并发编程

==========================

| Tedu Python 教学部 |

| --- |

| Author：吕泽|

-----------

**１、python线程适合多阻塞的IO并发操作.**

**２、进程和线程的区别、场景。**

[TOC]

## 多任务编程

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。

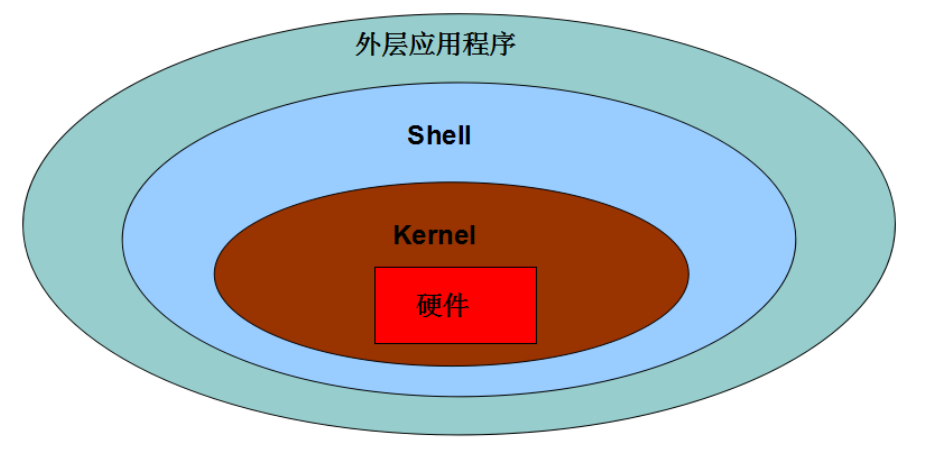
2. 系统中如何产生一个进程

【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

\* 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**

\* 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

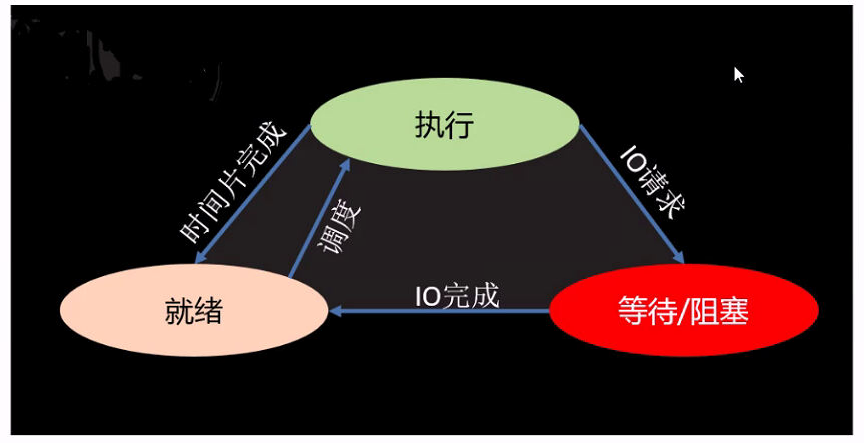
\* 进程状态

- 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

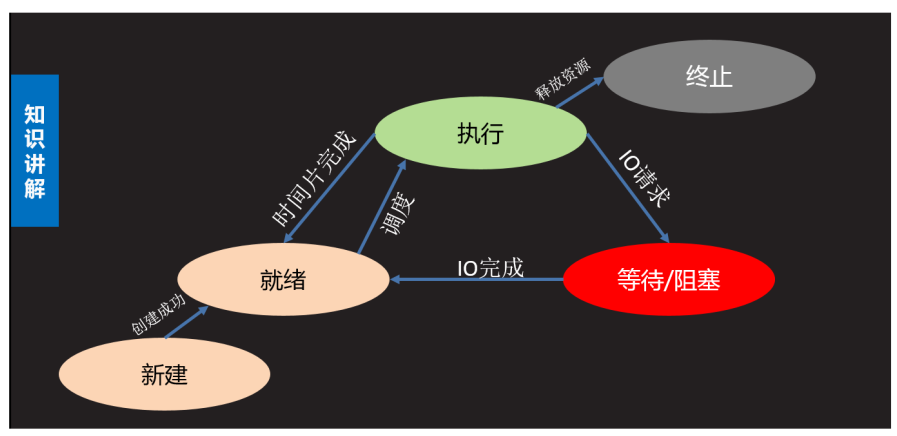
等待态 ： 进程暂时停止运行，让出cpu



- 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

>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

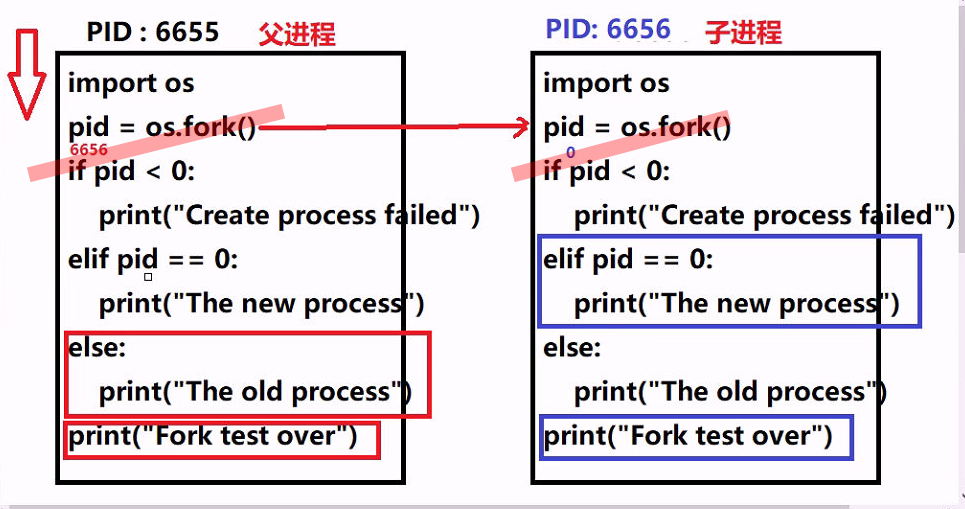
>>1. 什么是进程，进程和程序有什么区别

1)进程是动态的，而程序是静态的。  
  
(2)进程有一定的生命期，而程序是指令的集合，本身无“运动”的含义。没有建立进程的程序不能作为1个独立单位得到[操作系统](https://www.baidu.com/s?wd=操作系统&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)的认可。  
  
(3)1个程序可以对应多个进程，但1个进程只能对应1个程序。进程和程序的关系犹如演出和剧本的关系。

1.程序是永久的，而进程是暂时的;  
2.程序是静态的，而进程是动态的;  
3.程序具有并发性，而进程没有;  
4.进程是竞争计算机资源的基本单位，而程序不是;  
5.进程和程序不是一一对应的，一个程序可以对应多个进程即多个进程也可以执行同一个程序，一个进程可以执行一个或者多个程序。

>>2. 进程有哪些状态，状态之间如何转化

## 基于fork的多进程编程



### fork使用

\*\*\*代码示例：day7/fork.py\*\*\*

*"""*

*fork.py fork进程演示1*

*"""*

**import** os

**from** time **import** sleep

*# 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*

sleep(3)

print(**"The new process"**)

**else**:

*# 父进程执行部分*

sleep(2)

print(**"The old process"**)

print(**"Fork test over"**) *#　父子进程都执行*

\*\*\*代码示例：day7/fork1.py\*\*\*

*"""*

*fork1.py fork进程演示示例*

*"""*

**import** os

**from** time **import** sleep

print(**"============================"**)

a = 1

pid = os.fork()

**if** pid < 0:

print(**"Error"**)

**elif** pid == 0:

print(**"Child process"**)

print(**"a = "**,a) *# 从父进程空间获取的a*

a = 10000 *# 修改自己空间的a*

**else**:

sleep(1)

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，如果创建进程失败返回一个负数，如果成功则在原有进程中返回新进程的PID，在新进程中返回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：day7/get\_pid.py\*\*\*

*"""*

*获取进程PID号*

*"""*

**import** os

**import** time

pid = os.fork()

**if** pid < 0:

print(**"Error"**)

**elif** pid == 0:

time.sleep(1) *# 子进程孤儿*

print(**"Child PID:"**,os.getpid())

print(**"Get parent PID:"**,os.getppid())

