

IO并发编程

Tedu Python 教学部

Author: 吕泽

IO并发编程

- 进程线程编程

- 进程（process）

 - 进程理论基础

- 基于fork的多进程编程

 - fork使用

 - 进程相关函数

 - 孤儿和僵尸

 - 项目：群聊聊天室

- multiprocessing 模块创建进程

 - 进程创建方法

 - 自定义进程类

 - 进程池实现

- 进程间通信（IPC）

 - 消息队列

- 线程编程（Thread）

 - 线程基本概念

 - threading模块创建线程

 - 线程对象属性

 - 自定义线程类

- 同步互斥

 - 线程间通信方法

 - 线程同步互斥方法

 - 线程Event

 - 线程锁 Lock

 - 死锁及其处理

- python线程GIL

- 进程线程的区别联系

 - 区别联系

 - 使用场景

 - 要求

- 网络并发通信

 - 常见网络通信模型

 - 基于fork的多进程网络并发模型

 - 实现步骤

 - 基于threading的多线程网络并发

 - 实现步骤

 - ftp 文件服务器

- IO并发

 - IO 分类

 - 阻塞IO

 - 非阻塞IO

 - IO多路复用

 - select 方法

 - @@扩展: 位运算

 - poll方法

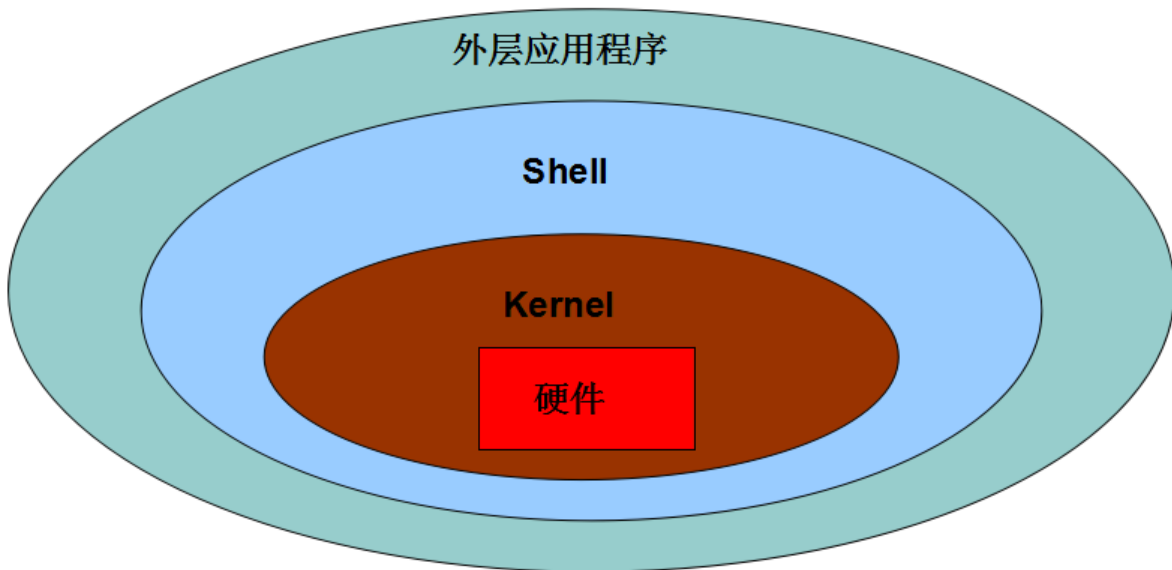
进程线程编程

1. 意义：充分利用计算机CPU的多核资源，同时处理多个应用程序任务，以此提高程序的运行效率。
2. 实现方案：多进程，多线程

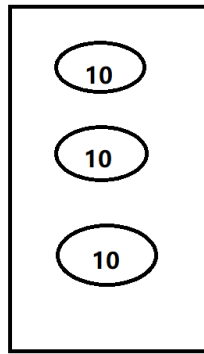
进程（process）

进程理论基础

1. 定义：程序在计算机中的一次运行。
 - 程序是一个可执行的文件，是静态的占有磁盘。
 - 进程是一个动态的过程描述，占有计算机运行资源，有一定的生命周期。
2. 系统中如何产生一个进程
 - 【1】 用户空间通过调用程序接口或者命令发起请求
 - 【2】 操作系统接收用户请求，开始创建进程
 - 【3】 操作系统调配计算机资源，确定进程状态等
 - 【4】 操作系统将创建的进程提供给用户使用



3. 进程基本概念



顺序执行 --》 30

3cpu同时执行--》 10

1cpu内核？

计算密集 --》 30

IO密集 --》 <30

- cpu时间片：如果一个进程占有cpu内核则称这个进程在cpu时间片上。
- PCB(进程控制块)：在内存中开辟的一块空间，用于存放进程的基本信息，也用于系统查找识别进程。
- 进程ID（PID）：系统为每个进程分配的一个大于0的整数，作为进程ID。每个进程ID不重复。

Linux查看进程ID： ps -aux

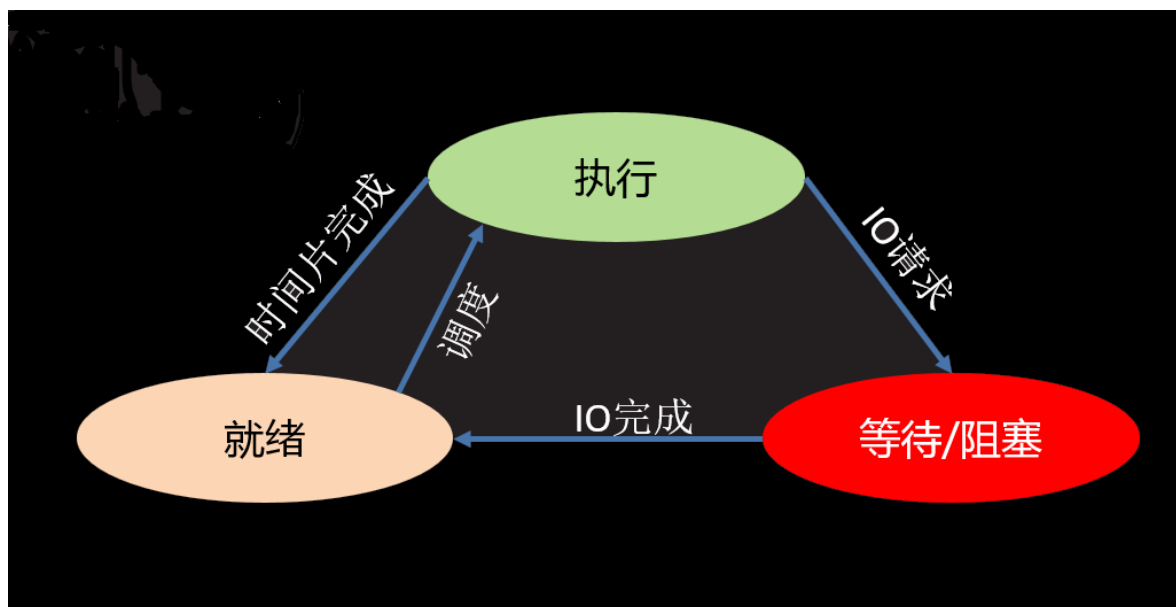
```

tarena@tedu:~$ ps -aux
USER      PID %CPU %MEM    VSZ   RSS  TTY  STAT  START  TIME  COMMAND
root         1   0.0   0.0 225524  6652   ?    Ss    11月27  2:01  /sbin/init
sp
root         2   0.0   0.0     0      0   ?    S     11月27  0:00  [kthreadd]
root         3   0.0   0.0     0      0   ?    I<    11月27  0:00  [rcu_gp]
root         4   0.0   0.0     0      0   ?    I<    11月27  0:00  [rcu_par_gp]
```

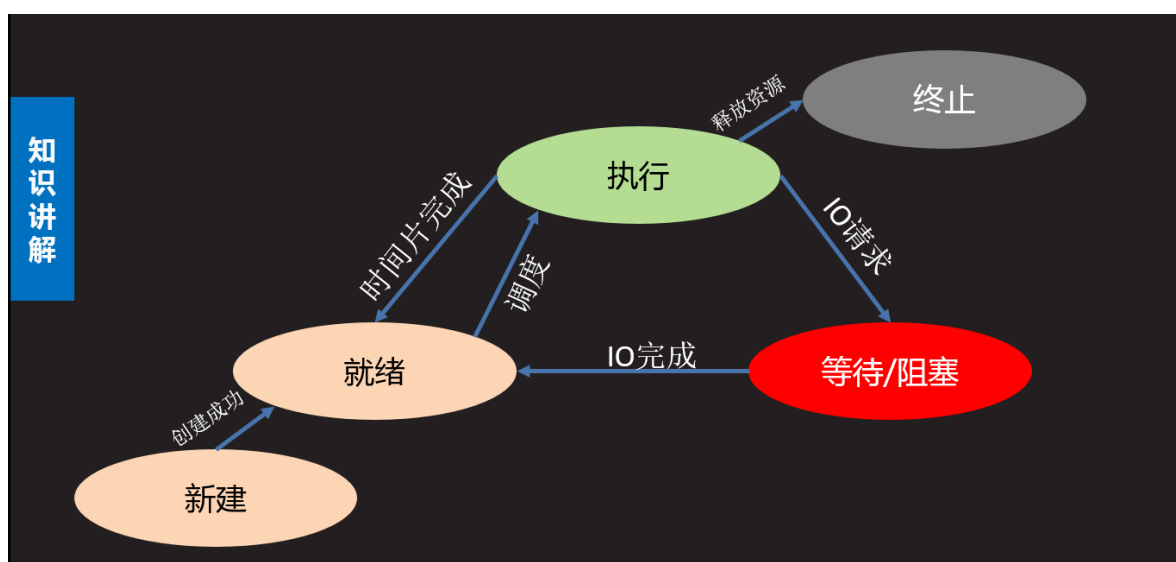
- 父子进程：系统中每一个进程(除了系统初始化进程)都有唯一的父进程，可以有0个或多个子进程。父子进程关系便于进程管理。

查看进程树： pstree

- 进程状态
 - 三态
 - 就绪态：进程具备执行条件，等待分配cpu资源
 - 运行态：进程占有cpu时间片正在运行
 - 等待态：进程暂时停止运行，让出cpu



- 五态 (在三态基础上增加新建和终止)
 新建： 创建一个进程，获取资源的过程
 终止： 进程结束，释放资源的过程



- 状态查看命令： `ps -aux` (STAT列)

