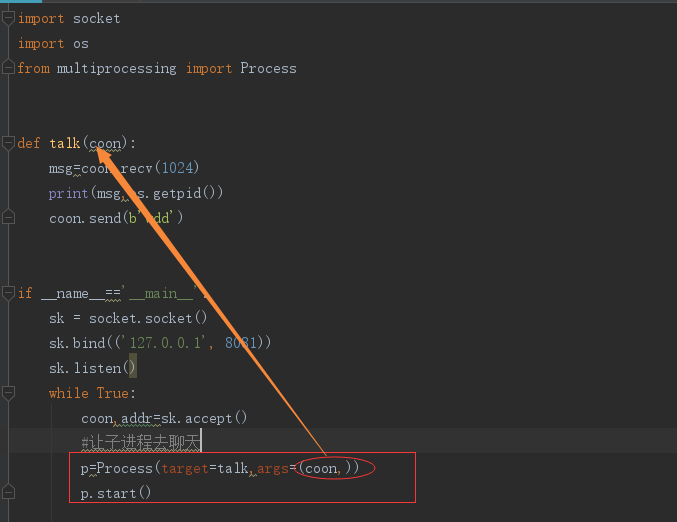
day35-04(子进程之间的数据隔离问题)

### 创建进程类



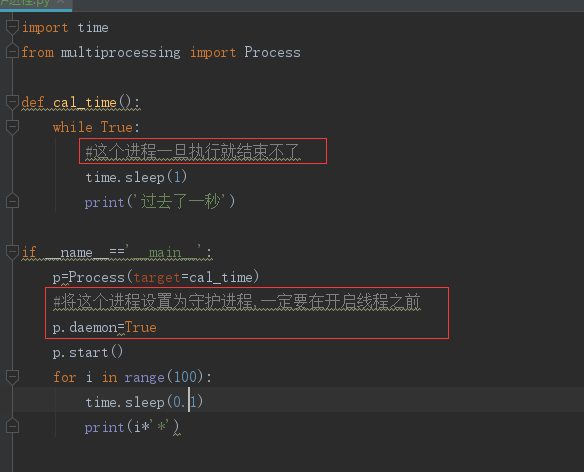
day36-01(进程操作实例和进程的其他方法)

## 开多个进程用异步聊天



## 进程操作实例和进程的其他方法

守护进程会随着主进程的代码执行结束而结束，但是正常的子进程没有执行完的时候主进程要一直等着。



守护进程中不能再开启子进程。

#判断进程是否活着，True代表活着，false代表不在了  
p.is\_alive()  
#结束一个进程，但是这个进程不会立即执行，这是个异步操作  
p.terminate()

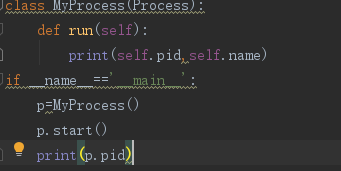
#查看进程名字

p.pid

#查看进程id

p.id

### 进程内部使用id和那么属性



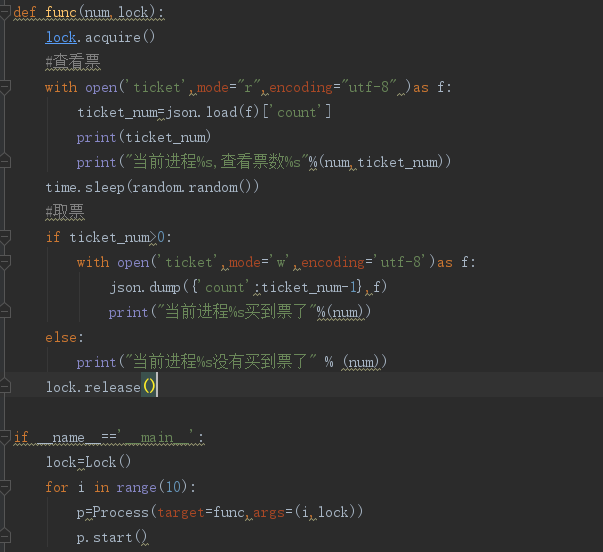
day36-02(锁)