**else**:

time.sleep(30)

print(**"Get child PID:"**,pid)

print(**"Parent PID:"**,os.getpid())

\*\*\*代码示例：day7/exit.py\*\*\*

**import** os,sys

*# os.\_exit(1)*

sys.exit(**"进程退出"**)

print(**"exit test"**)

>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。

>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

\*\*\*代码示例：day7/wait.py\*\*\*

*"""*

*wait.py 处理僵尸进程方法*

*"""*

**import** os

**from** time **import** sleep

pid = os.fork()

**if** pid < 0:

print(**"Error"**)

**elif** pid == 0:

print(**"Child process:"**,os.getpid())

sleep(2)

os.\_exit(3) *# 进程退出*

**else**:

pid,status = os.wait() *# 阻塞等待回收子进程*

print(**"pid:"**,pid)

print(**"status:"**,os.WEXITSTATUS(status))

**while True**: *# 让父进程不退出*

**pass**

```

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

```

\* 创建二级子进程处理僵尸

\*\*\*代码示例：day7/child.py\*\*\*

**from** time **import** sleep

**import** os

**def** f1():

**for** i **in** range(3):

sleep(2)

print(**"写代码"**)

**def** f2():

**for** i **in** range(2):

sleep(4)

print(**"测代码"**)

pid = os.fork()

**if** pid == 0: *# 一级子进程*

p = os.fork()

**if** p == 0: *# 二级子进程*

f1()

**else**:

os.wait()

f2() *# 父进程事件*

【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 ：

```python

import signal

signal.signal(signal.SIGCHLD,signal.SIG\_IGN)

```

*# signal 信号方法处理僵尸进程*

**import** os

**import** signal

*# 信号处理僵尸*

signal.signal(signal.SIGCHLD,signal.SIG\_IGN)

*# 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:"**,os.getpid())

**else**:

*# 父进程执行部分*

print(**"Process process"**)

**while True**:

**pass**

>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

### 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 ： xxxxxxxxxxx

【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： xxxxxxxxx

*"""*

*chat room*

*env: python3.6*

*socket udp & fork exc*

*"""*

**from** socket **import** \*

**import** os, sys

*# 全局变量：很多封装模块都要用或者有特定含义的变量*

HOST = **'0.0.0.0'**

PORT = 8888

ADDR = (HOST,PORT)

*# 存储用户* *{name:address}*

user = {}

*# 处理用户登录*

**def** do\_login(s,name,addr):

**if** name **in** user **or '管理员' in** name:

s.sendto(**"用户名存在"**.encode(),addr)

**return**

**else**:

s.sendto(**b'OK'**,addr) *# 可以进入*

*# 通知其他人*

msg = **"\n欢迎** **%s 加入群聊"**%name

**for** i **in** user:

s.sendto(msg.encode(),user[i])

user[name] = addr *# 加入字典*

*# 处理聊天*

**def** do\_chat(s,name,text):

msg = **"\n%s : %s"**%(name,text)

**for** i **in** user:

*# 不发送自己*

**if** i != name:

s.sendto(msg.encode(),user[i])

*# 处理退出*

**def** do\_quit(s,name):

msg = **"\n%s 退出了群聊"**%name

**for** i **in** user:

**if** i == name:

s.sendto(**b'EXIT'**,user[i])

**else**:

s.sendto(msg.encode(),user[i])

**del** user[name] *# 删除用户*

*# 循环获取客户端请求*

**def** do\_request(s):

**while True**:

data,addr = s.recvfrom(1024)

tmp = data.decode().split(**' '**,2)

*# 根据不同的请求类型，执行不同的事件*

**if** tmp[0] == **'L'**:

do\_login(s,tmp[1],addr)

**elif** tmp[0] == **'C'**:

do\_chat(s,tmp[1],tmp[2])

**elif** tmp[0] == **'Q'**:

do\_quit(s,tmp[1])

*# 搭建网络*

**def** main():

*# udp网络*

s = socket(AF\_INET,SOCK\_DGRAM)

s.bind(ADDR)

pid = os.fork()

**if** pid == 0:

*# 管理员消息处理*

**while True**:

msg = input(**"管理员消息:"**)

msg = **"C 管理员** **"**+ msg

s.sendto(msg.encode(),ADDR)

**else**:

do\_request(s) *# 接收客户端请求*

**if** \_\_name\_\_ == **'\_\_main\_\_'**:

main()

*"""*

*chat room 客户端*

*发送请求，展示结果*

*"""*

**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(**">>"**)

**except** KeyboardInterrupt:

text = **'quit'**

**if** text.strip() == **'quit'**:

msg = **"Q "** + name

s.sendto(msg.encode(),ADDR)

sys.exit(**"退出聊天室"**)

msg = **"C %s %s"**%(name,text)

s.sendto(msg.encode(),ADDR)

*# 接收消息*

**def** recv\_msg(s):

**while True**:

data,addr = s.recvfrom(4096)

*# 收到exit接收进程结束*

**if** data.decode() == **'EXIT'**:

sys.exit()

print(data.decode()+**'\n>>'**,end=**''**)

*# 搭建网络*

**def** main():

s = socket(AF\_INET,SOCK\_DGRAM)

*# 进入聊天室*

**while True**:

name = input(**"请输入昵称:"**)

msg = **"L "** + name

s.sendto(msg.encode(),ADDR)

*# 接收反馈*

data,addr = s.recvfrom(128)

**if** data == **b'OK'**:

print(**"您已进入聊天室"**)

**break**

**else**:

print(data.decode())

*# 已经进入聊天室*

pid = os.fork()

**if** pid < 0:

sys.exit(**"Error!"**)

**elif** pid == 0:

send\_msg(s,name)

**else**:

recv\_msg(s)

**if** \_\_name\_\_ == **'\_\_main\_\_'**:

main()

## multiprocessing 模块创建进程

### 进程创建方法

\*\*\*代码示例：day8/process1.py\*\*\*

*"""*

*multiprocessing 模块创建进程*

*1. 编写进程执行函数*

*2. 创建进程对象*

*3. 启动进程*

*4. 回收进程*

*"""*

**import** multiprocessing **as** mp

**from** time **import** sleep

a = 1

*# 进程函数*

**def** fun():

print(**"开始一个进程"**)

sleep(3)

**global** a

print(**"a:"**,a)

a = 10000

print(**"进程结束"**)

*# 创建进程对象*

p = mp.Process(target = fun)

p.start() *# 启动进程*

sleep(2)

print(**"父进程也干点事"**)

p.join() *# 回收进程*

print(**"======================="**)

print(**"a = "**,a)

**'''**

**p = os.fork()**

**if pid == 0:**

**fun()**

**else:**

**os.wait()**

**'''**

\*\*\*代码示例：day8/process2.py\*\*\*

*"""*

*同时创建多个子进程*

*"""*

**from** multiprocessing **import** Process

**from** time **import** sleep

**import** os

**def** th1():

sleep(3)

print(**"吃饭"**)

print(os.getppid(),**'--'**,os.getpid())

**def** th2():

*# a = input() # 不能使用标准输入*

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 = []

**for** th **in** things:

p = Process(target = th)

jobs.append(p) *# 列表存储一下进程对象*

p.start()

**for** i **in** jobs:

i.join()

\*\*\*代码示例：day8/process3.py\*\*\*

*"""*

*进程函数传参*

*"""*

**from** multiprocessing **import** Process

**from** time **import** \*

*# 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..."**)

*# 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

```python

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

```

```python

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

```python

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

3. 进程对象属性

\*\*\*代码示例：day8/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 = **"Tedu"**)

p.daemon = **True** *# 父进程退出子进程随之退出*

p.start()

time.sleep(2)

print(**"Name:"**,p.name)

print(**"PID:"**,p.pid) *# 进程PID*

print(**"is alive:"**,p.is\_alive())

>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：day8/myProcess.py\*\*\*

*"""*

*自定义进程类*

*"""*

**from** multiprocessing **import** Process

*# 自定义类*

**class** MyProcess(Process):

**def** \_\_init\_\_(self,value):

self.value = value

super().\_\_init\_\_() *# 加载父类init*

**def** f1(self):

print(**"步骤1"**)

**def** f2(self):

print(**"步骤2"**)