S 等待态

R 执行态

Z 僵尸

+ 前台进程 `#python3 http_test.py &` 后台运行
 l 有多线程的

- 进程的运行特征
 - 【1】 多进程可以更充分使用计算机多核资源
 - 【2】 进程之间的运行互不影响，各自独立
 - 【3】 每个进程拥有独立的空间，各自使用自己空间资源

面试要求

1. 什么是进程，进程和程序有什么区别
2. 进程有哪些状态，状态之间如何转化
3. 创建父进程的时候为啥要同时创建子进程？主要是开启程序的时候同时能运行多个任务，其次就是提高程序效率。

基于fork的多进程编程

fork使用

代码示例：day5/fork.py

```
"""
fork.py fork 演示
"""

import os
from time import sleep

pid = os.fork()
if pid < 0:
    print("Create process failed")
elif pid == 0:
    sleep(3)
    print("The new process")
else:
    sleep(4)
    print("The old process")

print("Fork test over")
```

代码示例：day5/fork1.py

```
"""
fork1.py fork演示2
"""

import os
from time import sleep
print("=====") #验证
a = 1
# 创建子进程
pid = os.fork()
if pid < 0:
    print('Create process failed')
elif pid == 0:
    # 子进程执行部分
    print("Child process")
    print("a =", a) # 从父进程拷贝了内存a的值
    a = 10000 # 只修改自己空间的a
else:
    sleep(1)
    # 父进程执行部分
    print("Parent process")
    print("a:", a)

print("All a=", a) # 父子进程都执行
```

```
pid = os.fork()
```

功能：创建新的进程

返回值：整数，如果创建（父）进程失败返回一个负数，如果成功则在原有（父）进程中返回新（子）进程的PID，在新（子）进程中返回0

注意

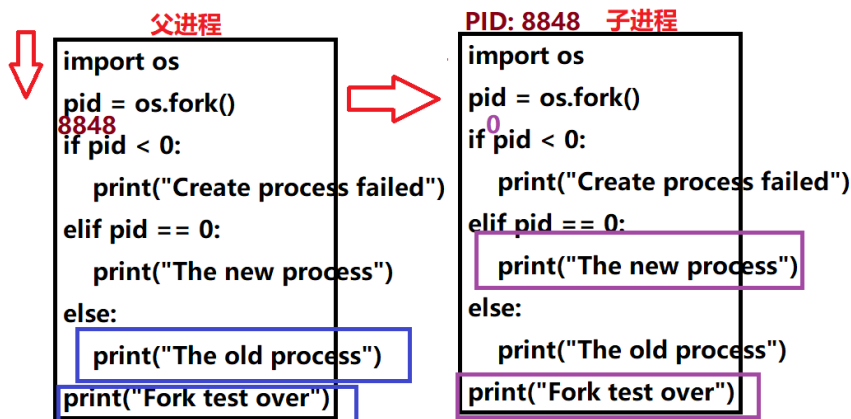
- 子进程会复制父进程全部内存空间，从fork下一句开始执行。
- 父子进程各自独立运行，运行顺序不一定。
- 利用父子进程fork返回值的区别，配合if结构让父子进程执行不同的内容几乎是固定搭配。
- 父子进程有各自特有特征比如PID PCB 命令集等。
- 父进程fork之前开辟的空间子进程同样拥有，父子进程对各自空间的操作不会相互影响。

进程相关函数

代码示例：day5/get_pid.py

```
"""
获取进程的PID号
"""
import os
from time import sleep

pid = os.fork()
if pid < 0:
    print("Error")
elif pid == 0:
    sleep(1)
    print("Child PID:", os.getpid()) # 子PID
    print("Get Parent PID:", os.getppid()) # 父PID
else:
    print("Get child PID:", pid) # 子PID
    print("Parent PID:", os.getpid()) # 父PID
```



代码示例：day5/exit.py*

```

"""
exit.py 进程退出
"""

import os,sys
# 父子进程退出互相不会影响对方
# os._exit(0) # 进程退出
sys.exit("退出进程")
print("exit process")

```

os.getpid()
 功能：获取一个进程的PID值
 返回值：返回当前进程的PID

os.getppid()
 功能：获取父进程的PID号
 返回值：返回父进程PID

os._exit(status)
 功能：结束一个进程
 参数：进程的终止状态

sys.exit([status])
 功能：退出进程
 参数：整数 表示退出状态
 字符串 表示退出时打印内容

孤儿和僵尸

1. 孤儿进程：父进程先于子进程退出，此时子进程成为孤儿进程。

特点：孤儿进程会被系统进程收养，此时系统进程就会成为孤儿进程新的父进程，孤儿进程退出该进程会自动处理。

2. 僵尸进程：子进程先于父进程退出，父进程又没有处理子进程的退出状态，此时子进程就会称为僵尸进程。ps：如果父进程退出则子进程也随即会销毁退出。

特点：僵尸进程虽然结束，但是会存留部分PCB在内存中，大量的僵尸进程会浪费系统的内存资源。

3. 如何避免僵尸进程产生

- 使用wait函数处理子进程退出

```

pid,status = os.wait()
功能：在父进程中阻塞等待处理子进程退出
返回值： pid 退出的子进程的PID
        status 子进程退出状态

```

代码示例：day6/wait.py

```

"""
模拟僵尸进程产生
"""

import os,sys
from time import sleep

```

```

import signal

# 信号处理僵尸进场
signal.signal(signal.SIGCHLD,signal.SIG_IGN)

pid = os.fork()
if pid < 0:
    print("Error")
elif pid == 0:
    sleep(4)
    print("Child PID:",os.getpid())
    sys.exit(2)
else:
    """
    os.wait() 处理僵尸进程
    """

    # pid,status = os.wait()
    # print("PID:",pid)
    # print("STATUS:",status) # 退出状态*256
    while True:      #死循环父进程不退出
        pass

```

```

'''
创建二级子进程
'''

from time import sleep
import os

def fun1():
    for i in range(3):
        sleep(2)
        print("写代码")

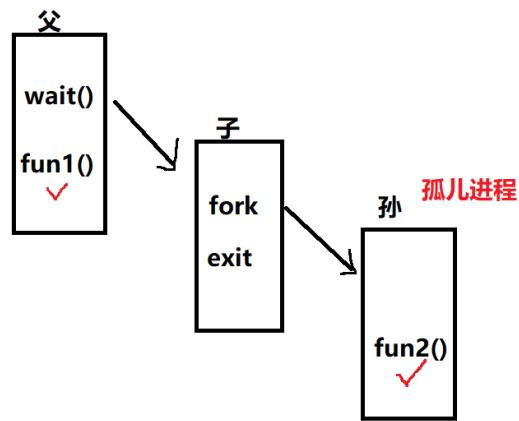
def fun2():
    for i in range(2):
        sleep(4)
        print("测代码")

p1 = os.fork()
if p1 == 0:
    p2 = os.fork()
    if p2 == 0:
        fun2() # 二级子进程做fun2
    else:
        os._exit(0) # 一级子进程退出
else:
    os.wait() # 等待p1子进程退出
    fun1()

```

#对以下图片的说明：

1.父进程创建子进程，子进程再创建一个二级子进程（孙子进程），此时父子进程阻塞等待处理儿子的僵尸进程，儿子进程退出，基本上同一时间父进程处理掉儿子进程，然后父进程去执行fun1(),孙子进程执行fun2()



- 创建二级子进程处理僵尸

代码示例：day6/child.py

- 【1】父进程创建子进程，等待回收子进程
- 【2】子进程创建二级子进程然后退出
- 【3】二级子进程称为孤儿，和原来父进程一同执行事件

- 通过信号处理子进程退出

原理：子进程退出时会发送信号给父进程，如果父进程忽略子进程信号，则系统就会自动处理子进程退出。

方法：使用signal模块在父进程创建子进程前写如下语句：

```
import signal
signal.signal(signal.SIGCHLD, signal.SIG_IGN)
```

特点：非阻塞，不会影响父进程运行。可以处理所有子进程退出

项目：群聊聊天室

功能：类似qq群功能

- 【1】有人进入聊天室需要输入姓名，姓名不能重复
- 【2】有人进入聊天室时，其他人会收到通知：xxx 进入了聊天室
- 【3】一个人发消息，其他人会收到：xxx：XXXXXXXXXXXX
- 【4】有人退出聊天室，则其他人也会收到通知:xxx退出了聊天室
- 【5】扩展功能：服务器可以向所有用户发送公告:管理员消息：XXXXXXXXXX

聊天室思路分析

1. 需求的整理：达到什么效果，如何工作

2. 技术点的分析

* 通信套接字
udp通信

* 数据流方向

c (client) --> s(server)

客户端 请求 --> 服务端处理

信息 客户端-----> 服务端 --> 其他客户端

* 服务端需要存储用户什么信息，怎么存

用户名 地址 这里采用字典进行存储

[(name,address),(name,address)...

{name:address,}

class Person:

def __init__(self,name,address):

self.name = name

self.address = address

* 收发消息

收消息和发消息使用不同的进程

3. 结构设计和协议的设定

* 网络结构搭建

* 进入聊天室

* 聊天

* 退出聊天室功能

* 管理员消息

封装结构: 函数

请求协议: L name

C text...

