

网络编程

概念：

1. C/S架构：
 1. C:client 客户端
 2. S: server 服务端，为所有的用户提供服务
2. B/S架构：只要在浏览器输入网址就可以直接使用了
 1. B:browser 浏览器
 2. S: server 服务端
3. B/S更好：更节省资源，不用更新，不依赖环境，
 - 统一了所有的web程序入口
4. C/S架构：安全性，程序比较庞大

mac地址：

是一个物理地址，在网卡中存在的，唯一的标识你的网络设备

1. mac IP地址：定位到一台机器
2. port 端口：0-65535，标识一台机器上的一个服务
3. IP+ port 能够唯一标识一台设备

ip地址：

1. 是一个逻辑地址
2. 是可以根据你的位置变化发生改变的
3. 能够在广域网中快速的定位你
4. 几个特殊IP
 1. 127.0.0.1 本地回环地址
 2. 0.0.0.0 表示你的所有网卡地址，标识本机回环地址+内网地址+公网地址

公网和内网

1. 公网ip：你能够在任意一个地方去访问的ip地址（不包含保留字段）
 1. ip v4：4位点分十进制组成0.0.0.0~255.255.255.255
 2. IP v6：6位冒分十六进制0：0:0:0:0~FFFF:FFFF:FFFF:FFFF:FFFF:FFFF
2. 内网：只能在一个区域内使用，除了这个区域就无法使用
 1. 所有的内网ip都要使用保留字段
 2. 192.168.0.0--192.168.255.255
 3. 10.0.0.0-10.255.255.255
 4. 172.16.0.0-172.31.255.255

```
# arp 协议 : 地址解析协议 通过一台机器的ip地址找到mac地址
# 交换机 广播 单播
# ip 协议 : ipv4 ipv6
# 内网的保留字段
# 192.168.0.0 - 192.168.255.255 2*16 65536
# 172.16.0.0 - 172.31.255.255 2**20 1048576
# 10.0.0.0 - 10.255.255.255 2**24 16777216
```

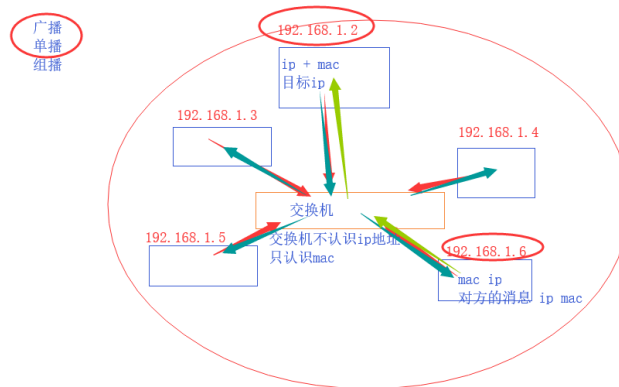
路由器和交换机:

1. 交换机: 完成局域网内通信

- 通过ip找mac地址: arp协议, 只用到广播和单播

```
# 路由器和交换机
```

```
# 交换机完成局域网内通信
# 路由器完成局域网间通信
```



- 广播
- 单播
- 组播

2. 路由器: 完成局域网间通信 (网关)

- 网关: 所有与局域网外部通信的时候所过的关口, 所有的请求都会在这里换上网关IP对外通信

3. 子网掩码 (了解) : 255.255.255.0

- ip 和子网掩码 按位于运算 (0或者1)
- 可以判断要寻找的机器是不是在一个局域网中

4. 端口号:

```
3306  mysql 数据库
6379  redis 端口
5000  flask 端口
```

TCP实时通信

```
```Python
##server
```

```
import socket

sk = socket.socket()
sk.bind(("127.0.0.1", 9043))
sk.listen()
conn, addr = sk.accept()
conn.send(b'hi')
msg = conn.recv(1024).decode("utf-8")
print(msg)
conn.close()
sk.close()
```