**def** run(self):

self.f1()

self.f2()

p = MyProcess(2)

p.start() *# 执行run，作为一个子进程执行*

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回收线程

### 进程池实现

\*\*\*代码示例：day8/pool.py\*\*\*

*"""*

*进程池使用示例*

*"""*

**from** multiprocessing **import** Pool

**from** time **import** sleep,ctime

*# 进程池执行事件*

**def** worker(msg):

sleep(2)

print(ctime(),**'--'**,msg)

*# 创建进程池*

pool = Pool(4)

*# 添加时间*

**for** i **in** range(10):

msg = **"Tedu %d"**%i

pool.apply\_async(func=worker,args=(msg,))

*# 关闭进程池*

pool.close()

*# 回收进程池*

pool.join()

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

```python

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

```python

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

```python

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

```

练习：

*"""*

*求100000以内所有质数之和*

*请分别用单进程，4进程，10进程完成*

*记录每种情况的执行时间 通过装饰器记录时间*

*"""*

*“””timeit*模块”””

*#* 获取函数执行时间

*import time*

*def timeit(f):*

*def wrapper(\*args,\*\*kwargs):*

*start\_time = time.time()*

*res = f(\*args,\*\*kwargs)*

*end\_time = time.time()*

*print("%s*函数执行时长*:%.6f"%(f.\_\_name\_\_,end\_time-start\_time))*

*return res*

*return wrapper*

*‘’’*执行模块’’’

**from** multiprocessing **import** Process

**from** timeit **import** timeit

*# 判断一个数是否是质数*

**def** isPrime(n):

**if** n <= 1:

**return False**

**for** i **in** range(2,int(n)):

**if** n % i == 0:

**return False**

**return True**

*# 单进程* *no\_multi\_process函数执行时长:26.169546*

*# @timeit*

*# def no\_multi\_process():*

*# prime = []*

*# for i in range(1,100001):*

*# if isPrime(i):*

*# prime.append(i)*

*# print(sum(prime))*

*#*

*# no\_multi\_process()*

*# 自定义进程类*

**class** Prime(Process):

**def** \_\_init\_\_(self,prime,begin,end):

super().\_\_init\_\_()

self.prime = prime *# 装质数的列表*

self.begin = begin *# 开始位置*

self.end = end *# 结束位置*

**def** run(self):

**for** i **in** range(self.begin,self.end):

**if** isPrime(i):

self.prime.append(i)

sum(self.prime)

*#4个进程* *use\_4\_process函数执行时长:15.554102*

*# @timeit*

*# def use\_4\_process():*

*# prime = []*

*# jobs = []*

*# for i in range(1,100001,25000):*

*# p = Prime(prime,i,i+25000)*

*# jobs.append(p)*

*# p.start()*

*# [i.join() for i in jobs] # 回收进程*

*#*

*# use\_4\_process()*

*# 10 进程* *use\_10\_process函数执行时长:13.969181*

@timeit

**def** use\_10\_process():

prime = []

jobs = []

**for** i **in** range(1,100001,10000):

p = Prime(prime,i,i+10000)

jobs.append(p)

p.start()

[i.join() **for** i **in** jobs] *# 回收进程*

use\_10\_process()

*"""*

*练习*:*使用进程池拷贝一个目录下所有的文件(普通文件)，实时打印拷贝的百分比进度。*

*"""*

**from** multiprocessing **import** Pool,Queue

**import** os

q = Queue() *# 消息队列*

*# 拷贝文件*

**def** copy\_file(file,old\_dir,new\_dir):

fr = open(old\_dir+file,**'rb'**)

fw = open(new\_dir+file,**'wb'**)

**while True**:

data = fr.read(1024)

**if not** data:

**break**

n = fw.write(data) *# 写入的字节数*

q.put(n) *# 放到消息队列*

*# 创建进程池*

**def** main():

*# 确定好源目录和备份目录*

path = **"/home/tarena/"**

dir = input(**"输入要拷贝的文件目录:"**)

old\_dir = path + dir + **'/'** *#源目录绝对路径*

new\_dir = path + dir+**"-备份/"**

os.mkdir(new\_dir)

*# 要拷贝的文件列表*

file\_list = os.listdir(old\_dir)

*# 获取文件总大小*

total\_size = 0

**for** file **in** file\_list:

total\_size += os.path.getsize(old\_dir+file)

print(**"总共大小：%d M"**%(total\_size/1024//1024))

pool = Pool(4)

*# 添加进程池事件*

**for** file **in** file\_list:

pool.apply\_async(copy\_file,args=(file,

old\_dir,

new\_dir))

pool.close()

copy\_size = 0 *# 已经拷贝的大小*

**while** copy\_size < total\_size:

copy\_size += q.get() *# 获取队列数量值*

print(**"拷贝了%.1f%%"**%(copy\_size/total\_size\*100))

pool.join()

**if** \_\_name\_\_ == **'\_\_main\_\_'**:

main()

## 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

### 管道通信(Pipe)

\*\*\*代码示例：day9/pipe.py\*\*\*

*"""*

*pipe.py 管道通信*

*注意： 管道对象需在父进程中创建，子进程从父进程中获取*

*"""*

**from** multiprocessing **import** Process,Pipe

*# 创建管道*

*# False单向管道* *fd1->recv fd2->send*

*# 不要在一个进程中同时使用fd1 fd2*

fd1,fd2 = Pipe(**False**)

**def** app1():

print(**"启动app1,请登录，（可以使用app2）"**)

print(**"向app2发请求"**)

fd1.send(**"app1需要：用户名，头像"**) *# 写管道*

data = fd1.recv()

print(**"Oh yeah"**,data)

**def** app2():

data = fd2.recv() *# 读管道*

print(**"app1请求:"**,data)

fd2.send({**'name'**:**'Han'**,**'image'**:**'有'**})

p1 = Process(target=app1)

p2 = Process(target=app2)

p1.start()

p2.start()

p1.join()

p2.join()

1. 通信原理

>在内存中开辟管道空间，生成管道操作对象，多个进程使用同一个管道对象进行读写即可实现通信

2. 实现方法

```python

from multiprocessing import Pipe

fd1,fd2 = Pipe(duplex = True)

功能: 创建管道

参数：默认表示双向管道

如果为False 表示单向管道

返回值：表示管道两端的读写对象

如果是双向管道均可读写

如果是单向管道fd1只读 fd2只写

fd.recv()

功能 ： 从管道获取内容

返回值：获取到的数据

fd.send(data)

功能： 向管道写入内容

参数： 要写入的数据

```

### 消息队列

\*\*\*代码示例：day9/queue\_0.py\*\*\*

*"""*

*queue\_0.py 消息队列演示*

*注意* *: 通过一个对象操作队列，满足先进先出原则*

*"""*

**from** multiprocessing **import** Queue,Process

**from** time **import** sleep

**from** random **import** randint

*# 创建消息队列*

q = Queue(5)

*# 请求进程*

**def** request():

**for** i **in** range(10):

sleep(0.5)

t = (randint(1,100),randint(1,100))

q.put(t)

print(**"====================="**)

*# 数据处理进程*

**def** handle():

**while True**:

sleep(2)

x,y = q.get()

print(**"数据处理结果** **x + y="**,x + y)

p1 = Process(target=request)

p2 = Process(target=handle)

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

```python

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

```

### 共享内存

\*\*\*代码示例：day9/value.py\*\*\*

*"""*

*value.py 开辟共享内存*

*注意: 共享内存只能有一个值*

*"""*

**from** multiprocessing **import** Process,Value

**import** time

**from** random **import** randint

*# 创建共享内存*

money = Value(**'i'**,5000)

*# 操作内存*

**def** man():

**for** i **in** range(30):

time.sleep(0.2)

money.value += randint(1,1000)

**def** girl():

**for** i **in** range(30):

time.sleep(0.15)

money.value -= randint(100,800)

p1 = Process(target=man)

p2 = Process(target=girl)

p1.start()

p2.start()

p1.join()

p2.join()

print(**"一个月余额:"**,money.value)

\*\*\*代码示例：day9/array.py\*\*\*

*"""*

*array.py 存放一组数据*

*"""*

**from** multiprocessing **import** Process,Array

*# 共享内存，初始[1,2,3,4,5]*

*# shm = Array('i',[1,2,3,4,5])*

*# shm = Array('i',4) #共享内存，初始[0,0,0,0]*

shm = Array(**'c'**,**b'Hello'**)