Q name

响应协议: OK FAIL

4. 逐个功能分析，罗列逻辑流程

* 网络结构搭建

udp网络

* 进入聊天室

客户端: 输入姓名

发送登录请求

得到登录结果

服务端: 接收请求

判断用户是否可以登录

向客户端回复结果

如果该客户端登录了则告知其他人

* 聊天

客户端: 创建新的进程

一个进程循环的接收消息

一个进程循环输入内容发送

服务端: 接收消息 C name xxxxx

转发给其他人 "name: xxxxxx"

* 退出聊天室功能

客户端 quit ctrl-c 退出

发送请求给服务端
结束进程

服务端 接收请求
删除该用户
告知其他人

* 管理员消息

#对象本身就是一个容器，比如类：

```
class Person:
    def __init__(self,name,address):
        self.name=name
        self.address=address
```

```
#chat_server.py
"""
chat room AID 1910
env: python3.6
author: Levi
socket udp & fork
"""

from socket import *
import os,sys

# 服务地址
ADDR = ('0.0.0.0',8888)

# 存储用户 {name:address}
user = {}

# 进入处理
def do_login(s,name,addr):
    if name in user or '管理' in name:
        s.sendto(b'FAIL',addr)
        return
    s.sendto(b'OK',addr)

# 通知其他人
msg = "\n欢迎%s进入聊天室"%name
for i in user:
    s.sendto(msg.encode(),user[i])
# 将其添加到用户字典
user[name] = addr

# 聊天
def do_chat(s,name,text):
    msg = "\n%s : %s"%(name,text)
    for i in user:
        # 刨除本人
        if i != name:
            s.sendto(msg.encode(),user[i])

#退出
```

```

def do_quit(s,name):
    msg = "\n%s退出群聊"%name
    for i in user:
        if i != name:
            # 告知其他人
            s.sendto(msg.encode(),user[i])
        else:
            # 让他本人的接收进程退出
            s.sendto(b'EXIT',user[i])
    del user[name]

# 功能分发函数
def do_request(s):
    while True:
        # 循环接受请求
        data,addr = s.recvfrom(1024)
        tmp = data.decode().split(' ',2)
        # 根据请求类型，选择功能模块去处理
        # L C Q
        if tmp[0] == 'L':
            do_login(s,tmp[1],addr) # 具体函数处理具体功能
        elif tmp[0] == 'C':
            do_chat(s,tmp[1],tmp[2])
        elif tmp[0] == 'Q':
            do_quit(s,tmp[1])

# 启动函数
def main():
    # udp网络服务
    s = socket(AF_INET,SOCK_DGRAM)
    s.bind(ADDR)

    pid = os.fork()
    if pid < 0:
        return
    elif pid == 0:
        # 管理员消息的发送
        while True:
            text = input("管理员消息:")
            msg = "C 管理员 "+text
            # 将消息从子进程发送给父进程
            s.sendto(msg.encode(),ADDR)
    else:
        do_request(s) # 父进程处理请求

if __name__ == '__main__':
    main()

```

```

"""
chat room client
"""

from socket import *
import os,sys

# 服务器地址
ADDR = ('127.0.0.1',8888)

```

```

# 发送消息
def send_msg(s,name):
    while True:
        try:
            text = input("Msg>>")
        except KeyboardInterrupt:
            text = 'quit'
        # 退出
        if text == 'quit':
            msg = "Q "+name
            s.sendto(msg.encode(),ADDR)
            sys.exit("退出群聊")
        msg = "C %s %s"%(name,text)
        s.sendto(msg.encode(),ADDR)

# 接收消息
def recv_msg(s):
    while True:
        try:
            data,addr = s.recvfrom(4096)
        except KeyboardInterrupt:
            sys.exit()
        if data == b'EXIT':
            sys.exit()
        print(data.decode()+"\nMsg>>',end=")

# 客户端启动函数
def main():
    s = socket(AF_INET,SOCK_DGRAM)
    # 进入聊天室部分
    while True:
        name = input("请输入姓名:")
        msg = "L "+name
        s.sendto(msg.encode(),ADDR)
        # 得到结果
        data,addr = s.recvfrom(128)
        if data.decode() == 'OK':
            print("您已进入聊天室")
            break
        else:
            print('进入聊天室失败')

    pid = os.fork()
    if pid < 0:
        return
    elif pid == 0:
        send_msg(s,name) # 子进程发送
    else:
        recv_msg(s) # 父进程接收

if __name__ == '__main__':
    main()

```

multiprocessing 模块创建进程

进程创建方法

代码示例：day6/process1.py

```
"""
Process进程创建方法
"""
import multiprocessing as mp
from time import sleep

a = 1

# 进程函数
def fun():
    print("开始一个进程")
    sleep(2)
    global a
    print("a = ",a)
    a = 10000
    print("子进程函数")

# 实例化进程对象
p = mp.Process(target=fun)
p.start() # 启动进程 执行target绑定函数

sleep(3)
print("父进程那点事")

p.join() # 回收进程
print("a:",a)

"""
p = os.fork()
if p == 0:
    fun()
    os._exit()
else:
    os.wait()
"""
```

代码示例：day6/process2.py

```
"""
创建多个子进程
"""
from multiprocessing import *
from time import sleep
import os

def th1():
    sleep(3)
    print("吃饭")
    print(os.getppid(), '---', os.getpid())

def th2():
    sleep(2)
```

```

print("睡觉")
print(os.getppid(), '---', os.getpid())

def th3():
    sleep(4)
    print("打豆豆")
    print(os.getppid(), '---', os.getpid())

things = [th1, th2, th3]
jobs = [] # 存储进程对象
for th in things:
    p = Process(target=th)
    jobs.append(p)
    p.start()

# 一起回收
for i in jobs:
    i.join()

```

代码示例：day6/process3.py

```

"""
给进程函数传参
"""

from multiprocessing import Process
from time import sleep

# 带参数的进程函数
def worker(sec, name):
    for i in range(3):
        sleep(sec)
        print("I'm %s"%name)
        print("I'm working..")

# 位置传参
# p = Process(target=worker, args=(2, 'Levi'))
p = Process(target=worker, args=(2,),
            kwargs={'name': 'Levi'})
p.start()
p.join()

```

1. 流程特点

- 【1】 将需要子进程执行的事件封装为函数
- 【2】 通过模块的Process类创建进程对象，关联函数
- 【3】 可以通过进程对象设置进程信息及属性
- 【4】 通过进程对象调用start启动进程
- 【5】 通过进程对象调用join回收进程

2. 基本接口使用

Process()
 功能： 创建进程对象
 参数： target 绑定要执行的目标函数
 args 元组，用于给target函数位置传参
 kwargs 字典，给target函数键值传参

p.start()
功能：启动进程

注意:启动进程此时target绑定函数开始执行，该函数作为子进程执行内容，此时进程真正被创建

p.join([timeout])
功能：阻塞等待回收进程
参数：超时时间

注意

- 使用multiprocessing创建进程同样是子进程复制父进程空间代码段，父子进程运行互不影响。
- 子进程只运行target绑定的函数部分，其余内容均是父进程执行内容。
- multiprocessing中父进程往往只用来创建子进程回收子进程，具体事件由子进程完成。
- multiprocessing创建的子进程中无法使用标准输入（input函数）

3. 进程对象属性

代码示例：day7/process_attr.py

```
"""
进程对象属性
"""
from multiprocessing import Process
import time

def tm():
    for i in range(3):
        print(time.ctime())
        time.sleep(2)

p = Process(target=tm,name='show_time')

# 设置daemon属性 在start之前
p.daemon = True

p.start()
print("Name:",p.name)
print("PID:",p.pid)
print("is alive:",p.is_alive())
time.sleep(1)
```

p.name 进程名称

p.pid 对应子进程的PID号

p.is_alive() 查看子进程是否在生命周期

p.daemon 设置父子进程的退出关系

- 如果设置为True则子进程会随父进程的退出而结束
- 要求必须在start()前设置
- 如果daemon设置成True 通常就不会使用 join()

自定义进程类

代码示例：day7/myProcess.py

```
"""
自定义进程类
"""

from multiprocessing import Process
from time import sleep, ctime

class MyProcess(Process):
    def __init__(self, value):
        self.value = value
        super().__init__() # 引用父类方法

    def fun1(self):
        sleep(1)
        print("第一步")

    def fun2(self):
        sleep(0.8)
        print("第二步")

    # 调用start会运行run
    def run(self):
        for i in range(self.value):
            self.fun1()
            self.fun2()

p = MyProcess(2)
p.start() # 运行run方法
p.join()
```

1. 创建步骤

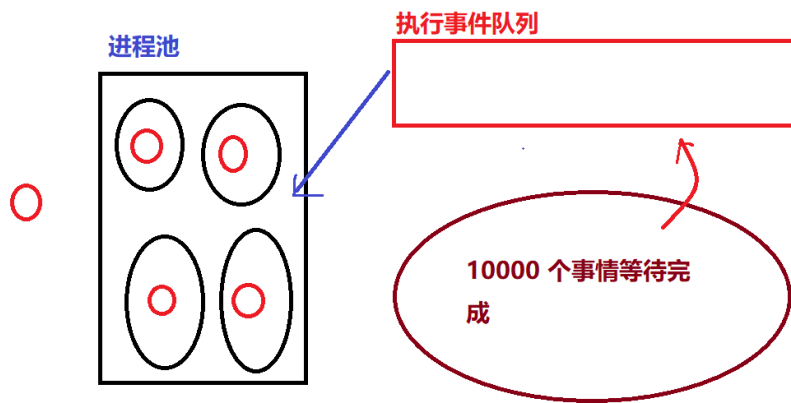
- 【1】 继承Process类
- 【2】 重写 __init__ 方法添加自己的属性，使用super()加载父类属性
- 【3】 重写run()方法

2. 使用方法

- 【1】 实例化对象
- 【2】 调用start自动执行run方法
- 【3】 调用join回收进程

进程池实现

代码示例：day7/pool.py



```
"""
进程池使用演示
"""

from multiprocessing import Pool
from time import sleep, ctime

# 进程池事件函数
def worker(msg):
    sleep(2)
    print(ctime(), '--', msg)
    return 8888

# 创建进程池
pool = Pool(4)

# 向进程池中添加事件
for i in range(10):
    msg = "Tedu %d" % i
    r = pool.apply_async(func=worker,
                        args=(msg,))

# 关闭进程池
pool.close()

# 回收进程池
pool.join()
print(r.get()) # 获取进程池事件函数返回值
```

1. 必要性

【1】进程的创建和销毁过程消耗的资源较多

【2】当任务量众多，每个任务在很短时间内完成时，需要频繁的创建和销毁进程。此时对计算机压力较大

【3】进程池技术很好的解决了以上问题。

2. 原理

创建一定数量的进程来处理事件，事件处理完进程不退出而是继续处理其他事件，直到所有事件全都处理完毕统一销毁。增加进程的重复利用，降低资源消耗。

3. 进程池实现

【1】 创建进程池对象，放入适当的进程

```
from multiprocessing import Pool
```

```
Pool(processes)
```

功能：创建进程池对象

参数：指定进程数量，默认根据系统自动判定

【2】 将事件加入进程池队列执行

```
pool.apply_async(func,args,kwds)
```

功能: 使用进程池执行 func事件

参数： func 事件函数

args 元组 给func按位置传参

kwds 字典 给func按照键值传参

返回值： 返回函数事件对象

【3】 关闭进程池

```
pool.close()
```

功能：关闭进程池

【4】 回收进程池中进程

```
pool.join()
```

功能：回收进程池中进程

进程间通信（IPC）

1. 必要性： 进程间空间独立，资源不共享，此时在需要进程间数据传输时就需要特定的手段进行数据通信。

2. 常用进程间通信方法

管道 消息队列 共享内存 信号 信号量 套接字

消息队列

代码示例：day7/queue_0.py

```
"""
```

```
queue_test.py
```

```
消息队列演示
```

```
"""
```

```
from multiprocessing import Queue,Process
```

```
from time import sleep
```

```
from random import randint,vuj8d
```

```
"""
```