```
```Python
##client

import socket

sk = socket.socket()

sk.connect(("127.0.0.1", 9043))

msg = sk.recv(1024).decode("utf-8")
print(msg)
sk.send("傻子".encode("utf-8"))
sk.close()
```
```

## 网络概念补充：

### 1. osi七层协议

1. 应用层
2. 表示层
3. 会话层
4. 传输层
5. 网络层
6. 数据链路层
7. 物理层

### 2. osi五层协议：

1. 应用层
2. 传输层 #端口 TCP/UDP
3. 网络层 # IP 路由器
4. 数据链路层 # mac, arp协议, 网卡和交换机
5. 物理层

### # 五层协议

|             |              |             |
|-------------|--------------|-------------|
| # 应用层(五层)   |              |             |
| # 传输层(四层)   | 端口 UDP TCP   | 四层交换机 四层路由器 |
| # 网络层(三层)   | ipv4 ipv6 协议 | 路由器 三层交换机   |
| # 数据链路层(二层) | mac arp 协议   | 网卡 (二层) 交换机 |
| # 物理层(一层)   |              |             |

### 3. TCP/IP--网络层arp

## TCP:

上传，下载，邮件，可靠，面向连接，速度慢，能传递的数据长度不限

### 1. 两边需要先连接

#### ◦ 三次握手

第一次握手：Client将标志位SYN置为1，随机产生一个值seq=j，并将该数据包发送给Server，Client进入SYN\_SENT状态，等待Server确认。

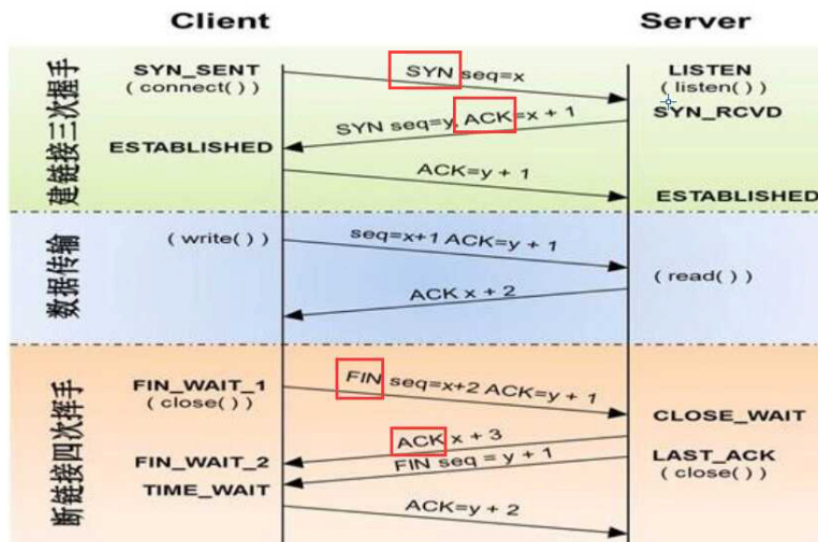
第二次握手：Server收到数据包后由标志位SYN=1知道Client请求建立连接，Server将标志位SYN和ACK都置为1，ack=j+1，随机产生一个值seq=K，并将该数据包发送给Client以确认连接请求，Server进入SYN\_RCVD状态。

第三次握手：Client收到确认后，检查ack是否为j+1，ACK是否为1，如果正确则将标志位ACK置为1，ack=K+1，并将该数据包发送给Server，Server检查ack是否为K+1，ACK是否为1，如果正确则连接建立成功，Client和Server进入ESTABLISHED状态，完成三次握手，随后Client与Server之间可以开始传输数据了。

### 2. 消息传递: (主要看发送，连接时的状态)

当应用程序希望通过 TCP 与另一个应用程序通信时，它会发送一个通信请求。这个请求必须被送到一个确切的地址。在双方“握手”之后，TCP 将在两个应用程序之间建立一个全双工 (full-duplex) 的通信。

这个全双工的通信将占用两个计算机之间的通信线路，直到它被一方或双方关闭为止。



### 3. 断开连接

#### ◦ 四次挥手

(第一次挥手)

当客户端向服务端请求断开，这时会发送fin的标记报文。

第二次挥手

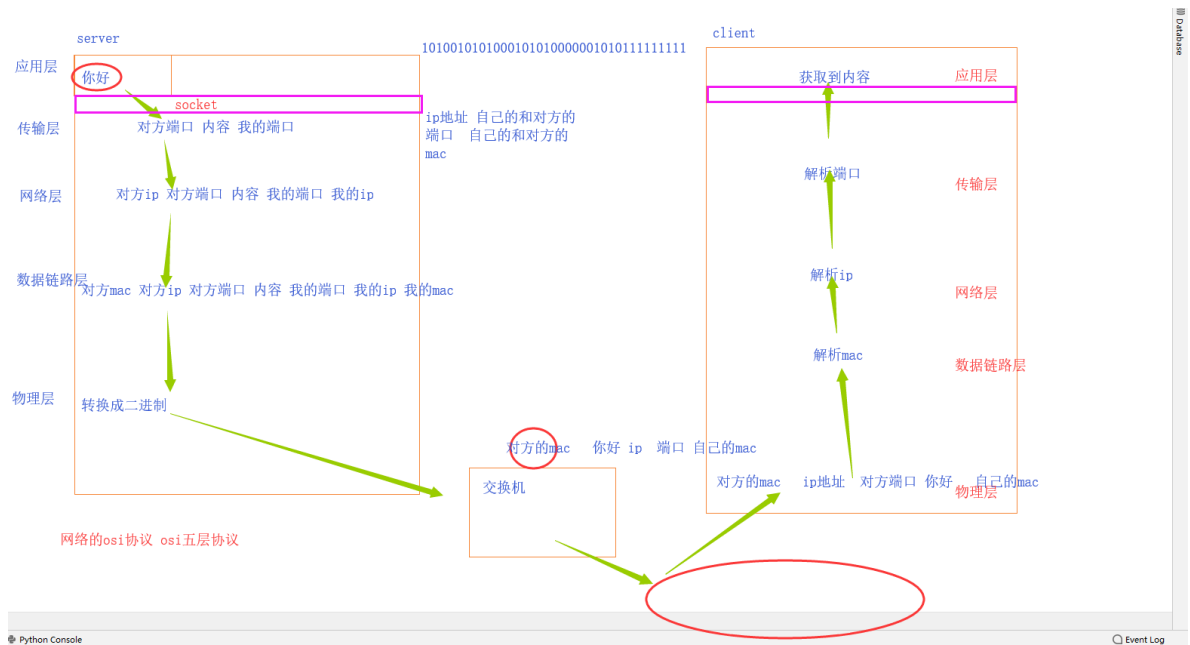
当服务端收到客户端fin断开报文时候，可能正在处理数据，此时服务端会发生ack报文。

第三次挥手

当服务端处理完成后，会再次向客户端发送FIN报文，此时可以断开连接。

第四次挥手

当客户端收到服务端的fin报文，会向服务端发送确认ACK，经过两个msl《Maximum Segment Lifetime》时长后断开连接。)



# TCP协议 上传下载\发邮件 可靠 面向连接 速度慢 能传递的数据长度不限

# 建立连接 三次握手

# 消息传递 可靠传输

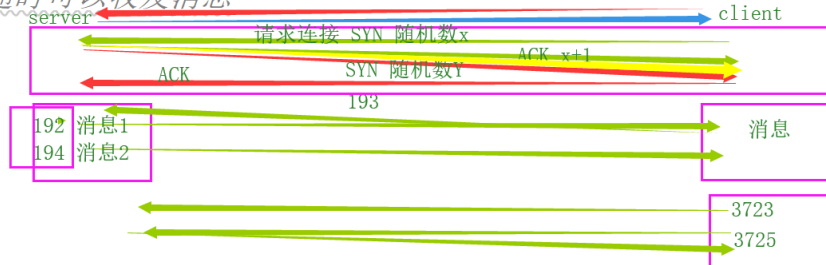
# 断开连接 四次挥手

# UDP协议 即时通讯工具 不可靠 面向数据报 速度快 能传递的数据长度有限

# 不管对方在不在 直接发送

# 不占连接

# 随时可以收发消息





### 1. 粘包现象会发生在发送端

1. 两条或多条消息间隔时间短，长度短，就会把两条消息在发送之前就拼在一起
2. 节省每一次发送消息回复的网络资源

### 2. 粘包现象会发生在接收端

1. 多条消息发送到缓存端，但没有被及时接收，或者接收的长度不足一次发送的长度
2. 数据与数据之间没有边界

### 3. 本质：发送的每一条数据之间没有边界

### 4. 步骤：

1. 计算要发送的数据字节长度
2. 把字结的长度变成4字节，借助struct

1. 计算要发送的数据字节长度
2. 把字节的长度变成4字节
3. 发送这4个字节
4. 发送数据

### 1. 服务端

```
import socket
import struct

def proto_send(msg):
 msg = msg.encode("utf-8")
 len_msg = len(msg)
 proto_len = struct.pack("i", len_msg)
 conn.send(proto_len)
 conn.send(msg)

sk = socket.socket()
sk.bind(("192.168.12.59", 8520))
sk.listen()

conn, addr = sk.accept()
msg1 = "hello"
msg2 = "world"
proto_send(msg1)
proto_send(msg2)
```

## 1. 客户端:

```
38 import struct
39 import socket
40
41 sk = socket.socket()
42
43
44 def proto_rcv():
45 len_msg = sk.recv(4)
46 len_msg = struct.unpack('i', len_msg)[0]
47 msg = sk.recv(len_msg)
48 return msg
49
50
51 sk.connect(("192.168.12.59", 8520))
52
53 for i in range(1000000): 2 * i
54 msg1 = proto_rcv()
55 print(msg1)
56 msg2 = proto_rcv()
57 print(msg2)
```

## 5. 验证客户端的合法性:

### 1. 客户端:

```
import socket
import hashlib
SECRET_KEY = b'alexbigs'

def check_client():
 randbytes = sk.recv(32)
 md5 = hashlib.md5(SECRET_KEY)
 md5.update(randbytes)
 code = md5.hexdigest().encode('utf-8')
 sk.send(code)
sk = socket.socket()
sk.connect(('127.0.0.1', 9001))
check_client()
print('正常的客户端通信')
```

### 1. 服务端:

```
import os
import socket
import hashlib
SECRET_KEY = b'alexbigsb'

def check_client(conn):
 randbytes = os.urandom(32)
 conn.send(randbytes)

 md5 = hashlib.md5(SECRET_KEY)
 md5.update(randbytes)
 code = md5.hexdigest()
 code_cli = conn.recv(32).decode('utf-8')
 return code == code_cli
```