**def** fun():

*# 迭代获取共享内存值*

**for** i **in** shm:

print(i)

shm[0] = **b'h'**

p = Process(target = fun)

p.start()

p.join()

*# for i in shm:*

*# print(i)*

print(shm.value) *# 用于打印共享内存字节串*

1. 通信原理：在内中开辟一块空间，进程可以写入内容和读取内容完成通信，但是每次写入内容会覆盖之前内容。

2. 实现方法

![](img/6\_ctype.png)

```python

from multiprocessing import Value,Array

obj = Value(ctype,data)

功能 ： 开辟共享内存

参数 ： ctype 表示共享内存空间类型 'i'-整数 'f'-浮点数 'c'-字节串

data 共享内存空间初始数据

返回值：共享内存对象

obj.value 对该属性的修改查看即对共享内存读写

# 数组 可遍历

obj = Array(ctype,data)

功能： 开辟共享内存空间

参数： ctype 表示共享内存数据类型

data 整数则表示开辟空间的大小，其他数据类型表示开辟空间存放的初始化数据

返回值：共享内存对象

Array共享内存读写： 通过遍历obj可以得到每个值，直接可以通过索引序号修改任意值。

\* 可以使用obj.value直接打印共享内存中的字节串

```

### 信号量（信号灯集）

\*\*\*代码示例：day9/sem.py\*\*\*

*"""*

*sem.py 信号量演示*

*注意: 信号量相当于资源，多个进程对数量进行控制*

*"""*

**from** multiprocessing **import** Process,Semaphore

**from** time **import** sleep

**import** os

*# 创建信号量*

sem = Semaphore(3)

*# 任务函数*

**def** handle():

sem.acquire() *# 执行任务必须消耗一个信号量*

print(**"开始执行任务："**,os.getpid())

sleep(2)

print(**"执行任务结束："**, os.getpid())

sem.release() *# 增加一个信号量*

**for** i **in** range(5):

p = Process(target = handle)

p.start()

1. 通信原理

>给定一个数量对多个进程可见。多个进程都可以操作该数量增减，并根据数量值决定自己的行为。

2. 实现方法

```python

from multiprocessing import Semaphore

# 信号量

sem = Semaphore(num)

功能 ： 创建信号量对象

参数 ： 信号量的初始值

返回值 ： 信号量对象

sem.acquire() 将信号量减1 当信号量为0时阻塞

sem.release() 将信号量加1

sem.get\_value() 获取信号量数量

```

## 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

\*\*\*代码示例：day9/thread1.py\*\*\*

*"""*

*thread1.py 线程基础示例*

*步骤：* *1. 封装线程函数*

*2. 创建线程对象*

*3. 启动线程*

*4. 回收线程*

*"""*

**import** threading

**from** time **import** sleep

**import** os

a = 1

*# 线程函数*

**def** music():

**global** a

print(**"a = "**,a)

a = 10000

**for** i **in** range(3):

sleep(2)

print(os.getpid(),**"播放:其实你太美"**)

*# 创建线程对象*

t = threading.Thread(target=music)

t.start() *# 启动线程*

*# 主线程执行*

**for** i **in** range(4):

sleep(1)

print(os.getpid(),**"播放：大碗宽面"**)

t.join() *# 回收线程*

print(**"a:"**,a)

\*\*\*代码示例：day9/thread2.py\*\*\*

*"""*

*thread2.py 线程函数参数示例*

*"""*

**from** threading **import** Thread

**from** time **import** sleep

*# 含有参数的线程函数*

**def** fun(sec,name):

print(**"%s线程开始执行"**%name)

sleep(sec)

print(**"%s执行完毕"**%name)

*# 创建多个线程*

jobs = []

**for** i **in** range(5):

t = Thread(target=fun,args=(2,),

kwargs={**'name'**:**'T%d'**%i})

jobs.append(t) *# 存储线程对象*

t.start()

**for** i **in** jobs:

i.join()

【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])

```

### 线程对象属性

\*\*\*代码示例：day9/thread\_attr.py\*\*\*

*"""*

*thread\_attr.py*

*线程属性演示*

*"""*

**from** threading **import** Thread

**from** time **import** sleep

**def** fun():

sleep(3)

print(**"进程属性设置"**)

t = Thread(target=fun,name=**"AID"**)

t.setDaemon(**True**) *# 主线程退出分支线程也退出*

t.start()

t.setName(**'Tarena'**)

print(**"Name:"**,t.getName())

print(**"is alive:"**,t.is\_alive())

print(**"is daemon:"**,t.isDaemon())

>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。

### 自定义线程类

\*\*\*代码示例：day9/myThread.py\*\*\*

**from** threading **import** Thread

**from** time **import** sleep,ctime

*# 完成这个类*

**class** MyClass(Thread):

*# 该方法可以修改，第8行不能传参数*

**def** \_\_init\_\_(self,target=**None**,args=(),kwargs={}):

super().\_\_init\_\_()

self.target = target

self.args = args

self.kwargs = kwargs

**def** run(self):

self.target(\*self.args,\*\*self.kwargs)

*# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*# 测试函数，该函数名称，参数都不确定。本函数只提供测试*

**def** player(sec,song):

**for** i **in** range(3):

print(**"Playing %s : %s"**%(song,ctime()))

sleep(sec)

t = MyClass(target = player,args=(3,),

kwargs={**'song'**:**'凉凉'**})

t.start()

t.join()

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

## 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

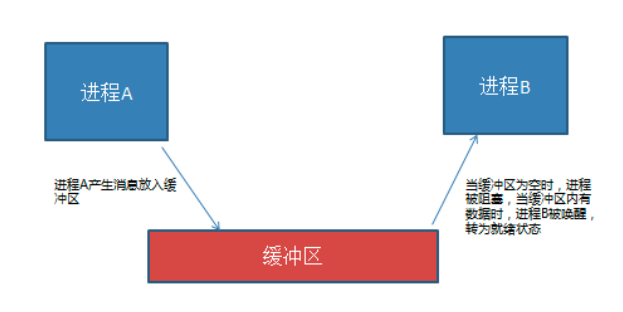
2. 共享资源争夺

\* 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。

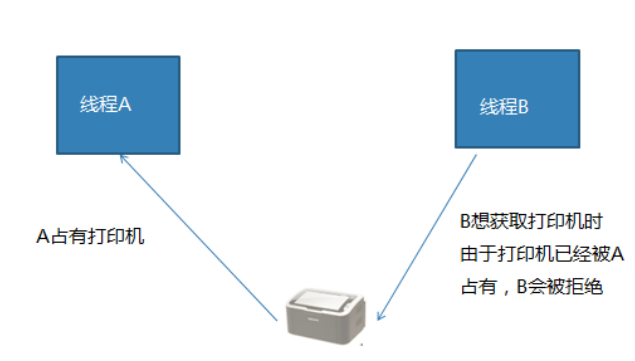
\* 影响 ： 对共享资源的无序操作可能会带来数据的混乱，或者操作错误。此时往往需要同步互斥机制协调操作顺序。

3. 同步互斥机制

>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

\*\*\*代码示例：day10/thread\_event.py\*\*\*

*"""*

*event 线程互斥方法演示*

*"""*

**from** threading **import** Thread,Event

s = **None**

e = Event()

**def** 杨子荣():

print(**"杨子荣前来拜山头"**)

**global** s

s = **"天王盖地虎"**

e.set()

t = Thread(target = 杨子荣)

t.start()

print(**"说对口令就是自己人"**)

e.wait() *# 等待e被set*

**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()

```python

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

```

#### 线程锁 Lock

\*\*\*代码示例：day10/thread\_lock.py\*\*\*