父进程中创建IO，子进程从父进程中获取IO对象，实际上他们操作的是同一个IO，属性相互影响

如果在各自进程中创建IO对象，那么这些IO对象互相没有任何影响

```
"""
```

```

# 创建消息队列
q = Queue(3)

def handle():
    while True:
        try:
            # 获取消息
            x,y = q.get(timeout=8)
        except Exception as e:
            print(e)
            break
        else:
            print("%d+%d=%d"%(x,y,x+y))

def request():
    for i in range(6):
        sleep(randint(1,16))
        x = randint(1,100)
        y = randint(1,100)
        q.put((x,y)) #存入消息

p1 = Process(target=handle)
p2 = Process(target=request)
p1.start()
p2.start()
p1.join()
p2.join()

```

1.通信原理

在内存中建立队列模型，进程通过队列将消息存入，或者从队列取出完成进程间通信。

2. 实现方法

```

from multiprocessing import Queue

q = Queue(maxsize=0)
功能: 创建队列对象
参数: 最多存放消息个数
返回值: 队列对象

q.put(data,[block,timeout])
功能: 向队列存入消息
参数: data 要存入的内容
block 设置是否阻塞 False为非阻塞
timeout 超时检测

q.get([block,timeout])
功能: 从队列取出消息
参数: block 设置是否阻塞 False为非阻塞
timeout 超时检测
返回值: 返回获取到的内容

q.full() 判断队列是否为满
q.empty() 判断队列是否为空
q.qsize() 获取队列中消息个数
q.close() 关闭队列

```

线程编程 (Thread)

线程基本概念

1. 什么是线程

- 【1】 线程被称为轻量级的进程
- 【2】 线程也可以使用计算机多核资源，是多任务编程方式
- 【3】 线程是系统分配内核的最小单元
- 【4】 线程可以理解为进程的分支任务

2. 线程特征

- 【1】 一个进程中可以包含多个线程
- 【2】 线程也是一个运行行为，消耗计算机资源
- 【3】 一个进程中的所有线程共享这个进程的资源
- 【4】 多个线程之间的运行互不影响各自运行
- 【5】 线程的创建和销毁消耗资源远小于进程
- 【6】 各个线程也有自己的ID等特征

threading模块创建线程

【1】 创建线程对象

```
from threading import Thread
```

```
t = Thread()
```

功能：创建线程对象

参数：target 绑定线程函数

args 元组 给线程函数位置传参

kwargs 字典 给线程函数键值传参

【2】 启动线程

```
t.start()
```

【3】 回收线程

```
t.join([timeout])
```

代码示例：day7/thread1.py

```
"""
thread1.py 线程基础使用
"""
import threading
from time import sleep
import os

a = 1

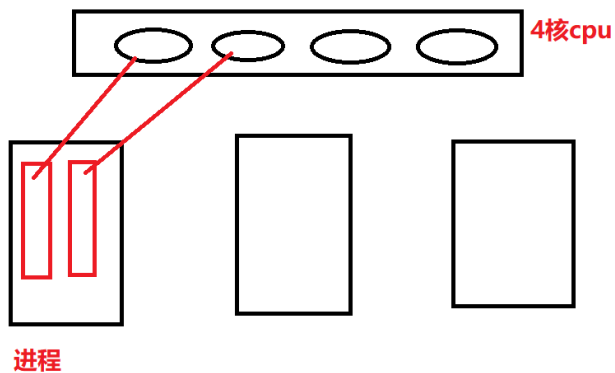
# 线程函数
def music():
    for i in range(3):
        sleep(2)
        print(os.getpid(), "播放黄河大合唱")
```

```

global a
print("a=",a)
a = 10000

# 创建线程对象
t = threading.Thread(target=music)
t.start() # 启动线程
for i in range(4):
    sleep(1)
    print(os.getpid(),"播放葫芦娃")
t.join()
print("a:",a)

```



代码示例：day7/thread2.py*

```

"""
线程创建示例2
"""
from threading import Thread
from time import sleep

# 含有参数的线程函数
def fun(sec,name):
    print("线程函数参数")
    sleep(sec)
    print("%s执行完毕"%name)

# 创建5个线程
jobs = []
for i in range(5):
    t = Thread(target=fun,args=(2,),
               kwargs={'name':'T%d'%i})
    jobs.append(t)
    t.start()
for i in jobs:
    i.join()

```

线程对象属性

t.name 线程名称

t.setName() 设置线程名称

t.getName() 获取线程名称

t.is_alive() 查看线程是否在生命周期

t.daemon 设置主线程和分支线程的退出关系

t.setDaemon() 设置daemon属性值

t.isDaemon() 查看daemon属性值

daemon为True时主线程退出分支线程也退出。要在start前设置，通常不和join一起使用。

代码示例：day8/thread_attr.py

```
"""
线程属性
"""
from threading import Thread
from time import sleep

def fun():
    sleep(15)
    print("线程对象属性")

t = Thread(target=fun,name='Tarena')

# 主线程退出分支线程也退出
# t.setDaemon(True)

t.start()

t.setName("Tedu") # 给线程起名字
print("Name:",t.getName())
print("is alive:",t.is_alive())
```

自定义线程类

1. 创建步骤

【1】 继承Thread类

【2】 重写__init__方法添加自己的属性，使用super()加载父类属性

【3】 重写run()方法

2. 使用方法

【1】 实例化对象

【2】 调用start自动执行run方法

【3】 调用join回收线程

代码示例：day8/myThread.py

```
"""
自定义线程类
"""
from threading import Thread
from time import sleep,ctime
```

```

class MyThread(Thread):
    def __init__(self, target=None, args=(), kwargs={}):
        super().__init__() # 此行不许动
        self.target = target
        self.args = args
        self.kwargs = kwargs

    def run(self):
        self.target(*self.args, **self.kwargs)

# =====
def player(sec, song):
    for i in range(3):
        print("Playing %s:%s"%(song, ctime()))
        sleep(sec)

t = MyThread(target=player, args=(2,),
             kwargs={'song': '凉凉'})
t.start()
t.join()

```

同步互斥

线程间通信方法

1. 通信方法

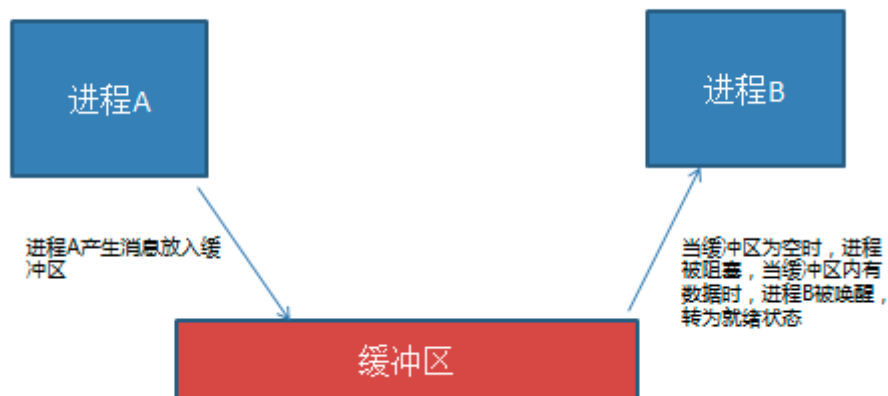
线程间使用全局变量进行通信

2. 共享资源争夺

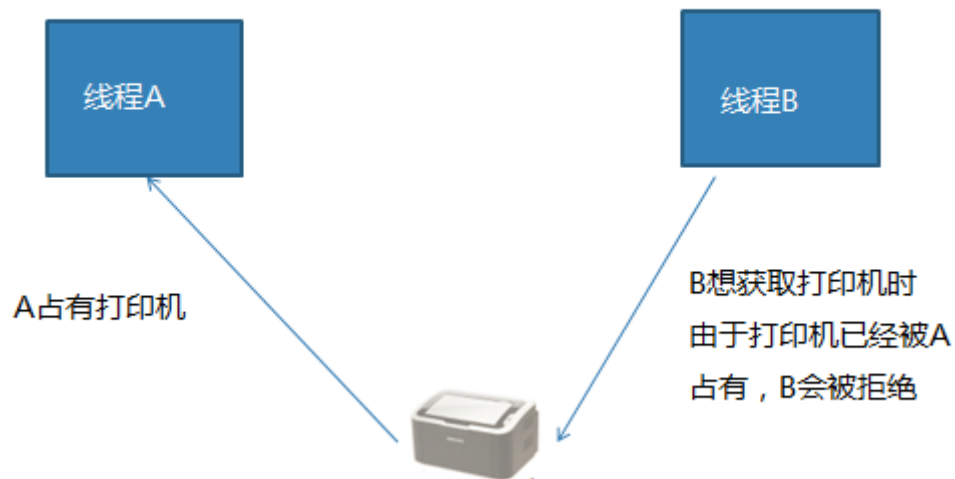
- 共享资源：多个进程或者线程都可以操作的资源称为共享资源。对共享资源的操作代码段称为临界区。
- 影响：对共享资源的无序操作可能会带来数据的混乱，或者操作错误。此时往往需要同步互斥机制协调操作顺序。

3. 同步互斥机制-----添加阻塞

同步：同步是一种协作关系，为完成操作，多进程或者线程间形成一种协调，按照必要的步骤有序执行操作。



互斥：互斥是一种制约关系，当一个进程或者线程占有资源时会进行加锁处理，此时其他进程线程就无法操作该资源，直到解锁后才能操作。



线程同步互斥方法

线程Event

```
from threading import Event

e = Event() 创建线程event对象

e.wait([timeout]) 阻塞等待e被set

e.set() 设置e，使wait结束阻塞

e.clear() 使e回到未被设置状态

e.is_set() 查看当前e是否被设置
```