```

sk = socket.socket()
sk.bind(('127.0.0.1',9001))
sk.listen()
while True:
 conn,addr = sk.accept()
 if not check_client(conn):continue
 print('进程正常的通信了')

import os
import hmac # hashlib==hmac
randbytes = os.urandom(32)
mac = hmac.new(SECRET_KEY,randbytes)
ret = mac.digest()
print(ret)

```

## 6. 并发的socket

### 1. 模块socketserver -- 上层模块.

客户端

```

import socket

sk = socket.socket()
sk.connect(('127.0.0.1',9001))

while True:
 ret = sk.recv(1024)
 print(ret)

```

```

import time
import socket

sk = socket.socket()
sk.bind(('127.0.0.1',9001))
sk.listen()

while True:
 conn,addr = sk.accept()
 n = 0
 while True:
 conn.send(str(n).encode('utf-8'))
 n+=1
 time.sleep(0.5)

```

## ftp作业讲解（登录，注册，大文件传输）：

### 1. 普通版：

客户端：

```

import os
import json
import socket

LOCAL_DIR = r'D:\Python_s25\day30\2 作业讲解\client\local'

```

```

sk = socket.socket()
sk.connect(('127.0.0.1',9001))

选择 要做的操作 是上传 还是下载
opt_lst = ['上传','下载']
for index,opt in enumerate(opt_lst,1):
 print(index,opt)
inp = int(input('>>>'))
if inp == 1: # 选上传
 path = input('请输入要上传的文件路径 : ')# 用户输入要上传的文件
 if os.path.isfile(path) :# 检测这个文件是否存在
 filename = os.path.basename(path) # 获取文件名
 filesize = os.path.getsize(path) # 获取文件大小
 opt_dic = {'filename':filename,'filesize':filesize,'operate':'upload'} #
把请求发过去
 json_opt = json.dumps(opt_dic)
 sk.send(json_opt.encode('utf-8'))
 # filesize = 16926596596198
 with open(path,'rb') as f:
 while filesize>0:
 content = f.read(4096)
 sk.send(content)
 filesize -= 4096
 elif inp == 2: # 选下载
 # remote文件夹中的所有文件你都可以下载
 filename = input('请输入要下载的文件名 : ') # 用户输入一个文件名
 opt_dic = {'filename': filename,'operate': 'download'}
 opt_bytes = json.dumps(opt_dic).encode('utf-8')
 sk.send(opt_bytes)# 把文件名发送到server端
 str_dic = sk.recv(1024).decode('utf-8')# 接收文件的大小
 size_dic = json.loads(str_dic)
 filepath = os.path.join(LOCAL_DIR,filename)
 with open(filepath,'wb') as f:
 while size_dic['filesize'] > 0:
 content = sk.recv(1024)
 f.write(content)
 size_dic['filesize'] -= len(content)

```

服务端:

```

import os
import json
import struct
import socket

REMOTE_DIR = r'D:\Python_s25\day30\2 作业讲解\server\remote'

sk = socket.socket()
sk.bind(('127.0.0.1',9001))
sk.listen()

conn,addr = sk.accept()
json_dic = conn.recv(1024).decode('utf-8')
opt_dic = json.loads(json_dic)
if opt_dic['operate'] == 'upload':
 filename = opt_dic['filename']
 filesize = opt_dic['filesize']
 filepath = os.path.join(REMOTE_DIR,filename)

```

```

with open(filepath, 'wb') as f:
 while filesize > 0:
 content = conn.recv(1024)
 f.write(content)
 filesize -= len(content)
 # 要接受的字节数不一定是你实际接收到的数据长度
elif opt_dic['operate'] == 'download':
 file_path = os.path.join(REMOTE_DIR, opt_dic['filename'])
 if os.path.isfile(file_path): # 判断是否存在这个文件
 size = os.path.getsize(file_path) # 返回文件大小
 dic = {'filesize': size}
 dic_bytes = json.dumps(dic).encode('utf-8')
 conn.send(dic_bytes)
 with open(file_path, 'rb') as f:
 while size > 0:
 content = f.read(4096)
 conn.send(content)
 size -= len(content)

```

## 2. 进阶版:

```

客户端:
import os
import json
import struct
import socket

LOCAL_DIR = os.path.join(os.path.dirname(__file__), 'local')

sk = socket.socket()
sk.connect(('127.0.0.1', 9001))

选择 要做的操作 是上传 还是下载
opt_lst = ['上传', '下载']
for index, opt in enumerate(opt_lst, 1):
 print(index, opt)
inp = int(input('>>>'))
if inp == 1: # 选上传
 path = input('请输入要上传的文件路径 : ') # 用户输入要上传的文件
 if os.path.isfile(path): # 检测这个文件是否存在
 filename = os.path.basename(path) # 获取文件名
 filesize = os.path.getsize(path) # 获取文件大小
 opt_dic = {'filename': filename, 'filesize': filesize, 'operate': 'upload'} #
 把请求发过去
 opt_bytes = json.dumps(opt_dic).encode('utf-8')
 num_bytes = struct.pack('i', len(opt_bytes))
 sk.send(num_bytes)
 sk.send(opt_bytes)
 # filesize = 16926596596198
 with open(path, 'rb') as f:
 while filesize > 0:
 content = f.read(4096)
 sk.send(content)
 filesize -= 4096
 elif inp == 2: # 选下载
 # remote文件夹中的所有文件你都可以下载
 filename = input('请输入要下载的文件名 : ') # 用户输入一个文件名

```

```

opt_dic = {'filename': filename, 'operate': 'download'}
opt_bytes = json.dumps(opt_dic).encode('utf-8')
num_bytes = struct.pack('i', len(opt_bytes))
sk.send(num_bytes)
sk.send(opt_bytes) # 把文件名发送到server端
bytes_len = sk.recv(4) # 要接受的数据的长度
print(bytes_len)
msg_len = struct.unpack('i', bytes_len)[0]
str_dic = sk.recv(msg_len).decode('utf-8') # 接收文件的大小
size_dic = json.loads(str_dic)
filepath = os.path.join(LOCAL_DIR, filename)
with open(filepath, 'wb') as f:
 while size_dic['filesize'] > 0:
 content = sk.recv(1024)
 f.write(content)
 size_dic['filesize'] -= len(content)

```

服务端:

```

import os
import json
import struct
import socket

REMOTE_DIR = os.path.join(os.path.dirname(__file__), 'remote')

sk = socket.socket()
sk.bind(('127.0.0.1', 9001))
sk.listen()

conn, addr = sk.accept()
num_by = conn.recv(4)
msg_len = struct.unpack('i', num_by)[0]
json_dic = conn.recv(msg_len).decode('utf-8')
opt_dic = json.loads(json_dic)

if opt_dic['operate'] == 'upload':
 filename = opt_dic['filename']
 filesize = opt_dic['filesize']
 filepath = os.path.join(REMOTE_DIR, filename)
 with open(filepath, 'wb') as f:
 while filesize > 0:
 content = conn.recv(1024)
 f.write(content)
 filesize -= len(content)
 # 要接受的字节数不一定是你实际接收到的数据长度

elif opt_dic['operate'] == 'download':
 file_path = os.path.join(REMOTE_DIR, opt_dic['filename'])
 if os.path.isfile(file_path): # 判断是否存在这个文件
 size = os.path.getsize(file_path) # 返回文件大小
 dic = {'filesize': size}
 dic_bytes = json.dumps(dic).encode('utf-8')
 bytes_len = len(dic_bytes)
 send_len = struct.pack('i', bytes_len)
 conn.send(send_len)
 conn.send(dic_bytes)

```