```python

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

```

### 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。

![](img/死锁.jpg)

2. 死锁产生条件

\*\*\*代码示例: day10/dead\_lock.py\*\*\*

*"""*

*死锁情形模拟*

*"""*

**from** threading **import** Thread,Lock

**from** time **import** sleep

*# 账户类*

**class** Account:

**def** \_\_init\_\_(self,\_id,balance,lock):

self.id = \_id *# id*

self.balance = balance *# 存款*

self.lock = lock *# 维护锁*

*# 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

*# 生成两个账户*

Tom = Account(**'Tom'**,12000,Lock())

Abby = Account(**'Abby'**,9000,Lock())

*# 转账 账户金额变动需要先上锁*

**def** transfer(from\_,to,amount):

**if** from\_.lock.acquire():

from\_.withdraw(amount) *# from\_账户钱减少*

sleep(0.1)

**if** to.lock.acquire():

to.deposit(amount) *# to 存钱*

to.lock.release()

from\_.lock.release()

print(**"%s给%s转了%d元"**%(from\_.id,to.id,amount))

t1 = Thread(target=transfer,args=(Tom,Abby,4000))

t2 = Thread(target=transfer,args=(Abby,Tom,1500))

t1.start()

t2.start()

t1.join()

t2.join()

print(**"Tom:"**,Tom.get\_balance())

print(**"Abby:"**,Abby.get\_balance())

练习

*"""*

*模拟开启多个线程，在多资源情况下共同下载一个文件*

*"""*

**import** os

**from** threading **import** Thread,Lock

**from** time **import** sleep

urls = [**"/home/tarena/桌面/"**,

**"/home/tarena/文档/"**,

**"/home/tarena/音乐/"**,

**"/home/tarena/下载/"**,

**"/home/tarena/视频/"**,

**"/home/tarena/图片/"**,

**"/home/tarena/模板/"**,

]

lock = Lock() *# 锁*

filename = input(**"要下载的文件:"**)

explorer = []

**for** i **in** urls:

*# 判断资源库路径中文件是否存在*

**if** os.path.exists(i+filename):

*# 存文件路径*

explorer.append(i+filename)

num = len(explorer) *# 获取有多少资源*

**if** num == 0:

print(**"没有资源"**)

os.\_exit(0)

size = os.path.getsize(explorer[0])

block\_size = size // num + 1

*# 共享资源*

fd = open(filename,**'wb'**) *# 下载的文件*

*# 下载文件*

**def** load(path,num):

f = open(path,**'rb'**) *# 从资源中读取内容*

seek\_types = block\_size \* num

f.seek(seek\_types)

size = block\_size

lock.acquire() *# 上锁*

fd.seek(block\_size \* num)

**while True**:

*# sleep(0.1)*

**if** size < 1024:

data = f.read(size)

fd.write(data)

**break**

**else**:

data = f.read(1024)

fd.write(data)

size -= 1024

lock.release()

n = 0 *# 给每个线程分配的是第几块*

jobs = []

**for** path **in** explorer:

t = Thread(target = load,args=(path,n))

jobs.append(t)

t.start()

n += 1

**for** i **in** jobs:

i.join()

*'''*

*分别用单进程 执行10次* *count io,记录时间*

*no thread CPU: 7.518454074859619*

*no thread IO: 4.570638418197632*

*用十个线程每个执行1次* *count io记录时间*

*multi Thread CPU: 6.699898958206177*

*multi Thread IO: 5.505040168762207*

*用十个进程每个执行1次* *count io记录时间*

*multi Process CPU: 3.598656415939331*

*multi Process io: 2.3045859336853027*

*'''*

**"""**

**测试用例**

**"""**

*# 计算*

**def** count(x,y):

c = 0

**while** c < 7000000:

x += 1

y += 1

c += 1

*# io*

**def** io():

write()

read()

**def** write():

f = open(**'test'**,**'w'**)

**for** i **in** range(1700000):

f.write(**"Hello world\n"**)

f.close()

**def** read():

f = open(**'test'**)

lines = f.readlines()

f.close()

*#no thread*

*# from thread\_test import \**

*# import time*

*#*

*# tm = time.time()*

*# for i in range(10):*

*# # count(1,1)*

*# io()*

*# print("no thread IO:",time.time() - tm)*

*# multi Thread*

**from** thread\_test **import** \*

**import** threading

**import** time

jobs = []

tm = time.time()

**for** i **in** range(10):

t = threading.Thread(target=io)

jobs.append(t)

t.start()

**for** i **in** jobs:

i.join()

print(**"multi Thread CPU:"**,time.time() - tm)

*# multi Process*

**from** thread\_test **import** \*

**import** multiprocessing **as** mp

**import** time

jobs = []

tm = time.time()

**for** i **in** range(10):

p = mp.Process(target=count,args=(1,1))

jobs.append(p)

p.start()

**for** i **in** jobs:

i.join()

print(**"multi Process CPU:"**,time.time() - tm)

*"""*

*死锁情形模拟*

*"""*

**from** threading **import** Thread,Lock

**from** time **import** sleep

*# 账户类*

**class** Account:

**def** \_\_init\_\_(self,\_id,balance,lock):

self.id = \_id *# id*

self.balance = balance *# 存款*

self.lock = lock *# 维护锁*

*# 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

*# 生成两个账户*

Tom = Account(**'Tom'**,12000,Lock())

Abby = Account(**'Abby'**,9000,Lock())

*# 转账 账户金额变动需要先上锁*

**def** transfer(from\_,to,amount):

**if** from\_.lock.acquire():

from\_.withdraw(amount) *# from\_账户钱减少*

sleep(0.1)

**if** to.lock.acquire():

to.deposit(amount) *# to 存钱*

to.lock.release()

from\_.lock.release()

print(**"%s给%s转了%d元"**%(from\_.id,to.id,amount))

t1 = Thread(target=transfer,args=(Tom,Abby,4000))

t2 = Thread(target=transfer,args=(Abby,Tom,1500))

t1.start()

t2.start()

t1.join()

t2.join()

print(**"Tom:"**,Tom.get\_balance())

print(**"Abby:"**,Abby.get\_balance())

>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行调配剥夺的。

>>\* 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{T0，T1，T2，···，Tn}中的T0正在等待一个T1占用的资源；T1正在等待T2占用的资源，……，Tn正在等待已被T0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。

## python线程GIL

1. python线程的GIL问题 （全局解释器锁）

>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行为

>\* 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

## 并发网络通信模型

### 常见网络模型

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，低频，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，客户端退出时再销毁该进程/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

\*\*\*代码实现: day10/fork\_server.py\*\*\*

*"""*

*fork\_server.py 基于fork多进程并发*

*重点代码*

*1. 创建监听套接字*

*2. 循环等待客户端连接*

*3. 客户端连接创建新的进程为客户端服务*

*4. 原进程继续等待其他客户端连接*

*5. 客户端退出，对应的进程也销毁*

*"""*

**from** socket **import** \*

**import** os

**import** signal

*# 全局变量*

HOST = **'127.0.0.1'**

PORT = 8888

ADDR = (HOST,PORT)

*# 客户端处理*

**def** handle(c):

**while True**:

data = c.recv(1024)

**if not** data:

**break**

print(data.decode())

c.send(**b'OK'**)

c.close()

*# 创建套接字*

s = socket()

s.setsockopt(SOL\_SOCKET,SO\_REUSEADDR,1)

s.bind(ADDR)

s.listen(5)

*# 处理僵尸进程*

signal.signal(signal.SIGCHLD,signal.SIG\_IGN)

print(**"Listen the port 8888..."**)

**while True**:

*# 循环等待客户端连接*

**try**:

c,addr = s.accept()

print(**"Connect from"**,addr)

**except** KeyboardInterrupt **asetsockopts** e:

os.\_exit(0)

**except** Exception **as** e:

print(e)

**continue**

*# 创建进程*

pid = os.fork()

**if** pid == 0:

s.close()

handle(c) *# 和客户端交互*

os.\_exit(0) *# 客户端结束后，子进程结束*

**else**:

c.close()

#### 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

### 基于threading的多线程网络并发

\*\*\*代码实现: day10/thread\_server.py\*\*\*

*"""*

*thread\_server.py 多线程并发模型*

*重点代码*

*创建监听套接字*

*循环接收客户端连接请求*

*当有新的客户端连接创建线程处理客户端请求*

*主线程继续等待其他客户端连接*

*当客户端退出，则对应分支线程退出*

*"""*

**from** socket **import** \*

**from** threading **import** Thread

**import** sys

*# 全局变量*

HOST = **'127.0.0.1'**

PORT = 8888

ADDR = (HOST,PORT)

*# 客户端处理*

**def** handle(c):

**while True**:

data = c.recv(1024)

**if not** data:

**break**

print(data.decode())

c.send(**b'OK'**)

c.close()

*# 创建套接字*

s = socket()

s.setsockopt(SOL\_SOCKET,SO\_REUSEADDR,1)

s.bind(ADDR)

s.listen(5)

print(**"Listen the port 8888..."**)

**while True**:

*# 循环等待客户端连接*

**try**:

c,addr = s.accept()

print(**"Connect from"**,addr)

**except** KeyboardInterrupt **as** e:

sys.exit(**"服务器退出"**)

**except** Exception **as** e:

print(e)

**continue**

*# 创建线程*

t = Thread(target=handle,args=(c,))

t.setDaemon(**True**)

t.start()

#### 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

### ftp 文件服务器

\*\*\*代码实现: day11/ftp\*\*\*

*'''*

*ftp 文件服务*