代码示例：day8/thread_event.py

```
"""
thread_event 互斥方法演示
"""
from threading import Thread, Event

s = None # 用于通信，共享资源
e = Event() # 事件对象

def 杨子荣():
    print("杨子荣前来拜山头")
    global s
    s = "天王盖地虎"
    e.set() # 解除阻塞

t = Thread(target=杨子荣)
t.start()
print("说对口令就是自己人")
e.wait() # 阻塞
if s == '天王盖地虎':
```

```

    print("宝塔镇河妖")
    print("确认过眼神，你是对的人")
else:
    print("打死他。。。。")
t.join()

```

线程锁 Lock

```

from threading import Lock

lock = Lock() 创建锁对象
lock.acquire() 上锁 如果lock已经上锁再调用会阻塞
lock.release() 解锁

with lock: 上锁
...
...
    with代码块结束自动解锁

```

代码示例：day8/thread_lock.py

```

"""
thread lock 互斥方法示例
"""
from threading import Thread, Lock

a = b = 0 # 共享资源
lock = Lock() # 定义锁

def value():
    while True:
        lock.acquire() # 上锁
        if a != b:
            print("a = %d,b = %d"%(a,b))
        lock.release() # 解锁

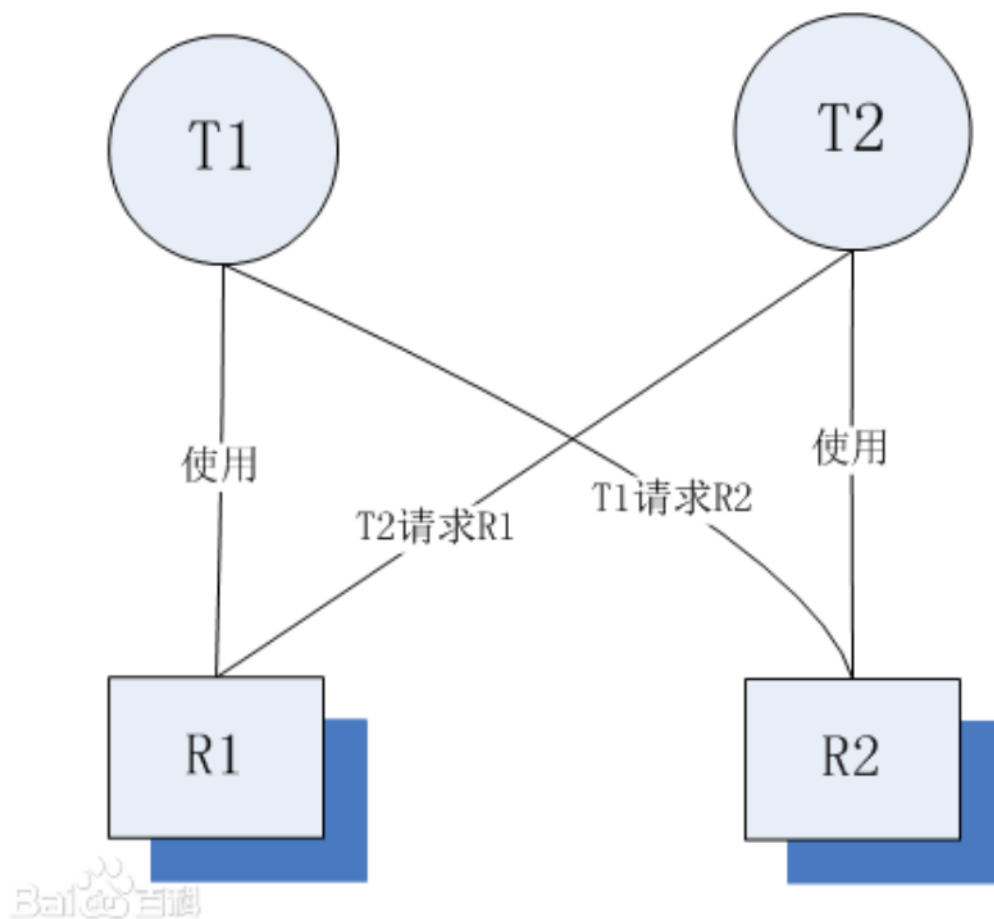
t = Thread(target=value)
t.start()
while True:
    with lock: # 上锁
        a += 1
        b += 1
    # 解锁
t.join()

```

死锁及其处理

1. 定义

死锁是指两个或两个以上的线程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁。



2. 死锁产生条件

死锁发生的必要条件

- 互斥条件：指线程对所分配到的资源进行排它性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求资源，则请求者只能等待，直至占有资源的进程用毕释放。
- 请求和保持条件：指线程已经保持至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求线程阻塞，但又对自己已获得的其它资源保持不放。
- 不剥夺条件：指线程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放,通常CPU内存资源是可以被系统强制调配剥夺的。
- 环路等待条件：指在发生死锁时，必然存在一个线程——资源的环形链，即进程集合 $\{T_0, T_1, T_2, \dots, T_n\}$ 中的 T_0 正在等待一个 T_1 占用的资源； T_1 正在等待 T_2 占用的资源，……， T_n 正在等待已被 T_0 占用的资源。

死锁的产生原因

简单来说造成死锁的原因可以概括成三句话：

- 当前线程拥有其他线程需要的资源
- 当前线程等待其他线程已拥有的资源
- 都不放弃自己拥有的资源

3. 如何避免死锁

死锁是我们非常不愿意看到的一种现象，我们要尽可能避免死锁的情况发生。通过设置某些限制条件，去破坏产生死锁的四个必要条件中的一个或者几个，来预防发生死锁。预防死锁是一种较易实现的方法。但是由于所施加的限制条件往往太严格，可能会导致系统资源利用率。

代码示例: day8/dead_lock.py

```
"""
模拟死锁的产生过程
"""

from time import sleep
from threading import Thread, Lock

# 账户类
class Account:
    def __init__(self, _id, balance, lock):
        self.id = _id # 账户名
        self.balance = balance # 存款
        self.lock = lock # 锁

    # 取钱
    def withdraw(self, amount):
        self.balance -= amount

    # 存钱
    def deposit(self, amount):
        self.balance += amount

    # 查看余额
    def get_balance(self):
        return self.balance

# 创建两个账户
Tom = Account('Tom', 5000, Lock())
Alex = Account('Alex', 8000, Lock())

def transfer(from_, to, amount):
    """
    :param from_: 从该账户转钱
    :param to: 给这个账户转入
    :param amount: 转账金额
    :return: None
    """
    from_.lock.acquire() # from_上锁
    from_.withdraw(amount) # from_减少
    sleep(0.1)
    from_.lock.release() # from_解锁
    to.lock.acquire() # to上锁
    to.deposit(amount) # to钱增加
    to.lock.release() # to解锁
    # from_.lock.release() # from_解锁

t1=Thread(target=transfer, args=(Tom, Alex, 1500))
t2=Thread(target=transfer, args=(Alex, Tom, 3000))
t1.start()
t2.start()
t1.join()
t2.join()

print("Tom:", Tom.get_balance())
print("Alex:", Alex.get_balance())
```

python线程GIL

1. python线程的GIL问题（全局解释器锁）--global interpreter lock

什么是GIL：由于python解释器设计中加入了解释器锁，导致python解释器同一时刻只能解释执行一个线程，大大降低了线程的执行效率。

导致后果：因为遇到阻塞时线程会主动让出解释器，去解释其他线程。所以python多线程在执行多阻塞高延迟IO时可以提升程序效率，其他情况并不能对效率有所提升。

GIL问题建议

- 尽量使用进程完成无阻塞的并发生为
- 不使用c作为解释器（Java C#）

2. 结论：在无阻塞状态下，多线程程序和单线程程序执行效率几乎差不多，甚至还不如单线程效率。但是多进程运行相同内容却可以有明显的效率提升。

进程线程的区别联系

区别联系

1. 两者都是多任务编程方式，都能使用计算机多核资源
2. 进程的创建删除消耗的计算机资源比线程多
3. 进程空间独立，数据互不干扰，有专门通信方法；线程使用全局变量通信
4. 一个进程可以有多个分支线程，两者有包含关系
5. 多个线程共享进程资源，在共享资源操作时往往需要同步互斥处理
6. 进程线程在系统中都有自己的特有属性标志，如ID,代码段，命令集等。

使用场景

1. 任务场景：如果是相对独立的任务模块，可能使用多进程，如果是多个分支共同形成一个整体任务可能用多线程
2. 项目结构：多种编程语言实现不同任务模块，可能是多进程，或者前后端分离应该各自为一个进程。
3. 难易程度：通信难度，数据处理的复杂度来判断用进程间通信还是同步互斥方法。

要求

1. 对进程线程怎么理解/说说进程线程的差异
2. 进程间通信知道哪些，有什么特点
3. 什么是同步互斥，你什么情况下使用，怎么用
4. 给一个情形，说说用进程还是线程，为什么
5. 问一些概念，僵尸进程的处理，GIL问题，进程状态

网络并发通信

常见网络通信模型

1. 循环服务器模型：循环接收客户端请求，处理请求。同一时刻只能处理一个请求，处理完毕后再处理下一个。

优点：实现简单，占用资源少

缺点：无法同时处理多个客户端请求

适用情况：处理的任务可以很快完成，客户端无需长期占用服务端程序。udp比tcp更适合循环。

2. 多进程/线程网络并发模型：每当一个客户端连接服务器，就创建一个新的进程/线程为该客户端服务，客户端退出后再销毁该进程/线程。

优点：能同时满足多个客户端长期占有服务端需求，可以处理各种请求。

缺点：资源消耗较大

适用情况：客户端同时连接量较少，需要处理行为较复杂情况。

3. IO并发模型：利用IO多路复用,异步IO等技术，同时处理多个客户端IO请求。

优点：资源消耗少，能同时高效处理多个IO行为

缺点：只能处理并发产生的IO事件，无法处理cpu计算

适用情况：HTTP请求，网络传输等都是IO行为。

基于fork的多进程网络并发模型

实现步骤

1. 创建监听套接字
2. 等待接收客户端请求
3. 客户端连接创建新的进程处理客户端请求
4. 原进程继续等待其他客户端连接
5. 如果客户端退出，则销毁对应的进程