```

with open(file_path, 'rb') as f:
 while size > 0:
 content = f.read(4096)
 conn.send(content)
 size -= len(content)

```

### 3. 函数版本:

客户端:

```

import os
import json
import struct
import socket

LOCAL_DIR = os.path.join(os.path.dirname(__file__), 'local')

def mysend(sk, opt_dic):
 opt_bytes = json.dumps(opt_dic).encode('utf-8')
 num_bytes = struct.pack('i', len(opt_bytes))
 sk.send(num_bytes)
 sk.send(opt_bytes)

def myrecv(sk):
 bytes_len = sk.recv(4) # 要接受的数据的长度
 msg_len = struct.unpack('i', bytes_len)[0]
 str_dic = sk.recv(msg_len).decode('utf-8') # 接收文件的大小
 size_dic = json.loads(str_dic)
 return size_dic

def upload(sk):
 path = input('请输入要上传的文件路径 : ') # 用户输入要上传的文件
 if os.path.isfile(path): # 检测这个文件是否存在
 filename = os.path.basename(path) # 获取文件名
 filesize = os.path.getsize(path) # 获取文件大小
 opt_dic = {'filename': filename, 'filesize': filesize, 'operate':
'upload'} # 把请求发过去
 mysend(sk, opt_dic)
 # filesize = 16926596596198
 with open(path, 'rb') as f:
 while filesize > 0:
 content = f.read(4096)

 sk.send(content)
 filesize -= 4096

def download(sk):
 # remote文件夹中的所有文件你都可以下载
 filename = input('请输入要下载的文件名 : ') # 用户输入一个文件名
 opt_dic = {'filename': filename, 'operate': 'download'}
 mysend(sk, opt_dic)
 size_dic = myrecv(sk)
 filepath = os.path.join(LOCAL_DIR, filename)
 with open(filepath, 'wb') as f:
 while size_dic['filesize'] > 0:
 content = sk.recv(1024)
 f.write(content)
 size_dic['filesize'] -= len(content)

```

```

if __name__ == '__main__':
 sk = socket.socket()
 sk.connect(('127.0.0.1',9001))
 while True:
 # 选择 要做的操作 是上传 还是下载
 opt_lst = [('上传',upload),('下载',download),('退出',exit())]
 for index,opt in enumerate(opt_lst,1):
 print(index,opt)
 inp = int(input('>>>'))
 opt_lst[inp-1][1](sk)

```

```

import os
import sys
import json
import struct
import socket

REMOTE_DIR = os.path.join(os.path.dirname(__file__),'remote')

def myrecv(conn):
 num_by = conn.recv(4)
 msg_len = struct.unpack('i', num_by)[0]
 json_dic = conn.recv(msg_len).decode('utf-8')
 opt_dic = json.loads(json_dic)
 return opt_dic

def mysend(conn,dic):
 dic_bytes = json.dumps(dic).encode('utf-8')
 bytes_len = len(dic_bytes)
 send_len = struct.pack('i', bytes_len)
 conn.send(send_len)
 conn.send(dic_bytes)

def upload(conn):
 filename = opt_dic['filename']
 filesize = opt_dic['filesize']
 filepath = os.path.join(REMOTE_DIR, filename)
 with open(filepath, 'wb') as f:
 while filesize > 0:
 content = conn.recv(1024)
 f.write(content)
 filesize -= len(content)
 # 要接受的字节数不一定是你实际接收到的数据长度

def download(conn):
 file_path = os.path.join(REMOTE_DIR, opt_dic['filename'])
 if os.path.isfile(file_path): # 判断是否存在这个文件
 size = os.path.getsize(file_path) # 返回文件大小
 dic = {'filesize': size}
 mysend(conn, dic)
 with open(file_path, 'rb') as f:
 while size > 0:
 content = f.read(4096)
 conn.send(content)
 size -= len(content)

```

```

if __name__ == '__main__':
 sk = socket.socket()
 sk.bind(('127.0.0.1', 9001))
 sk.listen()
 while True:
 conn, addr = sk.accept()
 while True:
 try:
 opt_dic = myrecv(conn)
 if hasattr(sys.modules[__name__], opt_dic['operate']):
 getattr(sys.modules[__name__], opt_dic['operate'])(conn)
 except:
 break
 conn.close()

```

## 并发编程:

### 操作系统:

1. 多道操作系统:
  1. 第一次提出了多个程序可以同时计算机中被计算
  2. 遇到IO操作就让出CPU
  3. 把CPU让给其他程序, 让其他程序先使用CPU
  4. CPU让出这件事, 占用部分时间
  5. 两个程序来回在CPU上切换, 程序不会混乱
    1. 每个程序有独立的内存空间
    2. 每个程序在切换的前后会把当前程序的运行状态记录下来
2. CPU计算和不计算操作
  1. I/O操作 (网络操作/文件操作): 输入输出: 都是相对于**内存**来说
    1. 本质: 都是文件操作
      1. input是写入文件, 然后通过读取文件把输入的内容加载到内存
      2. print是直接写入文件, 然后通过文件展示给用户看
    2. 阻塞: sleep\input\recv\accept\recvform是不需要CPU参与的
    3. 对文件的读取: 对**硬盘**的操作一次读取相当于90w条代码
  2. Input: 向**内存**输入数据
    - read\load\accept\recv\input\recvform\connect\close
  3. Output: 向**内存**输出数据
    - write\dump\print\send\sendto\accept\connect\close

### 进程 (计算机中最小的资源分配单位)

1. 含义: 运行中的程序就是进程
2. 进程与进程之间的数据是隔离的
3. 调度:
  1. 被操作系统调度的, 每个进程中至少有一个线程
  2. 短作业优先算法

3. 先来先服务算法、

4. 时间片轮转算法

5. 多级反馈算法:

1. 依赖队列进行运行

4. 启动: 交互 (双击, 右键), 在一个进程中启动另一个

1. 负责启动一个进程的程序被称为一个父进程

2. 被启动的进程被称为一个子进程

5. 销毁:

1. 关闭交互行为

2. 在父进程结束子进程

3. 被其他进程杀死

6. 父子进程:

1. 父进程开启子进程

2. 父进程还要负责对结束的子进程进行资源的回收

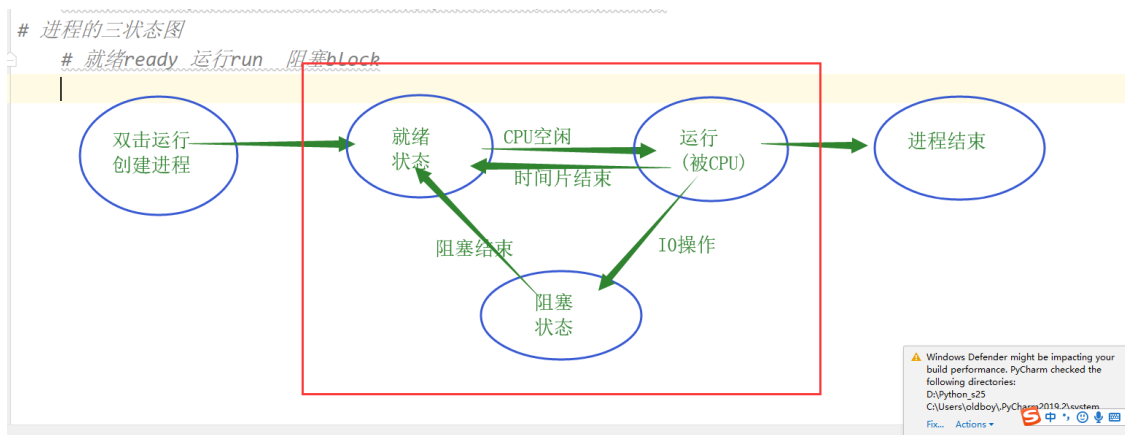
7. 进程ID--> processid --> pid

1. 在同一台机器上同一时刻不可能有两个重复的进程id

2. 进程id不能设置, 是操作系统随机分配的

3. 进程id随着多次运行一个程序可能会被多次分配, 每一次都不一样

8. 进程的三状态图:



1. 就绪状态ready

2. 运行状态run

3. 阻塞状态block

9. os里查看当前进程id: `getpid()`, process id: `getppid()`

10. multiprocessing模块: 所有和进程相关的机制都封装在multiprocessing模块中

1. 导入: `from multiprocessing import Process`