*1. 技术点*

*\* tcp传输*

*\* 数据并发： 多线程并发*

*2. 结构设计*

*类封装*

*3. 功能模块*

*\* 网络构建*

*\* 查看文件列表*

*\* 下载文件*

*\* 上传文件*

*4. 协议确定*

*查看文件列表 ：* *L*

*下载文件：* *G*

*上传文件: P*

*退出：* *Q*

*'''*

**"""**

**ftp 文件服务器 服务端**

**多进程/多线程并发** **socket**

**"""**

**from** socket **import** \*

**from** threading **import** Thread

**import** sys,os

**from** time **import** sleep

HOST = **'0.0.0.0'**

PORT = 8888

ADDR = (HOST,PORT)

FTP = **"/home/tarena/FTP/"** *# 文件库位置*

*# 实现具体功能*

**class** FtpServer(Thread):

*"""*

*查看文件列表，上传，下载，退出*

*"""*

**def** \_\_init\_\_(self,connfd):

self.connfd = connfd

super().\_\_init\_\_()

**def** do\_list(self):

files = os.listdir(FTP)

**if not** files:

self.connfd.send(**"充值VIP,百万图书任你选"**.encode())

**return**

**else**:

self.connfd.send(**b'OK'**)

sleep(0.1)

filelist = **""**

**for** file **in** files:

**if** file[0] != **'.' and** \

os.path.isfile(FTP+file):

filelist += file + **'\n'**

self.connfd.send(filelist.encode())

*# for file in files:*

*# # 不是隐藏文件并且是普通文件*

*# if file[0] != '.' and \*

*# os.path.isfile(FTP+file):*

*# sleep(0.1)*

*# self.connfd.send(file.encode())*

*# sleep(0.1)*

*# self.connfd.send(b'##')*

*# 下载文件*

**def** do\_get(self,filename):

**try**:

f = open(FTP+filename,**'rb'**)

**except** Exception:

*# 文件不存在*

self.connfd.send(**"vip可以下载"**.encode())

**return**

**else**:

self.connfd.send(**b'OK'**)

sleep(0.1)

*# 发送文件*

**while True**:

data = f.read(1024)

**if not** data:

sleep(0.1)

self.connfd.send(**b'##'**)

**break**

self.connfd.send(data)

f.close()

*# 上传*

**def** do\_put(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'##'**:

**break**

f.write(data)

f.close()

*# 分配任务*

**def** run(self):

**while True**:

*# 客户端请求*

data=self.connfd.recv(1024).decode()

*# 判断请求类型*

**if not** data **or** data == **'Q'**:

**return** *# 线程结束*

**elif** data == **'L'**:

self.do\_list()

**elif** data[0] == **'G'**:

filename = data.split(**' '**)[-1]

self.do\_get(filename)

**elif** data[0] == **'P'**:

filename = data.split(**' '**)[-1]

self.do\_put(filename)

*# 搭建网络并发模型*

**def** main():

*# 创建套接字*

s = socket()

s.setsockopt(SOL\_SOCKET, SO\_REUSEADDR, 1)

s.bind(ADDR)

s.listen(5)

print(**"Listen the port 8888..."**)

**while True**:

*# 循环等待客户端连接*

**try**:

c, addr = s.accept()

print(**"Connect from"**, addr)

**except** KeyboardInterrupt **as** e:

sys.exit(**"服务器退出"**)

**except** Exception **as** e:

print(e)

**continue**

*# 创建线程*

t = FtpServer(c)

t.setDaemon(**True**)

t.start()

**if** \_\_name\_\_ == **'\_\_main\_\_'**:

main()

*"""*

*ftp 文件服务 ，客户端*

*"""*

**from** socket **import** \*

**import** sys

**from** time **import** sleep

*# 服务器地址*

ADDR = (**'127.0.0.1'**,8888)

*# 具体功能实现*

**class** FtpClient:

*"""*

*实现具体功能请求*

*"""*

**def** \_\_init\_\_(self,sockfd):

self.sockfd = sockfd

*# 获取文件列表*

**def** do\_list(self):

self.sockfd.send(**b'L'**) *# 发送请求*

*# 等待回去，确认是否有文件列表*

data = self.sockfd.recv(128).decode()

**if** data == **'OK'**:

data = self.sockfd.recv(1024 \* 1024).decode()

print(data)

*# while True:*

*# data = self.sockfd.recv(128).decode()*

*# if data == '##':*

*# break*

*# print(data)*

**else**:

print(data) *# 不可以查看的原因*

*# 退出*

**def** do\_quit(self):

self.sockfd.send(**b'Q'**)

self.sockfd.close()

sys.exit(**"谢谢使用"**)

**def** do\_get(self,filename):

*# 发送请求*

self.sockfd.send((**'G '**+filename).encode())

*# 等待回复*

data = self.sockfd.recv(128).decode()

*# 接收文件*

**if** data == **'OK'**:

f = open(filename,**'wb'**)

*# 循环接收内容，写入文件*

**while True**:

data = self.sockfd.recv(1024)

**if** data == **b'##'**:

**break**

f.write(data)

f.close()

**else**:

print(data)

**def** do\_put(self,filename):

**try**:

f = open(filename,**'rb'**)

**except** Exception:

print(**"文件不存在"**)

**return**

*# 获取真正的文件名*

filename = filename.split(**'/'**)[-1]

*# 发送请求*

self.sockfd.send((**'P '**+filename).encode())

*# 等待回复*

data = self.sockfd.recv(128).decode()

**if** data == **'OK'**:

**while True**:

data = f.read(1024)

**if not** data:

sleep(0.1)

self.sockfd.send(**b'##'**)

**break**

self.sockfd.send(data)

f.close()

**else**:

print(data)

*# 网络大家，和终端输入命令选项*

**def** main():

sockfd = socket()

**try**:

sockfd.connect(ADDR)

**except** Exception **as** e:

print(e)

**return**

*# 实例化对象*

ftp = FtpClient(sockfd)

*# 循环发起请求*

**while True**:

print(**"\n=========Command=============="**)

print(**"\*\*\*\*\* list \*\*\*\*\*"**)

print(**"\*\*\*\*\* get file \*\*\*\*\*"**)

print(**"\*\*\*\*\* put file \*\*\*\*\*"**)

print(**"\*\*\*\*\* quit \*\*\*\*\*"**)

print(**"==============================="**)

cmd = input(**"Command:"**)

**if** cmd.strip() == **'list'**:

ftp.do\_list()

**elif** cmd.strip() == **'quit'**:

ftp.do\_quit()

**elif** cmd[:3] == **'get'**:

filename = cmd.split(**' '**)[-1]

ftp.do\_get(filename)

**elif** cmd[:3] == **'put'**:

filename = cmd.split(**' '**)[-1]

ftp.do\_put(filename)

**else**:

print(**"请输入正确命令!"**)

**if** \_\_name\_\_ == **'\_\_main\_\_'**:

main()

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

## IO并发

### IO 分类

>IO分类：阻塞IO ，非阻塞IO，IO多路复用，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

\*\*\*代码实现: day11/block\_io\*\*\*

*"""*

*block\_io.py 非阻塞io演示*

*"""*

**from** socket **import** \*

**from** time **import** ctime,sleep

f = open(**'log.txt'**,**'a'**) *# 日志文件*

s = socket()

s.bind((**'0.0.0.0'**,8888))

s.listen(5)

*# 设置套接字非阻塞*

*# s.setblocking(False)*

*# 设置超时时间*

s.settimeout(3)

**while True**:

**try**:

c,addr = s.accept()

print(**"Connect from"**,addr)

**except** (BlockingIOError,timeout) **as** e:

sleep(2)

f.write(ctime()+**':'**+str(e)+**'\n'**)

**else**:

data = c.recv(1024)

print(data)

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

### 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 优点：跨平台性好；缺点：效率一般，最多监控１０２４个ＩＯ

>poll方法： linux unix 　优点：监控ＩＯ数量没有限制　缺点：跨平台性一般，效率一般

>epoll方法： linux 　优点：监控ＩＯ没有限制，效率高　缺点：跨平台性差

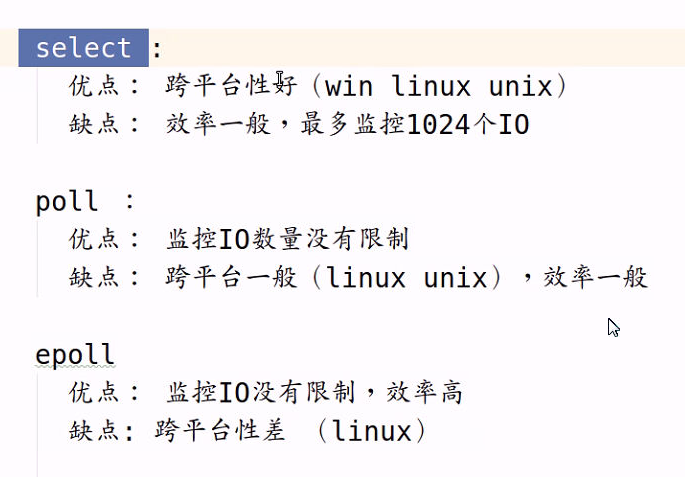
**ｓｅlect,poll:每次监控IO都需要将应用层关注的ＩＯ映射给内核处理**

**内核感知有ＩＯ准备就绪告知应用层，应用层需要再次轮寻找到就绪的ＩＯ**

**epoll ：直接将关注的ＩＯ放到内核空间进行监控，不必每次都从应用层映射给内容**

**内核会直接提供就绪的ＩＯ给应用层处理**

**epoll提供了ＥＰＯＬＬＥＴ的边缘触发方法，可以忽略不想处理的ＩＯ**



#### select 方法

\*\*\*代码实现: day11/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IO*

s = socket()

s.setsockopt(SOL\_SOCKET,SO\_REUSEADDR,1)

s.bind((**'0.0.0.0'**,8888))

s.listen(3)

*# 设置关注列表*

rlist = [s] *# 等待客户端连接*

wlist = []

xlist = []

*# 监控IO发生*

**while True**:

rs,ws,xs = select(rlist,wlist,xlist)

**for** r **in** rs:

**if** r **is** s:

*# 有客户端连接*

c,addr = r.accept()

print(**"Connect from"**,addr)

rlist.append(c) *# 连接对象加入监控*

**else**:

data = r.recv(1024).decode()

**if not** data:

rlist.remove(r) *# 取消对它关注*

r.close()

**continue**

print(data)

*# r.send(b'OK')*

wlist.append(r)

**for** w **in** ws:

w.send(**b'OK'**)

wlist.remove(w) *# 从写监控中移除*

*"""*

*select 演示*

*"""*

**from** select **import** select

**from** socket **import** \*

s = socket()

s.bind((**'127.0.0.1'**,8888))

s.listen(3)

f = open(**'12.txt'**)

print(**" 监控io "**)

rs,ws,xs = select([s],[s],[s])

print(**"rlist:"**,rs)

print(**"wlist:"**,ws)

print(**"xlist:"**,xs)

**from** socket **import** \*

*# 创建tcp套接字*

sockfd = socket() *# 默认值*

*# 连接服务器*

server\_addr = (**'127.0.0.1'**,8888) *# 服务器地址*

sockfd.connect(server\_addr)

*# 先发后收*

**while True**:

msg = input(**"Msg:"**)

**if not** msg:

**break**

sockfd.send(msg.encode()) *#字节串*

data = sockfd.recv(1024)

print(**"From server:"**,data.decode())

sockfd.close()