## 锁

from multiprocessing import Lock  
lock=Lock()  
#向锁要钥匙  
lock.acquire()  
#释放钥匙  
lock.release()

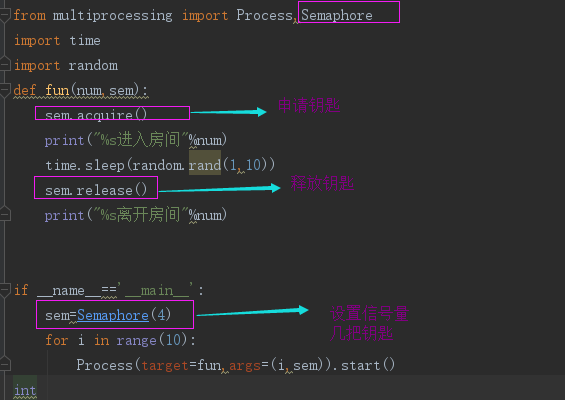
多进程实现并发

### 进程和锁结合（抢票）



day36-03(信号量)

## 信号量（我有一把锁，但是我 有好几把钥匙）



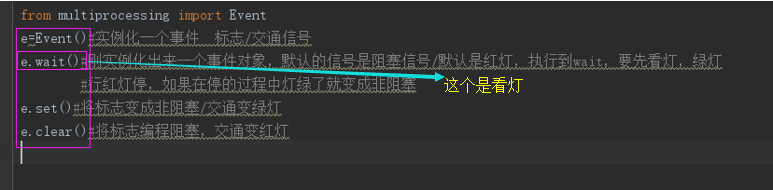
day36-04(事件和队列)