代码实现: day9/fork_server.py

```
"""
fork_server.py 基于fork的多进程服务
重点代码
"""
from socket import *
import os

# 全局变量
HOST = '0.0.0.0'
PORT = 8888
ADDR = (HOST,PORT)

# 处理客户端请求
def handle(c):
    pass

# 创建tcp套接字
s = socket()
s.setsockopt(SOL_SOCKET,SO_REUSEADDR,1)
s.bind(ADDR)
s.listen(3)

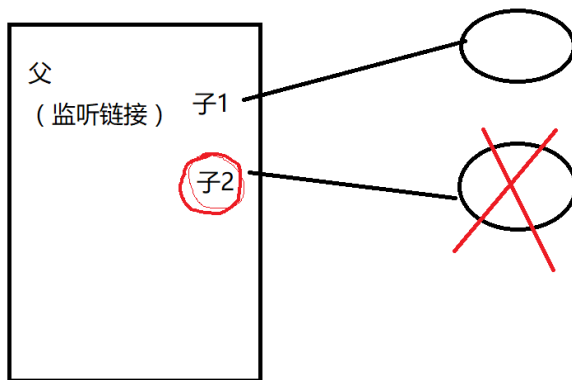
print("Listen the port 8888...")
while True:
    # 循环接受客户端链接
    try:
```

```

c,addr = s.accept()
print("Connect from",addr)
except KeyboardInterrupt:
    os._exit(0)
except Exception as e:
    print(e)
    continue

# 客户端链接处理
pid = os.fork()
if pid == 0:
    # 处理请求
    handle(c) # 处理具体请求
    os._exit(0)
    # 处理完请求子进程结束
else:
    continue # 父进程循环回去继续等待其他客户端

```



基于threading的多线程网络并发

实现步骤

1. 创建监听套接字
2. 循环接收客户端连接请求
3. 当有新的客户端连接创建线程处理客户端请求
4. 主线程继续等待其他客户端连接
5. 当客户端退出，则对应分支线程退出

代码实现: day9/thread_server.py

```

"""
thread_server.py 给予threading多线程并发
重点代码
"""
from socket import *
from threading import Thread

```

```

import sys

# 全局变量
HOST = '0.0.0.0'
PORT = 8888
ADDR = (HOST,PORT)

# 处理客户端请求
def handle(c):
    while True:
        data = c.recv(1024)
        if not data:
            break
        print(data.decode())
        c.send(b'OK')
    c.close()

# 创建tcp套接字
s = socket()
s.setsockopt(SOL_SOCKET,SO_REUSEADDR,1)
s.bind(ADDR)
s.listen(3)

print("Listen the port 8888...")
# 循环接收客户端链接
while True:
    try:
        c,addr = s.accept()
        print("Connect from",addr)
    except KeyboardInterrupt:
        sys.exit('服务器退出')
    except Exception as e:
        print(e)
        continue

# 创建线程处理
t = Thread(target=handle,args=(c,))
t.setDaemon(True) # 主线程退出其他线程也退出
t.start()

```

ftp 文件服务器

1. 功能

- 【1】 分为服务端和客户端，要求可以有多个客户端同时操作。
- 【2】 客户端可以查看服务器文件库中有什么文件。
- 【3】 客户端可以从文件库中下载文件到本地。
- 【4】 客户端可以上传一个本地文件到文件库。
- 【5】 使用print在客户端打印命令输入提示，引导操作

FTP文件服务思路分析

1. 需求分析

2. 技术分析

- * 并发处理：多线程并发
- * 数据传输：tcp传输

3. 功能模块封装

- * 基本功能封装为类

1. 查看内容
2. 上传
3. 下载

4. 协议设置

请求类型 请求内容
列表 LIST
上传 STOR
下载 RETR
退出 QUIT

5. 分模块进行逻辑分析

网络搭建

服务端：tcp多线程并发模型

客户端：链接服务端，打印命令提示界面，通过输入命令决定发送内容

获取文件列表

客户端：发送请求

得到回复

接收文件列表

服务端：收到请求

找到文件库

发送文件列表

下载

客户端：请求 得到回复 下载文件

服务端：接收请求 给出回复 传输文件

上传

退出

总结：

- * 总分结构
- * 理解并发编程带来的优势
- * 面向对象
- * 请求响应模式

代码实现: day9/ftp

```
"""
ftp文件处理服务端
env: python3.6
多线程并发 & socket
"""
from socket import *
from threading import Thread
import sys,os
import time
```

```

# 全局变量
HOST = '0.0.0.0'
PORT = 8080
ADDR = (HOST,PORT)
FTP = "/home/tarena/FTP/" # 文件库位置

# 实现文件传输的具体功能
class FTPServer(Thread):
    def __init__(self,connfd):
        super().__init__()
        self.connfd = connfd

# 处理文件列表发送
def do_list(self):
    # 获取文件列表
    file_list = os.listdir(FTP)
    if not file_list:
        self.connfd.send('文件库为空'.encode()) #19
        return
    else:
        self.connfd.send(b'OK') #19
        time.sleep(0.1)
    # 发送文件列表
    files = '\n'.join(file_list)
    self.connfd.send(files.encode())

# 文件下载
def do_retr(self,filename):
    try:
        f = open(FTP+filename,'rb')
    except Exception:
        self.connfd.send('文件不存在'.encode())
        return
    else:
        self.connfd.send(b'OK')
        time.sleep(0.1)

# 发送文件
while True:
    data = f.read(1024)
    if not data:
        time.sleep(0.1)
        self.connfd.send(b'##')
        break
    self.connfd.send(data)
f.close()

# 处理上传文件
def do_stor(self,filename):
    if os.path.exists(FTP + filename):
        self.connfd.send("文件已存在".encode())
        return
    else:
        self.connfd.send(b'OK')
    # 接收文件
    f = open(FTP+filename,'wb')
    while True:

```

```

        data = self.connfd.recv(1024)
        if data == b'##':
            # 文件发送完
            break
        f.write(data)
        f.close()

    def run(self):
        while True:
            # 接收客户端请求
            data = self.connfd.recv(1024).decode()
            if not data or data == 'QUIT':
                return # 线程结束
            elif data == 'LIST':
                self.do_list()
            elif data[:4] == 'RETR':
                filename = data.split(' ')[-1]
                self.do_retr(filename)
            elif data[:4] == 'STOR':
                filename = data.split(' ')[-1]
                self.do_stor(filename)

# 搭建网络模型
def main():
    # 创建tcp套接字
    s = socket()
    s.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
    s.bind(ADDR)
    s.listen(3)

    print("Listen the port 8080...")
    # 循环接收客户端链接
    while True:
        try:
            c, addr = s.accept()
            print("Connect from", addr)
        except KeyboardInterrupt:
            sys.exit('服务器退出')
        except Exception as e:
            print(e)
            continue

    # 创建线程处理
    t = FTPServer(c)
    t.setDaemon(True) # 主线程退出其他线程也退出
    t.start()

if __name__ == '__main__':
    main()

```

```

"""
ftp 文件服务 客户端
"""
from socket import *

```

```
import sys,time

# 服务器地址
ADDR = ('127.0.0.1',8080)

# 客户端处理类
class FTPClient:
    def __init__(self,sockfd):
        self.sockfd = sockfd

    # 获取文件列表
    def do_list(self):
        self.sockfd.send(b'LIST') # 发送请求 29
        # 等待回复
        data = self.sockfd.recv(128).decode()
        if data == 'OK':
            # 一次接收所有文件名称字符串(解决沾包)
            data = self.sockfd.recv(1024 * 1024 * 10)
            print(data.decode())
        else:
            print(data)

    def do_retr(self,filename):
        # 发送请求
        self.sockfd.send(('RETR '+filename).encode())
        # 等待回复
        data = self.sockfd.recv(128).decode()
        if data == 'OK':
            f = open(filename,'wb')
            # 循环接收
            while True:
                data = self.sockfd.recv(1024)
                if data == b'##':
                    # 文件发送完
                    break
                f.write(data)
            f.close()
        else:
            print(data)

    def do_stor(self,filename):
        try:
            f = open(filename,'rb')
        except Exception:
            print("文件不存在")
            return
        # 提取文件名称
        filename = filename.split('/')[-1]
        # 发送请求
        self.sockfd.send(('STOR ' + filename).encode())
        # 等待回复
        data = self.sockfd.recv(128).decode()
        if data == 'OK':
            while True:
                data = f.read(1024)
                if not data:
                    time.sleep(0.1)
                    self.sockfd.send(b'##')
```

```

        break
    self.sockfd.send(data)
    f.close()
else:
    print(data)

def do_quit(self):
    self.sockfd.send(b'QUIT')
    self.sockfd.close()
    sys.exit('谢谢使用')

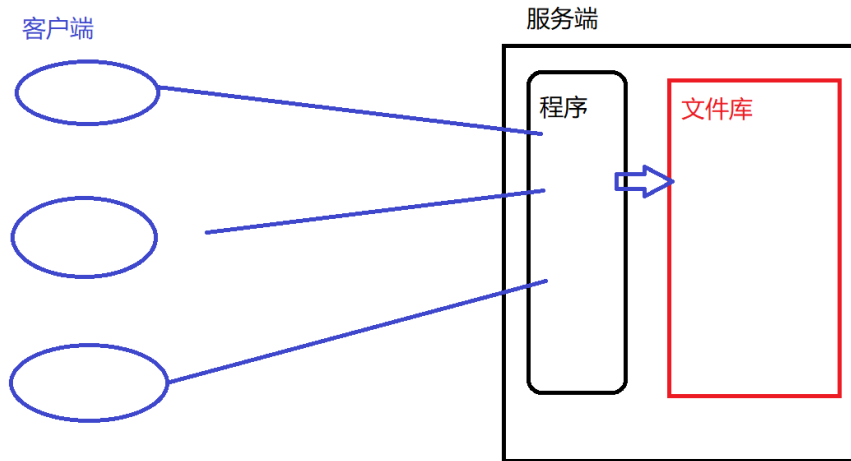
# 网络链接
def main():
    sockfd = socket()
    try:
        sockfd.connect(ADDR)
    except Exception as e:
        print(e)
        return

    ftp = FTPClient(sockfd) # 调用请求功能

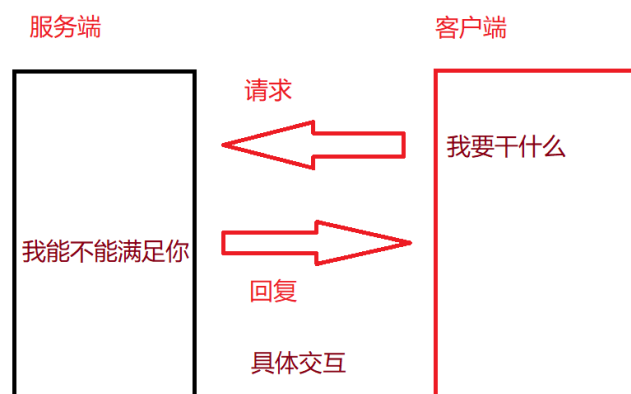
    # 循环的发送请求
    while True:
        print("=====命令选项=====")
        print("*****  LIST  *****")
        print("*****  STOR filename  *****")
        print("*****  RETR filename  *****")
        print("*****  QUIT  *****")
        print("=====")
        cmd = input("命令:")
        if cmd == 'LIST':
            ftp.do_list()
        elif cmd[:4] == 'RETR':
            filename = cmd.split(' ')[-1]
            ftp.do_retr(filename)
        elif cmd[:4] == 'STOR':
            filename = cmd.split(' ')[-1]
            ftp.do_stor(filename)
        elif cmd == 'QUIT':
            ftp.do_quit()
        else:
            print("请输入正确命令! ")

if __name__ == '__main__':
    main()