```python

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 存放关注的等待发生的IO事件

wlist 列表 存放关注的要主动处理的IO事件

xlist 列表 存放关注的出现异常要处理的IO

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

```

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

> << 左移

> >> 右移

```python

e.g. 14 --> 01110

19 --> 10011

**14 & 19 = 00010 = 2 一0则0**

**14 | 19 = 11111 = 31 一1则1**

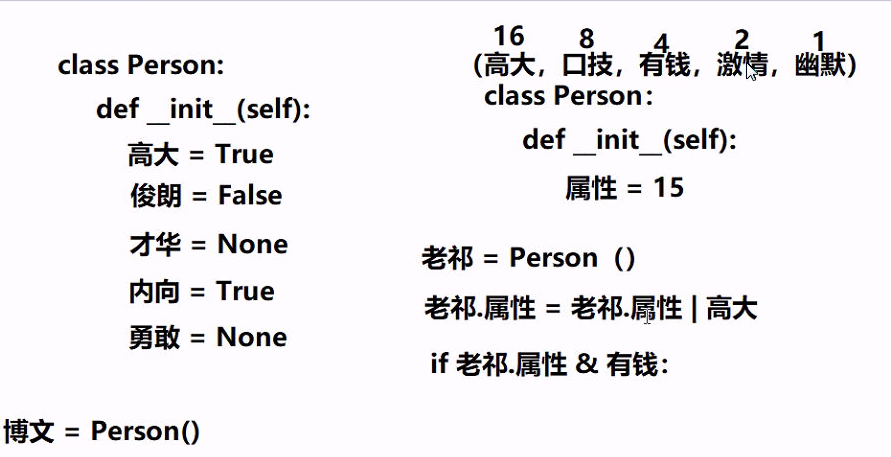
**14 ^ 19 = 11101 = 29 相同为0不同为1**

**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

\*\*\*代码实现: day12/poll\_server.py\*\*\*

*"""*

*poll 方法实现IO多路服用*

*重点代码*

*【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*

*"""*

**from** socket **import** \*

**from** select **import** \*

*# 创建套接字作为关注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，通过IO对象的fileno找到对象*

*# 字典内容与关注IO保持一直{fileno:io\_obj}*

fdmap = {s.fileno():s}

*# 关注s*

p.register(s,POLLIN|POLLERR)

*# 循环监控IO的发生*

**while True**:

events = p.poll()

print(events)

**for** fd,event **in** events:

**if** fd == s.fileno():

c,addr = fdmap[fd].accept()

print(**"Connect from"**,addr)

*# 添加新的关注对象，同时维护字典*

p.register(c,POLLIN)

fdmap[c.fileno()] = c

**elif** event & POLLIN:

data = fdmap[fd].recv(1024).decode()

**if not** data:

*# 客户端退出*

p.unregister(fd) *# 取消关注*

fdmap[fd].close()

**del** fdmap[fd]

**continue**

print(data)

p.register(fd,POLLOUT)

*# fdmap[fd].send(b'OK')*

**elif** event & POLLOUT:

fdmap[fd].send(**b'OK'**)

p.register(fd, POLLIN)

```python

p = select.poll()

功能 ： 创建poll对象

返回值： poll对象

```

```python

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）

POLLOUT 写IO事件 (wlist)

POLLERR 异常IO （xlist）

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

```

```python

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

```

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

\*\*\*代码实现: day12/epoll\_server.py\*\*\*

*"""*

*epoll 方法实现IO多路服用*

*重点代码*

*【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*

*"""*

**from** socket **import** \*

**from** select **import** \*

*# 创建套接字作为关注IO*

s = socket()

s.setsockopt(SOL\_SOCKET,SO\_REUSEADDR,1)

s.bind((**'127.0.0.1'**,8888))

s.listen(3)

*# 创建epoll对象*

ep = epoll()

*# 建立查找字典，通过IO对象的fileno找到对象*

*# 字典内容与关注IO保持一直{fileno:io\_obj}*

fdmap = {s.fileno():s}

*# 关注s*

ep.register(s,EPOLLIN|EPOLLERR)

*# 循环监控IO的发生*

**while True**:

events = ep.poll()

print(**"你有新的IO需要处理哦："**,events)

**for** fd,event **in** events:

**if** fd == s.fileno():

c,addr = fdmap[fd].accept()

print(**"Connect from"**,addr)