## 事件（同时是所有的进程陷入阻塞和恢复阻塞）和队列

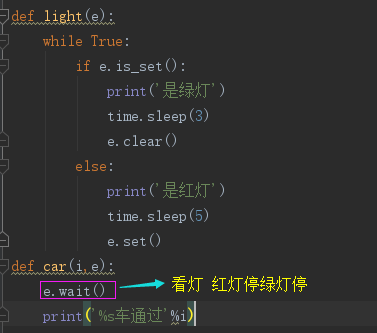
所有的锁都是同步阻塞（input,sleep，写在那个进程就阻塞那个进程）

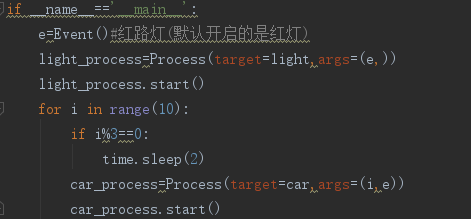
但是锁和信号量是异步阻塞

事件-------（异步阻塞，同时是所有的进程陷入阻塞和恢复阻塞）



### 红绿灯

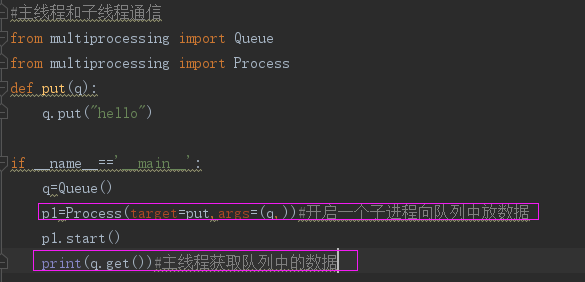




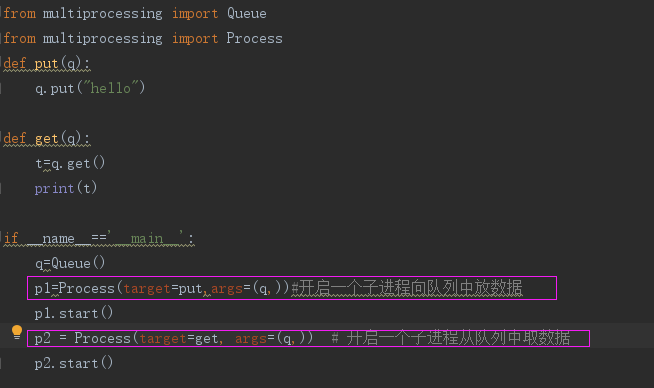
## 进程之间通信（队列）

#from multiprocessing import Queue  
#进程之间的同信通过multiprocessing的Queue模块  
#队列有两种创建方式，第一种不传参数，这个队列没有长度限制，传参数，创建一个有最大长度的队列  
#提供两个重要方法:put get  
#对于不定长的队列，put不会阻塞，对于定长的队列会阻塞  
#qsize不一定准确

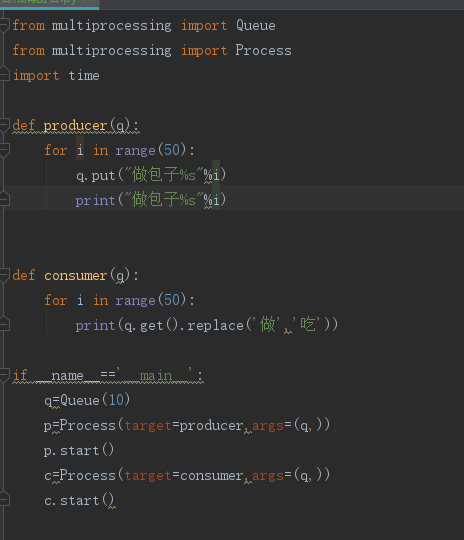
### 主线程和子线程通信



### 子线程和子线程通信



### 生产者和消费者



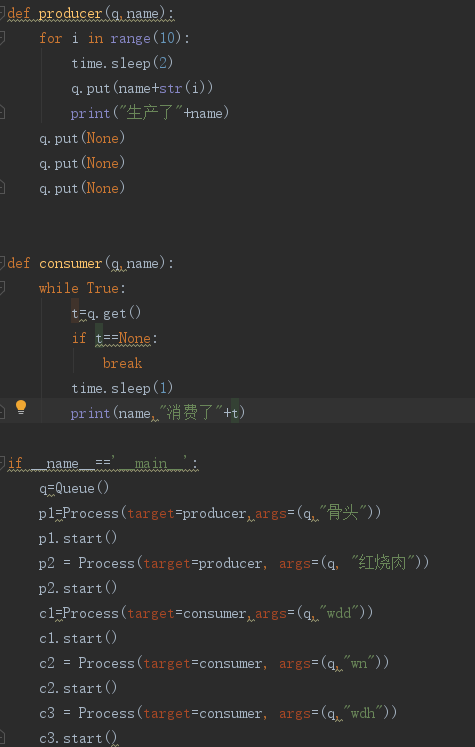
day37-01(生产者和消费者进阶)