```
import os
import time
from multiprocessing import Process

def func():
 """
 在子进程中执行的func
 :return:
 """
 print('子进程 :', os.getpid(), os.getppid())
 time.sleep(3)

if __name__ == '__main__':
 p = Process(target=func)
 p.start()
 print('主进程 :', os.getpid())
```

在执行这一步的时候给予进程开辟一个新的内存空间，但原来的代码继续执行，所以会先打印主进程，继续打印子进程

```
import os
import time
from multiprocessing import Process

def func():
 """
 在子进程中执行的func
 :return:
 """
 print('子进程 :', os.getpid(), os.getppid())
 time.sleep(3)

if __name__ == '__main__':
 p = Process(target=func)
 p.start()
 print('主进程 :', os.getpid())
```

4. 进程概念

主进程 : 4048  
子进程 : 11656 4048

- 11. 并行：多个cpu同时执行多个程序
- 12. 并发：多个程序在同时执行
- 13. 同步：一个程序执行完了再调用另一个，并且在调用的过程中还要等待这个程序执行完毕
- 14. 异步：一个程序在执行中调用了另一个，但是不等待这个任务完毕就继续执行下一个任务
- 15. 阻塞：CPU不工作
- 16. 非阻塞：CPU工作

## Process模块

- 1. 主进程是在子进程执行完毕之后才结束
- 2. 主进程回收子进程的资源

```

import os
import time
from multiprocessing import Process

def son_process():
 print('strat son',os.getpid())
 time.sleep(50)
 print('end son')

if __name__ == '__main__':
 print(os.getpid())
 Process(target=son_process).start()

```

### 3. 多个子进程:

```

from multiprocessing import Process
import os
import time
def son_process():
 print('strat son',os.getpid())
 time.sleep(1)
 print('end son')

if __name__ == '__main__':
 print(os.getpid())
 Process(target=son_process).start()
 Process(target=son_process).start()
 Process(target=son_process).start()

def son_process():
 print('strat son',os.getpid())
 time.sleep(1)
 print('end son')

if __name__ == '__main__':
 for i in range(5):
 Process(target=son_process).start()

```

### 4. 阻塞: join的用法

```

from multiprocessing import Process
import time
def son_process(n):
 print('start', n)
 time.sleep(2)
 print('end',n)

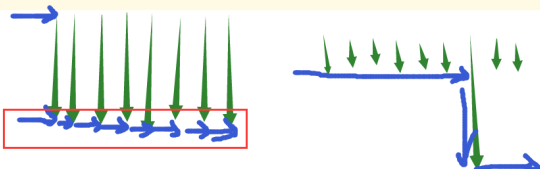
if __name__ == '__main__':
 p_l = []
 for i in range(10):
 p = Process(target=son_process,args=(i,))
 p.start() # start相当于告诉操作系统要开启一个子进程,而子进程的调度是由操作系
 统控制的
 p_l.append(p)
 for p in p_l:p.join() # join 如果执行这句话的时候子进程已经结束了,那么join就不
 阻塞了

```

```
print('所有任务结束')
```

```
import time
def son_process(n):
 print('start', n)
 time.sleep(2)
 print('end', n)

if __name__ == '__main__':
 p_l = []
 for i in range(10):
 p = Process(target=son_process, args=(i,))
 p.start()
 p_l.append(p)
 for p in p_l: p.join() # join 如果执行这句话的时候子进程已经结束了,那么join就不阻塞了
 print('所有任务结束')
```



5. 守护进程 (daemon)：守护进程会随着父进程的代码结束而结束

1. 正常情况下，父进程永远会等着子进程结束

面试题：

```
import time
from multiprocessing import Process
def son():
 while True:
 time.sleep(1)
 print("in son")

def son2():
 print("son2")
 time.sleep(10)
 print("end son2")

if __name__ == '__main__':
 p = Process(target=son)
 p.daemon = True #守护的是son进程
 p.start()
 p = Process(target=son2).start()
 time.sleep(5)
```

输出结果：

```
son2
in son
in son
in son
in son
end son2
```

```

import time
def son():
 while True:
 time.sleep(1)
 print('in son')

def son2():
 print('start son2')
 time.sleep(10)
 print('end son2')

if __name__ == '__main__':
 p = Process(target=son)
 p.daemon = True
 p.start()
 Process(target=son2).start()
 time.sleep(5)

```

2. 如果设置了守护进程，父进程的代码结束之后，守护进程也会跟着结束

3. 代码结束与进程结束的区别：

1. 没设置守护进程

- 子进程的代码和主进程的代码自己执行自己的，互相之间没关系
- 如果主进程的代码先结束，主进程不结束，等子进程代码结束，回收子进程的资源，主进程才结束
- 如果子进程的代码先结束，主进程边回收子进程的资源边执行自己的代码，当代码和资源都回收结束，主进程才结束

2. 设置了守护进程

- 子进程的代码和主进程的代码自己执行自己的，互相之间没关系
- 一旦主进程的代码先结束，主进程会先结束掉子进程，然后回收资源，然后主进程才结束

4. 异步非阻塞 (terminate) :

```

import time
from multiprocessing import Process
def son():
 while True:
 time.sleep(2)
 print("in son")
if __name__ == '__main__':
 p = Process(target=son)
 p.start()
 time.sleep(5)
 p.terminate()
 print("我还可以继续做事情")

```

输出结果：