*# 添加新的关注对象，同时维护字典*

ep.register(c,EPOLLIN|EPOLLET) *# 边缘触发*

fdmap[c.fileno()] = c

*# elif event & EPOLLIN:*

*# data = fdmap[fd].recv(1024).decode()*

*# if not data:*

*# # 客户端退出*

*# ep.unregister(fd) # 取消关注*

*# fdmap[fd].close()*

*# del fdmap[fd]*

*# continue*

*# print(data)*

*# ep.unregister(fd)*

*# ep.register(fd, POLLOUT)*

*# elif event & POLLOUT:*

*# fdmap[fd].send(b'OK')*

*# ep.unregister(fd)*

*# ep.register(fd, POLLIN)*

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

\* 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）

### 协程技术

#### 基础概念

1. 定义：纤程，微线程。是允许在不同入口点不同位置暂停或开始的计算机程序，简单来说，**协程就是可以暂停执行的函数。**

2. 协程原理 ： 记录一个函数的上下文，协程调度切换时会将记录的上下文保存，在切换回来时进行调取，恢复原有的执行内容，以便从上一次执行位置继续执行。

3. 协程优缺点

>优点

>>1. 协程完成多任务占用计算资源很少

>>2. 由于协程的多任务切换在应用层完成，因此切换开销少

>>3. 协程为单线程程序，无需进行共享资源同步互斥处理

>缺点

>

>> 协程的本质是一个单线程，无法利用计算机多核资源

--------------------------------

####扩展延伸@标准库协程的实现

python3.5以后，使用标准库asyncio和async/await 语法来编写并发代码。asyncio库通过对异步IO行为的支持完成python的协程。虽然官方说asyncio是未来的开发方向，但是由于其生态不够丰富，大量的客户端不支持awaitable需要自己去封装，所以在使用上存在缺陷。更多时候只能使用已有的异步库（asyncio等），功能有限

------------------------------

#### 第三方协程模

1. greenlet模块

\*\*\*示例代码: day12/greenlet\_0.py\*\*\*

*"""*

*协程行为展示*

*"""*

**from** greenlet **import** greenlet

**def** fun1():

print(**"执行fun1"**)

gr2.switch()

print(**"结束fun1"**)

gr2.switch()

**def** fun2():

print(**"执行fun2"**)

gr1.switch()

print(**"结束fun2"**)

*# 将函数变为协成函数*

gr1 = greenlet(fun1)

gr2 = greenlet(fun2)

gr1.switch()

\* 安装 ： sudo pip3 install greenlet

\* 函数

```python

greenlet.greenlet(func)

功能：创建协程对象

参数：协程函数

g.switch()

功能：选择要执行的协程函数

```

2. gevent模块

\*\*\*示例代码: day12/gevent\_test.py\*\*\*

*"""*

*gevent 协成模块示例*

*"""*

**import** gevent

*# 导入脚本执行time模块操作*

**from** gevent **import** monkey

monkey.patch\_time()

**from** time **import** sleep

*# 协程函数*

**def** foo(a,b):

print(**"Running foo .."**,a,b)

sleep(3)

print(**"Foo again"**)

**def** bar():

print(**"Running bar .."**)

sleep(2)

print(**"Bar again"**)

*# 生成协程对象*

f = gevent.spawn(foo,1,2)

g = gevent.spawn(bar)

gevent.joinall([f,g]) *# 阻塞等待f,g执行完*

\*\*\*示例代码: day12/gevent\_server.py\*\*\*

*"""*

*gevent server 基于协程的tcp并发*

*"""*

**import** gevent

**from** gevent **import** monkey

monkey.patch\_all() *# 执行脚本修改socket阻塞行为*

**from** socket **import** \*

**def** handle(c):

**while True**:

data = c.recv(1024).decode()

**if not** data:

**return**

print(data)

c.send(**b'OK'**)

*# 创建套接字*

s = socket()

s.setsockopt(SOL\_SOCKET,SO\_REUSEADDR,1)

s.bind((**'127.0.0.1'**,8888))

s.listen(5)

*# 循环接收来自客户端请求*

**while True**:

c,addr = s.accept()

print(**"Connect from"**,addr)

*# handle(c) # 循环方案*

gevent.spawn(handle,c) *# 协程方案*

\* 安装：sudo pip3 install gevent

\* 函数

```python

gevent.spawn(func,argv)

功能: 生成协程对象

参数：func 协程函数

argv 给协程函数传参（不定参）

返回值： 协程对象

gevent.joinall(list,[timeout])

功能: 阻塞等待协程执行完毕

参数：list 协程对象列表

timeout 超时时间

gevent.sleep(sec)

功能: gevent睡眠阻塞

参数：睡眠时间

\* gevent协程只有在遇到gevent指定的阻塞行为时才会自动在协程之间进行跳转

如gevent.joinall(),gevent.sleep()带来的阻塞

```

\* monkey脚本

>作用：在gevent协程中，协程只有遇到gevent指定类型的阻塞才能跳转到其他协程，因此，我们希望将普通的IO阻塞行为转换为可以触发gevent协程跳转的阻塞，以提高执行效率。

> 转换方法：gevent 提供了一个脚本程序monkey,可以修改底层解释IO阻塞的行为，将很多普通阻塞转换为gevent阻塞。

> 使用方法

>>【1】 导入monkey

from gevent import monkey

>>【2】 运行相应的脚本，例如转换socket中所有阻塞

monkey.patch\_socket()

>>【3】 如果将所有可转换的IO阻塞全部转换则运行all

monkey.patch\_all()

>>【4】 注意：脚本运行函数需要在对应模块导入前执行

### HTTPServer v2.0

\*\*\*day12/http\_server.py\*\*\*

*"""*

*httpserver 2.0*

*env: python3.6*

*io多路复用* *http练习*

*"""*

**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.dir = dir

self.address = (host, port)

*# select 监控列表*

self.rlist = []

self.wlist = []

self.xlist = []

self.create\_socket()

self.bind()

*# 创建套接字*

**def** create\_socket(self):

self.sockfd = socket()

self.sockfd.setsockopt(SOL\_SOCKET,

SO\_REUSEADDR,

1)

*# 绑定地址*

**def** bind(self):

self.sockfd.bind(self.address)

**def** serve\_forever(self):

self.sockfd.listen(3)

print(**"Listen the port %d"** % self.port)

*# 设置关注的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:

c, addr = r.accept()

self.rlist.append(c)

**else**:

*# 有客户端发请求*

self.handle(r)

*# 处理客户端请求*

**def** handle(self, connfd):

*# 接收http请求*

request = connfd.recv(4096)

*# 客户端断开处理*

**if not** request:

self.rlist.remove(connfd)

connfd.close()

**return**

*# 提取请求内容*

request\_line = request.splitlines()[0]

info = request\_line.decode().split(**' '**)[1]

print(info)

*# 根据info情况分类*

**if** info == **'/' or** info[-5:] == **'.html'**:

self.get\_html(connfd, info)

**else**:

self.get\_data(connfd, info)

**def** get\_html(self, connfd, info):

**if** info == **'/'**:

*# 主页*

filename = self.dir + **"/index.html"**

**else**:

*# 其他网页*

filename = self.dir + info

**try**:

f = open(filename)

**except** Exception:

*# 没有网页*

response = **"HTTP/1.1 404 Not Found\r\n"**

response += **'Content-Type:text/html\r\n'**

response += **'\r\n'**

response += **"<h1>Sorry</h1>"**

**else**:

*# 有网页*

response = **"HTTP/1.1 200 OK\r\n"**

response += **'Content-Type:text/html\r\n'**

response += **'\r\n'**

response += f.read()

**finally**:

connfd.send(response.encode())

**def** get\_data(self, connfd, info):

f = open(self.dir + **'/timg.jpeg'**, **'rb'**)

response = **"HTTP/1.1 200 OK\r\n"**

response += **"Content-Type:image/jpeg\r\n"**

response += **'\r\n'**

response = response.encode() + f.read()

connfd.send(response)

**if** \_\_name\_\_ == **'\_\_main\_\_'**:

*# 用户自己提供的内容*

HOST = **'0.0.0.0'**

PORT = 8000

DIR = **"./static"** *# 网页目录*

http = HTTPServer(HOST, PORT, DIR) *# 实例化对象*

http.serve\_forever() *# 启动服务*

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】 做基本的请求解析，根据具体请求返回具体内容，同时满足客户端简单的非网页请求情况

【3】 通过类接口形式进行功能封装