### 生产者和消费者进阶

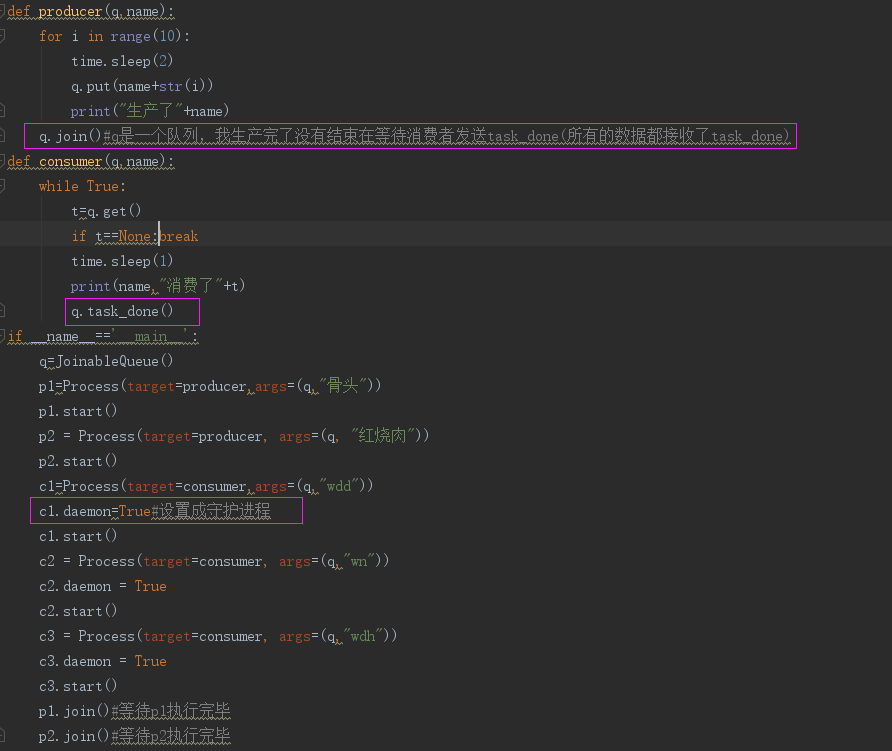
queue队列是一个数据安全，它内置加了一把锁

#### 生产者和消费者模型一

1. 消费者要处理多少数据是不确定的
2. 所以只能用While循环来处理数据，但是while循环无法结束
3. 需要生产者发送信号
4. 有多少消费者就需要发送多少个信号
5. 但是发送信号的数量需要根据生产者和消费者的数量进行计算，所以非常不方便

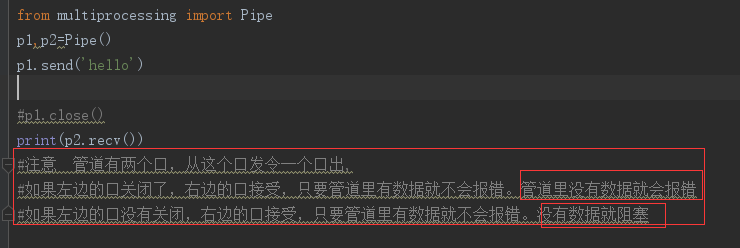


#### 生产者和消费者模型二（确保生产者和消费者同时结束，不在盲目等待）

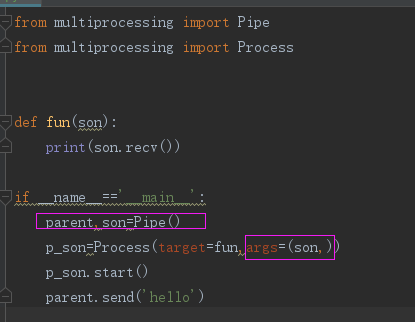


生产者在生产完所有的商品后会对队列执行q.join()这个行为，这个行为一直接收消费者发送的q.task\_done(),当所有的产品都接收到task\_done的时候，生产者执行完毕，然后用生产者调用join函数，这个函数会等待生产者生产完毕，由于消费者是守护进程，当生产者完毕的时候，消费者也结束了。