```



请求应答：



IO并发

IO 分类

IO分类：阻塞IO，非阻塞IO，IO多路复用(面试必问)，异步IO等

阻塞IO

1.定义：在执行IO操作时如果执行条件不满足则阻塞。阻塞IO是IO的默认形态。

2.效率：阻塞IO是效率很低的一种IO。但是由于逻辑简单所以是默认IO行为。

3.阻塞情况：

- 因为某种执行条件没有满足造成的函数阻塞
e.g. accept input recv
- 处理IO的时间较长产生的阻塞状态
e.g. 网络传输，大文件读写

非阻塞IO

1. 定义：通过修改IO属性行为，使原本阻塞的IO变为非阻塞的状态。

- 设置套接字为非阻塞IO

```
sockfd.setblocking(bool)
```

功能：设置套接字为非阻塞IO

参数：默认为True，表示套接字IO阻塞；设置为False则套接字IO变为非阻塞

- 超时检测：设置一个最长阻塞时间，超过该时间后则不再阻塞等待。

```
sockfd.settimeout(sec)
```

功能：设置套接字的超时时间

参数：设置的时间

代码实现: day9/block_io

```
"""
非阻塞IO演示
"""
from socket import *
from time import ctime,sleep

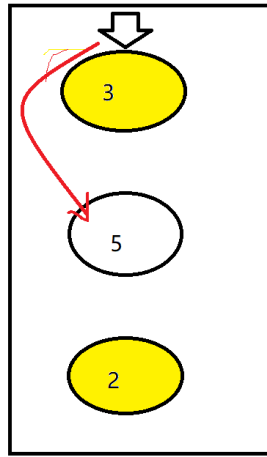
# 日志文件
f = open('run.log','a+')

# tcp套接字
sockfd = socket()
sockfd.bind(('127.0.0.1',8888))
sockfd.listen(3)

# 设置非阻塞
# sockfd.setblocking(False)

# 设置超时时间
sockfd.settimeout(2)

while True:
    sleep(3)
    print("Waiting for connect...")
    try:
        c,addr = sockfd.accept()
    except (BlockingIOError,timeout) as e:
        msg = "%s : %s\n"%(ctime(),e)
        f.write(msg)
        f.flush()
    else:
        data = c.recv(1024)
```



IO多路复用

1. 定义

同时监控多个IO事件，当哪个IO事件准备就绪就执行哪个IO事件。以此形成可以同时处理多个IO的行为，避免一个IO阻塞造成其他IO均无法执行，提高了IO执行效率。

2. 具体方案

select方法： windows linux unix

poll方法： linux unix

epoll方法： linux

select 方法

```
rs, ws, xs=select(rlist, wlist, xlist[, timeout])
```

功能: 监控IO事件，阻塞等待IO发生

参数: rlist 列表 读IO列表，添加等待发生的或者可读的IO事件 -----read/recv

wlist 列表 写IO列表，存放要可以主动处理的或者可写的IO事件-----write/send/connect

xlist 列表 异常IO列表，存放出现异常要处理的IO事件-----在linux系统没用,没有这种处理机制

timeout 超时时间

返回值: rs 列表 rlist中准备就绪的IO

ws 列表 wlist中准备就绪的IO

xs 列表 xlist中准备就绪的IO

代码实现: day10/select_server.py

```
"""
select tcp服务
重点代码
【1】将关注的IO放入对应的监控类别列表
【2】通过select函数进行监控
【3】遍历select返回值列表，确定就绪IO事件
【4】处理发生的IO事件
"""

from socket import *
from select import select

# 创建监听套接字，作为关注的IO
s = socket()
```



```

s.setsockopt(SOL_SOCKET,
             SO_REUSEADDR,1)
s.bind(('0.0.0.0',8888))
s.listen(3)

# 设置关注的IO
rlist = [s] # s的读IO行为
wlist = []
xlist = []

while True:
    # 循环监控s
    rs,ws,xs = select(rlist,wlist,xlist)
    for r in rs:
        if r is s:
            # 又有新的客户端链接
            c, addr = r.accept()
            print("Connect from", addr)
            rlist.append(c)
        else:
            # 某个客户端给我发消息
            data = r.recv(1024).decode()
            if not data:
                # 客户端断开
                rlist.remove(r) # 不再关注这个IO
                r.close()
                continue
            print(data)
            # r.send(b'OK')
            wlist.append(r) # 加入写IO

    for w in ws:
        w.send(b'OK')
        wlist.remove(w)

```

select 实现tcp服务

- 【1】 将关注的IO放入对应的监控类别列表
- 【2】 通过select函数进行监控
- 【3】 遍历select返回值列表，确定就绪IO事件
- 【4】 处理发生的IO事件

注意

wlist中如果存在IO事件，则select立即返回给ws
 处理IO过程中不要出现死循环占有服务端的情况
 IO多路复用消耗资源较少，效率较高

@@扩展: 位运算

定义： 将整数转换为二进制，按二进制位进行运算

运算符号：

&按位与
| 按位或
^ 按位异或-----相同为假,不同为真
<< 左移
>> 右移

e.g. 14 --> 01110
19 --> 10011

14 & 19 = 00010 = 2 —0则0
14 | 19 = 11111 = 31 —1则1
14 ^ 19 = 11101 = 29 相同为0不同为1
14 << 2 = 111000 = 56 向左移动低位补0
14 >> 2 = 11 = 3 向右移动去掉低位

poll方法

p = select.poll()
功能： 创建poll对象
返回值： poll对象

p.register(fd,event)
功能: 注册关注的IO事件
参数: fd 要关注的IO
event 要关注的IO事件类型
常用类型: POLLIN 读IO事件 (rlist) -----1
POLLOUT 写IO事件 (wlist)-----4
POLLERR 异常IO (xlist) -----8
POLLHUP 断开连接
e.g. p.register(sockfd,POLLIN|POLLERR)

p.unregister(fd)
功能: 取消对IO的关注
参数: IO对象或者IO对象的fileno

events = p.poll()
功能: 阻塞等待监控的IO事件发生
返回值: 返回发生的IO
events格式 [(fileno,event),(),...]
每个元组为一个就绪IO, 元组第一项是该IO的fileno, 第二项为该IO就绪的事件类型

代码实现: day10/poll_server.py

```
"""
poll_server.py tcp服务
重点代码

思路: poll() 的返回值不是IO对象
      建立字典 {fileno:io_obj}
"""
from socket import *
from select import *
```

```

# 创建监听套接字，作为关注的IO
s = socket()
s.setsockopt(SOL_SOCKET,
              SO_REUSEADDR,1)
s.bind(('0.0.0.0',8888))
s.listen(3)

# 创建poll对象
p = poll()

# 建立查找字典
fdmap = {s.fileno():s}

# 关注s套接字
p.register(s,POLLIN)

# 循环监控IO发生
while True:
    # 提交监控
    events = p.poll()
    print(events)
    for fd,event in events:
        if fd == s.fileno():
            c,addr = s.accept()
            print("Connect from",addr)
            p.register(c,POLLIN|POLLERR) # 添加新的关注IO
            fdmap[c.fileno()] = c # 注意维护字典与register保持一致
        else:
            # 通过文件描述符取得对象
            data = fdmap[fd].recv(1024).decode()
            print(data)

```

poll_server 步骤

- 【1】 创建套接字
- 【2】 将套接字register
- 【3】 创建查找字典，并维护
- 【4】 循环监控IO发生
- 【5】 处理发生的IO

epoll方法

1. 使用方法： 基本与poll相同

- 生成对象改为 epoll()
- 将所有事件类型改为EPOLL类型

2. epoll特点

- epoll 效率比select poll要高-----不必每次监控向系统层映射IO对象,在应用层可以直接操作就绪IO减少了IO遍历.
- epoll 监控IO数量比select要多
- epoll 的触发方式比poll要多 (EPOLLET边缘触发)

代码实现: day10/epoll_server.py

```

from socket import *
from select import *

```

```

# 创建监听套接字，作为关注的IO
s = socket()
s.setsockopt(SOL_SOCKET,
             SO_REUSEADDR,1)
s.bind(('0.0.0.0',8888))
s.listen(3)

# 创建epoll对象
p = epoll()

# 建立查找字典
fdmap = {s.fileno():s}

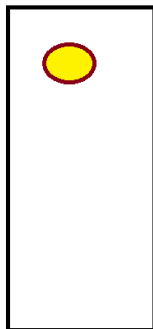
# 关注s套接字
p.register(s,E POLLIN)

# 循环监控IO发生
while True:
    # 提交监控
    events = p.poll()
    print("你有新的IO需要处理哦")
    for fd,event in events:
        if fd == s.fileno():
            c,addr = s.accept()
            print("Connect from",addr)
            # 添加新的关注IO
            p.register(c,E POLLIN|E POLLERR|E POLLET) # 边缘触发
            fdmap[c.fileno()] = c # 注意维护字典与register保持一致
        # elif event & E POLLIN:
        #     # 通过文件描述符取得对象
        #     data = fdmap[fd].recv(1024).decode()
        #     if not data:
        #         p.unregister(fd) # 不再关注
        #         fdmap[fd].close()
        #         del fdmap[fd] # 从字典删除
        #         continue
        #     print(data)
        #     fdmap[fd].send(b'OK')

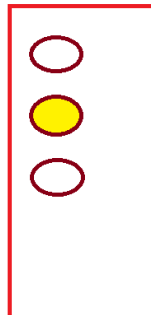
```