```

in son
in son
我还可以继续做事情

```

5. 进程方法总结：

```

import time
from multiprocessing import Process
def son():
 while True:

```

```

 time.sleep(1)
 print('in son')
 if __name__ == '__main__':
 p = Process(target=son)
 p.start()
 time.sleep(5)
 print(p.is_alive())
 p.terminate() # 异步非阻塞操作
 time.sleep(0.1)
 print(p.is_alive())
 print('我还可以继续做其他的事情,主进程的代码并不结束')

```

1. p.start(): 启动进程, 并调用该子进程中的p.run()
2. p.run():进程启动时运行的方法, 正是它去调用target指定的函数, 我们自定义类的类中一定要实现该方法
3. p.terminate():强制终止进程p, 不会进行任何清理操作, 如果p创建了子进程, 该子进程就成了僵尸进程, 使用该方法需要特别小心这种情况。如果p还保存了一个锁那么也将不会被释放, 进而导致死锁
4. p.is\_alive():如果p仍然运行, 返回True
5. p.join([timeout]):主线程等待p终止 (强调: 是主线程处于等的状态, 而p是处于运行的状态)。timeout是可选的超时时间, 需要强调的是, p.join只能join住start开启的进程, 而不能join住run开启的进程
6. 面向对象的方式开启子进程

不传参数:

```

from multiprocessing import Process
import os
class MyProcess(Process):
 def run(self):
 print(os.getpid())

if __name__ == "__main__":
 print("主进程: ",os.getpid())
 MyProcess().start()

```

传参数:

```

from multiprocessing import Process
import os
class MyProcess(Process):
 def __init__(self,arg1,arg2):
 super().__init__()
 self.a1 = arg1
 self.a2 = arg2
 def run(self):
 print(os.getpid(),self.a1,self.a2)

if __name__ == "__main__":
 print("主进程:",os.getpid())
 MyProcess(1,2).start()

```

```
面向对象的方式开启子进程
class MyProcess(Process):
 def run(self):
 print(os.getpid())

if __name__ == '__main__':
 print('主 :', os.getpid())
 MyProcess().start()
```

7. 进程之间的数据是隔离的

```
n = 0
def son():
 global n
 n += 1

if __name__ == '__main__':
 p_l = []
 for i in range(20):
 p = Process(target=son)
 p.start()
 p_l.append(p)
 for p in p_l: p.join()
 print(n)
```

## 锁：当多个进程同时操作文件/共享的一些数据的时候就会出现数据不安全

1. 目的：用来保证数据的安全
2. 如果多个进程同时对一个文件进行操作会出现什么问题？
  1. 读数据：可以同时读
  2. 写数据：不能同时写

```
from multiprocessing import Process
from multiprocessing import Lock
def change():
 print("一部分并发的代码，多个进程之间互相不干扰的进行")
 lock.acquire()
 with open("file", "r") as f:
 content = f.read()
 num = int(content)
 num += 1
 for i in range(1000000):
 i += 1
 with open("file", "w") as f:
 f.write(str(num))
 lock.release()
 print("另一部分并发的代码，多个进程之间互相不干扰的执行着")

if __name__ == '__main__':
```

```
lock = Lock()
for i in range(10):
 Process(target=change, args=(lock,)).start()
```

```
from multiprocessing import Process
from multiprocessing import Lock
def change(lock):
 print("一部分并发的代码，多个进程之间互相不干扰的进行")
 lock.acquire()
 with open("file", "r") as f:
 content = f.read()
 num = int(content)
 num += 1
 for i in range(1000000):
 i += 1
 with open("file", "w") as f:
 f.write(str(num))
 lock.release()
 print("另一部分并发的代码，多个进程之间互相不干扰的执行着")
if __name__ == '__main__':
 lock = Lock()
 for i in range(10):
 Process(target=change, args=(lock,)).start()
```

## 进程之间的通信 (IPC) : Inter Process Communication3

1. IPC= 内置的模块实现的机制：队列/管道 以及第三方工具：redis, rabbitMQ memcache
2. 进程之间数据是隔离的，意味着不能通信

```
from multiprocessing import Queue, Process

def son(q):
 print(q.get())

if __name__ == '__main__':
 q = Queue()
 p = Process(target=son, args=(q,))
 p.start()
 q.put(123)
输出结果:
123
```

3. 进程安全：在进程之间维护数据的安全
4. 队列是进程安全的（进程队列保证了进程的数据安全）
5. 队列是基于文件+锁实现的
6. 队列都是先进先出的
7. 队列提供的方法：
  1. get：是一个同步阻塞方法，会阻塞直到数据来

```
from multiprocessing import Queue, Process
q = Queue()
q.put({1,2,3})
num = q.get() #get是一个同步阻塞方法，会阻塞直到数据来
print(num)
```

输出结果: {1, 2, 3}

2. put: 是一个同步阻塞方法,会阻塞直到队列不满

```
q = Queue(2)
q.put({1,2,3})
q.put({1,2,3})
```

##Queue([maxsize])表示队列中最大项数，如果省略此参数，则表示大小无限制

```
q = Queue(2)
```

```
q.put({1,2,3})
```

```
q.put({1,2,3})
```

```
q.put({1,2,3})
```

限制队列的长度

3. q.get\_nowait()

```
import queue
q = Queue(3)
try:
 for i in range(4):
 q.put_nowait(i) #put_nowait 同步非阻塞方法
except queue.Full:
 print(i)

q2 = Queue(2)
try:
 print(q2.get_nowait())
except queue.Empty:
 pass
```

4. q.put(item [, block [, timeout ]])

将item放入队列。如果队列已满，此方法将阻塞至有空间可用为止。block控制阻塞行为，默认为True。如果设置为False，将引发Queue.Empty异常（定义在Queue库模块中）。timeout指定在阻塞模式中等待可用空间的时间长短。超时后将引发Queue.Full异常。



#### 5. q.qsize()

返回队列中目前项目的正确数量。此函数的结果并不可靠，因为在返回结果和在稍后程序中使用结果之间，队列中可能添加或删除了项目。在某些系统上，此方法可能引发 `NotImplementedError` 异常。

#### 6. q.empty()

如果调用此方法时 `q` 为空，返回 `True`。如果其他进程或线程正在往队列中添加项目，结果是不可靠的。也就是说，在返回和使用结果之间，队列中可能已经加入新的项目。

#### 7. q.full()

如果 `q` 已满，返回为 `True`。由于线程的存在，结果也可能是不可靠的（参考 `q.empty()` 方法）。

8. 生产者消费模型：生产者消费者模式是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通讯，而通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。

9. 队列：Queue == 管道Pipe + 锁 来实现

1. 管道Pipe：基于文件实现的（socket+pickle）== 数据不安全

2. 队列Queue：基于文件（socket+pickle）+ 锁（lock）实现的 == 数据安全

## 线程（计算机中能被操作系统调度的最小单位）：

1. 每个程序执行到哪个位置是被记录下来的
2. 在进程中，有一条线程是负责具体的执行程序的

## 本节重点

- 初识爬虫
- 单例模式,重点面试(需要默写)
  - 为什么要加锁?
  - 为什么要做判断?
- 进程和线程的区别?,重点.面试
- GIL锁,重点面试
- 线程的常用功能: start/join, 重点

## 爬虫小实例：

### 1. 安装和使用

1. 安装第三方模块： `pip3 install requests`
2. 使用： `import requests`

代码示例：

下载图片：

```
import requests
```

```
url_list = [
```

```
'https://www3.autoimg.cn/newsdfs/g26/M02/35/A9/120x90_0_autohomecar__ChsEe12AXQ6A00H_AAfocMs8nzU621.jpg',
```

```
'https://www2.autoimg.cn/newsdfs/g30/M01/3C/E2/120x90_0_autohomecar__ChcCSV2BBICAUntfAAdjJFd6800429.jpg',
```