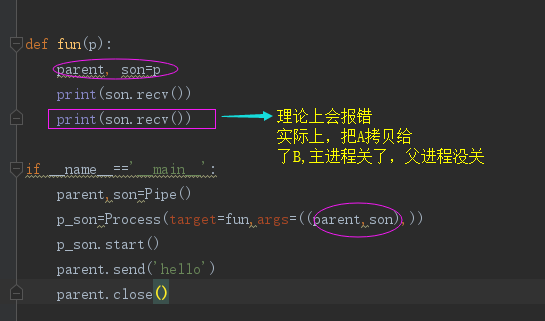
## Pipe管道



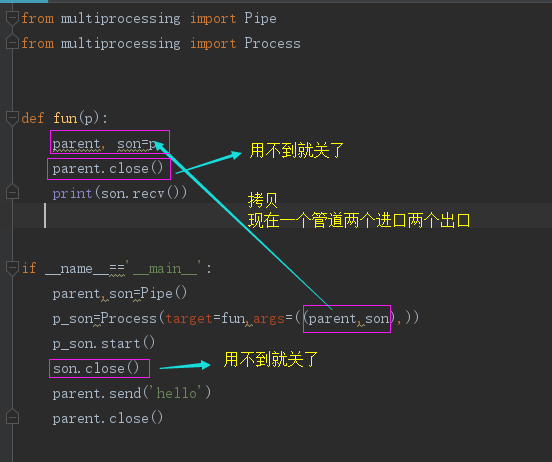
### 借用管道实现父子进程通信



### 注意事项



所以主线程和子线程在使用同一个pipe时，会在子线程拷贝一对进口和出口，所以如果我们的主线程用不到出口就把出口关了，子线程用不到入口就把入口关了，或者是主线程用不到进口就把进口关了，子线程用不到出口就不出口关了。



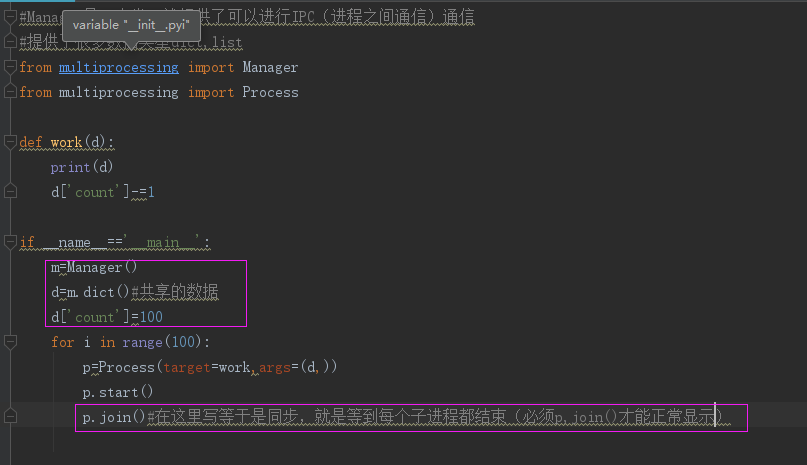
pipe是默认数据不安全的（队列等于管道加锁）

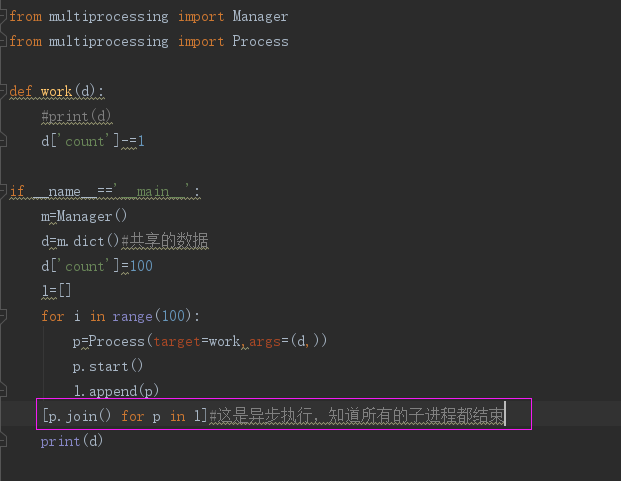


day37-03(进程之间的数据共享Manager)

## 进程之间的数据共享Manager







在manager中提供了很多数据支持但不确保数据安全的支持

day37-04(进程池)

## 进程池