EPOLL

应用程序



操作系统



HTTPServer v2.0

1. 主要功能：

- 【1】 接收客户端（浏览器）请求
- 【2】 解析客户端发送的请求
- 【3】 根据请求组织数据内容
- 【4】 将数据内容形成http响应格式返回给浏览器

2. 升级点：

- 【1】 采用IO并发，可以满足多个客户端同时发起请求情况
 - 【2】 通过类接口形式进行功能封装
 - 【3】 做基本的请求解析，根据具体请求返回具体内容，同时处理客户端的非网页请求行为
- day10/http_server.py

```
"""
httpserver 2.0
"""

from socket import *
from select import select

class HTTPServer:
    def __init__(self, host='0.0.0.0', port=80, dir=None):
        self.host = host
        self.port = port
        self.address = (host, port)
        self.dir = dir
        self.rlist = []
        self.wlist = []
        self.xlist = []
        # 直接创建套接字
        self.create_socket()

    # 创建套接字
    def create_socket(self):
        self.sockfd = socket()
        self.sockfd.setsockopt(SOL_SOCKET,
                               SO_REUSEADDR,
                               1)
        self.sockfd.bind(self.address)

    # 启动服务
    def serve_forever(self):
        self.sockfd.listen(3)
        print("Listen the port %d"%self.port)
        # IO多路复用方法监控IO
        self.rlist.append(self.sockfd)
        while True:
            rs, ws, xs = select(self.rlist,
                                self.wlist,
                                self.xlist)
            for r in rs:
                if r is self.sockfd:
                    # 浏览器链接
                    c, addr = r.accept()
                    self.rlist.append(c)
                else:
```

```

        # 处理具体请求
        self.handle(r)

# 处理客户端请求
def handle(self,connfd):
    request = connfd.recv(4096).decode()
    # 客户端断开
    if not request:
        self.rlist.remove(connfd)
        connfd.close()
        return

    # 解析请求，提取请求内容
    request_line = request.split('\n')[0]
    info = request_line.split(' ')[1]
    print(connfd.getpeername(),':',info)

    # 根据请求内容将其分为两类
    if info == '/' or info[-5:] == '.html':
        self.get_html(connfd,info)
    else:
        self.get_data(connfd,info)
    connfd.close()
    self.rlist.remove(connfd)

def get_data(self,connfd,info):
    response = "HTTP/1.1 200 OK\r\n"
    response += "Content-Type:text/html\r\n"
    response += '\r\n'
    response += "<h1>Waiting for httpserver 3.0</h1>"
    connfd.send(response.encode())

# 处理网页
def get_html(self,connfd,info):
    if info == '/':
        # 要主页
        filename = self.dir+'/index.html'
    else:
        # 具体的网页
        filename = self.dir + info
    try:
        fd = open(filename)
    except Exception:
        response = "HTTP/1.1 404 Not Found\r\n"
        response += "Content-Type:text/html\r\n"
        response += '\r\n'
        response += "<h1>Sorry....</h1>"
    else:
        response = "HTTP/1.1 200 OK\r\n"
        response += "Content-Type:text/html\r\n"
        response += '\r\n'
        response += fd.read()
    finally:
        # 将响应发送给浏览器
        connfd.send(response.encode())

if __name__ == '__main__':
    # 通过HTTPServer类快速搭建服务
    # 通过该服务让浏览器访问到我的网页

```

1. 使用流程

2. 需要用户确定的内容

用户决定的参数

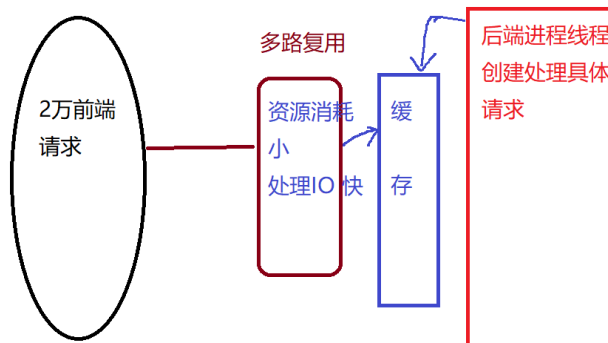
HOST = '0.0.0.0'

PORT = 8000

DIR = './static'

httpd = HTTPServer(HOST,PORT,DIR) # 生成对象

httpd.serve_forever() # 启动服务



if name == 'main':的作用

一个python文件通常有两种使用方法，第一是作为脚本直接执行，第二是 import 到其他的python 脚本中被调用（模块重用）执行。因此 if __name__ == 'main': 的作用就是控制这两种情况执行代码的过程，在 if __name__ == 'main': 下的代码只有在第一种情况下（即文件作为脚本直接执行）才会被执行，而 import 到其他脚本中是不会被执行的。举例说明如下：

■ 直接执行

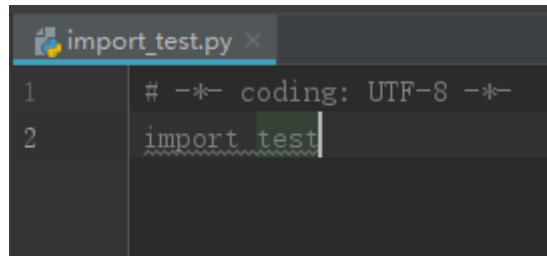
```
test.py x
1  # -*- coding: UTF-8 -*-
2
3  print('this is one')
4
5  if __name__ == '__main__':
6      print('this is two')
```

直接执行 test.py，结果如下图，可以成功 print 两行字符串。即**，if __name__=="__main__": 语句之前和之后的代码都被执行。**

```
C:\Users\TXB\PycharmProjects\TXB-1\venv\Scripts\python.exe C:/Users/TXB/PycharmProjects/TXB-1/test-2/test.py
this is one
this is two
```

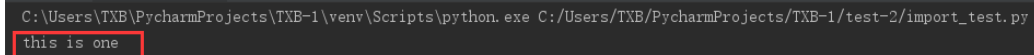
■ import 执行

然后在同一文件夹新建名称为 import_test.py 的脚本，输入如下代码：



```
import_test.py x
1 # -*- coding: UTF-8 -*-
2 import test
```

执行 **import_test.py 脚本**，输出结果如下：



```
C:\Users\TXB\PycharmProjects\TXB-1\venv\Scripts\python.exe C:/Users/TXB/PycharmProjects/TXB-1/test-2/import_test.py
this is one
```

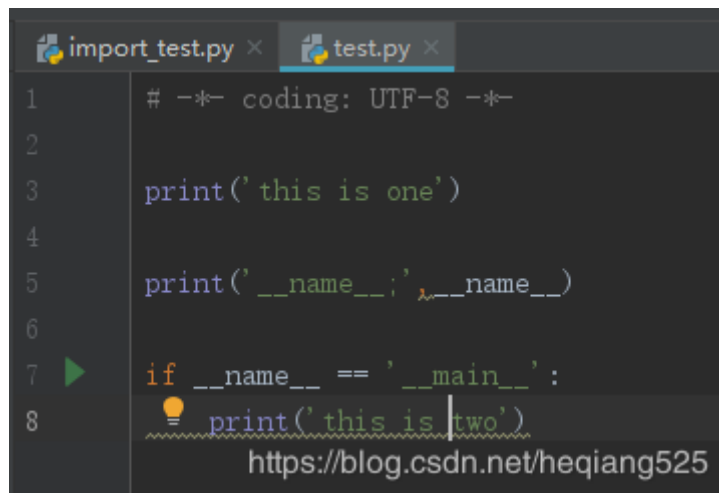
只输出了第一行字符串。即，**if __name__=="__main__": 之前的语句被执行，之后的没有被执行。**

if name == 'main':的运行原理

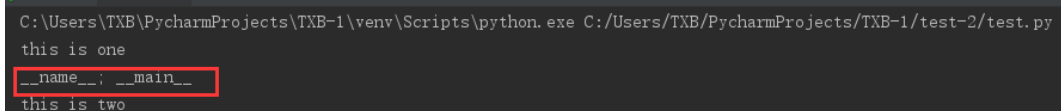
每个python模块（python文件，也就是此处的 test.py 和 import_test.py）都包含内置的变量 name，当该模块被直接执行的时候，name 等于文件名（包含后缀 .py）；如果该模块 import 到其他模块中，则该模块的 name 等于模块名称（不包含后缀.py）。

而“main”始终指当前执行模块的名称（包含后缀.py）。进而当模块被直接执行时，name == 'main' 结果为真。

为了进一步说明，我们在 test.py 脚本的 if name=="main": 之前加入 print(name)，即将 name 打印出来。文件内容和结果如下：



```
import_test.py x test.py x
1 # -*- coding: UTF-8 -*-
2 print('this is one')
3 print('__name__;', __name__)
4 if __name__ == '__main__':
5     print('this is two')
```



```
C:\Users\TXB\PycharmProjects\TXB-1\venv\Scripts\python.exe C:/Users/TXB/PycharmProjects/TXB-1/test-2/test.py
this is one
__name__: __main__
this is two
```

可以看出，此时变量name的值为"main"。

再执行 import_test.py，执行结果如下：


```
import_test.py × test.py ×
1 # -*- coding: UTF-8 -*-
2 |
3 import test
```

```
C:\Users\TXB\PycharmProjects\TXB-1\venv\Scripts\python.exe C:/Users/TXB/PycharmProjects/TXB-1/test-2/import_test.py
this is one
__name__: test
```

此时，test.py中的name变量值为test，不满足name=="main"的条件，因此，无法执行其后的代码。

阶段性总结

IO

文件IO: `open()` `read()` `write()` `close()`
`flush()` `seek()`
`os.getsize()`

网络IO: OSI模型 (tcp/ip模型) 三次握手四次回收 tcp和udp http协议

socket 编程 tcp编程: `socket()` `bind()` `listen()` `accept()` `recv()` `send()` `close()` `connect()`
udp编程: `socket()` `bind()` `recvfrom()` `sendto()` `close()`
套接字属性: `setsockopt()` `getpeername()` `fileno()` `setblocking()` `settimeout()`
struct 模块

并发 进程 `os.fork()`

`Process()` `start()` `join()`
进程池 `Pool()`
进程间通信 `Queue()`

线程 `Thread()` `start()` `join()`

同步互斥(死锁) `Event()` `Lock()`
GIL问题

对比: 什么是进程, 什么是线程

区别联系

进程状态, 僵尸进程

网络通信模型: 多进程多线程并发模型

IO多路复用: `select()`

`poll()` `register()` `unregister()`
`epoll()` ...
知道区别