```
'https://www3.autoimg.cn/newsdfs/g26/M0B/3C/65/120x90_0_autohomecar__ChcCP12
BFCmAI083AAGq7vK0sGY193.jpg'
]
```

```
for url in url_list:
 ret = requests.get(url)
 file_name = url.rsplit('/',maxsplit=1)[-1]
 with open(file_name,mode='wb') as f:
 # 下载小文件
 # f.write(ret.content)

 # 下载大文件
 for chunk in ret.iter_content():
 f.write(chunk)
```

注意：以上代码是基于串行实现的

## 线程与进程的区别

1. 进程是计算机资源分配的最小单位
2. 线程是计算机中能被CPU调度的最小单位
3. 一个进程中可以有多个线程
4. 同一个进程中的线程可以共享进程中的资源
5. 一个进程中至少有一个线程
6. 在Python中因为有GIL锁，同一时刻保证一个进程中只有一个线程可以被cpu调度，所以在使用python开发时计算密集型的代码使用多进程，IO密集型使用多线程
7. 默认进程之间无法进行资源共享,如果主要想要通讯可以基于:文件/网络/Queue.

## 多线程的应用（threading模块）

1. 快速应用：

```
import threading
def task(arg):
 pass
#实例化一个线程对象
t = threading.Thread(target=task,args = ("xxx",))
#将线程提交给cpu
t.start()
```

```
import threading
def task(arg):
 ret = requests.get(arg)
 filename = arg.rsplit("/",1)[-1]
 with open (filename,"wb")as f:
 f.write(ret.content)

for url in url_lst:
 #实例化一个线程对象
 t = threading.Thread(target=ask,arg=(url,))
 t.start()
```

## 2. 常见方法:

1. t.start(),将线程提交给cpu, 由cpu进行调度

2. t.join(),等待

■ join和start面试题:

```
import threading
loop = 1000000
num = 0

def _add(count):
 global num
 for i in range(count):
 num += 1
t = threading.Thread(target=_add, args=(loop,))
t.start()
t.join()
print(num)
```

```
loop = 10000000
number = 0

def _add(count):
 global number
 for i in range(count):
 number += 1

def _sub(count):
 global number
 for i in range(count):
 number -= 1

t1 = threading.Thread(target=_add, args=(loop,))
t2 = threading.Thread(target=_sub, args=(loop,))
t1.start()
t2.start()

print(number)
```

```
loop = 10000000
number = 0

def _add(count):
 global number
 for i in range(count):
 number += 1

def _sub(count):
 global number
 for i in range(count):
 number -= 1

t1 = threading.Thread(target=_add, args=(loop,))
t2 = threading.Thread(target=_sub, args=(loop,))
t1.start()
```

```
t2.start()
t1.join() # t1线程执行完毕,才继续往后走
t2.join() # t2线程执行完毕,才继续往后走

print(number)
```

```
loop = 10000000
number = 0

def _add(count):
 global number
 for i in range(count):
 number += 1

def _sub(count):
 global number
 for i in range(count):
 number -= 1

t1 = threading.Thread(target=_add, args=(loop,))
t2 = threading.Thread(target=_sub, args=(loop,))
t1.start()
t1.join() # t1线程执行完毕,才继续往后走
t2.start()
t2.join() # t2线程执行完毕,才继续往后走

print(number)
```

### 3. t.setDaemon(),设置成为守护线程

```
import threading
import time

def task(arg):
 time.sleep(5)
 print('任务')

t = threading.Thread(target=task, args=(11,))
t.setDaemon(True)
t.start()

print('END')
```

### 4. 线程名称的设置和获取

```
import threading

def task(arg):
 # 获取当前执行此代码的线程
 name = threading.current_thread().getName()
 print(name)

for i in range(10):
 t = threading.Thread(target=task, args=(11,))
 t.setName('日魔-%s' %i)
 t.start()
```

5. run(), 自定义线程时,cpu调度执行的方法

```
class RiMo(threading.Thread):
 def run(self):
 print('执行此线程', self._args)

obj = RiMo(args=(100,))
obj.start()
```

6. 练习题: 基于socket 和 多线程实现类似于socketserver模块的功能.

```
import socket
import threading

def task(connect, address):
 pass

server = socket.socket()
server.bind(('127.0.0.1', 9000))
server.listen(5)
while True:
 conn, addr = server.accept()
 # 处理用户请求
 t = threading.Thread(target=task, args=(conn, addr,))
 t.start()
```

## 线程安全

1. 多个线程同时操作一个“东西”，不存在数据混乱
2. 线程安全: logging模块/列表/字典/Queue
3. 线程不安全: 自己做文件操作/同时操作一个文件或数字
4. 使用锁来保证数据的安全，来了多个线程，使用锁让他们逐一执行
5. 线程之间数据安全
  1. 但凡是带着判断，判断之后要做xxxx都需要加锁
  2. if/while/+=/-=/\*=/ /= 都需要加锁
6. 带append, pop, extend方法的都是线程安全的
7. lock: 互斥锁

```

import threading
import time

num = 0
#加线程锁
lock= threading.Lock()

def task():
 global num
 # 申请锁
 lock.acquire()
 num += 1
 time .sleep(0.2)
 print(num)
 #释放锁
 lock.release()

 #with lock:
 #num +=1
 #time.sleep(0.2)
 #print(num)
for i in range(10):
 t = threading.Thread(target=task)
 t.start()

```

8. Rlock：递归锁，支持多次上锁，推荐使用递归锁

```

import threading
import time

num = 0
线程锁
lock = threading.RLock()

def task():
 global num
 # 申请锁
 lock.acquire()
 num += 1
 lock.acquire()
 time.sleep(0.2)
 print(num)
 # 释放锁
 lock.release()
 lock.release()

for i in range(10):
 t = threading.Thread(target=task)
 t.start()

```

9. 加强版使用线程锁完成单例模式：

```

import threading

```

```

import time
class Singleton:
 ininstance = None
 lock = threading.RLock()
 def __init__(self,name):
 self.name = name

 def __new__(cls,*args,**kwargs):
 if cls.ininstance:
 return cls.ininstance
 with cls.lock:
 if cls.ininstance:
 return cls.ininstance
 time.sleep(0.2)
 cls.ininstance = object.__new__(cls)
 return cls.ininstance
def task():
 obj = Singleton("x")
 print(obj)
for i in range(10):
 t = threading.Thread(target = task)
 t.start()

```

10. 死锁:

11. 锁两次

12. 多个锁交叉使用

```

import threading
import time
lock1 = threading.RLock()
lock2 = threading.RLock()
def fenglin():
 lock1.acquire()
 time.sleep(1)
 lock2.acquire()
def rimo():
 lock2.acquire()
 time.sleep(1)
 lock1.acquire()
t1 = threading.Thread(target=fenglin)
t2 = threading.Thread(target=rimo)
t1.start()
t2.start()

```

**GIL（全局解释器锁）：**导致了CPython解释器下同一个进程下的多个线程不能利用多核优势，用一时刻保证一个进程中只有一个线程可以被cpu调度，所以在使用python开发时计算密集型的代码使用多进程，IO密集型使用多线程

- 在Cpython中如果创建多线程无法应用计算机的多核优势
- 由于垃圾回收机制gc不能在多线程环境下正常进行引用计数
- 作用：

- 一个进程只有一个线程能被CPU调度
- 列表/字典线程安全起到了作用

## 本节重点

1. 线程池&进程池 [用法+示例2]
2. 协程 (进程/线程/协程的区别?) 最重要
3. 发邮件的功能
4. 锁
5. 线程安全
6. 生产者消费者模型

## 线程池

1. Python 2中没有线程池，Python3之后解释器才提供了线程池
2. 作用：保证程序中最多可以创建的线程个数，防止无节制的创建线程，导致性能降低

示例1:

```
import threading
import time
def task():
 time.sleep(0.2)
 print("开始")

num = int(input("请输入要执行的次数: "))
for i in range(num):
 t = threading.Thread(target=task)
 t.start()
```

示例2:

```
import time
from concurrent.futures import ThreadPoolExecutor
def task(n1,n2):
 time.sleep(0.1)
 print("开始")

#创建线程池
pool = ThreadPoolExecutor(10)
for i in range(100):
 pool.submit(task,i,1)
print("结束")
#等待线程池中的任务执行完毕后，再继续执行
pool.shutdown(True)
print("其他操作，依赖线程池进行的其他操作")
```

示例3:

```
import time
from concurrent.futures import ThreadPoolExecutor

def task(arg):
 time.sleep(2)
 print("开始")
 return 666

#创建线程池
```



```
pool = ThreadPoolExecutor(10)
ret = pool.map(task, range(10))
print("End", ret)
pool.shutdown(True)
for em in ret:
 print(em)
```

示例4:

```
import time
from concurrent.futures import ThreadPoolExecutor
def task(arg1, arg2):
 time.sleep(0.2)
 print("任务")
 return arg1 + arg2

#创建线程
pool = ThreadPoolExecutor(10)

future_lst = []
for i in range(10):
 #提交线程
 fu = pool.submit(task, i, 2)
 future_lst.append(fu)
#等待线程执行
pool.shutdown(True)
for fu in future_lst:
 print(fu.result())#result是固定的写法，阻塞方法
```

3. 回调函数:

## 进程池

```
import time
from concurrent.futures import ProcessPoolExecutor

def task(n1, n2):
 time.sleep(0.2)
 print("开始任务")
 return n1 + n2
if __name__ == '__main__':
 future = []
 #创建线程
 pool = ProcessPoolExecutor(61)#实际测试这里最大填写的是61，加上主进程之后就是62，进程池最大就是63
 for i in range(20):
 pool1 = pool.submit(task, i, 1)
 future.append(pool1)
 pool.shutdown(True)
 print("结束")
 for fu in future:
```

```
print(fu.result())
```

输出结果:

## 协程

1. 协程：对于单线程下，我们不可避免程序中出现io操作，但如果我们能在自己的程序中（即用户程序级别，而非操作系统级别）控制单线程下的多个任务能在一个任务遇到io阻塞时就切换到另外一个任务去计算，这样就保证了该线程能够最大限度地处于就绪态，即随时都可以被cpu执行的状态，相当于我们在用户程序级别将自己的io操作最大限度地隐藏起来，从而可以迷惑操作系统，让其看到：该线程好像是一直在计算，io比较少，从而更多的将cpu的执行权限分配给我们的线程。
2. 本质：在单线程下，由用户自己控制一个任务遇到io阻塞了就切换另外一个任务去执行，以此来提升效率。不能利用多核

```
#安装 pip3 install gevent

from greenlet import greenlet

def test1():
 print("1")
 gr2.switch()
 print("3")
 gr2.switch()

def tast2():
 print("2")
 gr1.switch()
 print("4")

gr1 = greenlet(test1)
gr2 = greenlet(tast2)
gr1.switch()
输出结果:
1
2
3
4
```

### 3. 协程+ IO切换

```
安装gevent; pip3 install gevent
from gevent import monkey
monkey.patch_all()

import gevent
import time

def eat():
 print("eat food is meat")
 time.sleep(0.2)
 print("eat food is su")
```

```
def play():
 print("play basketball")
 time.sleep(0.2)
 print("play football")
```

```
gr1 = gevent.spawn(eat)
gr2 = gevent.spawn(play)
gevent.joinall([gr1,gr2])
```

```
from gevent import monkey
monkey.patch_all()
```

固定格式，每次必须先书写这个

```
import gevent
import time
```

```
def eat():
 print("eat food is meat")
 time.sleep(0.2)
 print("eat food is su")
```

```
def play():
 print("play basketball")
 time.sleep(0.2)
 print("play football")
```

```
gr1 = gevent.spawn(eat)
gr2 = gevent.spawn(play)
gevent.joinall([gr1,gr2])
```

#### 4. 用协程+IO切换实现爬虫

```
from gevent import monkey
monkey.patch_all()
```

```
import gevent
import requests
```

```
def f1(url):
 print("GET:%s"%url)
 data = requests.get(url)
 print('%d bytes received from %s.' % (len(data.content), url))
```

```
def f2(url):
 print('GET: %s' % url)
 data = requests.get(url)
 print('%d bytes received from %s.' % (len(data.content), url))
```

```
def f3(url):
 print('GET: %s' % url)
```

```

data = requests.get(url)
print('%d bytes received from %s.' % (len(data.content), url))

gevent.joinall([
 gevent.spawn(f1, 'https://www.python.org/'),
 gevent.spawn(f2, 'https://www.yahoo.com/'),
 gevent.spawn(f3, 'https://github.com/'),
])

```

## 面试必问

- 进程，线程，协程的区别
  - 都可以提高并发能力
  - 进程/线程是计算机中真实存在的，协程是程序员人为创造出来的
  - 协程又可以称为“微线程”，实际上是让一个线程轮番去执行一些任务
  - 资源开销小，能够把单线程的效率提高，协程能够是别的io操作不如线程多

## 生产者销售者模型

### 队列

- Queue
- redis中的列表
- rabbitMQ

示例：

```

from queue import Queue
q = Queue()

q.put('123')
q.put('456')

v1 = q.get()
v2 = q.get()
print(v1,v2)

#默认阻塞
v1= q.get()
print(v1)

```

### 发邮件示例

```

import threading
from queue import Queue
import time

q = Queue()

def send(to, subject, text):
 import smtplib
 from email.mime.text import MIMEText
 from email.utils import formataddr

 # 以下为邮件内容

```

```

msg = MIMEText(text, 'plain', 'utf-8')
msg['From'] = formataddr(['大哥', 'heyulong1214@163.com'])
msg['To'] = formataddr(['好看', to])
msg['Subject'] = subject

sk = smtplib.SMTP_SSL("smtp.163.com", 465)
sk.login("heyulong1214", "hyl121400")
sk.sendmail("heyulong1214@163.com", [to,], msg.as_string())
sk.quit()

```

```

def product(h):
 """
 生产者添加任务
 :return:
 """
 print('生产者往队列中放了10个任务', h)
 info = {'to': '83919815@qq.com', 'text': '傻子，等我回来', 'subject': '好友请求'}
 q.put(info)

```

```

def consumer():
 """
 消费者
 :return:
 """
 while True:
 print("消费者来领取任务")
 info = q.get()
 print(info)
 send(info['to'], info['subject'], info['text'])

```

```

for h in range(10):
 t = threading.Thread(target=product, args=(h,))
 t.start()

```

```

for i in range(10):
 t = threading.Thread(target=consumer)
 t.start()

```

####